МОДУЛЬ 2

РАЗДЕЛ 4. КЛАССИЧЕСКИЕ МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

4.1. Структурное программирование

В 70-х гг. XX в. возник новый подход к разработке алгоритмов и программ, который получил название *структурного программирования*. Одним из первых инициаторов структурного программирования был профессор Э. Дейкстра. В 1965 г. он высказал предположение, что оператор GoTo (оператор безусловного перехода) вообще может быть исключён из языков программирования. По мнению Дейкстры, «квалификация программиста обратно пропорциональна числу операторов GoTo в его программах».

Достоинства структурного программирования по сравнению с интуитивным неструктурным программированием [18]:

1) уменьшение трудностей тестирования программ;

2) повышение производительности труда программистов;

3) повышение ясности и читабельности программ, что упрощает их сопровождение;

4) повышение эффективности объектного кода программ как с точки зрения времени их выполнения, так и с точки зрения необходимых затрат памяти.

К *концепциям* структурного программирования относятся:

- отказ от использования оператора безусловного перехода (GoTo);
- применение фиксированного набора управляющих конструкций;

- использование метода нисходящего проектирования (данный метод рассмотрен в подразд. 4.3).

В основу структурного программирования положено *требование*, чтобы каждый модуль алгоритма (программы) проектировался с единственным входом и единственным выходом. Программа представляется в виде множества *вложенных* модулей, каждый из которых имеет один вход и один выход.

Основой реализации структурированных программ является *принцип Бома и Джакопини*, в соответствии с которым любая программа может быть разработана с использованием лишь *трех базовых структур*:

1) функционального блока;
2) конструкции принятия двоичного (дихотомического) решения;
3) конструкции обобщенного цикла.

Подробно основные положения структурного программирования рассматриваются в дисциплине «Основы алгоритмизации и программирования».

4.2. Модульное проектирование программных средств

Модульное проектирование является одним из первых подходов к разработке структуры ПС и уже несколько десятилетий сохраняет свои позиции как в качестве классического подхода, так и в качестве основы для современных технологий разработки ПС.

При разработке модульных ПС могут использоваться *методы структурного проектирования* или *методы объектно-ориентированного проектирования*. Их целью является формирование структуры создаваемой программы – ее разделение по некоторым установленным правилам на структурные компоненты (***модуляризация***) с последующей иерархической организацией данных компонентов. Для различных языков программирования такими компонентами могут быть подпрограммы, внешние модули, объекты и т.п.

Методы объектно-ориентированного анализа и проектирования изучаются в других дисциплинах.

В данном разделе данной части рассмотрены методы структурного проектирования. Такие методы ориентированы на формирование структуры программного средства по функциональному признаку.

Классическое определение *идеальной* модульной программы формулируется следующим образом. ***Модульная программа*** – это программа, в которой любую часть логической структуры можно изменить, не вызывая изменений в ее других частях [17].

Признаки модульности программ:

1) программа состоит из модулей. Данный признак для модульной программы является очевидным;

2) модули являются независимыми. Это значит, что модуль можно изменять или модифицировать без последствий в других модулях;

3) условие «один вход – один выход». Модульная программа состоит из модулей, имеющих одну точку входа и одну точку выхода. В общем случае может быть более одного входа, но важно, чтобы точки входов были определены и другие модули не могли входить в данный модуль в произвольной точке.

Достоинства модульного проектирования:

1) упрощение разработки ПС;

2) исключение чрезмерной детализации обработки данных;

3) упрощение сопровождения ПС;

4) облегчение чтения и понимания программ;

5) облегчение работы с данными, имеющими сложную структуру.

Недостатки модульности:

1) модульный подход требует большего времени работы центрального процессора (в среднем на 5 – 10 %) за счет времени обращения к модулям;

2) модульность программы приводит к увеличению ее объема (в среднем на 5 – 10 %);

3) модульность требует дополнительной работы программиста и определенных навыков проектирования ПС.

Классические методы структурного проектирования модульных ПС делятся на три основные группы [17]:

1) методы нисходящего проектирования;

2) методы расширения ядра;

3) методы восходящего проектирования.

На практике обычно применяются различные сочетания этих методов.

