Планирование проектных задач

Основной задачей при планировании является определение WBS — Work Breakdown Structure (структуры распределения работ). Она составляется с помощью утилиты планирования проекта. Типовая WBS приведена на рис. 2.2.

Первыми выполняемыми задачами являются системный анализ и анализ требований. Они закладывают фундамент для последующих параллельных задач.

Системный анализ проводится с целью:

1) выяснения потребностей заказчика;
2) оценки выполнимости системы;
3) выполнения экономического и технического анализа;
4) распределения функций по элементам компьютерной системы (аппаратуре, программам, людям, базам данных и т. д.);
5) определения стоимости и ограничений планирования;
6) создания системной спецификации.

В *системной спецификации* описываются функции, характеристики системы, ограничения разработки, входная и выходная информация.

Анализ требований дает возможность:

1) определить функции и характеристики программного продукта;
2) обозначить интерфейс продукта с другими системными элементами;
3) определить проектные ограничения программного продукта;
4) построить модели: процесса, данных, режимов функционирования продукта;
5) создать такие формы представления информации и функций системы, которые можно использовать в ходе проектирования.

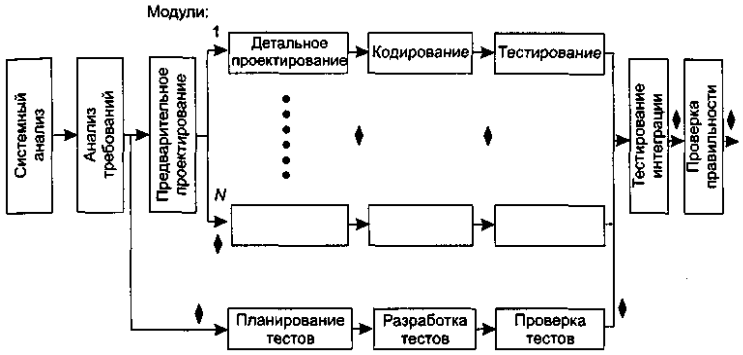


Рис. 2.2. Типовая структура распределения проектных работ

Результаты анализа сводятся в *спецификацию требований* к программному продукту.

Как видно из типовой структуры, задачи по проектированию и планированию тестов могут быть распараллелены. Благодаря модульной природе ПО для каждого модуля можно предусмотреть параллельный путь для детального (процедурного) проектирования, кодирования и тестирования. После получения всех модулей ПО решается задача тестирования интеграции — объединения элементов в единое целое. Далее проводится тестирование правильности, которое обеспечивает проверку соответствия ПО требованиям заказчика.

Ромбиками на рис. 2.2 обозначены вехи — процедуры контроля промежуточных результатов. Очень важно, чтобы вехи были расставлены через регулярные интервалы (вдоль всего процесса разработки ПО). Это даст руководителю возможность регулярно получать информацию о текущем положении дел. Вехи распространяются и на документацию как на один из результатов успешного решения задачи.

Параллельность действий повышает требования к планированию. Так как параллельные задачи выполняются асинхронно, планировщик должен определить межзадачные зависимости. Это гарантирует «непрерывность движения к объединению». Кроме того, руководитель проекта должен знать задачи, лежащие на критическом пути. Для того чтобы весь проект был выполнен в срок, необходимо выполнять в срок все критические задачи.

Основной рычаг в планирующих методах — вычисление границ времени выполнения задачи.

Обычно используют следующие оценки:

1. Раннее время начала решения задачи $T_{\min}^{in}$ (при условии, что все предыдущие задачи решены в кратчайшее время).

2. Позднее время начала решения задачи $T_{\max}^{in}$ (еще не вызывает общую задержку проекта).

3. Раннее время конца решения задачи $T_{\min}^{out}$.

$$T_{\min}^{out} + T_{\min}^{in} + T_{реш}.$$

4. Позднее время конца решения задачи $T_{\max}^{out}$.

$$T_{\max}^{out} + T_{\max}^{in} + T_{реш}.$$

5. Общий резерв — количество избытков и потерь планирования задач во времени, *не* приводящих к увеличению

длительности критического пути $T_{к.п}$.

Все эти значения позволяют руководителю (планировщику) количественно оценить успех в планировании, выполнении задач.

Рекомендуемое правило распределения затрат проекта — 40-20-40:

- на анализ и проектирование приходится 40% затрат (из них на планирование и системный анализ — 5%);
- на кодирование — 20%;
- на тестирование и отладку — 40%.

Размерно-ориентированные метрики

Размерно-ориентированные метрики прямо измеряют программный продукт и процесс его разработки. Основываются размерно-ориентированные метрики на LOC-оценках (Lines Of Code). LOC-оценка — это количество строк в программном продукте.

Исходные данные для расчета этих метрик сводятся в таблицу (табл. 2.1).

Таблица 2.1. Исходные данные для расчета LOC-метрик

Проект	Затраты, чел.-мес	Стоимость, тыс. $	KLOC, тыс. LOC	Прогр. док- ты, страниц	Ошибки	Люди
aaa01	24	168	12,1	365	29	3
bbb02	62	440	27,2	1224	86	5
ccc03	43	314	20,2	1050	64	6

Таблица содержит данные о проектах за последние несколько лет. Например, запись о проекте aaa01 показывает: 12 100 строк программы были разработаны за 24 человеко-месяца и стоили $168 000. Кроме того, по проекту aaa01 было разработано 365 страниц документации, а в течение первого года эксплуатации было зарегистрировано 29 ошибок. Разрабатывали проект aaa01 три человека.

На основе таблицы вычисляются размерно-ориентированные метрики производительности и качества (для каждого проекта):

$$\text{Производительность} = \frac{\text{Длина}}{\text{Затраты}}\left[\frac{\text{тыс.LOC}}{\text{чел.-мес}}\right];$$

$$\text{Качество} = \frac{\text{Ошибки}}{\text{Длина}}\left[\frac{\text{Единиц}}{\text{тыс.LOC}}\right];$$

$$\text{Удельная стоимость} = \frac{\text{Стоимость}}{\text{Длина}}\left[\frac{\text{Тыс\$}}{\text{LOC}}\right];$$

$$\text{Документированность} = \frac{\text{Страниц Документа}}{\text{Длина}}\left[\frac{\text{Страниц}}{\text{тыс.LOC}}\right].$$

Достоинства размерно-ориентированных метрик:

1) широко распространены;
2) просты и легко вычисляются.

Недостатки размерно-ориентированных метрик:

1) зависимы от языка программирования;
2) требуют исходных данных, которые трудно получить на начальной стадии проекта;
3) не приспособлены к непроцедурным языкам программирования.

Функционально-ориентированные метрики

Функционально-ориентированные метрики косвенно измеряют программный продукт и процесс его разработки. Вместо подсчета LOC-оценки при этом рассматривается не размер, а функциональность или полезность продукта.

Используется 5 информационных характеристик.

1. *Количество внешних вводов.* Подсчитываются все вводы пользователя, по которым поступают разные прикладные данные. Вводы должны быть отделены от запросов, которые подсчитываются отдельно.
2. *Количество внешних выводов.* Подсчитываются все выводы, по которым к пользователю поступают результаты, вычисленные программным приложением. В этом контексте выводы означают отчеты, экраны, распечатки, сообщения об ошибках. Индивидуальные единицы данных внутри отчета отдельно не подсчитываются.
3. *Количество внешних запросов.* Под запросом понимается диалоговый ввод, который приводит к немедленному программному ответу в форме диалогового вывода. При этом диалоговый ввод в приложении не сохраняется, а диалоговый вывод не требует выполнения вычислений. Подсчитываются все запросы — каждый учитывается отдельно.
4. *Количество внутренних логических файлов.* Подсчитываются все логические файлы (то есть логические группы данных, которые могут быть частью базы данных или отдельным файлом).
5. *Количество внешних интерфейсных файлов.* Подсчитываются все логические файлы из других приложений, на которые ссылается данное приложение.

Вводы, выводы и запросы относят к категории *транзакция.* Транзакция — это элементарный процесс, различаемый

пользователем и перемещающий данные между внешней средой и программным приложением. В своей работе транзакции используют внутренние и внешние файлы. Приняты следующие определения.

Внешний ввод — элементарный процесс, перемещающий данные из внешней среды в приложение. Данные могут поступать с экрана ввода или из другого приложения. Данные могут использоваться для обновления внутренних логических файлов. Данные могут содержать как управляющую, так и деловую информацию. Управляющие данные не должны модифицировать внутренний логический файл.

Внешний вывод — элементарный процесс, перемещающий данные, вычисленные в приложении, во внешнюю среду. Кроме того, в этом процессе могут обновляться внутренние логические файлы. Данные создают отчеты или выходные файлы, посылаемые другим приложениям. Отчеты и файлы создаются на основе внутренних логических файлов и внешних интерфейсных файлов. Дополнительно этот процесс может использовать вводимые данные, их образуют критерии поиска и параметры, не поддерживаемые внутренними логическими файлами. Вводимые данные поступают извне, но носят временный характер и не сохраняются во внутреннем логическом файле.

Внешний запрос — элементарный процесс, работающий как с вводимыми, так и с выводимыми данными. Его результат — данные, возвращаемые из внутренних логических файлов и внешних интерфейсных файлов. Входная часть процесса не модифицирует внутренние логические файлы, а выходная часть не несет данных, вычисляемых приложением (в этом и состоит отличие запроса от вывода).

Внутренний логический файл — распознаваемая пользователем группа логически связанных данных, которая размещена внутри приложения и обслуживается через внешние вводы.

Внешний интерфейсный файл — распознаваемая пользователем группа логически связанных данных, которая размещена внутри другого приложения и поддерживается им. Внешний файл данного приложения является внутренним логическим файлом в другом приложении.

Каждой из выявленных характеристик ставится в соответствие сложность. Для этого характеристике назначается низкий, средний или высокий ранг, а затем формируется числовая оценка ранга.

Для транзакций ранжирование основано на количестве ссылок на файлы и количестве типов элементов данных. Для файлов ранжирование основано на количестве типов элементов-записей и типов элементов данных, входящих в файл.

Тип элемента-записи — подгруппа элементов данных, распознаваемая пользователем в пределах файла.

Тип элемента данных — уникальное не рекурсивное (неповторяемое) поле, распознаваемое пользователем. В качестве примера рассмотрим табл. 2.2.

В этой таблице 10 элементов данных: День, Хиты, % от Сумма хитов, Сеансы пользователя, Сумма хитов (по рабочим дням), % от Сумма хитов (по рабочим дням), Сумма сеансов пользователя (по рабочим дням), Сумма хитов (по выходным дням), % от Сумма хитов (по выходным дням), Сумма сеансов пользователя (по выходным дням). Отметим, что поля День, Хиты, % от Сумма хитов, Сеансы пользователя имеют рекурсивные данные, которые в расчете не учитываются.

Таблица 2.2. Пример для расчета элементов данных

Уровень активности дня недели			
День	**Хиты**	**% от Сумма хитов**	**Сеансы пользователя**
Понедельник	1887	16,41	201
Вторник	1547	13,45	177
Среда	1975	17,17	195
Четверг	1591	13,83	191
Пятница	2209	19,21	200
Суббота	1286	11,18	121
Воскресенье	1004	8,73	111
Сумма по рабочим дням	9209	80,08	964
Сумма по выходным дням	2290	19,91	232

Примеры элементов данных для различных характеристик приведены в табл. 2.3, а табл. 2.4 содержит правила учета элементов данных из графического интерфейса пользователя (GUI).

Таблица 2.3. Примеры элементов данных

Информационная характеристика	**Элементы данных**
Внешние Вводы	Поля ввода данных, сообщения об ошибках, вычисляемые значения, кнопки
Внешние Выводы	Поля данных в отчетах, вычисляемые значения, сообщения об ошибках, заголовки столбцов, которые читаются из внутреннего файла
Внешние Запросы	Вводимые элементы: поле, используемое для поиска, щелчок мыши. Выводимые элементы — отображаемые на экране поля

Таблица 2.4. Правила учета элементов данных из графического интерфейса пользователя

Элемент данных	Правило учета
Группа радиокнопок	Так как в группе пользователь выбирает только одну радиокнопку, все радиокнопки группы считаются одним элементом данных
Группа флажков	Так как в группе пользователь может выбрать несколько флажков, каждый флажок считают

(переключателей)	элементом данных
Командные кнопки	Командная кнопка может определять действие добавления, изменения, запроса. Кнопка ОК может вызывать транзакции (различных типов). Кнопка Next может быть входным элементом запроса или вызывать другую транзакцию. Каждая кнопка считается отдельным элементом данных
Списки	Список может быть внешним запросом, но результат запроса может быть элементом данных внешнего ввода

Например, GUI для обслуживания клиентов может иметь поля Имя, Адрес, Город, Страна, Почтовый Индекс, Телефон, Email. Таким образом, имеется 7 полей или семь элементов данных. Восьмым элементом данных может быть командная кнопка (добавить, изменить, удалить). В этом случае каждый из внешних вводов Добавить, Изменить, Удалить будет состоять из 8 элементов данных (7 полей плюс командная кнопка).

Обычно одному экрану GUI соответствует несколько транзакций. Типичный экран включает несколько внешних запросов, сопровождающих внешний ввод.

Обсудим порядок учета сообщений. В приложении с GUI генерируются 3 типа сообщений: сообщения об ошибке, сообщения подтверждения и сообщения уведомления. Сообщения об ошибке (например, Требуется пароль) и сообщения подтверждения (например, Вы действительно хотите удалить клиента?) указывают, что произошла ошибка или что процесс может быть завершен. Эти сообщения не образуют самостоятельного процесса, они являются частью другого процесса, то есть считаются элементом данных соответствующей транзакции.

С другой стороны, уведомление является независимым элементарным процессом. Например, при попытке получить из банкомата сумму денег, превышающую их количество на счете, генерируется сообщение Не хватает средств для завершения транзакции. Оно является результатом чтения информации из файла счета и формирования заключения. Сообщение уведомления рассматривается как внешний вывод.

Данные для определения ранга и оценки сложности транзакций и файлов приведены в табл. 2.5-2.9 (числовая оценка указана в круглых скобках). Использовать их очень просто. Например, внешнему вводу, который ссылается на 2 файла и имеет 7 элементов данных, по табл. 2.5 назначается средний ранг и оценка сложности 4.

Таблица 2.5. Ранг и оценка сложности внешних вводов

Ссылки на файлы	**Элементы данных**		
	1-4	**5-15**	**>15**
0-1	Низкий (3)	Низкий (3)	Средний (4)
2	Низкий (3)	Средний (4)	Высокий (6)
>2	Средний (4)	Высокий (6)	Высокий (6)

Таблица 2.6. Ранг и оценка сложности внешних выводов

Ссылки на файлы	**Элементы данных**		
	1-4	**5-19**	**>19**
0-1	Низкий (4)	Низкий (4)	Средний (5)
2-3	Низкий (4)	Средний (5)	Высокий (7)
>3	Средний (5)	Высокий (7)	Высокий (7)

Таблица 2.7. Ранг и оценка сложности внешних запросов

Ссылки на файлы	**Элементы данных**		
	1-4	**5-19**	**>19**
0-1	Низкий (3)	Низкий (3)	Средний (4)
2-3	Низкий (3)	Средний (4)	Высокий (6)
>3	Средний (4)	Высокий (6)	Высокий (6)

Таблица 2.8. Ранг и оценка сложности внутренних логических файлов

Типы элементов-записей	**Элементы данных**		
	1-19	**20-50**	**>50**
1	Низкий (7)	Низкий (7)	Средний (10)
2-5	Низкий (7)	Средний (10)	Высокий (15)
>5	Средний (10)	Высокий (15)	Высокий (15)

Таблица 2.9. Ранг и оценка сложности внешних интерфейсных файлов

Типы элементов-записей	**Элементы данных**		
	1-19	**20-50**	**>50**

1	Низкий (5)	Низкий (5)	Средний (7)
2-5	Низкий (5)	Средний (7)	Высокий (10)
>5	Средний (7)	Высокий (10)	Высокий (10)

Отметим, что если во внешнем запросе ссылка на файл используется как на этапе ввода, так и на этапе вывода, она учитывается только один раз. Такое же правило распространяется и на элемент данных (однократный учет).

После сбора всей необходимой информации приступают к расчету метрики — *количества функциональных указателей FP* (Function Points). Автором этой метрики является А. Албрехт (1979) [7].

Исходные данные для расчета сводятся в табл. 2.10.

Таблица 2.10. Исходные данные для расчета FP-метрик

Имя характеристики	**Ранг, сложность, количество**			
	Низкий	**Средний**	**Высокий**	**Итого**
Внешние вводы	0x3 = __	0x4 = __	0x6 = __	= 0
Внешние выводы	0x4 = __	0x5 = __	0x7 = __	= 0
Внешние запросы	0x3 = __	0x4 = __	0x6 = __	= 0
Внутренние логические файлы	0x7 = __	0x 10= __	0x15 = __	= 0
Внешние интерфейсные файлы	0x5 = __	0x7 = __	0x10 = __	= 0
	Общее количество			= 0

В таблицу заносится количественное значение характеристики каждого вида (по всем уровням сложности). Места подстановки значений отмечены прямоугольниками (прямоугольник играет роль метки-заполнителя). Количественные значения характеристик умножаются на числовые оценки сложности. Полученные в каждой строке значения суммируются, давая полное значение для данной характеристики. Эти полные значения затем суммируются по вертикали, формируя общее количество.

Количество функциональных указателей вычисляется по формуле

$$\text{FP} = \text{Общее количество} \times (0{,}65 + 0{,}01 \times \sum_{i=1}^{14} F_i), \qquad (2.1)$$

где F_i — коэффициенты регулировки сложности.

Каждый коэффициент может принимать следующие значения: 0 — нет влияния, 1 — случайное, 2 — небольшое, 3 — среднее, 4 — важное, 5 — основное.

Значения выбираются эмпирически в результате ответа на 14 вопросов, которые характеризуют системные параметры приложения (табл. 2.11).

Таблица 2.11. Определение системных параметров приложения

№	**Системный параметр**	**Описание**
1	Передачи данных	Сколько средств связи требуется для передачи или обмена информацией с приложением или системой?
2	Распределенная обработка данных	Как обрабатываются распределенные данные и функции обработки?
3	Производительность	Нуждается ли пользователь в фиксации времени ответа или производительности?.
4	Распространенность используемой конфигурации	Насколько распространена текущая аппаратная платформа, на которой будет выполняться приложение?
5	Скорость транзакций	Как часто выполняются транзакции? (каждый день, каждую неделю, каждый месяц)
6	Оперативный ввод данных	Какой процент информации надо вводить в режиме онлайн?
7	Эффективность работы конечного пользователя	Приложение проектировалось для обеспечения эффективной работы конечного пользователя?
8	Оперативное обновление	Как много внутренних файлов обновляется в онлайновой транзакции?
9	Сложность обработки	Выполняет ли приложение интенсивную логическую или математическую обработку?
10	Повторная используемость	Приложение разрабатывалось для удовлетворения требований одного или многих пользователей?
11	Легкость инсталляции	Насколько трудны преобразование и инсталляция приложения?
12	Легкость эксплуатации	Насколько эффективны и/или автоматизированы процедуры запуска, резервирования и восстановления?

13	Разнообразные условия размещения	Была ли спроектирована, разработана и поддержана возможность инсталляции приложения в разных местах для различных организаций?
14	Простота изменений	Была ли спроектирована, разработана и поддержана в приложении простота изменений?

После вычисления FP на его основе формируются метрики производительности, качества и т. д.:

$$\text{Производитель} = \frac{\text{ФункцУказатель}}{\text{Затраты}}\left[\frac{\text{FP}}{\text{чел.}-\text{мес}}\right];$$

$$\text{Качество} = \frac{\text{Ошибки}}{\text{ФункцУказатель}}\left[\frac{\text{Единиц}}{\text{FP}}\right];$$

$$\text{Удельная стоимость} = \frac{\text{Стоимость}}{\text{ФункцУказатель}}\left[\frac{\text{Тыс.\$}}{\text{FP}}\right];$$

$$\text{Документированность} = \frac{\text{СтраницДокумента}}{\text{ФункцУказатель}}\left[\frac{\text{Страниц}}{\text{FP}}\right].$$

Область применения метода функциональных указателей — коммерческие информационные системы. Для продуктов с высокой алгоритмической сложностью используются метрики *указателей свойств* (Features Points). Они применимы к системному и инженерному ПО, ПО реального времени и встроенному ПО.

Для вычисления указателя свойств добавляется одна характеристика — *количество алгоритмов.* Алгоритм здесь определяется как ограниченная подпрограмма вычислений, которая включается в общую компьютерную программу. Примеры алгоритмов: обработка прерываний, инвертирование матрицы, расшифровка битовой строки. Для формирования указателя свойств составляется табл. 2.12.

Таблица 2.12. Исходные данные для расчета указателя свойств

№	Характеристика	Количество	Сложность	Итого
1	Вводы	0	x4	= 0
2	Выводы	0	x5	= 0
3	Запросы	0	x4	= 0
4	Логические файлы	0	x7	= 0
5	Интерфейсные файлы	0	x7	= 0
6	Количество алгоритмов	0	x3	= 0
		Общее количество		= 0

После заполнения таблицы по формуле (2.1) вычисляется значение указателя свойств. Для сложных систем реального времени это значение на 25-30% больше значения, вычисляемого по таблице для количества функциональных указателей.

Достоинства функционально-ориентированных метрик:

1. Не зависят от языка программирования.
2. Легко вычисляются на любой стадии проекта.

Недостаток функционально-ориентированных метрик: результаты основаны на субъективных данных, используются не прямые, а косвенные измерения. FP-оценки легко пересчитать в LOC-оценки. Как показано в табл. 2.13, результаты пересчета зависят от языка программирования, используемого для реализации ПО.

Таблица 2.13. Пересчет FP-оценок в LOC-оценки

Язык программирования	Количество операторов на один FP
Ассемблер	320
C	128
Кобол	106
Фортран	106
Паскаль	90
C++	64
Java	53
Ada 95	49
Visual Basic	32
Visual C++	34
Delphi Pascal	29
Smalltalk	22
Perl	21
HTML3	15

LISP	64
Prolog	64
Miranda	40
Haskell	38

Выполнение оценки в ходе руководства проектом

Процесс руководства программным проектом начинается с множества действий, объединяемых общим названием *планирование проекта.* Первое из этих действий — выполнение оценки. Оно закладывает фундамент для других действий по планированию проекта. При оценке проекта чрезвычайно высока цена ошибок. Очень важно провести оценку с минимальным риском.

Выполнение оценки проекта на основе LOC- и FP-метрик

Цель этой деятельности — сформировать предварительные оценки, которые позволят:
- предъявить заказчику корректные требования по стоимости и затратам на разработку программного продукта;
- составить план программного проекта.

При выполнении оценки возможны два варианта использования LOC- и FP-данных:
- в качестве оценочных переменных, определяющих размер каждого элемента продукта;
- в качестве метрик, собранных за прошлые проекты и входящих в метрический базис фирмы.

Обсудим шаги процесса оценки.
- *Шаг 1.* Область назначения проектируемого продукта разбивается на ряд функций, каждую из которых можно оценить индивидуально:

$$f_1, f_2, \ldots, f_n.$$

- *Шаг 2.* Для каждой функции f_i, планировщик формирует лучшую $LOC_{лучшi}$ ($FP_{лучшi}$), худшую $LOC_{худшi}$ ($FP_{худшi}$) и вероятную оценку $LOC_{вероятнi}$ ($FP_{вероятнi}$). Используются опытные данные (из метрического базиса) или интуиция. Диапазон значения оценок соответствует степени предусмотренной неопределенности.
- *Шаг 3.* Для каждой функции/ в соответствии с β-распределением вычисляется ожидаемое значение LOC- (или FP-) оценки:

$$LOC_{ожi} = (LOC_{лучшi} + LOC_{худшi} + 4x\ LOC_{вероятнi})/6.$$

- *Шаг 4.* Определяется значение LOC- или FP-производительности разработки функции.

Используется один из трех подходов:
1) для всех функций принимается одна и та же метрика средней производительности $ПРОИЗВ_{ср}$, взятая из метрического базиса;
2) для *i*-й функции на основе метрики средней производительности вычисляется настраиваемая величина производительности:

$$ПРОИЗВ_i = ПРОИЗВ_{ср} x (LOC_{ср} / LOC_{ожi}),$$

где $LOC_{ср}$ — средняя LOC-оценка, взятая из метрического базиса (соответствует средней производительности);
3) для *i*-й функции настраиваемая величина производительности вычисляется по аналогу, взятому из метрического базиса:

$$ПРОИЗВ_i = ПРОИЗВ_{анi} x (LOC_{анi} / LOC_{ожi}).$$

Первый подход обеспечивает минимальную точность (при максимальной простоте вычислений), а третий подход — максимальную точность (при максимальной сложности вычислений).
- *Шаг* 5. Вычисляется общая оценка затрат на проект: для первого подхода

$$ЗАТРАТЫ = \left(\sum_{i=1}^{n} LOC_{ожi}\right) / ПРОИЗВ_{ср}\ [чел.\text{-}мес];$$

для второго и третьего подходов

$$ЗАТРАТЫ = \sum_{i=1}^{n} (LOC_{ожi} / ПРОИЗВ_i)\ [чел.\text{-}мес]\cdot$$

- *Шаг 6.* Вычисляется общая оценка стоимости проекта: для первого и второго подходов

$$СТОИМОСТЬ = \left(\sum_{i=1}^{n} LOC_{ожi}\right) \times УД\_СТОИМОСТЬ_{ср},$$

где $УД\_СТОИМОСТЬ_{ср}$ — метрика средней стоимости одной строки, взятая из метрического базиса.
для третьего подхода

$$СТОИМОСТЬ = \sum_{i=1}^{n} (LOC_{ожi} \times УД\_СТОИМОСТЬ_{анi}),$$

где $УД\_СТОИМОСТЬ_{анi}$ — метрика стоимости одной строки аналога, взятая из метрического базиса. Пример применения данного процесса оценки приведем ниже.

Конструктивная модель стоимости

В данной модели для вывода формул использовался статистический подход — учитывались реальные результаты огромного количества проектов. Автор оригинальной модели — Барри Боэм (1981) —дал ей название COCOMO 81 (Constructive Cost Model) и ввел в ее состав три разные по сложности статистические подмодели [1].

Иерархию подмоделей Боэма (версии 1981 года) образуют:

- базисная COCOMO — статическая модель, вычисляет затраты разработки и ее стоимость как функцию размера программы;
- промежуточная COCOMO — дополнительно учитывает атрибуты стоимости, включающие основные оценки продукта, аппаратуры, персонала и проектной среды;
- усовершенствованная COCOMO — объединяет все характеристики промежуточной модели, дополнительно учитывает влияние всех атрибутов стоимости на каждый этап процесса разработки ПО (анализ, проектирование, кодирование, тестирование и т. д.).

Подмодели COCOMO 81 могут применяться к трем типам программных проектов. По терминологии Боэма, их образуют:

- *распространенный тип* — небольшие программные проекты, над которыми работает небольшая группа разработчиков с хорошим стажем работы, устанавливаются мягкие требования к проекту;
- *полунезависимый тип* — средний по размеру проект, выполняется группой разработчиков с разным опытом, устанавливаются как мягкие, так и жесткие требования к проекту;
- *встроенный тип* — программный проект разрабатывается в условиях жестких аппаратных, программных и вычислительных ограничений.

Уравнения базовой подмодели имеют вид

$$E=a_b \times (\mathrm{KLOC})^{b_b} \text{ [чел-мес]};$$

$$D = c_b \times (E)^{d_b} \text{ [мес]},$$

где E — затраты в человеко-месяцах, D — время разработки, KLOC — количество строк в программном продукте.

Коэффициенты a_b, b_b, c_b, d_b берутся из табл. 2.14.

Таблица 2.14. Коэффициенты для базовой подмодели COCOMO 81

Тип проекта	a_b	b_b	c_b	d_b
Распространенный	2,4	1,05	2,5	0,38
Полунезависимый	3,0	1,12	2,5	0,35
Встроенный	3,6	1,20	2,5	0,32

В 1995 году Боэм ввел более совершенную модель COCOMO II, ориентированную на применение в программной инженерии XXI века [21].

В состав COCOMO II входят:

- модель композиции приложения;
- модель раннего этапа проектирования;
- модель этапа пост-архитектуры.

Для описания моделей COCOMO II требуется информация о размере программного продукта. Возможно использование LOC-оценок, объектных указателей, функциональных указателей.

Модель композиции приложения

Модель композиции используется на ранней стадии конструирования ПО, когда:

- рассматривается макетирование пользовательских интерфейсов;
- обсуждается взаимодействие ПО и компьютерной системы;
- оценивается производительность;
- определяется степень зрелости технологии.

Модель композиции приложения ориентирована на применение объектных указателей.

Объектный указатель — средство косвенного измерения ПО, для его расчета определяется количество экранов (как элементов пользовательского интерфейса), отчетов и компонентов, требуемых для построения приложения. Как показано в табл. 2.15, каждый объектный экземпляр (экран, отчет) относят к одному из трех уровней сложности. Здесь места подстановки измеренных и вычисленных значений отмечены прямоугольниками (прямоугольник играет роль метки-заполнителя). В свою очередь, сложность является функцией от параметров клиентских и серверных таблиц данных (см. табл. 2.16 и 2.17), которые требуются для генерации экрана и отчета, а также от количества представлений и секций, входящих в экран или отчет.

Таблица 2.15. Оценка количества объектных указателей

Тип объекта	Количество	Вес			Итого
		Простой	Средний	Сложный	
Экран	0	x1	x2	x3	= 0
Отчет	0	x2	x5	x8	= 0
3GL компонент	0			x10	= 0

Объектные указатели	= 0

Таблица 2.16. Оценка сложности экрана

Экраны	**Количество серверных (срв) и клиентских (клт) таблиц данных**		
Количество представлений	Всего < 4 (< 2 срв, <3клт)	Всего < 8 (2-3 срв, 3-5 клт)	Всего > 8 (>3срв, >5клт)
<3	Простой	Простой	Средний
3-7	Простой	Средний	Сложный
>8	Средний	Сложный	Сложный

Таблица 2.17. Оценка сложности отчета

Отчеты	**Количество серверных (срв) и клиентских (клт) таблиц данных**		
Количество представлений	Всего < 4 (< 2 срв, < 3 клт)	Всего < 8 (2-3 срв, 3-5 клт)	Всего > 8 (>3срв, > 5 клт)
0 или 1	Простой	Простой	Средний
2 или 3	Простой	Средний	Сложный
>4	Средний	Сложный	Сложный

После определения сложности количество экранов, отчетов и компонентов взвешивается в соответствии с табл. 2.15. Количество объектных указателей определяется перемножением исходного числа объектных экземпляров на весовые коэффициенты и последующим суммированием промежуточных результатов.

Для учета реальных условий разработки вычисляется процент повторного использования программных компонентов %REUSE и определяется количество новых объектных указателей NOP:

NOP = (Объектные указатели) x [(100 - %REUSE) /100].

Для оценки затрат, основанной на величине NOP, надо знать скорость разработки продукта PROD. Эту скорость определяют по табл. 2.18, учитывающей уровень опытности разработчиков и зрелость среды разработки.

Проектные затраты оцениваются по формуле

ЗАТРАТЫ = NOP /PROD [чел.-мес],

где PROD — производительность разработки, выраженная в терминах объектных указателей.

Таблица 2.18. Оценка скорости разработки

Опытность / возможности разработчика	**Зрелость / возможности среды разработки**	**PROD**
Очень низкая	Очень низкая	4
Низкая	Низкая	7
Номинальная	Номинальная	13
Высокая	Высокая	25
Очень высокая	Очень высокая	50

В более развитых моделях дополнительно учитывается множество масштабных факторов, формирователей затрат, процедур поправок.

Модель раннего этапа проектирования

Модель раннего этапа проектирования используется в период, когда стабилизируются требования и определяется базисная программная архитектура.

Основное уравнение этой модели имеет следующий вид:

$$ЗАТРАТЫ = A \times РАЗМЕР^{B} \times M_e + ЗАТРАТЫ_{auto} [чел.-мес],$$

где:

- масштабный коэффициент $A = 2{,}5$;
- показатель B отражает нелинейную зависимость затрат от размера проекта (размер системы РАЗМЕР выражается в тысячах LOC);
- множитель поправки M_e зависит от 7 формирователей затрат, характеризующих продукт, процесс и персонал;
- слагаемое $ЗАТРАТЫ_{auto}$ отражает затраты на автоматически генерируемый программный код.

Значение показателя степени B изменяется в диапазоне 1,01... 1,26, зависит от пяти масштабных факторов W_i и вычисляется по формуле

$$B = 1{,}01 + 0{,}01\sum_{i=1}^{5} W_i .$$

Общая характеристика масштабных факторов приведена в табл. 2.19, а табл. 2.20 позволяет определить оценки этих факторов. Оценки принимают 6 значений: от очень низкой (5) до сверхвысокой (0).

Таблица 2.19. Характеристика масштабных факторов

Масштабный фактор (W_i)	**Пояснение**

Предсказуемость PREC	Отражает предыдущий опыт организации в реализации проектов этого типа. Очень низкий означает отсутствие опыта. Сверхвысокий означает, что организация полностью знакома с этой прикладной областью
Гибкость разработки FLEX	Отражает степень гибкости процесса разработки. Очень низкий означает, что используется заданный процесс. Сверхвысокий означает, что клиент установил только общие цели
Разрешение архитектуры /риска RESL	Отражает степень выполняемого анализа риска. Очень низкий означает малый анализ. Сверхвысокий означает полный и сквозной анализ риска
Связность группы TEAM	Отражает, насколько хорошо разработчики группы знают друг друга и насколько удачно они совместно работают. Очень низкий означает очень трудные взаимодействия. Сверхвысокий, означает интегрированную группу, без проблем взаимодействия
Зрелость процесса PMAT	Означает зрелость процесса в организации. Вычисление этого фактора может выполняться по вопроснику CMM

В качестве иллюстрации рассмотрим компанию, которая берет проект в малознакомой проблемной области. Положим, что заказчик не определил используемый процесс разработки и не допускает выделения времени на всесторонний анализ риска. Для реализации этой программной системы нужно создать новую группу разработчиков. Компания имеет возможности, соответствующие 2-му уровню зрелости согласно модели CMM. Возможны следующие значения масштабных факторов:

- предсказуемость. Это новый проект для компании — значение Низкий (4);
- гибкость разработки. Заказчик требует некоторого согласования — значение Очень высокий (1);
- разрешение архитектуры/риска. Не выполняется анализ риска, как следствие, малое разрешение риска — значение Очень низкий (5);
- связность группы. Новая группа, нет информации — значение Номинальный (3);
- зрелость процесса. Имеет место некоторое управление процессом — значение Номинальный (3).

Таблица 2.20. Оценка масштабных факторов

Масштабный фактор (W_i)	**Очень низкий 5**	**Низкий 4**
PREC	Полностью непредсказуемый проект	Главным образом, в значительной степени непредсказуемый
FLEX	Точный, строгий процесс разработки	Редкое расслабление в работе
RESL	Малое разрешение риска (20%)	Некоторое (40%)
TEAM	Очень трудное взаимодействие	Достаточно трудное взаимодействие
PREC	Полностью непредсказуемый проект	В значительной степени непредсказуемый
PMAT	Взвешенное среднее значение от количества ответов «Yes» на вопросник CMM Maturity	

Сумма этих значений равна 16, поэтому конечное значение степени B= 1,17. Вернемся к обсуждению основного уравнения модели раннего этапа проектирования. Множитель поправки M_e зависит от набора формирователей затрат, перечисленных в табл. 2.21.

Для каждого формирователя затрат определяется оценка (по 6-балльной шкале), где 1 соответствует очень низкому значению, а 6 — сверхвысокому значению. На основе оценки для каждого формирователя по таблице Боэма определяется множитель затрат EM_i Перемножение всех множителей затрат формирует множитель поправки:

$$M_e = \prod_{i=1}^{7} EM_i .$$

Слагаемое ЗАТРАТЫ$_{auto}$ используется, если некоторый процент программного кода генерируется автоматически. Поскольку производительность такой работы значительно выше, чем при ручной разработке кода, требуемые затраты вычисляются отдельно, по следующей формуле:

$$ЗАТРАТЫ_{auto} = (KALOC \times (AT /100)) / ATPROD,$$

где:

- KALOC — количество строк автоматически генерируемого кода (в тысячах строк);
- AT — процент автоматически генерируемого кода (от всего кода системы);
- ATPROD — производительность автоматической генерации кода.

Сомножитель AT в этой формуле позволяет учесть затраты на организацию взаимодействия автоматически генерируемого кода с оставшейся частью системы.

Далее затраты на автоматическую генерацию добавляются к затратам, вычисленным для кода, разработанного вручную.

Номинальный 3	**Высокий 2**	**Очень высокий 1**	**Сверхвысокий 0**

Отчасти непредсказуемый	Большей частью знакомый	В значительной степени знакомый	Полностью знакомый
Некоторое расслабление в работе	Большей частью согласованный процесс	Некоторое согласование процесса	Заказчик определил только общие цели
Частое (60%)	Большей частью (75%)	Почти всегда (90%)	Полное (100%)
Среднее взаимодействие	Главным образом кооперативность	Высокая кооперативность	Безукоризненное взаимодействие
Отчасти непредсказуемый	Большей частью знакомый	В значительной степени знакомый	Полностью знакомый

Взвешенное среднее значение от количества ответов «Yes» на вопросник CMM Maturity

Таблица 2.21. Формирователи затрат для раннего этапа проектирования

Обозначение	Название
PERS	Возможности персонала (Personnel Capability)
RCPX	Надежность и сложность продукта (Product Reliability and Complexity)
RUSE	Требуемое повторное использование (Required Reuse)
PDIF	Трудность платформы (Platform Difficulty)
PREX	Опытность персонала (Personnel Experience)
FCIL	Средства поддержки (Facilities)
SCED	График (Schedule)

Модель этапа постархитектуры

Модель этапа постархитектуры используется в период, когда уже сформирована архитектура и выполняется дальнейшая разработка программного продукта.

Основное уравнение постархитектурной модели является развитием уравнения предыдущей модели и имеет следующий вид:

$$\text{ЗАТРАТЫ} = A \times K_{\sim req} \times \text{РАЗМЕР}^{B} \times M_p + \text{ЗАТРАТЫ}_{\text{auto}} \text{ [чел.-мес]},$$

где

- коэффициент $K_{\sim req}$ учитывает возможные изменения в требованиях;
- показатель B отражает нелинейную зависимость затрат от размера проекта (размер выражается в KLOC), вычисляется так же, как и в предыдущей модели;
- в размере проекта различают две составляющие — новый код и повторно используемый код;
- множитель поправки M_p зависит от 17 факторов затрат, характеризующих продукт, аппаратуру, персонал и проект.

Изменчивость требований приводит к повторной работе, требуемой для учета предлагаемых изменений, оценка их влияния выполняется по формуле

$$K_{\sim req} = 1 + (\text{BRAK}/100),$$

где BRAK — процент кода, отброшенного (модифицированного) из-за изменения требований.

Размер проекта и продукта определяют по выражению

$$\text{РАЗМЕР} = \text{РАЗМЕР}_{\text{new}} + \text{РАЗМЕР}_{\text{reuse}} \text{ [KLOC]},$$

где

- $\text{РАЗМЕР}_{\text{new}}$ — размер нового (создаваемого) программного кода;
- $\text{РАЗМЕР}_{\text{reuse}}$ — размер повторно используемого программного кода.

Формула для расчета размера повторно используемого кода записывается следующим образом:

$$\text{РАЗМЕР}_{\text{reuse}} = \text{KASLOC} \times ((100 - AT)/ 100) \times (AA + SU + 0{,}4\, DM + 0{,}3\, CM + 0{,}3\, IM) /100,$$

где

- KASLOC — количество строк повторно используемого кода, который должен быть модифицирован (в тысячах строк);
- AT — процент автоматически генерируемого кода;
- DM — процент модифицируемых проектных моделей;
- CM — процент модифицируемого программного кода;
- IM — процент затрат на интеграцию, требуемых для подключения повторно используемого ПО;
- SU — фактор, основанный на стоимости понимания добавляемого ПО; изменяется от 50 (для сложного неструктурированного кода) до 10 (для хорошо написанного объектно-ориентированного кода);
- AA — фактор, который отражает стоимость решения о том, может ли ПО быть повторно используемым; зависит от размера требуемого тестирования и оценивания (величина изменяется от 0 до 8).

Правила выбора этих параметров приведены в руководстве по COCOMO II.

Для определения множителя поправки M_p основного уравнения используют 17 факторов затрат, которые могут быть разбиты на 4 категории. Перечислим факторы затрат, сгруппировав их по категориям.

Факторы продукта:

1) требуемая надежность ПО — RELY;

2) размер базы данных — DATA;

3) сложность продукта — CPLX;
4) требуемая повторная используемость — RUSE;
5) документирование требований жизненного цикла — DOCU.

Факторы платформы (виртуальной машины):

6) ограничения времени выполнения — TIME;
7) ограничения оперативной памяти — STOR;
8) изменчивость платформы — PVOL.

Факторы персонала:

9) возможности аналитика — ACAP;
10) возможности программиста — PCAP;
11) опыт работы с приложением — AEXP;
12) опыт работы с платформой — PEXP;
13) опыт работы с языком и утилитами — LTEX;
14) непрерывность персонала — PCON.

Факторы проекта:

15) использование программных утилит — TOOL;
16) мультисетевая разработка — SITE;
17) требуемый график разработки — SCED.

Для каждого фактора определяется оценка (по 6-балльной шкале). На основе оценки для каждого фактора по таблице Боэма определяется множитель затрат EM_i. Перемножение всех множителей затрат дает множитель поправки пост-архитектурной модели:

$$M_p = \prod_{i=1}^{17} EM_i \ .$$

Значение M_p отражает реальные условия выполнения программного проекта и позволяет троекратно увеличить (уменьшить) начальную оценку затрат.

ПРИМЕЧАНИЕ

Трудоемкость работы с факторами затрат минимизируется за счет использования специальных таблиц. Справочный материал для оценки факторов затрат приведен в приложении А.

От оценки затрат легко перейти к стоимости проекта. Переход выполняют по формуле:

СТОИМОСТЬ = ЗАТРАТЫ х РАБ_ КОЭФ,

где среднее значение рабочего коэффициента составляет $15 000 за человеко-месяц.

После определения затрат и стоимости можно оценить длительность разработки. Модель COCOMO II содержит уравнение для оценки календарного времени TDEV, требуемого для выполнения проекта. Для моделей всех уровней справедливо:

$$\text{Длительность (TDEV)} = [3{,}0 \times (\text{ЗАТРАТЫ})^{(0{,}33+0{,}2(B-1{,}01))}] \times \text{SCEDPercentage}/100 \text{ [мес]},$$

где

- B — ранее рассчитанный показатель степени;
- SCEDPercentage — процент увеличения (уменьшения) номинального графика.

Если нужно определить номинальный график, то принимается SCEDPercentage =100 и правый сомножитель в уравнении обращается в единицу. Следует отметить, что COCOMO II ограничивает диапазон уплотнения/растягивания графика (от 75 до 160%). Причина проста — если планируемый график существенно отличается от номинального, это означает внесение в проект высокого риска.

Рассмотрим пример. Положим, что затраты на проект равны 20 человеко-месяцев. Примем, что все масштабные факторы номинальны (имеют значения 3), поэтому, в соответствии с табл. 2.20, показатель степени 5=1,16. Отсюда следует, что номинальная длительность проекта равна

$$\text{TDEV} = 3, \text{Ox } (20)^{0{,}36} = 8{,}8 \text{ мес.}$$

Отметим, что зависимость между затратами и количеством разработчиков носит характер, существенно отличающийся от линейного. Очень часто увеличение количества разработчиков приводит к возрастанию затрат. В чем причина? Ответ прост:

- увеличивается время на взаимодействие и обучение сотрудников, согласование совместных решений;
- возрастает время на определение интерфейсов между частями программной системы.

Удвоение разработчиков не приводит к двукратному сокращению длительности проекта. Модель COCOMO II явно утверждает, что длительность проекта является функцией требуемых затрат, прямой зависимости от количества сотрудников нет. Другими словами, она устраняет миф нерадивых менеджеров в том, что добавление людей поможет ликвидировать отставание в проекте.

COCOMO II предостерегает от определения потребного количества сотрудников путем деления затрат на длительность проекта. Такой упрощенный подход часто приводит к срыву работ. Реальная картина имеет другой характер. Количество людей, требуемых на этапе планирования и формирования требований, достаточно мало. На этапах проектирования и кодирования потребность в увеличении команды возрастает, после окончания кодирования и тестирования численность необходимых сотрудников достигает минимума.

