Министерство науки и высшего образования РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

Разработка программных модулей
сборник учебно-методических материалов

для специальности 09.02.07 Информационные системы и программирование

Благовещенск

2020

Печатается по решению
редакционно-издательского совета

Составитель: Самохвалова С.Г.

Разработка программных модулей: сборник учебно-методических материалов для специальность 09.02.07 Информационные системы и программирование – Благовещенск: Амурский гос. ун-т, 2020

© Амурский государственный университет, 2020

© *Самохвалова С.Г., составитель*

КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА

Понятие жизненного цикла ПО

Разработка *ПО* — разновидность человеческой деятельности. Выделить ее компоненты можно, определив набор задач, которые нужно решить для достижения конечной цели — построения достаточно качественной системы в рамках заданных сроков и ресурсов. Для решения каждой такой задачи организуется вспомогательная *деятельность*, к которой можно также применить декомпозицию на отдельные, более мелкие деятельности, и т.д. В итоге должно стать понятно, как решать каждую отдельную подзадачу и всю задачу целиком на основе имеющихся решений для подзадач.

В качестве примеров деятельностей, которые нужно проводить для построения программной системы, можно привести **проектирование** — выделение отдельных модулей и *определение* связей между ними с целью минимизации зависимостей между частями проекта и достижения лучшей его обозримости в целом, **кодирование** — разработку кода отдельных модулей, разработку пользовательской документации, которая необходима для достаточно сложной системы.

Однако для корректного с точки зрения инженерии и экономики рассмотрения вопросов создания сложных систем необходимо, чтобы были затронуты и вопросы эксплуатации системы, внесения в нее изменений, а также самые первые действия в ходе ее создания — *анализ* потребностей пользователей и выработка решений, "изобретение" функций, удовлетворяющих эти потребности. Без этого невозможно, с одной стороны, учесть реальную эффективность системы в виде отношения полученных результатов ко всем сделанным затратам и, с другой стороны, правильно оценивать в ходе разработки степень соответствия системы реальным нуждам пользователей и заказчиков.

Все эти факторы приводят к необходимости рассмотрения всей совокупности деятельностей, связанных с созданием и использованием *ПО*, начиная с возникновения идеи о новом продукте и заканчивая удалением его последней копии.

Весь период существования *ПО*, связанный с подготовкой к его разработке, разработкой, использованием и модификациями, начиная с того момента, когда принимается решение разработать/приобрести/собрать из имеющихся компонентов новую систему или приходит сама идея о необходимости программы определенного рода, до того момента, когда полностью прекращается всякое ее использование, называют ***жизненным циклом ПО***.

В ходе *жизненного цикла ПО* проходит через *анализ предметной области*, сбор требований, проектирование, *кодирование*, тестирование, сопровождение и другие *виды деятельности*. Каждый *вид* представляет собой достаточно однородный набор действий, выполняемых для решения одной задачи или группы тесно связанных задач в рамках разработки и поддержки эксплуатации *ПО*.

При этом создаются и перерабатываются различного рода **артефакты** — создаваемые человеком информационные сущности, документы в достаточно общем смысле, участвующие в качестве входных данных и получающиеся в качестве результатов различных деятельностей.

Примерами артефактов являются: модель *предметной области*, описание требований, *техническое задание*, *архитектура* системы, проектная документация на систему в целом и на ее компоненты, прототипы системы и компонентов, собственно, исходный код, пользовательская документация, документация администратора системы, руководство *по* развертыванию, база пользовательских запросов, план проекта и пр.

На различных этапах в создание и эксплуатацию *ПО* вовлекаются люди, выполняющие различные *роли*.

Каждая *роль* может быть охарактеризована как абстрактная *группа* заинтересованных лиц, участвующих в деятельности *по* созданию и эксплуатации системы и решающих одни и те же задачи или имеющих одни и те же интересы *по* отношению к ней.

Примерами *ролей* являются: бизнес-аналитик, инженер *по* требованиям, *архитектор*, проектировщик пользовательского интерфейса, программист-кодировщик, технический писа-

тель, *тестировщик*, *руководитель проекта по* разработке, работник отдела продаж, *конечный пользователь*, *администратор* системы, инженер *по* поддержке и т.п.

Общую структуру *жизненного цикла* любого *ПО* задать невозможно, поскольку она существенно зависит от целей, для которых это *ПО* разрабатывается или приобретается, и от решаемых им задач. Структура *жизненного цикла* будет существенно разной у программы для форматирования кода, которая сначала делалась программистом для себя, а впоследствии была признана перспективной в качестве продукта и переработана, и у комплексной системы автоматизации предприятия, которая с самого начала задумывалась как таковая. Тем не менее, часто определяют основные элементы структуры *жизненного цикла* в виде *модели жизненного цикла ПО*.

Модель жизненного цикла ПО выделяет конкретные наборы *видов деятельности* (обычно разбиваемых на еще более мелкие активности), артефактов, ролей и их взаимосвязи, а также дает рекомендации *по* организации процесса в целом. Эти рекомендации включают ответы на вопросы о том, какие артефакты являются входными данными у каких *видов деятельности*, а какие появляются в качестве результатов, какие роли вовлечены в различные деятельности, как различные деятельности связаны друг с другом, каковы критерии качества полученных результатов, как оценить степень соответствия различных артефактов общим задачам проекта и когда можно переходить от одной деятельности к другой.

Жизненный цикл ПО является составной частью жизненного *цикла* программно-аппаратной системы, в которую это *ПО* входит. Поэтому часто различные его аспекты рассматриваются в связи с элементами жизненного *цикла* системы в целом.

Существует набор стандартов, определяющих различные элементы в структуре *жизненных циклов ПО* и программно-аппаратных систем. В качестве основных таких элементов выделяют **технологические процессы** — структурированные наборы деятельностей, решающих некоторую общую задачу или связанную совокупность задач, такие как процесс сопровождения *ПО*, *процесс обеспечения* качества, процесс разработки документации и пр. Процессы могут определять разные этапы *жизненного цикла* и увязывать их с различными *видами деятельностей*, артефактами и *ролями заинтересованных лиц*.

Стоит отметить, что процессом (или технологическим процессом) называют и набор процессов, увязанных для совместного решения более крупной задачи, например, всей совокупности деятельностей, входящих в *жизненный цикл ПО*. Таким образом, процессы могут разбиваться на подпроцессы, решающие частные подзадачи той задачи, с которой работает общий процесс.

Стандарты жизненного цикла

Чтобы получить *представление* о возможной структуре *жизненного цикла ПО*, обратимся сначала к соответствующим стандартам, описывающим технологические процессы. Международными организациями, такими как:

- IEEE — Institute of Electrical and *Electronic Engineers*, Институт инженеров по электротехнике и электронике;
- ISO — *International Standards* Organization, Международная организация по стандартизации;
- *EIA* — Electronic Industry Association, Ассоциация электронной промышленности;
- IEC — International *Electrotechnical Commission*, Международная комиссия по электротехнике;

а также некоторыми национальными и региональными институтами и организациями (в основном, американскими и европейскими, поскольку именно они оказывают наибольшее влияние на развитие технологий разработки *ПО* во всем мире):

- ANSI — American National Standards Institute, Американский национальный институт стандартов;
- *SEI* — Software Engineering Institute, Институт программной инженерии;
- *ECMA* — European Computer Manufactures Association, Европейская ассоциация производителей компьютерного оборудования;

разработан набор стандартов, регламентирующих различные аспекты *жизненного цикла* и вовлеченных в него процессов. *Список* и общее содержание этих стандартов представлены ниже.

Группа стандартов ISO

- **ISO/IEC 12207 Standard for Information Technology — Software Life Cycle Processes** (*процессы жизненного цикла ПО*, есть его российский аналог **ГОСТ Р-1999**).

Определяет общую структуру *жизненного цикла ПО* в виде 3 ступенчатой модели, состоящей из процессов, *видов деятельности* и задач. Стандарт описывает вводимые элементы в терминах их целей и результатов, тем самым задавая неявно возможные взаимосвязи между ними, но не определяя четко структуру этих связей, возможную организацию элементов в рамках проекта и метрики, по которым можно было бы отслеживать ход работ и их результативность.

Самыми крупными элементами являются *процессы жизненного цикла ПО (lifecycle processes)*. Всего выделено 18 процессов, которые объединены в 4 группы.

Таблица 2.1. Процессы жизненного цикла ПО по ISO 12207

Основные процессы	**Поддерживающие процессы**	**Организационные процессы**	**Адаптация**
Приобретение ПО; Передача ПО (в использование); Разработка ПО; Эксплуатация ПО;	Поддержка ПО Документирование; Управление конфигурациями; Обеспечение качества; Верификация; Валидация; Совместные экспертизы; Аудит; Разрешение проблем	Управление проектом; Управление инфраструктурой; Усовершенствование процессов; Управление персоналом	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Процессы строятся из отдельных *видов деятельности (activities)*.

Стандартом определены 74 *вида деятельности*, связанной с разработкой и поддержкой ПО. Ниже мы упомянем только некоторые из них.

- Приобретение ПО включает такие деятельности, как инициация приобретения, подготовка запроса предложений, подготовка контракта, анализ поставщиков, получение ПО и завершение приобретения.
- Разработка ПО включает развертывание *процесса разработки*, анализ системных требований, проектирование программно-аппаратной системы в целом, анализ требований к ПО, проектирование архитектуры ПО, детальное проектирование, кодирование и отладочное тестирование, интеграцию ПО, квалификационное тестирование ПО, системную интеграцию, квалификационное тестирование системы, развертывание (установку или инсталляцию) ПО, поддержку процесса получения ПО.
- Поддержка ПО включает развертывание процесса поддержки, анализ возникающих проблем и необходимых изменений, внесение изменений, экспертизу и передачу измененного ПО, перенос ПО с одной платформы на другую, изъятие ПО из эксплуатации.
- Управление проектом включает запуск проекта и определение его рамок, планирование, выполнение проекта и надзор за его выполнением, экспертизу и оценку проекта, свертывание проекта.

Каждый *вид деятельности* нацелен на решение одной или нескольких **задач (tasks)**. Всего определено 224 различные задачи. Например:

- Развертывание *процесса разработки* состоит из определения *модели жизненного цикла*, документирования и контроля результатов отдельных работ, выбора используемых стандартов, языков, инструментов и пр.
- Перенос ПО между платформами состоит из разработки плана переноса, оповещения пользователей, выполнения анализа произведенных действий и пр.

- **ISO/IEC 15288 Standard for Systems Engineering — System Life Cycle Processes** (*процессы жизненного цикла* систем).

Отличается от предыдущего нацеленностью на рассмотрение программно-аппаратных систем в целом.

В данный момент продолжается работа по приведению этого стандарта в соответствие с предыдущим.

ISO/IEC 15288 предлагает похожую схему рассмотрения жизненного цикла системы в виде набора процессов. Каждый процесс описывается набором его **результатов (outcomes)**, которые достигаются при помощи различных *видов деятельности*.

Всего выделено 26 процессов, объединяемых в 5 групп.

Таблица 2.2. Процессы жизненного цикла систем по ISO 15288

Процессы выработки соглашений	Процессы уровня организации	Процессы уровня проекта	Технические процессы	Специальные процессы
Приобретение системы; Поставка системы	Управление окружением; Управление инвестициями; Управление процессами; Управление ресурсами; Управление качеством	Планирование; Оценивание; Мониторинг; Управление рисками; Управление конфигурацией; Управление информацией; Выработка решений	Определение требований; Анализ требований; Проектирование архитектуры; Реализация; Интеграция; Верификация; Валидация; Передача в использование; Эксплуатация; Поддержка; Изъятие из эксплуатации	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Помимо процессов, определено 123 различных результата и 208 *видов деятельности*, нацеленных на их достижение. Например, определение требований имеет следующие результаты:

Должны быть поставлены технические задачи, которые предстоит решить.

Должны быть сформулированы системные требования.

Деятельности в рамках этого процесса следующие.

Определение границ функциональности системы.

Определение функций, которые необходимо поддерживать.

Определение критериев оценки качества при использовании системы.

Анализ и выделение требований по безопасности.

Анализ требований защищенности.

Выделение критических для данной системы аспектов качества и требований к ним.

Анализ целостности системных требований.

Демонстрация прослеживаемости требований.

Фиксация и поддержка набора системных требований.

- **ISO/IEC 15504 (SPICE) Standard for Information Technology — Software Process Assessment** (оценка *процессов разработки* и поддержки ПО).

Определяет правила оценки *процессов жизненного цикла ПО* и их возможностей, опирается на модель CMMI (см. ниже) и больше ориентирован на оценку процессов и возможностей их улучшения.

В качестве основы для *оценки процессов* определяет некоторую базовую модель, аналогичную двум описанным выше. В ней выделены категории процессов, процессы и *виды деятельности*.

Определяются 5 категорий, включающих 35 процессов и 201 *вид деятельности*:

Таблица 2.3. Процессы жизненного цикла ПО и систем по ISO 15504

Отношения "заказчик-поставщик"	Процессы уровня организации	Процессы уровня проекта	Инженерные процессы	Процессы поддержки
Приобретение ПО; Составление контракта;	Развитие бизнеса; Определение процес-	Планирование жизненного цикла;	Выделение системных требований и проектирование	Разработка документации;

Определение нужд заказчика; Проведение совместных экспертиз и аудитов; Подготовка к передаче; Поставка и развертывание; Поддержка эксплуатации; Предоставление услуг; Оценка удовлетворенности заказчиков	сов; Усовершенствование процессов; Обучение; Обеспечение переиспользования; Обеспечение инструментами; Обеспечение среды для работы	Планирование проекта; Построение команды; Управление требованиями; Управление качеством; Управление рисками; Управление ресурсами и графиком работ; Управление подрядчиками	системы в целом; Выделение требований ПО; Проектирование ПО; Реализация, интеграция и тестирование ПО; Интеграция и тестирование системы; Сопровождение системы и ПО	Управление конфигурацией; Обеспечение качества; Разрешение проблем; Проведение экспертиз

Например, приобретение ПО включает такие *виды деятельности*, как определение потребности в ПО, определение требований, подготовку стратегии покупки, подготовку запроса предложений, выбор поставщика.

Группа стандартов IEEE

- **IEEE 1074-1997 — IEEE Standard for Developing Software Life Cycle Processes** (стандарт на создание *процессов жизненного цикла* ПО).

Нацелен на описание того, как создать специализированный *процесс разработки* в рамках конкретного проекта. Описывает ограничения, которым должен удовлетворять любой такой процесс, и, в частности, общую структуру *процесса разработки*. В рамках этой структуры определяет основные *виды деятельностей*, выполняемых в этих процессах, и документы, требующиеся на входе и возникающие на выходе этих деятельностей. Всего рассматриваются 5 подпроцессов, 17 групп деятельностей и 65 *видов деятельности*.

Например, подпроцесс разработки состоит из групп деятельностей по выделению требований, по проектированию и по реализации. Группа деятельностей по проектированию включает архитектурное проектирование, проектирование баз данных, проектирование интерфейсов, детальное проектирование компонентов.

- **IEEE/EIA 12207-1997 — IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes** (промышленное использование стандарта ISO/IEC 12207 на *процессы жизненного цикла* ПО).

Аналог ISO/IEC 12207, сменил ранее использовавшиеся стандарты J-Std-016-1995 *EIA*/IEEE Interim Standard for Information Technology — *Software Life Cycle* Processes — Software Development *Acquirer-Supplier* Agreement (промежуточный стандарт на *процессы жизненного цикла* ПО и соглашения между поставщиком и заказчиком ПО) и стандарт министерства обороны США *MIL-STD*-498.

Группа стандартов CMM, разработанных SEI

- *Модель зрелости возможностей* **CMM (Capability Maturity Model)** предлагает унифицированный подход к оценке возможностей организации выполнять задачи различного уровня. Для этого определяются 3 уровня элементов: **уровни зрелости организации (maturity levels), ключевые области процесса (key process areas) и ключевые практики (key practices)**. Чаще всего под моделью CMM имеют в виду модель уровней зрелости. В настоящий момент CMM считается устаревающей и сменяется моделью CMMI .

Уровни зрелости.

CMM описывает различные степени зрелости процессов в организациях, определяя 5 уровней организаций.

- Уровень 1, начальный (initial).

Организации, разрабатывающие ПО, но не имеющие осознанного *процесса разработки*, не производящие планирования и оценок своих возможностей.

- Уровень 2, повторяемый (repeatable).

В таких организациях ведется учет затрат ресурсов и отслеживается ход проектов, установлены правила управления проектами, основанные на имеющемся опыте.

▪ Уровень 3, определенный (defined).

В таких организациях имеется принятый, полностью документированный, соответствующий реальному положению дел и доступный персоналу *процесс разработки* и сопровождения ПО. Он должен включать как управленческие, так и технические подпроцессы, а также обучение сотрудников работе с ним.

▪ Уровень 4, управляемый (manageable).

В этих организациях, помимо установленного и описанного процесса, используются измеримые показатели качества продуктов и результативности процессов, которые позволяют достаточно точно предсказывать объем ресурсов (времени, денег, персонала), необходимый для разработки продукта с определенным качеством.

▪ Уровень 5, совершенствующийся (optimizing).

В таких организациях, помимо процессов и методов их оценки, имеются методы определения слабых мест, определены процедуры поиска и оценки новых методов и техник разработки, обучения персонала работе с ними и их включения в общий процесс организации в случае повышения эффективности производства.

Ключевые области процесса.

Согласно CMM, уровни зрелости организации можно определять по использованию четко определенных техник и процедур, относящихся к различным ключевым областям процесса. Каждая такая область представляет собой набор связанных *видов деятельности*, которые нацелены на достижение целей, существенных для общей оценки результативности технологического процесса. Всего выделяется 18 областей. Множество областей, которые должны поддерживаться организацией, расширяется при переходе к более высоким уровням зрелости.

- К первому уровню не предъявляется никаких требований.
- Организации второго уровня зрелости должны поддерживать управление требованиями, планирование проектов, надзор за ходом проекта, управление подрядчиками, обеспечение качества ПО, управление конфигурацией.
- Организации третьего уровня должны, помимо деятельностей второго уровня, поддерживать проведение экспертиз, координацию деятельности отдельных групп, разработку программного продукта, интегрированное управление разработкой и сопровождением, обучение персонала, выработку и поддержку технологического процесса организации, контроль соблюдения технологического процесса организации.
- К деятельностям организаций четвертого уровня добавляются: управление качеством ПО и управление процессом, основанное на измеримых показателях.
- Организации пятого уровня зрелости должны дополнительно поддерживать управление изменениями процесса, управление изменениями используемых технологий и предотвращение дефектов.

Ключевые практики.

Ключевые области процесса описываются с помощью наборов ключевых практик. Ключевые практики классифицированы на несколько видов: обязательства (commitments to perform), возможности (abilities to perform), деятельности (activities performed), измерения (measurements and analysis) и проверки (verifying implementations).

Например, управление требованиями связано со следующими практиками:

▪ Обязательство.

Проекты должны следовать определенной политике организации по управлению требованиями.

▪ Возможности.

В каждом проекте должен определяться ответственный за анализ системных требований и привязку их к аппаратному, программному обеспечению и другим компонентам системы.

Требования должны быть документированы.

Для управления требованиями должны быть выделены адекватные ресурсы и бюджет.

Персонал должен проходить обучение в области управления требованиями.

- Деятельности.

Прежде чем быть включенными в проект, требования подвергаются анализу на полноту, адекватность, непротиворечивость и пр.

Выделенные требования используются в качестве основы для планирования и выполнения других работ.

Изменения в требованиях анализируются и включаются в проект.

- Измерение.

Производится периодическое определение статуса требований и статуса деятельности по управлению ими.

- Проверки.

Деятельность по управлению требованиями периодически анализируется старшими менеджерами.

Деятельность по управлению требованиями периодически и на основании значимых событий анализируется менеджером проекта.

Группа обеспечения качества проводит анализ и аудит деятельности по управлению требованиями и отчитывается по результатам этого анализа.

Таблица 2.4 суммирует информацию о количестве практик различных видов, приписанных к разным ключевым областям процесса.

Таблица 2.4. Количество ключевых практик в разных областях процесса по CMM версии 1.1

Уровни	Область процесса	Обязательства	Возможности	Деятельности	Измерения	Проверки
2	Управление требованиями	1	4	3	1	3
	Планирование проектов	2	4	15	1	3
	Надзор за ходом проекта	2	5	13	1	3
	Управление подрядчиками	2	3	13	1	3
	Обеспечение качества ПО	1	4	8	1	3
	Управление конфигурацией	1	5	10	1	4
3	Контроль соблюдения технологического процесса	3	4	7	1	1
	Выработка и поддержка технологического процесса	1	2	6	1	1
	Обучение персонала	1	4	6	2	3
	Интегрированное управление	1	3	11	1	3
	Разработка программного продукта	1	4	10	2	3
	Координация деятельности групп	1	5	7	1	3
	Проведение экспертиз	1	3	3	1	1
4	Управление процессом на основе метрик	2	5	7	1	3
	Управление качеством ПО	1	3	5	1	3
5	Предотвращение дефектов	2	4	8	1	3
	Управление изменениями технологий	3	5	8	1	2
	Управление изменениями процесса	2	4	10	1	2

Интегрированная *модель зрелости возможностей* **CMMI (Capability Maturity Model Integration)** .

Эта модель представляет собой результат интеграции моделей CMM для продуктов и процессов, а также для разработки ПО и разработки программно-аппаратных систем.

Основные изменения по сравнению с CMM следующие.

- Созданы два несколько отличающихся изложения модели — непрерывное и поэтапное. Первое предназначено для облегчения миграции от поддержки американского отраслевого стандарта *EIA*/*AIS* 713 и постепенного усовершенствования процессов за счет внедрения различных практик. Второе предназначено для облегчения миграции от поддержки CMM и поуровневого рассмотрения вводимых практик.

o Элементы модели получили четкие пометки о том, являются ли они обязательными (required), рекомендуемыми (expected) или информативными (informative).

o Используемые практики разделяются на общие (generic) и специфические (specific). Они дополняются набором общих и специфических целей, которые необходимы для достижения определенного уровня зрелости в определенных областях процесса.

o Некоторые уровни зрелости получили другие названия. Второй уровень назван управляемым (managed), а четвертый — управляемым на основе метрик (quantitatively managed).

o Набор выделяемых областей процесса и практик значительно изменился.

Все области процесса делятся на 4 категории. В приводимом ниже списке области процесса помечены номером уровня, начиная с которого они должны поддерживаться согласно CMMI.

▪ Управление процессом.

Включает выработку и поддержку процесса (3), контроль соблюдения процесса (3), обучение (3), измерение показателей процесса (4), внедрение инноваций (5).

Управление проектом.

Включает планирование проектов (2), контроль хода проекта (2), управление соглашениями с поставщиками (2), интегрированное управление проектами (3), управление рисками (3), построение команд (3), управление поставщиками (3) и измерение показателей результативности и хода проекта (4).

▪ Технические.

Включают выработку требований (3), управление требованиями (2), выработку технических решений (3), интеграцию продуктов (3), верификацию (3) и валидацию (3).

▪ Поддерживающие.

Включают управление конфигурацией (2), обеспечение качества продуктов и процессов (2), проведение измерений и анализ их результатов (2), управление окружением (3), анализ и принятие решений (3), анализ, разрешение и предотвращение проблем (5).

В целом перечисленные стандарты связаны так, как показано на рис. 2.1 (сплошные стрелки указывают направления исторического развития, жирная стрелка обозначает идентичность, пунктирные стрелки показывают влияние одних стандартов на другие).

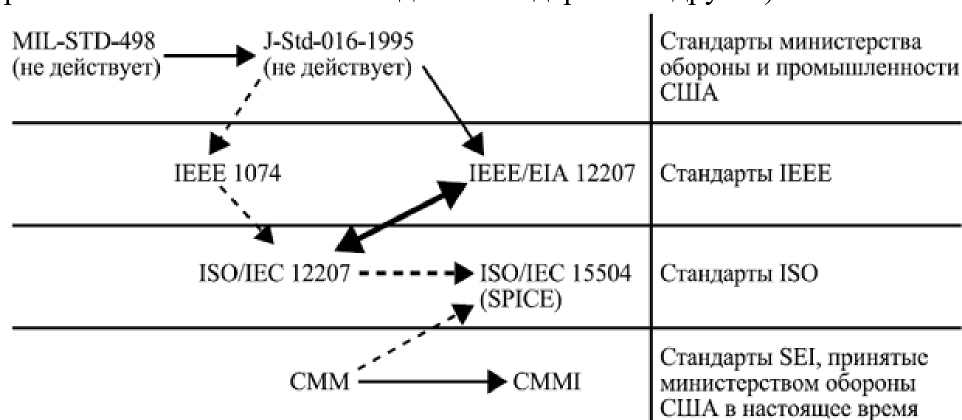


Рис. 2.1. Стандарты, описывающие структуру жизненного цикла ПО

Стандарты являются суммой опыта, который был накоплен экспертами в инженерии ПО на основе огромного количества проектов, проводившихся в рамках коммерческих структур США и Европы и в рамках военных контрактов. Большая часть стандартов создавалась как набор критериев отбора поставщиков программного обеспечения для министерства обороны США, и эту задачу они решают достаточно успешно.

Все рассмотренные стандарты определяют некоторый набор *видов деятельности*, из которых должен состоять *процесс разработки*, и задают ту или иную структуру на этих *видах деятельности*, выделяя их элементы. Вместе с тем, как можно заметить, они не могут быть сведены

без существенных изменений в единую *модель жизненного цикла ПО*. В целом имеющиеся стандарты слабо согласованы между собой.

Кроме того, данные стандарты не предписывают четких и однозначных схем построения *жизненного цикла ПО*, в частности, связей между отдельными деятельностями. Это сделано намеренно, поскольку ранее действовавшие стандарты типа *DoD*-Std-2167 были достаточно жестко привязаны к *каскадной модели жизненного цикла* и тем самым препятствовали использованию более прогрессивных технологий разработки.

Современные стандарты стараются максимально общим образом определить набор *видов деятельности*, которые должны быть представлены в рамках жизненного цикла (с учетом целей отдельных проектов — т.е. проект, не стремящийся достичь каких-то целей, может не включать деятельностей, связанных с их достижением), и описать их при помощи наборов входных документов и результатов.

Стоит заметить, что стандарты могут достаточно сильно разойтись с реальной разработкой, если в ней используются новейшие методы автоматизации разработки и сопровождения ПО. Стандарты организаций ISO и IEEE построены на основе имеющегося эмпирического опыта разработки, полученного в рамках распространенных некоторое время назад парадигм и инструментальных средств. Это не значит, что они устарели, поскольку их авторы имеют достаточно хорошее представление о новых методах и технологиях разработки и пытались смотреть вперед. Но при использовании новаторских технологий в создании ПО часть требований стандарта может обеспечиваться совершенно иными способами, чем это предусмотрено в нем, а часть артефактов может отсутствовать в рамках данной технологии, исчезнув внутри автоматизированных процессов.