Резюме

В идеальной модульной программе любую часть логической структуры можно изменить, не вызывая изменений в ее других частях. Идеальная модульная программа состоит из независимых модулей, имеющих один вход и один выход. Модульные программы имеют достоинства и недостатки. Существует три группы классических методов проектирования модульных ПС.

4.3. Нисходящее проектирование

Основное *назначение* нисходящего проектирования – служить средством разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо.

Суть метода нисходящего проектирования заключается в следующем.

На начальном шаге в соответствии с общими функциональными требованиями к программному средству разрабатывается его укрупненная структура без детальной проработки его отдельных частей. Затем выделяются функциональные требования более низкого уровня и в соответствии с ними разрабатываются отдельные компоненты программного средства, не детализированные на предыдущем шаге. Эти действия являются рекурсивными, то есть каждый из компонентов детализируется до тех пор, пока его составные части не будут окончательно уточнены. В последнем случае принимается решение о прекращении дальнейшего проектирования.

На каждом шаге нисходящего проектирования делается оценка правильности вносимых уточнений в контексте правильности функционирования разрабатываемого программного средства в целом.

Компоненты нижнего уровня ПС называются *программными модулями*. Для модулей характерны достаточная простота и прозрачность, позволяющие выполнять их непосредственное программирование.

Таким образом, на каждом шаге разработки уточняется реализация фрагмента

алгоритма, то есть решается более простая задача.

Следует отметить, что метод нисходящего проектирования положен в основу стандартного процесса разработки, регламентированного стандартом *СТБ ИСО/МЭК 12207–2003* (см. подразд. 1.2).

К основным классическим *стратегиям,* на которых основана реализация метода нисходящего проектирования, относятся [17]:

1) пошаговое уточнение; данная стратегия разработана Э. Дейкстрой;

2) анализ сообщений; данная стратегия базируется на работах группы авторов (Йодана, Константайна, Мейерса).

Эти стратегии отличаются способами определения начальных спецификаций требований, методами разбиения задачи на части и правилами записи (*нотациями*), положенными в основу проектирования ПС.

Резюме

Нисходящее проектирование служит средством разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо. Существуют различные стратегии реализации нисходящего проектирования. Основные из них – пошаговое уточнение и анализ сообщений.

4.3.1. Пошаговое уточнение

Пошаговое уточнение является одной из классических стратегий, реализующих метод нисходящего проектирования ПС [17].

При пошаговом уточнении на каждом следующем этапе декомпозиции детализируются программные компоненты очередного более низкого уровня. При этом результаты каждого этапа являются уточнением результатов предыдущего этапа лишь с небольшими изменениями.

Существуют различные способы реализации пошагового уточнения. Рассмотрим два классических способа:

1) проектирование программного средства с помощью псевдокода и управляющих конструкций структурного программирования;

2) использование комментариев для описания обработки данных.

Пошаговое уточнение требует, чтобы взаимное расположение строк текста программы обеспечивало ее читабельность. *Общее правило записи текста разрабатываемой программы*: служебные слова, которыми начинается и заканчивается та или иная управляющая конструкция, записываются на одной вертикали; все вложенные в данную конструкцию псевдокоды (или комментарии, или операторы программы) и управляющие конструкции записываются с отступом вправо.

Преимущества метода пошагового уточнения:

1) основное внимание при его использовании обращается на проектирование корректной структуры программы, а не на ее детализацию;

2) так как каждый последующий этап является уточнением предыдущего лишь с небольшими изменениями, то легко может быть выполнена проверка корректности процесса разработки на всех этапах.

Недостаток метода пошагового уточнения: на поздних этапах проектирования программного средства может обнаружиться необходимость в структурных изменениях, требующих пересмотра более ранних решений.

Резюме

При пошаговом уточнении на каждом следующем этапе декомпозиции детализируются программные компоненты очередного более низкого уровня. К классическим способам реализации пошагового уточнения относятся проектирование программы с помощью псевдокода и управляющих конструкций структурного программирования, а также использование комментариев для описания обработки данных.

4.3.2. Проектирование программных средств с помощью псевдокода и управляющих конструкций структурного программирования

Одним из классических способов реализации пошагового уточнения является проектирование ПС с помощью псевдокода и управляющих конструкций структурного

программирования [17].