Предварительная оценка программного проекта

В качестве иллюстрации применения методики оценки, изложенной в разделе «Выполнение оценки проекта на основе LOC- и FP-метрик», рассмотрим конкретный пример. Предположим, что поступил заказ от концерна «СУПЕРАВТО». Необходимо создать ПО для рабочей станции дизайнера автомобиля (РДА). Заказчик определил проблемную область проекта в своей спецификации:

- ПО РДА должно формировать 2- и 3-мерные изображения для дизайнера;
- дизайнер должен вести диалог с РДА и управлять им с помощью стандартизованного графического пользовательского интерфейса;
- геометрические данные и прикладные данные должны содержаться в базе данных РДА;
- модули проектного анализа рабочей станции должны формировать данные для широкого класса дисплеев SVGA;
- ПО РДА должно управлять и вести диалог со следующими периферийными устройствами: мышь, дигитайзер (графический планшет для ручного ввода), плоттер (графопостроитель), сканер, струйный и лазерный принтеры.

Прежде всего надо детализировать проблемную область. Следует выделить базовые функции ПО и очертить количественные границы. Очевидно, нужно определить, что такое «стандартизованный графический пользовательский интерфейс», какими должны быть размер и другие характеристики базы данных РДА и т. д.

Будем считать, что эта работа проделана и что идентифицированы следующие основные функции ПО:

1. Средства управления пользовательским интерфейсом СУПИ.
2. Анализ двухмерной графики А2Г.
3. Анализ трехмерной графики А3Г.
4. Управление базой данных УБД.
5. Средства компьютерной дисплейной графики КДГ.
6. Управление периферией УП.
7. Модули проектного анализа МПА.

Теперь нужно оценить каждую из функций количественно, с помощью LOC-оценки. По каждой функции эксперты предоставляют лучшее, худшее и вероятное значения. Ожидаемую LOC-оценку реализации функции определяем по формуле

$$LOC_{ожi} = (LOC_{лучшi} + LOC_{худшi} + 4 \times LOC_{вероятнi})/6,$$

результаты расчетов заносим в табл. 2.22.

Таблица 2.22. Начальная таблица оценки проекта

Функция	Лучш. [LOC]	Вероят. [LOC]	Худш. [LOC]	Ожид. [LOC]	Уд. стоимость [$/LOC]	Стоимость[$]	Произв. [LOC/ чел-мес]	Затраты [чел-мес]
СУПИ	1800	2400	2650	2340				
А2Г	4100	5200	7400	5380				
А3Г	4600	6900	8600	6800				
УБД	2950	3400	3600	3350				
КДГ	4050	4900	6200	4950				
УП	2000	2100	2450	2140				
МПА	6600	8500	9800	8400				
Итого				33360				

Для определения удельной стоимости и производительности обратимся в архив фирмы, где хранятся данные метрического базиса, собранные по уже выполненным проектам. Предположим, что из метрического базиса извлечены данные по функциям-аналогам, представленные в табл. 2.23.

Видно, что наибольшую удельную стоимость имеет строка функции управления периферией (требуются специфические и конкретные знания по разнообразным периферийным устройствам), наименьшую удельную стоимость — строка функции управления пользовательским интерфейсом (применяются широко известные решения).

Таблица 2.23. Данные из метрического базиса фирмы

Функция	$LOC_{ані}$	$УД\_СТОИМОСТЬ_{ані}$[$ / LOC]	$ПРОИЗВ_{ані}$[LOC/чел-мес]
СУПИ	585	14	1260
А_Г	3000	20	440
УБД	1117	18	720
КДГ	2475	22	400
УП	214	28	1400
МПА	1400	18	1800

Считается, что удельная стоимость строки является константой и не изменяется от реализации к реализации. Следовательно, стоимость разработки каждой функции рассчитываем по формуле

$$СТОИМОСТЬ_i = LOC_{ожi} \times УД\_СТОИМОСТЬ_{ані}.$$

Для вычисления производительности разработки каждой функции выберем самый точный подход — подход настраиваемой производительности:

$$ПРОИЗВ_i = ПРОИЗВ_{ані} \times (LOC_{ані} / LOC_{ожi}).$$

Соответственно, затраты на разработку каждой функции будем определять по выражению

$$ЗАТРАТЫ_i = (LOC_{ожi} / ПРОИЗВ_i)[чел.-мес].$$

Теперь мы имеем все необходимые данные для завершения расчетов. Заполним до конца таблицу оценки нашего проекта (табл. 2.24).

Таблица 2.24. Конечная таблица оценки проекта

Функция	Лучш.	Вероят.	Худш.	Ожид. [LOC]	Уд. стоимость [S/LOC]	Стоимость [$]	Произв. [LOC/ чел.-мес]	Затраты [чел-мес]
СУПИ	1800	2400	2650	2340	14	32760	315	7,4
А2Г	4100	5200	7400	5380	20	107600	245	21,9
АЗГ	4600	6900	8600	6800	20	136000	194	35,0
УБД	2950	3400	3600	3350	18	60300	240	13,9
КДГ	4050	4900	6200	4950	22	108900	200	24,7
УП	2000	2100	2450	2140	28	59920	140	15,2
МПА	6600	8500	9800	8400	18	151200	300	28,0
Итого				33360		656680		146

Учитывая важность полученных результатов, проверим расчеты с помощью FP-указателей. На данном этапе оценивания разумно допустить, что все информационные характеристики имеют средний уровень сложности. В этом случае результаты экспертной оценки принимают вид, представленный в табл. 2.25, 2.26.

Таблица 2.25. Оценка информационных характеристик проекта

Характеристика	Лучш.	Вероят.	Худш.	Ожид.	Сложность	Количество
Вводы	20	24	30	24	x 4	96
Выводы	12	15	22	16	x 5	80
Запросы	16	22	28	22	x 4	88
Логические файлы	4	4	5	4	x 10	40
Интерфейсные файлы	2	2	3	2	x 7	14
Общее количество						318

Таблица *2.26.* Оценка системных параметров проекта

Коэффициент регулировки сложности		Оценка
F_1	Передачи данных	2
F_2	Распределенная обработка данных	0
F_3	Производительность	4
F_4	Распространенность используемой конфигурации	3
F_5	Скорость транзакций	4
F_6	Оперативный ввод данных	5
F_7	Эффективность работы конечного пользователя	5
F_8	Оперативное обновление	3
F_9	Сложность обработки	5
F_{10}	Повторная используемость	4
F_{11}	Легкость инсталляции	3
F_{12}	Легкость эксплуатации	4
F_{13}	Разнообразные условия размещения	5
F_{14}	Простота изменений	5

Таким образом, получаем:

$$\text{FP} = \text{Общее количество x } (0{,}65 + 0{,}01 \text{ x } \sum_{i=1}^{14} F_i) = 318 \text{ x } 1{,}17 = 372.$$

Используя значение производительности, взятое в метрическом базисе фирмы,

Производительность = 2,55 [FP / чел.-мес],

вычисляем значения затрат и стоимости:

Затраты = FP / Производительность = 145,9 [чел.-мес],

Стоимость = Затраты x $4500 = $656500.

Итак, результаты проверки показали хорошую достоверность результатов. Но мы не будем останавливаться на достигнутом и организуем еще одну проверку, с помощью модели COCOMO II.

Примем, что все масштабные факторы и факторы затрат имеют номинальные значения. В силу этого показатель степени B = 1,16, а множитель поправки M_p= 1. Кроме того, будем считать, что автоматическая генерация кода и повторное использование компонентов не предусматриваются. Следовательно, мы вправе применить формулу

ЗАТРАТЫ = A x РАЗМЕР^B[чел.-мес]

и получаем:

ЗАТРАТЫ = $2{,}5(33{,}3)^{1{,}16}$ =145,87 [чел.-мес].

Соответственно, номинальная длительность проекта равна

$$\text{Длительность} = [3{,}0 \times (\text{ЗАТРАТЫ})^{(0{,}33+0{,}2(B-1{,}01))}] = 3(145{,}87)^{0{,}36} = 18[\text{мес}].$$

Подведем итоги. Выполнена предварительная оценка программного проекта. Для минимизации риска оценивания использованы три методики, доказавшие корректность полученных результатов.

Анализ чувствительности программного проекта

COCOMO II — авторитетная и многоплановая модель, позволяющая решать самые разнообразные задачи управления программным проектом.

Рассмотрим возможности этой модели в задачах анализа чувствительности — чувствительности программного проекта к изменению условий разработки.

Будем считать, что корпорация «СверхМобильныеСвязи» заказала разработку ПО для встроенной космической системы обработки сообщений. Ожидаемый размер ПО — 10 KLOC, используется серийный микропроцессор. Примем, что масштабные факторы имеют номинальные значения (показатель степени B = 1,16) и что автоматическая генерация кода не предусматривается. К проведению разработки привлекаются главный аналитик и главный программист высокой квалификации, поэтому средняя зарплата в команде составит $ 6000 в месяц. Команда имеет годовой опыт работы с этой проблемной областью и полгода работает с нужной аппаратной платформой.

В терминах COCOMO II проблемную область (область применения продукта) классифицируют как «операции с приборами» со следующим описанием: встроенная система для высокоскоростного мультиприоритетного обслуживания удаленных линий связи, обеспечивающая возможности диагностики.

Оценку пост-архитектурных факторов затрат для проекта сведем в табл. 2.27.

Из таблицы следует, что увеличение затрат в 1,3 раза из-за очень высокой сложности продукта уравновешивается их уменьшением вследствие высокой квалификации аналитика и программиста, а также активного использования программных утилит.

Таблица 2.27. Оценка пост-архитектурных факторов затрат

Фактор	Описание	Оценка	Множитель
RELY	Требуемая надежность ПО	Номинал.	1
DATA	Размер базы данных — 20 Кбайт	Низкая	0,93
CPLX	Сложность продукта	Очень высок.	1,3
RUSE	Требуемая повторная используемость	Номинал.	1
DOCU	Документирование жизненного цикла	Номинал.	1
TIME	Ограничения времени выполнения (70%)	Высокая	1,11
STOR	Ограничения оперативной памяти (45 из 64 Кбайт, 70%)	Высокая	1,06
PVOL	Изменчивость платформы (каждые 6 месяцев)	Номинал.	1
ACAP	Возможности аналитика (75%)	Высокая	0,83
PCAP	Возможности программиста (75%)	Высокая	0,87
AEXP	Опыт работы с приложением (1 год)	Номинал.	1
PEXP	Опыт работы с платформой (6 месяцев)	Низкая	1,12
LTEX	Опыт работы с языком и утилитами (1 год)	Номинал.	1
PCON	Непрерывность персонала (1 2% в год)	Номинал.	1
TOOL	Активное использование программных утилит	Высокая	0,86
SITE	Мультисетевая разработка (телефоны)	Низкая	1,1
SCED	Требуемый график разработки	Номинал.	1
Множитель поправки M_p			1,088

Рассчитаем затраты и стоимость проекта:

$$\text{ЗАТРАТЫ} = A \times \text{РАЗМЕР}^B \times M_p = 2{,}5(10)^{1{,}16} \times 1{,}088 = 36 \times 1{,}088 = 39[\text{чел.-мес}],$$

$$\text{СТОИМОСТЬ} = \text{ЗАТРАТЫ} \times \$6000 = \$234\,000.$$

Таковы стартовые условия программного проекта. А теперь обсудим несколько сценариев возможного развития событий.

Сценарий понижения зарплаты

Положим, что заказчик решил сэкономить на зарплате разработчиков. Рычаг — понижение квалификации аналитика и программиста. Соответственно, зарплата сотрудников снижается до $5000. Оценки их возможностей становятся номинальными, а соответствующие множители затрат принимают единичные значения:

$$EM_{ACAP} = EM_{PCAP} = 1.$$

Следствием такого решения является возрастание множителя поправки M_p= 1,507, а также затрат и стоимости:

$$\text{ЗАТРАТЫ} = 36 \times 1{,}507 = 54\ [\text{чел.-мес}],$$

$$\text{СТОИМОСТЬ} = \text{ЗАТРАТЫ} \times \$5000 = \$270\,000,$$

Проигрыш_ в_стоимости = $36 000.

Сценарий наращивания памяти

Положим, что разработчик предложил нарастить память — купить за $1000 чип ОЗУ емкостью 96 Кбайт (вместо 64 Кбайт). Это меняет ограничение памяти (используется не 70%, а 47%), после чего фактор STOR снижается до номинального:

EM_{STOR}=1-> M_p =1,026,

ЗАТРАТЫ = 36x1,026 = 37 [чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ x $6000 = $222000,

Выигрыш_ в_стоимости = $ 12 000.

Сценарий использования нового микропроцессора

Положим, что заказчик предложил использовать новый, более дешевый МП (дешевле на $1000). К чему это приведет? Опыт работы с его языком и утилитами понижается от номинального до очень низкого и EM_{LTEX} = 1,22, а разработанные для него утилиты (компиляторы, ассемблеры и отладчики) примитивны и ненадежны (в результате фактор TOOL понижается от высокого до очень низкого и EM_{TOOL}= 1,24):

M_p = (1,088 / 0,86) x 1,22 x 1,24 = 1,914,

ЗАТРАТЫ = 36x1,914 = 69[чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ x $6000 = $414000,

Проигрыш_в_стоимости = $180000.

Сценарий уменьшения средств на завершение проекта

Положим, что к разработке принят сценарий с наращиванием памяти:

ЗАТРАТЫ = 36 x 1,026 = 37 [чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ x $6000 = $222000.

Кроме того, предположим, что завершился этап анализа требований, на который было израсходовано $22 000 (10% от бюджета). После этого на завершение проекта осталось $200 000.

Допустим, что в этот момент «коварный» заказчик сообщает об отсутствии у него достаточных денежных средств и о предоставлении на завершение разработки только $170 000 (15%-ное уменьшение оплаты).

Для решения этой проблемы надо установить возможные изменения факторов затрат, позволяющие уменьшить оценку затрат на 15%.

Первое решение: уменьшение размера продукта (за счет исключения некоторых функций). Нам надо определить размер минимизированного продукта. Будем исходить из того, что затраты должны уменьшиться с 37 до 31,45 чел.-мес. Решим уравнение:

$$2{,}5\ (\text{НовыйРазмер})^{1{,}16} = 31{,}45\ \text{[чел.-мес]}.$$

Очевидно, что

$$(\text{НовыйРазмер})^{1{,}16} = 12{,}58,$$

$$(\text{НовыйРазмер})^{1{,}16} = 12{,}58^{1/1{,}16} = 8{,}872\ \text{[KLOC]}.$$

Другие решения:

- уменьшить требуемую надежность с номинальной до низкой. Это сокращает стоимость проекта на 12% (EM_{RELY} изменяется с 1 до 0,88). Такое решение приведет к увеличению затрат и трудностей при применении и сопровождении;
- повысить требования к квалификации аналитиков и программистов (с высоких до очень высоких). При этом стоимость проекта уменьшается на 15-19%. Благодаря программисту стоимость может уменьшиться на (1 - 0,74/0,87) x 100% = 15%. Благодаря аналитику стоимость может понизиться на (1 - 0,67/0,83) x 100% = 19%. Основная трудность — поиск специалистов такого класса (готовых работать за те же деньги);
- повысить требования к опыту работы с приложением (с номинальных до очень высоких) или требования к опыту работы с платформой (с низких до высоких). Повышение опыта работы с приложением сокращает стоимость проекта на (1- 0,81) x 100% = 19%; повышение опыта работы с платформой сокращает стоимость проекта на (1 - 0,88/1,12) x 100% = 21,4%. Основная трудность — поиск экспертов (специалистов такого класса);
- повысить уровень мультисетевой разработки с низкого до высокого. При этом стоимость проекта уменьшается на (1 - 0,92/1,1) x 100% = 16,4%;
- ослабить требования к режиму работы в реальном времени. Предположим, что 70%-ное ограничение по времени выполнения связано с желанием заказчика обеспечить обработку одного сообщения за 2 мс. Если же заказчик согласится на увеличение среднего времени обработки с 2 до 3 мс, то ограничение по времени станет равно (2 мс/3 мс) x 70% = 47%, в результате чего фактор TIME уменьшится с высокого до номинального, что приведет к экономии затрат на (1- 1/1,11) x 100%= 10%;
- учет других факторов затрат не имеет смысла. Некоторые факторы (размер базы данных, ограничения оперативной памяти, требуемый график разработки) уже имеют минимальные значения, для других трудно ожидать быстрого улучшения (использование программных утилит, опыт работы с языком и утилитами), третьи имеют оптимальные значения (требуемая повторная используемость, документирование требований жизненного цикла). На некоторые разработчик почти не может повлиять (сложность продукта, изменчивость платформы). Наконец, житейские неожиданности едва ли позволят улучшить принятое значение фактора «непрерывность персонала».

Какое же решение следует выбрать? Наиболее целесообразное решение — исключение отдельных функций продукта. Вторым (по предпочтительности) решением является повышение уровня мультисетевой разработки (все равно это придется

сделать в ближайшее время). В качестве третьего решения можно рассматривать ослабление требований к режиму работы в реальном времени. Принятие же других решений зависит от наличия необходимых специалистов или средств разработки. Впрочем, окончательное решение должно выбираться в процессе переговоров с заказчиком, когда учитываются все соображения.

Выводы.

1. Факторы затрат оказывают существенное влияние на выходные параметры программного проекта.
2. Модель COCOMO II предлагает широкий спектр факторов затрат, учитывающих большинство реальных ситуаций в «жизни» программного проекта.
3. Модель COCOMO II обеспечивает перевод качественного обоснования решения менеджера на количественные рельсы, тем самым повышая объективность принимаемого решения.

Метод анализа Джексона

Как и метод Варнье-Орра, метод Джексона появился в период революции структурного программирования. Фактически оба метода решали одинаковую задачу: распространить базовые структуры программирования (последовательность, выбор, повторение) на всю область конструирования сложных программных систем. Именно поэтому основные выразительные средства этих методов оказались так похожи друг на друга.

Методика Джексона

Метод Джексона (1975) включает 6 шагов [39]. Три шага выполняются на этапе анализа, а остальные — на этапе проектирования.

1. *Объект-действие.* Определяются объекты — источники или приемники информации и действия — события реального мира, воздействующие на объекты.
2. *Объект-структура.* Действия над объектами представляются диаграммами Джексона.
3. *Начальное моделирование.* Объекты и действия представляются как обрабатывающая модель. Определяются связи между моделью и реальным миром.
4. *Доопределение функций.* Выделяются и описываются сервисные функции.
5. *Учет системного времени.* Определяются и оцениваются характеристики планирования будущих процессов.
6. *Реализация.* Согласование с системной средой, разработка аппаратной платформы.

Шаг объект-действие

Начинается с определения проблемы на естественном языке.

Пример:

Разработать компьютерную систему для обслуживания университетских перевозок. Университет размещается на двух территориях. Для перемещения студентов используется один транспорт. Он перемещается между двумя фиксированными остановками. На каждой остановке имеется кнопка вызова.

При нажатии кнопки:

- если транспорт на остановке, то студенты заходят в него и перемещаются на другую остановку;
- если транспорт в пути, то студенты ждут прибытия на другую остановку, приема студентов и возврата на текущую остановку;
- если транспорт на другой остановке, то он ее покидает, прибывает на текущую остановку и принимает студентов, нажавших кнопку.

Транспорт должен стоять на остановке до появления запроса на обслуживание.

Описание исследуется для выделения объектов. Производится грамматический разбор. Возможны следующие кандидаты в объекты: территория, студенты, транспорт, остановка, кнопка. У нас нет нужды прямо использовать территорию, студентов, остановку — все они лежат вне области модели и отвергаются как возможные объекты. Таким образом, мы выбираем объекты транспорт и кнопка.

Для выделения действий исследуются все глаголы описания.

Кандидатами действий являются: перемещаться, прибывает, нажимать, принимать, покидать. Мы отвергаем перемещаться, принимать потому, что они относятся к студентам, а студенты не выделены как объект. Мы выбираем действия: прибывает, нажимать, покидать.

Заметим, что при выделении объектов и действий возможны ошибки. Например, отвергнув студентов, мы лишились возможности исследовать загрузку транспорта. Впрочем, список объектов и действий может модифицироваться в ходе дальнейшего анализа.

Шаг объект-структура

Структура объектов описывает последовательность действий над объектами (в условном времени).

Для представления структуры объектов Джексон предложил 3 типа структурных диаграмм. Они показаны на рис. 3.13. В

первой диаграмме к объектам применяется такое действие, как последовательность, во второй — выбор, в третьей — повторение.

Рассмотрим объектную структуру для транспорта (см. рис. 3.14). Условимся, что начало и конец истории транспорта — у первой остановки. Действиями, влияющими на объект, являются Покинуть и Прибыть.

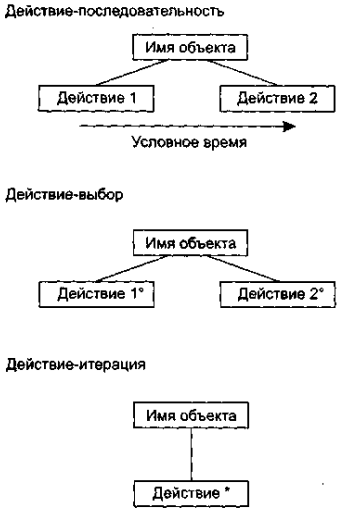


Рис. 3.13. Три типа структурных диаграмм Джексона

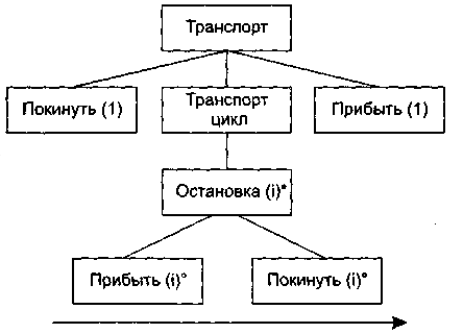


Рис. 3.14. Объектная структура для транспорта

Диаграмма показывает, что транспорт начинает работу у остановки 1, тратит основное время на перемещение между остановками 1 и 2 и окончательно возвращается на остановку 1. Прибытие на остановку, следующее за отъездом с другой остановки, представляется как пара действий Прибыть(*i*) и Покинуть(*i*). Заметим, что диаграмму можно сопровождать комментариями, которые не могут прямо представляться средствами метода. Например, «значение г в двух последовательных остановках должно быть разным».

Структурная диаграмма для объекта Кнопка показывает (рис. 3.15), что к нему многократно применяется действие Нажать.

Рис. 3.15. Структурная диаграмма для объекта Кнопка

В заключение заметим, что структурная диаграмма — время-ориентированное описание действий, выполняемых над объектом. Она создается для каждого объекта модели.

Шаг начального моделирования

Начальное моделирование — это шаг к созданию описания системы как модели реального мира. Описание создается с помощью диаграммы системной спецификации.

Элементами диаграммы системной спецификации являются физические процессы (имеют суффикс 0) и их модели (имеют суффикс 1). Как показано на рис. 3.16, предусматриваются 2 вида соединений между физическими процессами и моделями.

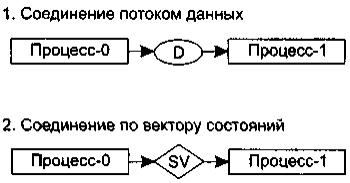


Рис. 3.16. Соединения между физическими процессами и их моделями

Соединение потоком данных производится, когда физический процесс передает, а модель принимает информационный поток. Полагают, что поток передается через буфер неограниченной емкости типа FIFO (обозначается овалом).

Соединение по вектору состояний происходит, когда модель наблюдает вектор состояния физического процесса. Вектор состояния обозначается ромбиком.

Диаграмма системной спецификации для системы обслуживания перевозок приведена на рис. 3.17.

ПРИМЕЧАНИЕ

При нажатии кнопки формируется импульс, который может быть передан в модель как элемент данных, поэтому для кнопки выбрано соединение потоком данных.

Датчики, регистрирующие прибытие и убытие транспорта, не формируют импульса, они воздействуют на электронный переключатель. Состояние переключателя может быть оценено. Поэтому для транспорта выбрано соединение по вектору состояний.

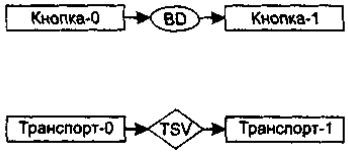


Рис. 3.17. Диаграмма системной спецификации для системы обслуживания перевозок

Для фиксации особенностей процессов-моделей Джексон предлагает специальное описание — структурный текст. Например, структурный текст для модели КНОПКА-1 имеет вид

```
КНОПКА-1
        читать BD;
        НАЖАТЬ цикл ПОКА BD
                нажать;
                читать BD;
        конец НАЖАТЬ;
конец КНОПКА-1;
```

Структура модели КНОПКА-1 отличается от структуры физического процесса КНОПКА-0 добавлением оператора для чтения буфера BD, который соединяет физический мир с моделью.

Прежде чем написать структурный текст для модели ТРАНСПОРТ-1, мы должны сделать ряд замечаний.

Во-первых, состояние транспорта будем отслеживать по переменным ПРИБЫЛ, УБЫЛ. Они отражают состояние электронного переключателя физического транспорта.

Во-вторых, для учета инерционности процессов в физическом транспорте в модель придется ввести дополнительные операции:

- ЖДАТЬ (ожидание в изменении состояния физического транспорта);
- ТРАНЗИТ (операция задержки в модели на перемещение транспорта между остановками).

С учетом замечаний структурная диаграмма модели примет вид, изображенный на рис. 3.18.

Соответственно, структурный текст модели записывается в форме

```
ТРАНСПОРТ-1
        опрос TSV;
        ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
                опрос TSV;
        конец ЖДАТЬ;
        покинуть(1);
        ТРАНЗИТ цикл ПОКА УБЫЛ(1)
                опрос TSV;
        конец ТРАНЗИТ;
        ТРАНСПОРТ цикл
                ОСТАНОВКА
                        прибыть(i);
                        ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)
                                опрос TSV;
                        конец ЖДАТЬ;
                        покинуть(i);
                        ТРАНЗИТ цикл ПОКА УБЫЛ(i)
                                опрос TSV;
                        конец ТРАНЗИТ;
                конец ОСТАНОВКА;
        конец ТРАНСПОРТ;
```

прибыть(1);
конец ТРАНСПОРТ-1;

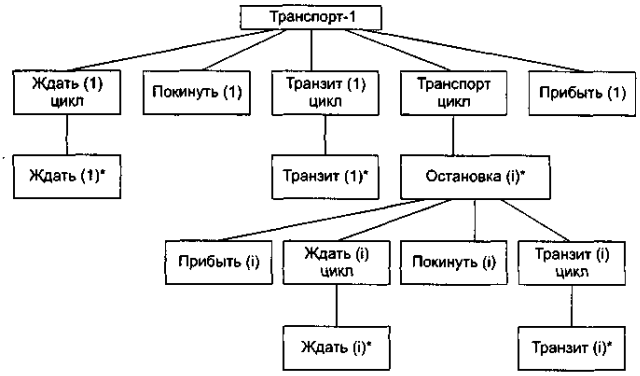


Рис. 3.18. Структурная диаграмма модели транспорта

Модульность

Модуль — фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность — свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса, модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы [52]. Проиллюстрируем эту точку зрения.

Пусть $C(x)$ — функция сложности решения проблемы x, $T(x)$ — функция затрат времени на решение проблемы x. Для двух проблем p_1 и p_2 из соотношения $C(p_1) > C(p_2)$ следует, что

$$T(p_1)>T(p_2). \quad (4.1)$$

Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Далее. Из практики решения проблем человеком следует:

$$C(p_1+p_2)>C(p_1)+C(p_2).$$

Отсюда с учетом соотношения (4.1) запишем:

$$T(p_1+p_2)>T(p_1)+T(p_2). \quad (4.2)$$

Соотношение (4.2) — это обоснование модульности. Оно приводит к заключению «разделяй и властвуй» — сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством (4.2), имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь отражена лишь часть реальности, ведь здесь не учитываются затраты на межмодульный интерфейс. Как показано на рис. 4.11, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.

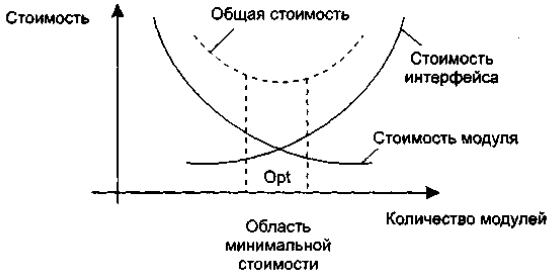


Рис. 4.11. Затраты на модульность

Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Увы, у нас нет необходимого опыта для гарантированного предсказания Opt. Впрочем, разработчики знают, что оптимальный модуль должен удовлетворять двум критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

Информационная закрытость

Принцип информационной закрытости (автор — Д. Парнас, 1972) утверждает: содержание модулей должно быть скрыто друг от друга [60]. Как показано на рис. 4.12, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

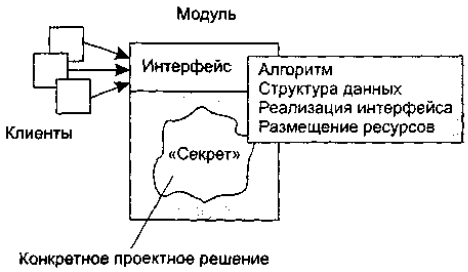


Рис. 4.12. Информационная закрытость модуля

Информационная закрытость означает следующее:

1) все модули независимы, обмениваются только информацией, необходимой для работы;
2) доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

- обеспечивается возможность разработки модулей различными, независимыми коллективами;
- обеспечивается легкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

Идеальный модуль играет роль «черного ящика», содержимое которого невидимо клиентам. Он прост в использовании — количество «ручек и органов управления» им невелико (аналогия с эксплуатацией телевизора). Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям. Обсудим характеристики внутренних и внешних связей модуля.

Связность модуля

Связность модуля (Cohesion) — это мера зависимости его частей [58], [70], [77]. Связность — внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, то есть тем «черней» его ящик (капсула, защитная оболочка модуля), тем меньше «ручек управления» на нем находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существует 7 типов связности:

1. **Связность по совпадению** (СС=0). В модуле отсутствуют явно выраженные внутренние связи.
2. **Логическая связность** (СС=1). Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок. При использовании такого модуля клиент выбирает только одну из подпрограмм.

Недостатки:

- сложное сопряжение;
- большая вероятность внесения ошибок при изменении сопряжения ради одной из функций.

3. **Временная связность** (СС=3). Части модуля не связаны, но необходимы в один и тот же период работы системы.

Недостаток: сильная взаимная связь с другими модулями, отсюда — сильная чувствительность внесению изменений.

4. **Процедурная связность** (СС=5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
5. **Коммуникативная связность** (СС=7). Части модуля связаны по данным (работают с одной и той же структурой данных).
6. **Информационная (последовательная) связность** (СС=9). Выходные данные одной части используются как входные данные в другой части модуля.
7. **Функциональная связность** (СС=10). Части модуля вместе реализуют одну функцию.

Отметим, что типы связности 1,2,3 — результат неправильного планирования архитектуры, а тип связности 4 — результат небрежного планирования архитектуры приложения.

Общая характеристика типов связности представлена в табл. 4.1.

Таблица 4.1. Характеристика связности модуля

Тип связности	Сопровождаемость	Роль модуля
Функциональная	Лучшая сопровождаемость	«Черный ящик»
Информационная (последовательная)		Не совсем «черный ящик»
Кэммуникативная		«Серый ящик»
Процедурная	Худшая сопровождаемость	«Белый» или «просвечивающий ящик»
Временная		«Белый ящик»
Логическая		
По совпадению		

Функциональная связность

Функционально связный модуль содержит элементы, участвующие в выполнении одной и только одной проблемной

задачи. Примеры функционально связных модулей:

- Вычислять синус угла;
- Проверять орфографию;
- Читать запись файла;
- Вычислять координаты цели;
- Вычислять зарплату сотрудника;
- Определять место пассажира.

Каждый из этих модулей имеет единичное назначение. Когда клиент вызывает модуль, выполняется только одна работа, без привлечения внешних обработчиков. Например, модуль Определять место пассажира должен делать только это; он не должен распечатывать заголовки страницы.

Некоторые из функционально связных модулей очень просты (например, Вычислять синус угла или Читать запись файла), другие сложны (например, Вычислять координаты цели). Модуль Вычислять синус угла, очевидно, реализует единичную функцию, но как может модуль Вычислять зарплату сотрудника выполнять только одно действие? Ведь каждый знает, что приходится определять начисленную сумму, вычеты по рассрочкам, подоходный налог, социальный налог, алименты и т. д.! Дело в том, что, несмотря на сложность модуля и на то, что его обязанность исполняют несколько подфункций, если его действия можно представить как единую проблемную функцию (с точки зрения клиента), тогда считают, что модуль функционально связен.

Приложения, построенные из функционально связных модулей, легче всего сопровождать. Соблазнительно думать, что любой модуль можно рассматривать как однофункциональный, но не надо заблуждаться. Существует много разновидностей модулей, которые выполняют для клиентов перечень различных работ, и этот перечень нельзя рассматривать как единую проблемную функцию. Критерий при определении уровня связности этих нефункциональных модулей — как связаны друг с другом различные действия, которые они исполняют.

Информационная связность

При информационной (последовательной) связности элементы-обработчики модуля образуют конвейер для обработки данных — результаты одного обработчика используются как исходные данные для следующего обработчика. Приведем пример:

Модуль Прием и проверка записи
    прочитать запись из файла
    проверить контрольные данные в записи
    удалить контрольные поля в записи
    вернуть обработанную запись
Конец модуля

В этом модуле 3 элемента. Результаты первого элемента (прочитать запись из файла) используются как входные данные для второго элемента (проверить контрольные данные в записи) и т. д.

Сопровождать модули с информационной связностью почти так же легко, как и функционально связные модули. Правда, возможности повторного использования здесь ниже, чем в случае функциональной связности. Причина — совместное применение действий модуля с информационной связностью полезно далеко не всегда.

Коммуникативная связность

При коммуникативной связности элементы-обработчики модуля используют одни и те же данные, например внешние данные. Пример коммуникативно связного модуля:

Модуль Отчет и средняя зарплата
    используется Таблица зарплаты служащих
    сгенерировать Отчет по зарплате
    вычислить параметр Средняя зарплата
    вернуть Отчет по зарплате. Средняя зарплата
Конец модуля

Здесь все элементы модуля работают со структурой Таблица зарплаты служащих.

С точки зрения клиента проблема применения коммуникативно связного модуля состоит в избыточности получаемых результатов. Например, клиенту требуется только отчет по зарплате, он не нуждается в значении средней зарплаты. Такой клиент будет вынужден выполнять избыточную работу — выделение в полученных данных материала отчета. Почти всегда разбиение коммуникативно связного модуля на отдельные функционально связные модули улучшает сопровождаемость системы.

Попытаемся провести аналогию между информационной и коммуникативной связностью.

Модули с коммуникативной и информационной связностью подобны в том, что содержат элементы, связанные по данным. Их удобно использовать, потому что лишь немногие элементы в этих модулях связаны с внешней средой. Главное различие между ними — информационно связный модуль работает подобно сборочной линии; его обработчики действуют в определенном порядке; в коммуникативно связном модуле порядок выполнения действий безразличен. В нашем примере не имеет значения, когда генерируется отчет (до, после или одновременно с вычислением средней зарплаты).

Процедурная связность

При достижении процедурной связности мы попадаем в пограничную область между хорошей сопровождаемостью (для

модулей с более высокими уровнями связности) и плохой сопровождаемостью (для модулей с более низкими уровнями связности). Процедурно связный модуль состоит из элементов, реализующих независимые действия, для которых задан порядок работы, то есть порядок передачи управления. Зависимости по данным между элементами нет. Например:

```
Модуль Вычисление средних значений
    используется Таблица-А. Таблица-В
    вычислить среднее по Таблица-А
    вычислить среднее по Таблица-В
    вернуть среднееТабл-А. среднееТабл-В
Конец модуля
```

Этот модуль вычисляет средние значения для двух полностью несвязанных таблиц Таблица-А и Таблица-В, каждая из которых имеет по 300 элементов.

Теперь представим себе программиста, которому поручили реализовать данный модуль. Соблазнившись возможностью минимизации кода (использовать один цикл в интересах двух обработчиков, ведь они находятся внутри единого модуля!), программист пишет:

```
Модуль Вычисление средних значений
    используется Таблица-А. Таблица-В
    суммаТабл-А := 0
    суммаТабл-В := 0
    для i := 1 до 300
            суммаТабл-А := суммаТабл-А + Таблица-А(i)
            суммаТабл-В :- суммаТабл-В + Таблица-В(i)
    конец для
    среднееТабл-А := суммаТабл-А / 300
    среднееТабл-В := суммаТабл-В / 300
    вернуть среднееТабл-А, среднееТабл-В
Конец модуля
```

Для процедурной связности этот случай типичен — независимый (на уровне проблемы) код стал зависимым (на уровне реализации). Прошли годы, продукт сдали заказчику. И вдруг возникла задача сопровождения — модифицировать модуль под уменьшение размера таблицы В. Оцените, насколько удобно ее решать.

Временная связность

При связности по времени элементы-обработчики модуля привязаны к конкретному периоду времени (из жизни программной системы).

Классическим примером временной связности является модуль инициализации:

```
Модуль Инициализировать Систему
    перемотать магнитную ленту 1
    Счетчик магнитной ленты 1 := 0
    перемотать магнитную ленту 2
    Счетчик магнитной ленты 2 := 0
    Таблица текущих записей : = пробел..пробел
    Таблица количества записей := 0..0
    Переключатель 1 : = выкл
    Переключатель 2 := вкл
Конец модуля
```

Элементы данного модуля почти не связаны друг с другом (за исключением того, что должны выполняться в определенное время). Они все — часть программы запуска системы. Зато элементы более тесно взаимодействуют с другими модулями, что приводит к сложным внешним связям.

Модуль со связностью по времени испытывает те же трудности, что и процедурно связный модуль. Программист соблазняется возможностью совместного использования кода (действиями, которые связаны только по времени), модуль становится трудно использовать повторно.

Так, при желании инициализировать магнитную ленту 2 в другое время вы столкнетесь с неудобствами. Чтобы не сбрасывать всю систему, придется или ввести флажки, указывающие инициализируемую часть, или написать другой код для работы с лентой 2. Оба решения ухудшают сопровождаемость.

Процедурно связные модули и модули с временной связностью очень похожи. Степень их непрозрачности изменяется от темного серого до светло-серого цвета, так как трудно объявить функцию такого модуля без перечисления ее внутренних деталей. Различие между ними подобно различию между информационной и коммуникативной связностью. Порядок выполнения действий более важен в процедурно связных модулях. Кроме того, процедурные модули имеют тенденцию к совместному использованию циклов и ветвлений, а модули с временной связностью чаще содержат более линейный код.

Логическая связность

Элементы логически связного модуля принадлежат к действиям одной категории, и из этой категории клиент выбирает выполняемое действие. Рассмотрим следующий пример:

```
Модуль Пересылка сообщения
    переслать по электронной почте
    переслать по факсу
```

послать в телеконференцию
переслать по ftp-протоколу
Конец модуля

Как видим, логически связный модуль — мешок доступных действий. Действия вынуждены совместно использовать один и тот же интерфейс модуля. В строке вызова модуля значение каждого параметра зависит от используемого действия. При вызове отдельных действий некоторые параметры должны иметь значение пробела, нулевые значения и т. д. (хотя клиент все же должен использовать их и знать их типы).

Действия в логически связном модуле попадают в одну категорию, хотя имеют не только сходства, но и различия. К сожалению, это заставляет программиста «завязывать код действий в узел», ориентируясь на то, что действия совместно используют общие строки кода. Поэтому логически связный модуль имеет:

- уродливый внешний вид с различными параметрами, обеспечивающими, например, четыре вида доступа;
- запутанную внутреннюю структуру со множеством переходов, похожую на волшебный лабиринт.

В итоге модуль становится сложным как для понимания, так и для сопровождения.

Связность по совпадению

Элементы связного по совпадению модуля вообще не имеют никаких отношений друг с другом:

Модуль Разные функции (какие-то параметры)
поздравить с Новым годом (...)
проверить исправность аппаратуры (...)
заполнить анкету героя (...)
измерить температуру (...)
вывести собаку на прогулку (...)
запастись продуктами (...)
приобрести «ягуар» (...)
Конец модуля

Связный по совпадению модуль похож на логически связный модуль. Его элементы-действия не связаны ни потоком данных, ни потоком управления. Но в логически связном модуле действия, по крайней мере, относятся к одной категории; в связном по совпадению модуле даже это не так. Словом, связные по совпадению модули имеют все недостатки логически связных модулей и даже усиливают их. Применение таких модулей вселяет ужас, поскольку один параметр используется для разных целей.

Чтобы клиент мог воспользоваться модулем Разные функции, этот модуль (подобно всем связным по совпадению модулям) должен быть «белым ящиком», чья реализация полностью видима. Такие модули делают системы менее понятными и труднее сопровождаемыми, чем системы без модульности вообще!

К счастью, связность по совпадению встречается редко. Среди ее причин можно назвать:

- бездумный перевод существующего монолитного кода в модули;
- необоснованные изменения модулей с плохой (обычно временной) связностью, приводящие к добавлению флажков.

Определение связности модуля

Приведем алгоритм определения уровня связности модуля.

1. Если модуль — единичная проблемно-ориентированная функция, то уровень связности — функциональный; конец алгоритма. В противном случае перейти к пункту 2.
2. Если действия внутри модуля связаны, то перейти к пункту 3. Если действия внутри модуля никак не связаны, то перейти к пункту 6.
3. Если действия внутри модуля связаны данными, то перейти к пункту 4. Если действия внутри модуля связаны потоком управления, перейти к пункту 5.
4. Если порядок действий внутри модуля важен, то уровень связности — информационный. В противном случае уровень связности — коммуникативный. Конец алгоритма.
5. Если порядок действий внутри модуля важен, то уровень связности — процедурный. В противном случае уровень связности — временной. Конец алгоритма.
6. Если действия внутри модуля принадлежат к одной категории, то уровень связности — логический. Если действия внутри модуля не принадлежат к одной категории, то уровень связности — по совпадению. Конец алгоритма.

Возможны более сложные случаи, когда с модулем ассоциируются несколько уровней связности. В этих случаях следует применять одно из двух правил:

- правило параллельной цепи. Если все действия модуля имеют несколько уровней связности, то модулю присваивают самый сильный уровень связности;
- правило последовательной цепи. Если действия в модуле имеют разные уровни связности, то модулю присваивают самый слабый уровень связности.

Например, модуль может содержать некоторые действия, которые связаны процедурно, а также другие действия, связные по совпадению. В этом случае применяют правило последовательной цепи и в целом модуль считают связным по совпадению.

Сцепление модулей

Сцепление (Coupling) — мера взаимозависимости модулей поданным [58], [70], [77]. Сцепление — внешняя характеристика модуля, которую желательно уменьшать.

Количественно сцепление измеряется степенью сцепления (СЦ). Выделяют 6 типов сцепления.

1. **Сцепление по данным** (СЦ=1). Модуль А вызывает модуль В.

Все входные и выходные параметры вызываемого модуля — простые элементы данных (рис. 4.13).

Рис. 4.13. Сцепление поданным

2. **Сцепление по образцу** (СЦ=3). В качестве параметров используются структуры данных (рис. 4.14).

Рис. 4.14. Сцепление по образцу

3. **Сцепление по управлению** (СЦ=4). Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посылая ему управляющие данные (рис. 4.15).

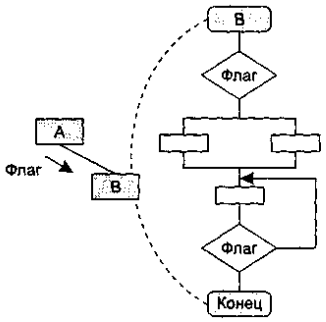


Рис. 4.15. Сцепление по управлению

4. **Сцепление по внешним ссылкам** (СЦ=5). Модули А и В ссылаются на один и тот же глобальный элемент данных.
5. **Сцепление по общей области** (СЦ=7). Модули разделяют одну и ту же глобальную структуру данных (рис. 4.16).
6. **Сцепление по содержанию** (СЦ=9). Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом (рис. 4.16).

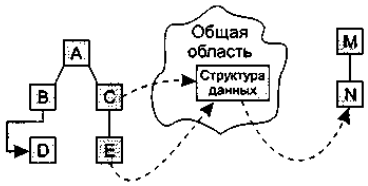


Рис. 4.16. Сцепление по общей области и содержанию

На рис. 4.16 видим, что модули В и D сцеплены по содержанию, а модули С, Е и N сцеплены по общей области.

Сложность программной системы

В простейшем случае сложность системы определяется как сумма мер сложности ее модулей. Сложность модуля может вычисляться различными способами.