Модели жизненного цикла

Все обсуждаемые стандарты так или иначе пытаются описать, как должен выглядеть любой *процесс разработки ПО*. При этом они вынуждены вводить слишком общие *модели жизненного цикла ПО*, которые тяжело использовать при организации конкретного проекта.

В рамках специфических *моделей жизненного цикла*, которые предписывают правила организации разработки *ПО* в рамках данной отрасли или организации, определяются более конкретные *процессы разработки*. Отличаются они от стандартов, прежде всего, большей детальностью и четким описанием связей между отдельными *видами деятельности*, определением потоков данных (документов и артефактов) в ходе *жизненного цикла*. Таких моделей довольно много, ведь фактически каждый раз, когда некоторая организация определяет собственный *процесс разработки*, в качестве основы этого процесса разрабатывается некоторая *модель жизненного цикла ПО*. В рамках данной лекции мы рассмотрим лишь несколько моделей. К сожалению, очень тяжело выбрать критерии, *по* которым можно было бы дать хоть сколько-нибудь полезную классификацию известных *моделей жизненного цикла*.

Наиболее широко известной и применяемой долгое время оставалась так называемая *каскадная* или **водопадная (waterfall)** *модель жизненного цикла*, которая, как считается, была впервые четко сформулирована в работе и впоследствии запечатлена в стандартах министерства обороны США в 70-80-х годах XX века. Эта модель предполагает последовательное выполнение различных *видов деятельности*, начиная с выработки требований и заканчивая сопровождением, с четким определением границ между этапами, на которых набор документов, созданный на предыдущей стадии, передается в качестве входных данных для следующей. Таким образом, каждый *вид деятельности* выполняется на какой-то одной фазе *жизненного цикла*. Предлагаемая в статье последовательность шагов разработки показана на рис. 2.2. "Классическая" *каскадная модель* предполагает только движение вперед *по* этой схеме: все необходимое для проведения очередной деятельности должно быть подготовлено в ходе предшествующих *работ*.

Рис. 2.2. Последовательность разработки согласно "классической" каскадной модели

Однако, если внимательно прочитать статью, оказывается, что она не предписывает следование именно этому порядку *работ*, а, скорее, представляет модель итеративного процесса (см. рис.2.3) — в ее последовательном виде эта модель закрепилась, *по*-видимому, в представлении чиновников из министерств и управленцев компаний, работающих с этими министерствами *по* контрактам. При реальной работе в соответствии с моделью, допускающей движение только в одну сторону, обычно возникают проблемы при обнаружении недоработок и ошибок, сделанных на ранних этапах. Но еще более тяжело иметь дело с изменениями окружения, в котором разрабатывается *ПО* (это могут быть изменения требований, смена подрядчиков, изменения политик разрабатывающей или эксплуатирующей организации, изменения отраслевых стандартов, появление конкурирующих продуктов и пр.).

Работать в соответствии с этой моделью можно, только если удается предвидеть заранее возможные перипетии хода проекта и тщательно собирать и интегрировать информацию на первых этапах, с тем чтобы впоследствии можно было пользоваться их результатами без оглядки на возможные изменения.

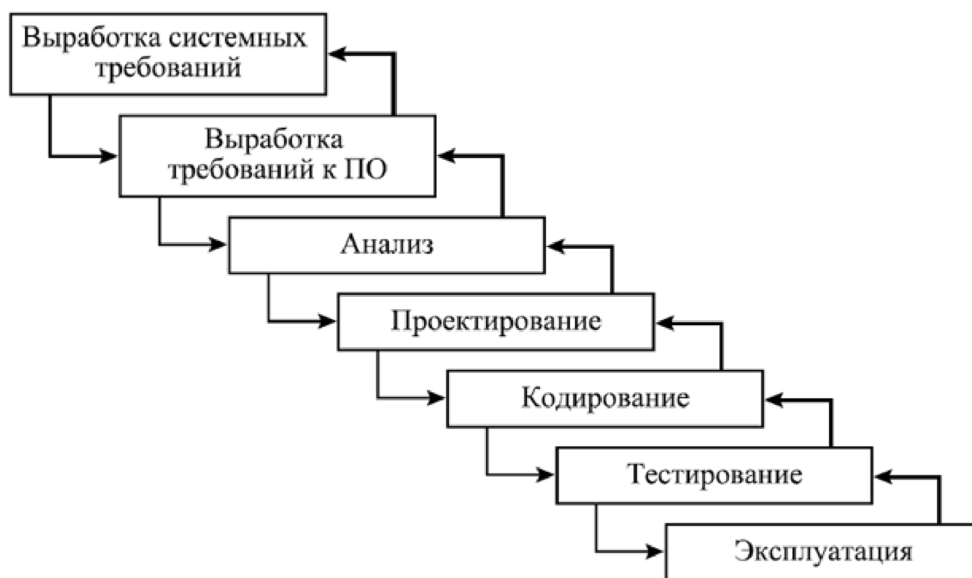


Рис. 2.3. Ход разработки, предлагаемый в статье

Среди разработчиков и исследователей, имевших дело с разработкой сложного *ПО*, практически с самого зарождения индустрии производства программ большую популярность имели модели эво-

люционных или итеративных процессов, поскольку они обладают большей гибкостью и способностью работать в меняющемся окружении.

Итеративные или **инкрементальные модели** (известно несколько таких моделей) предполагают *разбиение* создаваемой системы на набор кусков, которые разрабатываются с помощью нескольких последовательных проходов всех *работ* или их части.

На первой итерации разрабатывается кусок системы, не зависящий от других. При этом большая часть или даже полный цикл *работ* проходится на нем, затем оцениваются результаты и на следующей итерации либо первый кусок переделывается, либо разрабатывается следующий, который может зависеть от первого, либо как-то совмещается доработка первого куска с добавлением новых функций. В результате на каждой итерации можно анализировать промежуточные результаты *работ* и реакцию на них всех заинтересованных лиц, включая пользователей, и вносить корректирующие изменения на следующих итерациях. Каждая *итерация* может содержать полный набор *видов деятельности* — от анализа требований до ввода в эксплуатацию очередной части *ПО*.

Рис. 2.4. Возможный ход работ по итеративной модели

Каскадная модель с возможностью возвращения на предшествующий шаг при необходимости пересмотреть его результаты, становится *итеративной*.

Итеративный процесс предполагает, что разные *виды деятельности* не привязаны намертво к определенным этапам разработки, а выполняются *по* мере необходимости, иногда повторяются, до тех пор, пока не будет получен нужный результат.

Вместе с гибкостью и возможностью быстро реагировать на изменения, *итеративные модели* привносят дополнительные сложности в *управление проектом* и отслеживание его хода. При использовании итеративного подхода значительно сложнее становится адекватно оценить текущее состояние проекта и спланировать долгосрочное развитие событий, а также предсказать сроки и ресурсы, необходимые для обеспечения определенного качества результата.

Развитием идеи итераций является ***спиральная*** модель жизненного *цикла ПО*, предложенная Боэмом (Boehm). Она предлагает каждую итерацию начинать с выделения целей и планирования очередной итерации, определения основных альтернатив и ограничений при ее выполнении, их оценки, а также оценки возникающих рисков и определения способов избавления от них, а заканчивать итерацию оценкой результатов проведенных в ее рамках *работ*.

Основным ее новым элементом является общая структура действий на каждой итерации — планирование, *определение* задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных *работ* итерации и оценка их результатов.

Название " *спиральная* " эта модель получила из-за изображения хода *работ* в "полярных координатах", в которых угол соответствует выполняемому этапу в рамках общей структуры итераций, а удаление от начала координат — затраченным ресурсам.

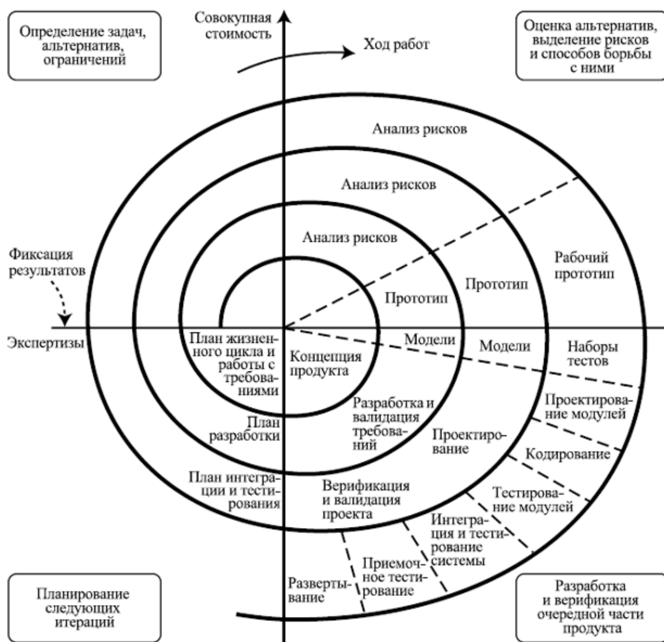


Рис. 2.5. Изображение хода работ по спиральной модели согласно Боему

Рис. 2.5 показывает возможное развитие проекта *по спиральной модели*. Количество витков, а также расположение и набор *видов деятельности* в правом нижнем квадранте могут изменяться в зависимости от результатов планирования и анализа рисков, проводимых на предыдущих этапах.

На следующей лекции мы рассмотрим в деталях два современных итеративных *процесса разработки* —унифицированный *процесс разработки* Rational и экстремалььное *программирование*.

"Тяжелые" и "легкие" процессы разработки

В этой лекции мы рассмотрим детально два процесса разработки — *унифицированный процесс Rational (Rational Unified Process, RUP)* и *экстремальное программирование (Extreme Programming, XP)*. Оба они являются примерами итеративных процессов, но построены на основе различных предположений о природе разработки программного обеспечения и, соответственно, достаточно сильно отличаются.

RUP является примером так называемого *"тяжелого" процесса*, детально описанного и предполагающего поддержку собственно разработки исходного кода *ПО* большим количеством вспомогательных действий. Примерами подобных действий являются разработка планов, технических заданий, многочисленных проектных моделей, проектной документации и пр. Основная цель такого процесса — отделить успешные практики разработки и сопровождения *ПО* от конкретных людей, умеющих их применять. Многочисленные вспомогательные действия дают надежду сделать возможным успешное решение задач *по* конструированию и поддержке сложных систем с помощью имеющихся работников, не обязательно являющихся суперпрофессионалами.

Для достижения этого выполняется иерархическое пошаговое детальное описание предпринимаемых в той или иной ситуации действий, чтобы можно было научить обычного работника действовать аналогичным образом. В ходе проекта создается много промежуточных документов, позволяющих разработчикам последовательно разбивать стоящие перед ними задачи на более простые. Эти же документы служат для проверки правильности решений, принимаемых на каждом шаге, а также отслеживания общего хода *работ* и уточнения оценок ресурсов, необходимых для получения желаемых результатов.

Экстремальное программирование, наоборот, представляет так называемые ***"живые" (agile) методы разработки***, называемые также **"легкими"** процессами. Они делают упор на использовании хороших разработчиков, а не хорошо отлаженных процессов разработки. *Живые методы* избегают фиксации четких схем действий, чтобы обеспечить большую гибкость в каждом

конкретном проекте, а также выступают против разработки дополнительных документов, не вносящих непосредственного вклада в получение готовой работающей программы.

Унифицированный процесс Rational

RUP является довольно сложной, детально проработанной *итеративной моделью жизненного цикла ПО*.

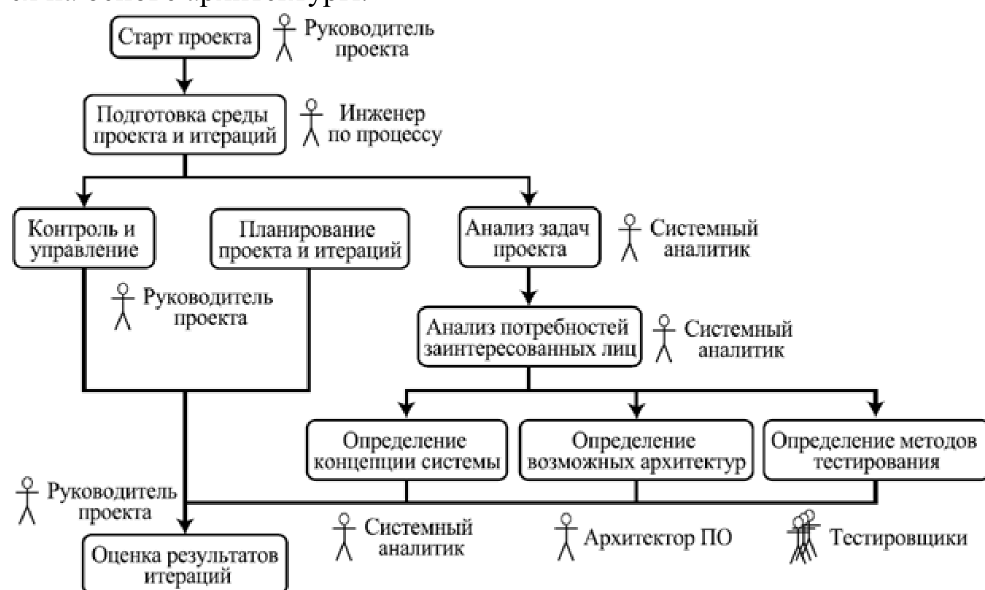
Исторически *RUP* является развитием модели процесса разработки, принятой в компании Ericsson в 70–80-х годах XX века. Эта модель была создана Джекобсоном (Ivar Jacobson), впоследствии, в 1987 году, основавшим собственную компанию Objectory AB именно для развития технологического процесса разработки *ПО* как отдельного продукта, который можно было бы переносить в другие организации. После вливания Objectory в Rational в 1995 году разработки Джекобсона были интегрированы с работами Ройса (Walker Royce, сын автора "классической" *каскадной модели*), Крухтена (Philippe Kruchten) и Буча (Grady Booch), а также с развивавшимся параллельно **универсальным языком моделирования (Unified Modeling Language, UML)**.

RUP основан на трех ключевых идеях:

- Весь ход работ направляется итоговыми целями проекта, выраженными в виде **вариантов использования (use cases)** — сценариев взаимодействия результирующей программной системы с пользователями или другими системами, при выполнении которых пользователи получают значимые для них результаты и услуги. Разработка начинается с выделения вариантов использования и на каждом шаге контролируется степенью приближения к их реализации.
- Основным решением, принимаемым в ходе проекта, является **архитектура** результирующей программной системы. Архитектура устанавливает набор компонентов, из которых будет построено ПО, ответственность каждого из компонентов (т.е. решаемые им подзадачи в рамках общих задач системы), четко определяет интерфейсы, через которые они могут взаимодействовать, а также способы взаимодействия компонентов друг с другом.

Архитектура является одновременно основой для получения качественного ПО и базой для планирования работ и оценок проекта в терминах времени и ресурсов, необходимых для достижения определенных результатов. Она оформляется в виде набора графических моделей на языке UML.

- Основой процесса разработки являются **планируемые** и **управляемые итерации**, объем которых (реализуемая в рамках итерации функциональность и набор компонентов) определяется на основе архитектуры.



UP выделяет в жизненном цикле 4 основные фазы, в рамках каждой из которых возможно проведение нескольких итераций. Кроме того, разработка системы может пройти через несколько циклов, включающих все 4 фазы.

1.**Фаза начала проекта (Inception)**

Основная цель этой фазы — достичь компромисса между всеми заинтересованными лицами относительно задач проекта и выделяемых на него ресурсов.

На этой стадии определяются основные цели проекта, руководитель и бюджет, основные средства выполнения — технологии, инструменты, ключевые исполнители. Также, возможно, происходит апробация выбранных технологий, чтобы убедиться в возможности достичь целей с их помощью, и составляются предварительные планы проекта.

На эту фазу может уходить около 10% времени и 5% трудоемкости одного цикла.

Пример хода работ показан на рис. 3.1.

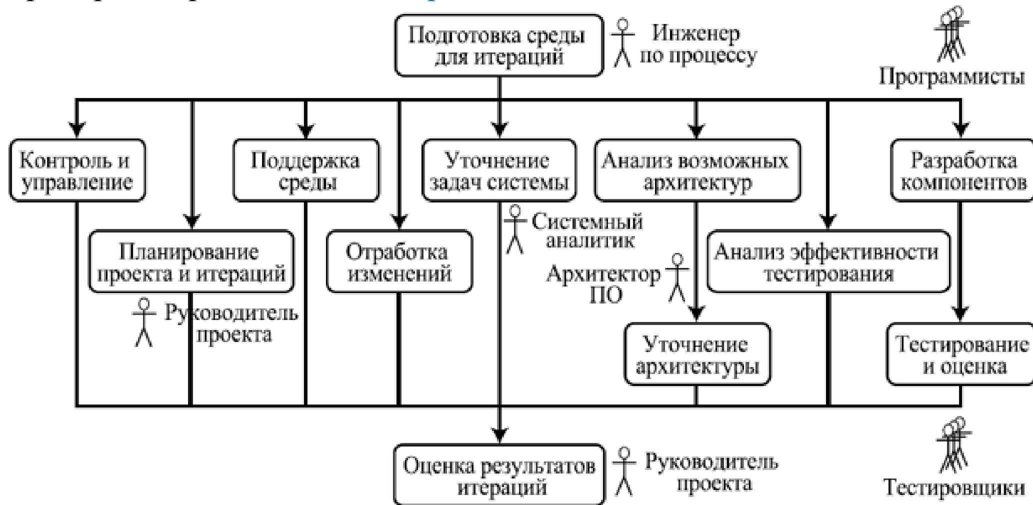


Рис. 3.2. Пример хода работ на фазе проектирования

2.**Фаза проектирования (Elaboration)**

Основная цель этой фазы — на базе основных, наиболее существенных требований разработать стабильную базовую архитектуру продукта, которая позволяет решать поставленные перед системой задачи и в дальнейшем используется как основа разработки системы.

На эту фазу может уходить около 30% времени и 20% трудоемкости одного цикла.

Пример хода работ представлен на рис. 3.2.

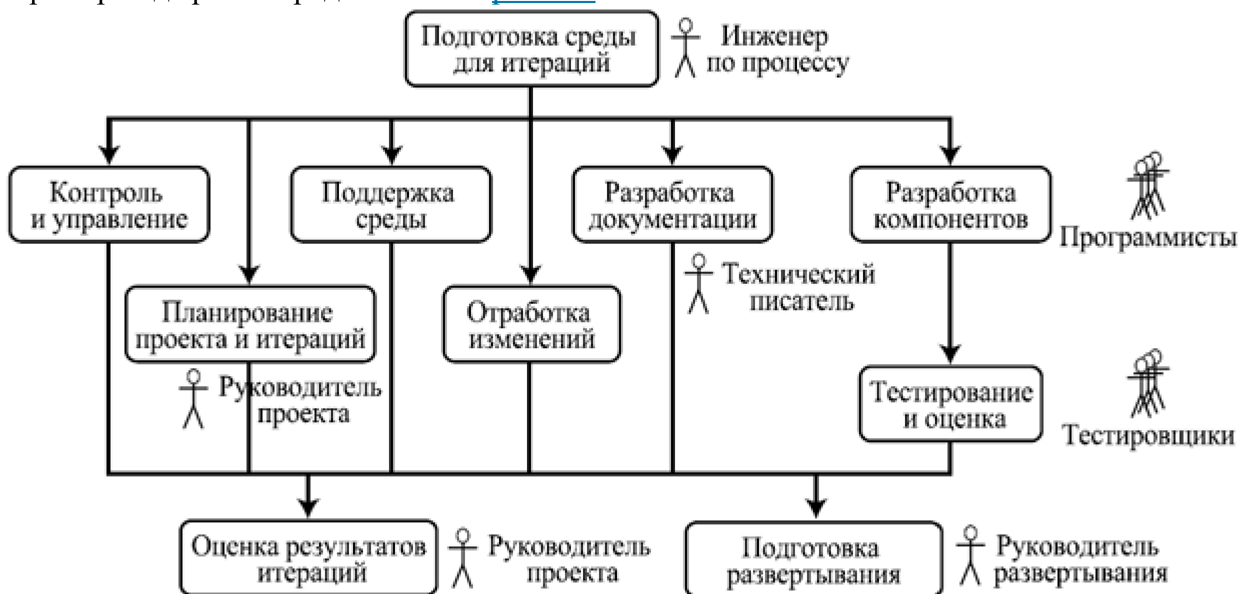


Рис. 3.3. Пример хода работ на фазе построения

3.**Фаза построения (Construction)**

Основная цель этой фазы — детальное прояснение требований и разработка системы, удовлетворяющей им, на основе спроектированной ранее архитектуры. В результате должна получиться система, реализующая все выделенные варианты использования.

На эту фазу уходит около 50% времени и 65% трудоемкости одного цикла.

Пример хода работ на этой фазе представлен на рис.3.3.

4.**Фаза внедрения (Transition)**

Цель этой фазы — сделать систему полностью доступной конечным пользователям. На этой стадии происходит развертывание системы в ее рабочей среде, бета-тестирование, подгонка мелких деталей под нужды пользователей.

На эту фазу может уходить около 10% времени и 10% трудоемкости одного цикла.

Пример хода работ представлен на рис. 3.4.

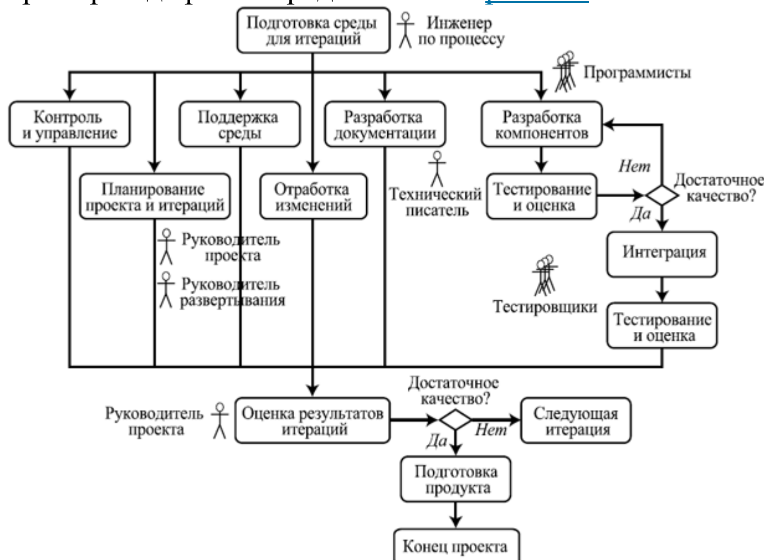


Рис. 3.4. Пример хода работ на фазе внедрения

Артефакты, вырабатываемые в ходе проекта, могут быть представлены в виде баз данных и таблиц с информацией различного типа, разных видов документов, исходного кода и объектных модулей, а также моделей, состоящих из отдельных элементов. Основные артефакты и *потоки данных* между ними согласно *RUP* изображены на рис. 3.5.

Наиболее важные с точки зрения *RUP* артефакты проекта — это модели, описывающие различные аспекты будущей системы. Большинство моделей представляют собой наборы диаграмм *UML*. Основные используемые виды моделей следующие:

Модель вариантов использования (Use-Case Model).

Эта модель определяет требования к ПО — то, что система должна делать — в виде набора вариантов использования. Каждый вариант использования задает сценарий взаимодействия системы с **действующими лицами (actors)** или ролями, дающий в итоге значимый для них результат. Действующими лицами могут быть не только люди, но и другие системы, взаимодействующие с рассматриваемой. Вариант использования определяет основной ход событий, развивающийся в нормальной ситуации, а также может включать несколько альтернативных сценариев, которые начинают работать только при специфических условиях.

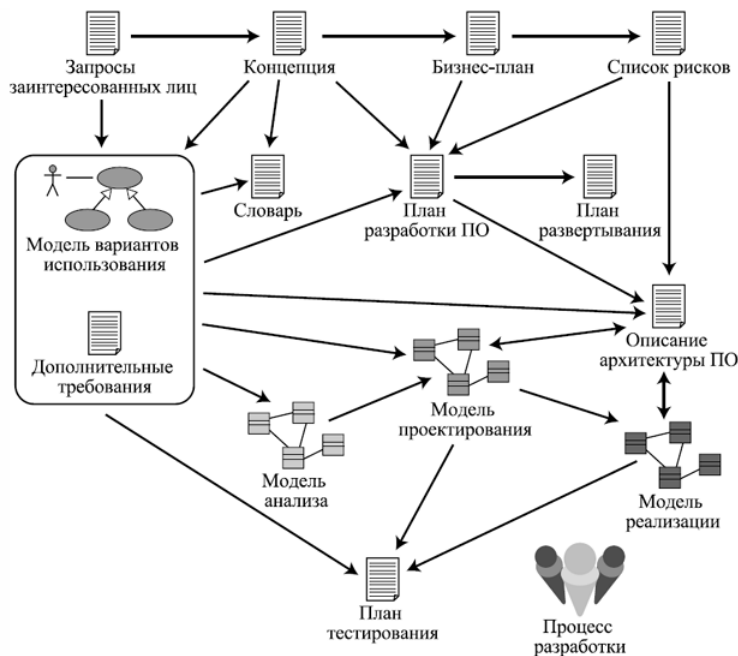


Рис. 3.5. Основные артефакты проекта по RUP и потоки данных между ними

Модель вариантов использования служит основой для проектирования и оценки готовности системы к внедрению.

Примером варианта использования может служить сценарий действий клиента Интернет-магазина по отношению к сайту этого магазина, в результате которых клиент заказывает товар, например, книги. Такой вариант использования можно назвать "Заказ товара". Если нас интересует сайт магазина только как программная система, результатом можно считать то, что запись о сделанном заказе занесена в базу данных, а оператору заказов отправлено электронное письмо, содержащее всю информацию, которая необходима для того, чтобы можно было сформировать заказ. В нее входит контактная информация покупателя, идентификатор заказа и, например, список заказанных книг с их ISBN, их количество для каждого наименования и номера партий для удобства их поиска на складе. При этом выполнение остальной части варианта использования — это дело других составляющих системы под названием "Интернет-магазин".

Эта работа может включать звонок или письмо клиенту и подтверждение, что именно он сделал заказ, вопрос об удобных для него форме, времени и адресе доставки и форме оплаты, формирование заказа, передача его для доставки курьеру, доставка и подтверждение получения заказа и оплаты. В нашем примере действующими лицами являются клиент, делающий заказ, и оператор заказов.

Альтернативные сценарии в рамках данного варианта могут выполняться, если, например, заказанного пользователем товара нет на складе или сам пользователь находится на плохом счету в магазине из-за неоплаченных прежних заказов, или, наоборот, он является привилегированным клиентом или представителем крупной организации.