При использовании данного способа разбиение программы на модули осуществляется эвристическим способом. На каждом этапе проектирования осуществляется *выбор необходимых управляющих конструкций*, но *операции с данными по возможности не уточняются* (это откладывается на возможно более поздние сроки). Таким образом, фактически проектируется управляющая структура программы.

На этапе, когда принимается решение о прекращении дальнейшего уточнения, оставшиеся неопределенными функции становятся вызываемыми модулями или подпрограммами, а проектируемый модуль – управляющим модулем.

Пример 4.3

Рассмотрим пример проектирования программы с использованием псевдокода и управляющих конструкций структурного программирования. Пусть программа обрабатывает файл дат. Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

В данном примере в качестве псевдокода используются предложения, состоящие из русских слов, соединенных между собой символом подчеркивания.

Первый этап пошагового уточнения

Задается заголовок программы, соответствующий ее назначению.

```
Program Обработка_дат.
```

Второй этап пошагового уточнения

Определяются основные структурные компоненты программы в соответствии с ее основными функциями.

```
Program Обработка_дат;
    Отделить_правильные_даты_от_неправильных {*}
    Сортировать_правильные_даты
    Выделить_зимние_и_летние_даты
    Обработать_неправильные_даты
```

End.

Фрагмент программы, детализированный на втором этапе, располагается правее детализированного на первом этапе.

Третий этап пошагового уточнения

Дальнейшая детализация программы. Детализация фрагмента {*}. Возможно появление необходимости в использовании управляющих конструкций структурного программирования.

```
Program Обработка_дат;
    While не_конец_входного_файла Do
        Begin
            Прочитать_дату
            Проанализировать_правильность_даты
        End
    Сортировать_правильные_даты
    Выделить_зимние_и_летние_даты
    Обработать_неправильные_даты
End.
```

Жирным шрифтом здесь выделены служебные слова цикла с предусловием **While,** представляющего собой одну из управляющих конструкций структурного программирования.

На некоторых этапах уточнения можно приостановить определение некоторых функций, выделяя их в вызываемые модули в том случае, если они функционально независимы от основной обработки (например, в примере это могут быть функции «Проанализировать правильность даты» и «Сортировать правильные даты»).

Процесс детализации продолжается, пока не будет принято решение о прекращении дальнейшей детализации. В этом случае все действия, записанные в последней программе в виде предложений, необходимо оформить в виде подпрограмм. Такие подпрограммы могут быть реализованы, например, с использованием принципа структурного программирования.

Резюме

При использовании псевдокода и управляющих конструкций структурного программирования проектируется управляющая структура программы. На каждом этапе проектирования осуществляется выбор необходимых управляющих конструкций. Уточнение операций с данными по возможности откладывается на поздние сроки.

4.3.3. Использование комментариев для описания обработки данных

Еще одним классическим способом реализации пошагового уточнения является использование комментариев для описания обработки данных [17].

При этом способе на каждом этапе уточнений используются *управляющие конструкции структурного программирования*, а *правила обработки данных не детализируются*. Они описываются в виде комментариев.

На каждом этапе уточнений блоки, представленные комментариями, частично детализируются. Но сами комментарии при этом не выбрасываются. В результате после окончания проектирования получается хорошо прокомментированный текст программы.

Наиболее часто используются следующие ***виды комментариев***:

1) *заголовки*. Объясняют назначение основных компонентов программы на отдельных этапах пошаговой детализации;

2) *построчные*. Описывают мелкие фрагменты программы;

3) *вводные*. Помещаются в начале текста программы и задают общую информацию о ней (например назначение программы, сведения об авторах, дата написания, используемый метод решения, время выполнения, требуемый объем памяти).

Считается, что один из самых больших недостатков программы – отсутствие в ней комментариев. В среднем нужно руководствоваться следующими ***нормами комментариев***:

- 4 – 5 строк комментария-заголовка на каждый компонент программы (подпрограмму, блок, модуль и т.п.);
- по одному комментарию на каждые две-три строки исходного текста для построчных комментариев.

Пример 4.4.