Например, М. Холстед (1977) предложил меру длины N модуля [33]:

$$N \approx n_1 log_2 (n_1) + n_2 log_2 (n_2),$$

где n_1 — число различных операторов, n_2 — число различных операндов.

В качестве второй метрики М. Холстед рассматривал объем V модуля (количество символов для записи всех операторов и операндов текста программы):

$$V = N \times log_2 (n_1 + n_2).$$

Вместе с тем известно, что любая сложная система состоит из элементов и системы связей между элементами и что игнорировать внутрисистемные связи неразумно.

Том МакКейб (1976) при оценке сложности ПС предложил исходить из топологии внутренних связей [49]. Для этой цели он разработал метрику цикломатической сложности:

$$V(G) = E\text{-}N + 2,$$

где E — количество дуг, а.N — количество вершин в управляющем графе ПС. Это был шаг в нужном направлении. Дальнейшее уточнение оценок сложности потребовало, чтобы каждый модуль мог представляться как локальная структура, состоящая из элементов и связей между ними.

Таким образом, при комплексной оценке сложности ПС необходимо рассматривать меру сложности модулей, меру сложности внешних связей (между модулями) и меру сложности внутренних связей (внутри модулей) [28], [56]. Традиционно со внешними связями сопоставляют характеристику «сцепление», а с внутренними связями — характеристику «связность».

Вопросы комплексной оценки сложности обсудим в следующем разделе.

Характеристики иерархической структуры программной системы

Иерархическая структура программной системы — основной результат предварительного проектирования. Она определяет состав модулей ПС и управляющие отношения между модулями. В этой структуре модуль более высокого уровня (начальник) управляет модулем нижнего уровня (подчиненным).

Иерархическая структура не отражает процедурные особенности программной системы, то есть последовательность операций, их повторение, ветвления и т. д. Рассмотрим основные характеристики иерархической структуры, представленной на рис. 4.17.

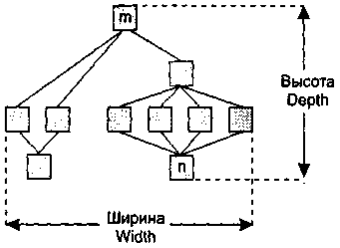


Рис. 4.17. Иерархическая структура программной системы

Первичными характеристиками являются количество вершин (модулей) и количество ребер (связей между модулями). К ним добавляются две глобальные характеристики — высота и ширина:

- **высота** — количество уровней управления;
- **ширина** — максимальное из количеств модулей, размещенных на уровнях управления.

В нашем примере высота = 4, ширина = 6.

Локальными характеристиками модулей структуры являются коэффициент объединения по входу и коэффициент разветвления по выходу.

Коэффициент объединения по входу Fan_in(*i*) — это количество модулей, которые прямо управляют *i*-м модулем.

В примере для модуля n: Fan_in(*n*)=4.

Коэффициент разветвления по выходу Fan_out(*i*) — это количество модулей, которыми прямо управляет *i*-й модуль.

В примере для модуля m: Fan_out(*m*)=3.

Возникает вопрос: как оценить качество структуры? Из практики проектирования известно, что лучшее решение обеспечивается иерархической структурой в виде дерева.

Степень отличия реальной проектной структуры от дерева характеризуется невязкой структуры. Как определить невязку?

Вспомним, что полный граф (complete graph) с *n* вершинами имеет количество ребер

$$e_c = n(n-1)/2,$$

а дерево (tree) с таким же количеством вершин — существенно меньшее количество ребер

$$e_t = n\text{-}1.$$

Тогда формулу невязки можно построить, сравнивая количество ребер полного графа, реального графа и дерева.

Для проектной структуры с *n* вершинами и *e* ребрами невязка определяется по выражению

$$Nev = \frac{e - e_t}{e_c - e_t} = \frac{(e - n + 1) \times 2}{n \times (n-1) - 2 \times (n-1)} = \frac{2 \times (e - n + 1)}{(n-1) \times (n-2)}.$$

Значение невязки лежит в диапазоне от 0 до 1. Если Nev = 0, то проектная структура является деревом, если Nev = 1, то проектная структура — полный граф.

Ясно, что невязка дает грубую оценку структуры. Для увеличения точности оценки следует применить характеристики связности и сцепления.

Хорошая структура должна иметь низкое сцепление и высокую связность.

Л. Констентайн и Э. Йордан (1979) предложили оценивать структуру с помощью коэффициентов Fan_in(*i*) и Fan_out(*i*) модулей [77].

Большое значение Fan_in(*i*) — свидетельство высокого сцепления, так как является мерой зависимости модуля. Большое значение Fan_out(*i*) говорит о высокой сложности вызывающего модуля. Причиной является то, что для координации подчиненных модулей требуется сложная логика управления.

Основной недостаток коэффициентов Fan_in(*i*) и Fan_out(*i*) состоит в игнорировании веса связи. Здесь рассматриваются только управляющие потоки (вызовы модулей). В то же время информационные потоки, нагружающие ребра структуры, могут существенно изменяться, поэтому нужна мера, которая учитывает не только количество ребер, но и количество информации, проходящей через них.

С. Генри и Д. Кафура (1981) ввели информационные коэффициенты ifan_in(*i*) и ifan_out(*j*) [35]. Они учитывают количество элементов и структур данных, из которых *i*-й модуль берет информацию и которые обновляются *j*-м модулем соответственно.

Информационные коэффициенты суммируются со структурными коэффициентами sfan_in(*i*) и sfan_out(*j*), которые учитывают только вызовы модулей.

В результате формируются полные значения коэффициентов:

$$\text{Fan\_in}\ (i) = \text{sfan\_in}\ (i) + \text{ifan\_in}\ (i),$$

$$\text{Fan\_out}\ (j) = \text{sfan\_out}\ (j) + \text{ifan\_out}\ (j).$$

На основе полных коэффициентов модулей вычисляется метрика общей сложности структуры:

$$S = \sum_{i=1}^{n} \text{length}(i) \times (\text{Fan\_in}(i) + \text{Fan\_out}(i))^2,$$

где length(*i*) — оценка размера *i*-го модуля (в виде LOC- или FP-оценки).

Метод проектирования Джексона

Для иллюстрации проектирования по этому методу продолжим пример с системой обслуживания перевозок.

Метод Джексона включает шесть шагов [39]. Три первых шага относятся к этапу анализа. Это шаги: *объект — действие, объект — структура, начальное моделирование.* Их мы уже рассмотрели.

Доопределение функций

Следующий шаг — доопределение функций. Этот шаг развивает диаграмму системной спецификации этапа анализа. Уточняются процессы-модели. В них вводятся дополнительные функции. Джексон выделяет 3 типа сервисных функций:

1. Встроенные функции (задаются командами, вставляемыми в структурный текст процесса-модели).
2. Функции впечатления (наблюдают вектор состояния процесса-модели и вырабатывают выходные результаты).
3. Функции диалога.

Они решают следующие задачи:

- наблюдают вектор состояния процесса-модели;
- формируют и выводят поток данных, влияющий на действия в процессе-модели;
- выполняют операции для выработки некоторых результатов.

Встроенную функцию введем в модель ТРАНСПОРТ-1. Предположим, что в модели есть панель с лампочкой, сигнализирующей о прибытии. Лампочка включается командой LON(i), а выключается командой LOFF(i). По мере перемещения транспорта между остановками формируется поток LAMP-команд. Модифицированный структурный текст модели ТРАНСПОРТ-1 принимает вид

```
ТРАНСПОРТ-1
    LON(l);
    опрос SV;
    ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
            опрос SV;
    конец ЖДАТЬ;
    LOFF(l);
    покинуть(1);
    ТРАНЗИТ цикл ПОКА УБЫЛ(1)
            опрос SV;
    конец ТРАНЗИТ;
    ТРАНСПОРТ цикл
            ОСТАНОВКА;
                    прибыть(i);
                    LON(i);
                    ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)
                            опрос SV;
                    конец ЖДАТЬ;
                    LOFF(i);
                    покинуть(i);
                    ТРАНЗИТ цикл ПОКА УБЫЛ(i)
                            опрос SV;
                    конец ТРАНЗИТ;
            конец ОСТАНОВКА;
    конец ТРАНСПОРТ;
    прибыть(1);
конец ТРАНСПОРТ-1;
```

Теперь введем функцию впечатления. В нашем примере она может формировать команды для мотора транспорта: START, STOP.

Условия выработки этих команд.

- Команда STOP формируется, когда датчики регистрируют прибытие транспорта на остановку.
- Команда START формируется, когда нажата кнопка для запроса транспорта и транспорт ждет на одной из остановок.

Видим, что для выработки команды STOP необходима информация только от модели транспорта. В свою очередь, для выработки команды START нужна информация как от модели КНОПКА-1, так и от модели ТРАНСПОРТ-1. В силу этого для реализации функции впечатления введем функциональный процесс М-УПРАВЛЕНИЕ. Он будет обрабатывать внешние данные и формировать команды START и STOP.

Ясно, что процесс М-УПРАВЛЕНИЕ должен иметь внешние связи с моделями ТРАНСПОРТ-1 и КНОПКА. Соединение с

моделью КНОПКА организуем через вектор состояния BV. Соединение с моделью ТРАНСПОРТ-1 организуем через поток данных S1D.

Для обеспечения М-УПРАВЛЕНИЯ необходимой информацией опять надо изменить структурный текст модели ТРАНСПОРТ-1. В нем предусмотрим занесение сообщения Прибыл в буфер S1D:

```
ТРАНСПОРТ-1
        LON(l);
        опрос SV;
        ЖДАТЬ цикл ПОКА ПРИБЫЛ(1)
                опрос SV;
        конец ЖДАТЬ;
        LOFF(l);
        Покинуть(1);
        ТРАНЗИТ цикл ПОКА УБЫЛ(1)
                опрос SV;
        конец ТРАНЗИТ;
        ТРАНСПОРТ цикл
        ОСТАНОВКА;
                прибыть(i):
                записать Прибыл в S1D;
                LON(i);
                ЖДАТЬ цикл ПОКА ПРИБЫЛ(i)
                        опрос SV;
                конец ЖДАТЬ;
                LOFF(i);
                покинуть(i);
                ТРАНЗИТ цикл ПОКА УБЫЛ(i)
                                опрос SV;
                        конец ТРАНЗИТ;
                конец ОСТАНОВКА;
        конец ТРАНСПОРТ;
        прибыть(1);
        записать Прибыл в S1D;
конец ТРАНСПОРТ-1;
```

Очевидно, что при такой связи процессов необходимо гарантировать, что процесс ТРАНСПОРТ-1 выполняет операции опрос SV, а процесс М-УПРАВЛЕНИЕ читает сообщения Прибытия в S1D с частотой, достаточной для своевременной остановки транспорта. Временные ограничения, планирование и реализация должны рассматриваться в последующих шагах проектирования.

В заключение введем функцию диалога. Свяжем эту функцию с необходимостью развития модели КНОПКА-1. Следует различать первое нажатие на кнопку (оно формирует запрос на поездку) и последующие нажатия на кнопку (до того, как поездка действительно началась).

Диаграмма дополнительного процесса КНОПКА-2, в котором учтено это уточнение, показана на рис. 5.7.

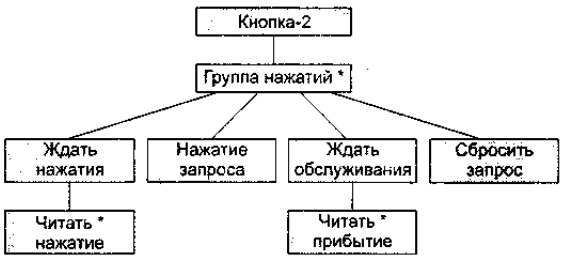


Рис. 5.7. Диаграмма дополнительного процесса КНОПКА-2

Внешние связи модели КНОПКА-2 должны включать:

- одно соединеннее моделью КНОПКА-1 — организуется через поток данных BID (для приема сообщения о нажатии кнопки);
- два соединения с процессом М-УПРАВЛЕНИЕ — одно организуется через поток данных MBD (для приема сообщения о прибытии транспорта), другое организуется через вектор состояния BV (для передачи состояния переключателя Запрос).

Таким образом, КНОПКА-2 читает два буфера данных, заполняемых процессами КНОПКА-1 и М-УПРАВЛЕНИЕ, и формирует состояние внутреннего электронного переключателя Запрос. Она реализует функцию диалога.

Структурный текст модели КНОПКА-2 может иметь следующий вид:

```
КНОПКА-2
        Запрос := НЕТ;
        читать B1D;
        ГрНАЖ цикл
                ЖдатьНАЖ цикл ПОКА Не НАЖАТА
                        читать B1D;
                конец ЖдатьНАЖ;
                Запрос := ДА;
```

читать MBD;
ЖдатьОБСЛУЖ цикл ПОКА Не ПРИБЫЛ
читать MBD;
конец ЖдатьОБСЛУЖ;
Запрос := НЕТ; читать B1D;
конец ГрНАЖ;
конец КНОПКА-2;
Диаграмма системной спецификации, отражающая все изменения, представлена на рис. 5.8.

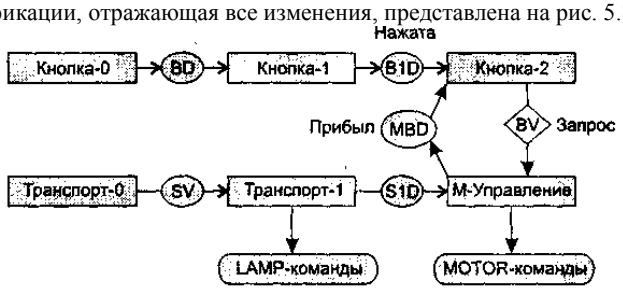


Рис. 5.8. Полная диаграмма системной спецификации

Встроенная в ТРАНСПОРТ-1 функция вырабатывает LAMP-команды, функция впечатления модели М-УПРАВЛЕНИЕ генерирует команды управления мотором, а модель КНОПКА-2 реализует функцию диалога (совместно с процессом М-УПРАВЛЕНИЕ).

Учет системного времени

На шаге учета системного времени проектировщик определяет временные ограничения, накладываемые на систему, фиксирует дисциплину планирования. Дело в том, что на предыдущих шагах проектирования была получена система, составленная из последовательных процессов. Эти процессы связывали только потоки данных, передаваемые через буфер, и взаимные наблюдения векторов состояния. Теперь необходимо произвести дополнительную синхронизацию процессов во времени, учесть влияние внешней программно-аппаратной среды и совместно используемых системных ресурсов.

Временные ограничения для системы обслуживания перевозок, в частности, включают:

- временной интервал на выработку команды STOP; он должен выбираться путем анализа скорости транспорта и ограничения мощности;
- время реакции на включение и выключение ламп панели.

Для рассмотренного примера нет необходимости вводить специальный механизм синхронизации. Однако при расширении может потребоваться некоторая синхронизация обмена данными.

Тестирование «черного ящика»

Известны: функции программы.
Исследуется: работа каждой функции на всей области определения.
Как показано на рис. 6.2, основное место приложения тестов «черного ящика» — интерфейс ПО.

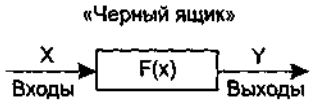


Рис. 6.2. Тестирование «черного ящика»

Эти тесты демонстрируют:

- как выполняются функции программ;
- как принимаются исходные данные;
- как вырабатываются результаты;
- как сохраняется целостность внешней информации.

При тестировании «черного ящика» рассматриваются системные характеристики программ, игнорируется их внутренняя логическая структура. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 10^{10} тестовых вариантов. Отметим также, что тестирование «черного ящика» не реагирует на многие особенности программных ошибок.

Тестирование «белого ящика»

Известна: внутренняя структура программы.
Исследуются: внутренние элементы программы и связи между ними (рис. 6.3).

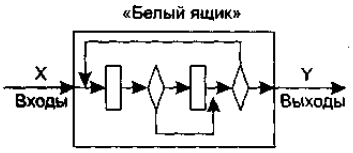


Рис. 6.3. Тестирование «белого ящика»

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование также затруднительно. Особенности этого принципа тестирования рассмотрим отдельно.

Особенности тестирования «белого ящика»

Обычно тестирование «белого ящика» основано на анализе управляющей структуры программы [2], [13]. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов (путей) ее графа управления.

В этом случае формируются тестовые варианты, в которых:

- гарантируется проверка всех независимых маршрутов программы;
- проходятся ветви True, False для всех логических решений;
- выполняются все циклы (в пределах их границ и диапазонов);
- анализируется правильность внутренних структур данных.

Недостатки тестирования «белого ящика»:

1. Количество независимых маршрутов может быть очень велико. Например, если цикл в программе выполняется k раз, а внутри цикла имеется n ветвлений, то количество маршрутов вычисляется по формуле

$$m = \sum_{i=1}^{k} n^i \ .$$

   При $n = 5$ и $k = 20$ количество маршрутов $m = 10^{14}$. Примем, что на разработку, выполнение и оценку теста по одному маршруту расходуется 1 мс. Тогда при работе 24 часа в сутки 365 дней в году на тестирование уйдет 3170 лет.
2. Исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней.
3. В программе могут быть пропущены некоторые маршруты.
4. Нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных (это ошибки, обусловленные выражениями типа if abs (a-b) < eps..., if(a+b+c)/3=a...).

Достоинства тестирования «белого ящика» связаны с тем, что принцип «белого ящика» позволяет учесть особенности программных ошибок:

1. Количество ошибок минимально в «центре» и максимально на «периферии» программы.
2. Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.
3. При записи алгоритма ПО в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).
4. Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.

Каждая из этих причин является аргументом для проведения тестирования по принципу «белого ящика». Тесты «черного ящика» не смогут реагировать на ошибки таких типов.

Способ тестирования базового пути

Тестирование базового пути — это способ, который основан на принципе «белого ящика». Автор этого способа — Том МакКейб (1976) [49].

Способ тестирования базового пути дает возможность:

- получить оценку комплексной сложности программы;
- использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

Потоковый граф

Для представления программы используется потоковый граф. Перечислим его особенности.

1. Граф строится отображением управляющей структуры программы. В ходе отображения закрывающие скобки условных операторов и операторов циклов (end if; end loop) рассматриваются как отдельные (фиктивные) операторы.
2. Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько

операторов программы.

3. Дуги потокового графа отображают поток управления в программе (передачи управления между операторами). Дуга — это ориентированное ребро.
4. Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного — две дуги.
4. Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в несколько предикатных узлов. Составным называют условие, в котором используется одна или несколько булевых операций (OR, AND).
5. Например, фрагмент программы

```
if a OR b
      then x
      else y
end if;
```

вместо прямого отображения в потоковый граф вида, показанного на рис. 6.4, отображается в преобразованный потоковый граф (рис. 6.5).

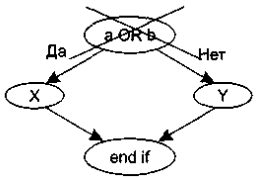


Рис. 6.4. Прямое отображение в потоковый граф

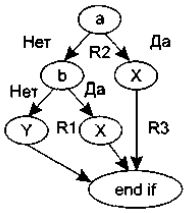


Рис. 6.5. Преобразованный потоковый граф

6. Замкнутые области, образованные дугами и узлами, называют регионами.

7. Окружающая граф среда рассматривается как дополнительный регион. Например, показанный здесь граф имеет три региона — Rl, R2, R3.

Пример 1. Рассмотрим процедуру сжатия:

```
      процедура сжатие
1           выполнять пока нет EOF
1                 читать запись;
2                 если запись пуста
3                 то удалить запись:
4                 иначе если поле a >= поля b
5                       то удалить b;
6                       иначе удалить a;
7a                конец если;
7a          конец если;
7b          конец выполнять;
8     конец сжатие;
```

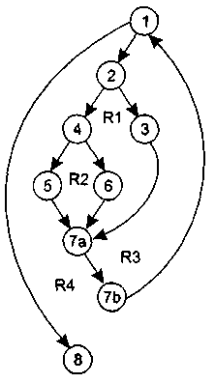


Рис. 6.6. Преобразованный потоковый граф процедуры сжатия

Она отображается в потоковый граф, представленный на рис. 6.6. Видим, что этот потоковый граф имеет четыре региона.

Цикломатическая сложность

Цикломатическая сложность — метрика ПО, которая обеспечивает количественную оценку логической сложности программы. В способе тестирования базового пути Цикломатическая сложность определяет:

- количество независимых путей в базовом множестве программы;
- верхнюю оценку количества тестов, которое гарантирует однократное выполнение всех операторов.

Независимым называется любой путь, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути.

ПРИМЕЧАНИЕ

Путь начинается в начальном узле, а заканчивается в конечном узле графа. Независимые пути формируются в порядке от самого короткого к самому длинному.

Перечислим независимые пути для потокового графа из примера 1:

Путь 1: 1-8.

Путь 2: 1-2-3-7a-7b-1-8.

Путь 3: 1-2-4-5-7a-7b-1-8.

Путь 4: 1-2-4-6-7a-7b-1-8.

Заметим, что каждый новый путь включает новую дугу.

Все независимые пути графа образуют базовое множество.

Свойства базового множества:

1) тесты, обеспечивающие его проверку, гарантируют:

- однократное выполнение каждого оператора;
- выполнение каждого условия по True-ветви и по False-ветви;

2) мощность базового множества равна цикломатической сложности потокового графа.

Значение 2-го свойства трудно переоценить — оно дает априорную оценку количества независимых путей, которое имеет смысл искать в графе.

Цикломатическая сложность вычисляется одним из трех способов:

1) цикломатическая сложность равна количеству регионов потокового графа;

2) цикломатическая сложность определяется по формуле

$$V(G)=E-N+2,$$

где E — количество дуг, N — количество узлов потокового графа;

3) цикломатическая сложность формируется по выражению $V(G)=p+1$, где p — количество предикатных узлов в потоковом графе G.

Вычислим цикломатическую сложность графа из примера 1 каждым из трех способов:

1) потоковый граф имеет 4 региона;

2) $V(G)$ = 11 дуг - 9 узлов + 2 = 4;

3) $V(G)$ = 3 предикатных узла +1=4.

Таким образом, цикломатическая сложность потокового графа из примера 1 равна четырем.

Шаги способа тестирования базового пути

Для иллюстрации шагов данного способа используем конкретную программу — процедуру вычисления среднего значения:

```
        процедура сред;
1               i := 1;
1               введено := 0;
1               колич := 0;
1               сум := 0;
      вып пока 2 -вел( i ) <> stop и введено <=500 - 3
4                       введено:= введено + 1;
      если 5 -вел( i ) >= мин и вел( i ) <= макс - 6
7                               то колич := колич + 1;
7                               сум := сум + вел( i );
8                       конец если;
8                       i := i + 1;
9     конец вып;
10    если колич > 0
11              то сред := сум / колич;
12    иначе сред := stop;
13    конец если;
13    конец сред;
```

Заметим, что процедура содержит составные условия (в заголовке цикла и условном операторе). Элементы составных условий для наглядности помещены в рамки.

Шаг 1. На основе текста программы формируется потоковый граф:

- нумеруются операторы текста (номера операторов показаны в тексте процедуры);
- производится отображение пронумерованного текста программы в узлы и вершины потокового графа (рис. 6.7).

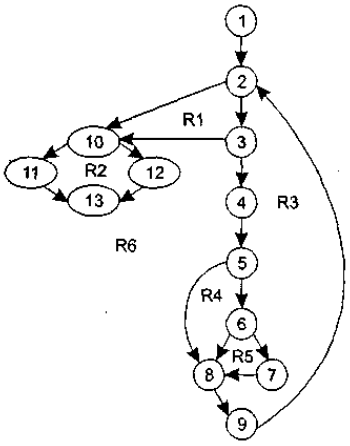


Рис. 6.7. Потоковый граф процедуры вычисления среднего значения

Шаг 2. Определяется цикломатическая сложность потокового графа — по каждой из трех формул:
1) *V(G)* = 6 регионов;
2) *V(G)* = 17 дуг - 13 узлов + 2 = 6;
3) *V(G)* = 5 предикатных узлов + 1 = 6.
Шаг 3. Определяется базовое множество независимых линейных путей:
Путь 1: 1-2-10-11-13; /вел=stop, колич>0.
Путь 2: 1-2-10-12-13;/вел=stop, колич=0.
Путь 3: 1-2-3-10-11-13; /попытка обработки 501-й величины.
Путь 4: 1-2-3-4-5-8-9-2-... /вел<мин.
Путь 5: 1-2-3-4-5-6-8-9-2-... /вел>макс.
Путь 6: 1-2-3-4-5-6-7-8-9-2-... /режим нормальной обработки.

ПРИМЕЧАНИЕ
Для удобства дальнейшего анализа по каждому пути указаны условия запуска. Точки в конце путей 4, 5, 6 указывают, что допускается любое продолжение через остаток управляющей структуры графа.

Шаг 4. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути.
Каждый тестовый вариант формируется в следующем виде:
Исходные данные (ИД):
Ожидаемые результаты (ОЖ.РЕЗ.):
Исходные данные должны выбираться так, чтобы предикатные вершины обеспечивали нужные переключения — запуск только тех операторов, которые перечислены в конкретном пути, причем в требуемом порядке.
Определим тестовые варианты, удовлетворяющие выявленному множеству независимых путей.
Тестовый вариант для пути 1 **ТВ1**:
ИД: вел(*k*) = допустимое значение, где $k < i$; вел(*i*) = stop, где $2 < i < 500$.
ОЖ.РЕЗ.: корректное усреднение основывается на *k* величинах и правильном подсчете.

ПРИМЕЧАНИЕ
Путь не может тестироваться самостоятельно, а должен тестироваться как часть путей 4, 5, 6 (трудности проверки 11-го оператора).

Тестовый вариант для пути 2 **ТВ2**:
ИД: вел(1)=stop.
ОЖ.РЕЗ.: сред=stop, другие величины имеют начальные значения.
Тестовый вариант для пути 3 **ТВ3**:
ИД: попытка обработки 501-й величины, первые 500 величин должны быть правильными.
ОЖ.РЕЗ.: корректное усреднение основывается на *k* величинах и правильном подсчете.
Тестовый вариант для пути 4 **ТВ4**:
ИД: вел(*i*)=допустимое значение, где $i \le 500$; вел(*k*) < мин, где $k < i$.
ОЖ.РЕЗ.: корректное усреднение основывается на *k* величинах и правильном подсчете.
Тестовый вариант для пути 5 **ТВ5**:
ИД: вел(*i*)=допустимое значение, где $i \le 500$; вел(*k*) > макс, где $k < i$.
ОЖ.РЕЗ.: корректное усреднение основывается на *n* величинах и правильном подсчете.
Тестовый вариант для пути 6 **ТВ6**:
ИД: вел(*i*)=допустимое значение, где $i \le 500$.
ОЖ.РЕЗ.: корректное усреднение основывается на *n* величинах и правильном подсчете.
Реальные результаты каждого тестового варианта сравниваются с ожидаемыми результатами. После выполнения всех тестовых вариантов гарантируется, что все операторы программы выполнены по меньшей мере один раз.
Важно отметить, что некоторые независимые пути не могут проверяться изолированно. Такие пути должны проверяться при тестировании другого пути (как часть другого тестового варианта).

Способы тестирования условий

Цель этого семейства способов тестирования — строить тестовые варианты для проверки логических условий программы. При этом желательно обеспечить охват операторов из всех ветвей программы.

Рассмотрим используемую здесь терминологию.

Простое условие — булева переменная или выражение отношения.

Выражение отношения имеет вид

E1 <оператор отношения> E2,

где El, E2 — арифметические выражения, а в качестве оператора отношения используется один из следующих операторов: <, >, =, $\neq$, $\leq$, $\geq$.

Составное условие состоит из нескольких простых условий, булевых операторов и круглых скобок. Будем применять булевы операторы OR, AND (&), NOT. Условия, не содержащие выражений отношения, называют булевыми выражениями.

Таким образом, элементами условия являются: булев оператор, булева переменная, пара скобок (заключающая простое или составное условие), оператор отношения, арифметическое выражение. Эти элементы определяют типы ошибок в условиях.

Если условие некорректно, то некорректен по меньшей мере один из элементов условия. Следовательно, в условии возможны следующие типы ошибок:

- ошибка булева оператора (наличие некорректных / отсутствующих / избыточных булевых операторов);
- ошибка булевой переменной;
- ошибка булевой скобки;
- ошибка оператора отношения;
- ошибка арифметического выражения.

Способ тестирования условий ориентирован на тестирование каждого условия в программе. Методики тестирования условий имеют два достоинства. Во-первых, достаточно просто выполнить измерение тестового покрытия условия. Во-вторых, тестовое покрытие условий в программе — это фундамент для генерации дополнительных тестов программы.

Целью тестирования условий является определение не только ошибок в условиях, но и других ошибок в программах. Если набор тестов для программы А эффективен для обнаружения ошибок в условиях, содержащихся в А, то вероятно, что этот набор также эффективен для обнаружения других ошибок в А. Кроме того, если методика тестирования эффективна для обнаружения ошибок в условии, то вероятно, что эта методика будет эффективна для обнаружения ошибок в программе.

Существует несколько методик тестирования условий.

Простейшая методика — *тестирование ветвей.* Здесь для составного условия *С* проверяется:

- каждое простое условие (входящее в него);
- True-ветвь;
- False-ветвь.

Другая методика — *тестирование области определения.* В ней для выражения отношения требуется генерация 3-4 тестов. Выражение вида

E1 <оператор отношения> E2

проверяется тремя тестами, которые формируют значение E1 большим, чем E2, равным E2 и меньшим, чем E2.

Если оператор отношения неправилен, а E1 и E2 корректны, то эти три теста гарантируют обнаружение ошибки оператора отношения.

Для определения ошибок в E1 и E2 тест должен сформировать значение E1 большим или меньшим, чем E2, причем обеспечить как можно меньшую разницу между этими значениями.

Для булевых выражений с n переменными требуется набор из 2^n тестов. Этот набор позволяет обнаружить ошибки булевых операторов, переменных и скобок, но практичен только при малом n. Впрочем, если в булево выражение каждая булева переменная входит только один раз, то количество тестов легко уменьшается.

Обсудим способ тестирования условий, базирующийся на приведенных выше методиках.

Тестирование ветвей и операторов отношений

Способ тестирования ветвей и операторов отношений (автор К. Таи, 1989) обнаруживает ошибки ветвления и операторов отношения в условии, для которого выполняются следующие ограничения [72]:

- все булевы переменные и операторы отношения входят в условие только по одному разу;
- в условии нет общих переменных.

В данном способе используются естественные ограничения условий (ограничения на результат). Для составного условия *С,* включающего *n* простых условий, формируется ограничение условия:

$$ОУ_c = (d_1,d_2,d_3.....d_n),$$

где d_i — ограничение на результат *i*-го простого условия.

Ограничение на результат фиксирует возможные значения аргумента (переменной) простого условия (если он один) или соотношения между значениями аргументов (если их несколько).

Если i-е простое условие является булевой переменной, то его ограничение на результат состоит из двух значений и имеет вид

$$d_i = (\text{true, false}).$$

Если *j*-е простое условие является выражением отношения, то его ограничение на результат состоит из трех значений и имеет следующий вид:

$$d_j = (>,<,=).$$

Говорят, что ограничение условия $ОУ_c$ (для условия *С*) покрывается выполнением *С,* если в ходе этого выполнения результат каждого простого условия в *С* удовлетворяет соответствующему ограничению в $ОУ_c$.

На основе ограничения условия ОУ создается ограничивающее множество ОМ, элементы которого являются сочетаниями

всех возможных значений $d_1, d_2, d_3, ..., d_n$.

Ограничивающее множество — удобный инструмент для записи задания на тестирование, ведь оно составляется из сведений о значениях переменных, которые влияют на значение проверяемого условия. Поясним это на примере. Положим, надо проверить условие, составленное из трех простых условий:

$$b\&(x>y)\&a.$$

Условие принимает истинное значение, если все простые условия истинны. В терминах значений простых условий это соответствует записи

(true, true, true),

а в терминах ограничений на значения аргументов простых условий — записи

(true, >, true).

Ясно, что вторая запись является прямым руководством для написания теста. Она указывает, что переменная *b* должна иметь истинное значение, значение переменной *x* должно быть больше значения переменной *y,* и, наконец, переменная *a* должна иметь истинное значение.

Итак, каждый элемент ОМ задает отдельный тестовый вариант. Исходные данные тестового варианта должны обеспечить соответствующую комбинацию значений простых условий, а ожидаемый результат равен значению составного условия.

Пример 1. В качестве примера рассмотрим два типовых составных условия:

$$C_{\&} = a \ \& \ b, \ C_{or} = a \ or \ b,$$

где *a* и *b* — булевы переменные. Соответствующие ограничения условий принимают вид

$$\text{ОУ}_{\&}=(d_1,d_2), \ \text{ОУ}_{or}=(d_1,d_2),$$

где $d_1 = d_2$ = (true, false).

Ограничивающие множества удобно строить с помощью таблицы истинности (табл. 6.1).

Таблица 6.1. Таблица истинности логических операций

Вариант	***a***	***b***	***a & b***	***a or b***
1	false	false	false	false
2	false	true	false	true
3	true	false	false	true
4	true	true	true	true

Видим, что таблица задает в ОМ четыре элемента (и соответственно, четыре тестовых варианта). Зададим вопрос — каковы возможности минимизации? Можно ли уменьшить количество элементов в ОМ?

С точки зрения тестирования, нам надо оценивать влияние составного условия на программу. Составное условие может принимать только два значения, но каждое из значений зависит от большого количества простых условий. Стоит задача — избавиться от влияния избыточных сочетаний значений простых условий.

Воспользуемся идеей сокращенной схемы вычисления — элементы выражения вычисляются до тех пор, пока они влияют на значение выражения. При тестировании необходимо выявить ошибки переключения, то есть ошибки из-за булева оператора, оперируя значениями простых условий (булевых переменных). При таком инженерном подходе справедливы следующие выводы:

- для условия типа И *(a & b)* варианты 2 и 3 поглощают вариант 1. Поэтому ограничивающее множество имеет вид:
$$\text{ОМ}_{\&} = \{(\text{false, true}), (\text{true, false}), (\text{true, true})\};$$
- для условия типа ИЛИ *(a* or *b)* варианты 2 и 3 поглощают вариант 4. Поэтому ограничивающее множество имеет вид:
$$\text{ОМ}_{or} = \{(\text{false, false}), (\text{false, true}), (\text{true, false})\}.$$

Рассмотрим шаги **способа тестирования ветвей и операторов отношений.**

Для каждого условия в программе выполняются следующие действия:

1) строится ограничение условий ОУ;
2) выявляются ограничения результата по каждому простому условию;
3) строится ограничивающее множество ОМ. Построение выполняется путем подстановки в константные формулы $\text{ОМ}_{\&}$ или ОМ_{OR} выявленных ограничений результата;
4) для каждого элемента ОМ разрабатывается тестовый вариант.

Пример 2. Рассмотрим составное условие *C*1 вида:

$$B_1 \ \&(E_1,E_2),$$

где B_1 — булево выражение, E_1, E_2 — арифметические выражения.

Ограничение составного условия имеет вид

$$\text{ОУ}_{C_1} = (d_1,d_2),$$

где ограничения простых условий равны

$$d_1 = (\text{true, false}), \ d_2 = (=, <, >).$$

Проводя аналогию между C_1 и $C_{\&}$ (разница лишь в том, что в C_1 второе простое условие — это выражение отношения), мы можем построить ограничивающее множество для C_1 модификацией

$$\text{ОМ}_{\&} = \{(\text{false, true}), (\text{true, false}), (\text{true, true})\}.$$

Заметим, что *true* для $(E_1 = E_2)$ означает =, а false для $(E_1 = E_2)$ означает или <, или >. Заменяя (true, true) и (false, true), ограничениями (true, =) и (false, =) соответственно, а (true, false) — ограничениями (true, <) и (true, >), получаем ограничивающее множество для C_1:

$$\text{ОМ}_{C_1} = \{(\text{false},=),(\text{true},<),(\text{true},>),(\text{true},=)\}.$$

Покрытие этого множества гарантирует обнаружение ошибок булевых операторов и операторов отношения в C_1.

Пример 3. Рассмотрим составное условие C_2 вида

$$(E_3 > E_4)\&(E_1 = E_2),$$

где E_1, E_2, E_3, E_4 — арифметические выражения. Ограничение составного условия имеет вид

$$\text{ОУ}_{C_2} = (d_1, d_2),$$

где ограничения простых условий равны

$$d_1 = (=, <, >),\ d_2 = (=, <, >).$$

Проводя аналогию между C_2 и C_1 (разница лишь в том, что в C_2 первое простое условие — это выражение отношения), мы можем построить ограничивающее множество для C_2 модификацией ОМ_{C_1}:

$$\text{ОМ}_{C_2} = \{(=, =), (<, =), (>, <), (>, >), (>, =)\}.$$

Покрытие этого ограничивающего множества гарантирует обнаружение ошибок операторов отношения в C_2.

Способ тестирования потоков данных

В предыдущих способах тесты строились на основе анализа управляющей структуры программы. В данном способе анализу подвергается информационная структура программы.

Работу любой программы можно рассматривать как обработку потока данных, передаваемых от входа в программу к ее выходу.

Рассмотрим пример.

Пусть потоковый граф программы имеет вид, представленный на рис. 6.8. В нем сплошные дуги — это связи по управлению между операторами в программе. Пунктирные дуги отмечают информационные связи (связи по потокам данных). Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в вершине 1 определяются значения переменных *a, b;*
- значение переменной *a* используется в вершине 4;
- значение переменной *b* используется в вершинах 3, 6;
- в вершине 4 определяется значение переменной c, которая используется в вершине 6.

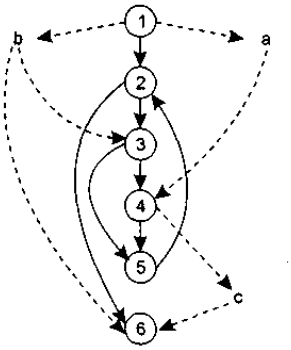


Рис. 6.8. Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

- множество определений данных

DEF(i) = { x | i -я вершина содержит определение x};

- множество использований данных:

USE (i) = { x | i -я вершина использует x}.

Под *определением данных* понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

$$x := f(\ldots).$$

Использование данных — это применение элемента в выражении, где происходит обращение к элементу данных, но не изменение элемента. Признак использования — имя элемента стоит в правой части оператора присваивания:

$$\square := f(x).$$

Здесь место подстановки другого имени отмечено прямоугольником (прямоугольник играет роль метки-заполнителя).

Назовём *DU-цепочкой (цепочкой определения-использования)* конструкцию *[x, i,j],* где *i,j* — имена вершин; x определена в *i*-й вершине $(x \in \text{DEF}(i))$ и используется в *j* -й вершине $(x \in \text{USE}(j))$.

В нашем примере существуют следующие DU-цепочки:

[a,1,4],[b, 1,3], [b, 1,6], [c, 4, 6].

Способ *DU-тестирования* требует охвата всех DU-цепочек программы. Таким образом, разработка тестов здесь проводится на основе анализа жизни всех данных программы.

Очевидно, что для подготовки тестов требуется выделение маршрутов — путей выполнения программы на управляющем графе. Критерий для выбора пути — покрытие максимального количества DU-цепочек.

Шаги способа DU-тестирования:

1) построение управляющего графа (УГ) программы;
2) построение информационного графа (ИГ);
3) формирование полного набора DU-цепочек;
4) формирование полного набора отрезков путей в управляющем графе (отображением набора DU-цепочек информационного графа, рис. 6.9);

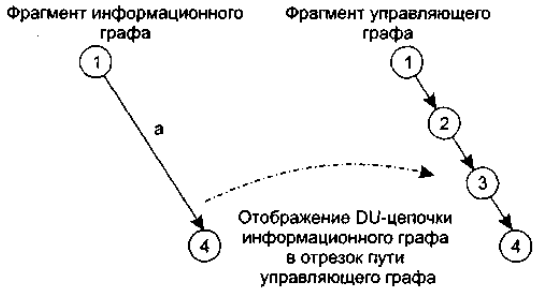


Рис. 6.9. Отображение DU-цепочки *в* отрезок пути

5) построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа;
6) подготовка тестовых вариантов.

Достоинства DU-тестирования:

- простота необходимого анализа операционно-управляющей структуры программы;
- простота автоматизации.

Недостаток DU-тестирования: трудности в выборе минимального количества максимально эффективных тестов.

Область использования DU-тестирования: программы с вложенными условными операторами и операторами цикла.

Тестирование циклов

Цикл — наиболее распространенная конструкция алгоритмов, реализуемых в ПО. Тестирование циклов производится по принципу «белого ящика», при проверке циклов основное внимание обращается на правильность конструкций циклов.

Различают 4 типа циклов: простые, вложенные, объединенные, неструктурированные. Структура циклов приведена на рис. 6.10.

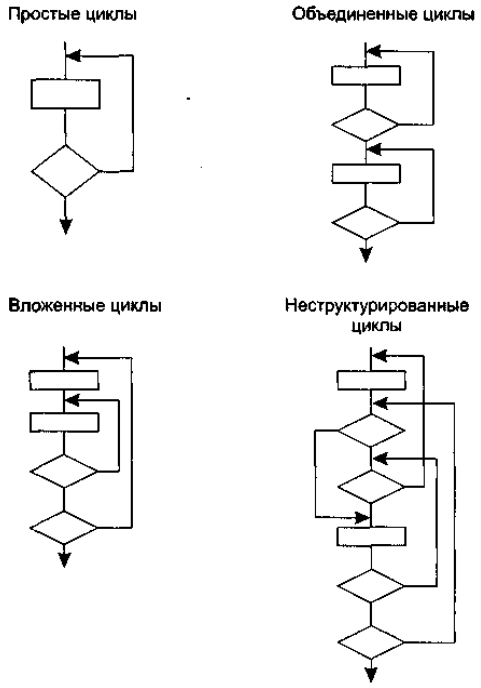


Рис. 6.10. Типовые структуры циклов

Простые циклы

Для проверки простых циклов с количеством повторений *n* может использоваться один из следующих наборов тестов:

1) прогон всего цикла;
2) только один проход цикла;
3) два прохода цикла;
4) *m* проходов цикла, где $m<n$;
5) $n - 1$, n, $n + 1$ проходов цикла.

Вложенные циклы

С увеличением уровня вложенности циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов [13]. Для сокращения количества тестов применяется специальная методика, в которой используются такие понятия, как объемлющий и вложенный циклы (рис. 6.11).

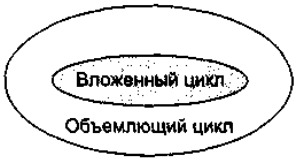


Рис. 6.11. Объемлющий и вложенный циклы

Порядок тестирования вложенных циклов иллюстрирует рис. 6.12.

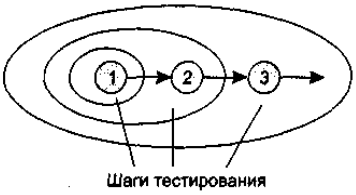


Рис. 6.12. Шаги тестирования вложенных циклов

Шаги тестирования.

1. Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.
2. Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.
3. Переходят в следующий по порядку объемлющий цикл. Выполняют его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.
4. Работа продолжается до тех пор, пока не будут протестированы все циклы.

Объединенные циклы

Если каждый из циклов независим от других, то используется техника тестирования простых циклов. При наличии зависимости (например, конечное значение счетчика первого цикла используется как начальное значение счетчика второго цикла) используется методика для вложенных циклов.

Неструктурированные циклы

Неструктурированные циклы тестированию не подлежат. Этот тип циклов должен быть переделан с помощью структурированных программных конструкций.

Искусство отладки

Отладка — это локализация и устранение ошибок. Отладка является следствием успешного тестирования. Это значит, что если тестовый вариант обнаруживает ошибку, то процесс отладки уничтожает ее.

Итак, процессу отладки предшествует выполнение тестового варианта. Его результаты оцениваются, регистрируется несоответствие между ожидаемым и реальным результатами. Несоответствие является симптомом скрытой причины. Процесс отладки пытается сопоставить симптом с причиной, вследствие чего приводит к исправлению ошибки. Возможны два исхода процесса отладки:

1) причина найдена, исправлена, уничтожена;
2) причина не найдена.

Во втором случае отладчик может предполагать причину. Для проверки этой причины он просит разработать дополнительный тестовый вариант, который поможет проверить предположение. Таким образом, запускается итерационный процесс коррекции ошибки.

Возможные разные способы проявления ошибок:

1) программа завершается нормально, но выдает неверные результаты;
2) программа зависает;
3) программа завершается по прерыванию;
4) программа завершается, выдает ожидаемые результаты, но хранимые данные испорчены (это самый неприятный вариант).