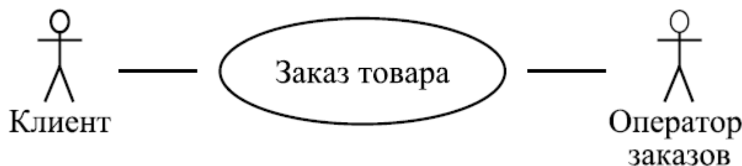


Рис. 3.6. Пример варианта использования и действующих лиц

Модель анализа (Analysis Model).

Она включает основные классы, необходимые для реализации выделенных вариантов использования, а также возможные связи между классами. Выделяемые классы разбиваются на три разновидности — **интерфейсные**, **управляющие** и **классы данных**. Эти классы представляют собой набор сущностей, в терминах которых работа системы должна представляться пользователям. Они являются понятиями, с помощью которых достаточно удобно объяснять себе и другим происходящее внутри системы, не слишком вдаваясь в детали.

Интерфейсные классы (boundary classes) соответствуют устройствам или способам обмена данными между системой и ее окружением, в том числе пользователями. **Классы данных (entity classes)** соответствуют наборам данных, описывающих некоторые однотипные сущности внутри системы. Эти сущности являются абстракциями представлений пользователей о данных, с которыми работает система. **Управляющие классы (control classes)** соответствуют алгоритмам, реализующим какие-то значимые преобразования данных в системе и управляющим обменом данными с ее окружением в рамках вариантов использования.

В нашем примере с Интернет-магазином можно было бы выделить следующие классы в *модели анализа*: интерфейсный класс, предоставляющий информацию о товаре и возможность сделать заказ; интерфейсный класс, представляющий сообщение оператору; *управляющий класс*, обрабатывающий введенную пользователем информацию и преобразующий ее в данные о заказе и сообщение оператору; класс данных о заказе. Соответствующая модель приведена на рис. 3.7.

Каждый класс может играть несколько ролей в реализации одного или нескольких вариантов использования. Каждая роль определяет его обязанности и свойства, тоже являющиеся частью *модели анализа*.

В рамках других подходов *модель анализа* часто называется **концептуальной моделью** системы. Она состоит из набора классов, совместно реализующих все варианты использования и служащих основой для понимания работы системы и объяснения ее правил всем заинтересованным лицам.

Рис. 3.7. Пример модели анализа для одного варианта использования

Модель проектирования (Design Model)

Модель проектирования является детализацией и специализацией *модели анализа*. Она также состоит из классов, но более четко определенных, с более точным и детальным *распределением обязанностей*, чем классы *модели анализа*. Классы модели проектирования должны быть специализированы для конкретной используемой платформы. Каждая такая платформа может включать: операционные системы всех вовлеченных машин; используемые языки программирования; интерфейсы и классы конкретных *компонентных сред*, таких как J2EE, .NET, COM или CORBA; интерфейсы выбранных для использования систем управления базами данных, СУБД, например, Oracle или MS SQL Server; используемые библиотеки разработки пользовательского интерфейса, такие как *swing* или swt в Java, MFC или gtk; интерфейсы взаимодействующих систем и пр.

В нашем примере, прежде всего, необходимо детализировать классы, уточнить их функциональность. Скажем, для того, чтобы клиенту было удобнее делать заказ, нужно предоставить ему список имеющихся товаров, какие-то способы навигации и поиска в этом списке, а также деталь-

ную информацию о товаре. Это значит, что интерфейс заказа товара реализуется в виде набора классов, представляющих, например, различные страницы сайта магазина. Точно так же данные заказа должны быть детализированы в виде нескольких таблиц в СУБД, включающих, как правило, данные самого заказа (дату, ссылку на данные клиента, строки с количеством отдельных товаров и ссылками на товары), данные товаров, клиента и пр. Кроме того, для реляционной СУБД понадобятся классы-посредники между ее таблицами и объектной структурой остальной программы. Обработчик заказа может быть реализован в виде набора объектов нескольких классов, например, с выделенным отдельно набором часто изменяемых политик (скидки на определенные категории товаров и определенным категориям клиентов, сезонные скидки, рекламные комплекты и пр.) и более постоянным общим алгоритмом обработки.

Далее, приняв, например, решение реализовывать систему с помощью технологий J2EE или .NET, мы тем самым определяем дополнительные ограничения на структуру классов, да и на само их количество. О правилах построения ПО на основе этих технологий рассказывается в следующих лекциях.

Модель реализации (Implementation Model).

Под **моделью реализации** в рамках *RUP* и UML понимают набор компонентов результирующей системы и связей между ними. Под компонентом здесь имеется в виду **компонент сборки** — минимальный по размерам кусок кода системы, который может участвовать или не участвовать в определенной ее конфигурации, единица сборки и *конфигурационного управления*. Связи между компонентами представляют собой зависимости между ними. Если компонент зависит от другого компонента, он не может быть поставлен отдельно от него.

Часто компоненты представляют собой отдельные файлы с исходным кодом. Далее мы познакомимся с компонентами J2EE, состоящими из нескольких файлов.

Модель развертывания (Deployment Model)

Модель развертывания представляет собой набор узлов системы, являющихся физически отдельными устройствами, которые способны обрабатывать информацию — серверами, рабочими станциями, принтерами, контроллерами датчиков и пр., со связями между ними, образованными различного рода сетевыми соединениями. Каждый узел может быть нагружен некоторым множеством компонентов, определенных в модели реализации.

Цель построения модели развертывания — определить физическое положение компонентов распределенной системы, обеспечивающее выполнение ею нужных функций в тех местах, где эти функции будут доступны и удобны для пользователей.

В нашем примере Web-сайта магазина узлами системы являются один или несколько компьютеров, на которых развернуты Web-сервер, пересылающий по запросу пользователя текст нужной странички, набор программных компонентов, отвечающих за генерацию страничек, обработку действий пользователя и взаимодействие с базой данных, и СУБД, в рамках которой работает база данных системы. Кроме того, в систему входят все компьютеры клиентов, на которых работает Web-браузер, делающий возможным просмотр страничек сайта и пересылку кодированных действий пользователя для их обработки.

Модель тестирования (Test Model или Test Suite)

В рамках этой модели определяются **тестовые варианты** или **тестовые примеры (test cases)** и **тестовые процедуры (test scripts)**. Первые являются определенными сценариями работы одного или нескольких действующих лиц с системой, разворачивающимися в рамках одного из вариантов использования. Тестовый вариант включает, помимо входных данных на каждом шаге, где они могут быть введены, условия выполнения отдельных шагов и корректные ответы системы для всякого шага, на котором ответ системы можно наблюдать. В отличие от вариантов использования, в тестовых вариантах четко определены входные данные, и, соответственно, тестовый вариант либо вообще не имеет альтернативных сценариев, либо предусматривает альтернативный порядок действий в том случае, если система может вести себя недетерминированно и выдавать разные результаты в ответ на одни и те же действия. Все другие альтернативы обычно заканчиваются вынесением вердикта о некорректной работе системы.

Тестовая процедура представляет собой способ выполнения одного или нескольких тестовых вариантов и их составных элементов (отдельных шагов и проверок). Это может быть инструкция по ручному выполнению входящих в тестовый вариант действий или программный компонент, автоматизирующий запуск тестов.

Для выделенного варианта использования "Заказ товара" можно определить следующие тестовые варианты:

- заказать один из имеющихся на складе товаров и проверить, что сообщение об этом заказе поступило оператору;
- заказать большое количество товаров и проверить, что все работает так же;
- заказать отсутствующий на складе товар и проверить, что в ответ приходит сообщение о его отсутствии;
- сделать заказ от имени пользователя, помещенного в "черный список", и проверить, что в ответ приходит сообщение о неоплаченных прежних заказах.

RUP также определяет **дисциплины**, включающие различные наборы деятельностей, которые в разных комбинациях и с разной интенсивностью выполняются на разных фазах. В документации по процессу каждая дисциплина сопровождается довольно большой диаграммой, поясняющей действия, которые нужно выполнить в ходе работ в рамках данной дисциплины, артефакты, с которыми надо иметь дело, и роли вовлеченных в эти действия лиц.

Моделирование предметной области (бизнес-моделирование, Business Modeling)

Задачи этой деятельности — понять предметную область или бизнес-контекст, в которых должна будет работать система, и убедиться, что все заинтересованные лица понимают его одинаково, осознать имеющиеся проблемы, оценить их возможные решения и их последствия для бизнеса организации, в которой будет работать система.

В результате моделирования предметной области должна появиться ее модель в виде набора диаграмм классов (объектов предметной области) и деятельностей (представляющих бизнес-операции и бизнес-процессы). Эта модель служит основой *модели анализа*.

Определение требований (Requirements)

Задачи — понять, что должна делать система, и убедиться во взаимопонимании по этому поводу между заинтересованными лицами, определить границы системы и основу для планирования проекта и оценок затрат ресурсов в нем.

Требования принято фиксировать в виде модели вариантов использования.

Анализ и проектирование (Analysis and Design)

Задачи — выработать архитектуру системы на основе требований, убедиться, что данная архитектура может быть основой работающей системы в контексте ее будущего использования.

В результате проектирования должна появиться *модель проектирования*, включающая диаграммы классов системы, диаграммы ее компонентов, *диаграммы взаимодействий* между объектами в ходе реализации вариантов использования, диаграммы состояний для отдельных объектов и *диаграммы развертывания*.

Реализация (Implementation)

Задачи — определить структуру исходного кода системы, разработать код ее компонентов и протестировать их, интегрировать систему в работающее целое.

Тестирование (Test)

Задачи — найти и описать дефекты системы (проявления недостатков ее качества), оценить ее качество в целом, оценить, выполнены или нет гипотезы, лежащие в основе проектирования, оценить степень соответствия системы требованиям.

Развертывание (Deployment)

Задачи — установить систему в ее рабочем окружении и оценить ее работоспособность на том месте, где она должна будет работать.

Управление конфигурациями и изменениями (Configuration and Change Management)

Задачи — определение элементов, подлежащих хранению в репозитории проекта и правил построения из них согласованных конфигураций, поддержание целостности текущего состояния системы, проверка согласованности вносимых изменений.

Управление проектом (Project Management)

Задачи — планирование, управление персоналом, обеспечение взаимодействия на благо проекта между всеми заинтересованными лицами, управление рисками, отслеживание текущего состояния проекта.

Управление средой проекта (Environment)

Задачи — подстройка процесса под конкретный проект, выбор и замена технологий и инструментов, используемых в проекте.

Первые пять дисциплин считаются рабочими, остальные — поддерживающими. Распределение объемов *работ по* дисциплинам в ходе проекта выглядит, согласно руководству *по RUP*, примерно так, как показано на рис. 3.8.

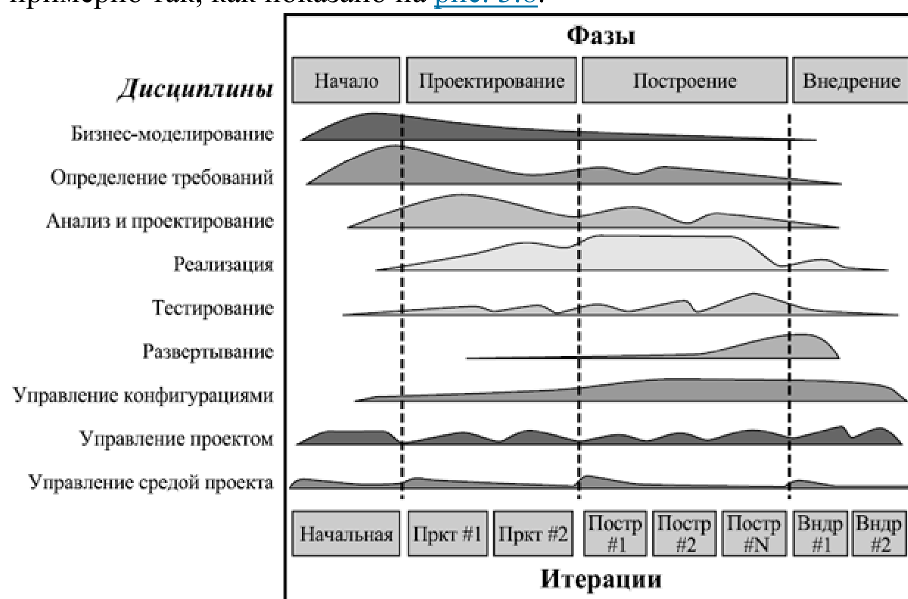


Рис. 3.8. Распределение работ между различными дисциплинами в проекте по RUP

Техники, используемые в *RUP* :

- Выработка концепции проекта (project vision) в его начале для четкой постановки задач.
- Управление по плану.
- Снижение рисков и отслеживание их последствий, как можно более раннее начало работ по преодолению рисков.
- Тщательное экономическое обоснование всех действий — делается только то, что нужно заказчику и не приводит к невыгодности проекта.
- Как можно более раннее формирование базовой архитектуры.
- Использование *компонентной архитектуры*.
- Прототипирование, инкрементная разработка и тестирование.
- Регулярные оценки текущего состояния.
- Управление изменениями, постоянная отработка изменений извне проекта.
- Нацеленность на создание продукта, работоспособного в реальном окружении.
- Нацеленность на качество.
- Адаптация процесса под нужды проекта.

Анализ предметной области

Для того чтобы разработать программную систему, приносящую реальные выгоды определенным пользователям, необходимо сначала выяснить, какие же задачи она должна решать для этих людей и какими свойствами обладать.

Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц. Однако сформулировать *требования* к сложной системе не так легко. В большинстве случаев будущие пользователи могут перечислить набор свойств, который они хотели бы видеть, но никто не даст гарантий, что это — исчерпывающий *список*. Кроме того, часто сама формулировка этих свойств будет непонятна большинству программистов: могут прозвучать фразы типа "должно использоваться и частотное, и временное уплотнение каналов", "передача клиента должна быть мягкой", "для обычных швов отмечайте бригаду, а для доверительных — конкретных сварщиков", и это еще не самые тяжелые для понимания примеры.

Чтобы *ПО* было действительно полезным, важно, чтобы оно удовлетворяло реальные потребности людей и организаций, которые часто отличаются от непосредственно выражаемых пользователями желаний. Для выявления этих потребностей, а также для выяснения смысла высказанных *требований* приходится проводить достаточно большую дополнительную работу, которая называется ***анализом предметной области*** или **бизнес-моделированием**, если речь идет о потребностях коммерческой организации. В результате этой деятельности разработчики должны научиться понимать язык, на котором говорят пользователи и заказчики, выявить цели их деятельности, определить набор задач, решаемых ими.

В *дополнение* стоит выяснить, какие вообще задачи нужно уметь решать для достижения этих целей, выяснить свойства результатов, которые хотелось бы получить, а также определить набор сущностей, с которыми приходится иметь дело при решении этих задач. Кроме того, *анализ предметной области* позволяет выявить места возможных улучшений и оценить последствия принимаемых решений о реализации тех или иных функций.

После этого можно определять область ответственности будущей программной системы — какие именно из выявленных задач будут ею решаться, при решении каких задач она может оказать существенную помощь и чем именно. Определив эти задачи в рамках общей системы задач и деятельностей пользователей, можно уже более точно сформулировать *требования к ПО*.

Анализом предметной области занимаются системные аналитики или бизнес-аналитики, которые передают полученные ими знания другим членам проектной команды, сформулировав их на более понятном разработчикам языке. Для передачи этих знаний обычно служит некоторый набор моделей, в виде графических схем и текстовых документов.

Анализ деятельности крупной организации, такой как банк с сетью региональных отделений, нефтеперерабатывающий завод или компания, производящая автомобили, дает огромные объемы информации. Из этой информации надо уметь отбирать существенную, а также уметь находить в ней пробелы — области деятельности, информации *по* которым недостаточно для четкого представления о решаемых задачах. Значит, всю получаемую информацию надо каким-то образом систематизировать. Для систематизации сбора информации о больших организациях и дальнейшей разработки систем, поддерживающих их *деятельность*, применяется *схема Захмана* или **архитектурная схема предприятия (enterprise architecture framework)**.

	Мотивация	Люди	Данные	Функции	Место	Время
Контекст	Цели и стратегия бизнеса	Важные для бизнеса организации	Вещи, значимые для бизнеса	Основные бизнес-процессы	География бизнеса	События и периоды, важные для бизнеса
Модель бизнеса	Бизнес-план, частные цели и стратегии	Модели потоков работ	Семантические модели Бизнес-сущности и их связи	Модели бизнес-процессов	Система логистики	Базовый график работ
Системная модель	Модель бизнес-правил	Архитектура пользовательского интерфейса	Концептуальная модель данных	Архитектура приложений	Архитектура распределенной системы	Структура обработки событий
Технологическая модель	Модель правил обработки событий	Архитектура представления	Физическая модель данных	Архитектура программно-аппаратной системы	Технологическая архитектура	Структура циклов управления
Детальное представление	Спецификации правил работы системы	Спецификации ролей и прав доступа	Спецификации форматов данных	Код программных компонентов	Спецификации архитектуры сети	Спецификации обработки событий и прерываний
Работающая организация	Стратегия и тактика	Структура организации	Данные	Выполняемые функции	Географическое расположение и сети	Планы

Рис. 4.1. Схема Захмана. Приведены примеры моделей для отдельных клеток

В основе ***схемы Захмана*** лежит следующая идея: *деятельность* даже очень большой организации можно описать, используя ответы на простые вопросы — зачем, кто, что, как, где и когда — и разные уровни рассмотрения. Обозначенные 6 вопросов определяют 6 аспектов рассмотрения.

- Цели организации и базовые правила, по которым она работает.
- Персонал, подразделения и другие элементы организационной структуры, связи между ними.
- Сущности и данные, с которыми имеет дело организация.
- Выполняемые организацией и различными ее подразделениями функции и операции над данными.
- Географическое распределение элементов организации и связи между географически разделенными ее частями.
- Временные характеристики и ограничения на деятельность организации, значимые для ее деятельности события.

Также выделены несколько уровней рассмотрения, из которых при бизнес-моделировании особенно важны три верхних:

- Самый крупный — уровень организации в целом, рассматриваемой в ее развитии совместно с окружением, уровень общего планирования ее деятельности. Этот уровень содержит долговременные цели и задачи организации как цельной системы, основные связи организации с внешним миром и основные виды ее деятельности.
- Уровень бизнеса, на котором организация рассматривается во всех аспектах как отдельная сущность, имеющая определенную структуру, которая соответствует ее основным задачам.
- Системный уровень, на котором определяются концептуальные модели всех аспектов организации, без привязки к конкретным их воплощениям и реализациям, например, *логическая модель данных* в виде набора сущностей и связей между ними, логическая архитектура системы автоматизации в виде набора узлов с привязанными к ним функциями и пр.

Наиболее удобной формой представления информации при *анализе предметной области* являются графические диаграммы различного рода. Они позволяют достаточно быстро зафиксировать полученные знания, быстро восстанавливать их в памяти и успешно объясняться с заказчиками и другими заинтересованными лицами. Набросать рисунок из прямоугольников и связывающих их стрелок обычно можно гораздо быстрее, чем записать соответствующий объем информации, и на рисунке за один взгляд видно гораздо больше, чем в тексте. Изредка встречаются лю-

ди, лучше ориентирующиеся в текстах и более адекватно их понимающие, но чаще рисунки все же более удобны для иллюстрации мыслей и объяснения сложных вещей.

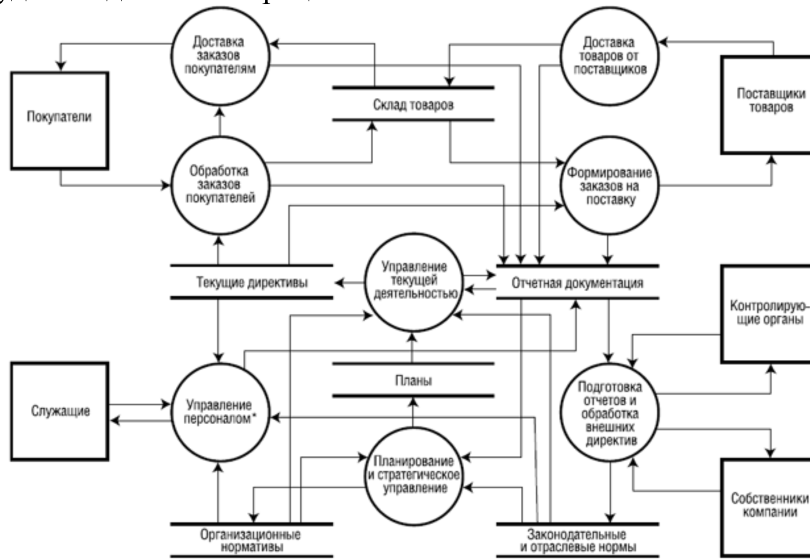


Рис. 4.2. Схема деятельности компании в нотации Йордана-ДеМарко

Для описания поведения сложных систем и деятельности крупных организаций используются *диаграммы потоков данных (data flow diagrats)*. Эти *диаграммы* содержат 4 вида графических элементов: **процессы**, представляющие собой любые *трансформации данных* в рамках описываемой системы, **хранилища данных**, внешние *по* отношению к системе **сущности** и **потоки данных** между элементами трех предыдущих видов.

Используются несколько систем обозначений для перечисленных элементов, наиболее известны *нотация* Йордана-ДеМарко и *нотация* Гэйна-Сарсона, обе предложенные в 1979 году. Рис. 4.3 показывает *диаграмму потоков данных*, которая описывает *деятельность* компании, управляющей небольшим магазином. Эта *диаграмма* изображена в нотации Йордана-ДеМарко: процессы изображаются кружками, *внешние сущности* — прямоугольниками, а хранилища данных — двумя горизонтальными параллельными линиями. На рис. 4.3 изображена та же *диаграмма* в нотации Гейна-Сарсона: на ней процессы — прямоугольники со скругленными углами, *внешние сущности* — прямоугольники с тенью, а хранилища данных — вытянутые горизонтально прямоугольники без правого ребра.

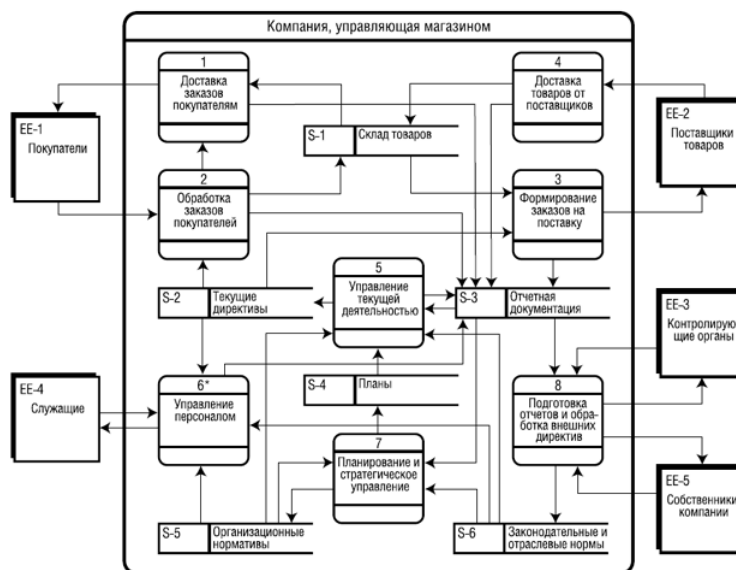


Рис. 4.3. Схема деятельности компании в нотации Гэйна-Сарсона

Процессы на *диаграммах потоков данных* могут уточняться: если некоторый процесс устроен достаточно сложно, для него можно нарисовать отдельную диаграмму, описывающую *потоки данных* внутри этого процесса. На ней показываются те элементы, с которыми этот процесс связан потоками данных, и составляющие его более мелкие процессы и хранилища. Таким образом, возникает иерархическая структура процессов. Обычно на самом верхнем уровне находится один процесс, представляющий собой систему в целом, и набор внешних сущностей, с которыми она взаимодействует.

На рис. 4.4 показана возможная *детализация* процесса "*Управление персоналом*".

Диаграммы потоков данных появились как один из первых инструментов представления деятельности сложных систем при использовании **структурного анализа**. Для представления структуры данных в этом подходе используются *диаграммы сущностей и связей*, изображающие набор *сущностей предметной области* и связей между ними. И сущности, и связи на таких диаграммах могут иметь атрибуты. Пример такой диаграммы представлен на рис. 4.5.

Хотя методы структурного анализа могут значительно помочь при анализе систем и организаций, дальнейшая разработка системы, поддерживающей их *деятельность*, с использованием объектно-ориентированного подхода часто требует дополнительной работы *по* переводу полученной информации в объектно-ориентированные модели.

Методы объектно-ориентированного анализа предназначены для обеспечения более удобной передачи информации между моделями анализируемых систем и моделями разрабатываемого *ПО*. В качестве графических моделей в этих методах вместо *диаграмм потоков данных* используются рассматривавшиеся при обсуждении *RUP диаграммы вариантов использования*, а вместо *диаграмм сущностей и связей — диаграммы классов*.

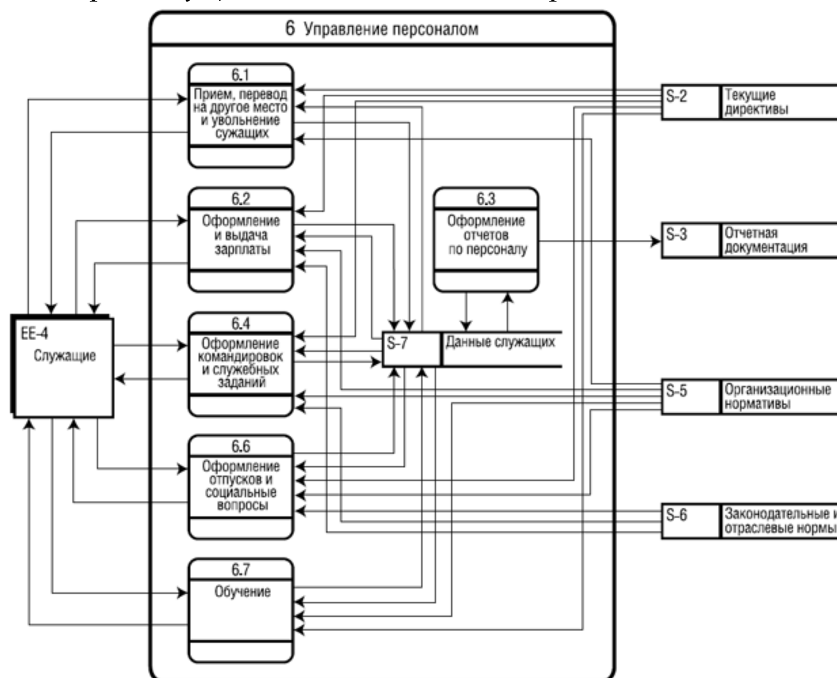


Рис. 4.4. Детализация процесса "Управление персоналом"

Однако *диаграммы вариантов использования* несут несколько меньше информации *по* сравнению с соответствующими *диаграммами потоков данных*: на них процессы и хранилища в соответствии с принципом объединения данных и методов работы с ними объединяются в *варианты использования*, и остаются только связи между *вариантами использования* и *действующими лицами* (аналогом внешних сущностей). Для представления остальной информации каждый *вариант использования* может дополняться набором разнообразных диаграмм *UML* — диаграммами деятельностей, *диаграммами сценариев* и пр. Обо всех этих видах диаграмм будет рассказано в лекции, посвященной архитектуре программного обеспечения.