Рассмотрим тот же пример проектирования программы обработки файла дат, который был рассмотрен в п. 4.3.2. Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

Первый этап пошагового уточнения

Записываются вводный комментарий и комментарий к заголовку программы.

```
{Программа обработки дат. Разработчик Иванов И. И.}
{Заголовок программы}
```

Второй этап пошагового уточнения

Определяются основные шаги обработки дат. Сохраняются комментарии предыдущего этапа и добавляются комментарии текущего этапа.

```
{Программа обработки дат. Разработчик Иванов И. И.}
{Заголовок программы}
Program {Обработка дат}
    {Отделение правильных дат от неправильных} {*}
    {Сортировка правильных дат}
    {Выделение зимних и летних дат}
    {Обработка неправильных дат}
End.
```

Третий этап пошагового уточнения

Дальнейшая детализация программы. Детализация фрагмента *. Сохраняются комментарии предыдущих этапов и добавляются комментарии текущего этапа. Возможно появление необходимости в использовании управляющих конструкций структурного программирования.

```
{Программа обработки дат. Разработчик Иванов И. И.}
{Заголовок программы}
```

```
Program {Обработка дат}
    {Отделение правильных дат от неправильных} {*}
    While {не конец входного файла} Do
        Begin
            {Чтение даты}
            {Анализ правильности даты}
        End
    {Сортировка правильных дат}
    {Выделение зимних и летних дат}
    {Обработка неправильных дат}
End.
```

Процесс продолжается до принятия решения о прекращении дальнейшей детализации. После этого все действия, записанные в последней программе в виде комментариев, которые не были реализованы, необходимо оформить в виде подпрограмм.

Резюме

При использовании комментариев для описания обработки данных проектируется управляющая структура программы. На каждом этапе проектирования осуществляется выбор необходимых управляющих конструкций. Операции обработки данных представляются в виде коментариев. Существуют комментарии-заголовки, построчные комментарии и вводные комментарии. Имеются общепринятые нормы использования комментариев.

4.3.4. Анализ сообщений

Анализ сообщений является второй из рассматриваемых классических стратегий, реализующих метод нисходящего проектирования. Анализ сообщений используется в первую очередь для структуризации ПС обработки информации и основывается на анализе потоков данных, обрабатываемых программным средством [17].

Пример 4.5

Рассмотрим тот же пример проектирования программы обработки файла дат, который был рассмотрен в пп. 4.3.2, 4.3.3. Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, перенести летние и зимние даты в выходной файл, вывести неправильные даты.

Рис. 4.1, 4.2 иллюстрируют возможные варианты укрупненного и детализированного представления потоков информации и обрабатывающих их процессов для данной программы, представленные с помощью диаграмм потоков данных (Data Flow Diagram, DFD) в нотации Йодана–ДеМарко (см. подразд. 5.3).

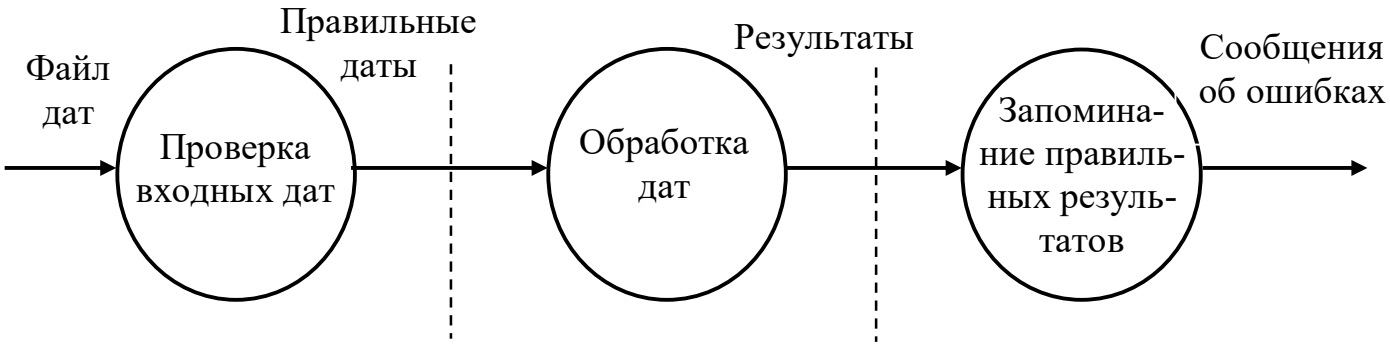

Рис. 4.1. Укрупненная диаграмма потоков данных
для программы обработки файла дат

В соответствии со стратегией анализа сообщений первоначальный поток данных разбивается на три потока: первый содержит непреобразованные входные данные, второй – потоки преобразования, третий – только выходную информацию. Границы, разделяющие эти потоки, на рис. 4.1, 4.2 показаны штриховыми линиями. Они делят диаграмму на три части.