Характер проявления ошибок также может меняться. Симптом ошибки может быть:

- постоянным;
- мерцающим;
- пороговым (проявляется при превышении некоторого порога в обработке — 200 самолетов на экране отслеживаются, а 201-й — нет);
- отложенным (проявляется только после исправления маскирующих ошибок).

В ходе отладки мы встречаем ошибки в широком диапазоне: от мелких неприятностей до катастроф. Следствием увеличения ошибок является усиление давления на отладчика — «найди ошибки быстрее!!!». Часто из-за этого давления разработчик устраняет одну ошибку и вносит две новые ошибки.

Английский термин *debugging* (отладка) дословно переводится как «ловля блох», который отражает специфику процесса — погоню за объектами отладки, «блохами». Рассмотрим, как может быть организован этот процесс «ловли блох» [3], [64].

Различают две группы методов отладки:

- аналитические;
- экспериментальные.

Аналитические методы базируются на анализе выходных данных для тестовых прогонов. Экспериментальные методы базируются на использовании вспомогательных средств отладки (отладочные печати, трассировки), позволяющих уточнить характер поведения программы при тех или иных исходных данных.

Общая стратегия отладки — обратное прохождение от замеченного симптома ошибки к исходной аномалии (месту в программе, где ошибка совершена).

В простейшем случае место проявления симптома и ошибочный фрагмент совпадают. Но чаще всего они далеко отстоят друг от друга.

Цель отладки — найти оператор программы, при исполнении которого правильные аргументы приводят к неправильным результатам. Если место проявления симптома ошибки не является искомой аномалией, то один из аргументов оператора должен быть неверным. Поэтому надо перейти к исследованию предыдущего оператора, выработавшего этот неверный аргумент. В итоге пошаговое обратное прослеживание приводит к искомому ошибочному месту.

В разных методах прослеживание организуется по-разному. В аналитических методах — на основе логических заключений о поведении программы. Цель — шаг за шагом уменьшать область программы, подозреваемую в наличии ошибки. Здесь определяется корреляция между значениями выходных данных и особенностями поведения.

Основное *преимущество аналитических методов отладки* состоит в том, что исходная программа остается без изменений.

В экспериментальных методах для прослеживания выполняется:

1. Выдача значений переменных в указанных точках.
2. Трассировка переменных (выдача их значений при каждом изменении).
3. Трассировка потоков управления (имен вызываемых процедур, меток, на которые передается управление, номеров операторов перехода).

Преимущество экспериментальных методов отладки состоит в том, что основная рутинная работа по анализу процесса вычислений перекладывается на компьютер. Многие трансляторы имеют встроенные средства отладки для получения информации о ходе выполнения программы.

Недостаток экспериментальных методов отладки — в программу вносятся изменения, при исключении которых могут появиться ошибки. Впрочем, некоторые системы программирования создают специальный отладочный экземпляр программы, а в основной экземпляр не вмешиваются.

Примеры диаграмм классов

В качестве первого примера на рис. 11.17 показана диаграмма классов системы управления полетом летательного аппарата.

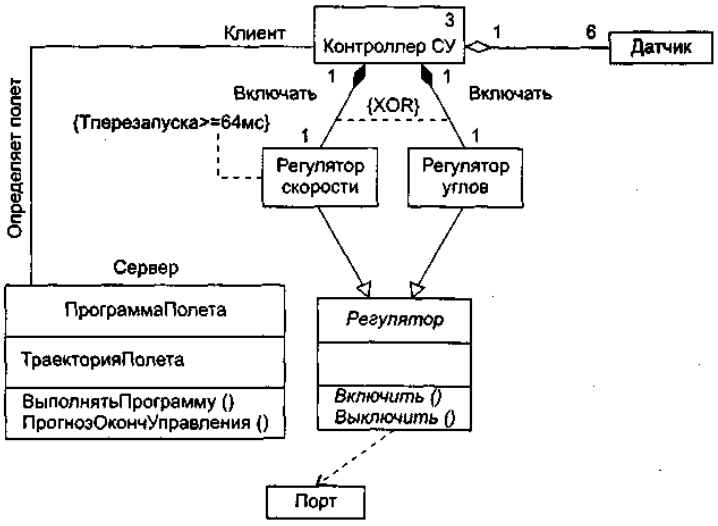


Рис. 11.17. Диаграмма классов системы управления полетом

Здесь представлен класс ПрограммаПолета, который имеет свойство ТраекторияПолета, операцию-модификатор ВыполнятьПрограмму () и операцию-селектор ПрогнозОкончУправления (). Имеется ассоциация между этим классом и классом Контроллер СУ — экземпляры программы задают параметры движения, которые должны обеспечивать экземпляры контроллера.

Класс Контроллер СУ — агрегат, чьи экземпляры включают по одному экземпляру классов Регулятор скорости и Регулятор углов, а также по шесть экземпляров класса Датчик. Экземпляры Регулятора скорости и Регулятора углов включены в агрегат физически (с помощью отношения *композиция),* а экземпляры Датчика — по ссылке, то есть экземпляр Контроллера СУ включает лишь указатели на объекты-датчики. Регулятор скорости и Регулятор углов — это подклассы абстрактного суперкласса *Регулятор,* который передает им в наследство абстрактные операции *Включить ( )* и *Выключить ()*. В свою очередь, класс *Регулятор* использует конкретный класс Порт.

Как видим, ассоциация имеет имя (Определяет полет), роли участников ассоциации явно указаны (Сервер, Клиент). Отношения композиции также имеют имена (Включать), причем на эти отношения наложено ограничение — контроллер не может включать Регулятор скорости и Регулятор углов одновременно.

Для класса Контроллер СУ задано ограничение на множественность — допускается не более трех экземпляров этого класса. Класс Регулятор скорости имеет ограничение другого типа — повторное включение его экземпляра разрешается не раньше, чем через 64 мс.

В качестве второго примера на рис. 11.18 приведена диаграмма классов для информационной системы театра. Эту систему образует 6 классов.

Классы-агрегаты Театр и Труппа имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке. Частями Театра являются Зрители и Труппы, а частями Труппы — Актеры. Отношения агрегации между классом Театр и классами Труппа и Зритель слегка отличны. Театр может состоять из одной или нескольких трупп, но каждая труппа находится в одном и только одном театре. С другой стороны, в театр может ходить любое количество зрителей (включая нулевое количество), причем зритель может посещать один или несколько театров.

Между классами Труппа и Актер существуют два отношения — агрегация и ассоциация. Агрегация показывает, что каждый актер работает в одной или нескольких труппах, а в каждой труппе должен быть хотя бы один актер. Ассоциация отображает, что каждой труппой управляет только один актер — художественный руководитель, а некоторые актеры не являются руководителями.

Ассоциация между классами Спектакль и Актер фиксирует, что в спектакле должен быть занят хотя бы один актер, впрочем, актер может играть в любом количестве спектаклей (или вообще может ничего не играть).

Между классами Спектакль и Зритель тоже определена ассоциация. Она поясняет, что зритель может смотреть любое число спектаклей, а на каждом спектакле может быть любое число зрителей.

И наконец, на диаграмме отображены два отношения наследования, утверждающие, что и в зрителях, и в актерах есть человеческое начало.

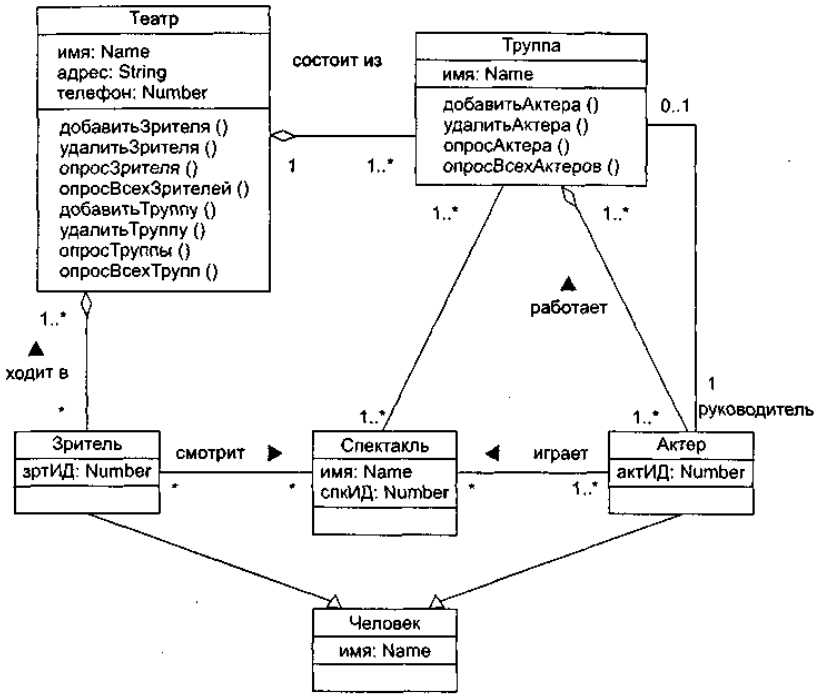


Рис. 11.18. Диаграмма классов информационной системы театра

Диаграммы Use Case

Диаграмма Use Case определяет поведение системы с точки зрения пользователя. Диаграмма Use Case рассматривается как главное средство для первичного моделирования динамики системы, используется для выяснения требований к разрабатываемой системе, фиксации этих требований в форме, которая позволит проводить дальнейшую разработку. В

русской литературе диаграммы Use Case часто называют диаграммами прецедентов, или диаграммами вариантов использования.

В состав диаграмм Use Case входят элементы Use Case, актеры, а также отношения зависимости, обобщения и ассоциации. Как и другие диаграммы, диаграммы Use Case могут включать примечания и ограничения. Кроме того, диаграммы Use Case могут содержать пакеты, используемые для группировки элементов модели в крупные фрагменты.

Актеры и элементы Use Case

Вершинами в диаграмме Use Case являются актеры и элементы Use Case. Их обозначения показаны на рис. 12.26.

Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.

Рис. 12.26. Обозначения актера и элемента Use Case

Актер — это роль объекта вне системы, который прямо взаимодействует с ее частью — конкретным элементом (элементом Use Case). Различают актеров и пользователей. Пользователь — это физический объект, который использует систему. Он может играть несколько ролей и поэтому может моделироваться несколькими актерами. Справедливо и обратное — актером могут быть разные пользователи.

Например, для коммерческого летательного аппарата можно выделить двух актеров: пилота и кассира. Сидоров — пользователь, который иногда действует как пилот, а иногда — как кассир. Как изображено на рис. 12.27, в зависимости от роли Сидоров взаимодействует с разными элементами Use Case.

Рис. 12.27. Модель Use Case

Элемент Use Case — это описание последовательности действий (или нескольких последовательностей), которые выполняются системой и производят для отдельного актера видимый результат.

Один актер может использовать несколько элементов Use Case, и наоборот, один элемент Use Case может иметь несколько актеров, использующих его. Каждый элемент Use Case задает определенный путь использования системы. Набор всех элементов Use Case определяет полные функциональные возможности системы.

Отношения в диаграммах Use Case

Между актером и элементом Use Case возможен только один вид отношения — ассоциация, отображающая их взаимодействие (рис. 12.28). Как и любая другая ассоциация, она может быть помечена именем, ролями, мощностью.

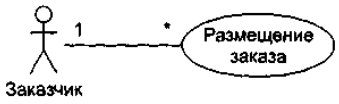


Рис. 12.28. Отношение ассоциации

Между актерами допустимо отношение обобщения (рис. 12.29), означающее, что экземпляр потомка может взаимодействовать с такими же разновидностями экземпляров элементов Use Case, что и экземпляр родителя.

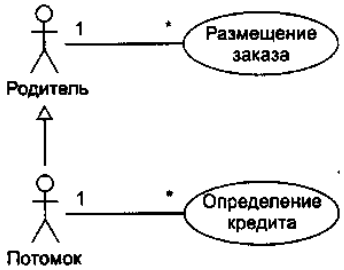


Рис. 12.29. Отношение обобщения между актерами

Между элементами Use Case определены отношение обобщения и две разновидности отношения зависимости — включения и расширения.

Отношение обобщения (рис. 12.30) фиксирует, что потомок наследует поведение родителя. Кроме того, потомок может дополнить или переопределить поведение родителя. Элемент Use Case, являющийся потомком, может замещать элемент Use Case, являющийся родителем, в любом месте диаграммы.

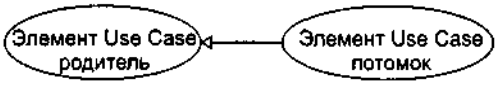


Рис. 12.30. Отношение обобщения между элементами Use Case

Отношение включения (рис. 12.31) между элементами Use Case означает, что базовый элемент Use Case *явно* включает поведение другого элемента Use Case в точке, которая определена в базе. Включаемый элемент Use Case никогда не используется самостоятельно — его конкретизация может быть только частью другого, большего элемента Use Case. Отношение включения является примером отношения делегации. При этом в отдельное место (включаемый элемент Use Case) помещается определенный набор обязанностей системы. Далее остальные части системы могут агрегировать в себя эти обязанности (при необходимости).

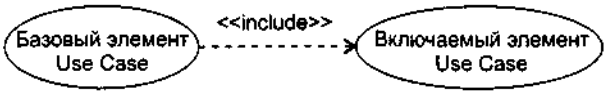


Рис. 12.31. Отношение включения между элементами Use Case

Отношение расширения (рис. 12.32) между элементами Use Case означает, что базовый элемент Use Case *неявно* включает поведение другого элемента Use Case в точке, которая определяется косвенно расширяющим элементом Use Case. Базовый элемент Use Case может быть автономен, но при определенных условиях его поведение может расширяться поведением из другого элемента Use Case. Базовый элемент Use Case может расширяться только в определенных точках — точках расширения. Отношение расширения применяется для моделирования выбираемого поведения системы. Таким способом можно отделить обязательное поведение от необязательного поведения. Например, можно использовать отношение расширения для отдельного подпотока, который выполняется только при определенных условиях, находящихся вне поля зрения базового элемента Use Case. Наконец, можно моделировать отдельные потоки, вставка которых в определенную точку управляется актером.

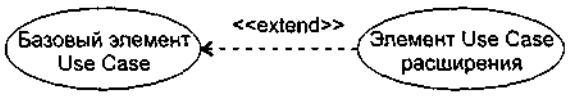


Рис. 12.32. Отношение расширения между элементами Use Case

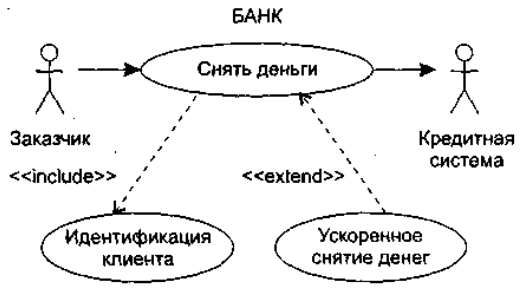


Рис. 12.33. Простейшая диаграмма Use Case для банка

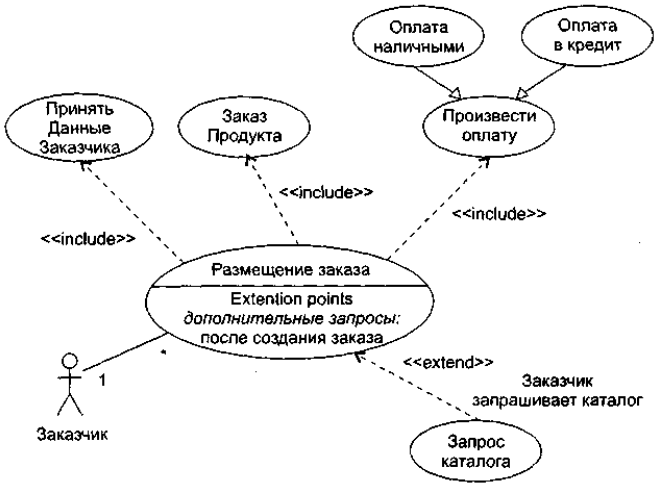


Рис. 12.34. Диаграмма Use Case для обслуживания заказчика

Пример простейшей диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис.

12.33.

Как показано на рис. 12.34, внутри элемента Use Case может быть дополнительная секция с заголовком Extention points. В этой области перечисляются точки расширения. В указанную здесь точку *дополнительные запросы* вставляется последовательность действий от расширяющего элемента Use Case Запрос каталога. Для справки отмечено, что точка расширения размещена после действий, обеспечивающих создание заказа. На этом же рисунке отображены отношения наследования между элементами Use Case. Видно, что элементы Use Case Оплата наличными и Оплата в кредит наследуют поведение элемента Use Case Произвести оплату и являются его специализациями.

Работа с элементами Use Case

Элемент Use Case описывает, *что* должна делать система, но не определяет, *как* она должна это делать. При моделировании это позволяет отделять внешнее представление системы от ее внутреннего представления.

Поведение элемента Use Case описывается потоком событий. Начальное описание выполняется в текстовой форме, прозрачной для пользователя системы. В потоке событий выделяют:

- основной поток и альтернативные потоки поведения;
- как и когда стартует и заканчивается элемент Use Case;
- когда элемент Use Case взаимодействует с актерами;
- какими данными обмениваются актер и система.

Для уточнения и формализации потоков событий используют диаграммы последовательности. Обычно одна диаграмма последовательности определяет главный поток в элементе Use Case, а другие диаграммы — потоки исключений.

В общем случае один элемент Use Case описывает набор последовательностей, в котором каждая последовательность представляет возможный поток событий. Каждая последовательность называется сценарием. Сценарий — конкретная последовательность действий, которая иллюстрирует поведение. Сценарии являются для элемента Use Case почти тем же, чем являются экземпляры для класса. Говорят, что сценарий — это экземпляр элемента Use Case.

Спецификация элементов Use Case

Спецификация элемента Use Case — основной источник информации для выполнения анализа и проектирования системы. Очень важно, чтобы содержание спецификации было представлено в полной и конструктивной форме. В общем случае спецификация включает главный поток, подпотоки и альтернативные потоки поведения. В качестве шаблона спецификации представим описание элемента Use Case *«Покупать авиабилет»* для модели информационной системы авиакассы.

Предусловие: перед началом этого элемента Use Case должен быть выполнен элемент Use Case «Заполнить базу данных авиарейсов».

Главный поток

Этот элемент Use Case начинается, когда покупатель регистрируется в системе и вводит свой пароль. Система проверяет, правилен ли пароль (E-1), и предлагает покупателю выбрать одно из действий: СОЗДАТЬ, УДАЛИТЬ, ПРОВЕРИТЬ, ВЫПОЛНИТЬ, ВЫХОД.

1. Если выбрано действие СОЗДАТЬ, выполняется подпоток S-1: создать заказ авиабилета.
2. Если выбрано действие УДАЛИТЬ, выполняется подпоток S-2: удалить заказ авиабилета.
3. Если выбрано действие ПРОВЕРИТЬ, выполняется подпоток S-3: проверить заказ авиабилета.
4. Если выбрано действие ВЫПОЛНИТЬ, выполняется подпоток S-4: реализовать заказ авиабилета.
5. Если выбрано действие ВЫХОД, элемент Use Case заканчивается.

Подпотоки

S-1: создать заказ авиабилета. Система отображает диалоговое окно, содержащее поля для пункта назначения и даты полета. Покупатель вводит пункт назначения и дату полета (E-2). Система отображает параметры авиарейсов (E-3). Покупатель выбирает авиарейс. Система связывает покупателя с выбранным авиарейсом (E-4). Возврат к началу элемента Use Case.

S-2: удалить заказ авиабилета. Система отображает параметры заказа. Покупатель подтверждает решение о ликвидации заказа (E-5). Система удаляет связь с покупателем (E-6). Возврат к началу элемента Use Case.

S-3: проверить заказ авиабилета. Система выводит (E-7) и отображает параметры заказа авиабилета: номер рейса, пункт назначения, дата, время, место, цену. Когда покупатель указывает, что он закончил проверку, выполняется возврат к началу элемента Use Case.

S-4: реализовать заказ авиабилета. Система запрашивает параметры кредитной карты покупателя. Покупатель вводит параметры своей кредитной карты (E-8). Возврат к началу элемента Use Case.

Альтернативные потоки

E-1: введен неправильный ID-номер покупателя. Покупатель может повторить ввод ID-номера или прекратить элемент Use Case.

E-2: введены неправильные пункт назначения/дата полета. Покупатель может повторить ввод пункта назначения/даты полета или прекратить элемент Use Case.

E-3: нет подходящих авиарейсов. Покупатель информируется, что в данное время такой полет невозможен. Возврат к началу элемента Use Case.

E-4: не может быть создана связь между покупателем и авиарейсом. Информация сохраняется, система создаст эту связь позже. Элемент Use Case продолжается.

E-5: введен неправильный номер заказа. Покупатель может повторить ввод правильного номера заказа или прекратить элемент Use Case.

E-6: не может быть удалена связь между покупателем и авиарейсом. Информация сохраняется, система будет удалять эту связь позже. Элемент Use Case продолжается.

E-7: система не может вывести информацию заказа. Возврат к началу элемента Use Case.

E-8: некорректные параметры кредитной карты. Покупатель может повторить ввод параметров карты или прекратить элемент Use Case.

Таким образом, в данной спецификации зафиксировано, что внутри элемента Use Case находится один основной поток и двенадцать вспомогательных потоков действий. В дальнейшем разработчик может принять решение о выделении из этого элемента Use Case самостоятельных элементов Use Case. Очевидно, что если самостоятельный элемент Use Case содержит подпоток, то его следует подключать к базовому элементу Use Case отношением include. В свою очередь, самостоятельный элемент Use Case с альтернативным потоком подключается к базовому элементу Use Case отношением extend.

Пример диаграммы Use Case

Наибольшие трудности при построении диаграмм Use Case вызывает применение отношений включения и расширения. Очень важно разобраться в отличительных особенностях этих отношений, специфике взаимодействия элементов Use Case, соединяемых с их помощью.

Пример диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рис. 12.35.

В этой диаграмме один базовый элемент Use Case Сеанс банкомата, который взаимодействует с актером Клиент. К базовому элементу Use Case подключены два расширяющих элемента Use Case (Состояние, Снять) и два включаемых элемента Use Case (Идентификация клиента, Проверка счета). В свою очередь, к элементу Use Case Идентификация клиента подключен включаемый элемент Use Case Проверить достоверность, а к элементу Use Case Снять — расширяющий элемент Use Case Захват карты (он же расширяет элемент Use Case Проверить достоверность).

Видим, что элемент Use Case Сеанс банкомата имеет две точки расширения (диалог возможен, выдача квитанции), а элементы Use Case Снять и Проверить достоверность — по одной точке расширения (проверка снятия и проверка соответственно). В точки расширения возможна вставка поведения из расширяющего элемента Use Case. Вставка происходит, если выполняется условие расширения:

- для расширяющего элемента Use Case Состояние — это условие запрос состояния;
- для расширяющего элемента Use Case Снять — это условие запрос снятия;
- для расширяющего элемента Use Case Захват карты — это условие список подозрений.

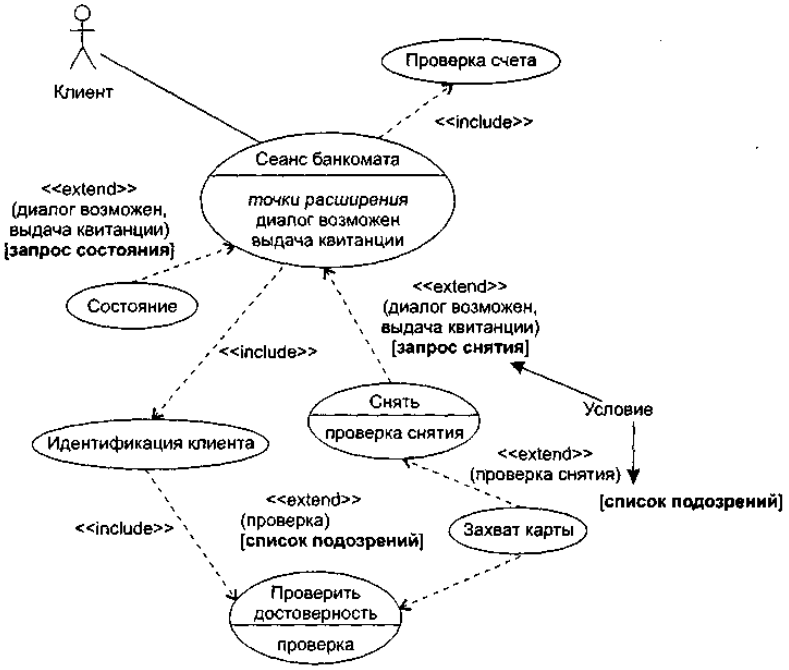


Рис. 12.35. Использование включения и расширения

Для расширяемого (базового) элемента Use Case эти условия являются внешними, то есть формируемыми вне его. Иными словами, элементу Use Case Сеанс банкомата ничего не известно об условиях запрос состояния и запрос снятия, а элементам Use Case Снять и Проверить достоверность — об условии список подозрений. Условия расширения являются следствиями событий, происходящих во внешней среде.

Стрелки расширения в диаграмме подписаны. Помимо стереотипа, здесь указаны:

- в круглых скобках — имена точек расширения;
- в квадратных скобках — условие расширения.

Описание расширяющего элемента Use Case разделено на сегменты, каждый сегмент обслуживает одну точку расширения базового элемента Use Case.

Количество сегментов расширяющего элемента Use Case равно количеству точек расширения базового элемента Use Case. Первый сегмент расширяющего элемента Use Case начинается с условия расширения, условие записывается только один раз, его действие распространяется и на все остальные сегменты.

Поведение базового элемента Use Case задается внутренним потоком событий, вплоть до точки расширения. В точке расширения возможно выполнение расширяющего элемента Use Case, после чего возобновляется работа внутреннего потока.

Спецификации элементов Use Case рассматриваемой диаграммы имеют следующий вид:

Элемент Use Case Сеанс банкомата

include (Идентификация клиента)	//включение
include (Проверка счета)	//включение
(диалог возможен)	//первая точка расширения
напечатать заголовок квитанции	
(выдача квитанции)	//вторая точка расширения
конец сеанса	

Расширяющий элемент Use Case Состояние

сегмент	//начало первого сегмента
принять запрос состояния	//условие расширения
отобразить информацию о состоянии счета	
сегмент	//вторая точка расширения
конец сеанса	

Расширяющий элемент Use Case Снять

сегмент	//начало первого сегмента
принять запрос снятия	//условие расширения
определить сумму	
(проверка снятия)	//точка расширения
сегмент	//начало второго сегмента
напечатать снимаемую сумму	
выдать наличные деньги	

Расширяющий элемент Use Case Захват карты

сегмент	//начало единственного сегмента
принять список подозрений	//условие расширения
проглотить карту	
конец сеанса	

Включаемый элемент Use Case Идентификация клиента

получить имя клиента	
include (Проверить достоверность)	//включение
получить номер счета клиента	

Включаемый элемент Use Case Проверка счета

установить соединение с базой данных счетов
получить состояние и ограничения счета

Включаемый элемент Use Case Проверить достоверность

установить соединение с базой данных клиентов
получить параметры клиента
(проверка) //точка расширения

Опишем возможное поведение модели, задаваемое этой диаграммой.

Актер Клиент инициирует действия базового элемента Use Case Сеанс банкомата. На первом шаге запускается включаемый элемент Use Case Идентификация клиента. Этот элемент Use Case получает имя клиента и запускает элемент Use Case Проверить достоверность, в результате чего устанавливается соединение с базой данных клиентов и получаются параметры клиента.

Если к этому моменту исполняется условие расширения список подозрений, то «срабатывает» расширяющий элемент Use Case Захват карты, карта арестовывается и работа системы прекращается.

В противном случае происходит возврат к элементу Use Case Идентификация клиента, который получает номер счета клиента и возвращает управление базовому элементу Use Case.

Базовый элемент Use Case переходит ко второму шагу работы — вызывает включаемый элемент Use Case Проверка счета, который устанавливает соединение с базой данных счетов и получает состояние и ограничения счета.

Управление опять возвращается к базовому элементу Use Case. Базовый элемент Use Case переходит к первой точке расширения диалог возможен. В этой точке возможно подключение одного из двух расширяющих элементов Use Case.

Положим, что к этому моменту выполняется условие расширения запрос состояния, поэтому запускается первый сегмент элемента Use Case Состояние. В результате отображается информация о состоянии счета и управление передается базовому элементу Use Case. В базовом элементе Use Case печатается заголовок квитанции и обеспечивается переход ко второй точке расширения выдача квитанции.

Поскольку в активном состоянии продолжает находиться расширяющий элемент Use Case Состояние, запускается его второй сегмент — в квитанции печатается информация о состоянии счета.

В последний раз управление возвращается к базовому элементу Use Case — завершается сеанс работы банкомата.

Построение модели требований

Напомним, что основное назначение диаграмм Use Case — определение требований заказчика к будущему программному приложению. Обсудим разработку ПО для машины утилизации, которая принимает использованные бутылки, банки, ящики. Для определения элементов Use Case, которые должны выполняться в системе, вначале определяют актеров.

Выбор актеров

Поиск актеров — большая работа. Сначала выделяют первичных актеров, использующих систему по прямому назначению. Каждый из первичных актеров участвует в выполнении одной или нескольких главных задач системы. В нашем примере первичным актером является Потребитель. Потребитель кладет в машину бутылки, получает квитанцию от машины.

Кроме первичных, существуют и вторичные актеры. Они наблюдают и обслуживают систему. Вторичные актеры существуют только для того, чтобы первичные актеры могли использовать систему. В нашем примере вторичным актером является Оператор. Оператор обслуживает машину и получает дневные отчеты о ее работе. Мы не будем нуждаться в операторе, если не будет потребителей.

Таким образом, внешняя среда машины утилизации имеет вид, представленный на рис. 12.36.

Рис. 12.36. Внешняя среда машины утилизации

Деление актеров на первичных и вторичных облегчает выбор системной архитектуры в терминах основного функционального назначения. Системную структуру определяют в основном первичные актеры. Именно от них в систему приходят главные изменения. Поэтому полное выделение первичных актеров гарантирует, что архитектура системы будет настроена на большинство важных пользователей.

Определение элементов Use Case

После выбора внешней среды можно выявить внутренние функциональные возможности системы. Для этого определяются элементы Use Case.

Каждый элемент Use Case задает некоторый путь использования системы, выполнение некоторой части функциональных возможностей. Полная совокупность элементов Use Case определяет все существующие пути использования системы.

Элемент Use Case — это последовательность взаимодействий в диалоге, выполняемом актером и системой. Запускается элемент Use Case актером, поэтому удобно выявлять элементы Use Case с помощью актеров.

Рассматривая каждого актера, мы решаем, какие элементы Use Case он может выполнять. Для этого изучается описание системы (с точки зрения актера) или проводится обсуждение с теми, кто будет действовать как актер.

Перейдем к примеру. Потребитель — первичный актер, поэтому начнем с этой роли. Этот актер должен выполнять возврат утилизируемых элементов. Так формируется элемент Use Case Возврат элемента. Приведем его текстовое описание:

> Начинается, когда потребитель начинает возвращать банки, бутылки, ящики. Для каждого элемента, помещенного в машину утилизации, система увеличивает количество элементов, принятых от Потребителя, и общее количество элементов этого типа за день.
>
> После сдачи всех элементов Потребитель нажимает кнопку квитанции, чтобы получить квитанцию, на которой напечатаны названия возвращенных элементов и общая сумма возврата.

Следующий актер — Оператор. Он получает дневной отчет об элементах, сданных за день. Это образует элемент Use Case Создание дневного отчета. Его описание:

Начинается оператором, когда он хочет получить информацию об элементах, сданных за день.

Система печатает количество элементов каждого типа и общее количество элементов, полученных за день.

Доя подготовки к созданию нового дневного отчета сбрасывается в ноль параметр Общее количество.

Кроме того, актер Оператор может изменять параметры сдаваемых элементов. Назовем соответствующий элемент Use Case Изменение элемента. Его описание:

Могут изменяться цена и размер каждого возвращаемого элемента. Могут добавляться новые типы элементов.

После выявления всех элементов диаграмма Use Case для системы принимает вид, показанный на рис. 12.37.

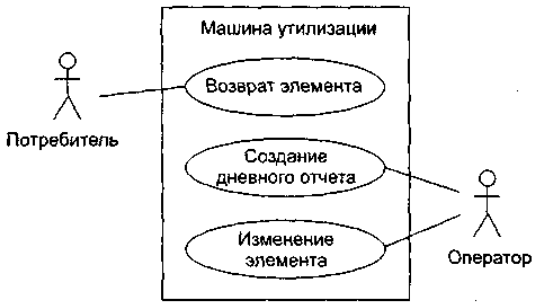


Рис. 12.37. Диаграмма Use Case для машины утилизации

Чаще всего полные описания элементов Use Case формируются за несколько итераций. На каждом шаге в описание вводятся дополнительные детали. Например, окончательное описание Возврата элемента может иметь следующий вид:

Когда потребитель возвращает сдаваемый элемент, элемент измеряется системой. Измерения позволяют определить тип элемента. Если тип допустим, то увеличивается количество элементов этого типа, принятых от Потребителя, и общее количество элементов этого типа за день.

Если тип недопустим, то на панели машины высвечивается «недействительно».

Когда Потребитель нажимает кнопку квитанции, принтер печатает дату. Производятся вычисления. По каждому типу принятых элементов печатается информация: название, принятое количество, цена, итого для типа. В конце печатается сумма, которую должен получить потребитель.

Не всегда очевидно, как распределить функциональные возможности по отдельным элементам Use Case и что является вариантом одного и того же элемента Use Case. Основной критерий выбора — сложность элемента Use Case. При анализе вариантов поведения рассматривают их различия. Если различия малы, варианты встраивают в один элемент Use Case. Если различия велики, то варианты описываются как отдельные элементы Use Case.

Обычно элемент Use Case задает одну основную и несколько альтернативных последовательностей событий.

Каждый элемент Use Case выделяет частный аспект функциональных возможностей системы. Поэтому элементы Use Case обеспечивают инкрементную схему анализа функций системы. Можно независимо разрабатывать элементы Use Case для разных функциональных областей, а позднее соединить их вместе (для формирования полной модели требований).

Вывод: на основе элементов Use Case в каждый момент времени можно концентрировать внимание на одной частной проблеме, что позволяет вести параллельную разработку.

Расширение функциональных возможностей

Для добавления в элемент Use Case новых действий удобно применять отношение расширения. С его помощью базовый элемент Use Case может быть расширен новым элементом Use Case.

В нашем примере поведение системы не определено для случая, когда сдаваемый элемент застрял. Введем элемент Use Case Элемент Застрял, который будет расширять базовый элемент Use Case Возврат Элемента (рис. 12.38).

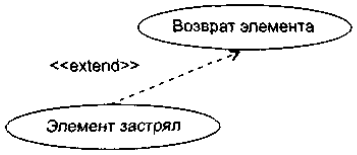


Рис 12.38. Расширение элемента Use Case возврат элемента

Описание элемента Use Case Элемент застрял может иметь следующий вид:

Если элемент застрял, для вызова Оператора вырабатывается сигнал тревоги. После удаления застрявшего элемента Оператор сбрасывает сигнал тревоги. В результате Потребитель может продолжить сдачу элементов. Величина ИТОГО сохраняет правильное значение. Цена застрявшего элемента не засчитывается.

Таким образом, описание базового элемента остается прежним, простым. Еще один пример приведен на рис. 12.39.

Здесь мы видим только один базовый элемент Use Case Сеанс работы. Все остальные элементы Use Case могут добавляться как расширения. Базовый элемент Use Case при этом остается почти без изменений.

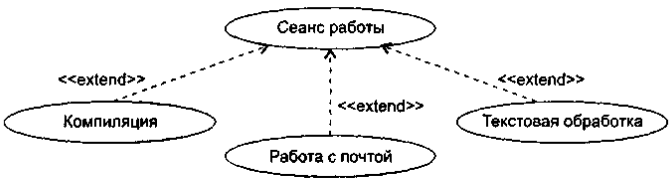


Рис. 12.39. Применение отношения расширения

Отношение расширения определяет прерывание базового элемента Use Case, которое происходит для вставки другого элемента Use Case. Базовый элемент Use Case не знает, будет выполняться прерывание или нет. Вычисление условий прерывания находится вне компетенции базового элемента Use Case.

В расширяющем элементе Use Case указывается ссылка на то место базового элемента Use Case, куда он будет вставляться (при прерывании). После выполнения расширяющего элемента Use Case продолжается выполнение базового элемента Use Case.

Обычно расширения используют:

- для моделирования вариантных частей элементов Use Case;
- для моделирования сложных и редко выполняемых альтернативных последовательностей;
- для моделирования подчиненных последовательностей, которые выполняются только в определенных случаях;
- для моделирования систем с выбором на основе меню.

Главное, что следует помнить: решение о выборе, подключении варианта на основе расширения принимается вне базового элемента Use Case. Если же вы вводите в базовый элемент Use Case условную конструкцию, конструкцию выбора, то придется применять отношение включения. Это случай, когда «штурвал управления» находится в руках базового элемента Use Case.

Уточнение модели требований

Уточнение модели сводится к выявлению одинаковых частей в элементах Use Case и извлечению этих частей. Любые изменения в такой части, выделенной в отдельный элемент Use Case, будут автоматически влиять на все элементы Use Case, которые используют ее совместно.

Извлеченные элементы Use Case называют абстрактными. Они не могут быть конкретизированы сами по себе, применяются для описания одинаковых частей в других, конкретных элементах Use Case. Таким образом, описания абстрактных элементов Use Case используются в описаниях конкретных элементов Use Case. Говорят, что конкретный элемент Use Case находится в отношении «включает» с абстрактным элементом Use Case.

Вернемся к нашему примеру. В этом примере два конкретных элемента Use Case Возврат элемента и Создание дневного отчета имеют общую часть — действия, обеспечивающие печать квитанции. Поэтому, как показано на рис. 12.40, можно выделить абстрактный элемент Use Case Печать. Этот элемент Use Case будет специализироваться на выполнении распечаток.

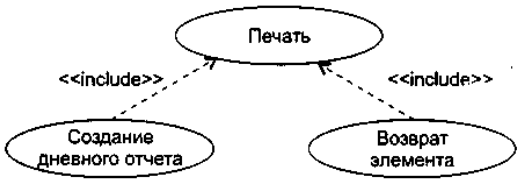


Рис. 12.40. Применение отношения включения

В свою очередь, абстрактные элементы Use Case могут использоваться другими абстрактными элементами Use Case. Так образуется иерархия. При построении иерархии абстрактных элементов Use Case руководствуются правилом: выделение элементов Use Case прекращается при достижении уровня отдельных операций над объектами.

Выделение абстрактных элементов Use Case можно упростить с помощью абстрактных актеров.

Абстрактный актер — это общий фрагмент роли в нескольких конкретных актерах. Абстрактный актер выражает подобия в элементах Use Case. Конкретные актеры находятся в отношении наследования с абстрактным актером. Так, в машине утилизации конкретные актеры имеют одно общее поведение: они могут получать квитанцию. Поэтому можно определить одного абстрактного актера — Получателя квитанции. Как показано на рис. 12.41, наследниками этого актера являются Потребитель и Оператор.

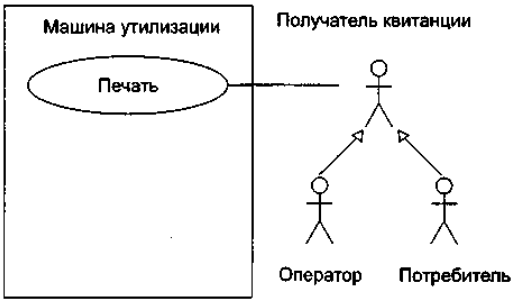


Рис. 12.41. Выделение абстрактного актера

Выводы:

1. Абстрактные элементы Use Case находят извлечением общих последовательностей из различных элементов Use Case.
2. Отношение «включает» применяется, если несколько элементов Use Case имеют общее поведение. Цель: устранить повторения, ликвидировать избыточность.
1. Кроме того, это отношение часто используют для ограничения сложности большого элемента Use Case.
2. Отношение «расширяет» применяется, когда описывается вариация, дополняющая нормальное поведение.

Кооперации и паттерны

Кооперации (сотрудничества) являются средством представления комплексных решений в разработке ПО на высшем, архитектурном уровне. С одной стороны, ^ кооперации обеспечивают компактность цельной спецификации программного продукта, с другой стороны — несут в себе реализации потоков управления и данных, а также структур данных.

В терминологии фирмы Rational (вдохновителя и организатора побед языка UML) кооперации называют реализациями элементов Use Case, да и обозначения их весьма схожи (рис. 12.42).

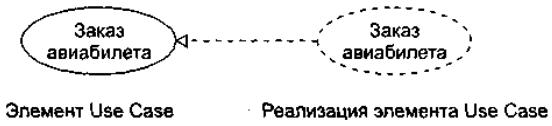


Рис. 12.42. Элемент Use Case и его реализация

Обратите внимание на то, что и связаны эти элементы отношением реализации: кооперация реализует конкретный элемент Use Case.

Кооперации содержат две составляющие — статическую (структурную) и динамическую (поведенческую).

Статическая составляющая кооперации задает структуру совместно работающих классов и других элементов (интерфейсов, компонентов, узлов). Чаще всего для этого используют одну или несколько диаграмм классов. Динамическая составляющая кооперации определяет поведение совместно работающих элементов. Обычно для определения применяют одну или несколько диаграмм последовательности.

Таким образом, если заглянуть под «обложку» кооперации, мы увидим набор разнообразных диаграмм. Например, требования к информационной системе авиакассы задаются множеством элементов Use Case, каждый из которых реализуется отдельной кооперацией. Все эти кооперации применяют одни и те же классы, но все же имеют разную функциональную организацию. В частности, поведение кооперации для заказа авиабилета может описываться диаграммой последовательности, показанной на рис. 12.43.

Соответственно, структура кооперации для заказа авиабилета может иметь вид, представленный на рис. 12.44.

Важно понимать, что кооперации отражают понятийный аспект архитектуры системы. Один и тот же элемент может участвовать в различных кооперациях. Ведь речь здесь идет не о владении элементом, а только о его применении.

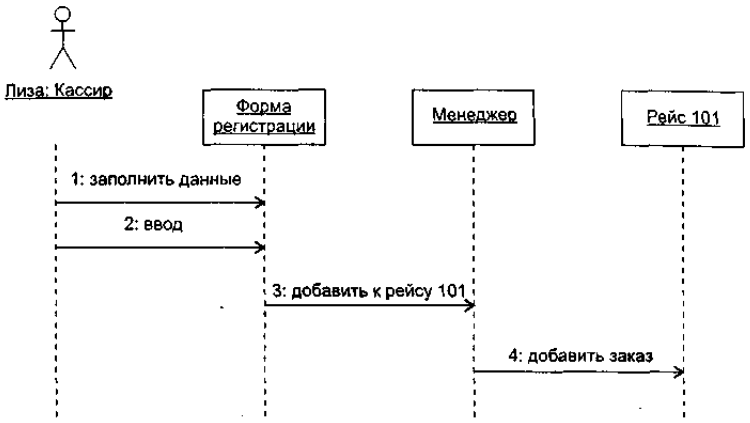


Рис. 12.43. Динамическая составляющая кооперации Заказ авиабилета

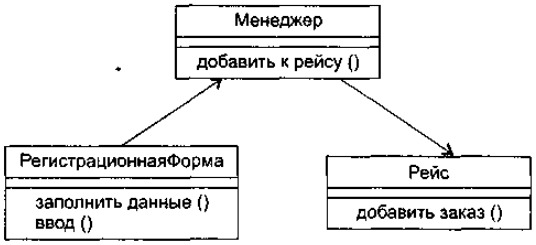


Рис. 12.44. Статическая составляющая кооперации Заказ авиабилета

Рис. 12.45. Обозначение паттерна

Параметризованные, то есть настраиваемые кооперации называют паттернами (образцами). Паттерн является решением типичной проблемы в определенном контексте. Обозначение паттерна имеет вид, представленный на рис. 12.45.

На место параметров настройки паттерна подставляются различные фактические параметры, в результате создаются разные кооперации.

Паттерны рассматриваются как крупные строительные блоки. Их использование приводит к существенному сокращению затрат на анализ и проектирование ПО. повышению качества и правильности разработки на логическом уровне, ведь паттерны создаются опытными профессионалами и отражают проверенные и оптимизированные решения [26], [31], [68].

Итак, паттерны — это наборы готовых решений, рецепты, предлагающие к повторному использованию самое ценное для разработчика — сплав мирового опыта по созданию ПО.