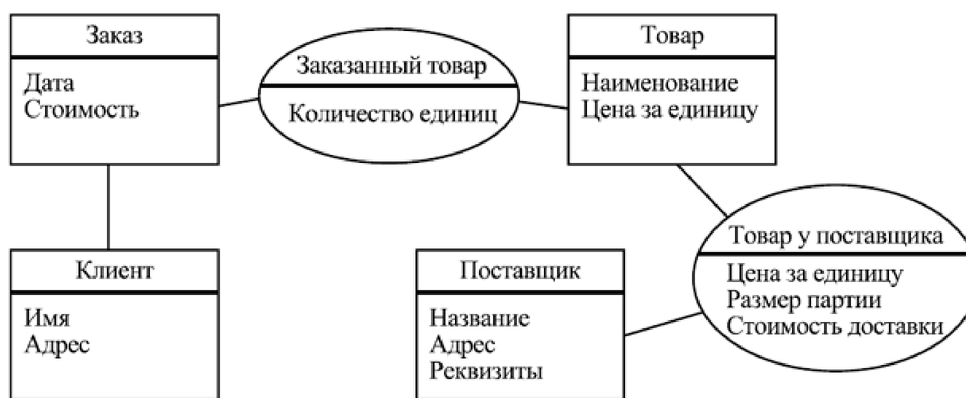


Рис. 4.5. Модель сущностей и связей

Выделение и анализ требований

После получения общего представления о деятельности и целях организаций, в которых будет работать будущая программная система, и о ее *предметной области*, можно определить более четко, какие именно задачи система будет решать. Кроме того, важно понимать, какие из задач стоят наиболее остро и обязательно должны быть поддержаны уже в первой версии, а какие могут быть отложены до следующих версий или вообще вынесены за рамки области ответственности системы. Эта *информация* выявляется при анализе потребностей возможных пользователей и заказчиков.

Потребности определяются на основе наиболее актуальных проблем и задач, которые пользователи и заказчики видят перед собой. При этом требуется аккуратное выявление значимых проблем, *определение* того, насколько хорошо они решаются при текущем положении дел, и расстановка приоритетов при рассмотрении недостаточно хорошо решаемых, поскольку чаще всего решить сразу все проблемы невозможно.

Формулировка потребностей может быть разбита на следующие этапы.

1. Выделить одну-две-три основных проблемы.
2. Определить причины возникновения проблем, оценить степень их влияния и выделить наиболее существенные из проблем, влекущие появление остальных.
3. Определить ограничения на возможные решения.

Формулировка потребностей не должна накладывать лишних ограничений на возможные решения, удовлетворяющие им. Нужно попытаться сформулировать, что именно является проблемой, а не предлагать сразу возможные решения.

Например, формулировки "система должна использовать *СУБД Oracle* для хранения данных", "нужно, чтобы при вводе неверных данных раздавался звуковой сигнал" не очень хорошо описывают потребности. Исключением в первом случае может быть особая ситуация, например, если *СУБД Oracle* уже используется для хранения других данных, которые должны быть интегрированы с рассматриваемыми: при этом ее использование становится внешним ограничением. Соответствующие потребности лучше описать так: "нужно организовать надежное и удобное для интеграции с другими системами *хранение данных*", "необходимо предотвращать попадание некорректных данных в хранилище".

При выявлении потребностей пользователей анализируются модели деятельности пользователей и организаций, в которых они работают, для выявления проблемных мест. Также используются такие приемы, как анкетирование, демонстрация возможных сеансов работы будущей системы, интерактивные опросы, где пользователям предоставляется возможность самим предложить варианты внешнего вида системы и ее работы или поменять предложенные кем-то другим, демонстрация прототипа системы и др.

После выделения основных потребностей нужно решить вопрос о разграничении области ответственности будущей системы, т.е. определить, какие из потребностей надо пытаться удовлетворить в ее рамках, а какие — нет.

При этом все *заинтересованные лица* делятся на пользователей, которые будут непосредственно использовать создаваемую систему для решения своих задач, и вторичных заинтересованных лиц, которые не решают своих задач с ее помощью, но чьи интересы так или иначе затрагиваются ею. *Потребности пользователей* нужно удовлетворить в первую *очередь* и на это нужно выделить больше усилий, а интересы вторичных заинтересованных лиц должны быть только адекватно учтены в итоговой системе.

На основе выделенных потребностей пользователей, отнесенных к области ответственности системы, формулируются возможные *функции* будущей системы, которые представляют собой услуги, предоставляемые системой и удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения таких *функций* можно брать из имеющегося опыта разработчиков (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки идей.

Формулировка функций должна быть достаточно короткой, ясной для пользователей, без лишних деталей. Например:

- Все данные о сделках и клиентах будут сохраняться в базе данных.
- Статус выполнения заказа клиент сможет узнать через Интернет.
- Система будет поддерживать до 10000 одновременно работающих пользователей.
- Расписание проведения ремонтных работ будет строиться автоматически.

Предлагая те или иные функции, нужно уметь аккуратно оценивать их влияние на структуру и *деятельность* организаций, в рамках которых будет использоваться *ПО*. Это можно сделать, имея полученные при *анализе предметной области* модели их текущей деятельности.

Имея набор функций, достаточно хорошо поддерживающий решение наиболее существенных задач, с которыми придется работать разрабатываемой системе, можно составлять *требования* к ней, представляющие собой детализацию работы этих функций. Соотношение между проблемами, потребностями, функциями и требованиями показано на рис. 4.6.

Рис. 4.6. Соотношение между проблемами, потребностями, функциями и требованиями

При этом часто нужно учитывать, что *ПО* является частью программно-аппаратной системы, требования к которой надо преобразовать в требования к программной и аппаратной ее составляющим. В последнее время, в связи со значительным падением цен на мощное *аппаратное обеспечение* общего назначения, фокус внимания переместился, в основном, на *программное обеспечение*. Во многих проектах аппаратная платформа определяется из общих соображений, а поддержку большинства нужных функций осуществляет *ПО*.

Каждое *требование* раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть *требований* исходит из потребностей и

пожеланий заинтересованных лиц и решений, удовлетворяющих эти потребности и пожелания, а часть — из *внешних ограничений*, накладываемых на систему, например, основными законами той *предметной области*, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр.

Еще до перехода от функций к *требованиям* полезно расставить приоритеты и оценить трудоемкость их реализации и рискованность. Это позволит отказаться от реализации наименее важных и наиболее трудоемких, не соответствующих бюджету проекта функций еще до их детальной проработки, а также выявить возможные проблемные места проекта — наиболее трудоемкие и неясные из вошедших в него функций.

Общая характеристика структурного программирования

На самом деле изложение структурного стиля не может уместиться в рамки одной лекции. Но данный стиль программирования (вернее, его вариант, основанный на *циклах* и массивах, слегка пополненный рекурсивными процедурами) описывается и навязывается как единственно возможный во всех ныне предлагаемых учебных пособиях по программированию на традиционных языках. В связи с этим мы **имеем право** предположить, что обучающийся знаком с ним (более того, знаком только с ним, и мы надеемся, что он еще не потерял способность воспринимать другие стили). И хотя Вы считаете, что с этим вариантом структурного стиля уже освоились, особенности, опускаемые в традиционных изложениях, могут полностью изменить Ваш взгляд на данный стиль.

Мы рассматриваем структурное *программирование* как равноправный член сообщества альтернативных ему друзей-соперников.

В теории схем программ было замечено, что некоторые случаи блок-схем легче поддаются анализу . Поэтому естественно было выделить такой *класс* блок-схем, что и сделали итальянские ученые С. Бем и К. Джакопини в 1966 г. Они доказали, что любую блок-схему можно привести к структурированному виду, использовав несколько дополнительных булевых переменных. *Э. Дейкстра* подчеркнул, что программы в таком виде, как правило, являются легче понимаемыми и модифицируемыми, так как каждый блок имеет один вход и один *выход*.

В качестве методики структурного программирования *Э. Дейкстра* предложил пользоваться лишь конструкциями *цикла* и условного оператора, изгоняя go to как концептуально противоречащее этому стилю

Пожалуй, это первое *концептуальное противоречие*, явно отмеченное и учтенное в теории и практике программирования (и даже во всей современной науке). Но, поскольку не было даже наметок теории неформализуемых понятий, и на работу с ними переносили *опыт* работы с малыми формализациями, структурное противоречие было воспринято следующим образом.

К несчастью, оператор go to формально совместим с другими конструкциями традиционных (тогда говорили - универсальных) алгоритмических языков. Но реально он плохо взаимодействует с ними. Значит, он плох сам по себе.

Структурное *программирование* основано главным образом на теоретическом аппарате теории рекурсивных функций. *Программа* рассматривается как частично-рекурсивный оператор над библиотечными подпрограммами и исходными операциями. Структурное *программирование* базируется также на теории доказательств, прежде всего на естественном выводе. Структура программы соответствует структуре простейшего математического рассуждения, не использующего сложных лемм и абстрактных понятий[2].

Средства структурного программирования в первую *очередь* включаются во все языки программирования традиционного типа и во многие нетрадиционные языки. Они занимают основное *место* в учебных курсах программирования и в теоретических работах.

При структурном программировании присваивания и локальные действия становятся органичной частью программы. Достаточно лишь внимательно следить, чтобы каждая *переменная* в модуле использовалась для одной конкретной цели, и не допускать "экономии", при которой ненужная в данном месте *переменная* временно используется под совсем другое *значение*. Такая

"экономия" запутывает структуру информационных зависимостей, которая при данном стиле должна быть хорошо согласована со структурой программы.

Структурное *программирование* естественно возникает во многих классах задач, прежде всего в таких, где **задача естественно расщепляется на подзадачи, а информация - на достаточно независимые структуры данных**. Основной его *инвариант*:

действия и условия локальны.

Необходимой чертой хорошей реализации структурного стиля программирования является соблюдение согласованности, а в идеале и единства, следующих компонентов программы:

1. **Структура информационного пространства**. Содержательно любую задачу можно описать как переработку объектов, полный набор которых называется **информационным пространством** задачи.

Для структурного стиля программирования требуется следующее. Задача разбивается на подзадачи, и таким образом выстраивается дерево вложенности подзадач. Информационное пространство структурируется в точном соответствии с деревом вложенности: для каждой подзадачи оно состоит из ее **локальных объектов**, определяемых вместе с подзадачей и для нее, и так называемых **глобальных объектов**, определяемых как информационное пространство непосредственно объемлющей подзадачи. Таким образом, информационное пространство всей задачи (подзадачи самого верхнего уровня) расширяется по мере перехода к подзадачам за счет их локальных объектов. Для различных дочерних подзадач одной подзадачи оно имеет общую часть - информационное пространство родительской подзадачи[3].

2. **Структуры управления**. Стиль структурного программирования в его общепринятом варианте предполагает использование строго ограниченного набора *управляющих конструкций*: последовательность операторов, условные и выбирающие операторы, все вычислительные ветви которых сходятся в одной точке программы, а также процедуры, вычисления которых всегда заканчиваются возвратом управления в точку вызова.

3. К структурным операторам добавляются **либо** *циклы*, **либо** *рекурсии*.

Внимание!

Этой альтернативы Вы не встретите в традиционных изложениях структурного программирования. *Концептуальное противоречие* между *циклами* и *рекурсиями* намного мягче, чем между операторами структурного программирования и *структурными переходами*, и оно отмечается лишь в виде изредка встречающихся прагматических указаний (благих пожеланий) не смешивать их произвольно.

4. **Потоки передачи данных**. Разбивая задачу на подзадачи, программист предусматривает их взаимодействие по данным: одни подзадачи передают другим данные для переработки.

5. **Структуры данных**. Данные объединяются в логически связанные фрагменты, соответствующие структурам задачи либо вспомогательных конструкций, вводимых для ее решения.

6. **Призраки**. Часто даже сама программа не может быть объяснена через понятия, которые используются внутри нее. Еще чаще это происходит для ее связей с внешним миром. Понимание программы возможно лишь после сопоставления реальных внутрипрограммных объектов с идеальными внепрограммными. Эти идеальные внепрограммные объекты (*призраки*) часто не просто не нужны, но даже вредны для исполнения программы[4].

Первым обратил внимание на необходимость введения *призраков* для логического и концептуального анализа программ *Г. С. Цейтин* в 1971 г. В Америке это "независимо" открыли заново в 1979 г., хотя упомянутая статья *Цейтина* была опубликована на английском языке в общедоступном издании. Даже название сущностям было дано то же самое...

Подпорки в программе - значения, конструкции и сущности, которые не нужны для понимания задачи и программы, не определяются сущностью задачи и алгоритма, но вынужденно вставляются для реализации данного алгоритма на конкретной системе.

Подпорки противоположны *призракам*. На самом деле они являются той формой, в которой материя проникает в программу, и в этом качестве противостоят всей совокупности идеальных сущностей, порождающих структуру программы: как реальных, так и призрачных, - и порою гру-

бо ее искажают. Но без *подпорок* программа просто не будет работать или будет работать неэффективно.

Для структурного программирования весьма важно требование:

Все структуры подчиняются структуре информационного пространства.

Это общее требование конкретизируется в следующие.

1. Необходимо, чтобы структура управления программы была согласована со структурой ее информационного пространства. Каждой структуре управления соответствуют согласующиеся с ней структуры данных и часть информационного пространства. Это условие позволяет человеку легко отслеживать порядок выполнения конструкций в программе.
2. Подзадачи могут обмениваться данными только посредством обращения к объектам из общей части их информационных пространств (в современных языках чаще всего к глобальным).
3. Информационные потоки должны протекать согласно иерархии структур управления; мы должны четко видеть для каждого блока программы, что он имеет на входе и что дает на выходе. Таким образом, **свойства каждого логически завершенного фрагмента программы должны ясно осознаваться и в идеале четко описываться в самом тексте программы и в сопровождающей ее** документации[5].
4. Описание переменных, представляющих перерабатываемые объекты, а также других, вспомогательных переменных при структурном программировании строго подчиняется разбиению задачи на подзадачи.
5. Все *призраки* действуют на своем структурном месте и соответствуют идеальным сущностям, которые, согласно парадоксу изобретателя, должны вводиться для эффективного решения задачи.
6. Все *подпорки* строго локализованы в том месте, где их вынуждены ввести. Желательно даже обозначать их по-другому, чем идеальные сущности, например, оставляя мнемонические имена лишь для идеальных сущностей, а *подпорки* именовать джокерами типа x или i. Необходимо строго следить за тем, чтобы *подпорки* не искажали идеальную структуру программы.

Структурное *программирование* лучше всего описано теоретически, но частные описания не сведены в единую систему. Одни книги трактуют его с точки зрения программиста, другие - с точки зрения теоретика. Так что даже здесь единой системы взглядов еще нет, хотя, видимо, все основания для ее формирования уже имеются.

Сети данных

Рассмотрим основную структуру данных, которая появляется при структурном программировании. Учет этой структуры позволяет преобразовать благие пожелания о согласованности информационных потоков и хода передач управления в достаточно строгую методику.

Сеть данных может быть формально описана как **ациклический ориентированный граф, в котором все ко-пути (т. е. пути, взятые наоборот) конечны и вершинам которого сопоставлены значения**.

Рассмотрим пример. Известному стандартному приему программирования в языках без кратных присваиваний - обмену двух значений через промежуточное

```
z := second;
second := first;
first := z;
```

соответствует следующая *сеть данных*:

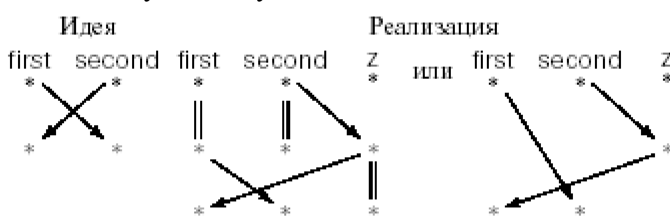


Рис. 14.1.1. Обмен значений

Здесь first, second, z можно считать комментариями, а сами данные опущены, поскольку их конкретные значения не важны.

На этом примере видно, что порой для лучшего структурирования сети целесообразно вводить дополнительные вершины, соответствующие сохраняющимся значениям. *Ребро*, ведущее из одной такой вершины в другую, обозначается при помощи стрелочки, похожей на *равенство*. Видно так же, как материя воздействует на идею, заставляя вводить дополнительные *операторы* и дополнительные значения.

В данном случае *переменная* z и включающие ее *операторы* являются *подпорками*, и, если их исключить, *сеть данных* становится проще. Но в общераспространенных языках программирования нет кратных присваиваний типа

```
first,second:=second,first;
```

Даже если бы они были, представьте себе, как неудобно станет читать длинное кратное *присваивание* и понимать, какое же *выражение* какой переменной присваивается!

В случае программы вычисления факториала[6] *сеть* потенциально бесконечна вниз, поскольку аргументом может быть любое число, но по структуре еще проще:

Рис. 14.1.2.

Перекрестных зависимостей между параметрами нет, следовательно, возможны две известные реализации факториала: циклическая и рекурсивная. Покажем их на разных языках, ибо все равно, на каком традиционном языке их писать.

```
function fact(n: integer): integer;
var j,res: integer;
begin
res:=1;
for j:=1 to n do res:=res*j;
result:=res;
end;
int fact(int n)
{if (n==0) return(1);
 else return(n*fact(n-1));}
```

Схема построения циклической программы называется **потоковой обработкой**. Значения на следующей итерации *цикла* зависят от значений на предыдущей.

Для чисел Фибоначчи структура уже несколько сложнее предыдущих, поскольку каждое следующее *число Фибоначчи* зависит от двух предыдущих, но метод потоковой обработки применим и здесь.

```
int fib(int n)
{int fib1,fib2;
fib1=1; fib2=1;
  if (n>2){
```

```
    for (int i=2;i<n;i++){
      int j; j=fib1+fib2; fib1=fib2; fib2=j;
    }
  };
  return(fib2);
}
```

Итак, в потоке изменяется структура из двух элементов. Ее можно было бы прямо описать как структуру данных, и это следовало бы сделать, будь *программа* хоть чуть-чуть посложнее. Тогда вместо *подпорки* j пришлось бы ввести в качестве *подпорки* новое *значение* структуры.

В программе имеется еще одна *подпорка* - *параметр цикла* i, который нужен лишь для формальной организации *цикла*.

Рекурсивная реализация чисел Фибоначчи пишется еще проще и служит великолепным примером того, как презренная материя убивает красивую, но неглубокую идею.

```
int fib(int n)
{ if (n<3) return(1);
   else return(fib(n-1)+fib(n-2));
}
```

Если n достаточно велико, каждое из предыдущих значений функции Фибоначчи будет вычисляться много раз, причем без всякого толку: результат всегда будет один и тот же! Зато все *подпорки* убраны...

В следующем примере неэффективность *рекурсии* по сравнению с хорошо организованным *циклом* еще более очевидная. Пусть надо найти *путь*, на котором можно собрать максимальное количество золота, через *сеть* значений, подобную показанной на рис. 14.1.

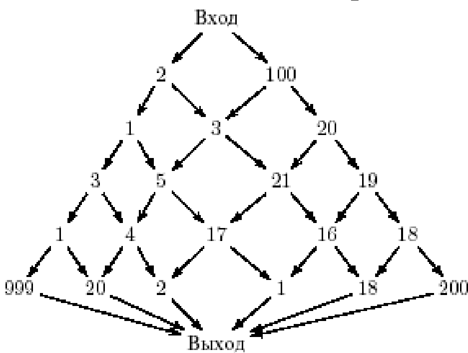


Рис. 14.1. Золотая гора

При циклической организации вычислений нам придется посчитать *значение* в каждой точке горы всего один раз (найти добычу на оптимальном пути в эту точку). Здесь используется то свойство оптимальных путей, которое делает возможным так называемые методы волны или *динамического программирования*: каждый начальный *отрезок* локально оптимального пути локально оптимален. Это свойство является *призраком*, стоящим за эффективной циклической реализацией алгоритма, а многочисленные пути, соответственно, призрачными значениями, которые не нужно вычислять. В рекурсивной реализации мы не учитываем данного *призрака*, и он беспощадно мстит за вопиющее незнание теории.

Теперь рассмотрим случай, когда рекурсивная реализация намного изящнее циклической, легче обобщается и не хуже по эффективности[7]

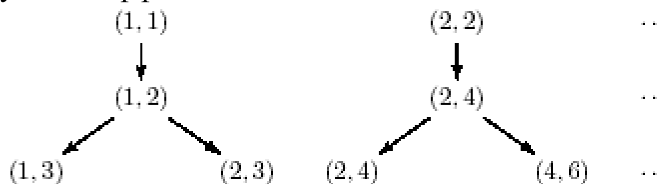


Рис. 14.1.3. Алгоритм Евклида

В данном случае *путь* для получения результата не разветвляется, и нам остается лишь двигаться по нему в правильном направлении (от цели к исходным данным) и достаточно большими шагами.

```
function Euklides(n,m: integer) integer; {
  предполагаем m<=n}
begin
  if n=m then resut:=n
    else result:=Euklides(n mod m, m);
end;
```

Если пытаться вычислить наибольший общий делитель методом движения от данных к цели, то нам придется построить громадный *массив* значений НОД, лишь ничтожная часть значений в котором будет нужна для построения результата. *Затраты* на *вычисление* каждого отдельного элемента в данном случае малы, а при обратном направлении движения повторный счет не возникает.

Алгоритм Евклида в простейшем случае моделирует ту ситуацию, которая появляется в задачах обработки рекурсивных структур, например списков. То, что в отдельных случаях возникает повторный счет, - небольшое зло, когда эти случаи редки и нерегулярны. Повторный счет возникает и в циклических программах, поскольку найти в большом массиве совпадающие элементы порою труднее, чем заново посчитать нужные нам значения. Именно поэтому *рекурсия* оказалась столь эффективным методом работы со списками и позволила построить адекватный первопорядковый фундамент для современного функционального программирования.

Рассмотрим два крайних случая движения по сети. Когда *сеть* представлена в виде структуры данных, естественно возникает метод ленивого движения по сети, когда после вычисления значений в очередной точке выбирается одна из точек, для которой все предыдущие значения уже вычислены, и вычисляются значения в данной точке. Как видно, в частности на примере золотой горы, этот метод недетерминирован и в значительной степени может быть распараллелен. Но в конкретном алгоритме нам придется выбрать конкретный способ ленивого движения, и он может быть крайне неудачен: например, он будет провоцировать длительное движение в *тупик*, когда у нас есть короткий *путь* к цели. Даже в теоретических исследованиях приходится накладывать условия на метод ленивого движения, чтобы гарантировать достижение результата .

Другой крайний случай движения по сети, когда *сеть* делится на одинаковые слои. Например, в сети

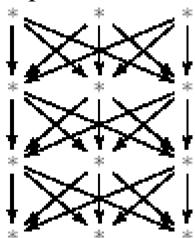


Рис. 14.1.4. Полностью заменяемый массив

можно представить слой *статическим массивом* и вроде бы полностью забыть о самой сети. Забытый *призрак* мстит за себя, в частности, при необходимости *распараллеливания вычислений*, и предыдущий случай отнюдь не эквивалентен следующему.

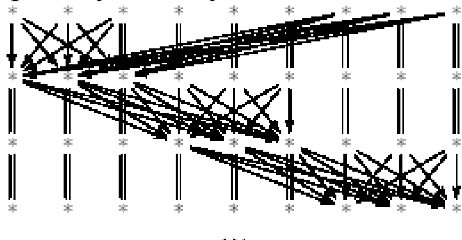


Рис. 14.1.5. Постепенно заменяемый массив

Конечно же большая *сеть данных* становится необозримой. Справиться с нею можно, лишь разбив *сеть* на подсети. Таким образом, блоки программы соответствуют относительно автономным подсетям.

Еще *Э. Дейкстра* предложил в каждом блоке описывать импортированные и экспортируемые им глобальные значения. Но такая "писанина" раздражала хакеров и в итоге так и не вошла в общепризнанные системы программирования. Сейчас индустриальные технологии требуют таких описаний, но из-за отсутствия поддержки на уровне синтаксического анализа все это остается благими пожеланиями, так что, если хотите, чтобы Ваша *программа* была понятна хотя бы Вам, описывайте все перекрестные информационные связи!

Резюмируя вышеизложенное, можно сделать следующие выводы.

1.*Сеть данных* сама по себе в программу не переходит, в программу переходят лишь некоторые свойства сети в качестве *призраков* и некоторые куски сети в качестве *реальных значений*.

2.Программа определяется не только *сетью данных*, но и конкретной дисциплиной движения по этой сети.

3.В случае, если очередные слои сети, появляющиеся при движении согласно заданной дисциплине, примерно одинаковы и состоят из многих взаимосвязанных значений, у нас возникает циклическая программа.

4.В случае, если вычисление можно свести к вычислениям для независимых элементов сети, чаще всего удобней *рекурсия*.

5.Самый общий способ движения по сети - ленивое движение, когда мы имеем право вычислить следующий объект сети, если вычислены все его предшественники.

6.Структурное программирование нейтрально по отношению к тому, каким именно способом будет исполняться полученная программа: последовательно, детерминированно, недетерминированно, совместно, параллельно либо даже на распределенной системе, - поскольку сеть лишь частично предписывает порядок действий.

7.Конкретный выбор порядка действий в последовательной детерминированной программе является *подпоркой*, о которой постоянно забывают. Поэтому он чаще всего вредит при перестройке программы.

Выбор

Рассмотренные до сих пор *сети данных* представляли в первую *очередь* тот случай, когда в программе нет значительных альтернативных блоков. Условие было лишь средством проверки перехода от одного этапа вычислений к другому. Однако на самом деле, как правило, *программа* содержит выбор. Для представления выбора в языках программирования имеются условные *операторы* и *операторы* выбора. Рассмотрим, что же стоит за выбором.

Пример 14.3.1. Пусть в некоторый момент исполнения программы Вам необходимо временно выбросить больший из двух хранимых в основной памяти обрабатываемых блоков на *диск*. Поскольку разница в длине менее 2^{16}=65536) несущественна, мы можем записать выбор примерно в следующей форме.

```
if
  length(A)-lengtn(B)>65536 -> {
    Save(A); Dispose(A); A_present:=false;},
  length(A)-lengtn(B)<65536 -> {
    Save(B); Dispose(B); B_present:=false;}
fi
```

Мы воспользовались данной формой, чтобы ярче подчеркнуть условия, при которых производятся действия.