Данные, подлежащие обработке с помощью процесса «Обработка дат» (см. рис. 4.1), могут не включать все входные даты, но это еще часть входных данных. Процесс «Обработка дат» в общем случае может включать различные виды преобразования данных (например, кодирование, декодирование, вычисления и др.). Результаты данного процесса представляют собой выходные данные, хотя они могут быть еще неотформатированными, неотредактированными, возможно, неверными.

Три части программы, соответствующие трем потокам данных, принято называть соответственно ***истоком, преобразователем и стоком*** [18].

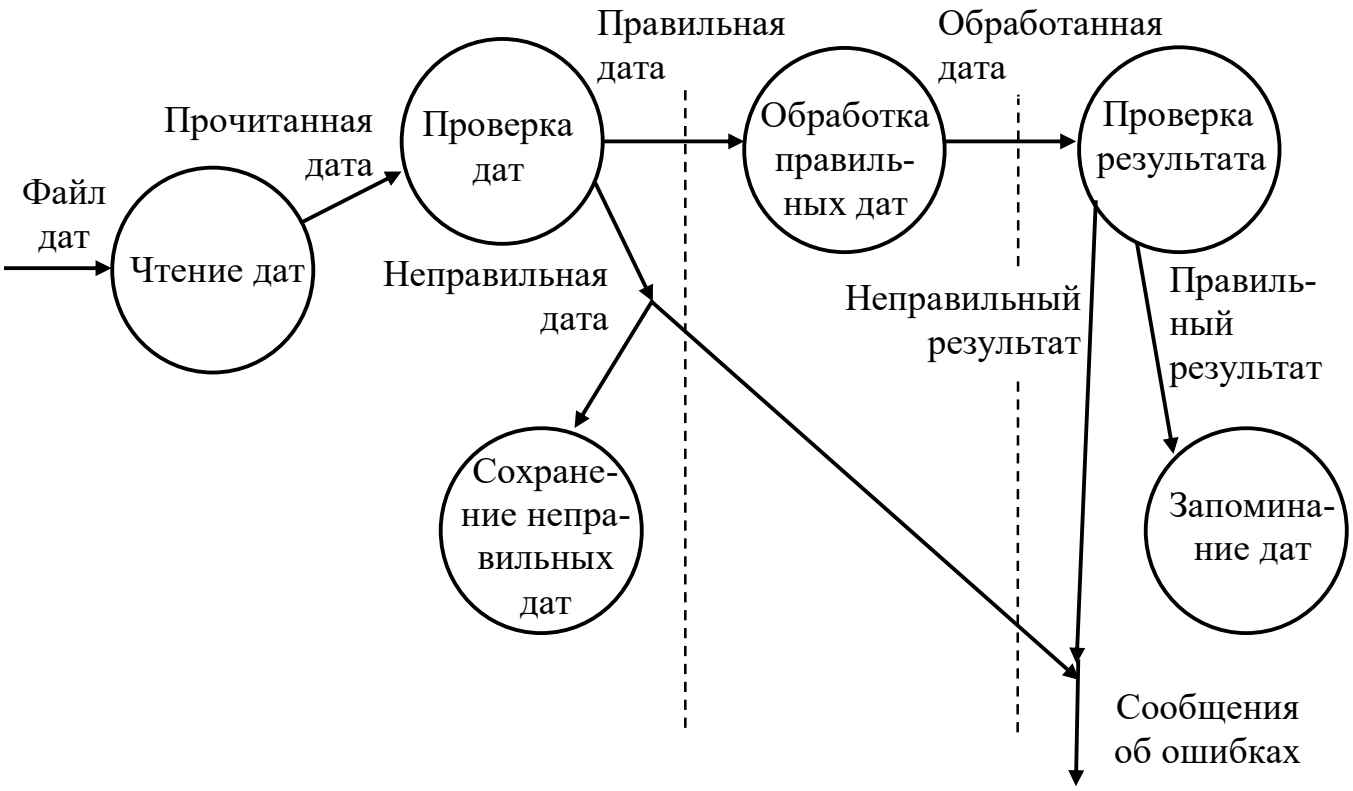

Рис. 4.2. Детализированная диаграмма потоков данных
для программы обработки файла дат