Наиболее распространенные паттерны формализуют и сводят в единые каталоги. Самым известным каталогом проектных паттернов, обеспечивающих этап проектирования ПО, считают каталог «Команды четырех» (Э. Гамма и др.). Он включает в себя 23 паттерна, разделенные на три категории [31]. Как показано в табл. 12.1, по мнению «Команды четырех», описание паттерна должно состоять из четырех основных частей.

Таблица 12.1. Описание паттерна

Раздел	Описание
Имя	Выразительное имя паттерна дает возможность указать проблему проектирования, ее решение и последствия ее решения. Использование имен паттернов повышает уровень абстракции проектирования
Проблема	Формулируется проблема проектирования (и ее контекст), на которую ориентировано применение паттерна. Задаются условия применения
Решение	Описываются элементы решения, их отношения, обязанности, сотрудничество. Решение представляется в обобщенной форме, которая должна конкретизироваться при применении. Фактически приводится шаблон решения — его можно использовать в самых разных ситуациях
Результаты	Перечисляются следствия применения паттерна и вытекающие из них компромиссы. Такая информация позволяет оценить эффективность применения паттерна в данной ситуации

Обсудим применение нескольких паттернов из каталога «Команды четырех».

Паттерн Наблюдатель

Паттерн Наблюдатель (Observer) задает между объектами такую зависимость «один-ко-многим», при которой изменение состояния одного объекта приводит к оповещению и автоматическому обновлению всех зависящих от него объектов.

Проблема. При разбиении системы на набор совместно работающих объектов появляется необходимость поддерживать их согласованное состояние. При этом желательно минимизировать сцепление, так как высокое сцепление уменьшает возможности повторного использования. Например, во многих случаях требуется отображение данных состояния в различных графических формах и форматах. При этом объекту, формирующему состояние, не нужно знать о формах его отображения — отсутствие такого интереса благотворно влияет на необходимое сцепление. Паттерн Наблюдатель можно применять в следующих случаях:

- когда необходимо организовать непрямое взаимодействие объектов уровня логики приложения с интерфейсом пользователя. Таким образом достигается низкое сцепление между уровнями;
- когда при изменении состояния одного объекта должны изменить свое состояние все зависимые объекты, причем количество зависимых объектов заранее неизвестно;
- когда один объект должен рассылать сообщения другим объектам, не делая о них никаких предположений. За счет этого образуется низкое сцепление между объектами.

Решение. Принцип решения иллюстрирует рис. 12.46. Ключевыми элементами решения являются *субъект* и *наблюдатель.* У *субъекта* может быть любое количество зависимых от него *наблюдателей.* Когда происходят изменения в состоянии *субъекта, наблюдатели* автоматически об этом уведомляются. Получив уведомление, *наблюдатель* опрашивает *субъекта,* синхронизуя с ним свое отображение состояния.

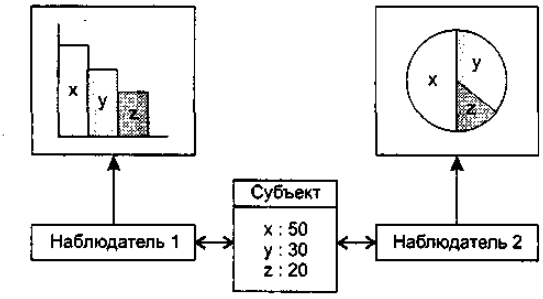


Рис. 12.46. Различные графические отображения состояния субъекта

Такое взаимодействие между элементами соответствует схеме *издатель-подписчик.* Издатель рассылает сообщения об изменении своего состояния, не имея никакой информации о том, какие объекты являются подписчиками. На получение таких уведомлений может подписаться любое количество наблюдателей.

Структурная составляющая паттерна Наблюдатель представлена на рис. 12.47. В ней определены два абстрактных класса, *Субъект* и *Наблюдатель.* Кроме того, здесь показаны два конкретных класса, КонкрСубъект и КонкрНаблюдатель, которые наследуют свойства и операции абстрактных классов. Они подключаются к паттерну в процессе его настройки. Состояние формируется Конкретным субъектом, который унаследовал от *Субъекта* операции, позволяющие ему добавлять и удалять Конкретных наблюдателей, а также уведомлять их об изменении своего состояния. Конкретный наблюдатель автоматически отображает состояние и реализует абстрактную операцию *Обновить() Наблюдателя,* обеспечивающую обновление отображаемого состояния.

ПРИМЕЧАНИЕ
Курсивом в данном абзаце отображены имена абстрактных классов и операций (это требование языка UML).

Динамическая составляющая паттерна Наблюдатель показана на рис. 12.48. На рисунке представлено поведение паттерна при взаимодействии субъекта с двумя наблюдателями.

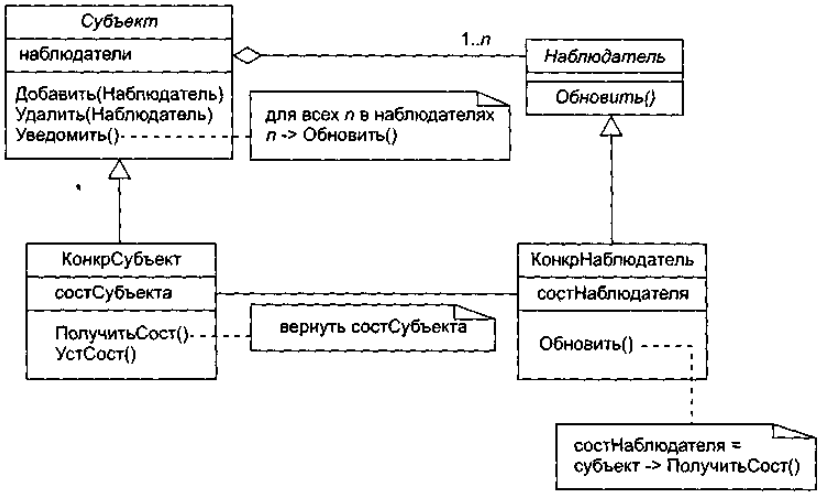


Рис. 12.47. Структурная составляющая паттерна Наблюдатель

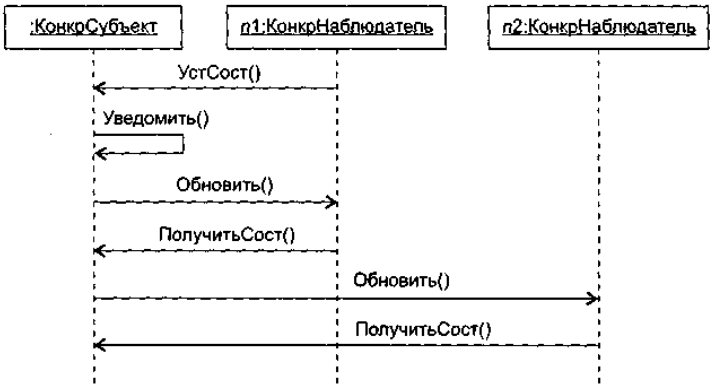


Рис. 12.48. Динамическая составляющая паттерна Наблюдатель

Результаты. Субъекту известно только об абстрактном наблюдателе, он ничего не знает о конкретных наблюдателях. В результате между этими объектами устанавливается минимальное сцепление (это достоинство). Изменения в субъекте могут привести к неоправданно большому количеству обновлений наблюдателей — ведь наблюдателю неизвестно, что именно изменилось в субъекте, затрагивают ли его произошедшие изменения (это недостаток).

Обозначение паттерна Наблюдатель приведено на рис. 12.49, где показано, что у него два параметра настройки — субъект и наблюдатель.

Рис. 12.49. Обозначение паттерна Наблюдатель

Эти параметры обозначают роли, которые будут играть конкретные классы, используемые при настройке паттерна на конкретное применение. Например, настройку паттерна на отображение текущего фильма кинофестиваля иллюстрирует рис. 12.50.

Рис. 12.50. Настройка паттерна Наблюдатель

Видим, что подключаемые конкретные классы (Кинопрограмма, Текущий фильм) соединяются с символом паттерна

пунктирными линиями. Каждая пунктирная линия подписана ролью (именем параметра), которую играет конкретный класс в формируемой кооперации. Таким образом, в данном случае класс Кинопрограмма становится подклассом абстрактного класса *Субъект* в паттерне, а класс Текущий фильм — подклассом абстрактного класса *Наблюдатель* в паттерне.

Паттерн Компоновщик

Паттерн Компоновщик (Composite) обеспечивает представление иерархий часть-целое, объединяя объекты в древовидные структуры.

Проблема. Очень часто возникает необходимость создавать маленькие компоненты (примитивы), объединять их в более крупные компоненты (контейнеры), более крупные компоненты соединять в большие компоненты, большие компоненты — в огромные и т. д. При этом клиентам приходится различать компоненты-примитивы и компоненты-контейнеры, работать с ними по-разному. Это усложняет приложение. Паттерн Компоновщик позволяет ликвидировать это различие, его можно применять в следующих случаях:

- необходимо построить иерархию объектов вида часть-целое;
- нужно унифицировать использование как составных, так и индивидуальных объектов.

Решение. Ключевым элементом решения является абстрактный класс *Компонент,* который является одновременно и примитивом, и контейнером. В нем объявлены:

- абстрактная операция примитива *Работать*( );
- абстрактные операции контейнера — управления примитивами-потомками *Добавить(Компонент)* и *Удалить(Компонент),* а также доступа к потомку *Получить-Потомка*().

Структурная составляющая паттерна Компоновщик представлена на рис. 12.51.

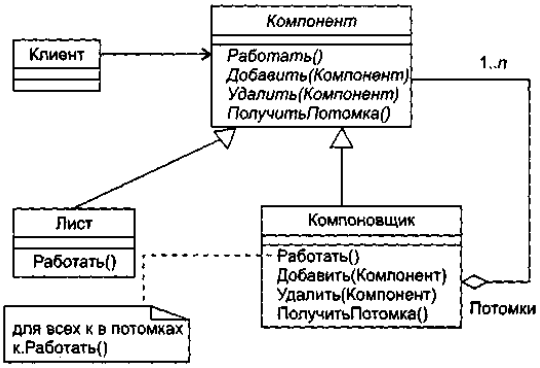


Рис. 12.51. Структурная составляющая паттерна Компоновщик

Из рисунка видно, что с помощью паттерна организуется рекурсивная композиция.

Класс *Компонент* служит простым элементом дерева, класс Компоновщик является рекурсивным элементом, а класс Лист — конечным элементом дерева. Класс *Компонент* служит родителем классов Лист и Компоновщик. Отметим, что класс Компоновщик является агрегатом составных частей — экземпляров класса *Компонент* (таким образом задается рекурсия).

Клиенты используют интерфейс класса *Компонент* для взаимодействия с объектами дерева. Если получателем запроса клиента является объект-лист, то он и обрабатывает запрос. Если же получателем является составной объект-компоновщик, то он перенаправляет запрос своим потомкам, возможно выполняя дополнительные действия до или после перенаправления.

Результаты. Паттерн определяет иерархии, состоящие из классов-примитивов и классов-контейнеров, облегчает добавление новых разновидностей компонентов. Он упрощает организацию клиентов (клиент не должен учитывать специфику адресуемого объекта). Недостаток применения паттерна — трудность в наложении ограничений на объекты, которые можно включать в композицию.

Обозначение паттерна Компоновщик приведено на рис. 12.52, где показано, что у него три параметра настройки — компонент, компоновщик и лист.

Настройку паттерна на графическое приложение иллюстрирует рис. 12.53.

Рис. 12.52. Обозначение паттерна Компоновщик

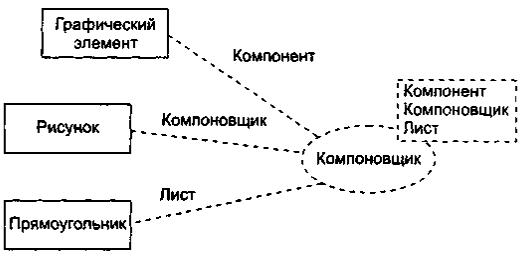


Рис. 12.53. Настройка паттерна Компоновщик

В этом случае основной операцией приложения становится операция Рисовать(). Подразумевается, что такая операция входит в состав каждого из подключаемых классов, то есть классов Рисунок, Прямоугольник и Графический элемент. Операции Рисовать() должны заместить операции Работать() в классах паттерна.

Паттерн Команда

Паттерн Команда (Command) выполняет преобразование запроса в объект, обеспечивая:

- параметризацию клиентов с различными запросами;
- постановку запросов в очередь и их регистрацию;
- поддержку отмены операций.

Проблема. Достаточно часто нужно посылать запрос, не зная, выполнение какой конкретной операции запрошено и кто является получателем запроса. В этих случаях следует отделить объект, инициирующий запрос, от объекта, способного выполнить запрос. В результате обеспечивается высокая гибкость разработки пользовательского интерфейса — можно связывать различные пункты меню с определенной функцией, динамически подменять команды и т. д. Паттерн Команда применяется в следующих случаях:

- объекты параметризируются действием. В процедурных языках параметризация осуществляется при помощи функции обратного вызова, которая регистрируется для последующего вызова. Паттерн Команда предлагает объектно-ориентированную замену функций обратного вызова;
- необходимо обеспечить отмену операций. Это возможно благодаря хранению истории выполнения операций;
- необходимо регистрировать изменения состояния для восстановления системы в случае аварийного отказа;
- необходимо создание сложных операций, которые строятся на основе примитивных операций.

Решение. Основным элементом решения является абстрактный класс *Команда,* обеспечивающий одну абстрактную операцию *Выполнять*(). Конкретные подклассы этого класса реализуют операцию Выполнять(). Они задают пару получатель-действие. Получатель запоминается в экземплярной переменной подкласса. Запрос получателю посылается в ходе исполнения конкретной операции Выполнять().

Структурная составляющая паттерна Команда показана на рис. 12.54. Классы этой структуры имеют следующие обязанности:

- *Команда* объявляет интерфейс для выполнения операции;
- КонкрКоманда определяет связь между экземпляром класса Получатель и действием, реализует Выполнять(), вызывая нужную операцию получателя;
- Клиент создает объект класса КонкрКоманда и устанавливает его получателя;
- Инициатор просит команду выполнить запрос;
- Получатель умеет выполнять запрашиваемые операции.

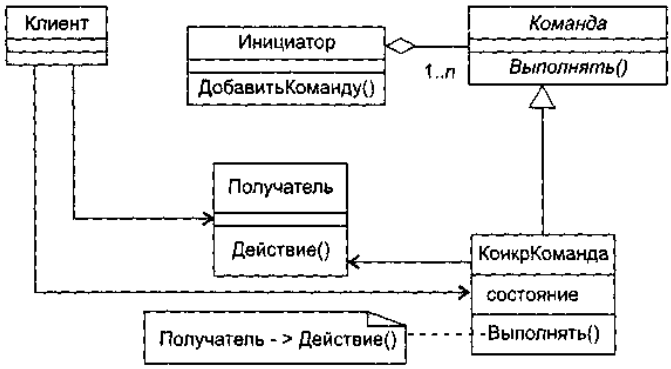


Рис. 12.54. Структурная составляющая паттерна Команда

В качестве конкретной команды могут выступать команда Открыть, команда Вставить. Инициатором может быть Пункт Меню, а получателем — Документ.

Объекты этого паттерна осуществляют следующие взаимодействия:

- клиент создает объект класса КонкрКоманда и задает его получателя;
- объект класса Инициатор сохраняет объект класса КонкрКоманда;
- инициатор вызывает операцию Выполнять() объекта класса КонкрКоманда;
- объект класса КонкрКоманда вызывает операцию своего получателя для исполнения запроса.

Результаты. Применение паттерна Команда приводит к следующему:

- объект, запрашивающий операцию, отделяется от объекта, умеющего выполнять запрос;
- объекты-команды являются полноценными объектами. Их можно использовать и расширять обычным способом;
- из простых команд легко компонуются составные команды;
- легко добавляются новые команды (изменять существующие классы при этом не требуется).

Обозначение паттерна Команда приведено на рис. 12.55, где показано, что у него четыре параметра настройки — клиент, команда, инициатор и получатель.

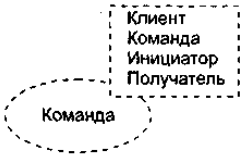


Рис. 12.55. Обозначение паттерна Команда

Настройку паттерна на приложение с графическим меню иллюстрирует рис. 12.56.

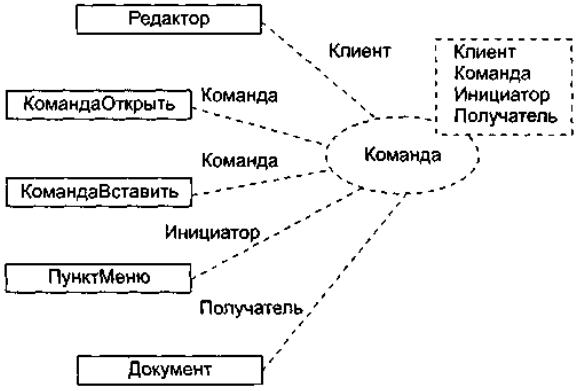


Рис. 12.56. Настройка паттерна Команда

Очевидно, что в получаемой кооперации конкретный класс Редактор играет роль клиента, классы КомандаОткрыть и КомандаВставить становятся классами Конкретных Команд (и подклассами абстрактного класса *Команда),* класс ПунктМеню замещает класс Инициатор паттерна, а конкретный класс Документ замещает класс Получатель паттерна.

Бизнес-модели

Достаточно часто перед тем, как решиться на заказ ПО, организация проводит бизнес-моделирование. Цели бизнес-моделирования:

- отобразить структуру и процессы деятельности организации;
- обеспечить ясное, комплексное и, главное, одинаковое понимание нужд организации как сотрудниками, так и будущими разработчиками ПО;
- сформировать реальные требования к программному обеспечению деятельности организации.

Для достижения этих целей разрабатываются две модели: Q бизнес-модель Use Case; а бизнес-объектная модель.

Бизнес-модель Use Case задает внешнее представление бизнес-процессов организации (с точки зрения внешней среды — клиентов и партнеров).

Как показано на рис. 12.57, бизнес-модель Use Case строится с помощью бизнес-актеров и бизнес-элементов Use Case — простого расширения средств, используемых в обычных диаграммах Use Case.

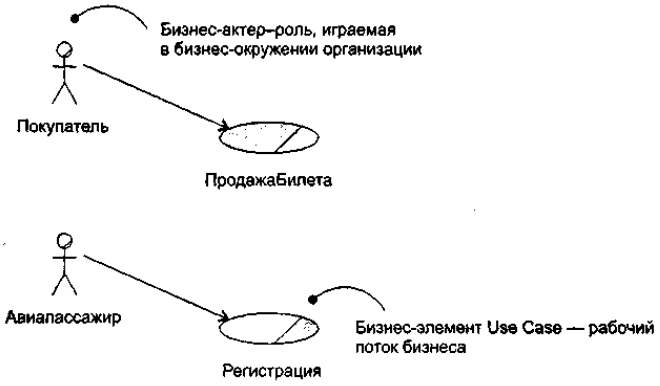


Рис. 12.57. Фрагмент бизнес-модели Use Case для аэропорта

Бизнес-актеры определяют внешние сущности и людей, с которыми взаимодействует бизнес. Бизнес-актер представляет собой человека, но информационная система, взаимодействующая с бизнесом, также может играть роль такого актера.

Бизнес-элементы Use Case изображают различные рабочие потоки бизнеса. Последовательности действий в бизнес-элементах Use Case обычно описываются диаграммами деятельности.

Бизнес-объектная модель отражает внутреннее представление бизнес-процессов организации (с точки зрения ее сотрудников).

Как показано на рис. 12.58, бизнес-объектная модель строится с помощью бизнес-работников и бизнес-сущностей — классов со специальными стереотипами. Эти классы имеют специальные графические обозначения.

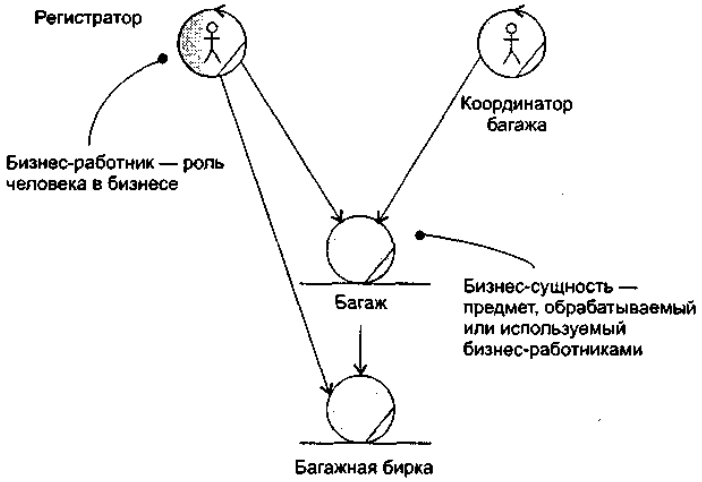


Рис. 12.58. Фрагмент бизнес-объектной модели аэропорта

Бизнес-работник — абстракция человека, действующего в бизнесе. Бизнес-сущности являются «предметами», обрабатываемыми или используемыми бизнес-работниками по мере выполнения бизнес-элемента Use Case. Например, бизнес-сущность представляет собой документ или существенную часть продукта. Фактически бизнес-объектная модель отображается с помощью диаграмм классов.

Основы компонентной объектной модели

Компонентная объектная модель (COM) — фундамент компонентно-ориентированных средств для всего семейства операционных систем Windows. Рассмотрение этой модели является иллюстрацией комплексного подхода к созданию и использованию компонентного программного обеспечения [5].

COM определяет стандартный механизм, с помощью которого одна часть программного обеспечения предоставляет свои услуги другой части. Общая архитектура предоставления услуг в библиотеках, приложениях, системном и сетевом программном обеспечении позволяет COM изменить подход к созданию программ.

COM устанавливает понятия и правила, необходимые для определения объектов и интерфейсов; кроме того, в ее состав входят программы, реализующие ключевые функции.

В COM любая часть ПО реализует свои услуги с помощью объектов COM. Каждый объект COM поддерживает несколько интерфейсов. Клиенты могут получить доступ к услугам объекта COM только через вызовы операций его интерфейсов — у них нет непосредственного доступа к данным объекта.

Представим объект COM с интерфейсом РаботаСФайлами. Пусть в этот интерфейс входят операции ОткрытьФайл, ЗаписатьФайл и ЗакрытьФайл. Если разработчик захочет ввести в объект COM поддержку преобразования форматов, то объекту потребуется еще один интерфейс ПреобразованиеФорматов, возможно, с единственной операцией ПреобразоватьФормат. Операции каждого из интерфейсов сообща предоставляют связанные друг с другом услуги: либо работу с файлами, либо преобразование их форматов.

Как показано на рис. 13.14, объект COM всегда реализуется внутри некоторого сервера. Сервер может быть либо динамически подключаемой библиотекой (DLL), подгружаемой во время работы приложения, либо отдельным самостоятельным процессом (EXE). Отметим, что здесь мы не будем применять графику языка UML, а воспользуемся принятыми в COM обозначениями.

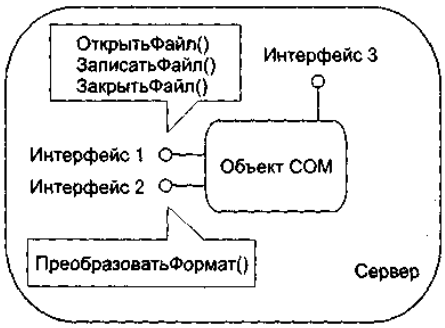


Рис. 13.14. Организация объекта COM

Для вызова операции интерфейса клиент объекта COM должен получить указатель на его интерфейс. Клиенту требуется отдельный указатель для каждого интерфейса, операции которого он намерен вызывать. Например, как показано на рис. 13.15, клиенту нашего объекта COM нужен один указатель интерфейса для вызова операций интерфейса РаботаСФайлами, а другой — для вызова операций интерфейса ПреобразованиеФорматов.

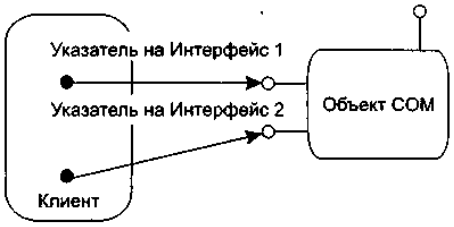


Рис. 13.15. Доступ клиента к интерфейсам объекта COM

Получив указатель на нужный интерфейс, клиент может использовать услуги объекта, вызывая операции этого интерфейса. С точки зрения программиста, вызов операции аналогичен вызову локальной процедуры или функции. Но на самом деле исполняемый при этом код может быть частью или библиотеки, или отдельного процесса, или операционной системы (он даже может располагаться на другом компьютере).

Благодаря COM клиентам нет нужды учитывать эти отличия — доступ ко всему осуществляется единообразно. Другими словами, в COM для доступа к услугам, предоставляемым любыми типами ПО, используется одна общая модель.

Организация интерфейса COM

Каждый интерфейс объекта COM — контракт между этим объектом и его клиентами. Они обязуются: объект — поддерживать операции интерфейса в точном соответствии с его определениями, а клиент — корректно вызывать операции. Для обеспечения контракта надо задать:

- идентификацию каждого интерфейса;
- описание операций интерфейса;
- реализацию интерфейса.

Идентификация интерфейса

У каждого интерфейса COM два имени. Простое, символьное имя предназначено для людей, оно не уникально (допускается, чтобы это имя было одинаковым у двух интерфейсов). Другое, сложное имя предназначено для использования программами. Программное имя уникально, это позволяет точно идентифицировать интерфейс.

Принято, чтобы символьные имена COM-интерфейсов начинались с буквы I (от Interface). Например, упомянутый нами интерфейс для работы с файлами должен называться IРаботаСФайлами, а интерфейс преобразования их форматов — IПреобразованиеФорматов.

Программное имя любого интерфейса образуется с помощью глобально уникального идентификатора (globally unique identifier — GUID). GUID интерфейса считается идентификатором интерфейса (interface identifier — IID). GUID — это 16-байтовая величина (128-битовое число), генерируемая автоматически.

Уникальность во времени достигается за счет включения в каждый GUID метки времени, указателя момента создания. Уникальность в пространстве обеспечивается цифровыми параметрами компьютера, который использовался для генерации GUID.

Описание интерфейса

Для определения интерфейсов применяют специальный язык — язык описания интерфейсов (Interface Definition Language — IDL). Например, IDL-описание интерфейса для работы с файлами 1РаботаСФайлами имеет вид

```
[ object.
        uuid(E7CDODOO-1827-11CF-9946-444553540000) ]
interface IРаботаСФайлами: IUnknown {
        import "unknown.idl"
        HRESULT ОткрытьФайп ([in] OLECHAR имя [31]);
        HRESULT ЗаписатьФайл ([in] OLECHAR имя [31]);
        HRESULT ЗакрытьФайл ([in] OLECHAR имя [31]);
}
```

Описание интерфейса начинается со слова object, отмечающего, что будут использоваться расширения, добавленные COM к оригинальному IDL. Далее записывается программное имя (IID интерфейса), оно начинается с ключевого слова uuid (Universal Unique Identifier — универсально уникальный идентификатор). UUID является синонимом термина GUID.

В третьей строке записывается имя интерфейса — РаботаСФайлами, за ним — двоеточие и имя другого интерфейса — IUnknown. Такая запись означает, что интерфейс РаботаСФайлами наследует все операции, определенные для интерфейса IUnknown, то есть клиент, у которого есть указатель на интерфейс IРаботаСФайлами, может вызывать и операции интерфейса IUnknown. IUnknown является главным интерфейсом для COM, все остальные интерфейсы — его наследники.

В четвертой строке указывается оператор import. Поскольку данный интерфейс наследует от IUnknown, может потребоваться IDL-описание для IUnknown. Аргумент оператора import указывает, в каком файле находится нужное описание.

Ниже в описании интерфейса приводятся имена и параметры трех операций — ОткрытьФайл, ЗаписатьФайл и ЗакрытьФайл. Все они возвращают величину типа HRESULT, указывающую корректность обработки вызова. Для каждого параметра в IDL приводится направление передачи (в данном примере in), тип и название.

Считается, что такого описания достаточно для заключения контракта между объектом COM и его клиентом.

По правилам COM запрещается любое изменение интерфейса (после его публикации). Для реализации изменений нужно

вводить новый интерфейс. Такой интерфейс может быть наследником старого интерфейса, но отличен от него и имеет другое уникальное имя.

Значение правила запрета на изменение интерфейса трудно переоценить. Оно — залог стабильности работы в среде, где множество клиентов взаимодействует с множеством COM-объектов и где независимая модификация COM-объектов — обычное дело. Именно это правило позволяет клиентам старых версий не пострадать при введении новых версий COM-объектов. Новая версия обязана поддерживать и старый COM-интерфейс.

Реализация интерфейса

COM задает стандартный двоичный формат, который должен реализовать каждый COM-объект и для каждого интерфейса. Стандарт гарантирует, что любой клиент может вызывать операции любого объекта, причем независимо от языков программирования, на которых написаны клиент и объект.

Структуру интерфейса IРаботаСФайлами, соответствующую двоичному формату, «поясняет рис. 13.16.

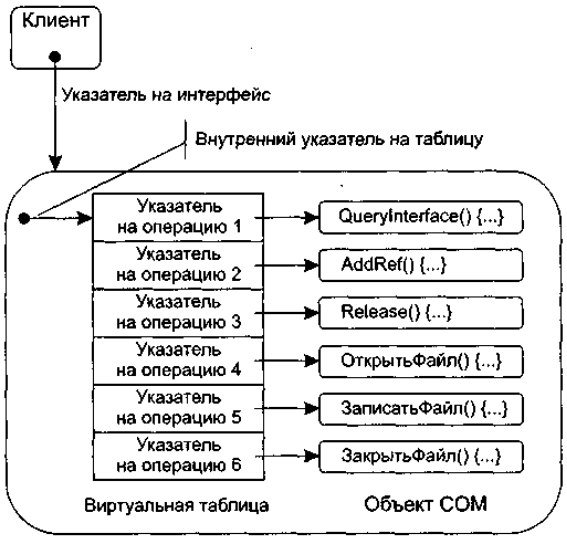


Рис. 13.16. Внутренняя структура интерфейса IРаботаСФайлами

Внешний указатель на интерфейс (указатель клиента) ссылается на внутренний указатель объекта COM. Внутренний указатель — это адрес виртуальной таблицы. Виртуальная таблица содержит указатели на все операции интерфейса.

Первые три элемента виртуальной таблицы являются указателями на операции, унаследованные от интерфейса IUnknown. Видно, что на собственные операции интерфейса IРаботаСФайлами указывают 4-, 5- и 6-й элементы виртуальной таблицы. Такая ситуация типична для любого COM-интерфейса.

Обработка клиентского вызова выполняется в следующем порядке:

- с помощью указателя на виртуальную таблицу извлекается указатель на требуемую операцию интерфейса;
- указатель на операцию обеспечивает доступ к ее реализации;
- исполнение кода операции обеспечивает требуемую услугу.

Unknown — базовый интерфейс COM

Интерфейс lUnknown обеспечивает минимальное «снаряжение» каждого объекта COM. Он содержит три операции и предоставляет любому объекту COM две функциональные возможности:

- операция QueryInterface() позволяет клиенту получить указатель на любой интерфейс объекта (из другого указателя интерфейса);
- операции AddRef() и Release() обеспечивают механизм управления временем жизни объекта.

Свой первый указатель на интерфейс объекта клиент получает при создании объекта COM. Порядок получения других указателей на интерфейсы (для вызова их операций) поясняет рис. 13.17, где расписаны три шага работы. В качестве параметра операции QueryInterface задается идентификатор требуемого интерфейса (IID). Если требуемый интерфейс отсутствует, операция возвращает значение NULL.

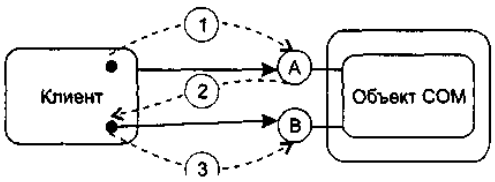


Рис. 13.17. Получение указателя на интерфейс с помощью QueryInterface: *1* — с помощью указателя на интерфейс А клиент запрашивает указатель на интерфейс В, вызывая QueryInterface (IID_B); *2* — объект возвращает указатель на интерфейс В; *3* — теперь клиент может вызывать операции из интерфейса В

Имеет смысл отметить и второе важное достоинство операции QueryInterface. В сочетании с требованием неизменности COM-интерфейсов она позволяет «убить двух зайцев»:

- развивать компоненты;
- обеспечивать стабильность клиентов, использующих компоненты.

Поясним это утверждение. По законам COM-этики новый COM-объект должен нести в себе и старый COM-интерфейс, а операция QueryInterface всегда обеспечит доступ к нему.

Ну а теперь обсудим правила жизни, а точнее смерти COM-объекта. В многоликой COM-среде бремя ответственности за решение вопроса о финализации должно лежать как на клиенте, так и на COM-объекте. Можно сказать, что фирма Microsoft (создатель этой модели) разработала самурайский кодекс поведения COM-объекта — он должен сам себя уничтожить. Возникает вопрос — когда? Когда он перестанет быть нужным всем своим клиентам, когда вытечет песок из часов его жизни. Роль песочных часов играет счетчик ссылок (СЧС) COM-объекта.

Правила финализации COM-объекта очень просты:

- при выдаче клиенту указателя на интерфейс выполняется СЧС+1;
- при вызове операции AddRef выполняется СЧС+1;
- при вызове операции Release выполняется СЧС-1;
- при СЧС=0 объект уничтожает себя.

Конечно, клиент должен помогать достойному харакири объекта-самурая:

- при получении от другого клиента указателя на интерфейс COM-объекта он должен вызвать в этом объекте операцию AddRef;
- в конце работы с объектом он обязан вызвать его операцию Release.

Серверы COM-объектов

Каждый COM-объект существует внутри конкретного сервера. Этот сервер содержит программный код реализации операций, а также данные активного COM-объекта. Один сервер может обеспечивать несколько объектов и даже несколько COM-классов. Как показано на рис. 13.18, используются три типа серверов:

- Сервер «в процессе» (in-process) — объекты находятся в динамически подключаемой библиотеке и, следовательно, выполняются в том же процессе, что и клиент;
- Локальный сервер (out-process) — объекты находятся в отдельном процессе, выполняющемся на том же компьютере, что и клиент;
- Удаленный сервер — объекты находятся в DLL или в отдельном процессе, которые расположены на удаленном от клиента компьютере.

С точки зрения логики, клиенту безразлично, в сервере какого типа находится COM-объект — создание объекта, получение указателя на его интерфейсы, вызов его операций и финализация выполняются всегда одинаково. Хотя временные затраты на организацию взаимодействия в каждом из трех случаев, конечно, отличаются друг от друга.

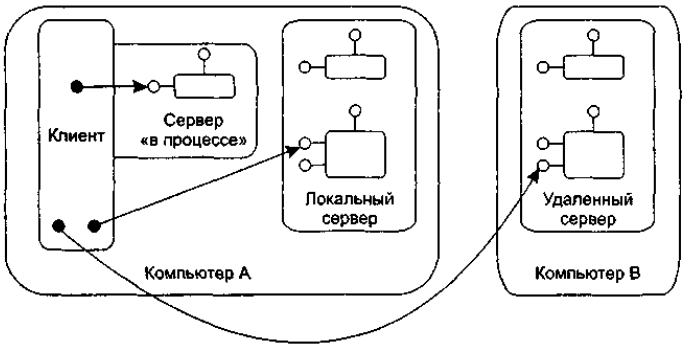


Рис. 13.18. Различные серверы COM-объектов

Преимущества COM

В качестве кратких выводов отметим основные преимущества COM.

1. COM обеспечивает удобный способ фиксации услуг, предоставляемых разными фрагментами ПО.
2. Общий подход к созданию всех типов программных услуг в COM упрощает проблемы разработки.
3. COM безразличен язык программирования, на котором пишутся COM-объекты и клиенты.
4. COM обеспечивает эффективное управление изменением программ — замену текущей версии компонента на новую версию с дополнительными возможностями.

Работа с COM-объектами

При работе с COM-объектами приходится их создавать, повторно использовать, размещать в других процессах, описывать в библиотеке операционной системы. Рассмотрим каждый из этих вопросов.

Создание COM-объектов

Создание COM-объекта базируется на использовании функций библиотеки COM. Библиотека COM:

- содержит функции, предлагающие базовые услуги объектам и их клиентам;
- предоставляет клиентам возможность запуска серверов COM-объектов.

Доступ к услугам библиотеки COM выполняется с помощью вызовов обычных функций. Чаще всего имена функций библиотеки COM начинаются с префикса «Co». Например, в библиотеке имеется функция CoCreateInstance.

Для создания COM-объекта клиент вызывает функцию библиотеки COM CoCreateInstance. В качестве параметров этой функции посылаются идентификатор класса объекта CLSID и IID интерфейса, поддерживаемого объектом. С помощью CLSID библиотека ищет сервер класса (это делает диспетчер управления сервисами SCM — Service Control Manager). Поиск производится в системном реестре (Registry), отображающем CLSID в адрес исполняемого кода сервера. В системном реестре должны быть зарегистрированы классы всех COM-объектов.

Закончив поиск, библиотека COM запускает сервер класса. В результате создается неинициализированный COM-объект, то есть объект, данные которого не определены. Описанный процесс иллюстрирует рис. 13.19.

Как правило, после получения указателя на созданный COM-объект клиент предлагает объекту самоинициализироваться, то есть загрузить себя конкретными значениями данных. Эту процедуру обеспечивают стандартные COM-интерфейсы IPersistFile, IPersistStorage и IPersistStream.

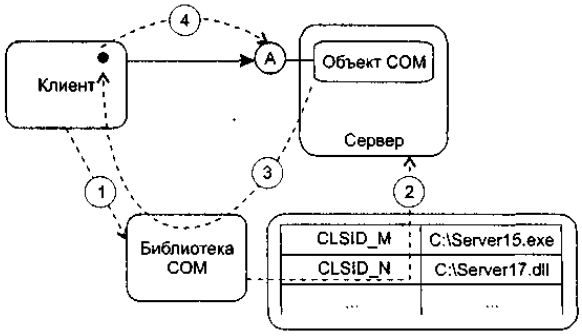


Рис. 13.19. Создание одиночного COM-объекта: *1* —клиент вызывает CoCreatelnstance (CLSID M, IID A); *2* — библиотека COM находит сервер и запускает его; *3* — библиотека COM возвращает указатель на интерфейс А; *4* — теперь клиент может вызывать операции COM-объекта

Параметры функции CoCreateInstance, используемой клиентом, позволяют также задать тип сервера, который нужно запустить (например, «в процессе» или локальный).

В более общем случае клиент может создать несколько COM-объектов одного и того же класса. Для этого клиент использует фабрику класса (class factory) — COM-объект, способный генерировать объекты одного конкретного класса.

Фабрика класса поддерживает интерфейс IClassfactory, включающий две операции. Операция Createlnstance создает COM-объект — экземпляр конкретного класса, имеет параметр — идентификатор интерфейса, указатель на который надо вернуть клиенту. Операция LockServer позволяет сохранять сервер фабрики загруженным в память.

Клиент вызывает фабрику с помощью функции библиотеки COM CoGetClassObject:

CoGetClassObject (<CLSID создаваемого обьекта>, < IID интерфейса IClassFactory>)

В качестве третьего параметра функции можно задать тип запускаемого сервера.

Библиотека COM запускает фабрику класса и возвращает указатель на интерфейс IClassFactory этой фабрики. Дальнейший порядок работы с помощью фабрики иллюстрирует рис. 13.20.

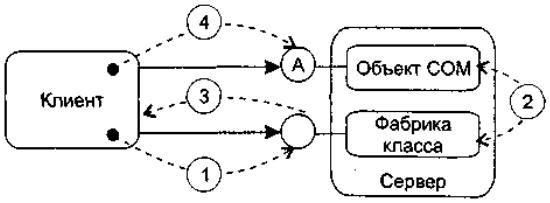


Рис. 13.20. Создание COM-объекта с помощью фабрики класса: *1* — клиент вызывает IClassFactory :: Createlnstance (IID A); *2* — фабрика класса создает COM-объект и получает указатель на его интерфейс; *3* — фабрика класса возвращает указатель на интерфейс А COM-объекта; *4* — теперь клиент может вызывать операции COM-объекта

Клиент вызывает операцию IClassFactory::CreateInstance фабрики, в качестве параметра которой задает идентификатор необходимого интерфейса объекта (IID). В ответ фабрика класса создает COM-объект и возвращает указатель на заданный интерфейс. Теперь клиент применяет возвращенный указатель для вызова операций COM-объекта.

Очень часто возникает следующая ситуация — существующий COM-класс заменили другим, поддерживающим как старые, так и дополнительные интерфейсы и имеющим другой CLSID. Появляется задача — обеспечить использование нового COM-класса старыми клиентами. Обычным решением является запись в системный реестр соответствия между старым и новым CLSID. Запись выполняется с помощью библиотечной функции CoTreatAsClass:

CoTreatAsClass (<старый CLSID>, <новый CLSID>).

Повторное использование COM-объектов

Известно, что основным средством повторного использования существующего кода является наследование реализации (новый класс наследует реализацию операций существующего класса). COM не поддерживает это средство. Причина — в типовой COM-среде базовые объекты и объекты-наследники создаются, выпускаются и обновляются независимо. В этих условиях изменения базовых объектов могут вызвать непредсказуемые последствия в объектах-наследниках их реализации. COM предлагает другие средства повторного использования — включение и агрегирование.

Применяются следующие термины:

- внутренний объект — это базовый объект;
- внешний объект — это объект, повторно использующий услуги внутреннего объекта.

При включении (делегировании) внешний объект является обычным клиентом внутреннего объекта. Как показано на рис. 13.21, когда клиент вызывает операцию внешнего объекта, эта операция, в свою очередь, вызывает операцию внутреннего объекта.

Рис. 13.21. Повторное использование COM-объекта с помощью включения

При этом внутренний объект ничего не замечает.

Достоинство включения — простота. Недостаток — низкая эффективность при длинной цепочке «делегирующих» объектов.

Недостаток включения устраняет агрегирование. Оно позволяет внешнему объекту обманывать клиентов — представлять в качестве собственных интерфейсы, реализованные внутренним объектом. Как показано на рис. 13.22, когда клиент запрашивает у внешнего объекта указатель на такой интерфейс, ему возвращается указатель на внутренний, агрегированный интерфейс. Клиент об агрегировании ничего не знает, зато внутренний объект обязательно должен знать о том, что он агрегирован.

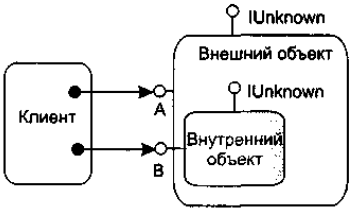


Рис. 13.22. Повторное использование COM-объекта с помощью агрегирования

Зачем требуется такое знание? В чем причина? Ответ состоит в необходимости особой реализации внутреннего объекта. Она должна обеспечить правильный подсчет ссылок и корректную работу операции QueryInterface.

Представим две практические задачи:

- запрос клиентом у внутреннего объекта (с помощью операции QueryInterface) указателя на интерфейс внешнего объекта;
- изменение клиентом счетчика ссылок внутреннего объекта (с помощью операции AddRef) и информирование об этом внешнего объекта.

Ясно, что при автономном и независимом внутреннем объекте их решить нельзя. В противном же случае решение элементарно — внутренний объект должен отказаться от собственного интерфейса IUnknown и применять только операции IUnknown внешнего объекта. Иными словами, адрес собственного IUnknown должен быть замещен адресом IUnknown агрегирующего объекта. Это указывается при создании внутреннего объекта (за счет добавления адреса как параметра операции CoCreateInstance или операции IClassFactory::CreateInstance).

Маршалинг

Клиент может содержать прямую ссылку на COM-объект только в одном случае — когда COM-объект размещен в сервере «в процессе». В случае локального или удаленного сервера, как показано на рис. 13.23, он ссылается на посредника.

Посредник — COM-объект, размещенный в клиентском процессе и предоставляющий клиенту те же интерфейсы, что и запрашиваемый объект. Запрос клиентом операции через такую ссылку приводит к исполнению кода посредника.

Посредник принимает параметры, переданные клиентом, и упаковывает их для дальнейшей пересылки. Эта процедура называется маршалингом. Затем посредник (с помощью средства коммуникации) посылает запрос в процесс, который на самом деле реализует COM-объект.