Предложенная форма записи базируется на концепции *охраняемых команд*, предложенной *Э. Дейкстрой*. ***Охраняемая команда*** исполняется **лишь** при условии, когда выполнена охрана. Но если данный текст читает программист, он должен понимать, что 'лишь' не всегда означает, что при выполнении условия *команда* будет выполнена. *Оператор выбора* по *Дейкстре* состоит из *множества охраняемых команд*. В конкретном синтаксисе мы используем для них форму

Guard -> Command

Относительное расположение *охраняемых команд* в операторе выбора безразлично[8]. Выполняется **одна** из *охраняемых команд*, охрана которой истинна. Имеющиеся в языках конкретные формы условных предложений и предложений выбора являются *подпорками* для реализации *охраняемых команд*.

Из изложенного следует, что по своей сути выбор так же недетерминирован, как и *исполнение* структурной программы. Если выполнено несколько охран, с точки зрения задачи абсолютно все равно, какое из действий выбирать. Однако имеющиеся средства программирования[9] заставляют нас однозначно сделать выбор, и конечно же почти всегда мы забываем написать в комментариях, что на самом деле выбор безразличен, а затем при модификациях программы появляются *заплатки* на *подпорках* и т. п.

Когда имеется выбор, мы вынуждены переходить от *сети данных* к более сложной структуре: &- $\vee$ -графам. Некоторые вершины могут быть помечены как $\vee$ -вершины, это означает, что достаточно получить один из результатов, соответствующий входящим дугам, и инициировать лишь одно из исполнений, соответствующее выходящей дуге. Для структурированности &- $\vee$ -графа необходимо, чтобы он был сетью, удовлетворяющей следующему условию: имеется *инъекция* ψ, сопоставляющая каждой $\vee$ -вершине ν, из которой выходит несколько дуг, $\vee$ -вершину ψ(ν), из которой выходит лишь одна *дуга*, такую, что любой *путь*, проходящий через первую вершину, проходит и через вторую. Это неудобоваримое теоретическое условие всего лишь формулирует на точном языке, что $\vee$ -вершины должны группироваться в структуры следующего вида, показанного на рис. 14.2 (количество вариантов может быть любым).

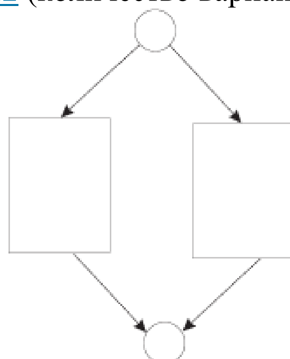


Рис. 14.2. Сеть охраняемых команд

О дисциплине циклического структурного программирования

Сейчас сосредоточимся на том варианте структурного программирования, который ориентируется на *циклы* и массивы.

Прежде всего, нужно остановиться на совместимости структурного циклического программирования с *рекурсиями*. *Опыт* показывает, что процедура, в которой есть ее *рекурсивный вызов* внутри *цикла*, практически почти всегда ошибочна, теоретически же она выходит за рамки примитивной *рекурсии* (см. курс логики и теории алгоритмов) и, как следствие, становится практически невычислимой. С тем, чтобы здесь не попасть впросак, соблюдайте простое правило.

Внимание!

Не используйте рекурсивный вызов процедуры внутри цикла! Рекурсия и циклы должны быть "территориально разделены"!

Данное правило пригодно в подавляющем большинстве случаев. Оно является конкретизацией для структурного программирования известного политико-социологического наблюдения:

Самые яростные противоречия возникают либо между двумя близкими сектами, либо при борьбе двух фракций одной и той же секты.

Люди старшего поколения еще помнят, как во время ожесточенной вражды между китайскими и советскими коммунистами китайцы не переставая повторяли, что у них "из десяти пальцев девять - общие".

Тем не менее иногда бывают исключения. Рассмотрим, например, схему поиска вглубь на дереве.

```
int search (ELEMENT x)
ELEMENT y; int result;
if (good(x)){
  return id(x)}
  else for(int i=0; i<100; i++)
     {y=get_successor(x,i);
      result=search(y);
      if (result>0) return result;
     }
  return 0;
}
```

14.4.1.

Здесь *рекурсии* вместе с *циклом* задают *обход дерева* возможностей, и гибельного размножения рекурсивных вызовов не происходит. Причина этого исключения в том, что *цикл* в данной программе - всего лишь *подпорка* для *рекурсии*. В обычном программировании нет функционалов типа *mapcar* языка *LISP*, применяющих свой первый *аргумент* ко всем членам второго.

Структурное *программирование* основано на предположении о локальности действий и условий, поэтому для него в особенности органично подходит иерархическое *разбиение* задачи на подзадачи - так называемое ***нисходящее планирование***.

При *нисходящем планировании* начинают с предположения о том, что поставленная задача решена. Затем пытаются реализовать решение общей задачи, выделяя те его блоки, которые оказались неэлементарными, в конструкции следующего уровня. В итоге мы получаем структуру взаимоотношений между конструкциями второго уровня и спецификации на каждую из этих конструкций. Если мы реализуем каждую из конструкций в точном соответствии с полученной спецификацией, то получится решение общей задачи.

В *планировании процессов* деятельности каждая из подзадач является процессом, чаще всего реализуемым отдельным человеком или группой людей, в конструировании технических систем каждая из подзадач дает блок конструкции, в программировании она дает *модуль* (в наиболее часто встречающихся достаточно простых случаях - подпрограмму).

Нисходящее *программирование* - *конкретизация нисходящего проектирования* для нужд программирования. Оно обладает следующими преимуществами.

- Нисходящее программирование породило хорошо согласованное с ним нисходящее построение структур данных, а нисходящее построение структур данных является простейшим частным случаем глубокой идеальной концепции - абстракции данных. Согласно парадоксу изобретателя, даже простейшие частные случаи высокоуровневых концепций при удачном применении улучшают все характеристики умственных конструкций (в частности, программ).
- При таком проектировании на каждом уровне можно ограничиваться одной моделью вычислений, в частности операционной, на которую ориентировано структурное программирование.
- Возможно раннее программирование, когда прототип программы, работающий хотя бы в условиях грубой эмуляции будущих решений низкого уровня, позволяет продемонстрировать, как будет работать полная программа, и улучшить ее пользовательские характеристики.

Ни один конкретный способ анализа либо синтеза не является универсальным. Нисходящее *программирование*, конечно же, тоже таковым не является. Причины тому, в частности, следующие.

- Невозможность при нисходящем структурном программировании увидеть тождественность процедур, работающих на разных ветвях декомпозиции, а тем более унифицировать несколько формально различных процедур на разных ветвях в одну общую.
- Недостаточный учет особенностей, которые могут возникнуть после конкретной реализации запланированных блоков программы, что довольно часто вызывает необходимость полной

перепланировки системы после реализации ее блоков, а вслед за такой перепланировкой реализованные для других спецификаций блоки часто оказываются отнюдь не лучшими из возможных.

- Недостаточный перенос опыта и наработок из одного проекта в другой.
- В нынешних методиках *нисходящего проектирования* ко всему перечисленному добавляется еще и навязывание последовательного стиля мышления, что мешает воспользоваться преимуществами нетрадиционных машин.

Это указывает на необходимость средств, позволяющих поднимать уровень понятий. Поэтому в разработке структурных программ применяется также подход, получивший название ***восходящего программирования***. К примеру, когда строят библиотеку, занимаются обобщением задачи. Части, выделяемые в виде библиотечных средств, выбираются таким образом, чтобы они были применимы в различных контекстах.

При создании глобального контекста могут применяться всевозможные стили. Например, модули могут программироваться в стиле автоматного программирования. Если разделение стилей по модулям, пакетам и другим автономным структурным единицам проведено строго, то эклектического их смешения не происходит.

Реальный проект - это смесь нисходящего и восходящего подходов:

- сначала сверху вниз для выяснения крупных строительных блоков;
- затем попытка движения снизу вверх, чтобы спроецировать понятия, оформившиеся ранее, на абстрактные структуры, допускающие адекватную реализацию;
- далее проверка соответствия, углубление нисходящей декомпозиции и обобщение понятий, выделенных восходящими приемами.

Как в нисходящем, так и в восходящем подходе важно обеспечить согласование потоков управления и потоков данных. Основным инструментом здесь может быть взгляд на программу со стороны *сетей данных*. Поскольку *цикл* появляется как реализация послойного движения по сети, а *массив* - как *представление* слоя сети, то видно, что на самом деле основной информационной структурой *цикла* является слой сети. Например, в *цикле*, реализующем числа Фибоначчи, - это структура из fib1 и fib2. Структура, представляющая очередной слой сети, называется **реальным параметром** (в отличие от формального параметра, диктуемого языком программирования и часто являющегося *подпоркой*) *цикла*. Реальный *параметр* мы будем называть просто параметром.

Далее, поскольку *цикл* возникает при повторении однородных действий, а действия диктуются свойствами реальных объектов программы, в *цикле* должно быть общее свойство, зависящее от параметра и называемое **инвариантом цикла**. Например, в *цикле* сортировки - это соотношение между отсортированными и неотсортированными фрагментами массива.

При формулировке инварианта *цикла* мы практически с неизбежностью наталкиваемся на парадокс изобретателя, который заставляет нас вводить призрачные значения, например *дерево* разбиения массива на отсортированные подмассивы в эффективных алгоритмах сортировки.

Методика циклического и рекурсивного структурного программирования с использованием инвариантов и недетерминированности (но без явного упоминания *сетей данных*) прекрасно изложена в учебных пособиях Алагича, Арбиба и Гриса. Настольной книгой программиста должна служить также книга Кормена и др. (учебник *MIT*) , в которой можно найти множество хороших примеров того, как находить правильные программные решения и со вкусом использовать *подпорки*.

Сделаем еще один шаг. Поскольку *присваивание* с точки зрения решаемой задачи лишь средство задать элементы нового слоя сети, есть смысл считать, что во время всей итерации *цикла параметр* фиксирован, его изменение происходит лишь в промежутке между двумя итерациями. Таким образом, **для согласования потоков данных и потоков управления необходимо сосредоточить все присваивания реальным сущностям в одном месте: либо в конце, либо в начале очередной итерации**. Более того, в принципе (и даже не совсем теоретическом, но плохо поддерживаемом современными системами программирования) присваивания можно было

бы вообще изгнать, сделав переход старого значения реального параметра *цикла* к новому неявным, так же, как это сделано для формального, скажем, в языке *Pascal*.

Призраки и инварианты необходимо осознавать до начала написания текста работающей программы. Если Вы оставите это на потом, то наверняка спутаете подпорки, которые займут у Вас основную часть мысленных ресурсов на завершающем этапе реализации, с сущностями. Более того, восстановить обоснование по тексту уже готовой программы, как ни парадоксально, обычно труднее, чем построить его заранее.

Переходы и выдаваемые значения

В общее употребление структурное *программирование* вошло после популяризировавшей его работы *Э. Дейкстры*, в которой, к сожалению, не было даже намека на его ограничения. Ограничения структурного программирования вытекают как из самой его сути, так и из *теоремы Бема-Джакопини*. Применение *структурных переходов*, которые ввел в практику и теорию Д. Кнут (откопавший оригинальную работу Бема-Джакопини и четко выделивший ограничения дейкстровского структурного подхода[13]), избавляет от многих недостатков, присущих методике *Дейкстры*. ***Структурные переходы*** - переходы лишь вперед и на более высокий уровень структурной иерархии управления, ни в каком случае не выводящие нас за пределы данного модуля.

```
/* Примеры структурных goto.*/
. . .
do {y = f( x ,y)};
  if (y>0) break;
  x=g(x,y);
}while (x>0);
. . .
{. . .
  {
   if (All_Done)goto Success;
  }
. . .
Success: Hurra;}
```

14.5.1.

Структурные переходы в настоящее время также включаются в общераспространенные языки программирования. Их использование понемногу проникает и в учебные курсы.

Структурные переходы являются паллиативом. Они возникли из-за необходимости выразить мысль о том, что успех либо неудача глобального процесса может выявиться внутри одной из решаемых подзадач, и дальнейшая работа и над текущей задачей, и над всей последовательностью вложенных подзадач становится просто бессмысленной. В этом случае нужны даже не переходы, а *операторы* завершения. Но они во многих распространенных языках действуют лишь на один уровень иерархии вверх, а даже теоретически этого недостаточно. Стоит заметить, что между идеей и ее корректной реализацией часто проходят долгие годы. Ныне в общераспространенном языке *Java* завершители наконец-то более или менее корректно реализованы.

Есть одно ограничение *структурных переходов*, известное с 80-х гг. XX века, по крайней мере один раз достоверно повредившее создателям отечественной серии машин Эльбрус, в которых на аппаратном уровне поддерживалось структурное и функциональное *программирование*.

Структурные переходы (в том числе и завершители) некорректны, когда они выводят нас из функции-параметра вызова другой функции. Ирония в том, что абсолютно четкая и полная реализация завершителей еще до осознания необходимости данного средства и тем более задолго до осознания пределов его применимости была проделана именно там, где они в общем случае некорректны (в языке *LISP*). В нем, как мы видели, процедуры являются полноправными значениями, могут быть параметрами и результатами других процедур. Самый очевидный случай некорректности (правда, вылавливаемый системой обнаружения динамических ошибок *Common Lisp*),

когда мы внутри созданной процедуры завершаем блок, существовавший в момент создания процедуры, но переставший существовать в тот момент, когда ее вызвали. Наверняка многие наталкивались на непонятное поведение программ с завершителями, но в соответствии с общераспространенным "позитивным" мышлением не обращали внимания на данный феномен.

Есть еще одна, на первый взгляд исключительно привлекательная, возможность. В обычных языках программирования часто раздражает то, что *значение*, выработанное один раз, не удается сразу же использовать в следующем операторе, и, более того, по какой-то очевидной глупости совершенно безвредное и использованное еще в *Algol* 60 понятие условного выражения, подобного

```
if x=0 then sin(y) else tan(y)
```

оказалось выброшено из некоторых распространенных языков (может быть, потому, что в данном случае часть else обязательна). В том же языке *LISP*, где была (достаточно уникальный случай в программировании) создана четкая и достаточно *замкнутая система* концепций (ни одной неувязки, которую можно было диагностировать при тогдашнем состоянии теории и практики, найти в нем не удалось), каждый оператор выдает *значение*, которое может быть использовано объемлющим оператором.

Эту концепцию попытались перенести на язык традиционного типа в языке *Алгол* 68. На первый взгляд получилось очень красиво. Например, оператор

```
if x>0 then a1 else a2 fi [if y>0 then j else i fi]:=
if z>0 then sin(u) else tan(u) fi
```

намного компактней и красивей последовательности условных операторов, которую придется написать в *Pascal* или C++. Но концепция вырабатываемого значения оказалась в *концептуальном противоречии* и с оператором присваивания, и с совместными вычислениями. Например, согласно семантике Алгола 68, следующая *запись*

```
(x:=3, y:=x+y, z:=x+y)
```

является блоком программы, в котором все три присваивания исполняются совместно. Но очевидно, что в данном случае результат вычислений неоднозначен. Одновременно, поскольку значения не забываются, эта же конструкция является изображением массива из трех элементов либо записи с тремя полями.

Следовательно, если распространить систему выдаваемых значений на все *операторы* структурного программирования, то появляются формально допустимые и соблазнительные для хакерских трюков неустойчивые выражения. *Выражение*, изменяющее свой *контекст* (в частности, *присваивание*), выдавать *значение* не должно, чтобы не было соблазна поместить его в окружение, где оно будет вызывать трудно диагностируемую неоднозначность.

Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин, или компьютеров, ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых можно преобразовать в понятные компьютеру инструкции, и любая *вычислительная задача* будет решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специализированные языки программирования, созданные для разработки программ, предназначенных для решения вычислительных задач. Примерами таких языков могут служить FOCAL (FOrmula CALculator) и *FORTRAN* (FORmula TRANslator).

Основой такой методологии разработки программ являлась процедурная, или *алгоритмическая*, организация структуры программного кода. Это было настолько естественно для решения вычислительных задач, что целесообразность такого подхода ни у кого не вызывала сомнений. Исходным в данной методологии было понятие алгоритма. *Алгоритм* - это способ решения вычислительных и других задач, точно описывающий определенную последовательность действий, которые необходимо выполнить для достижения заданной цели. Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или системы линейных уравнений.

При увеличении объемов программ для упрощения их разработки появилась необходимость разбивать большие задачи на подзадачи. В языках программирования возникло и закрепилось новое понятие процедуры. Использование процедур позволило разбивать большие задачи на подзадачи и таким образом упростило написание больших программ. Кроме того, процедурный подход позволил уменьшить объем программного кода за счет написания часто используемых кусков кода в виде процедур и их применения в различных частях программы.

Как и *алгоритм*, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая также получила название процедуры. Например, на языке *Pascal* описание процедуры выглядит следующим образом:

```
Procedure printGreeting(name: String)
Begin
  Write("Hello, ");
  WriteLn(name);
End;
```

Назначение данной процедуры - вывести на экран приветствие Hello, Name, где Name передается в процедуру в качестве входного параметра.

Со временем вычислительные задачи становились все сложнее, а значит, и решающие их программы увеличивались в размерах. Их разработка превратилась в серьезную проблему. Когда *программа* становится все больше, ее приходится разделять на все более мелкие фрагменты. Основой для такого разбиения как раз и стала процедурная *декомпозиция*, при которой отдельные части программы, или модули, представляли собой совокупность процедур для решения одной или нескольких задач. Одна из основных особенностей процедурного программирования заключается в том, что оно позволило создавать библиотеки подпрограмм (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта. При процедурном подходе для визуального представления алгоритма выполнения программы применяется так называемая ***блок-схема*** . Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90. Пример *блок-схемы* изображен на рисунке.

Рис. 2.1. Пример блок-схемы.

Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов *по* программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода goto может заметно осложнить понимание кода. Такие запутанные программы сравнивали с порцией спагетти (*bowl* of spaghetti), имея в виду многочисленные переходы от одного фрагмента программы к другому, или, что еще хуже, возврат от конечных операторов программы к начальным. Ситуация казалась настолько драматичной, что многие предлагали исключить оператор goto из языков программирования. Именно с этого времени отсутствие безусловных переходов стали считать хорошим стилем программирования.

Дальнейшее увеличение программных систем способствовало формированию новой точки зрения на процесс разработки программ и написания программных кодов, которая получила название методологии структурного программирования. Ее основой является процедурная *декомпозиция предметной области* решаемой задачи и организация отдельных модулей в виде совокупности процедур. В рамках этой методологии получило развитие *нисходящее проектирование* программ, или проектирование "сверху вниз". Пик популярности идей структурного программирования приходится на конец 70-х - начало 80-х годов.

В этот период основным показателем сложности разработки программы считался ее размер. Вполне серьезно обсуждались такие оценки сложности программ, как количество строк программного кода. Правда, при этом делались некоторые предположения относительно синтаксиса самих строк, которые должны были соответствовать определенным требованиям. Например, каждая строка кода должна была содержать не более одного оператора. Общая трудоемкость разработки программ оценивалась специальной единицей измерения - "человеко-месяц", или "человеко-год". А профессионализм программиста напрямую связывался с количеством строк программного кода, который он мог написать и отладить в течение, скажем, месяца.

Методология объектно-ориентированного программирования

Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою *очередь*, потребовало дополнительных ресурсов для организации их согласованной работы. В процессе разработки приложений заказчик зачастую изменял *функциональные требования*, что еще более усложняло процесс создания программного обеспечения.

Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. В эпоху "больших машин" основными потребителями программного обеспечения были такие крупные заказчики, как большие производственные предприятия, финансовые компании, государственные учреждения. *Стоимость* таких вычислительных устройств для небольших предприятий и организаций была слишком высока.

Позже появились персональные компьютеры, которые имели гораздо меньшую *стоимость* и были значительно компактнее. Это позволило широко использовать их в малом и среднем бизнесе. Основными задачами в этой области являются обработка данных и манипулирование ими, поэтому вычислительные и расчетно-алгоритмические задачи с появлением персональных компьютеров отошли на второй план. Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало *объектно-ориентированное программирование* (*ООП*).

После составления технического задания начинается этап проектирования, или дизайна, будущей системы. *Объектно-ориентированный подход* к проектированию основан на представлении *предметной области* задачи в виде *множества* моделей для независимой от языка разработки программной системы на основе ее прагматики.

Последний термин нуждается в пояснении. *Прагматика* определяется целью разработки программной системы, например, обслуживание клиентов банка, управление работой аэропорта,

обслуживание чемпионата мира *по* футболу и т.п. В формулировке цели участвуют предметы и понятия реального мира, имеющие *отношение* к создаваемой системе (см. рисунок 2.2). При объектно-ориентированном подходе эти предметы и понятия заменяются моделями, т.е. определенными формальными конструкциями.

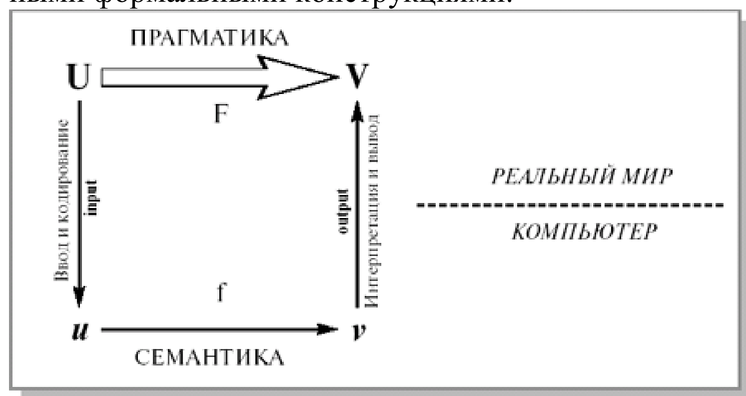


Рис. 2.2. Семантика (смысл программы с точки зрения выполняющего ее компьютера) и прагматика (смысл программы с точки зрения ее пользователей)

Модель содержит не все признаки и свойства представляемого ею предмета или понятия, а только те, которые существенны для разрабатываемой программной системы. Таким образом, модель "беднее", а следовательно, проще представляемого ею предмета или понятия.

Простота модели *по* отношению к реальному предмету позволяет сделать ее формальной. Благодаря такому характеру моделей при разработке можно четко выделить все зависимости и *операции* над ними в создаваемой программной системе. Это упрощает как разработку и изучение (*анализ*) моделей, так и их реализацию на компьютере.

Объектно-ориентированный подход обладает такими преимуществами, как:

- уменьшение сложности программного обеспечения;
- повышение надежности программного обеспечения;
- обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Более детально преимущества и недостатки *объектно-ориентированного программирования* будут рассмотрены в конце лекции, так как для их понимания необходимо *знание* основных понятий и положений *ООП*.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. *ООП* является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

Объекты

По определению будем называть ***объектом*** понятие, абстракцию или любой предмет с четко очерченными границами, имеющий смысл в контексте рассматриваемой прикладной проблемы. Введение *объектов* преследует две цели:

- понимание прикладной задачи (проблемы);
- введение основы для реализации на компьютере.

Примеры *объектов*: форточка, Банк "Империал", Петр Сидоров, сберкнижка и т.д.

Каждый *объект* имеет определенное время жизни. В процессе выполнения программы, или функционирования какой-либо реальной системы, могут создаваться новые *объекты* и уничтожаться уже существующие.

Гради Буч дает следующее *определение объекта*:

Объект - это мыслимая или реальная сущность, обладающая характерным *поведением* и отличительными характеристиками и являющаяся важной в предметной области.

Каждый *объект* имеет *состояние*, обладает четко определенным *поведением* и уникальной идентичностью.

Состояние

Рассмотрим пример. Любой человек может находиться в некотором положении (*состоянии*): стоять, сидеть, лежать, и - в то же время совершать какие-либо действия.

Например, человек может прыгать, если он стоит, и не может - если он лежит, для этого ему потребуется сначала встать. Также в *объектно-ориентированном программировании состояние объекта* может определяться наличием или отсутствием связей между моделируемым *объектом* и другими *объектами*. Более подробно все возможные связи между *объектами* будут рассмотрены в разделе "Типы отношений между *классами* ".

Например, если у человека есть удочка (у него есть связь с *объектом* "Удочка"), он может ловить рыбу, а если удочки нет, то такое действие невозможно. Из этих примеров видно, что набор действий, которые может совершать человек, зависит от параметров *объекта*, его моделирующего.

Для рассмотренных выше примеров такими характеристиками, или атрибутами, *объекта* "Человек" являются:

- текущее положение человека (стоит, сидит, лежит);
- наличие удочки (есть или нет).

В конкретной задаче могут появиться и другие свойства, например, физическое *состояние*, здоровье (больной человек обычно не прыгает).

Состояние (state) - совокупный результат *поведения объекта*: одно из стабильных условий, в которых *объект* может существовать, охарактеризованных количественно; в любой момент времени *состояние объекта* включает в себя перечень (обычно статический) свойств *объекта* и текущие значения (обычно динамические) этих свойств.

Поведение

Для каждого *объекта* существует определенный набор действий, которые с ним можно произвести. Например, возможные действия с некоторым файлом операционной системы ПК:

- создать;
- открыть;
- читать из файла;
- писать в файл;
- закрыть;
- удалить.

Результат выполнения действий зависит от *состояния объекта* на момент совершения действия, т.е. нельзя, например, удалить файл, если он открыт кем-либо (заблокирован). В то же время действия могут менять внутреннее *состояние объекта* - при открытии или закрытии файла свойство "открыт" принимает значения "да" или "нет", соответственно.

Программа, написанная с использованием *ООП*, обычно состоит из множества *объектов*, и все эти *объекты* взаимодействуют между собой. Обычно говорят, что взаимодействие между *объектами* в программе происходит посредством передачи сообщений между ними.

В терминологии объектно-ориентированного подхода понятия "действие", "сообщение" и "метод" являются синонимами. Т.е. выражения "выполнить действие над *объектом* ", "вызвать метод *объекта* " и "послать сообщение *объекту* для выполнения какого-либо действия" эквивалентны. Последняя фраза появилась из следующей *модели. Программу*, построенную по технологии *ООП*, можно представить себе как виртуальное пространство, заполненное *объектами*, которые условно "живут" некоторой жизнью. Их активность проявляется в том, что они вызывают друг у друга методы, или посылают друг другу сообщения. Внешний интерфейс *объекта*, или набор его методов,- это описание того, какие сообщения он может принимать.

Поведение (behavior) - действия и реакции *объекта*, выраженные в терминах передачи сообщений и изменения *состояния* ; видимая извне и воспроизводимая активность *объекта* .

Уникальность

Уникальность - это то, что отличает *объект* от других *объектов*. Например, у вас может быть несколько одинаковых монет. Даже если абсолютно все их свойства (атрибуты) одинаковы (год выпуска, номинал и т.д.) и при этом вы можете использовать их независимо друг от друга, они по-прежнему остаются разными монетами.

В машинном представлении под параметром *уникальности объекта* чаще всего понимается адрес размещения *объекта* в памяти.