Преобразователь – это основная часть программы, *исток* выполняет функцию управления входным потоком данных, *сток* выполняет функцию управления выходным потоком данных.

На рис. 4.3 представлен общий вид DFD-диаграммы разбиения любой программы на исток – преобразователь – сток. Линии на диаграмме показывают потоки передачи данных между процессами.

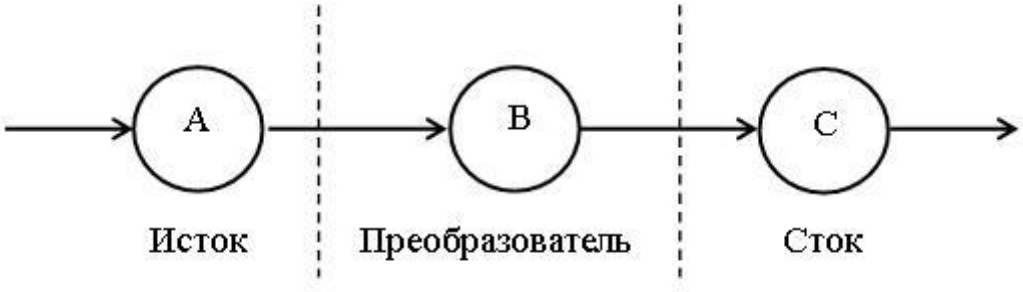

Рис. 4.3. DFD-диаграмма разбиения программы
на исток-преобразователь-сток

Рис. 4.4 содержит соответствующую схему иерархии компонентов программы. В схеме иерархии линии указывают связи по управлению между компонентами, а также изображают отношения типа вызывающий – вызываемый.

Процесс декомпозиции программы заключается в рекурсивном использовании метода разбиения на исток – преобразователь – сток на отдельных ветвях ее древовидной структуры. На нижнем уровне в результате формируется совокупность программных модулей – наименьших единиц проектирования программы. Каждый из модулей может быть реализован в зависимости от назначения, сложности и размера как независимый модуль, внутренняя подпрограмма или некоторая часть основной программы.

Следует отметить, что не все компоненты программы обязательно должны быть разбиты на три компонента более низкого уровня. Результат декомпозиции компонента-стока должен обязательно содержать сток, компонента-преобразователя – преобразователь, компонента-истока – исток. Вызывающий компонент – это главный сток для компонента-истока и главный исток для компонента-стока.

Рис. 4.4. Схема иерархии компонентов программы

Если структуры данных имеют большие размеры, управление ими осуществляется на уровне детализации данных для компонентов истока и стока.

На рис. 4.5 представлена иерархическая структура компонентов для первого уровня декомпозиции программы обработки файла дат, на рис. 4.6 – для второго уровня декомпозиции.

На данных рисунках приняты следующие обозначения:

1 – проверка входных дат (исток);

2 – обработка дат (преобразователь);

3 – запоминание правильных результатов (сток);

4 – чтение дат (исток истока 1);

5 – проверка дат (преобразователь истока 1);

6 – сохранение неправильных дат (сток истока 1);

7 – обработка правильных дат (преобразователь преобразователя 2);

8 – проверка результата (преобразователь стока 3);

9 – запоминание дат (сток стока 3).

Количество уровней декомпозиции определяется сложностью задачи и необходимой степенью детализации.