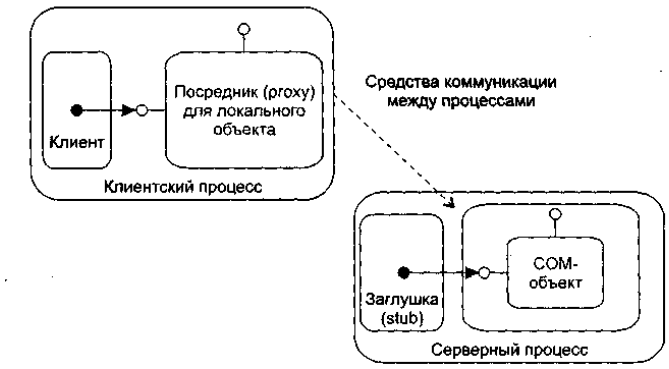


Рис. 13.23. Организация маршалинга и демаршалинга

По прибытии в процесс локального сервера запрос передается заглушке. Заглушка распаковывает параметры запроса и вызывает операцию COM-объекта. Эта процедура называется демаршалингом. После завершения COM-операции результаты возвращаются в обратном направлении.

Код посредника и заглушки автоматически генерируется компилятором MIDL (Microsoft IDL) по IDL-описанию интерфейса.

IDL-описаниеи библиотека типа

Помимо информации об интерфейсах, IDL-описание может содержать информацию о библиотеке типа.

Библиотека типа определяет важные для клиента характеристики COM-объекта: имя его класса, поддерживаемые интерфейсы, имена и адреса элементов интерфейса.

Рассмотрим пример приведенного ниже IDL-описания объекта для работы с файлами. Оно состоит из 3 частей. Первые две части описывают интерфейсы IРаботаСФайлами и IПреобразованиеФорматов, третья часть— библиотеку типа ФайлыБибл. По первым двум частям компилятор MIDL генерирует код посредников и заглушек, по третьей части — код библиотеки типа:

```
----------1-я часть
[ object,
                uuid(E7CDODOO-1827-11CF-9946-444553540000) ]
interface IРаботаСФайлами; IUnknown
    { import "unknown.idl"
                    HRESULT ОткрытьФайл ([in] OLECHAR имя[31]);
                    HRESULT ЗаписатьФайл ([in] OLECHAR имя[31]);
                    HRESULT ЗакрытьФайл ([in] OLECHAR имя[31]);
    }
---------- 2-я часть
[ object.
                uuid(5FBDD020-1863-11CF-9946-444553540000) ]
interface IПреобразованиеФорматов: IUnknown
    { HRESULT ПреобразоватьФормат ([in] OLECHAR имя[31],
                                   [in] OLECHAR формат[31]);
    }
------------ 3-я часть
[ uuid(B253E460-1826-11CF-9946-444553540000),
        version (1.0)]
library ФайлыБибл
{ importlib ("stdole32.tlb");
[uuid(B2ECFAAO-1827-11CF-9946-444553540000) ]
coclass СоФайлы
    {   interface IРаботаСФайлами;
        interface IпреобразованиеФорматов;
    }
}
```

Описание библиотеки типа начинается с ее уникального имени (записывается после служебного слова uuid), затем указывается номер версии библиотеки.

После служебного слова library записывается символьное имя библиотеки (ФайлыБибл).

Далее в операторе importlib указывается файл со стандартными определениями IDL - stdole32.tlb.

Тело описания библиотеки включает только один элемент — COM-класс (coclass), на основе которого создается COM-объект.

В начале описания COM-класса приводится его уникальное имя (это и есть идентификатор класса — CLSID), затем символьное имя — СоФайлы. В теле класса перечислены имена поддерживаемых интерфейсов — РаботаСФайлами и IПреобразованиеФорматов.

Как показано на рис. 13.24, доступ к библиотеке типа выполняется по стандартному интерфейсу ITypeLib, а доступ к

отдельным элементам библиотеки — по интерфейсу ITypeInfo.

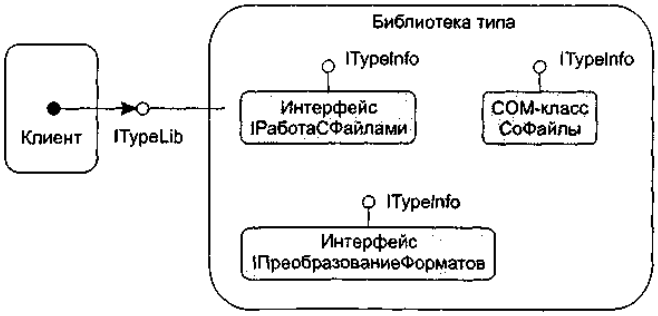


Рис. 13.24. Доступ к библиотеке типа

Диаграммы размещения

Диаграмма размещения (развертывания) — вторая из двух разновидностей диаграмм реализации UML, моделирующих физические аспекты объектно-ориентированных систем. Диаграмма размещения показывает конфигурацию обрабатывающих узлов в период работы системы, а также компоненты, «живущие» в них.

Элементами диаграмм размещения являются узлы, а также отношения зависимости и ассоциации. Как и другие диаграммы, диаграммы размещения могут включать примечания и ограничения. Кроме того, диаграммы размещения могут включать компоненты, каждый из которых должен жить в некотором узле, а также содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты. При необходимости визуализации конкретного варианта аппаратной топологии в диаграммы размещения могут помещаться объекты.

Узлы

Узел — физический элемент, который существует в период работы системы и представляет компьютерный ресурс, имеющий память, а возможно, и способность обработки. Графически узел изображается как куб с именем (рис. 13.25).

Рис. 13.25. Обозначение узла

Как и класс, узел может иметь дополнительную секцию, отображающую размещаемые в нем элементы (рис. 13.26).

Рис. 13.26. Размещение компонентов в узле

Сравним узлы с компонентами. Конечно, у них есть сходные характеристики:

- наличие имени;
- возможность быть вложенным;
- наличие экземпляров.

Последняя характеристика говорит о том, что на предыдущем рисунке изображен тип Контроллера. Изображение конкретного экземпляра, принадлежащего этому типу, представлено на рис. 13.27.

Теперь обсудим отличия узлов от компонентов. Во-первых, они принадлежат к разным уровням иерархии в физической реализации системы. Физически система состоит из узлов, а узлы — из компонентов. Во-вторых, у каждого из них свое назначение. Компонент предназначен для физической упаковки и материализации набора логических элементов (классов и кооперации). Узел же является тем местом, где физически размещаются компоненты, то есть играет роль «квартиры» для компонентов.

Рис. 13.27. Экземпляр узла

Отношение между узлом и компонентами, которые он размещает, можно отобразить явно. Отношение зависимости между узлом Контроллер и компонентами Вводы.exe, Выводы.exe иллюстрирует рис. 13.28. Правда, чаще всего такие отношения не отображаются. Их удобно представлять в отдельной спецификации узла.

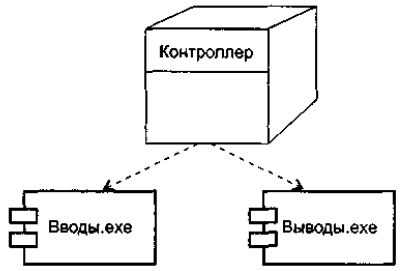


Рис. 13.28. Зависимость узла от компонентов

Группировку набора объектов или компонентов, размещаемых в узле, обычно называют распространяемым модулем.

Для узла, как и для класса, можно задать свойства и операции. Например, можно определить свойства БыстродействиеПроцессора, ЕмкостьПамяти, а также операции Запустить, Выключить.

Использование диаграмм размещения

Диаграммы размещения используют для моделирования статического представления того, как размещается система. Это представление поддерживает распространение, поставку и инсталляцию частей, образующих физическую систему.

Графически диаграмма размещения — это граф из узлов (или экземпляров узлов), соединенных ассоциациями, которые показывают существующие коммуникации. Экземпляры узлов могут содержать экземпляры компонентов, живущих или запускаемых в узлах. Экземпляры компонентов могут содержать объекты. Как показано на рис. 13.29, компоненты соединяются друг с другом пунктирными стрелками зависимостей (прямо или через интерфейсы).

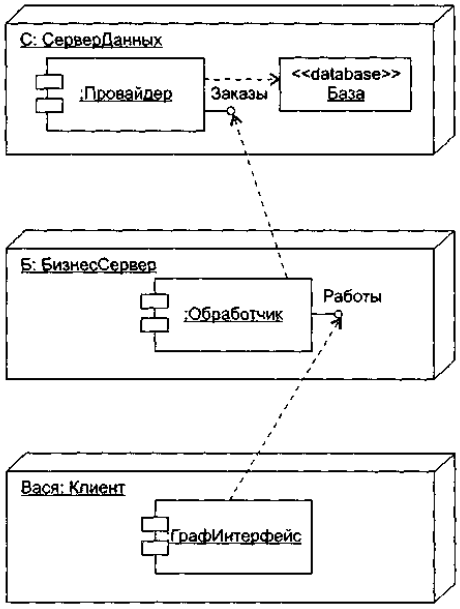


Рис. 13.29. Моделирование размещения компонентов

На этой диаграмме изображена типовая трехуровневая система:

- уровень базы данных реализован экземпляром С узла СерверДанных;
- уровень бизнес-логики представлен экземпляром Б узла БизнесСервер;
- уровень графического интерфейса пользователя образован экземпляром Вася узла Клиент.

В узле сервера данных показано размещение анонимного экземпляра компонента Провайдер и объекта База со стереотипом <<database>>. Узел бизнес-сервера содержит анонимный экземпляр компонента Обработчик, а узел клиента — анонимный экземпляр компонента ГрафИнтерфейс. Кроме того, здесь явно отображены интерфейсы компонентов Провайдер и Обработчик, имеющие, соответственно, имена Заказы и Работы.

Как представлено на рис. 13.30, перемещение компонентов от узла к узлу (или объектов от компонента к компоненту) отмечается стереотипом <<becomes>> на отношении зависимости. В этом случае считают, что компонент (объект) резидентен в узле (компоненте) только в пределах некоторого кванта времени. На рисунке видим, что возможность миграции предоставлена объектам X и Y.

Иногда полезно определить физическое распределение компонентов по процессорам и другим устройствам системы. Есть три способа моделирования распределения:

- графически распределение не показывать, а документировать его в текстовых спецификациях узлов;
- соединять каждый узел с размещаемыми компонентами отношениями зависимости;

- в дополнительной секции узла указывать список размещаемых компонентов.

Диаграмма размещения, иллюстрирующая третий способ моделирования, показана на рис. 13.31.

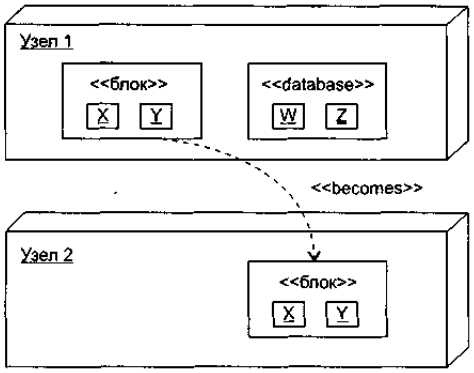


Рис. 13.30. Моделирование перемещения компонентов и объектов

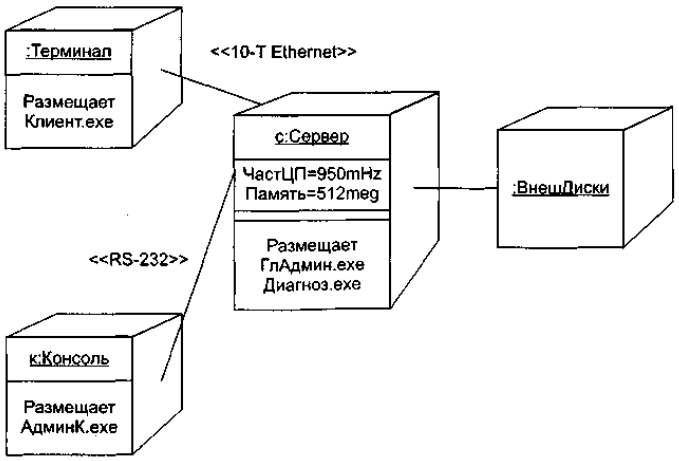


Рис. 13.31. Распределение компонентов в системе

На рисунке показаны два анонимных экземпляра узлов (:ВнешДиски, :Терминал) и два экземпляра узлов с именем (с для Сервера и к для Консоли). Каждый процессор нарисован с дополнительной секцией, в которой показаны размещенные компоненты. В экземпляре Сервера, кроме того, отображены его свойства (ЧастЦП, Память) и их значения.

С помощью стереотипов заданы характеристики физических соединений между процессорами: одно из них определено как Ethernet-соединение, другое — как последовательное RS-232-соединение.

Метрики связности по методам

Д. Биемен и Б. Кенг предложили метрики связности класса, которые основаны на прямых и косвенных соединениях между парами методов [15]. Если существуют общие экземплярные переменные (одна или несколько), используемые в паре методов, то говорят, что эти методы соединены прямо. Пара методов может быть соединена косвенно, через другие прямо соединенные методы.

На рис. 14.2 представлены отношения между элементами класса Stack. Прямоугольниками обозначены методы класса, а овалами — экземплярные переменные. Связи показывают отношения использования между методами и переменными.

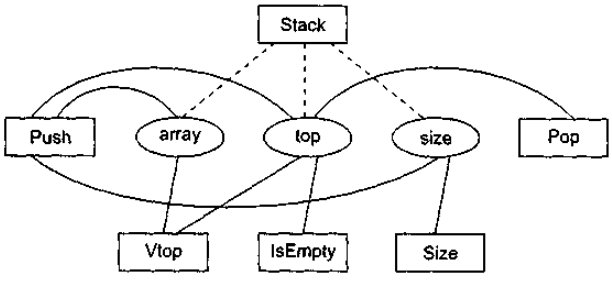


Рис. 14. 2. Отношения между элементами класса Stack

Из рисунка видно, что экземплярная переменная top используется методами Stack, Push, Pop, Vtop и IsEmpty. Таким образом, все эти методы попарно прямо соединены. Напротив, методы Size и Pop соединены косвенно: Size соединен прямо с Push, который, в свою очередь, прямо соединен с Pop. Метод Stack является конструктором класса, то есть функцией инициализации. Обычно конструктору доступны все экземплярные переменные класса, он использует эти переменные совместно со всеми другими методами. Следовательно, конструкторы создают соединения и между такими методами, которые никак не связаны друг с другом. Поэтому ни конструкторы, ни деструкторы здесь не учитываются. Связи между конструктором и экземплярными переменными на рис. 14.2 показаны пунктирными линиями.

Для формализации модели вводятся понятия абстрактного метода и абстрактного класса.

Абстрактный метод AM(M) — это представление реального метода *M* в виде множества экземплярных переменных, которые прямо или косвенно используются методом.

Экземплярная переменная прямо используется методом *M,* если она появляется в методе как лексема данных. Экземплярная переменная может быть определена в том же классе, что и *M,* или же в родительском классе этого класса. Множество экземплярных переменных, прямо используемых методом *M,* обозначим как *DU(M).*

Экземплярная переменная косвенно используется методом *M,* если: 1) экземплярная переменная прямо используется другим методом *M′*, который вызывается (прямо или косвенно) из метода *M;* 2) экземплярная переменная, прямо используемая методом *M′,* находится в том же объекте, что и *M.*

Множество экземплярных переменных, косвенно используемых методом *M,* обозначим как *IU(M).*

Количественно абстрактный метод формируется по выражению:

$$AM(M) = DU(M) \cup IU(M).$$

Абстрактный класс AC(C) — это представление реального класса *C* в виде совокупности абстрактных методов, причем каждый абстрактный метод соответствует видимому методу класса *C.* Количественно абстрактный класс формируется по выражению:

$$AC(C) = [[AM(M) \mid M \in V(C)]],$$

где *V(C)* — множество всех видимых методов в классе *C* и в классах — предках для *C.*

Отметим, что AM-представления различных методов могут совпадать, поэтому в AC могут быть дублированные элементы. В силу этого AC записывается в форме мультимножества (двойные квадратные скобки рассматриваются как его обозначение).

Локальный абстрактный класс LAC(C) — это совокупность абстрактных методов, где каждый абстрактный метод соответствует видимому методу, определенному только внутри класса *C.* Количественно абстрактный класс формируется по выражению:

$$LAC(C)=[[AM(M)|M \in LV(C)]],$$

где $LV(C)$ — множество всех видимых методов, определенных в классе *C.*

Абстрактный класс для стека, приведенного в табл. 14.2, имеет вид:

$$AC\ (Stack) = [[\{top\}, \{size\}, \{array, top\}, \{array, top, size\}, \{pop\}]].$$

Поскольку класс Stack не имеет суперкласса, то справедливо:

$$AC\ (Stack) = LAC\ (Stack)$$

Пусть $NP(C)$ — общее количество пар абстрактных методов в $AC(C).$ NP определяет максимально возможное количество прямых или косвенных соединений в классе. Если в классе *C* имеются *N* методов, тогда $NP(C) = N*(N- 1)/2$. Обозначим:

- $NDC(C)$ — количество прямых соединений AC(Q;
- $NIC(C)$ — количество косвенных соединений в AC(C).

Тогда метрики связности класса можно представить в следующем виде:

- *сильная связность класса (Tight Class Cohesion (TCC))* определяется относительным количеством прямо соединенных методов:

$$TCC(C) = NDC(C) / NP\ (C);$$

- *слабая связность класса (Loose Class Cohesion (LCC))* определяется относительным количеством прямо или косвенно соединенных методов:

$$LCC(C) = (NDC(C) + NIC(C)) / NP(C).$$

Очевидно, что всегда справедливо следующее неравенство:

$$LCC(C)>=TCC(C).$$

Для класса Stack метрики связности имеют следующие значения:

$$TCC(Stack)=7/10=0,7$$

$$LCC(Stack)=10/10=l$$

Метрика TCC показывает, что 70% видимых методов класса Stack соединены прямо, а метрика LCC показывает, что все видимые методы класса Stack соединены прямо или косвенно.

Метрики TCC и LCC индицируют степень связанности между видимыми методами класса. Видимые методы либо определены в классе, либо унаследованы им. Конечно, очень полезны метрики связности для видимых методов, которые определены только внутри класса — ведь здесь исключается влияние связности суперкласса. Очевидно, что метрики локальной связности класса определяются на основе локального абстрактного класса. Отметим, что для локальной связности экземплярные переменные и вызываемые методы могут включать унаследованные переменные.

Сцепление объектов

В классическом методе Л. Констентайна и Э. Йордана определены шесть типов сцепления, которые ориентированы на процедурное проектирование [77].

Принципиальное преимущество объектно-ориентированного проектирования в том, что природа объектов приводит к созданию слабо сцепленных систем. Фундаментальное свойство объектно-ориентированного проектирования заключается в скрытости содержания объекта. Как правило, содержание объекта невидимо внешним элементам. Степень автономности объекта достаточно высока. Любой объект может быть замещен другим объектом с таким же интерфейсом.

Тем не менее наследование в объектно-ориентированных системах приводит к другой форме сцепления. Объекты, которые наследуют свойства и операции, сцеплены с их суперклассами. Изменения в суперклассах должны проводиться осторожно, так как эти изменения распространяются во все классы, которые наследуют их характеристики.

Таким образом, сами по себе объектно-ориентированные механизмы не гарантируют минимального сцепления. Конечно, классы — мощное средство абстракции данных. Их введение уменьшило поток данных между модулями и, следовательно, снизило общее сцепление внутри системы. Однако количество типов зависимостей между модулями выросло. Появились отношения наследования, делегирования, реализации и т. д. Более разнообразным стал состав модулей в системе (классы,

объекты, свободные функции и процедуры, пакеты). Отсюда вывод: необходимость измерения и регулирования сцепления в объектно-ориентированных системах обострилась.

Рассмотрим объектно-ориентированные метрики сцепления, предложенные М. Хитцем и Б. Монтазери [38].

Зависимость изменения между классами

Зависимость изменения между классами CDBC (Change Dependency Between Classes) определяет потенциальный объем изменений, необходимых после модификации класса-сервера SC (server class) на этапе сопровождения. До тех пор, пока реальное количество необходимых изменений класса-клиента CC (client class) неизвестно, CDBC указывает количество методов, на которые влияет изменение SC.

CDBC зависит от:

- области видимости изменяемого класса-сервера внутри класса-клиента (определяется типом отношения между CS и CC);
- вида доступа CC к CS (интерфейсный доступ или доступ реализации).

Возможные типы отношений приведены в табл. 14.3, где n — количество методов класса CC, $\mathfrak{D}$ — количество методов CC, потенциально затрагиваемых изменением.

Таблица 14.3. Вклад отношений между клиентом и сервером в зависимость изменения

Тип отношения	$\mathfrak{D}$
SC не используется классом CC	0
SC — класс экземплярной переменной в классе CC	n
Локальные переменные типа SC используются внутри /-методов класса CC	j
SC является суперклассом CC	n
SC является типом параметра для/-методов класса CC	j
CC имеет доступ к глобальной переменной класса SC	n

Конечно, здесь предполагается, что те элементы класса-сервера SC, которые доступны классу-клиенту CC, являются предметом изменений. Авторы исходят из следующей точки зрения: если класс SC является «зрелой» абстракцией, то предполагается, что его интерфейс более стабилен, чем его реализация. Таким образом, многие изменения в реализации SC могут выполняться без влияния на его интерфейс. Поэтому вводится фактор стабильности интерфейса для класса-сервера, он обозначается как к ($0 < k < 1$). Вклад доступа к интерфейсу в зависимость изменения можно учесть умножением на ($1 - k$).

Метрика для вычисления степени CDBC имеет вид:

$$A = \sum_{\substack{\text{Доступ } i \\ \text{к реализации}}} \alpha_i + (1-k) \times \sum_{\substack{\text{Доступ i} \\ \text{к интерфейсу}}} \alpha_i \quad ;$$

$$\text{CDBC(CC, SC)} = \min(n, A).$$

Пути минимизации CDBC:

1) ограничение доступа к интерфейсу класса-сервера;

2) ограничение видимости классов-серверов (спецификаторами доступа public, protected, private).

Локальность данных

Локальность данных LD (Locality of Data) — метрика, отражающая качество абстракции, реализуемой классом. Чем выше локальность данных, тем выше самодостаточность класса. Эта характеристика оказывает сильное влияние на такие внешние характеристики, как повторная используемость и тестируемость класса.

Метрика LD представляется как отношение количества локальных данных в классе к общему количеству данных, используемых этим классом.

Будем использовать терминологию языка C++. Обозначим как $M_i(1 \le i \le n)$ методы класса. В их число не будем включать методы чтения/записи экземплярных переменных. Тогда формулу для вычисления локальности данных можно записать в виде:

$$LD = \frac{\sum_{i=1}^{a} |L_i|}{\sum_{i=1}^{a} |T_i|},$$

где:

- $L_i(1 \le i \le n)$ — множество локальных переменных, к которым имеют доступ методы M_i (прямо или с помощью методов чтения/записи). Такими переменными являются: непубличные экземплярныё переменные класса; унаследованные защищенные экземплярныё переменные их суперклассов; статические переменные, локально определенные в M_i ;
- $T_i(1 \le i \le n)$ — множество всех переменных, используемых в M_i, кроме динамических локальных переменных, определенных в M_i.

Для обеспечения надежности оценки здесь исключены все вспомогательные переменные, определенные в M_i , — они не играют важной роли в проектировании.

Защищенная экземплярная переменная, которая унаследована классом *C,* является локальной переменной для его

экземпляра (и следовательно, является элементом L_i), даже если она не объявлена в классе C. Использование такой переменной методами класса не вредит локальности данных, однако нежелательно, если мы заинтересованы уменьшить значение CDBC.

Набор метрик Чидамбера и Кемерера

В 1994 году С. Чидамбер и К. Кемерер (Chidamber и Кетегег) предложили шесть проектных метрик, ориентированных на классы [24]. Класс — фундаментальный элемент объектно-ориентированной (ОО) системы. Поэтому измерения и метрики для отдельного класса, иерархии классов и сотрудничества классов бесценны для программного инженера, который должен оценить качество проекта.

Набор Чидамбера-Кемерера наиболее часто цитируется в программной индустрии и научных исследованиях. Рассмотрим каждую из метрик набора.

Метрика 1: Взвешенные методы на класс WMC (Weighted Methods Per Class)

Допустим, что в классе C определены n методов со сложностью $c_1...,c_2,..., c_n$. Для оценки сложности может быть выбрана любая метрика сложности (например, цикломатическая сложность). Главное — нормализовать эту метрику так, чтобы номинальная сложность для метода принимала значение 1. В этом случае

$$WMC = \sum_{i=1}^{n} C_i$$

Количество методов и их сложность являются индикатором затрат на реализацию и тестирование классов. Кроме того, чем больше методов, тем сложнее дерево наследования (все подклассы наследуют методы их родителей). С ростом количества методов в классе его применение становится все более специфическим, тем самым ограничивается возможность многократного использования. По этим причинам метрика WMC должна иметь разумно низкое значение.

Очень часто применяют упрощенную версию метрики. При этом полагают C_i= 1, и тогда WMC — количество методов в классе.

Оказывается, что подсчитывать количество методов в классе достаточно сложно. Возможны два противоположных варианта учета.

1. Подсчитываются только методы текущего класса. Унаследованные методы игнорируются. Обоснование — унаследованные методы уже подсчитаны в тех классах, где они определялись. Таким образом, инкрементность класса — лучший показатель его функциональных возможностей, который отражает его право на существование. Наиболее важным источником информации для понимания того, что делает класс, являются его собственные операции. Если класс не может отреагировать на сообщение (например, в нем отсутствует собственный метод), тогда он пошлет сообщение родителю.
2. Подсчитываются методы, определенные в текущем классе, и все унаследованные методы. Этот подход подчеркивает важность пространства состояний в понимании класса (а не инкрементности класса).

Существует ряд промежуточных вариантов. Например, подсчитываются текущие методы и методы, прямо унаследованные от родителей. Аргумент в пользу данного подхода — на поведение дочернего класса наиболее сильно влияет специализация родительских классов.

На практике приемлем любой из описанных вариантов. Главное — не менять вариант учета от проекта к проекту. Только в этом случае обеспечивается корректный сбор метрических данных.

Метрика WMC дает относительную меру сложности класса. Если считать, что все методы имеют одинаковую сложность, то это будет просто количество методов в классе. Существуют рекомендации по сложности методов. Например, М. Лоренц считает, что средняя длина метода должна ограничиваться 8 строками для Smalltalk и 24 строками для C++ [45]. Вообще, класс, имеющий максимальное количество методов среди классов одного с ним уровня, является наиболее сложным; скорее всего, он специфичен для данного приложения и содержит наибольшее количество ошибок.

Метрика 2: Высота дерева наследования DIT (Depth of Inheritance Tree)

DIT определяется как максимальная длина пути от листа до корня дерева наследования классов. Для показанной на рис. 14.3 иерархии классов метрика DIT равна 3.

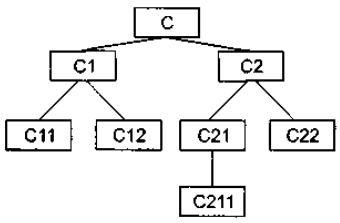


Рис. 14.3. Дерево наследования классов

Соответственно, для отдельного класса DIT, это длина максимального пути от данного класса до корневого класса в иерархии классов.

По мере роста DIT вероятно, что классы нижнего уровня будут наследовать много методов. Это приводит к трудностям в предсказании поведения класса. Высокая иерархия классов (большое значение DIT) приводит к большей сложности проекта, так как означает привлечение большего количества методов и классов.

Вместе с тем, большое значение DIT подразумевает, что многие методы могут использоваться многократно.

Метрика 3: Количество детей NOC (Number of children)

Подклассы, которые непосредственно подчинены суперклассу, называются его детьми. Значение NOC равно количеству детей, то есть количеству непосредственных наследников класса в иерархии классов. На рис. 14.3 класс *C*2 имеет двух детей — подклассы *C*21 и *C*22.

С увеличением NOC возрастает многократность использования, так как наследование — это форма повторного использования.

Однако при возрастании NOC ослабляется абстракция родительского класса. Это означает, что в действительности некоторые из детей уже не являются членами родительского класса и могут быть неправильно использованы.

Кроме того, количество детей характеризует потенциальное влияние класса на проект. По мере роста NOC возрастает количество тестов, необходимых для проверки каждого ребенка.

Метрики DIT и NOC — количественные характеристики формы и размера структуры классов. Хорошо структурированная объектно-ориентированная система чаще бывает организована как лес классов, чем как сверхвысокое дерево. По мнению Г. Буча, следует строить сбалансированные по высоте и ширине структуры наследования: обычно не выше, чем 7 ± 2 уровня, и не шире, чем 7 + 2 ветви [22].

Метрика 4: Сцепление между классами объектов CBO (Coupling between object classes)

CBO — это количество сотрудничеств, предусмотренных для класса, то есть количество классов, с которыми он соединен. Соединение означает, что методы данного класса используют методы или экземплярные переменные другого класса.

Другое определение метрики имеет следующий вид: CBO равно количеству сцеплений класса; сцепление образует вызов метода или свойства в другом классе.

Данная метрика характеризует статическую составляющую внешних связей классов.

С ростом CBO многократность использования класса, вероятно, уменьшается. Очевидно, что чем больше независимость класса, тем легче его повторно использовать в другом приложении.

Высокое значение CBO усложняет модификацию и тестирование, которое следует за выполнением модификации. Понятно, что, чем больше количество сцеплений, тем выше чувствительность всего проекта к изменениям в отдельных его частях. Минимизация межобъектных сцеплений улучшает модульность и содействует инкапсуляции проекта.

CBO для каждого класса должно иметь разумно низкое значение. Это согласуется с рекомендациями по уменьшению сцепления стандартного программного обеспечения.

Метрика 5: Отклик для класса RFC (Response For a Class)

Введем вспомогательное определение. Множество отклика класса RS — это множество методов, которые могут выполняться в ответ на прибытие сообщений в объект этого класса. Формула для определения RS имеет вид

$$RS = \{M\} U_{all\_i} \{R_i\},$$

где $\{R_i\}$ — множество методов, вызываемых методом г, $\{M\}$ — множество всех методов в классе.

Метрика RFC равна количеству методов во множестве отклика, то есть равна мощности этого множества:

$$\text{RFC} - \text{card}\{\text{RS}\}.$$

Приведем другое определение метрики: RFC — это количество методов класса плюс количество методов других классов, вызываемых из данного класса.

Метрика RFC является мерой потенциального взаимодействия данного класса с другими классами, позволяет судить о динамике поведения соответствующего объекта в системе. Данная метрика характеризует динамическую составляющую внешних связей классов.

Если в ответ на сообщение может быть вызвано большое количество методов, то усложняются тестирование и отладка класса, так как от разработчика тестов требуется больший уровень понимания класса, растет длина тестовой последовательности.

С ростом RFC увеличивается сложность класса. Наихудшая величина отклика может использоваться при определении времени тестирования.

Метрика 6: Недостаток связности в методах LCOM (Lack of Cohesion in Methods)

Каждый метод внутри класса обращается к одному или нескольким свойствам (экземплярным переменным). Метрика *LCOM* показывает, насколько методы не связаны друг с другом через свойства (переменные). Если все методы обращаются к одинаковым свойствам, то LCOM = 0.

Введем обозначения:

- НЕ СВЯЗАНЫ — количество пар методов без общих экземплярных переменных;
- СВЯЗАНЫ — количество пар методов с общими экземплярными переменными.
- I_j— набор экземплярных переменных, используемых методом M_j

Очевидно, что

$$\text{НЕ СВЯЗАНЫ} = \text{card}\ \{I_{ij} \mid I_i \cap I_j = 0\},$$
$$\text{СВЯЗАНЫ} = \text{card}\ \{I_{ij} \mid I_i \cap I_j \neq 0\}.$$

Тогда формула для вычисления недостатка связности в методах примет вид

$$LCOM = \begin{cases} \text{НЕ СВЯЗАНЫ - СВЯЗАНЫ, если (НЕСВЯЗАНЫ} > \text{СВЯЗАНЫ)}; \\ 0 \text{ в противном случае.} \end{cases}$$

Можно определить метрику по-другому: LCOM — это количество пар методов, не связанных по свойствам класса, минус количество пар методов, имеющих такую связь.

Рассмотрим примеры применения метрики LCOM.

Пример 1: В классе имеются методы: *M1, M2, M3, M4.* Каждый метод работает со своим набором экземплярных переменных:

$$I_1=\{a, b\};\ I_2=\{a, c\};\ I_3=\{x, y\};\ I_4=\{m, n\}.$$

В этом случае

$$\text{НЕ СВЯЗАНЫ} = \text{card}\ (I_{13}, I_{14}, I_{23}, I_{24}, I_{34}) = 5;\ \text{СВЯЗАНЫ} = \text{card}\ (I_{12}) = 1.$$

$$\text{LCOM} = 5\text{-}1=4.$$

Пример 2: В классе используются методы: *M1, M2, M3.* Для каждого метода задан свой набор экземплярных переменных:

$$I_1 = \{a,b\};I_2=\{a,c\};I_3=\{x,y\},$$

$$\text{НЕ СВЯЗАНЫ} = \text{card}\ (I_{13}, I_{23}) = 2;\ \text{СВЯЗАНЫ} = \text{card}\ (I_{12}) = 1,$$

$$\text{LCOM} = 2\text{-}\ 1 = 1.$$

Связность методов внутри класса должна быть высокой, так как это содействует инкапсуляции. Если LCOM имеет высокое значение, то методы слабо связаны друг с другом через свойства. Это увеличивает сложность, в связи с чем возрастает вероятность ошибок при проектировании.

Высокие значения LCOM означают, что класс, вероятно, надо спроектировать лучше (разбиением на два или более отдельных класса). Любое вычисление LCOM помогает определить недостатки в проектировании классов, так как эта метрика характеризует качество упаковки данных и методов в оболочку класса.

Вывод: связность в классе желательно сохранять высокой, то есть следует добиваться низкого значения LCOM.

Набор метрик Чидамбера-Кемерера — одна из пионерских работ по комплексной оценке качества ОО-проектирования. Известны многочисленные предложения по усовершенствованию, развитию данного набора. Рассмотрим некоторые из них.

Недостатком метрики WMC является зависимость от реализации. Приведем пример. Рассмотрим класс, предлагающий операцию интегрирования. Возможны две реализации:

1) несколько простых операций:

Set_interval (min. max)

Setjnethod (method)

Set_precision (precision)

Set_function_to_integrate (function)

Integrate;

2) одна сложная операция:

Integrate (function, min, max. method, precision)

Для обеспечения независимости от этих реализаций можно определить метрику WMC2:

$$WMC2 = \sum_{i=1}^{n} (\text{Количество параметров } i-\text{го метода}).$$

Для нашего примера WMC2 = 5 и для первой, и для второй реализации. Заметим, для первой реализации WMC = 5, а для второй реализации WMC = 1.

Дополнительно можно определить метрику *Среднее число аргументов метода ANAM* (Average Number of Arguments per Method):

$$\text{ANAM} = \text{WMC2/WMC}.$$

Полезность метрики ANAM объяснить еще легче. Она ориентирована на принятые в ОО-проектировании решения — применять простые операции с малым количеством аргументов, а несложные операции — с многочисленными аргументами.

Еще одно предложение — ввести метрику, симметричную метрике LCOM. В то время как формула метрики LCOM имеет вид:

$$\text{LCOM} = \max\ (0, \text{НЕ СВЯЗАНЫ} - \text{СВЯЗАНЫ}),$$

симметричная ей метрика *Нормализованная NLCOM* вычисляется по формуле:

$$\text{NLCOM} = \text{СВЯЗАНЫ}/(\text{НЕ СВЯЗАНЫ} + \text{СВЯЗАНЫ}).$$

Диапазон значений этой метрики: $0 \leq \text{NLCOM} \leq 1$, причем чем ближе NLCOM к 1, тем выше связанность класса.

В наборе Чидамбера-Кемерера отсутствует метрика для прямого измерения информационной закрытости класса. В силу этого была предложена метрика *Поведенческая закрытость информации BIH* (Behaviourial Information Hiding):

$$\text{BIH} - (\text{WEOC/WIEOC}),$$

где WEOC — *взвешенные внешние операции на класс* (фактически это WMC);

WIEOC — *взвешенные внутренние и внешние операции на класс.*

WIEOC вычисляется так же, как и WMC, но учитывает полный набор операций, реализуемых классом. Если BIH = 1, класс показывает другим классам все свои возможности. Чем меньше BIM, тем меньше видимо поведение класса. BIH может рассматриваться и как мера сложности. Сложные классы, вероятно, будут иметь малые значения BIH, а простые классы — значения, близкие к 1. Если класс с высокой WMC имеет значение BIH, близкое к 1, следует выяснить, почему он настолько видим извне.

Использование метрик Чидамбера-Кемерера

Поскольку основу логического представления ПО образует структура классов, для оценки ее качества удобно использовать метрики Чидамбера-Кемерера. Пример расчета метрик для структуры, показанной на рис. 14.4, представлен в табл. 14.4.

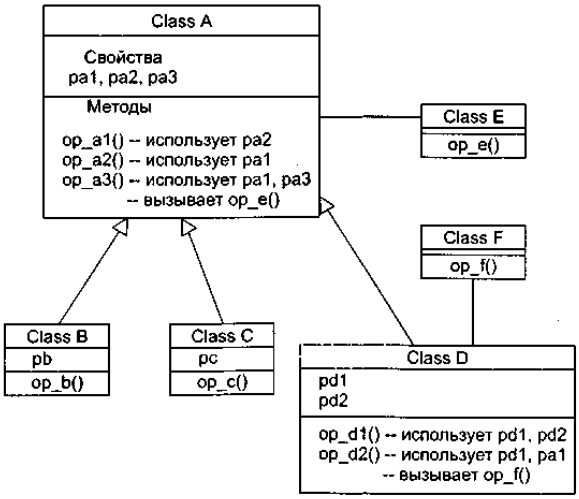


Рис. 14.4. Структура классов для расчета метрик Чидамбера-Кемерера

Прокомментируем результаты расчета. Класс Class A имеет три метода (op_al(), op_a2(), op_a3()), трех детей (Class B, Class C, Class D) и является корневым классом. Поэтому метрики WMC, NOC и DIT имеют, соответственно, значения 3, 3 и 0.

Метрика CBO для класса Class A равна 1, так как он использует один метод из другого класса (метод op_e() из класса Class E, он вызывается из метода op_a3()). Метрика RFC для класса Class A равна 4, так как в ответ на прибытие в этот класс сообщений возможно выполнение четырех методов (три объявлены в этом классе, а четвертый метод op_e() вызывается из op_a3()).

Таблица 14.4. Пример расчета метрик Чидамбера-Кемерера

Имя класса	WMC	DIT	NOC	CBO	RFC	LCOM
Class A	3	0	3	1	4	1
Class B	1	1	0	0	1	0
Class C	1	1	0	0	1	0
Class D	2	1	0	2	3	0

Для вычисления метрики LCOM надо определить количество пар методов класса. Оно рассчитывается по формуле

$$C_m^2 = m!/(2(m-2)!),$$

где m — количество методов класса.

Поскольку в классе три метода, возможны три пары: op_al( )&op_a2(), op_al( )&op_a3() и op_a2( )&op_a3(). Первая и вторая пары не имеют общих свойств, третья пара имеет общее свойство (pal). Таким образом, количество несвязанных пар равно 2, количество связанных пар равно 1, и LCOM = 2-1 = 1.

Отметим также, что для класса Class D метрика CBO равна 2, так как здесь используются свойство pal и метод op_f() из других классов. Метрика LCOM в этом классе равна 0, поскольку методы op_dl() и op_d2() связаны по свойству pdl, а отрицательное значение запрещено.

Метрики Лоренца и Кидда

Коллекция метрик Лоренца и Кидда — результат практического, промышленного подхода к оценке ОО-проектов [45].

Метрики, ориентированные на классы

М. Лоренц и Д. Кидд подразделяют метрики, ориентированные на классы, на четыре категории: метрики размера, метрики наследования, внутренние и внешние метрики.

Размерно-ориентированные метрики основаны на подсчете свойств и операций для отдельных классов, а также их средних значений для всей ОО-системы. Метрики наследования акцентируют внимание на способе повторного использования операций в иерархии классов. Внутренние метрики классов рассматривают вопросы связности и кодирования. Внешние метрики исследуют сцепление и повторное использование.

Метрика 1: Размер класса CS (Class Size)

Общий размер класса определяется с помощью следующих измерений:

- общее количество операций (вместе с приватными и наследуемыми экземплярными операциями), которые инкапсулируются внутри класса;
- количество свойств (вместе с приватными и наследуемыми экземплярными свойствами), которые инкапсулируются классом.

Метрика WMC Чидамбера и Кемерера также является взвешенной метрикой размера класса.

Большие значения CS указывают, что класс имеет слишком много обязанностей. Они уменьшают возможность повторного использования класса, усложняют его реализацию и тестирование.

При определении размера класса унаследованным (публичным) операциям и свойствам придают больший удельный вес. Причина — приватные операции и свойства обеспечивают специализацию и более локализованы в проекте.

Могут вычисляться средние количества свойств и операций класса. Чем меньше среднее значение размера, тем больше вероятность повторного использования класса.

Рекомендуемое значение CS $\leq$ 20 методов.

Метрика 2: Количество операций, переопределяемых подклассом, NOO (Number of Operations Overridden by a Subclass)

Переопределением называют случай, когда подкласс замещает операцию, унаследованную от суперкласса, своей собственной версией.

Большие значения NOO обычно указывают на проблемы проектирования. Ясно, что подкласс должен расширять операции суперкласса. Расширение проявляется в виде новых имен операций. Если же NOO велико, то разработчик нарушает абстракцию суперкласса. Это ослабляет иерархию классов, усложняет тестирование и модификацию программного обеспечения.

Рекомендуемое значение NOO $\leq$ 3 методов.

Метрика 3: Количество операций, добавленных подклассом, NOA (Number of Operations Added by a Subclass)

Подклассы специализируются добавлением приватных операций и свойств. С ростом NOA подкласс удаляется от абстракции суперкласса. Обычно при увеличении высоты иерархии классов (увеличении DIT) должно уменьшаться значение NOA на нижних уровнях иерархии.

Для рекомендуемых значений CS = 20 и DIT = 6 рекомендуемое значение NOA $\leq$ 4 методов (для класса-листа).

Метрика 4: Индекс специализации SI (Specialization Index)

Обеспечивает грубую оценку степени специализации каждого подкласса. Специализация достигается добавлением, удалением или переопределением операций:

$$SI = (NOO \times уровень) / M_{общ},$$

где *уровень* — номер уровня в иерархии, на котором находится подкласс, $M_{общ}$ — общее количество методов класса.

Пример расчета индексов специализации приведен на рис. 14.5.

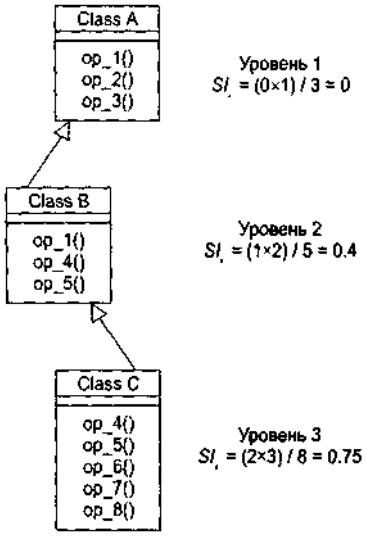


Рис. 14_05

Рис. 14.5. Расчет индексов специализации классов

Чем выше значение SI, тем больше вероятность того, что в иерархии классов есть классы, нарушающие абстракцию суперкласса.

Рекомендуемое значение SI $\leq$0,15.

Операционно-ориентированные метрики

Эта группа метрик ориентирована на оценку операций в классах. Обычно методы имеют тенденцию быть небольшими как по размеру, так и по логической сложности. Тем не менее реальные характеристики операций могут быть полезны для глубокого понимания системы.

Метрика 5: Средний размер операции OS_{AVG} (Average Operation Size)

В качестве индикатора размера может использоваться количество строк программы, однако LOC-оценки приводят к известным проблемам. Альтернативный вариант — *«количество сообщений, посланных операцией».*

Рост значения метрики означает, что обязанности размещены в классе не очень удачно. Рекомендуемое значение $OS_{AVG} \leq 9$.

Метрика 6: Сложность операции OC (Operation Complexity

Сложность операции может вычисляться с помощью стандартных метрик сложности, то есть с помощью LOC- или FP-оценок, метрики цикломатической сложности, метрики Холстеда.

М. Лоренц и Д. Кидд предлагают вычислять OC суммированием оценок с весовыми коэффициентами, приведенными в табл. 14.5.

Таблица 14.5. Весовые коэффициенты для метрики OC

Параметр	Вес
Вызовы функций API	5,0
Присваивания	0,5
Арифметические операции	2,0
Сообщения с параметрами	3,0
Вложенные выражения	0,5
Параметры	0,3
Простые вызовы	7,0
Временные переменные	0,5
Сообщения без параметров	1,0

Поскольку операция должна быть ограничена конкретной обязанностью, желательно уменьшать OC.