Identity (*уникальность*) *объекта* состоит в том, что всегда можно определить, указывают две ссылки на один и тот же *объект* или на разные *объекты*. При этом два *объекта* могут во всем быть похожими, их образ в памяти может представляться одинаковыми последовательностями байтов, но, тем не менее, их Identity может быть различна.

Наиболее распространенной ошибкой является понимание *уникальности* как имени ссылки на *объект*. Это неверно, т.к. на один *объект* может указывать несколько ссылок, и ссылки могут менять свои значения (ссылаться на другие *объекты*).

Итак, ***уникальность*** (identity) - свойство *объекта*; то, что отличает его от других *объектов* .

Классы

Все монеты из предыдущего примера принадлежат одному и тому же *классу объектов* (именно с этим связана их одинаковость). Номинальная *стоимость* монеты, металл, из которого она изготовлена, форма - это атрибуты *класса*. Совокупность атрибутов и их значений характеризует *объект*. Наряду с термином "*атрибут*" часто используют термины "свойство" и "*поле*", которые в *объектно-ориентированном программировании* являются синонимами.

Все *объекты* одного и того же *класса* описываются одинаковыми наборами атрибутов. Однако *объединение объектов* в *классы* определяется не наборами атрибутов, а семантикой. Так, например, *объекты* "конюшня" и "лошадь" могут иметь одинаковые атрибуты: цена и возраст. При этом они могут относиться к одному *классу*, если рассматриваются в задаче просто как *товар*, либо к разным *классам*, если в рамках поставленной задачи будут использоваться *по*-разному, т.е. над ними будут совершаться различные действия.

Объединение объектов в *классы* позволяет рассмотреть задачу в более общей постановке. *Класс* имеет имя (например, "лошадь"), которое относится ко всем *объектам* этого *класса*. Кроме того, в *классе* вводятся имена атрибутов, которые определены для *объектов*. В этом смысле описание *класса* аналогично описанию типа структуры или записи (*record*), широко применяющихся в процедурном программировании; при этом каждый *объект* имеет тот же смысл, что и экземпляр структуры (*переменная* или константа соответствующего типа).

Формально ***класс*** - это *шаблон поведения объектов* определенного типа с заданными параметрами, определяющими *состояние*. Все экземпляры одного *класса* (*объекты*, порожденные от одного *класса*) имеют один и тот же набор свойств и общее *поведение*, то есть одинаково реагируют на одинаковые сообщения.

Apple
String sort
float weight
setSort()
String getSort()
setWeight()
float getWeight()

В соответствии с ***UML*** (*Unified* Modelling *Language* - унифицированный *язык моделирования*), *класс* имеет следующее графическое *представление*.

Класс изображается в виде прямоугольника, состоящего из трех частей. В верхней части помещается название *класса*, в средней - свойства *объектов класса*, в нижней - действия, которые можно выполнять с *объектами* данного *класса* (методы).

Каждый *класс* также может иметь специальные методы, которые автоматически вызываются при создании и уничтожении *объектов* этого *класса*:

- ***конструктор*** (constructor) - выполняется при создании *объектов* ;
- ***деструктор*** (*destructor*) - выполняется при уничтожении *объектов*.

Обычно *конструктор* и *деструктор* имеют специальный *синтаксис*, который может отличаться от синтаксиса, используемого для написания обычных методов *класса*.

Инкапсуляция

Инкапсуляция (encapsulation) - это сокрытие реализации *класса* и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам какого-либо *класса* из методов других *классов*. Для доступа к свойствам *класса* принято задействовать специальные методы этого *класса* для получения и изменения его свойств.

Внутри *объекта* данные и методы могут обладать различной степенью открытости (или доступности). Степени доступности, принятые в языке Java, подробно будут рассмотрены в лекции 6. Они позволяют более тонко управлять свойством *инкапсуляции*.

Открытые члены *класса* составляют внешний интерфейс *объекта*. Это та функциональность, которая доступна другим *классам*. Закрытыми обычно объявляются все свойства *класса*, а также вспомогательные методы, которые являются деталями реализации и от которых не должны зависеть другие части системы.

Благодаря сокрытию реализации за внешним интерфейсом *класса* можно менять внутреннюю логику отдельного *класса*, не меняя код остальных компонентов системы. Это свойство называется ***модульность***.

Обеспечение доступа к свойствам *класса* только через его методы также дает ряд преимуществ. Во-первых, так гораздо проще контролировать корректные значения полей, ведь прямое обращение к свойствам отслеживать невозможно, а значит, им могут присвоить некорректные значения.

Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, потребуется изменить лишь ряд методов одного *класса*, а не вводить эту функциональность во все части системы.

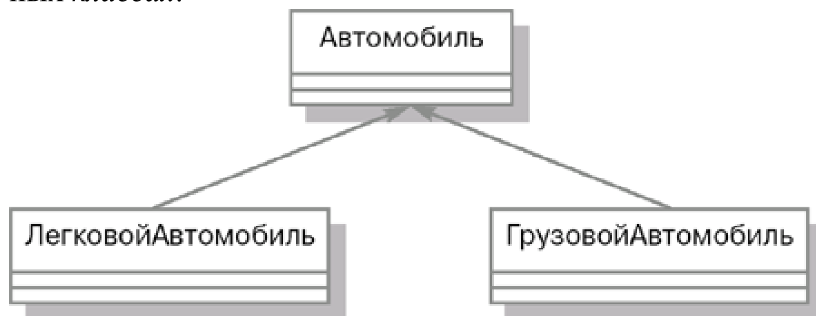
Наконец, программный код, написанный с использованием данного принципа, легче отлаживать. Для того чтобы узнать, кто и когда изменил свойство интересующего нас *объекта*, достаточно добавить вывод отладочной информации в тот метод *объекта*, посредством которого осуществляется доступ к свойству этого *объекта*. При использовании прямого доступа к свойствам *объектов* программисту пришлось бы добавлять вывод отладочной информации во все участки кода, где используется интересующий нас *объект*.

Наследование

Наследование (inheritance) - это отношение между *классами*, при котором *класс* использует структуру или *поведение* другого *класса* (одиночное *наследование*), или других (множественное *наследование*) *классов*. *Наследование* вводит иерархию "общее/частное", в которой ***подкласс*** наследует от одного или нескольких более общих ***суперклассов***. *Подклассы* обычно дополняют или переопределяют унаследованную структуру и *поведение*.

В качестве примера можно рассмотреть задачу, в которой необходимо реализовать *классы* "Легковой автомобиль" и "Грузовой автомобиль". Очевидно, эти два *класса* имеют общую функциональность. Так, оба они имеют 4 колеса, двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, независимо от того, грузовой он или легковой, 5- или 12-местный. Разумно вынести эти общие свойства и функциональность в отдельный *класс*, например, "Автомобиль" и наследовать от него *классы* "Легковой автомобиль" и "Гру-

зовой автомобиль", чтобы избежать повторного написания одного и того же кода в разных *классах*.



Отношение обобщения обозначается сплошной линией с треугольной стрелкой на конце. Стрелка указывает на более общий *класс* (***класс-предок*** или *суперкласс*), а ее отсутствие - на более специальный *класс* (***класс-потомок*** или *подкласс*).

Использование *наследования* способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

В рассмотренном примере применено одиночное *наследование*. Некоторый *класс* также может наследовать свойства и *поведение* сразу нескольких *классов*. Наиболее популярным примером применения множественного *наследования* является проектирование системы учета товаров в зоомагазине.

Все животные в зоомагазине являются наследниками *класса* "Животное", а также наследниками *класса* "Товар". Т.е. все они имеют возраст, нуждаются в пище и воде и в то же время имеют цену и могут быть проданы.

Множественное *наследование* на диаграмме изображается точно так же, как одиночное, за исключением того, что линии *наследования* соединяют *класс-потомок* сразу с несколькими *суперклассами*.

Не все объектно-ориентированные языки программирования содержат языковые конструкции для описания множественного *наследования*.

В языке Java множественное *наследование* имеет ограниченную поддержку через интерфейсы и будет рассмотрено в лекции 8.

Полиморфизм

Полиморфизм является одним из фундаментальных понятий в *объектно-ориентированном программировании* наряду с *наследованием* и *инкапсуляцией*. Слово " *полиморфизм* " греческого происхождения и означает "имеющий много форм". Чтобы понять, что оно означает применительно к *объектно-ориентированному программированию*, рассмотрим пример.

Предположим, мы хотим создать векторный графический редактор, в котором нам нужно описать в виде *классов* набор графических примитивов - Point, Line, Circle, Box и т.д. У каждого из этих *классов* определим метод draw для отображения соответствующего примитива на экране.

Очевидно, придется написать код, который при необходимости отобразить рисунок, будет последовательно перебирать все примитивы, на момент отрисовки находящиеся на экране, и вызывать метод draw у каждого из них. Человек, не знакомый с *полиморфизмом*, вероятнее всего, создаст несколько массивов (отдельный массив для каждого типа примитивов) и напишет код, который последовательно переберет элементы из каждого массива и вызовет у каждого элемента метод draw. В результате получится примерно следующий код:

```
...
//создание пустого массива, который может
// содержать объекты Point с максимальным
// объемом 1000
Point[] p = new Point[1000];
```

```
Line[] l = new Line[1000];
Circle[] c = new Circle[1000];
Box[] b = new Box[1000];
...
// предположим, в этом месте происходит
// заполнение всех массивов соответствующими
// объектами
...
for(int i = 0; i < p.length;i++) {
//цикл с перебором всех ячеек массива.
  //вызов метода draw() в случае,
  // если ячейка не пустая.
  if(p[i]!=null) p[i].draw();
}

for(int i = 0; i < l.length;i++) {
  if(l[i]!=null) l[i].draw();
}

for(int i = 0; i < c.length;i++) {
  if(c[i]!=null) c[i].draw();
}

for(int i = 0; i < b.length;i++) {
  if(b[i]!=null) b[i].draw();
}
...
```

Недостатком написанного выше кода является дублирование практически идентичного кода для отображения каждого типа примитивов. Также неудобно то, что при дальнейшей модернизации нашего графического редактора и добавлении возможности рисовать новые типы графических примитивов, например Text, Star и т.д., при таком подходе придется менять существующий код и добавлять в него определения новых массивов, а также обработку содержащихся в них элементов.

Используя *полиморфизм*, мы можем значительно упростить реализацию подобной функциональности. Прежде всего, создадим общий родительский *класс* для всех наших *классов*. Пусть таким *классом* будет Point. В результате получим иерархию *классов*, которая изображена на рисунке 2.3.

У каждого из дочерних *классов* метод draw переопределен таким образом, чтобы отображать экземпляры каждого *класса* соответствующим образом.

Для описанной выше иерархии классов, используя *полиморфизм*, можно написать следующий код:

```
...
Point p[] = new Point[1000];
p[0] = new Circle();
p[1] = new Point();
p[2] = new Box();
p[3] = new Line();
...
for(int i = 0; i < p.length;i++) {
  if(p[i]!=null) p[i].draw();
```

}
...

В описанном выше примере массив p[] может содержать любые *объекты*, порожденные от наследников *класса* Point. При вызове какого-либо метода у любого из элементов этого массива будет выполнен метод того *объекта*, который содержится в ячейке массива. Например, если в ячейке p[0] находится *объект* Circle, то при вызове метода draw следующим образом:

p[0].draw()

нарисуется круг, а не точка.

Полиморфизм (*polymorphism*) - положение *теории типов*, согласно которому имена (например, переменных) могут обозначать *объекты* разных (но имеющих общего родителя) *классов*. Следовательно, любой *объект*, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

В процедурном программировании тоже существует понятие *полиморфизма*, которое отличается от рассмотренного механизма в *ООП*. Процедурный *полиморфизм* предполагает возможность создания нескольких процедур или функций с одним и тем же именем, но разным количеством или различными типами передаваемых параметров. Такие одноименные функции называются ***перегруженными***, а само явление - ***перегрузкой*** (overloading). *Перегрузка* функций существует и в *ООП* и называется *перегрузкой* методов.

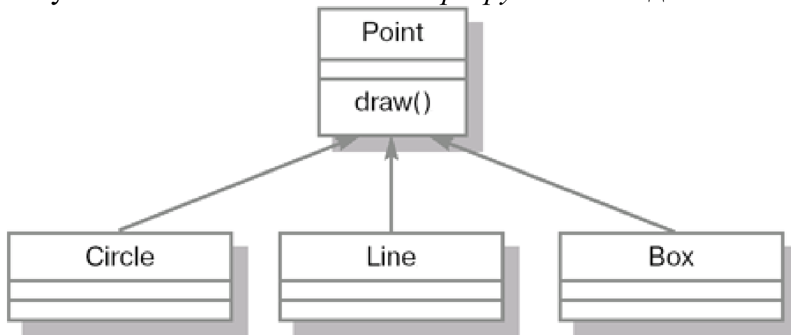


Рис. 2.3. Пример иерархии классов.

Примером использования *перегрузки* методов в языке Java может служить *класс* PrintWriter, который применяется, в частности, для вывода сообщений на консоль. Этот *класс* имеет множество методов println, которые различаются типами и/или количеством входных параметров. Вот лишь несколько из них:

```
void println()
  // переход на новую строку
void println(boolean x)
  // выводит значение булевской
  // переменной (true или false)
void println(String x)
  // выводит строку - значение
  // текстового параметра.
```

Определенные сложности возникают при вызове ***перегруженных методов***. В Java существуют специальные правила, которые позволяют решать эту проблему. Они будут рассмотрены в соответствующей лекции.

Типы отношений между классами

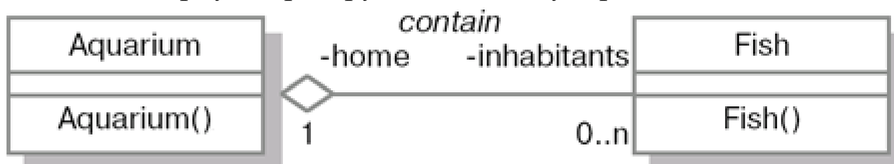
Как правило, любая *программа*, написанная на объектно-ориентированном языке, представляет собой некоторый набор связанных между собой *классов*. Можно провести аналогию между написанием программы и строительством дома. Подобно тому, как стена складывается из кирпичей, компьютерная *программа* с использованием *ООП* строится из *классов*. Причем эти *классы* должны иметь *представление* друг о друге, для того чтобы сообща выполнять поставленную задачу.

Возможны следующие связи между *классами* в рамках объектной модели (приводятся лишь наиболее простые и часто используемые виды связей, подробное их рассмотрение выходит за рамки этой ознакомительной лекции):

- *агрегация (Aggregation);*
- *ассоциация (Association);*
- *наследование (Inheritance);*
- *метаклассы* (*Metaclass*).

Агрегация

Отношение между *классами* типа "содержит" (contain) или "состоит из" называется агрегацией, или включением. Например, если аквариум наполнен водой и в нем плавают рыбки, то можно сказать, что аквариум агрегирует в себе воду и рыбок.



Такое отношение включения, или агрегации (aggregation), изображается линией с ромбиком на стороне того *класса*, который выступает в качестве владельца, или контейнера. Необязательное название отношения записывается посередине линии.

В нашем примере отношение contain является двунаправленным.

Объект класса Aquarium содержит несколько *объектов* Fish. В то же время каждая рыбка "знает", в каком именно аквариуме она живет. Каждый *класс* имеет свою роль в агрегации, которая указывает, какое место занимает *класс* в данном отношении. *Имя роли* не является обязательным элементом обозначений и может отсутствовать на диаграмме.

В примере можно видеть роль home *класса* Aquarium (аквариум является домом для рыбок), а также роль inhabitants *класса* Fish (рыбки являются обитателями аквариума). Название роли обычно совпадает с названием соответствующего поля в *классе*. Изображение такого поля на диаграмме излишне, если уже указано *имя роли*. Т.е. в данном случае *класс* Aquarium будет иметь свойство (поле) inhabitants, а *класс* Fish - свойство home.

Число *объектов*, участвующих в отношении, записывается рядом с именем роли. Запись " 0..n " означает "от нуля до бесконечности". Приняты также обозначения:

- " 1..n " - от единицы до бесконечности;
- " 0 " - ноль;
- " 1 " - один;
- " n " - фиксированное количество;
- " 0..1 " - ноль или один.

Код, описывающий рассмотренную модель и явление агрегации, может выглядеть, например, следующим образом:

```
// определение класса Fish
public class Fish {
  // определения поля home
  // (ссылка на объект Aquarium)
  private Aquarium home;

  public Fish() {
  }
}
// определение класса Aquarium
public class Aquarium {
  // определения поля inhabitants
```

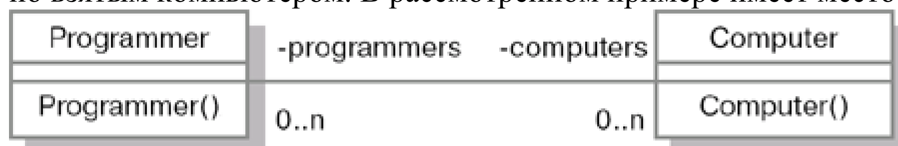
```
  // (массив ссылок на объекты Fish)
  private Fish inhabitants[];
  public Aquarium() {
  }
}
```

Ассоциация

Если *объекты* одного *класса* ссылаются на один или более *объектов* другого *класса*, но ни в ту, ни в другую сторону отношение между *объектами* не носит характера "владения", или контейнеризации, такое отношение называют ***ассоциацией*** (association).

Отношение *ассоциации* изображается так же, как и отношение агрегации, но линия, связывающая *классы*,- простая, без ромбика.

В качестве примера можно рассмотреть программиста и его компьютер. Между этими двумя *объектами* нет агрегации, но существует четкая взаимосвязь. Так, всегда можно установить, за какими компьютерами работает какой-либо программист, а также какие люди пользуются отдельно взятым компьютером. В рассмотренном примере имеет место *ассоциация* "многие-ко-многим".



В данном случае между экземплярами *классов* Programmer и Computer в обе стороны используется отношение " 0..n ", т.к. программист, в принципе, может не работать с компьютером (если он теоретик или на пенсии). В свою очередь, компьютер может никем не использоваться (если он новый и еще не установлен).

Код, соответствующий рассмотренному примеру, будет, например, следующим:

```
public class Programmer {
  private Computer computers[];
  public Programmer() {
  }
}
public class Computer {
  private Programmer programmers[];
  public Computer() {
  }
}
```

Наследование

Наследование является важным случаем отношений между двумя или более *классами*. Подробно оно рассматривалось выше.

Метаклассы

Итак, любой *объект* имеет структуру, состоящую из полей и методов. *Объекты*, имеющие одинаковую структуру и семантику, описываются одним *классом*, который и является, по сути, определением структуры *объектов*, порожденных от него.

В свою очередь, каждый *класс*, или описание, всегда имеет строгий шаблон, задаваемый языком программирования или выбранной объектной моделью. Он определяет, например, допустимо ли множественное *наследование*, какие существуют ограничения на именование *классов*, как описываются поля и методы, набор существующих типов данных и многое другое. Таким образом, *класс* можно рассматривать как *объект*, у которого есть свойства (имя, список полей и их типы, список методов, список аргументов для каждого метода и т.д.). Также *класс* может обладать *поведением*, то есть поддерживать методы. А раз для любого *объекта* существует шаблон, описывающий свойства и *поведение* этого *объекта*, значит, его можно определить и для *класса*. Такой шаблон, задающий различные *классы*, называется ***метаклассом***.

Чтобы представить себе, что такое *метакласс*, рассмотрим пример некой бюрократической организации. Будем считать, что все *классы* в такой системе представляют собой строгие инструкции, которые описывают, что нужно сделать, чтобы породить новый *объект* (например, нанять нового служащего или открыть новый отдел). Как и полагается *классам*, они описывают все свойства новых *объектов* (например, зарплату и профессиональный уровень для сотрудников, площадь и имущество для отделов) и их *поведение* (обязанности служащих и функции подразделений).

В свою очередь, написание новой инструкции можно строго регламентировать. Скажем, необходимо использовать специальный бланк, придерживаться правил оформления и заполнить все обязательные поля (например, номер инструкции и фамилии ответственных работников). Такая "инструкция инструкций" и будет представлять собой *метакласс* в *ООП*.

Итак, *объекты* порождаются от *классов*, а *классы* - от *метакласса*. Он, как правило, в системе только один. Но существуют языки программирования, в которых можно создавать и использовать собственные *метаклассы*, например язык Python.

В частности, функциональность *метакласса* может быть следующая: при формировании *класса* он будет просматривать список всех методов в *классе* и, если имя метода имеет вид set_XXX или get_XXX, автоматически создавать поле с именем XXX, если такого не существует.

Поскольку *метакласс* сам является *классом*, то нет никакого смысла в создании "мета-мета-классов".

В языке Java также есть *метакласс*. Это *класс*, который так и называется - Class (описывает *классы*), он располагается в основной библиотеке java.lang. Виртуальная машина использует его по прямому назначению. Когда загружается очередной .class -файл, содержащий описание нового *класса*, JVM порождает *объект класса* Class, который будет хранить его структуру. Таким образом, Java использует концепцию *метакласса* в самых практических целях.

С помощью Class реализована поддержка статических (static) полей и методов. Этот *класс* содержит ряд методов, полезных для разработчиков. Они будут рассмотрены в следующих лекциях.

Достоинства ООП

Одной из самых значительных проблем проектирования является сложность. Чем больше и сложнее программная система, тем важнее разбить ее на небольшие, четко очерченные части. Чтобы справиться со сложностью, необходимо абстрагироваться от деталей. В этом смысле *классы* представляют собой весьма удобный инструмент.

- *Классы* позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.
- Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- *Инкапсуляция* позволяет привнести свойство *модульности*, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.

ООП дает возможность создавать расширяемые системы. Это одно из основных достоинств *ООП*, и именно оно отличает данный подход от традиционных методов программирования. *Расширяемость* означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе исполнения программы.

Полиморфизм оказывается полезным преимущественно в следующих ситуациях.

- Обработка разнородных структур данных. Программы могут работать, не различая вида *объектов*, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.

- Изменение *поведения* во время исполнения. На этапе исполнения один *объект* может быть заменен другим, что позволяет легко, без изменения кода, адаптировать алгоритм в зависимости от того, какой используется *объект*.
- Реализация работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом *объектов*.
- Создание "каркаса" (framework). Независимые от приложения части предметной области могут быть реализованы в виде набора универсальных *классов*, или каркаса (framework), и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Часто многоразового использования программного обеспечения не удается добиться из-за того, что существующие компоненты уже не отвечают новым требованиям. *ООП* помогает этого достичь без нарушения работы уже имеющихся компонентов, что позволяет извлечь *максимум* из многоразового использования компонентов.

- Сокращается время на разработку, которое может быть отдано другим задачам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с ним программ.
- Если программа опирается на стандартные компоненты, ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает использование.

Недостатки ООП

Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях *классов* поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному *классу*. Для получения такой информации нужны специальные инструменты, вроде навигаторов *классов*. Если конкретный *класс* расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому *классу*. Реализация *операции*, таким образом, рассредотачивается *по* нескольким *классам*, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными, зато их намного больше. В коротких методах легче разобраться, но они неудобны тем, что код для обработки сообщения иногда "размазан" *по* многим маленьким методам.

Инкапсуляцией данных не следует злоупотреблять. Чем больше логики и данных скрыто в недрах *класса*, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с *классом* этих данных знать не требуется.

Многие считают, что *ООП* является неэффективным. Как же обстоит дело в действительности? Мы должны проводить четкую грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

1. Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления их поиска в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных С-программ.

В гибридных языках типа Oberon-2, Object Pascal и C++ отправка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает.

Однако существует другой фактор, который влияет на время выполнения: это *инкапсуляция* данных. Рекомендуется не предоставлять прямой доступ к полям *класса*, а выполнять каждую операцию над данными через методы. Такая схема приводит к необходимости выполнения процедурного вызова каждый раз при доступе к данным. Однако если *инкапсуляция* используется только там, где она необходима (т.е. в тех случаях, когда это становится преимуществом), то замедление вполне приемлемое.

2. Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информацию о типе *объекта*. Такая информация хранится в дескрипторе типа и он выделяется один на *класс*. Каждый *объект* имеет невидимый указатель на дескриптор типа для своего *класса*. Таким образом, в объектно-ориентированных программах необходимая дополнительная память выражается в одном указателе для *объекта* и в одном дескрипторе типа для *класса*.

3. Излишняя универсальность. Неэффективность также может означать, что в программе реализованы избыточные возможности. В библиотечном *классе* часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом. Это не влияет на время выполнения, но сказывается на размере кода.

Одно из возможных решений - строить базовый *класс* с минимальным числом методов, а затем уже реализовывать различные расширения этого *класса*, которые позволят нарастить функциональность. Другой подход - дать компоновщику возможность удалять лишние методы. Такие интеллектуальные компоновщики уже существуют для различных языков и операционных систем.

Но нельзя утверждать, что *ООП* неэффективно. Если *классы* используются лишь там, где это действительно необходимо, то потеря эффективности из-за повышенного расхода памяти и меньшей производительности незначительна. Кроме того, *надежность* программного обеспечения и быстрота его написания часто бывает важнее, чем *производительность*.

Паттерны, их классификация

При реализации проектов по разработке программных систем и моделированию бизнес-процессов встречаются ситуации, когда решение проблем в различных проектах имеют сходные структурные черты. Попытки выявить похожие схемы или структуры в рамках *объектно-ориентированного анализа и проектирования* привели к появлению понятия паттерна, которое из абстрактной категории превратилось в непременный *атрибут* современных CASE-средств

Паттерны ООАП различаются степенью детализации и уровнем абстракции. Предлагается следующая общая классификация паттернов по категориям их применения:

- *Архитектурные паттерны*
- *Паттерны проектирования*
- *Паттерны анализа*
- *Паттерны тестирования*
- *Паттерны реализации*

Архитектурные паттерны(Architectural patterns) - множество предварительно определенных подсистем со спецификацией их ответственности, правил и базовых принципов установления отношений между ними.

Архитектурные паттерны предназначены для спецификации фундаментальных схем структуризации программных систем. Наиболее известными паттернами этой категории являются паттерны GRASP (*General Responsibility Assignment Software Pattern*). Эти паттерны относятся к уровню системы и подсистем, но не к уровню классов. Как правило, формулируются в обобщенной форме, используют обычную терминологию и не зависят от области приложения. Паттерны этой категории систематизировал и описал К. Ларман.

Паттерны проектирования (Design patterns) - специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними.

Паттерны проектирования описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте. Наиболее известными паттернами этой категории являются паттерны GoF (*Gang* of Four), названные в честь Э. Гаммы, Р. Хелма, Р. Джонсона и Дж. Влиссидеса, которые систематизировали их и представили общее описание. Паттерны GoF включают в себя 23 паттерна. Эти паттерны не зависят от языка реализации, но их реализация зависит от области приложения.