Рис. 4.5. Схема иерархии программы обработки дат
(декомпозиция первого уровня)

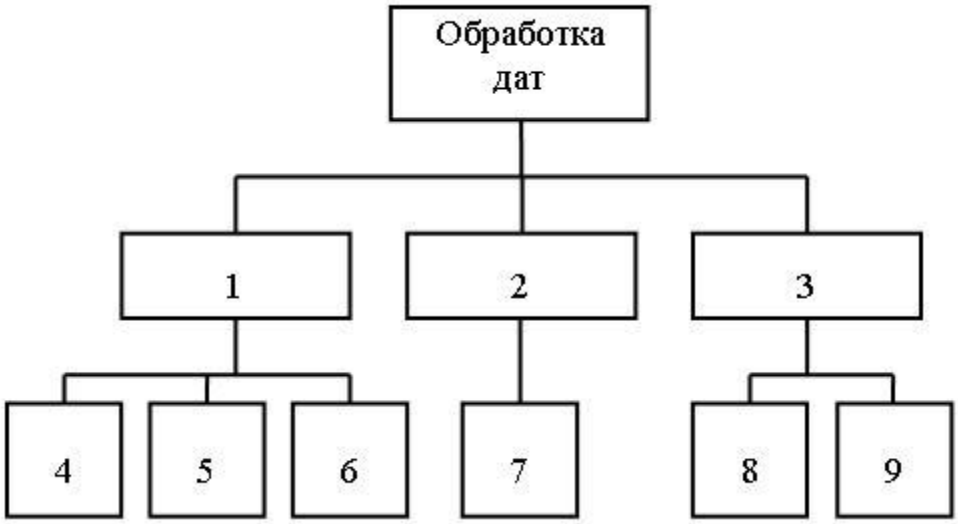

Рис. 4.6. Схема иерархии программы обработки дат
(декомпозиция второго уровня)

Каждый компонент программы при информационном обмене использует определенную часть данных. Однако в иерархической структуре программы информационные связи между компонентами не отражены. Поэтому описание иерархической структуры должно содержать *таблицу взаимодействия компонентов*, показывающую передачу данных между ними. В этой таблице должны быть определены все способы информационного обмена, задаваемые как при помощи формальных параметров, так и с помощью глобальных переменных.

В табл. 4.1 представлены информационные связи между компонентами программы обработки дат. В таблице отражены те данные, которые передаются каждому компоненту в момент его вызова. Входные данные управляющего компонента изображаются как выходные для вызываемых им компонентов. Выходные данные управляющего компонента отображаются как входные для вызываемых им компонентов. Поэтому компонент-исток имеет выходные данные, сток – входные данные, преобразователь – и те, и другие.

Резюме

Анализ сообщений основывается на анализе потоков данных, обрабатываемых программным средством. Первоначальный поток данных разбивается на три потока: исток, преобразователь и сток. Процесс декомпозиции заключается в рекурсивном использовании метода разбиения на исток – преобразователь – сток на отдельных ветвях иерархической структуры программы. Описание иерархической структуры должно содержать таблицу взаимодействия компонентов, показывающую передачу данных между ними.

Таблица 4.1

Таблица связей между компонентами

Компонент	Вход	Выход
1	—	Правильные даты
2	Правильные даты	Результаты
3	Результаты	—
4	—	Прочитанная дата
5	Прочитанная дата	Правильная дата Неправильная дата
6	Неправильная дата	—
7	Правильная дата	Обработанная дата
8	Обработанная дата	Правильный результат Неправильный результат

Компонент	Вход	Выход
9	Правильный результат	—

4.4. Методы восходящего проектирования

При использовании восходящего проектирования в первую очередь выделяются функции нижнего уровня, которые должно выполнять программное средство. Эти функции реализуются с помощью программных модулей самых нижних уровней. Затем на основе этих модулей проектируются программные компоненты более высокого уровня. Данные компоненты реализуют функции более высокого уровня. Процесс продолжается, пока не будет завершена разработка всего программного средства.

В чистом виде метод восходящего проектирования используется крайне редко. Основным его *недостатком* является то, что программисты начинают разработку программного средства с несущественных, вспомогательных деталей. Это затрудняет проектирование программного средства в целом.

Метод восходящего проектирования *целесообразно* применять в следующих случаях:

- существуют разработанные модули, которые могут быть использованы для выполнения некоторых функций разрабатываемой программы;
- заранее известно, что некоторые простые или стандартные модули потребуются нескольким различным частям программы (например, подпрограмма анализа ошибок, ввода-вывода и т.п.).