Рекомендуемое значение OC $\leq$ 65 (для предложенного суммирования).

Метрика 7: Среднее количество параметров на операцию NP_{AVG} (Average Number of Parameters per operation)

Чем больше параметров у операции, тем сложнее сотрудничество между объектами. Поэтому значение NP_{AVG} должно быть как можно меньшим.

Рекомендуемое значение NP_{AVG} = 0,7.

Метрики для ОО-проектов

Основными задачами менеджера проекта являются планирование, координация, отслеживание работ и управление программным проектом.

Одним из ключевых вопросов планирования является оценка размера программного продукта. Прогноз размера продукта обеспечивают следующие ОО-метрики.

Метрика 8: Количество описаний сценариев NSS (Number of Scenario Scripts)

Это количество прямо пропорционально количеству классов, требуемых для реализации требований, количеству состояний для каждого класса, а также количеству методов, свойств и сотрудничеств. Метрика NSS — эффективный индикатор размера программы.

Рекомендуемое значение NSS — не менее одного сценария на публичный протокол подсистемы, отражающий основные функциональные требования к подсистеме.

Метрика 9: Количество ключевых классов NKC (Number of Key Classes)

Ключевой класс прямо связан с коммерческой проблемной областью, для которой предназначена система. Маловероятно, что ключевой класс может появиться в результате повторного использования существующего класса. Поэтому значение NKC достоверно отражает предстоящий объем разработки. М. Лоренц и Д. Кидд предполагают, что в типовой ОО-системе на долю ключевых классов приходится 20-40% от общего количества классов. Как правило, оставшиеся классы реализуют общую инфраструктуру (GUI, коммуникации, базы данных).

Рекомендуемое значение: если NKC < 0,2 от общего количества классов системы, следует углубить исследование проблемной области (для обнаружения важнейших абстракций, которые нужно реализовать).

Метрика 10: Количество подсистем NSUB (NumberofSUBsystem)

Количество подсистем обеспечивает понимание следующих вопросов: размещение ресурсов, планирование (с акцентом на параллельную разработку), общие затраты на интеграцию.

Рекомендуемое значение: NSUB > 3.

Значения метрик NSS, NKC, NSUB полезно накапливать как результат каждого выполненного ОО-проекта. Так формируется метрический базис фирмы, в который также включаются метрические значения по классами и операциям. Эти исторические данные могут использоваться для вычисления метрик производительности (среднее количество классов на разработчика или среднее количество методов на человеко-месяц). Совместное применение метрик позволяет оценивать затраты, продолжительность, персонал и другие характеристики текущего проекта.

Набор метрик Фернандо Абреу

Набор метрик *MOOD* (Metrics for Object Oriented Design), предложенный Ф. Абреу в 1994 году, — другой пример академического подхода к оценке качества ОО-проектирования [6]. Основными целями MOOD-набора являются:

1) покрытие базовых механизмов объектно-ориентированной парадигмы, таких как инкапсуляция, наследование, полиморфизм, посылка сообщений;
2) формальное определение метрик, позволяющее избежать субъективности измерения;
3) независимость от размера оцениваемого программного продукта;
4) независимость от языка программирования, на котором написан оцениваемый продукт.

Набор MOOD включает в себя следующие метрики:

1) фактор закрытости метода (MHF);
2) фактор закрытости свойства (AHF);
3) фактор наследования метода (MIF);
4) фактор наследования свойства (AIF);
5) фактор полиморфизма (POF);
6) фактор сцепления (COF).

Каждая из этих метрик относится к основному механизму объектно-ориентированной парадигмы: инкапсуляции (MHF и AHF), наследованию (MIF и AIF), полиморфизму (POF) и посылке сообщений (COF). В определениях MOOD не используются специфические конструкции языков программирования.

Метрика 1: Фактор закрытости метода MHF (Method Hiding Factor)

Введем обозначения:

- $M_v(C_i)$ — количество видимых методов в классе C_i (интерфейс класса);
- $M_h(C_i)$ — количество скрытых методов в классе C_i (реализация класса);
- $M_d(C_i) = M_v(C_i) + M_h(C_i)$ — общее количество методов, определенных в классе C, (унаследованные методы не учитываются).

Тогда формула метрики MHF примет вид:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)},$$

где TC — количество классов в системе.

Если видимость *m-го* метода *i*-го класса из *j*-го класса вычислять по выражению:

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1, \text{if} \begin{cases} j \neq 1 \\ C_j \text{ может вызвать } M_{mi} \end{cases} \\ 0, \text{else} \end{cases},$$

а процентное количество классов, которые видят *m*-й метод *i*-го класса, определять по соотношению:

$$V(M_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

то формулу метрики MHF можно представить в виде:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}.$$

В числителе этой формулы MHF — сумма закрытости всех методов во всех классах. Закрытость метода — процентное количество классов, из которых данный метод невидим. Знаменатель MHF — общее количество методов, определенных в рассматриваемой системе.

С увеличением MHF уменьшаются плотность дефектов в системе и затраты на их устранение. Обычно разработка класса представляет собой пошаговый процесс, при котором к классу добавляется все больше и больше деталей (скрытых методов). Такая схема разработки способствует возрастанию как значения MHF, так и качества класса.

Метрика 2: Фактор закрытости свойства AHF (Attribute Hiding Factor)

Введем обозначения:

- $A_v(C_i)$ — количество видимых свойств в классе C_i (интерфейс класса);
- $A_h(C_i)$ — количество скрытых свойств в классе C_i (реализация класса);
- $A_d(C_i) = A_v(C_i) + A_h(C_i)$ — общее количество свойств, определенных в классе C_i (унаследованные свойства не учитываются).

Тогда формула метрики *AHF* примет вид:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)},$$

где TC — количество классов в системе.

Если видимость *m*-го свойства *i*-го класса из *j*-го класса вычислять по выражению:

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1, \text{if} \begin{cases} j \neq 1 \\ C_j \text{ может вызвать } A_{mi} \end{cases} \\ 0, \text{else} \end{cases},$$

а процентное количество классов, которые видят *m*-е свойство *i*-го класса, определять по соотношению:

$$V(A_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1},$$

то формулу метрики *AHF*можно представить в виде:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}.$$

В числителе этой формулы AHF — сумма закрытости всех свойств во всех классах. Закрытость свойства — процентное количество классов, из которых данное свойство невидимо. Знаменатель AHF — общее количество свойств, определенных в рассматриваемой системе.

В идеальном случае все свойства должны быть скрыты и доступны только для методов соответствующего класса (AHF = 100%).

Метрика 3: Фактор наследования метода MIF (Method Inheritance Factor)

Введем обозначения:

- $M_i(C_i)$ — количество унаследованных и не переопределенных методов в классе C_i;
- $M_0(C_i)$ — количество унаследованных и переопределенных методов в классе C_i;
- $M_n(C_i)$ — количество новых (не унаследованных и переопределенных) методов в классе C_i;
- $M_d(C_i) = M_n(C_i) + M_0(C_i)$ — количество методов, определенных в классе C_i;
- $M_a(C_i) = M_d(C_i) + M_i(C_i)$ — общее количество методов, доступных в классе C_i.

Тогда формула метрики MIF примет вид:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_i(C_i)}.$$

Числителем MIF является сумма унаследованных (и не переопределенных) методов во всех классах рассматриваемой системы. Знаменатель MIF — это общее количество доступных методов (локально определенных и унаследованных) для всех классов.

Значение MIF = 0 указывает, что в системе отсутствует эффективное наследование, например, все унаследованные методы переопределены.

С увеличением MIF уменьшаются плотность дефектов и затраты на исправление ошибок. Очень большие значения MIF (70-80%) приводят к обратному эффекту, но этот факт нуждается в дополнительной экспериментальной проверке. Сформулируем «осторожный» вывод: умеренное использование наследования — подходящее средство для снижения плотности дефектов и затрат на доработку.

Метрика 4: Фактор наследования свойства AIF (Attribute Inheritance Factor)

Введем обозначения:

- $A_i(C_i)$ — количество унаследованных и не переопределенных свойств в классе C_i;
- $A_0(C_i)$ — количество унаследованных и переопределенных свойств в классе C_i;
- $A_n(C_i)$ — количество новых (не унаследованных и переопределенных) свойств в классе C_i;
- $A_d(C_i) = A_n(C_i) + A_0(C_i)$ — количество свойств, определенных в классе C_i;

- $A_a(C_i) = A_d(C_i) + A_i(C_i)$ — общее количество свойств, доступных в классе C_i.

Тогда формула метрики AIF примет вид:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}.$$

Числителем AIF является сумма унаследованных (и не переопределенных) свойств во всех классах рассматриваемой системы. Знаменатель AIF — это общее количество доступных свойств (локально определенных и унаследованных) для всех классов.

Метрика 5: Фактор полиморфизма POF (Polymorphism Factor)

Введем обозначения:

- $M_0(C_i)$ — количество унаследованных и переопределенных методов в классе C_i;
- $M_n(C_i)$ — количество новых (не унаследованных и переопределенных) методов в классе C_i;
- $DC(C_i)$ — количество потомков класса C_i;
- $M_d(C_i) = M_n(C_i) + M_0(C_i)$ — количество методов, определенных в классе C_i.

Тогда формула метрики POF примет вид:

$$POF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}.$$

Числитель POF фиксирует реальное количество возможных полиморфных ситуаций. Очевидно, что сообщение, посланное в класс C_i связывается (статически или динамически) с реализацией именуемого метода. Этот метод, в свою очередь, может или представляться несколькими «формами», или переопределяться (в потомках C_i).

Знаменатель POF представляет максимальное количество возможных полиморфных ситуаций для класса C_i. Имеется в виду случай, когда все новые методы, определенные в C_i, переопределяются во всех его потомках.

Умеренное использование полиморфизма уменьшает как плотность дефектов, так и затраты на доработку. Однако при POF > 10% возможен обратный эффект.

Метрика 6: Фактор сцепления COF (Coupling Factor)

В данном наборе сцепление фиксирует наличие между классами отношения «клиент-поставщик» (client-supplier). Отношение «клиент-поставщик» (C_c =>C_s) здесь означает, что класс-клиент содержит но меньшей мере одну не унаследованную ссылку на свойство или метод класса-поставщика.

$$is\_client(C_c, C_s) = \begin{cases} 1, \text{if } C_c => C_s \cap C_c \neq C_s, \\ 0, \text{else}, \end{cases}$$

Если наличие отношения «клиент-поставщик» определять по выражению:

$$COF = \frac{\sum_{i=1}^{TC} [\sum_{j=1}^{TC} is\_client(C_i, C_j)]}{TC^2 - TC}.$$

то формула для вычисления метрики COF примет вид:

Знаменатель COF соответствует максимально возможному количеству сцеплений в системе с TC-классами (потенциально каждый класс может быть поставщиком для других классов). Из рассмотрения исключены рефлексивные отношения — когда класс является собственным поставщиком. Числитель COF фиксирует реальное количество сцеплений, не относящихся к наследованию.

С увеличением сцепления классов плотности дефектов и затрат на доработку также возрастают. Сцепления отрицательно влияют на качество ПО, их нужно сводить к минимуму. Практическое применение этой метрики доказывает, что сцепление увеличивает сложность, уменьшает инкапсуляцию и возможности повторного использования, затрудняет понимание и усложняет сопровождение ПО.

Метрики для объектно-ориентированного тестирования

Рассмотрим проектные метрики, которые, по мнению Р. Байндера (Binder), прямо влияют на тестируемость ОО-систем [17]. Р. Байндер сгруппировал эти метрики в три категории, отражающие важнейшие проектные характеристики.

Метрики инкапсуляции

К метрикам инкапсуляции относятся: «Недостаток связности в методах LCOM», «Процент публичных и защищенных PAP (Percent Public and Protected)» и «Публичный доступ к компонентным данным PAD (Public Access to Data members)».

Метрика 1: Недостаток связности в методах LCOM

Чем выше значение LCOM, тем больше состояний надо тестировать, чтобы гарантировать отсутствие побочных эффектов при работе методов.

Метрика 2: Процент публичных и защищенных PAP (Percent Public and Protected)

Публичные свойства наследуются от других классов и поэтому видимы для этих классов. Защищенные свойства являются специализацией и приватны для определенного подкласса. Эта метрика показывает процент публичных свойств класса. Высокие значения *PAP* увеличивают вероятность побочных эффектов в классах. Тесты должны гарантировать обнаружение побочных эффектов.

Метрика 3: Публичный доступ к компонентным данным PAD (Public Access to Data members)

Метрика показывает количество классов (или методов), которые имеют доступ к свойствам других классов, то есть нарушают их инкапсуляцию. Высокие значения приводят к возникновению побочных эффектов в классах. Тесты должны гарантировать обнаружение таких побочных эффектов.

Метрики наследования

К метрикам наследования относятся «Количество корневых классов NOR (Number Of Root classes)», «Коэффициент объединения по входу FIN», «Количество детей NOC» и «Высота дерева наследования DIT».

Метрика 4: Количество корневых классов NOR (Number Of Root classes)

Эта метрика подсчитывает количество деревьев наследования в проектной модели. Для каждого корневого класса и дерева наследования должен разрабатываться набор тестов. С увеличением NOR возрастают затраты на тестирование.

Метрика 5: Коэффициент объединения по входу FIN

В контексте О-О-смистем FIN фиксирует множественное наследование. Значение FIN > 1 указывает, что класс наследует свои свойства и операции от нескольких корневых классов. Следует избегать FIN > 1 везде, где это возможно.

Метрика 6: Количество детей NOC

Название говорит само за себя. Метрика заимствована из набора Чидамбера-Кемерера.

Метрика 7: Высота дерева наследования DIT

Метрика заимствована из набора Чидамбера-Кемерера. Методы суперкласса должны повторно тестироваться для каждого подкласса.

В дополнение к перечисленным метрикам Р. Байндер выделил метрики сложности класса (это метрики Чидамбера-Кемерера — WMC, CBO, RFC и метрики для подсчета количества методов), а также метрики полиморфизма.

Метрики полиморфизма

Рассмотрим следующие метрики полиморфизма: «Процентное количество не переопределенных запросов OVR», «Процентное количество динамических запросов DYN», «Скачок класса Bounce-C» и «Скачок системы Bounce-S».

Метрика 8: Процентное количество не переопределенных запросов OVR

Процентное количество от всех запросов в тестируемой системе, которые не приводили к перекрытию модулей. Перекрытие может приводить к непредусмотренному связыванию. Высокое значение OVR увеличивает возможности возникновения ошибок.

Метрика 9: Процентное количество динамических запросов DYN

Процентное количество от всех сообщений в тестируемой системе, чьи приемники определяются в период выполнения. Динамическое связывание может приводить к непредусмотренному связыванию. Высокое значение DYN означает, что для проверки всех вариантов связывания метода потребуется много тестов.

Метрика 10: Скачок класса Bounce-C

Количество скачущих маршрутов, видимых тестируемому классу. Скачущий маршрут — это маршрут, который в ходе динамического связывания пересекает несколько иерархий классов-поставщиков. Скачок может приводить к непредусмотренному связыванию. Высокое значение Bounce-C увеличивает возможности возникновения ошибок.

Управление риском

Словарь русского языка С. И. Ожегова и Н. Ю. Шведовой определяет риск как *«возможность опасности, неудачи»*. Влияние риска вычисляют по выражению

$$RE = P(UO) \times L(UO),$$

где:

- RE — показатель риска (Risk Exposure — подверженность риску);
- P(UO) — вероятность неудовлетворительного результата (Unsatisfactory Outcome);
- L(UO) — потеря при неудовлетворительном результате.

При разработке программного продукта неудовлетворительным результатом может быть: превышение бюджета, низкая надежность, неправильное функционирование и т. д. Управление риском включает шесть действий:

1. Идентификация риска — выявление элементов риска в проекте.
2. Анализ риска — оценка вероятности и величины потери по каждому элементу риска.
3. Ранжирование риска — упорядочение элементов риска по степени их влияния.
4. Планирование управления риском — подготовка к работе с каждым элементом риска.
5. Разрешение риска — устранение или разрешение элементов риска.
6. Наблюдение риска — отслеживание динамики элементов риска, выполнение корректирующих действий.

Первые три действия относят к этапу оценивания риска, последние три действия — к этапу контроля риска [20].

Идентификация риска

В результате идентификации формируется список элементов риска, специфичных для данного проекта.

Выделяют три категории источников риска: проектный риск, технический риск, коммерческий риск.

Источниками проектного риска являются:

- выбор бюджета, плана, человеческих ресурсов программного проекта;
- формирование требований к программному продукту;
- сложность, размер и структура программного проекта;
- методика взаимодействия с заказчиком.

К источникам технического риска относят:

- трудности проектирования, реализации, формирования интерфейса, тестирования и сопровождения;
- неточность спецификаций;
- техническая неопределенность или отсталость принятого решения.

Главная причина технического риска — реальная сложность проблем выше предполагаемой сложности.

Источники коммерческого риска включают:

- создание продукта, не требующегося на рынке;
- создание продукта, опережающего требования рынка (отстающего от них);
- потерю финансирования.

Лучший способ идентификации — использование проверочных списков риска, которые помогают выявить возможный риск. Например, проверочный список десяти главных элементов программного риска может иметь представленный ниже вид.

1. Дефицит персонала.
2. Нереальные расписание и бюджет.
3. Разработка неправильных функций и характеристик.
4. Разработка неправильного пользовательского интерфейса.
5. Слишком дорогое обрамление.
6. Интенсивный поток изменения требований.
7. Дефицит поставляемых компонентов.
8. Недостатки в задачах, разрабатываемых смежниками.
9. Дефицит производительности при работе в реальном времени.
10. Деформирование научных возможностей.

На практике каждый элемент списка снабжается комментарием — набором методик для предотвращения источника риска.

После идентификации элементов риска следует количественно оценить их влияние на программный проект, решить вопросы о возможных потерях. Эти вопросы решаются на шаге анализа риска.

Анализ риска

В ходе анализа оценивается вероятность возникновения P_i и величина потери L_i для каждого выявленного i-го элемента риска. В результате вычисляется влияние RE_i i-го элемента риска на проект.

Вероятности определяются с помощью экспертных оценок или на основе статистики, накопленной за предыдущие разработки. Итоги анализа, как показано в табл. 15.1, сводятся в таблицу.

Таблица 15.1. Оценка влияния элементов риска

Элемент риска	Вероятность, %	Потери	Влияние риска
1. Критическая программная ошибка	3-5	10	30-50

2. Ошибка потери ключевых данных	3-5	8	24-40
3. Отказоустойчивость недопустимо снижает производительность	4-8	7	28-56
4. Отслеживание опасного условия как безопасного	5	9	45
5. Отслеживание безопасного условия как опасного	5	3	15
6. Аппаратные задержки срывают планирование	6	4	24
7. Ошибки преобразования данных приводят к избыточным вычислениям	8	1	8
8. Слабый интерфейс пользователя снижает эффективность работы	6	5	30
9. Дефицит процессорной памяти	1	7	7
10. СУБД теряет данные	2	2	4

Ранжирование риска

Ранжирование заключается в назначении каждому элементу риска приоритета, который пропорционален влиянию элемента на проект. Это позволяет выделить категории элементов риска и определить наиболее важные из них. Например, представленные в табл. 15.1 элементы риска упорядочены по их приоритету.

Для больших проектов количество элементов риска может быть очень велико (30-40 элементов). В этом случае управление риском затруднено. Поэтому к элементам риска применяют принцип Парето 80/20. Опыт показывает, что 80% всего проектного риска приходятся на долю 20% от общего количества элементов риска. В ходе ранжирования определяют эти 20% элементов риска (их называют существенными элементами). В дальнейшем учитывается влияние только существенных элементов риска.

Планирование управления риском

Цель планирования — сформировать набор функций управления каждым элементом риска. Введем необходимые определения.

В планировании используют понятие эталонного уровня риска. Обычно выбирают три эталонных уровня риска: превышение стоимости, срыв планирования, упадок производительности. Они могут быть причиной прекращения проекта. Если комбинация проблем, создающих риск, станет причиной превышения любого из этих уровней, работа будет остановлена. В фазовом пространстве риска эталонному уровню риска соответствует эталонная точка. В эталонной точке решения « продолжать проект» и «прекратить проект» имеют одинаковую силу. На рис. 15.3 показана кривая останова, составленная из эталонных точек.

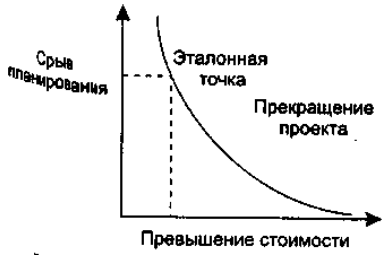


Рис. 15.3. Кривая останова проекта

Ниже кривой располагается рабочая область проекта, выше кривой — запретная область (при попадании в эту область проект должен быть прекращен).

Реально эталонный уровень редко представляется как кривая, чаще это сфера, в которой есть области неопределенности (в них принять решение невозможно).

Теперь рассмотрим последовательность шагов планирования.

1. Исходными данными для планирования является набор четверок $[R_i\ P_i,\ L_i,\ RE_i]$, где R_i — 2-й элемент риска, P_i — вероятность i-го элемента риска, L_i — потеря по i-му элементу риска, RE_i — влияние i-го элемента риска.
2. Определяются эталонные уровни риска в проекте.
3. Разрабатываются зависимости между каждой четверкой $[R_i\ P_i,\ L_i,\ RE_i]$ и каждым эталонным уровнем.
4. Формируется набор эталонных точек, образующих сферу останова. В сфере останова предсказываются области неопределенности.
5. Для каждого элемента риска разрабатывается план управления. Предложения плана составляются в виде ответов на вопросы «зачем, что, когда, кто, где, как и сколько».
6. План управления каждым элементом риска интегрируется в общий план программного проекта.

Разрешение и наблюдение риска

Основанием для разрешения и наблюдения является план управления риском. Работы по разрешению и наблюдению производятся с начала и до конца процесса разработки.

Разрешение риска состоит в плановом применении действий по уменьшению риска.

Наблюдение риска гарантирует:

- цикличность процесса слежения за риском;
- вызов необходимых корректирующих воздействий.

Для управления риском используется эффективная методика *«Отслеживание 10 верхних элементов риска»*. Эта методика концентрирует внимание на факторах повышенного риска, экономит много времени, минимизирует «сюрпризы» разработки.

Рассмотрим шаги методики *«Отслеживания 10 верхних элементов риска»*.

1. Выполняется выделение и ранжирование наиболее существенных элементов риска в проекте.
2. Производится планирование регулярных просмотров (проверок) процесса разработки. В больших проектах (в группе больше 20 человек) просмотр должен проводиться ежемесячно, в остальных проектах — чаще.
3. Каждый просмотр начинается с обсуждения изменений в 10 верхних элементах риска (их количество может изменяться от 7 до 12). В обсуждении фиксируется текущий приоритет каждого из 10 верхних элементов риска, его приоритет в предыдущем просмотре, частота попадания элемента в список верхних элементов. Если элемент в списке опустился, он по-прежнему нуждается в наблюдении, но не требует управляющего воздействия. Если элемент поднялся в списке, или только появился в нем, то элемент требует повышенного внимания. Кроме того, в обзоре обсуждается прогресс в разрешении элемента риска (по сравнению с предыдущим просмотром).
4. Внимание участников просмотра концентрируется на любых проблемах в разрешении элементов риска.

Этапы унифицированного процесса разработки

Обсудим назначение, цели, содержание и основные итоги каждого этапа унифицированного процесса разработки.

Этап НАЧАЛО (Inception)

Главное назначение этапа — запустить проект.

Цели этапа НАЧАЛО:

- определить область применения проектируемой системы (ее предназначение, границы, интерфейсы с внешней средой, критерий признания — приемки);
- определить элементы Use Case, критические для системы (основные сценарии поведения, задающие ее функциональность и покрывающие главные проектные решения);
- определить общие черты архитектуры, обеспечивающей основные сценарии, создать демонстрационный макет;
- определить общую стоимость и план всего проекта и обеспечить детализированные оценки для этапа развития;
- идентифицировать основные элементы риска. Основные действия этапа НАЧАЛО:
- формулировка области применения проекта — выявление требований и ограничений, рассматриваемых как критерий признания конечного продукта;
- планирование и подготовка бизнес-варианта и альтернатив развития для управления риском, определение персонала, проектного плана, а также выявление зависимостей между стоимостью, планированием и полезностью;
- синтезирование предварительной архитектуры, развитие компромиссных решений проектирования; определение решений разработки, покупки и повторного использования, для которых можно оценить стоимость, планирование и ресурсы.

В итоге этапа НАЧАЛО создаются следующие артефакты:

- спецификация представления основных проектных требований, ключевых характеристик и главных ограничений;
- начальная модель Use Case (20% от полного представления); а начальный словарь проекта;
- начальный бизнес-вариант (содержание бизнеса, критерий успеха — прогноз дохода, прогноз рынка, финансовый прогноз);
- начальное оценивание риска;
- проектный план, в котором показаны этапы и итерации.

Этап РАЗВИТИЕ (Elaboration)

Главное назначение этапа — создать архитектурный базис системы.

Цели этапа РАЗВИТИЕ:

- определить оставшиеся требования, функциональные требования формулировать как элементы Use Case;
- определить архитектурную платформу системы;
- отслеживать риск, устранить источники наибольшего риска;
- разработать план итераций этапа КОНСТРУИРОВАНИЕ.

Основные действия этапа РАЗВИТИЕ:

- развитие спецификации представления, полное формирование критических элементов Use Case, задающих дальнейшие решения;
- развитие архитектуры, выделение ее компонентов.

В итоге этапа РАЗВИТИЕ создаются следующие артефакты:

- модель Use Case (80% от полного представления);
- дополнительные требования (нефункциональные требования, а также другие требования, которые не связаны с конкретным элементом Use Case);
- описание программной архитектуры;
- выполняемый архитектурный макет;
- пересмотренный список элементов риска и пересмотренный бизнес-вариант;
- план разработки для всего проекта, включающий крупноблочный проектный план и показывающий итерации и критерий эволюции для каждой итерации.

Обсудим более подробно главную цель этапа РАЗВИТИЕ — создание архитектурного базиса.

Архитектура объектно-ориентированной системы многомерна — она описывается множеством параллельных представлений. Как показано на рис. 15.4, обычно используется «4+1»-представление [44].

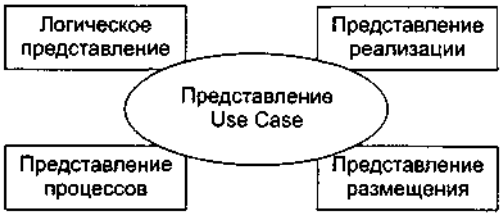


Рис. 15.4. «4+1»-представление архитектуры

Представление Use Case описывает систему как множество взаимодействий с точки зрения внешних актеров. Это представление создается на этапе НАЧАЛО жизненного цикла и управляет оставшейся частью процесса разработки.

Логическое представление содержит набор пакетов, классов и отношений. Изначально создается на этапе развития и усовершенствуется на этапе конструирования.

Представление процессов создается для параллельных программных систем, содержит процессы, потоки управления, межпроцессорные коммуникации и механизмы синхронизации. Представление изначально создается на этапе развития, усовершенствуется на этапе конструирования.

Представление реализации содержит модули и подсистемы. Представление изначально создается на этапе развития и усовершенствуется на этапе конструирования.

Представление размещения содержит физические узлы системы и соединения между узлами. Создается на этапе развития.

В качестве примера рассмотрим порядок создания логического представления архитектуры. Для решения этой задачи исследуются элементы Use Case, разработанные на этапе НАЧАЛО. Рассматриваются экземпляры элементов Use Case — сценарии. Каждый сценарий преобразуется в диаграмму последовательности. Далее в диаграммах последовательности выделяются объекты. Объекты группируются в классы. Классы могут группироваться в пакеты.

Согласно взаимодействиям между объектами в диаграммах последовательности устанавливаются отношения между классами. Для обеспечения функциональности в классы добавляются свойства (они определяют их структуру) и операторы (они определяют поведение). Для размещения общей структуры и поведения создаются суперклассы.

В качестве другого примера рассмотрим разработку плана итераций для этапа КОНСТРУИРОВАНИЕ. Такой план должен задавать управляемую серию архитектурных реализаций, каждая из которых увеличивает свои функциональные возможности, а конечная — покрывает все требования к полной системе. Главным источником информации являются элементы Use Case и диаграммы последовательности. Будем называть их обобщенно — сценариями. Сценарии группируются так, чтобы обеспечивать реализацию определенной функциональности системы. Кроме того, группировки должны устранять наибольший (в данный момент) риск в проекте.

План итераций включает в себя следующие шаги:

1. Определяются все элементы риска в проекте. Устанавливаются их приоритеты.
2. Выбирается группа сценариев, которым соответствуют элемент риска с наибольшим приоритетом. Сценарии исследуются. Порядок исследования определяется не только степенью риска, но и важностью для заказчика, а также потребностью ранней разработки базовых сценариев.
3. В результате анализа сценариев формируются классы и отношения, которые их реализуют.
4. Программируются сформированные классы и отношения.
5. Разрабатываются тестовые варианты.
6. Тестируются классы и отношения. Цель — проверить выполнение функционального назначения сценария.
7. Результаты объединяются с результатами предыдущих итераций, проводится тестирование интеграции.
8. Оценивается итерация. Выделяется необходимая повторная работа. Она назначается на будущую итерацию.

Этап КОНСТРУИРОВАНИЕ (Construction)

Главное назначение этапа — создать программный продукт, который обеспечивает начальные операционные возможности.

Цели этапа КОНСТРУИРОВАНИЕ:

- минимизировать стоимость разработки путем оптимизации ресурсов и устранения необходимости доработок;
- добиться быстрого получения приемлемого качества;
- добиться быстрого получения контрольных версий (альфа, бета и т. д.).

Основные действия этапа КОНСТРУИРОВАНИЕ:

- управление ресурсами, контроль ресурсов, оптимизация процессов;
- полная разработка компонентов и их тестирование (по сформулированному критерию эволюции);
- оценивание реализаций продукта (по критерию признания из спецификации представления).

В итоге этапа КОНСТРУИРОВАНИЕ создаются следующие артефакты:

- программный продукт, готовый для передачи в руки конечных пользователей;
- описание текущей реализации;
- руководство пользователя.

Реализации продукта создаются в серии итераций. Каждая итерация выделяет конкретный набор элементов риска, выявленных на этапе развития. Обычно в итерации реализуется один или несколько элементов Use Case. Типовая итерация включает следующие действия:

1. Идентификация реализуемых классов и отношений.
2. Определение в классах типов данных (для свойств) и сигнатур (для операций). Добавление сервисных операций, например операций доступа и управления. Добавление сервисных классов (классов-контейнеров, классов-контроллеров). Реализация отношений ассоциации, агрегации и наследования.
3. Создание текста на языке программирования.
4. Создание(обновление) документации.
5. Тестирование функций реализации продукта.
6. Объединение текущей и предыдущей реализаций. Тестирование итерации.

Этап ПЕРЕХОД (Transition)

Главное назначение этапа — применить программный продукт в среде пользователей и завершить реализацию продукта.

Этап начинается с предъявления пользователям бета-реализации продукта. В ней обнаруживаются ошибки, они корректируются в последующих бета-реализациях. Параллельно решаются вопросы размещения, упаковки и сопровождения продукта. После завершения бета-периода тестирования продукт считается реализованным.

Оценка качества проектирования

Качество проектирования оценивают с помощью объектно-ориентированных метрик, введенных в главе 14.

Этап РАЗВИТИЕ

Качество логического представления архитектуры оценивают по метрикам:

- WMC — взвешенные методы на класс;
- NOC — количество детей;
- DIT — высота дерева наследования;
- NOM — суммарное количество методов, определенных во всех классах системы;
- NC — общее количество классов в системе.

Метрики WMC, NOC вычисляются для каждого класса, кроме того, формируются их средние значения в системе. Метрики DIT, NOM, NC вычисляются для всей системы.

Этап КОНСТРУИРОВАНИЕ

На каждой итерации конструирования продукта вычисляются метрики:

- WMC — взвешенные методы на класс;
- NOC — количество детей;
- CBO — сцепление между классами объектов;
- RFC — отклик для класса;
- LCOM — недостаток связности в методах;
- CS — размер класса;
- NOO — количество операций, переопределяемых подклассом;
- NOA — количество операций, добавленных подклассом;
- SI — индекс специализации;
- OS_{avg} — средний размер операции;
- NP_{avg} — среднее количество параметров на операцию;
- NC — общее количество классов в системе;
- $LOC_{\sum}$ — суммарная LOC-оценка всех методов системы;
- DIT — высота дерева наследования;
- NOM — суммарное количество методов в системе.

Метрики WMC, NOC, CBO, RFC, LCOM, CS, NOO, NOA, SI, OS_{AVG}, NP_{AVG} вычисляются для каждого класса, кроме того, формируются их средние значения в системе. Метрики DIT, NOM, NC, LOC_S вычисляются для всей системы.

На последней итерации дополнительно вычисляется набор метрик MOOD, предложенный Абреу:

- MHF — фактор закрытости метода;
- AHF — фактор закрытости свойства;
- MIF — фактор наследования метода;
- AIF — фактор наследования свойства;
- POF — фактор полиморфизма;
- COF — фактор сцепления.

Пример объектно-ориентированной разработки

Для иллюстрации унифицированного процесса рассмотрим фрагмент разработки, выполненной автором совместно с Ольвией Комашиловой. Поставим задачу — разработать оконный интерфейс пользователя, который будет использоваться прикладными программами.

Этап НАЧАЛО

Оконный интерфейс пользователя(WUI) — среда, управляемая событиями. Действия в среде инициируются функциями обратного вызова, которые вызываются в ответ на событие — пользовательский ввод. Ядром WUI является цикл обработки событий, который организуется менеджером ввода.

WUI должен обеспечивать следующие типы неперекрывающихся окон:

- простое окно, в которое может быть выведен текст;
- окно меню, в котором пользователь может задать вариант действий — выбор подменю или функции обратного вызова.

Идентификация актеров

Актерами для *WUI* являются:

- пользователь прикладной программы, использующей *WUI;*
- администратор системы, управляющий работой *WUI.*

Внешнее окружение *WUI* имеет вид, представленный на рис. 15.5.

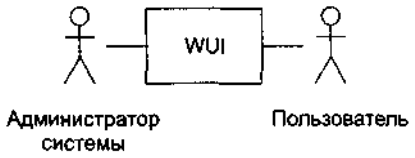


Рис. 15.5. Внешнее окружение WUI

Идентификация элементов Use Case

В WUI могут быть выделены два элемента Use Case:

- управление окнами;
- использование окон.

Диаграмма Use Case для среды WUI представлена на рис. 15.6.

Рис. 15.6. Диаграмма Use Case для среды WUI

Описания элементов Use Case

Описание элемента Use Case Управление окнами.

Действия начинаются администратором системы. Администратор может создать, удалить или модифицировать окно.

Описание элемента Use Case Использование окон.

Действия начинаются пользователем прикладной программы. Обеспечивается возможность работы с меню и простыми окнами.

Этап РАЗВИТИЕ

На этом этапе создаются сценарии для элементов Use Case, разрабатываются диаграммы последовательности (формализующие текстовые представления сценариев), проектируются диаграммы классов и планируется содержание следующего этапа разработки.

Сценарии для элемента Use Case Управление окнами

В элементе Use Case Управление окнами заданы три потока событий — три сценария.

1. Сценарий Создание окна.

Устанавливаются координаты окна на экране, стиль рамки окна. Образ окна сохраняется в памяти. Окно выводится на экран. Если создается окно меню, содержащее обращение к функции обратного вызова, то происходит установка этой функции. В конце менеджер окон добавляет окно в список управляемых окон WUI.

2. Сценарий Изменение стиля рамки.

Указывается символ, с помощью которого будет изображаться рамка. Образ окна сохраняется в памяти. Окно перерисовывается на экране.

3. Сценарий Уничтожение окна.

Менеджер окон получает указание удалить окно. Менеджер окон снимает окно с регистрации (в массиве управляемых окон WUI). Окно снимает отображение с экрана.

Развитие описания элемента Use Case Использование окон

Действия начинаются с ввода пользователем символа. Символ воспринимается менеджером ввода. В зависимости от значения введенного символа выполняется один из следующих вариантов:

при значении ENTER - вариант ОКОНЧАНИЯ ВВОДА;

при переключающем значении - вариант ПЕРЕКЛЮЧЕНИЯ;

при обычном значении - символ выводится в активное окно.

Вариант ОКОНЧАНИЯ ВВОДА:

при активном окне меню выбирается пункт меню. В ответ либо выполняется функция обратного вызова (закрепленная за этим пунктом меню), либо вызывается подменю (соответствующее данному пункту меню);

при активном простом окне выполняется переход на новую строку окна.

Вариант ПЕРЕКЛЮЧЕНИЯ.

При вводе переключающего символа:

ESC - активным становится окно меню;

TAB - активным становится следующее простое окно;

Ctrl-E - все окна закрываются и сеанс работы заканчивается.

Далее из описания элемента Use Case Использование окон выделяются два сценария: Использование простого окна и Использование окна меню.

На следующем шаге сценарии элементов Use Case преобразуются в диаграммы последовательности — за счет этого достигается формализация описаний, требуемая для построения диаграмм классов. Для построения диаграмм последовательности проводится грамматический разбор каждого сценария элемента Use Case: значащие существительные превращаются в объекты, а значащие глаголы — в сообщения, пересылаемые между объектами.

Диаграммы последовательности

Диаграммы изображены на рис. 15.7-15.11.

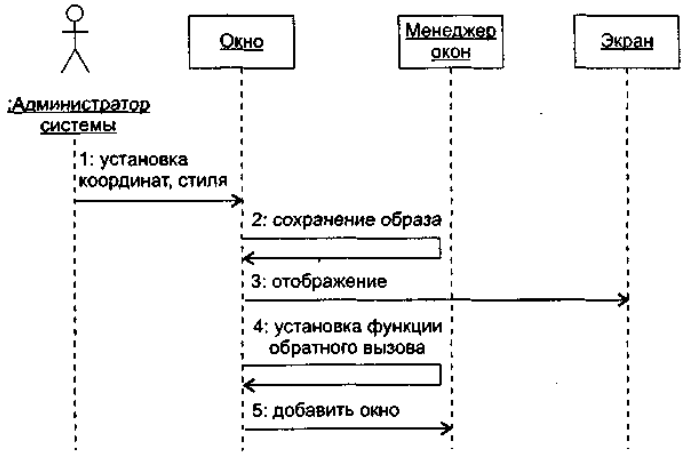


Рис. 15.7. Диаграмма последовательности Создание окна

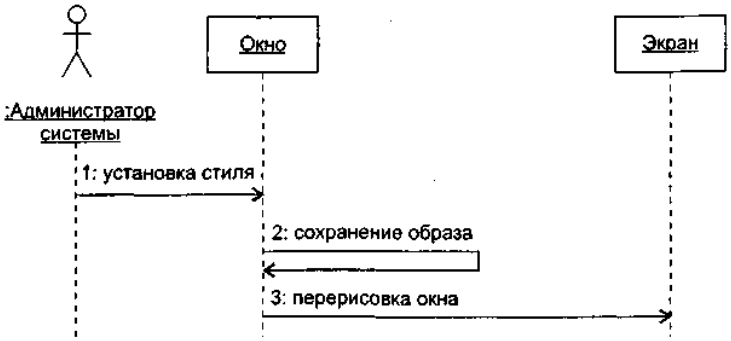
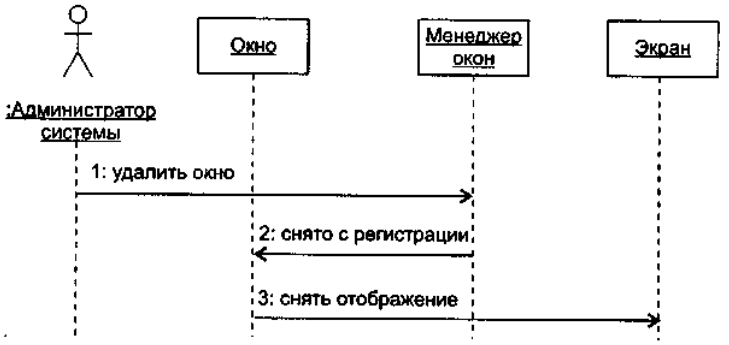


Рис. 15.8. Диаграмма последовательности Изменение стиля рамки



15.9. Диаграмма последовательности Уничтожение окна

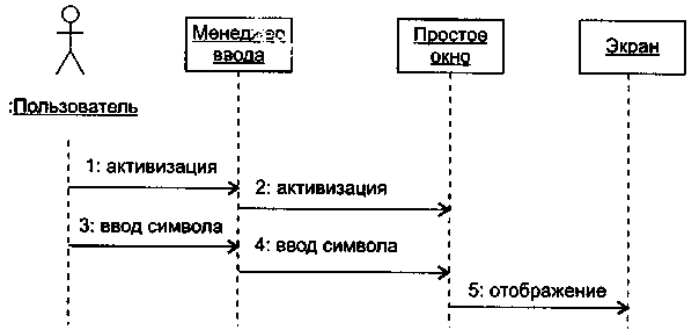


Рис. 15.10. Диаграмма последовательности Использование простого окна

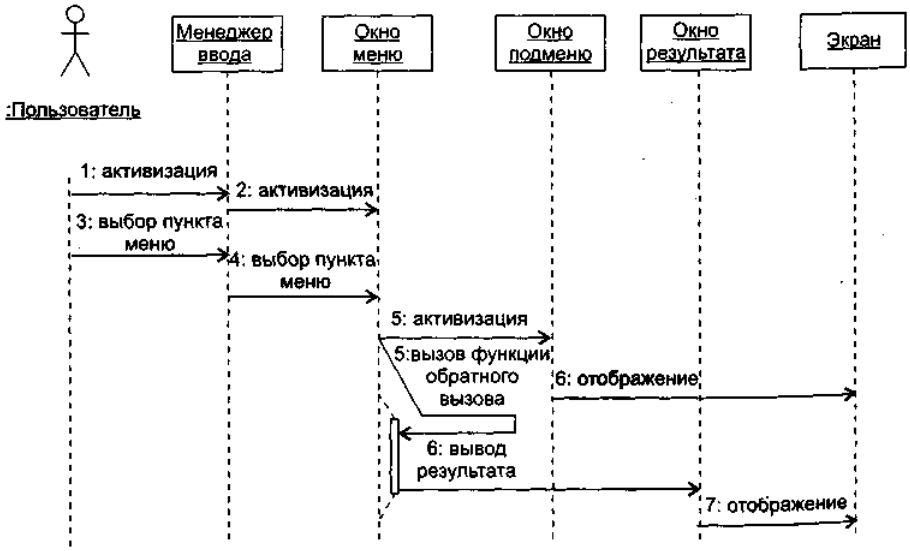


Рис. 15.11. Диаграмма последовательности Использование окна меню

Создание классов

Работа по созданию классов (и включению их в диаграмму классов) требует изучения содержания всех диаграмм последовательности. Проводится она в три этапа.

На первом этапе выявляются и именуются классы. Для этого просматривается каждая диаграмма последовательности. Любой объект в этой диаграмме должен принадлежать конкретному классу, для которого надо придумать имя. Например, резонно предположить, что объекту Менеджер окон должен соответствовать класс Window_Manager, поэтому класс Window_Manager следует ввести в диаграмму. Конечно, если в другой диаграмме последовательности опять появится подобный объект, то дополнительный класс не образуется.

На втором этапе выявляются операции классов. На диаграмме последовательности такая операция соответствует стрелке (и имени) сообщения, указывающей на линию жизни объекта класса. Например, если к линии жизни объекта Менеджер окон подходит стрелка сообщения добавить окно, то в класс Window_Manager нужно ввести си операцию add_to_list().

На третьем этапе определяются отношения ассоциации между классами — они обеспечивают пересылки сообщений между соответствующими объектами.

В нашем примере анализ диаграмм последовательности позволяет выделить следующие классы:

- Window — класс, объектами которого являются простые окна;
- Menu — класс, объектами которого являются окна меню. Этот класс является потомком класса Window;
- Menu_title — класс, объектом которого является окно главного меню. Класс является потомком класса Menu;
- Screen — класс, объектом которого является экран. Этот класс обеспечивает позиционирование курсора, вывод

изображения на экран дисплея, очистку экрана;

- Input_Manager — объект этого класса управляет взаимодействием между пользователем и окнами интерфейса. Его обязанности: начальные установки среды WUI, запуск цикла обработки событий, закрытие среды WUI;
- Window_Manager — осуществляет общее управление окнами, отображаемыми на экране. Используется менеджером ввода для получения доступа к конкретному окну.