Паттерны анализа (Analysis patterns) - специальные схемы для представления общей организации процесса моделирования.

Паттерны анализа относятся к одной или нескольким предметным областям и описываются в терминах *предметной области*. Наиболее известными паттернами этой группы являются паттерны бизнес-моделирования *ARIS* (*Architecture* of Integrated *Information* Systems), которые характеризуют абстрактный *уровень представления* бизнес-процессов. В дальнейшем *паттерны анализа* конкретизируются в типовых моделях с целью выполнения аналитических оценок или имитационного моделирования бизнес-процессов.

Паттерны тестирования (Test patterns) - специальные схемы для представления общей организации процесса тестирования программных систем.

К этой категории паттернов относятся такие паттерны, как тестирование черного ящика, белого ящика, отдельных классов, системы. Паттерны этой категории систематизировал и описал М. Гранд. Некоторые из них реализованы в инструментальных средствах, наиболее известными из которых является *IBM* Test Studio. В связи с этим *паттерны тестирования* иногда называют стратегиями или схемами тестирования.

Паттерны реализации (Implementation patterns) - совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода.

Эта категория паттернов делится на следующие подкатегории: паттерны организации программного кода, паттерны *оптимизации программного кода*, паттерны устойчивости кода, паттерны разработки графического интерфейса пользователя и др. Паттерны этой категории описаны в работах М. Гранда, К. Бека, Дж. Тидвелла и др. Некоторые из них реализованы в популярных интегрированных средах программирования в форме шаблонов создаваемых проектов. В этом случае выбор шаблона программного приложения позволяет получить некоторую заготовку программного кода.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

Практическое занятие 1. Оценка сложности алгоритмов сортировки

Цель. Приобретение навыков исследования временной сложности алгоритмов и определения ее асимптотических оценок.

1. Требования к содержанию, оформлению и порядку выполнения

В содержательной части отчета по выполнению практической работы требуется привести описание алгоритма, выбранного согласно своему варианту, провести его анализ и определить асимптотические оценки его временной сложности. Алгоритм рекомендуется оформлять с помощью блок-схем.

2. Теоретическая часть

Проблема выбора алгоритма. Понятие временной сложности

Если способов решения задачи может быть несколько, то становится вопрос выбора наилучшего из существующих алгоритмов. Что означает – наилучший алгоритм? Каков критерий этой наилучшести? Так что, выбор подходящего алгоритма вызывает определенные трудности. В самом деле, на чем основывать свой выбор, если алгоритм должен удовлетворять следующим противоречащим друг другу требованиям. Быть простым для понимания, перевода в программный код и отладки.

Эффективно использовать компьютерные ресурсы и выполняться по возможности быстро.

Если написанная программа должна выполняться только несколько раз, то первое требование наиболее важно. Стоимость рабочего времени программиста обычно значительно превышает стоимость машинного времени выполнения программы, поэтому стоимость программы оптимизируется по стоимости написания (а не выполнения) программы.

Если мы имеем дело с задачей, решение которой требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа должна выполняться многократно. Поэтому, с финансовой точки зрения, более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее, чем более простая программа). Но и в этой ситуации разумнее сначала реализовать простой алгоритм, чтобы определить, как должна себя вести более сложная программа. При построении сложной программной системы желательно реализовать ее простой прототип, на котором можно провести необходимые измерения и смоделировать ее поведение в целом, прежде чем приступать к разработке окончательного варианта. Таким образом, программисты должны быть осведомлены не только о методах построения быстрых программ, но и знать, когда их следует применить (желательно с минимальными программистскими усилиями).

На время выполнения программы влияют следующие факторы.

Ввод исходной информации в программу.

Качество скомпилированного кода исполняемой программы.

Машинные инструкции, используемые для выполнения программы.

Временная сложность алгоритма соответствующей программы.

Поскольку время выполнения программы зависит от ввода исходных данных, его можно определить как функцию от исходных данных. Но зачастую время выполнения программы зависит не от самих исходных данных, а от их "размера". В общем случае длина (размерность) входных данных является наиболее подходящей мерой объема входной информации. Обычно говорят, что время выполнения программы имеет порядок $T(n)$ от входных данных размераn.

Однако для многих программ время выполнения действительно является функцией входных данных, а не их размера. В этой ситуации $T(n)$ определяется как время выполнения *в наихудшем случае,* т.е. как максимум времени выполнения по всем входным данным размера n, или время выполнения *в наилучшем случае*, т.е. как минимум выполнения по всем входным данным размера n.

Наиболее точной оценкой времени выполнения программы является аналитическое выражение для функции: $t=T(n)$, гдеt– время выполнения программы от момента ее запуска до получения окончательного результата. Для ее получения необходимо проделать множество экспериментов, запуская программу с различными исходными данными и в широких пределах варьируя их размерность. Результаты экспериментов необходимо затабулировать и подобрать функцию, наилучшем образом приближающую, получившеюся таблицу.

Однако данный подход имеет множество недостатков. Во первых, точное измерение времени выполнения программы невозможно. Во вторых, функция, аппроксимирующая таблицу экспериментальных данных, существенно зависит от количества и качества проведенных экспериментов и, как правило, весьма приближенно отражает искомую временную зависимость. В третьих, на время выполнения существенное влияние оказывают второй и третий из выше перечисленных факторов – компилятор, используемый для компиляции программы, и машина, на которой выполняется программа. При изменении компилятора и (или) компьютера, с помощью которых создается и исполняется программа, результаты измерения времени могут существенно отличаться.

Эти факторы влияют на то, что для измерения времени выполнения $T(n)$ невозможно применить стандартные единицы измерения, такие как секунды или миллисекунды. Поэтому остается только делать заключения, подобные "время выполнения такого-то программы пропорционально n^2". Константы пропорциональности также нельзя точно определить, поскольку они зависят от компилятора, компьютера и других факторов.

Указанные соображения позволяют сделать вывод о том, что функциональная зависимость и единица измерения $T(n)$ не может быть точно определена. В практике оценки времени выполнения программ, $T(n)$ понимают как количество элементарных шагов – инструкций, выполняемых на идеализированном компьютере. Это позволяет абстрагироваться от факторов, вносимых конкретным компилятором и конкретной вычислительной машиной. Единственным значимым и определяющем фактором, в этом случае остается временная сложность алгоритма. Поэтому очень часто время выполнения программы отождествляют с временной сложностью ее алгоритма.

Таким образом, ***временной сложностью алгоритма*** называется зависимость времени выполнения алгоритма от количества обрабатываемых входных данных, измеряемое в "шагах" (инструкциях алгоритма), которые необходимо выполнить алгоритму для достижения запланированного результата.

Асимптотические соотношения оценки временной сложности

Как было показано ранее, время выполнения программы невозможно определить точно путем проведения серии экспериментов. Другой способ оценки временной сложности – математически оценить время исполнения подсчетом операций. Проиллюстрируем этот подход на примере задачи вычисления значения многочлена.

Задача вычисления значения многочлена. Дан многочлен степени n,

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_i x^i + \dots + a_1 x^1 + a_0, \quad (3.7)$$

заданный массивом своих коэффициентов $A=\{A[0], A[1],\dots, A[n]\}$ и значение переменной x. Вычислить значение многочлена $S = Pn(x)$.

Можно предложить следующий способ ее решения (*алгоритм 1*): для каждого слагаемого, кроме a_0 возвести x в заданную степень последовательным умножением и затем помножить на коэффициент. Затем слагаемые сложить. На рисунке 1. приведена блок-схема этого алгоритма.

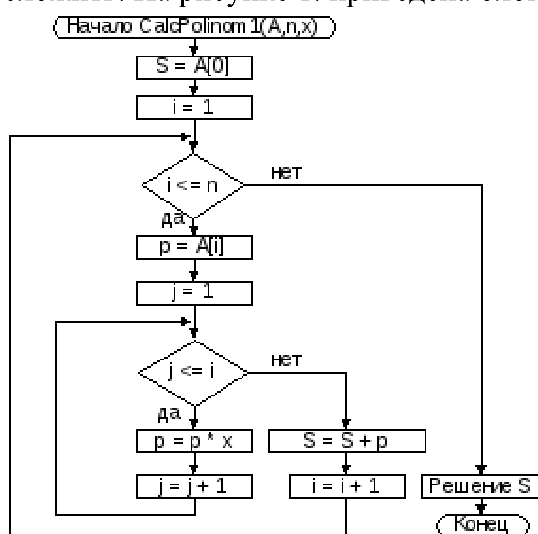


Рис. 1. Вычисления значения многочлена (алгоритм 1)

Для простоты анализа временной сложности $T(n)$ будем принимать в расчет только операции, связанные с вычислением значения многочлена игнорируя операции сравнения, инициализации и изменения значений параметров цикла.

Вычисление i-го слагаемого($i=1..n$) требует i умножений. Значит, всего $1 + 2 + 3 + \dots + n = n(n+1)/2$ умножений. Кроме того, требуется n сложений и одна операция начального присваивания значения a_0. Таким образом, временная сложность этого алгоритма: $T(n) = n(n+1)/2 + n + 1 = n^2/2 + 3n/2 + 1$ операций.

Теперь рассмотрим другой подход (<u>*алгоритм 2*</u>): в формуле (3.7) вынесем x-ы за скобки и перепишем многочлен в виде

$P_n(x) = a_0 + x(a_1 + x(a_2 + \dots (a_i + .. x(a_{n-1} + a_n x)))$. (3.8)

Расстановка скобок диктует такой порядок вычислений:

$$S_0 = a_n,$$
$$S_1 = S_0 x + a_{n-1},$$
$$S_2 = S_1 x + a_{n-2},$$
$$\ldots,$$
$$S_i = S_{i-1} x + a_{n-i},$$
$$\ldots,$$
$$Sn = Sn\text{-}1\ x + a0,\ Pn(x) = Sn.$$

Рассмотренный метод называется схемой Горнера. На рисунке 2 приведена блок-схема алгоритма вычисления значения многочлена по схеме Горнера.

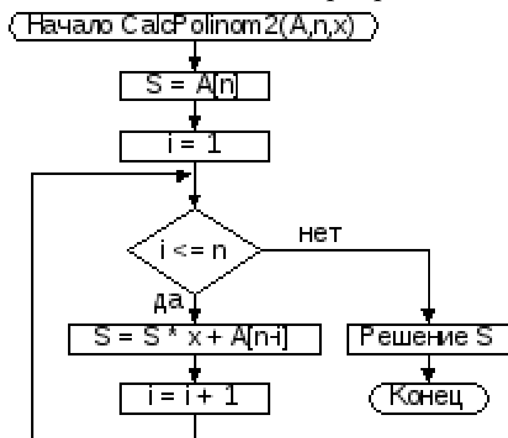


Рис. 2. Вычисления значения многочлена по схеме Горнера (алгоритм 2)

Подсчет временной сложности в этом случае гораздо проще: для вычисления S_i требуется 1 умножение и 1 сложение. Всего такая итерация осуществляется *n* раз. Таким образом, временная сложность этого алгоритма: *T(n) =n* умножений +*n* сложений =*2n* операций.

Зачастую такая подробная оценка временной сложности не требуется. Вместо нее приводят лишь асимптотическую скорость возрастания количества операций при увеличении *n*.

Так, например, функция $T(n) = n^2/2 + 3n/2 + 1$ возрастает приблизительно как $n^2/2$ (отбрасываем сравнительно медленно растущее слагаемое $3n/2+1$). Константный множитель ½ также убираем и получаем асимптотическую оценку для алгоритма 1, которая обозначается специальным символом $O(n^2)$ (читается как "*О большое от эн квадрат*"). Это – верхняя оценка, т.е количество операций (а значит, и время работы) растет не быстрее, чем квадрат количества элементов. В этом случае говорят, что *T(n) есть* $O(n^2)$, или *T(n) имеет порядок* $O(n^2)$.

Чтобы почувствовать, что это такое, рассмотрим на таблицу, где приведены числа, иллюстрирующие скорость роста для нескольких разных функций.

Если считать, что числа в таблице соответствуют микросекундам, то для задачи с *n=1048576* элементами алгоритму с временем работы порядка *O(log n)* потребуется 20 микросекунд, алгоритму со временем порядка *O(n)*– 17 минут, а алгоритму с временем работы порядка $O(n^2)$– более 12 дней. Теперь преимущество алгоритма 2 с оценкой *O(n)* перед алгоритмом 1 достаточно очевидно.

n	**$\log_2 n$**	**$n \log_2 n$**	**n^2**
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 576	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Наилучшей является оценка $O(1)$. В этом случае время вообще не зависит от n, т.е постоянно при любом количестве элементов.

Таким образом, $O(\ )$– "урезанная" верхняя оценка временной сложности алгоритма, отражающая поведение этой сложности в пределе при увеличении размера задачи до бесконечности. Она называется *асимптотической временной сложностью* или *верхним порядком роста временной сложности.*

Определение 3.1. $O(f(n))$– множество функций $T(n)$, для которых существуют такие константысиn_0, что $T(n) \leq c\ f(n)$ для всех $n \geq n_0$. Запись $T(n) = O(f(n))$ дословно обозначает, что $T(n)$ принадлежит множеству $O(f(n))$. Обратное выражение $O(f(n)) = T(n)$ не имеет смысла.

В частности, можно сказать, что $T(n) = 50\ n$ принадлежит $O(n^2)$. Здесь мы имеем дело с неточной оценкой. Разумеется, $T(n) \leq 50\ n^2$ при $n \geq 1$, однако более сильным утверждением было бы $T(n) = O(n)$, так как для$c=50$ и $n_0 = 1$ верно $T(n) \leq cn,\ n \geq n_0$.

Наряду с оценкой $O(n)$ используется оценка $\Omega(n)$ [читается как "*Омега большое от эн*"]. Она обозначает нижнюю оценку роста функции.

Определение 3.2. $\Omega(g(n))$– множество функций $T(n)$, для которых существуют такие константысиn_0, что $c\ g(n) \leq T(n)$ для всех $n \geq n_0$. В этом случае говорят, что $T(n)$имеет *нижний порядок роста* $g(n)$.

Например, пусть количество операций алгоритма описывает функция $T(n) = \Omega(n^2)$. Это значит, что даже в самом удачном случае будет произведено не менее порядкаn^2действий. В то время как оценка $T(n) = O(n^3)$ гарантирует, что в самом худшем случае действий будет порядка n^3, не больше.

Также используется оценка $\Theta(n)$ [читается как "*Тэта от эн*"], которая является комбинаций $O(n)$ и $\Omega(n)$. Она является точной оценкой асимптотики.

Определение 3.3. $\Theta(g(n))$– множество функций $T(n)$, для которых существуют такие константы c_1, c_2 и n_0, что $c_1\ g(n) \leq T(n) \leq c_2\ g(n)$ для всех $n \geq n_0$. Оценка $\Theta(g(n))$ существует только тогда, когда $O(g(n))$ и $\Omega(g(n))$совпадают и равна им.

Для рассмотренных выше алгоритмов вычисления многочлена временная сложность первого алгоритма порядка $O(n^2)$, $\Omega(n^2)$ и $\Theta(n^2)$, а второго – порядка $O(n)$, $\Omega(n)$ и $\Theta(n)$. Если добавить к первому алгоритму проверки на $x=0$ в возведении в степень, то на самых удачных исходных данных (когда$x=0$) имеем порядка n проверок, 0 умножений и 1 сложение, что дает новую оценку $\Omega(n)$ вместе со старой $O(n^2)$. Как правило, основное внимание все же обращается на верхнюю оценку $O(\ )$, поэтому, несмотря на "улучшение", алгоритм 2 остается предпочтительнее.

Итак, $O(\ )$– асимптотическая оценка алгоритма на худших входных данных,$\Omega(\ )$ –на лучших входных данных,$\Theta(\ )$– сокращенная запись одинаковых $O(\ )$и $\Omega(\ )$. Интуитивный смысл этих оценок:

$$\text{При } n \to \infty\ T(n) = \begin{cases} O(g(n)) : T(n) \text{ растет не быстрее } g(n); \\ \Omega(g(n)) : T(n) \text{ растет не медленне } g(n); \\ \Theta(g(n)) : T(n) \text{ растет так же, как и } g(n). \end{cases}$$

Вычисление временной сложности

Теоретическое нахождение времени выполнения программ (даже без определения констант пропорциональности) — сложная математическая задача. Однако на практике определение времени выполнения (также без нахождения значения констант) является вполне разрешимой задачей – для этого нужно знать только несколько базовых принципов.

1. *При оценке за функцию берется количество операций, возрастающее быстрее всего.* То есть, если в программе одна функция, например, умножение, выполняется $O(n)$ раз, а сложение – $O(n^2)$ раз, то общая сложность программы –$O(n^2)$, так как в конце концов при увеличении n более быстрые (в определенное, константное число раз) сложения станут выполнятся настолько часто, что будут влиять на быстродействие куда больше, нежели медленные, но редкие умножения. Это принцип обуславливает *правило сумм асимптотических соотношений.* Пусть $T1(n)$ и $T2(n)$ — вре-

мя выполнения двух программных фрагментов *P1* и *P2, T1(n)* имеет степень роста *O(f(n)),* а *T2(n) —O(g(n)).* Тогда *T1(n) +T2(n),* т.е. время последовательного выполнения фрагментов *P1* и*P2,* имеет степень роста *O(max(f(n), g(n))).*

2. *При оценке O( ) константы не учитываются.* Пусть один алгоритм делает *2500n + 1000* операций, а другой *–2n+1*. Оба они имеют оценку *O(n)*, так как их время выполнения растет линейно. Этот принцип является следствием *правила произведений,* которое заключается в следующем. Если $T_1(n)$ и $T_2(n)$ имеют степени роста *O(f(n))* и *O(g(n))* соответственно, то произведение $T_1(n) \cdot T_2(n)$ имеет степень роста *O(f(n)g(n)).*

3.Другое следствие опускания константы – алгоритм со временем $O(n^2)$может работать значительно быстрее алгоритма *O(n)*при малых *n*. За счет того, что реальное количество операций первого алгоритма может быть$n^2 + 10n + 6$, а второго $-1000000n + 5$. Впрочем, второй алгоритм рано или поздно обгонит первый.n^2растет куда быстрее*1000000n*.

4.*Основание логарифма внутри символа O( ) не пишется.* Причина этого весьма проста. Пусть у нас есть $O(\log_2 n)$. Но$\log_2 n = \log_3 n / \log_3 2$, а$\log_3 2$, как и любую константу, асимптотика – символ *O( )*не учитывает. Таким образом, $O(\log_2 n) = O(\log_3 n)$. К любому основанию мы можем перейти аналогично, а значит и писать его не имеет смысла.

Теперь дадим несколько правил анализа программ. В общем случае время выполнения оператора или группы операторов можно параметризовать с помощью размера входных данных и/или одной или нескольких переменных. Но для времени выполнения программы в целом допустимым параметром может быть только *n,* размер входных данных.

1. Время выполнения оператора присваивания складывается из времени вычисления выражения и времени выполнения операции присваивания. Если выражение не содержит вызовов функций (в том числе перегруженных операций), то на его вычисление тратится некоторое постоянное время. Операция присваивания так же требует некоторого постоянного времени. Таким образом, время выполнения операторов присваивания обычно имеет порядок *O(1).*

2. Время выполнения операторов ввода – вывода обычно имеет порядок *O(1).*

3. Время выполнения последовательности операторов определяется с помощью правила сумм. Поэтому степень роста времени выполнения последовательности операторов без определения констант пропорциональности совпадает с наибольшим временем выполнения оператора в данной последовательности.

4. Время выполнения условных операторов состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения. Время вычисления логического выражения обычно имеет порядок *O(1).* Время для всей конструкции ***if-then-else*** состоит из времени вычисления логического выражения и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения ***true*** (истина) и при значении ***false*** (ложь).

5. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок *O(1)*). Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла. Время выполнения каждого цикла, если в программе их несколько, должно определяться отдельно.

6. Для программ, содержащих несколько процедур или функций (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения этих подпрограмм, начиная с той, которая не имеет вызовов других процедур или функций. Так как мы предположили, что все подпрограммы нерекурсивные, то должна существовать хотя бы одна подпрограмма, не имеющая вызовов других процедур или функций. Затем можно определить время выполнения подпрограмм, вызывающих эту подпрограмму, используя уже вычисленное время выполнения вызываемой подпрограммы. Продолжая

этот процесс, найдем время выполнения всех подпрограмм и, наконец, время выполнения всей программы.

7. Если есть рекурсивные процедуры или функции, то нельзя упорядочить все подпрограммы таким образом, чтобы каждая подпрограмма вызывала только процедуры или функции, время выполнения которых подсчитано на предыдущем шаге. В этом случае мы должны с каждой рекурсивной процедурой (функцией) связать временною функцию *T(n),* где *n* определяет объем ее аргументов. Затем мы должны получить *рекуррентное соотношение* для *T(n)*, т.е. уравнение (или неравенство) для *T(n),* где участвуют значения *T(k)* для различных значений *k<n.*

Общий метод решения таких рекуррентных соотношений состоит в последовательном раскрытии выражений *T(к)* в правой части уравнения (путем подстановки в исходное соотношение *k* вместо *n)* до тех пор, пока не получится формула, у которой в правой части отсутствует *T*. При этом часто приходится находить суммы различных последовательностей; если значения таких сумм нельзя вычислить точно, то для сумм находятся верхние границы, что позволяет, в свою очередь, получить верхние границы для *T(n).*

3. Общая постановка задачи

Требуется провести анализ и оценку временной сложности заданного алгоритма. Варианты заданий представлены в таблице.

4. Список индивидуальных данных

Данные для выполнения практической работы сведены в таблице.

Вариант	*Алгоритм*
1	Алгоритм быстрого возведения в степень
2	Алгоритм быстрого возведения в степень
3	Алгоритм сортировки обменом
4	Алгоритм сортировки выбором
5	Алгоритм сортировки вставками
6	Алгоритм быстрой сортировки

5. *Пример выполнения работы*

Рассмотрим 1 вариант. Требуется провести анализ и оценку временной сложности алгоритма быстрого возведения в степень.

Алгоритм предназначен для решения следующей задачи: дано число *x*и натуральное (целое неотрицательное) число$n \geq 0$. Вычислить значение функции $f(x) = x^n$.

Число *n*можно представить в виде:

$$n = d_m \cdot 2^m + d_{m-1} \cdot 2^{m-1} + \ldots + d_1 \cdot 2^1 + d_0 \cdot 2^0,$$

где $d_i \in \{0,1\},\ 0 \leq i \leq m$ – двоичная цифра, $m = \lfloor \log_2 n \rfloor$ – ближайшее целое не превосходящееlog₂ *n*(количество двоичных знаков, необходимых для записи числа минус 1).

Тогда:

$$x^n = x^{d_m 2^m} \cdot x^{d_{m-1} 2^{m-1}} \cdot \ldots \cdot x^{d_1 2} \cdot x^{d_0}.$$

Введем обозначения: $c_i = x^{2^i}$, $0 \leq i \leq m$, тогда предыдущую формулу можно представить в виде:

$$x^n = \prod_{i=0}^{m} c_i^{d_i}.$$

Легко заметить, что $c_0 = x$, и$c_i = c_{i-1}^2$ для всех $1 \leq i \leq m$.

Другое важное замечание состоит в том, что:

$$c_i^{d_i} = \begin{cases} c_i, \text{если } d_i = 1 \\ 1, \text{если } d_i = 0 \end{cases}.$$

Алгоритм быстрого возведения в степень основан на выше приведенных соображениях. Его блок-схема приведена на рис. Л2.1.

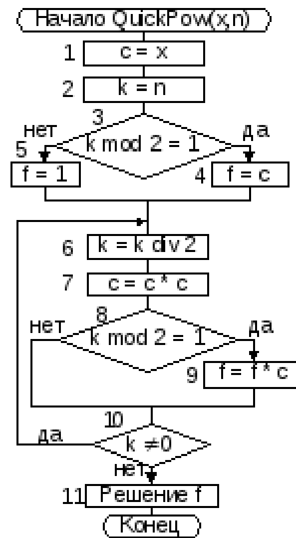


Рис.1. Алгоритм быстрого возведения в степень

Проведем его анализ. Сначала отметим, что, степень, в которую возводится число, может служить мерой объема входных данных и все операторы присваивания и сравнения имеют некоторое постоянное время выполнения, независящее от размера входных данных.

Наибольшее влияние на временную сложность алгоритма оказывают циклические конструкции. Рассмотрим группу блоков (6) – (10) алгоритма. Эти блоки составляют цикл с постусловием. Тело цикла (блоки (6) – (9)) имеет временную сложность порядка *O(1)*. Кроме этого в цикле выполняется проверка условия завершения цикла (блок 10), которая так же имеет время выполнения порядка *O(1)*. Определим количество повторений тела цикла в зависимости от n. Таблица содержит динамику изменения параметра цикла k в зависимости от номера итерации и параметра n.

№ итерации	Значение параметра цикла k													
	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	...	$n=7$	$n=8$	...	$n=15$	$n=16$	...	$n=31$	$n=32$	...
1	0	1	2	3	4	…	7	8	…	15	16	…	31	32	…
2		0	1	1	2	…	3	4	…	7	8	…	15	16	…
3			0	0	1	…	1	2	…	3	4	…	7	8	…
4					0	…	0	1	…	1	2	…	3	4	…
5								0	…	0	1	…	1	2	…
6											0	…	0	1	…
7														0	…

Анализируя данные, приведенные в таблице, можно отметить, что на 1-й итерации $k_1 = n \ div \ 2^0 = n,$ на этой и каждой последующей итерации k уменьшается в два раза, по сравнению с предыдущем значением, до тех пор, пока k не станет равно 1 (пусть это будет итерация $m+1$). На следующей итерации: $k_{m+2} = k_{m+1} \ div \ 2 \ = 0$, что обеспечивает условие завершения цикла. Таким образом, для произвольного n>0 имеем:

$$k_1 = n \ div \ 2^0 = n;$$
$$k_2 = n \ div \ 2^1;$$
$$k_3 = n \ div \ 2^2;$$
$$..................$$
$$k_i = n \ div \ 2^{i-1};$$
$$..................$$
$$k_{m+1} = n \ div \ 2^m = 1;$$
$$k_{m+2} = n \ div \ 2^{m+1} = 0.$$

Всего получается m+2 итераций, где m определяется соотношением $2^m \le n < 2^{m+1}$. Из последнего соотношения следует: $m = \lfloor \log_2 n \rfloor$ – ближайшее целое не превосходящееlog$_2$ n. Итого: количество повторений цикла равняется $\lfloor \log_2 n \rfloor + 2$, а временная сложность группы блоков (6) – (10) составляет *O(*log *n)*.