Для оптимизации ресурсов создается абстрактный суперкласс Root_Window. Он определяет минимальные обязанности, которые должен реализовать любой тип окна (а (посылка символа в окно, перевод окна в активное/пассивное состояние, перерисовка окна, возврат информации об окне). Все остальные классы окон являются его потомками.

Для реализации функций, определенных в сценариях, в классы добавляются свойства и операции. По результатам формирования свойств и операций классов обновляется содержание диаграмм последовательности.

Начальное представление иерархии классов WUI показано на рис. 15.12. Результаты начальной оценки качества проекта сведены в табл. 15.2.

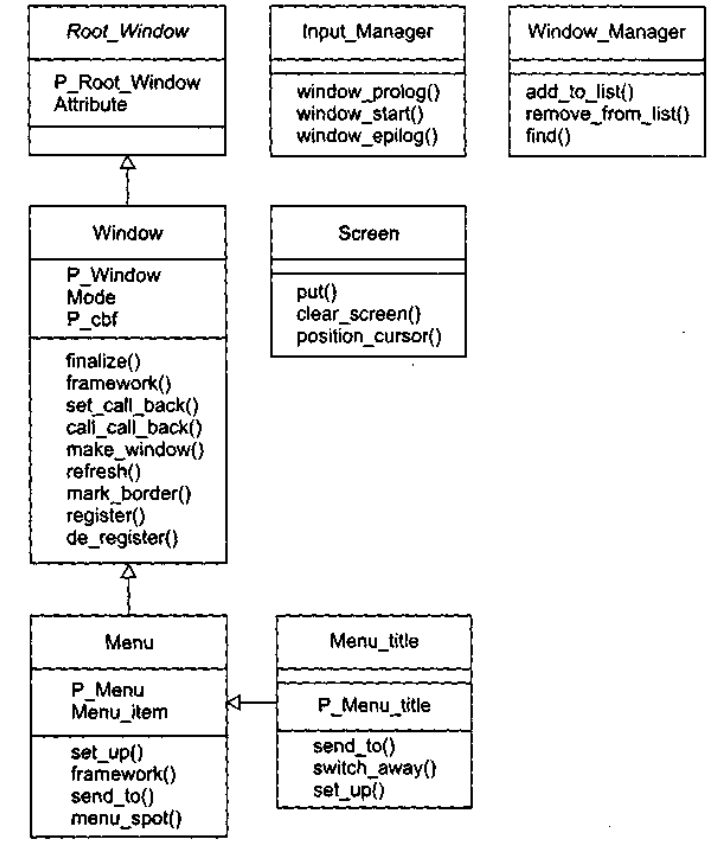


Рис. 15.12. Начальная диаграмма классов WUI

Таблица 15.2. Результаты начальпий оценки качества WUI

Метрика	Input_ Manager	Window_ Manager	Screen	Root_ Window	Window	Menu	Menu_ title	Среднее значение
WMC	3	3	3	0	9	4	3	3,57
NOC	-	-	-	1	1	1	0	0,43
Метрики, вычисляемые для системы								
DIT	3							
NC	7							
NOM	25							

Отметим, что для упрощения рисунка на этой диаграмме не показаны существующие между классами отношения ассоциации. В реальной диаграмме они обязательно отображаются — без них экземпляры классов не смогут взаимодействовать друг с другом.

Планирование итераций конструирования

На данном шаге составляется план итераций, который определяет порядок действий на этапе конструирования. Цель каждой итерации — уменьшить риск разработки конечного продукта. Для создания начального плана анализируются элементы Us Case, их сценарии и диаграммы последовательности. Устанавливается приоритет их реализации. При завершении каждой итерации будет повторно вычисляться риск. Оценка риска может привести к необходимости обновления плана итераций.

Положим, что максимальный риск связан с реализацией элемента Use Case Управление окнами, причем наиболее опасна разработка сценария Создание окна, среднюю опасность несет сценарий Уничтожение окна и малую опасность — Изменение стиля рамки.

В связи с этими соображениями начальный план итераций принимает вид:

Итерация 1 — реализация сценариев элемента Use Case Управление окнами:

1. Создание окна.
2. Уничтожение окна.
3. Изменение стиля рамки.

Итерация 2 — реализация сценариев элемента Use Case Использование окон:

4. Использование простого окна.
5. Использование окна меню.

Этап КОНСТРУИРОВАНИЕ

Рассмотрим содержание итераций на этапе конструирования.

Итерация 1 — реализация сценариев элемента Use Case Управление окнами

Для реализации сценария Создание окна программируются следующие операции класса Window:

- framework — создание каркаса окна;
- register — регистрация окна;
- set_call_back — установка функции обратного вызова;
- make_window — задание видимости окна.

Далее реализуются операции общего управления окнами, методы класса Window_Manager:

- add_to_list — добавление нового окна в массив управляемых окон;
- find — поиск окна с заданным переключающим символом.

Программируются операции класса Input-Manager:

- window_prolog — инициализация WUI;
- window_start — запуск цикла обработки событий;
- window_epilog — закрытие WUI.

В ходе реализации перечисленных операций выясняется необходимость и программируется содержание вспомогательных операций.

1. В классе Window_Manager:
   - write_to — форматный вывод сообщения в указанное окно;
   - hide_win — удаление окна с экрана;
   - switchAwayFromTop — подготовка окна к переходу в пассивное состояние;
   - switch_to_top — подготовка окна к переходу в активное состояние;
   - window_fatal — формирование донесения об ошибке;
   - top — переключение окна в активное состояние;
   - send_to_top — посылка символа в активное окно.
2. В классе Window:
   - put — три реализации для записи в окно символьной, строковой и числовой информации;
   - create — создание макета окна (используется операцией framework);
   - position — изменение позиции курсора в окне;
   - about — возврат информации об окне;
   - switch_to — пометка активного окна;
   - switch_away — пометка пассивного окна;
   - send_to — посылка символа в окно для обработки.

Второй шаг первой итерации ориентирован на реализацию сценария Уничтожение окна. Основная операция — finalize (метод класса Window), она выполняет разрушение окна. Для ее обеспечения создаются вспомогательные операции:

- de_register — удаление окна из массива управляемых окон;
- remove_from_list (метод класса Window_Manager) — вычеркивание окна из регистра.

Для реализации сценария Изменение стиля рамки создаются операции в классе Window:

- mark_border — построение новой рамки окна;
- refresh — перерисовка окна на экране.

В конце итерации создаются операции класса Screen:

- dear_screen — очистка экрана;
- position_cursor — позиционирование курсора;
- put — вывод на экран дисплея строк, символов и чисел.

Результаты оценки качества первой итерации представлены в табл. 15.3.

Таблица 15.3. Оценки качества *WUI* после первой итерации

Метрика	lnput_ Manager	Window_ Manager	Screen	Root_ Window	Window	Среднее значение
WMC	0,12	0,42	0,11	0	0,83	0,3
NOC	-	-	-	1	0	0,2
CBO	3	3	0	1	2	1,8

RFC	6	11	0	0	23	8
LCOM	3	0	5	0	0	1,6
CS	3/2	10/8	5/1	0/2	18/22	7,2/7
NOO	-	-	-	0	0	0
NOA	-	-	-	0	18	3,6
SI	-	-	-	0	0	0
OS_{AVG}	4	4,2	2,2	0	4,6	3
NP_{AVG}	0	1,3	1	0	2,4	0,9

Метрики, вычисляемые для системы	
DIT	1
NC	5
MOM	35
$LOC_{\sum}$	148

Итерация 2 — реализация сценариев элемента Use Case Использование окон

На этой итерации реализуем методы классов Menu и Menu_title, а также добавим необходимые вспомогательные методы в класс Window.

Отметим, что операции, обеспечивающие сценарий Использование простого окна, в основном уже реализованы (на первой итерации). Осталось запрограммировать следующие операции — методы класса Window:

- call_call_back — вызов функции обратного вызова;
- initialize — управляемая инициализация окна;
- clear — очистка окна с помощью пробелов;
- new_line — перемещение на следующую строку окна.

Для обеспечения сценария Использование окна меню создаются следующие операции.

1. В классе Menu:

- framework — создание каркаса окна-меню;
- send_to — обработка пользовательского ввода в окно-меню;
- menu_spot — выделение выбранного элемента меню;
- set_up — заполнение окна-меню именами элементов;
- get_menu_name — возврат имени выбранного элемента меню;
- get_cur_selected_detaits — возврат указателя на выбранное окно и функцию обратного вызова.

2. В классе Menu_title:

- send_to — выделение новой строки меню или вызов функции обратного вызова;
- switch_away — возврат в базовое окно-меню более высокого уровня;
- set_up — установки окна меню-заголовка.

Результаты оценки качества второй итерации представлены в табл. 15.4.

Таблица 15.4. Оценки качества WUI после второй итерации

Метрика	**lnput_ Manager**	**Window_ Manager**	**Screen**	**Root_ Window**	**Window**	**Menu**	**Menu title**	**Среднее значение**
WMC	0,12	0,42	0,11	0	0,98	0,33	0,27	0,32
NOC	-	-	-	1	1	1	0	0,4
CBO	3	3	0	1	2	2	3	2
RFC	6	11	0	0	27	9	12	9,4
LCOM	3	0	5	0	0	0	0	1,1
CS	3/2	10/8	5/1	0/2	22/22	28/24	11/12	11,3/10,1
NOO	-	-	-	0	0	2	3	0,7
NOA	-	-	-	0	22	6	0	4
SI	-	-	-	0	0	0,23	0,46	0,1
os_{we}	4	4,2	2,2	0	4,45	4,13	9	4,0
NP_{AVG}	0	1,3	1	0	2,18	4,63	1,67	1,5

Метрики, вычисляемые для системы	
DIT	3
NC	7
MOM	48
LOC_Z	223

Сравним оценки качества первой и второй итераций.

1. Рост системных оценок $LOC_{\sum}$, NOM, а также средних значений метрик WMC, RFC, CS, CBO и NOO — свидетельство возрастания сложности продукта.
2. Увеличение значения DIT и среднего значения NOC говорит об увеличении возможности многократного использования классов.
3. На второй итерации в среднем была ослаблена абстракция классов, о чем свидетельствует увеличение средних значений NOC, NOA, SI.
4. Рост средних значений OS_{AVG} и NP_{AVG} говорит о том, что сотрудничество между объектами усложнилось.
5. Среднее значение CBO указывает на увеличение сцепления между классами (это нежелательно), зато снижение среднего значения LCOM свидетельствует, что связность внутри классов увеличилась (таким образом, снизилась вероятность ошибок в ходе разработки).

Вывод: качество разработки в среднем возросло, так как, несмотря на увеличение средних значений сложности и сцепления (за счет добавления в иерархию наследования новых классов), связность внутри классов была увеличена.

В практике проектирования достаточно типичны случаи, когда в процессе разработки меняются исходные требования или появляются дополнительные требования к продукту. Предположим, что в конце второй итерации появилось дополнительное требование — ввести в WUI новый тип окна — диалоговое окно. Диалоговое окно должно обеспечивать не только вывод, но и ввод данных, а также их обработку.

Для реализации этого требования вводится третья итерация конструирования.

Итерация 3 — разработка диалогового окна

Шаг 1: Спецификация представления диалогового окна.

На этом шаге фиксируется представление заказчика об обязанностях диалогового окна. Положим, что оно имеет следующий вид:

1. Диалоговое окно накапливает посылаемые в него символы, отображая их по мере получения.
2. При получении символа конца сообщения (ENTER) полная строка текста принимается в функцию обратного вызова, связанную с диалоговым окном.
3. Функция обратного вызова реализует обслуживание, требуемое пользователю.
4. Функция обратного вызова обеспечивается прикладным программистом.

Шаг 2: Модификация диаграммы Use Case для WUI.

Очевидно, что дополнительное требование приводит к появлению дополнительного элемента Use Case, который находится в отношении «расширяет» с базовым г элементом Use Case Использование окон.

Диаграмма Use Case принимает вид, представленный на рис. 15.13.

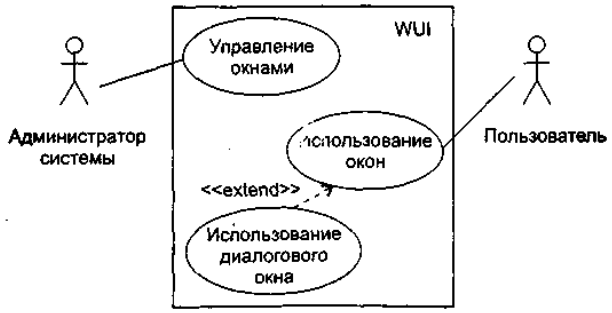


Рис. 15.13. Модифицированная диаграмма Use Case для WUI

Шаг 3: Описание элемента Use Case Использование диалогового окна.

Действия начинаются с ввода пользователем переключающего символа, активизирующего данный тип окна. Символ воспринимается менеджером ввода. Далее пользователь вводит данные, которые по мере поступления отображаются в диалоговом окне. После нажатия пользователем символа окончания ввода (ENTER) данные передаются в функцию обратного вызова как параметр. Выполняется функция обратного вызова, результат выводится в простое окно результата.

Шаг 4: Диаграмма последовательности Использование диалогового окна.

Диаграмма последовательности для сценария Использование диалогового окна показана на рис. 15.14.

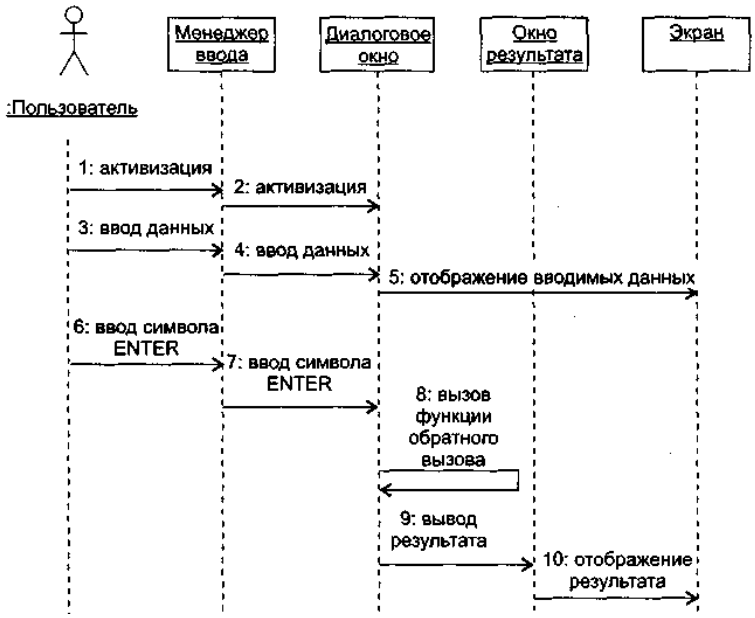


Рис. 15.14. Диаграмма последовательности Использование диалогового окна

Шаг 5: Создание класса.

Для реализации сценария Использование диалогового окна создается новый класс Dialog, который является наследником класса Window. Объекты класса Dialog образуют диалоговые окна.

Класс Dialog переопределяет следующие операции, унаследованные от класса Window:

- framework — формирование диалогового окна. Параметры операции: имя диалогового окна, координаты, ширина окна, заголовок окна и ссылка на функцию обратного вызова. Операция создает каркас окна, устанавливает для него функцию обратного вызова, делает окно видимым и регистрирует его в массиве управляемых окон;
- send_to — обрабатывает пользовательский ввод, посылаемый в диалоговое окно. Окно запоминает символы, вводимые пользователем, а после нажатия пользователем клавиши ENTER вызывает функцию обратного вызова, обрабатывающую эти данные.

Конечное представление иерархии классов WUI показано на рис. 15.15. Результаты оценки качества проекта (в конце третьей итерации) сведены в табл. 15.5. Динамика изменения значений для метрик класса показана в табл. 15.6.

Таблица 15.5. Оценки качества WUI после третьей итерации

Метрика	lnput_ Manager	Window Manager	Screen	Root Window	Window	Menu	Menu-title	Dialog	Среднее значение
WMC	0,12	0,42	0,11	0	0,98	0,33	0,27	0,23	0,31
NOC	-	-	-	1	2	1	0	0	0,5
CBO	3	3	0	1	2	2	3	2	2
RFC	6	11	0	0	27	9	12	7	9,1
LCOM	3	0	5	0	0	0	0	0	1
CS	3/2	10/8	5/1	0/2	22/22	28/24	11/12	24/14	12,2/10,6
NOO	-	-	-	0	0	2	3	2	0,9
NOA	-	-	-	0	22	6	0	0	3,5
SI	-	-	-	0	0	0,23	0,46	0,27	0,14
OS_{AVG}	4	4,2	2,2	0	4,45	4,13	9	11,5	4,9
NP_{AVG}	0	1,3	1	0	2,18	4,63	1,67	4	1,8
Метрики, вычисляемые для системы									
DIT	3								
NC	8								
NOM	50								
LOC_{Σ}	246								

Таблица 15.6. Средние значения метрик класса на разных итерациях

Метрика	Итерация 1	Итерация 2	Итерация 3
WMC	0,3	0,32	0,31
NOC	0,2	0,4	0,5
CBO	1,8	2	2

RFC	8	9,4	9,1
LCOM	1,6	1,1	1
CS	7,2/7	11,3/10,1	12,2/10,6
NOO	0	0,7	0,9
NOA	3,6	4	3,5
SI	0	0,1	0,14
OS_{AVG}	3	4,0	4,9
NP_{AVG}	0,9	1,5	1,8
DIT	1	3	3
NC	5	7	8
NOM	35	48	50
LOC_{Σ}	148	223	246

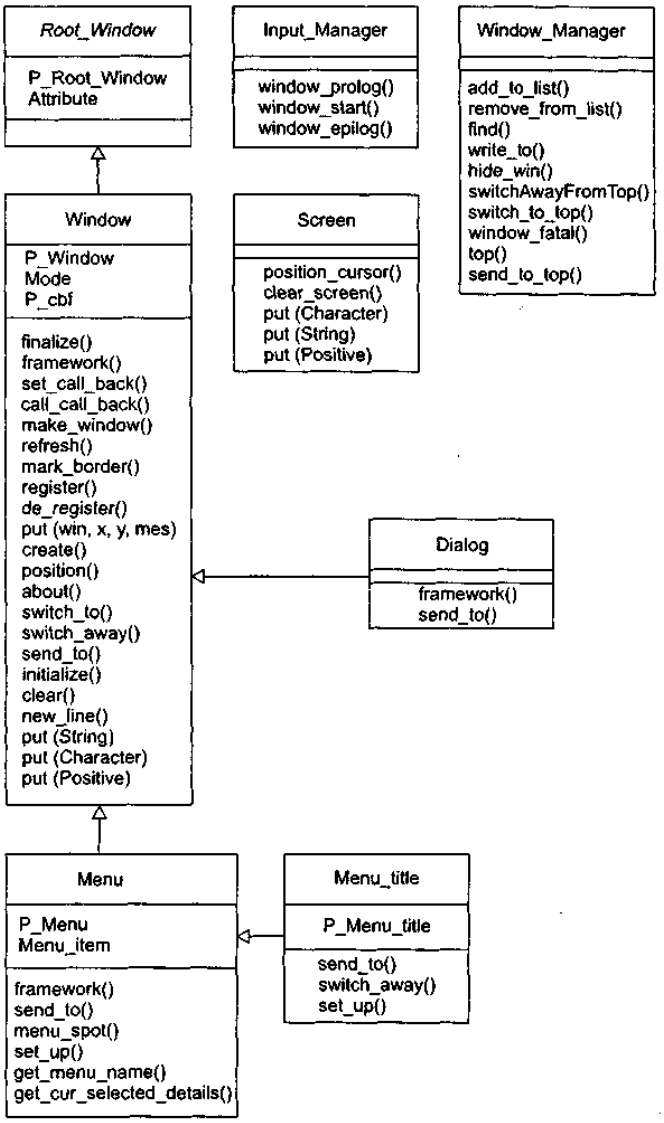


Рис. 15.15. Конечная диаграмма классов WUI

Сравним средние значения метрик второй и третьей итераций:

1. Общая сложность WUI возросла (увеличились значения LOC_{Σ}, NOM и NC), однако повысилось качество классов (уменьшились средние значения WMC и RFC).
2. Увеличились возможности многократного использования классов (о чем свидетельствует рост среднего значения NOC и уменьшение среднего значения WMC).
3. Возросла средняя связность класса (уменьшилось среднее значение метрики LCOM).
4. Уменьшилось среднее значение сцепления класса (сохранилось среднее значение CBO и уменьшилось среднее значение RFC).

Вывод: качество проекта стало выше.

На последней итерации рассчитаны значения интегральных метрик Абреу, они представлены в табл. 15.7. Эти данные также характеризуют качество проекта и подтверждают наши выводы.

Таблица 15.7. Значения метрик Абреу для WUI

Метрика	Значение
MHF	0,49
AHF	0,49
MIF	0,49
AIF	0,29
POF	0,69
COF	0,25

Метрические данные проекта помещают в метрический базис фирмы-разработчика, тем самым обеспечивается возможность их использования в последующих разработках.

Разработка в стиле экстремального программирования

Базовые понятия и методы XP-процесса разработки обсуждались в разделе «XP-процесс» главы 1. Напомним, что основная область применения XP — небольшие проекты с постоянно изменяющимися требованиями заказчика [10], [11], [12], [75]. Заказчик может не иметь точного представления о том, что должно быть сделано. Функциональность разрабатываемого продукта может изменяться каждые несколько месяцев. Именно в этих случаях XP позволяет достичь максимального успеха.

Основным структурным элементом XP-процесса является XP-реализация. Рассмотрим ее организацию.

XP-реализация

Структура XP-реализации показана на рис. 15.16.

Исходные требования к продукту фиксируются с помощью пользовательских историй. Истории позволяют оценить время, необходимое для разработки продукта. Они записываются заказчиком и задают действия, которые должна выполнять для него программная система. Каждая история занимает три-четыре текстовых предложения в терминах заказчика. Кроме того, истории служат для создания тестов приемки. Тесты приемки используют для проверки правильности, реализации пользовательских историй.

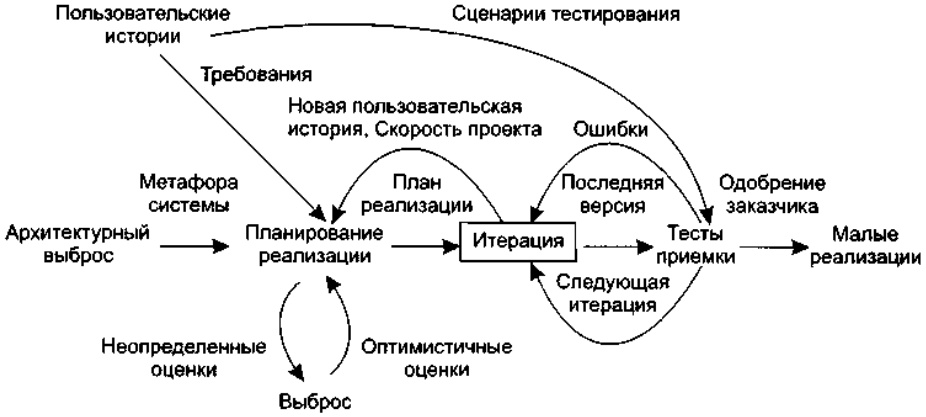


Рис. 15.16. Структура XP-реализации

Очень часто истории путают с традиционными требованиями к системе. Самое важное отличие состоит в уровне детализации. Истории обеспечивают только такие детали, которые необходимы для оценки времени их реализации (с минимальным риском). Когда наступит время реализовать историю, разработчики получают у заказчика более детальное описание требований.

Каждая история получает оценку идеальной длительности — одну, две или три недели разработки (время, за которое история может быть реализована при отсутствии других работ). Если срок превышает три недели, историю следует разбить на несколько частей. При длительности менее недели нужно объединить несколько историй.

Другое отличие истории от требований — во главу угла ставятся желания заказчика. В историях следует избегать описания технических подробностей, таких как организация баз данных или описания алгоритмов.

Если разработчики не знают, как оценить историю, они создают выброс — быстрое решение, содержащее ответ на трудные вопросы. Выброс — минимальное решение, выполняемое в черновом коде и впоследствии выбрасываемое. Результат выброса — знание, достаточное для оценивания.

Итогом архитектурного выброса является создание метафоры системы. Метафора системы определяет ее состав и именование элементов. Она позволяет удержать команду разработчиков в одних и тех же рамках при именовании классов, методов, объектов. Это очень важно для понимания общего проекта системы и повторного использования кодов. Если разработчик правильно предугадывает, как может быть назван объект, это приводит к экономии времени. Следует применять такие правила именования объектов, которые каждый сможет понять без специальных знаний о системе.

Следующий шаг — планирование реализации. Планирование устанавливает правила того, каким образом вовлеченные в проект стороны (заказчики, разработчики и менеджеры) принимают соответствующие решения. Главным итогом является

план выпуска версий, охватывающий всю реализацию. Далее этот план используется для создания планов каждой итерации. Итерации детально планируются непосредственно перед началом каждой из них. Важнейшая задача — правильно оценить сроки выполнения работ по каждой из пользовательских историй. Часто при составлении плана заказчик пытается сократить сроки. Этого делать не стоит, чтобы не вызвать впоследствии проблем. Основная философия планирования строится на том, что имеются четыре параметра измерения проекта — объем, ресурсы, время и качество. Объем — сколько работы должно быть сделано, ресурсы — как много используется разработчиков, время — когда проект будет закончен, качество — насколько хорошо будет реализована и протестирована система. Можно, конечно, установить только три из четырех параметров, но равновесие всегда будет достигаться за счет оставшегося параметра.

План реализации определяет даты выпуска версий и пользовательские истории, которые будут воплощены в каждой из них. Исходя из этого, можно выбрать истории для очередной итерации. В течение итерации создаются тесты приемки, которые выполняются в пределах этой итерации и всех последующих, чтобы обеспечить правильную работу системы. План может быть пересмотрен в случае значительного отставания или опережения по итогам одной из итераций.

В каждую XP-реализацию многократно вкладывается базовый элемент — XP-итерация. Рассмотрим организацию XP-итерации.

XP-итерация

Структура XP-итерации показана на рис. 15.17.

Организация итерации придает XP-процессу динамизм, требуемый для обеспечения готовности разработчиков к постоянным изменениям в проекте. Не следует выполнять долгосрочное планирование. Вместо этого выполняется краткосрочное планирование в начале каждой итерации. Не стоит пытаться работать с незапланированными задачами — до них дойдет очередь в соответствии с планом реализации.

Привыкнув не добавлять функциональность заранее и использовать краткосрочное планирование, разработчики смогут легко приспосабливаться к изменению требований заказчика.

Цель планирования, с которого начинается итерация — выработать план решения программных задач. Каждая итерация должна длиться от одной до трех недель. Пользовательские истории внутри итерации сортируются в порядке их значимости для заказчика. Кроме того, добавляются задачи, которые не смогли пройти предыдущие тесты приемки и требуют доработки.

Пользовательским историям и тестам с отказом в приемке сопоставляются задачи программирования. Задачи записываются на карточках, совокупность карточек образует детальный план итерации.

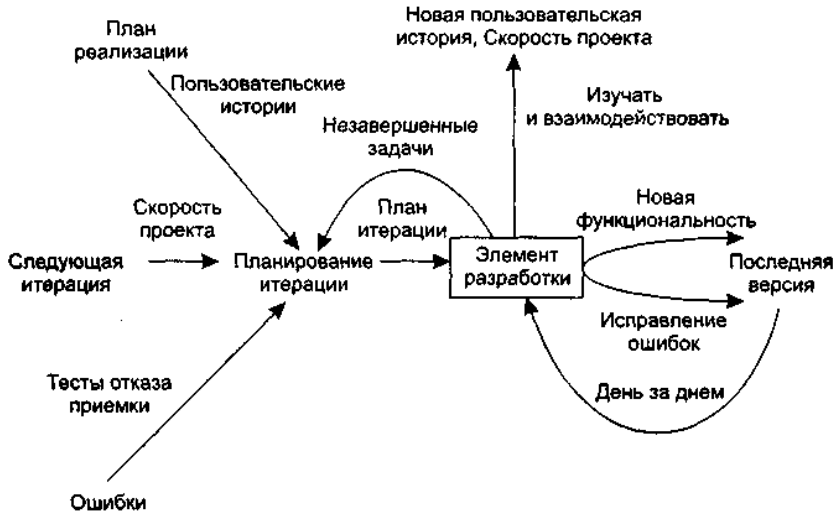


Рис. 15.17. Структура XP-итерации

Для решения каждой из задач требуется от одного до трех дней. Задачи, для которых нужно менее одного дня, группируются вместе, а большие задачи разбивают на более мелкие задачи.

Разработчики оценивают количество и длительность задач. Для определения фиксированной длительности итерации используют метрику «скорость проекта», вычисленную по предыдущей итерации (количество завершенных в ней задач/дней программирования). Если предполагаемая скорость превышает предыдущую скорость, заказчик выбирает истории, которые следует отложить до более поздней итерации. Если итерация слишком мала, к разработке принимается дополнительная история. Вполне допустимая практика — переоценка историй и пересмотр плана реализации после каждых трех или пяти итераций. Первоочередная реализация наиболее важных историй — гарантия того, что для клиента делается максимум возможного. Стиль разработки, основанный на последовательности итераций, улучшает подвижность процесса разработки.

В каждую XP-итерацию многократно вкладывается строительный элемент — элемент XP-разработки. Рассмотрим организацию элемента XP-разработки.

Элемент XP-разработки

Структура элемента XP-разработки показана на рис. 15.18.

День XP-разработчика начинается с установочной встречи. Ее цели: обсуждение проблем, нахождение решений и определение точки приложения усилий всей команды.

Участники утренней встречи стоят и располагаются по кругу, так можно избежать длинных дискуссий. Все остальные встречи проходят на рабочих местах, за компьютером, где можно просматривать код и обсуждать новые идеи.

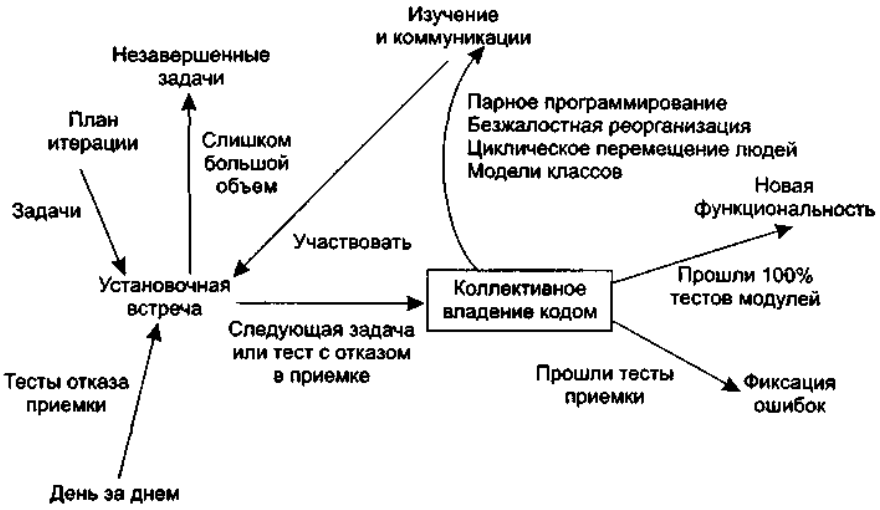


Рис. 15.18. Структура элемента XP-разработки

Весь день XP-разработчика проходит под лозунгом коллективного владения кодом программной системы. В результате этого происходит фиксация ошибок и добавление новой функциональности в систему.

Следует удерживаться от соблазна добавлять в продукт функциональность, которая будет востребована позже. Полагают, что только 10% такой функциональности будет когда-либо использовано, а потери составят 90% времени разработчика. В XP считают, что дополнительная функциональность только замедляет разработку и исчерпывает ресурсы. Предлагается подавлять такие творческие порывы и концентрироваться на текущих, запланированных задачах.

Коллективное владение кодом

Организацию коллективного владения кодом иллюстрирует рис. 15.19.

Коллективное владение кодом позволяет каждому разработчику выдвигать новые идеи в любой части проекта, изменять любую строку программы, добавлять функциональность, фиксировать ошибку и проводить реорганизацию. Один человек просто не в состоянии удержать в голове проект нетривиальной системы. Благодаря коллективному владению кодом снижается риск принятия неверного решения (главным разработчиком) и устраняется нежелательная зависимость проекта от одного человека.

Работа начинается с создания тестов модуля, она должна предшествовать программированию модуля. Тесты необходимо помещать в библиотеку кодов вместе с кодом, который они тестируют. Тесты делают возможным коллективное создание кода и защищают код от неожиданных изменений. В случае обнаружения ошибки также создается тест, чтобы предотвратить ее повторное появление.

Кроме тестов модулей, создаются тесты приемки, они основываются на пользовательских историях. Эти тесты испытывают систему как «черный ящик» и ориентированы на требуемое поведение системы.

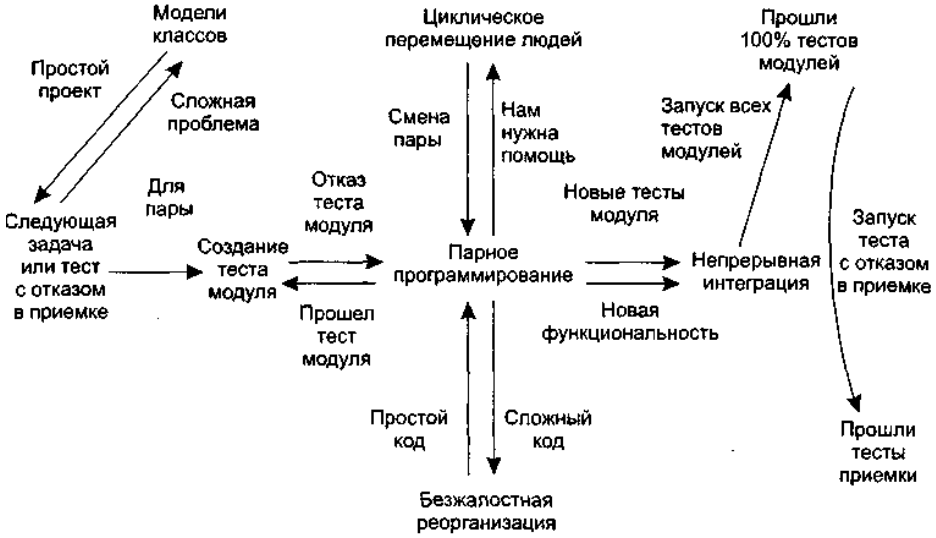


Рис. 15.19. Организация коллективного владения кодом

На основе результатов тестирования разработчики включают в очередную итерацию работу над ошибками. Вообще, следует помнить, что тестирование — один из краеугольных камней XP.

Все коды в проекте создаются парами программистов, работающими за одним компьютером. Парное программирование приводит к повышению качества без дополнительных затрат времени. А это, в свою очередь, уменьшает расходы на будущее сопровождение программной системы.

Оптимальный вариант для парной работы — одновременно сидеть за компьютером, передавая друг другу клавиатуру и мышь. Пока один человек набирает текст и думает (тактически) о создаваемом методе, второй думает (стратегически) о размещении метода в классе.

Во время очередной итерации всех сотрудников перемещают на новые участки работы. Такие перемещения помогают устранить изоляцию знаний и «узкие места». Особенно полезна смена одного из разработчиков при парном программировании.

Замечено, что программисты очень консервативны. Они продолжают использовать код, который трудно сопровождать, только потому, что он все еще работает. Боязнь модификации кода у них в крови. Это приводит к предельному понижению эффективности систем. В XP считают, что код нужно постоянно обновлять — удалять лишние части, убирать ненужную функциональность. Этот процесс называют реорганизацией кода (refactoring). Поощряется безжалостная реорганизация, сохраняющая простоту проектных решений. Реорганизация поддерживает прозрачность и целостность кода, обеспечивает его легкое понимание, исправление и расширение. На реорганизацию уходит значительно меньше времени, чем на сопровождение устаревшего кода. Увы, нет ничего вечного — когда-то отличный модуль теперь может быть совершенно не нужен.

И еще одна составляющая коллективного владения кодом — непрерывная интеграция.

Без последовательной и частой интеграции результатов в систему разработчики не могут быть уверены в правильности своих действий. Кроме того, трудно вовремя оценить качество выполненных фрагментов проекта и внести необходимые коррективы.

По возможности XP-разработчики должны интегрировать и публично отображать, демонстрировать код каждые несколько часов. Интеграция позволяет объединить усилия отдельных пар и стимулирует повторное использование кода.

Взаимодействие с заказчиком

Одно из требований XP — постоянное участие заказчика в проведении разработки. По сути, заказчик является одним из разработчиков.

Все этапы XP требуют непосредственного присутствия заказчика в команде разработчиков. Причем разработчикам нужен заказчик-эксперт. Он создает пользовательские истории, на основе которых оценивается время и назначаются приоритеты работ. В ходе планирования реализации заказчик указывает, какие истории следует включить в план реализации. Активное участие он принимает и при планировании итерации.

Заказчик должен как можно раньше увидеть программную систему в работе. Это позволит как можно раньше испытать систему и дать отзыв о ее работе. Поскольку при укрупненном планировании заказчик остается в стороне, разработчикам нужно постоянно общаться с заказчиком, чтобы получать как можно больше сведений при реализации задач программирования. Нужен заказчик и на этапе функционального тестирования, при проведении тестов приемки.

Таким образом, активное участие заказчика не только предотвращает появление некачественной системы, но и является непременным условием выполнения разработки.

Стоимость изменения и проектирование

В основе организации прогнозирующих (тяжеловесных) процессов конструирования лежит утверждение об экспоненциальной кривой стоимости изменения. Согласно этой кривой, по мере развития проекта стоимость внесения изменений экспоненциально возрастает (рис. 15.20) — то, что на этапе формирования требований стоит единицу, на этапе сопровождения будет стоить тысячу.

В основе адаптивного (облегченного) XP-процесса лежит предположение, что экспоненциальную кривую можно сгладить (рис. 15.21) [25], [30], [36], [37], [62]. Такое сглаживание, с одной стороны, возникает при применении методологии XP, а с другой стороны, оно же в ней и применяется. Это еще раз подчеркивает тесную взаимосвязь между методами XP: нельзя использовать методы, которые опираются на сглаживание, не используя другие методы, которые это сглаживание производят.

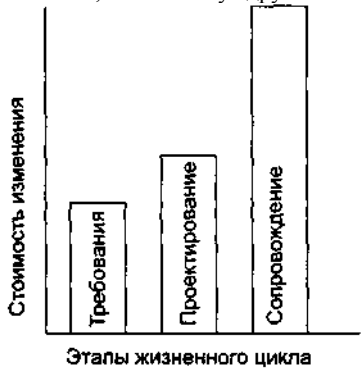


Рис. 15.20. Экспонента стоимости изменения в прогнозирующем процессе

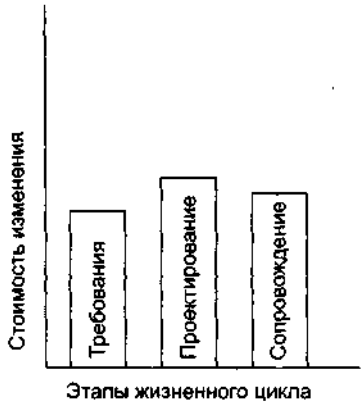


Рис. 15.21. Сглаживание стоимости изменения в адаптивном процессе

К числу основных методов, осуществляющих сглаживание, относят:

- тотальное тестирование;
- непрерывную интеграцию;
- реорганизацию (рефакторинг).

Повторим — надежность кода, которую обеспечивает сквозное тестирование, создает базис успеха, обеспечивает остальные возможности XP-разработки. Непрерывная интеграция необходима для синхронной работы всех сотрудников, так чтобы любой разработчик мог вносить в систему свои изменения и не беспокоиться об интеграции с остальными членами команды. Совместные усилия этих двух методов оказывают существенное воздействие на кривую стоимости изменений в программной системе.

О влиянии реорганизации (рефакторинга) очень интересно пишет Джим Хайсмит (Jim Highsmith) [37]. Он приводит аналогию с весами (см. рис. 15.22 и 15.23). На одной чаше весов лежит предварительное проектирование, на другой — реорганизация. В прогнозирующих процессах разработки перевешивает предварительное проектирование, поскольку скорость изменений низкая (на рисунке скорость иллюстрируется положением точки равновесия). В адаптивных, облегченных процессах перевешивает реорганизация, так как скорость изменений высокая. Это не означает отказа от предварительного проектирования. Однако теперь можно говорить о существовании баланса между двумя подходами к проектированию, из которых можно выбрать наиболее подходящий подход.

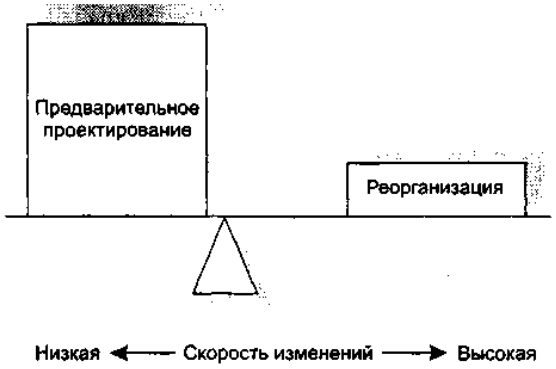


Рис. 15.22. Балансировка проектирования и реорганизации при прогнозе

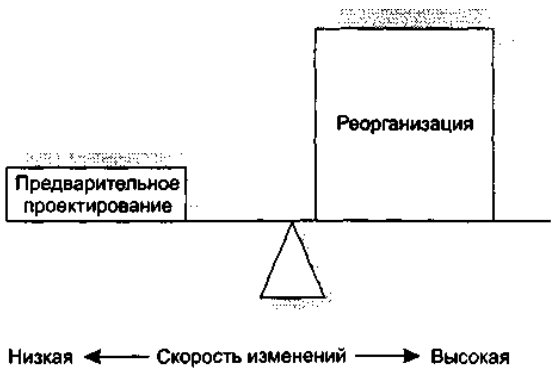


Рис. 15.23. Балансировка проектирования и реорганизации при адаптации

Итак, в адаптивных процессах вообще и в XP-процессе в частности приветствуется простое проектирование.

Простое проектирование основывается на двух принципах:

- это вам не понадобится;
- ищите самое простое решение, которое может сработать.

Что означает первый принцип? Кажется, что все понятно — не надо сегодня писать код, который понадобится завтра. И все же сложности возникают, например, при создании гибких элементов (повторно используемых компонентов и паттернов),

— ведь при этом вы смотрите в будущее, заранее добавляете к общей стоимости работ стоимость нужного проектирования и рассчитываете впоследствии вернуть эти деньги.

Тем не менее XP не советует заниматься созданием гибких элементов заранее. Лучше, если они будут добавляться по мере необходимости. Если сегодня нужен класс Арифметика, который выполняет только сложение, то сегодня я буду встраивать в этот класс именно сложение. Даже если я уверен, что уже на следующей итерации понадобится умножение и мне легко реализовать его сейчас, все равно следует отложить эту работу до следующей итерации — когда в ней появится реальная необходимость.

Экономически такое поведение оправданно — незапланированная работа всегда крадет ресурсы у запланированной. Кроме того, отклонение от плана — это нарушение соглашений с заказчиком. К тому же появляется риск сорвать выполнение текущей работы. И даже если появилось свободное время, решение о его заполнении принимает заказчик («он сверху видит все, ты так и знай!»).

И еще одно оправдание — возможность ошибиться, ведь у нас еще нет подробных требований заказчика. А чем раньше мы введем в проект ошибочное решение, тем хуже.

Теперь о простоте решения. XP-идеолог Кент Бек приводит четыре критерия простой системы:

- система успешно проходит все тесты;
- код системы ясно раскрывает все изначальные замыслы;
- в ней отсутствует дублирование кода;
- используется минимально возможное количество классов и методов.

Успешное тестирование — довольно понятный критерий. Отсутствие дублирования кода, минимальное количество классов/методов — тоже ясные требования. А как расшифровать слова «раскрывает изначальные замыслы»?

XP всячески подчеркивает, что хороший код — это Код, который можно легко прочесть и понять. Если вы хотите сделать комплимент XP-разработчику и скажете, что он пишет «умный код», будьте уверены — вы оскорбили человека.

Словом, сложную конструкцию труднее осмыслить. Понятно, что будущие модификации продукта приведут к его усложнению. Так зачем же усложнять заранее?

Такой стиль работы абсурден, если внедрять его в прогнозирующий, обычный процесс и игнорировать остальные методы XP. В комплексе с остальными XP-причудами он может стать действительно полезным.