Время выполнения конструкции ветвления (блоки (3), (4) и (5)) состоит из времени вычисления логического выражения (блок (3)) и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения ***true*** (блок (4)) и при значении ***false*** (блок (5)). Каждый из этих блоков имеет время выполнения порядка *0(1)*, соответственно вся конструкция в целом имеет временную сложность порядка *O(1)*. Группа блоков (1) – (2), так же как и блок (11), имеет время выполнения порядка *O(1)*. Группы блоков (1) – (2), (3) – (5), (6) – (10) и (11) выполняются последовательно. Согласно правилу сумм, общая временная сложность алгоритма $T(n) = O(\max(1,1,\log n,1)) = O(\log n)$.

Следует отметить, что в этом алгоритме имеет место ситуация: $T(n) = \Omega(\log n)$, и следовательно $T(n) = \Theta(\log n)$.

Вывод. В практической работе проведен анализ алгоритма быстрого возведения в степень. Его временная сложность имеет верхнюю оценку $O(\log n)$. Нижняя оценка составляет $\Omega(\log n)$, следовательно, имеет место оценка $\Theta(\log n)$. Данный алгоритм имеет лучшую оценку временной сложности чем тривиальный алгоритм, который, "на вскидку" имеет временную сложность порядка $O(n)$.

Контрольные вопросы

1. Понятие временной сложности алгоритма.
2. Определение асимптотических оценок временной сложности.
3. Основные принципы получения асимптотических оценок.
4. Правила анализа алгоритмов с целью определения их временной сложности.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНЫЙ ЗАНЯТИЯМ

Лабораторная работа 8. «Использование структурных шаблонов »

Цель: ознакомиться с основными шаблонами проектирования, научиться применять их при проектировании и разработке ПО.

Теоретические сведения

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

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

Сообразное использование паттернов проектирования дает разработчику ряд неоспоримых преимуществ. Приведем некоторые из них. Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех элементов и связей, которые значимы при решении поставленной задачи. Помимо этого, модель, построенная с использованием паттернов проектирования, более проста и наглядна в изучении, чем стандартная модель. Тем не менее, несмотря на простоту и наглядность, она позволяет глубоко и всесторонне проработать архитектуру разрабатываемой системы с использованием специального языка. Применение паттернов проектирования повышает устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы. Кроме того, трудно переоценить роль использования паттернов при интеграции информационных систем организации. Также следует упомянуть, что совокупность паттернов проектирования, по сути, представляет собой единый сло-

варь проектирования, который, будучи унифицированным средством, незаменим для общения разработчиков друг другом.

Но самое главное любой шаблон проектирования может стать палкой о двух концах: если он будет применен не к месту, это может обернуться катастрофой и создать вам много проблем в последующем. В то же время, реализованный в нужном месте, в нужное время, он может стать для вас настоящим спасителем.

Есть три основных вида шаблонов проектирования:

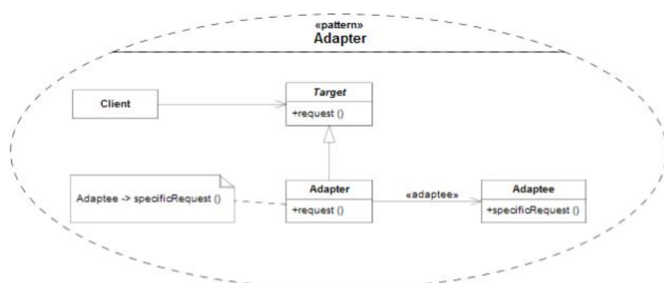
- структурные;
- порождающие;
- поведенческие.

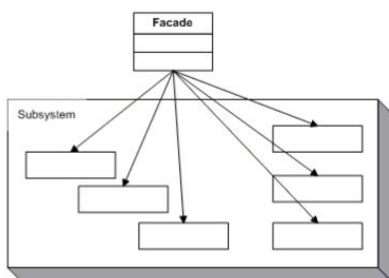
Структурные шаблоны определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Порождающие шаблоны шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Поведенческие шаблоны определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Нужно учесть, что шаблонов очень много, и рассмотрение всех шаблонов может занять целый учебный курс. Поэтому ниже кратко рассмотрены лишь ***некоторые*** из существующих паттернов..





Структурные шаблоны

Адаптер (Adapter)

Проблема: необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

Решение: преобразовать исходный интерфейс компонента к другому виду с помощью промежуточного объекта адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту адаптеру.

Класс ***Adapter*** приводит интерфейс класса ***Adaptee*** в соответствие с интерфейсом класса ***Target*** (наследником которого является ***Adapter***). Это позволяет объекту ***Client*** использовать объект ***Adaptee*** (посредством адаптера ***Adapter***) так, словно он является экземпляром класса ***Target***.

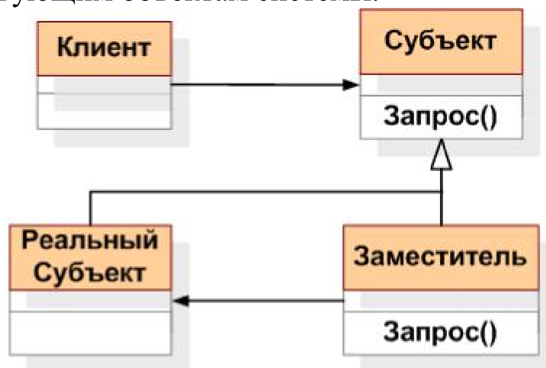
Таким образом ***Client*** обращается к интерфейсу ***Target***, реализованному в наследнике ***Adapter***, который перенаправляет обращение к ***Adaptee***.

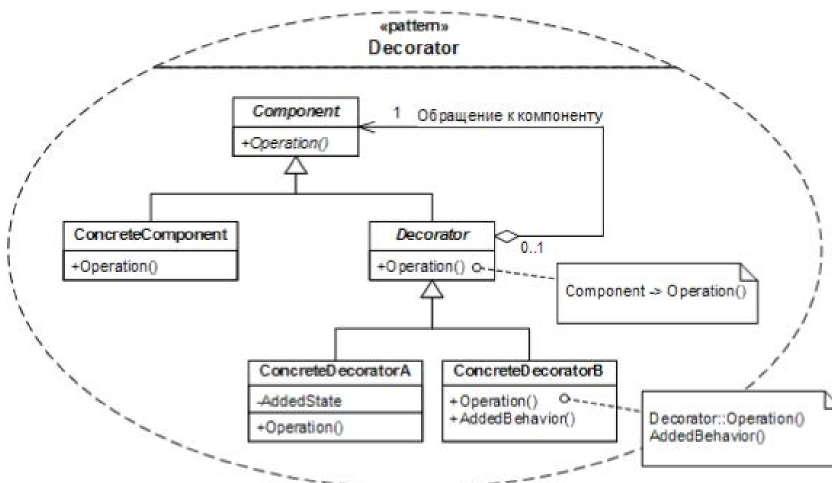
Шаблон Адаптер позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах.

Этот шаблон позволяет в процессе проектирования не принимать во внимание возможные различия в интерфейсах уже существующих классов. Если есть класс, обладающий требуемыми методами и свойствами (по крайней мере, концептуально), то при необходимости всегда можно воспользоваться шаблоном Адаптер для приведения его интерфейса к нужному виду.

Фасад (Facade)

Шаблон “фасад” структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.





Проблема: как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

Решение: определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.

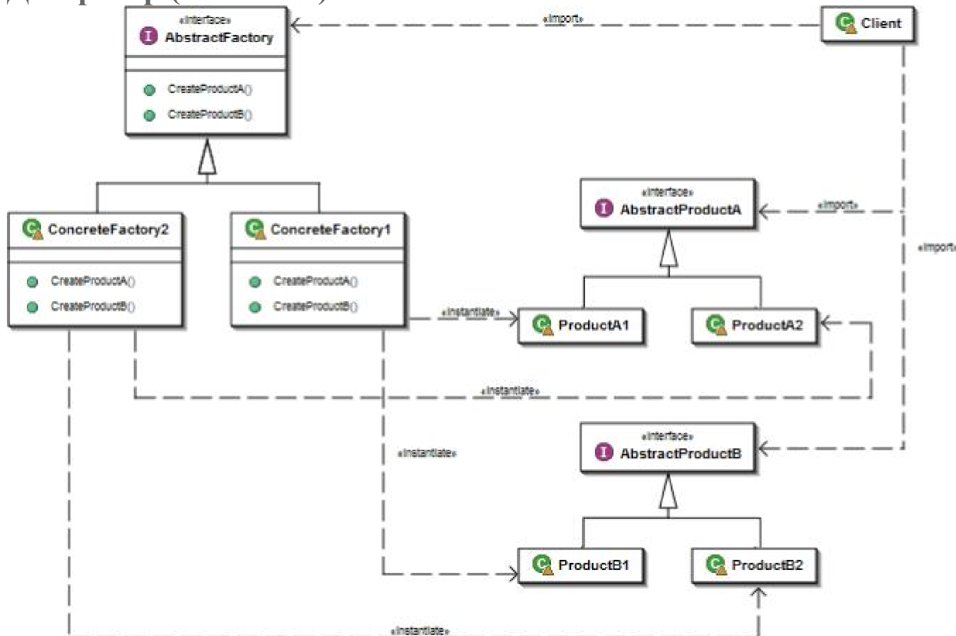
Заместитель (Proxy)

Заместитель — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы.

Проблема: необходимо управлять доступом к объекту так, чтобы не создавать громоздкие объекты «по требованию».

Решение: создать суррогат громоздкого объекта. «Заместитель» хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса «Заместитель» может обращаться к объекту класса «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы). Поскольку интерфейс «Реального Субъекта» идентичен интерфейсу «Субъекта», так, что «Заместителя» можно подставить вместо «Реального Субъекта». «Заместитель» контролирует доступ к «Реальному Субъекту», может отвечать за создание или удаление «Реального Субъекта». «Субъект» определяет общий для «Реального Субъекта» и «Заместителя» интерфейс, так, что «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту».

Декоратор (Decorator)



Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Задача: объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

Решение: шаблон “декоратор” предусматривает расширение функциональности объекта без определения подклассов.

Класс ***ConcreteComponent*** — класс, в который с помощью шаблона Декоратор добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса ***ConcreteComponent***. В подобных случаях класс ***ConcreteComponent*** является уже не конкретным, а абстрактным. Абстрактный класс ***Component*** определяет интерфейс для использования всех этих классов.

Порождающие шаблоны

Абстрактная фабрика (Abstract Factory)

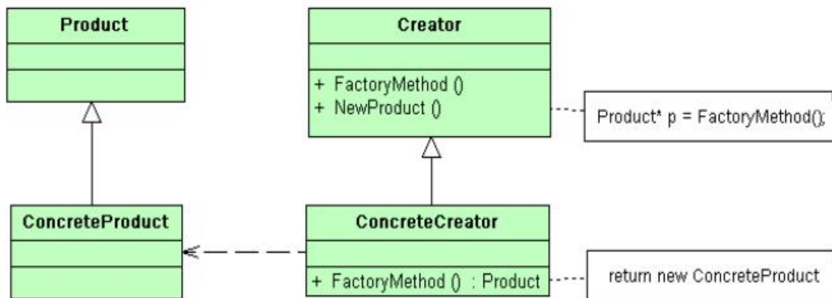
Абстрактная фабрика — порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемыми объектами, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для окон-

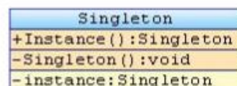
ного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Этот шаблон предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Фабричный метод (Factory Method)

Фабричный метод — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, Фабрика делегирует создание объектов наследникам





родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Шаблон определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

Структура:

Product — продукт; определяет интерфейс объектов, создаваемых абстрактным методом. ***ConcreteProduct*** — конкретный продукт, реализует интерфейс ***Product***.

Creator — создатель; объявляет фабричный метод, который возвращает объект типа ***Product***. Может также содержать реализацию этого метода «по умолчанию»; может вызывать фабричный метод для создания объекта типа Product.

ConcreteCreator — конкретный создатель; переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса ***ConcreteProduct***.

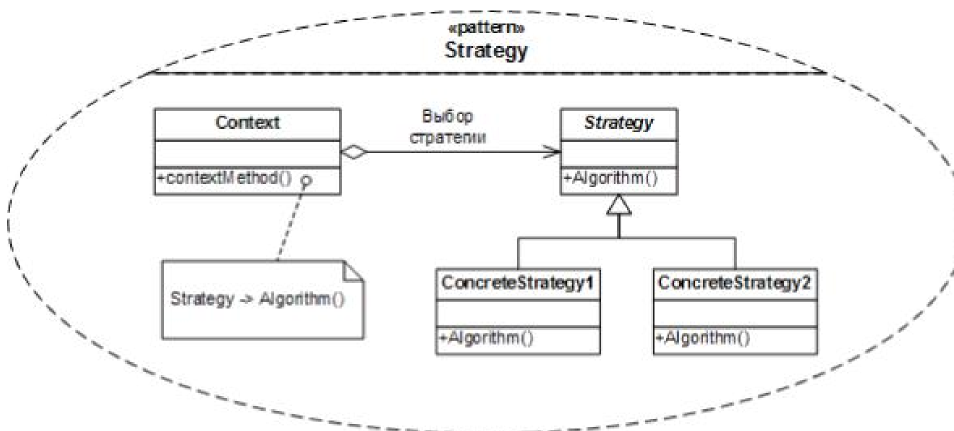
Одиночка (Singleton)

Одиночка — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр класса с глобальной точкой доступа.

Существенно то, что можно пользоваться именно ***экземпляром*** класса, так как при этом во многих случаях становится доступной более широкая функциональность.

Глобальный «одинокий» объект — именно объект, а не набор процедур, не привязанных ни к какому объекту — бывает нужен:

- ●если используется существующая объектноориентированная библиотека;
- ●если есть шансы, что один объект когданибудь превратится в несколько;
- ●если интерфейс объекта (например, игрового мира) слишком сложен, и не стоит засорять основное пространство имён большим количеством функций;
- ●если, в зависимости от какихнибудь условий и настроек, создаётся один из нескольких объектов. Например, в зависимости от того, ведётся лог или нет, создаётся или настоящий объект, пишущий в файл, или «заглушка», ничего не делающая.



Поведенческие шаблоны

Стратегия (Strategy)

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектовклиентов, которые его используют.

Проблема: по типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

Решение: отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста.

Класс Strategy определяет, как будут использоваться различные алгоритмы. Конкретные классы ConcreteStrategy реализуют эти различные алгоритмы. Класс Context использует конкретные классы ConcreteStrategy посредством ссылки на конкретный тип абстрактного класса Strategy. Классы Strategy и Context взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу Strategy требуется посылать запросы классу Context). Класс Context пересылает классу Strategy запрос, поступивший от его классаклиента.

Наблюдатель (Observer)

Наблюдатель — поведенческий шаблон проектирования. Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

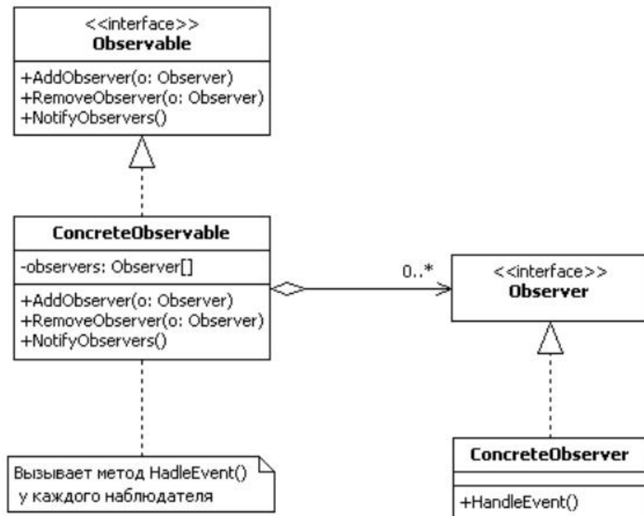
Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

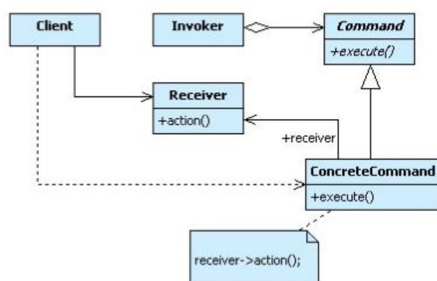
При реализации шаблона «наблюдатель» обычно используются следующие классы:

- **Observable** — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;
- **Observer** — интерфейс, с помощью которого наблюдатель получает оповещение;
- **ConcreteObservable** — конкретный класс, который реализует интерфейс Observable;
- **ConcreteObserver** — конкретный класс, который реализует интерфейс Observer.

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

- существует, как минимум, один объект, рассылающий сообщения;
- имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;
- нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.





Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

Команда (Command)

Команда — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Интерфейс командного объекта определяется абстрактным базовым классом Command и в самом простом случае имеет единственный метод execute(). Производные классы определяют получателя запроса (указатель на объект получатель) и необходимую для выполнения операцию (метод этого объекта). Метод execute() подклассов Command просто вызывает нужную операцию получателя.

Впаттерне Command может быть до трех участников:

●Клиент, создающий экземпляр командного объекта.

●Инициатор запроса, использующий командный объект.

●Получатель запроса.

Сначала клиент создает объект ConcreteCommand, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод execute(). Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании

– функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Задания для самостоятельной работы

Необходимо сделать следующее:

1.Нарисовать в PlantUML диаграмму классов реализуемой программы. (проектирование)

2.Реализовать программу на Java. (реализация)

Вариант №1, 9, 17, 25

Шаблон “Стратегия”. Проект “Принтеры”. В проекте должны быть реализованы разные модели принтеров, которые выполняют разные виды печати.

Вариант №2, 10, 18, 26

Шаблон “Наблюдатель”. Проект “Оповещение постов ГАИ”. В проекте должна быть реализована отправка сообщений всем постам ГАИ.

Вариант №3, 11, 19, 27

Шаблон “Декоратор”. Проект “Универсальная электронная карта”. В проекте должна быть реализована универсальная электронная карта, в которой есть функции паспорта, страхового полиса, банковской карты и т. д.

Вариант №4, 12, 20, 28

Шаблон “Фабричный метод”. Проект “Фабрика смартфонов”. В проекте должно быть реализовано создание смартфонов с различными характеристиками.

Вариант №5, 13, 21, 29

Шаблон “Абстрактная фабрика”. Проект “Заводы по производству автомобилей”. В проекте должно быть реализована возможность создавать автомобили различных типов на разных заводах.

Вариант №6, 14, 22, 30

Шаблон “Команда”. Проект “Клавиатура настраимаемого калькулятора”. Цифровые и арифметические кнопки имеют фиксированную функцию, а остальные могут менять своё назначение.

Вариант №7, 15, 23

Шаблон “Адаптер”. Проект “Часы”. В проекте должен быть реализован адаптер, который дает возможность пользоваться часами со стрелками так же, как и цифровыми часами. В классе “Часы со стрелками” хранятся повороты стрелок. Пример использования шаблона в главе 7.

Вариант №8, 16, 24

Шаблон “Фасад”. Проект “Компьютер”. В проекте должен быть реализован “компьютер”, который выполняет основные функции, к примеру, включение, выключение, запуск ОС, запуск программы, и т.д, не раскрывая клиенту деталей выполнения этой операции.

Контрольные вопросы

1. Для чего предназначены поведенческие паттерны проектирования?
2. Какие задачи решает шаблон «Адаптер»?
3. Какие классы входят в состав паттерна «Command», каковы их обязанности?

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К САМОСТОЯТЕЛЬНОЙ РАБОТЕ СТУДЕНТОВ

Самостоятельная работа развивает мотивационную составляющую образовательной деятельности студентов, акцентируясь на самообразовании и самовоспитании, осуществляемых в интересах повышения профессиональной компетенции. Она развивает систему общеучебных умений, способствующих ее рациональной организации:

- планировать собственную образовательную деятельность,
- четко ставить систему задач,
- вычленять среди них главные направления работы,
- избирать способы наиболее быстрого и экономного решения поставленных задач,
- осуществлять оперативный контроль за выполнением задания,
- оперативно вносить коррективы в самостоятельную работу, анализировать промежуточные и общие итоги работы,
- сравнивать полученные результаты с намеченными в начале работы целями, выявлять причины отклонений и определять пути их коррекции в дальнейшей работе.
-

Самостоятельная работа включает два вида – аудиторную и внеаудиторную. В первом случае она выполняется на учебных занятиях под руководством преподавателя и по его заданию. Студенты обеспечиваются необходимым учебным материалом и дидактическими материалами.

Внеаудиторная самостоятельная работа выполняется по заданию преподавателя, но без его непосредственного участия. Видами заданий для внеаудиторной работы являются: изучение текста учебной литературы, конспектирование текста, работа с конспектом лекции, ответы на контрольные вопросы при выполнении индивидуального задания, тестирование, решение задач, продумывание алгоритма будущей программы, работа с компьютером, а именно, кодирование и отладка программы, подготовка отчета по лабораторным заданиям, подготовка к сдаче экзамена.

Основные формы самостоятельной учебной работы:

работа над конспектом лекции: лекции – основной источник информации по многим предметам, позволяющий не только изучить материал, но и получить представление о наличии других источников, сопоставить разные взгляды на основные проблемы данного курса. Лекции предоставляют возможность «интерактивного» обучения, когда есть возможность задавать преподавателю вопросы и получать на них ответы. Поэтому имеет смысл находить время для хотя бы беглого просмотра информации по материалу лекций (учебники, справочники и пр.) и непонятные, а также дискуссионные моменты обсуждать с преподавателем, другими студентами;

подготовка к практическому занятию: производится, как правило, с использованием методических пособий, состоит в теоретической подготовке (особенно для семинаров) и выполнении практических заданий (решение задач, ответы на вопросы и т.д.).

Существует ряд форм практических занятий:

- лабораторные занятия с оборудованием (иногда с постановкой опытов);
- практикум по освоению тех или иных навыков, методик;
- семинар (с разбором теоретических вопросов в рамках какой-либо темы);
- коллоквиум (семинар но итогам изучения нескольких родственных тем);

подготовка к семинарскому занятию производится по правилам выполнения задания практической работы, обычно по определенному вопросу и более или менее узкому кругу литературы

(часто всего два-три учебных пособия);

доработка конспекта лекции с применением учебника, методической литературы, дополнительной литературы: этот вид самостоятельной работы студентов особенно важен в том случае, когда изучаемый предмет содержит много неоднозначно трактуемых вопросов, проблем. Тогда преподаватель заведомо не может успеть изложить различные точки зрения, и студент должен ознакомиться с ними по имеющейся литературе.

Кроме того, рабочая программа предметов предполагает рассмотрение некоторых относительно несложных тем только во время самостоятельных занятий, без чтения лектором; подбор, изучение, анализ и конспектирование рекомендованной литературы; самостоятельное изучение отдельных тем, параграфов; консультации по сложным, непонятным вопросам лекций, семинаров, зачетов;

подготовка к зачету: данная форма СРС может быть весьма разнообразной по своей сути, так как сам зачет бывает различным. Он проводится обычно по итогам семестра перед сессией в письменной или устной форме, причем преподаватель может включать в него вопросы как практических занятий, так и лекционных (что особенно уместно, когда по данному предмету не сдается экзамен).

Главное отличие зачета от экзамена – почти всегда не пяти-, а двухбалльная система оценки (сдал – не сдал), что делает его получение несколько более простым делом. С другой стороны, порой процедура его сдачи достаточно сложна, а иногда применяется и пятибалльная оценка (так называемый дифференцированный зачет).

Таким образом, для сдачи зачета необходимо, прежде всего, выполнить все требования преподавателя, что предполагает знание этих требований. Нужно как можно раньше выяснить, какие вопросы предстоит готовить и каковы правила самой процедуры (учитывается ли посещаемость, надо ли пропущенные занятия отрабатывать, а если надо, то каким образом и т.д.). Практика показывает, что хорошее посещение занятий является почти полной гарантией получения зачета, так как тогда можно быть в курсе всех требований преподавателя. И, напротив, большое количество пропусков может осложнить жизнь даже сильному студенту.

Кроме того, необходимо учитывать, что проблемы могут появиться при распространенном подходе студента к практическим занятиям, когда многие работают первые месяцы вполсилы, накапливая задолженности по выполнению рефератов, практических заданий, конспектов и пр., а перед сессией пытаются все это сделать за одну неделю. Старайтесь распределять силы равномерно по всей дистанции семестра, и тогда зачетная неделя перед сессией будет не самой напряженной, а самой разгрузочной.

ЛИТЕРАТУРА

1. Рамбо Дж.UML 2.0. Объектно-ориентированное моделирование и разработка / Дж. Рамбо, М. Блаха.– М. и др. : «Питер», 2007.– 544 с.

2. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.– СПб.: «Питер», 2009.– 366 с.

3. Трофимов, В. В. Основы алгоритмизации и программирования : учебник для среднего профессионального образования / В. В. Трофимов, Т. А. Павловская ; под редакцией В. В. Трофимова. — Москва : Издательство Юрайт, 2019. — 137 с. — (Профессиональное образование). — ISBN 978-5-534-07321-8. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://www.biblio-online.ru/bcode/441286

4. Черткова, Е. А. Программная инженерия. Визуальное моделирование программных систем : учебник для среднего профессионального образования / Е. А. Черткова. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 147 с. — (Профессиональное образование). — ISBN 978-5-534-09823-5. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://www.biblio-online.ru/bcode/441255

5. Зубкова, Т. М. Технология разработки программного обеспечения : учебное пособие для СПО / Т. М. Зубкова. — Саратов : Профобразование, 2019. — 468 с. — ISBN 978-5-4488-0354-3. —

Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/86208.html

6. Гниденко, И. Г. Технология разработки программного обеспечения : учебное пособие для среднего профессионального образования / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — Москва : Издательство Юрайт, 2019. — 235 с. — (Профессиональное образование). — ISBN 978-5-534-05047-9. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://www.biblio-online.ru/bcode/438444

7. Кувшинов, Д. Р. Основы программирования : учебное пособие для среднего профессионального образования / Д. Р. Кувшинов. — Москва : Издательство Юрайт, 2019 ; Екатеринбург : Изд-во Урал. ун-та. — 105 с. — (Профессиональное образование). — ISBN 978-5-534-07560-1 (Издательство Юрайт). — ISBN 978-5-7996-1411-9 (Изд-во Урал. ун-та). — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://www.biblio-online.ru/bcode/441571

8. Токманцев, Т. Б. Алгоритмические языки и программирование : учебное пособие для СПО / Т. Б. Токманцев ; под редакцией В. Б. Костоусова. — 2-е изд. — Саратов, Екатеринбург : Профобразование, Уральский федеральный университет, 2019. — 102 с. — ISBN 978-5-4488-0510-3, 978-5-7996-2899-4. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/87785.html

СОДЕРЖАНИЕ

КРАТКОЕ ИЗЛОЖЕНИЕ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА 3

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ 55

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ 63

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К САМОСТОЯТЕЛЬНОЙ РАБОТЕ СТУДЕНТОВ 71

ЛИТЕРАТУРА 72