Обычно используется *сочетание* методов нисходящего и восходящего проектирования. Такое сочетание возможно различными *способами*. Ниже рассмотрены два из них [18].

Первый способ сочетания

Выделяются ключевые (наиболее важные) модули промежуточных уровней разрабатываемой программы. Затем проектирование ведется нисходящим и восходящим методами одновременно (рис. 4.7).

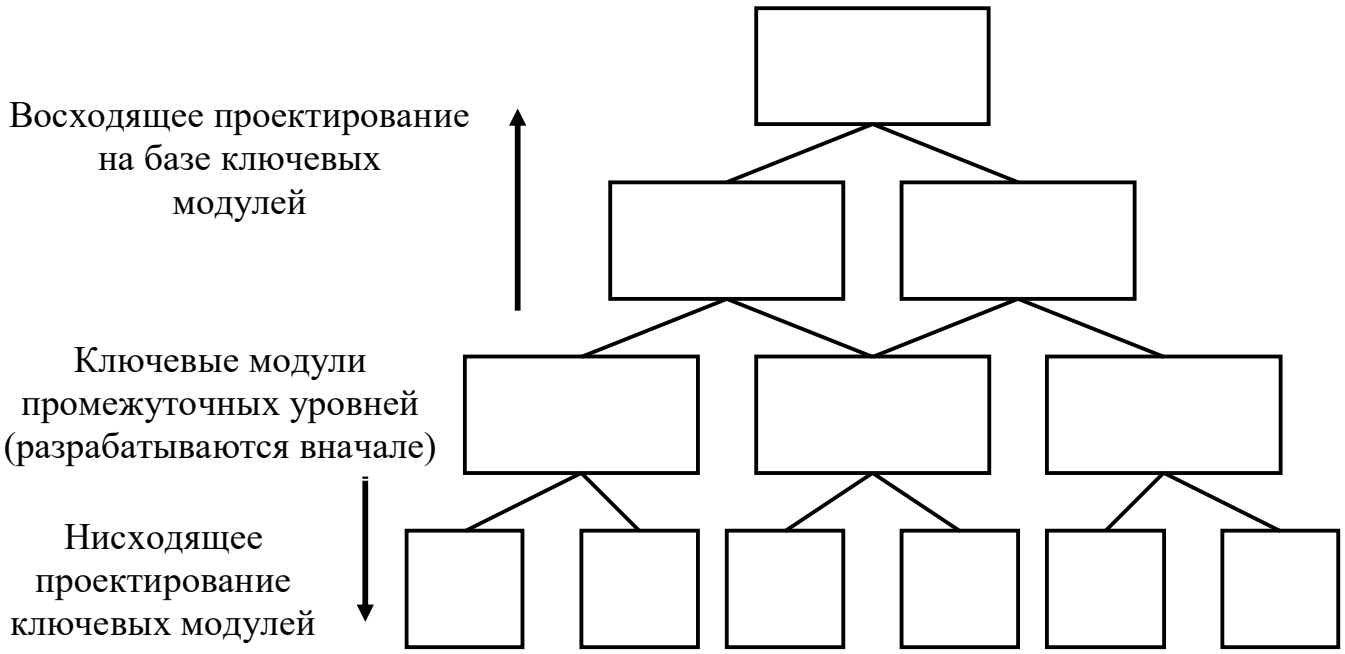

Рис. 4.7. Проектирование программы нисходящим и восходящим методами на базе ключевых модулей

Второй способ сочетания

Проектируются модули нижнего уровня (те, которые необходимо спроектировать заранее). Затем программа проектируется одновременно нисходящим и восходящим методами (рис. 4.8).

При таком способе проектирования наиболее важной задачей является согласование интерфейса между верхними и нижними уровнями программы, выполняемое в последнюю очередь. Это является существенным недостатком данного способа сочетания. Разработчики должны обладать достаточно высокой квалификацией, чтобы не оказалось, что верхняя и нижняя части программы несовместимы между собой.

Резюме

При использовании метода восходящего проектирования в первую очередь реализуются функции нижнего уровня программы. На основе полученных модулей проектируются программные компоненты более высокого уровня. Часто используется сочетание методов нисходящего и восходящего проектирования. Такое сочетание возможно различными способами.

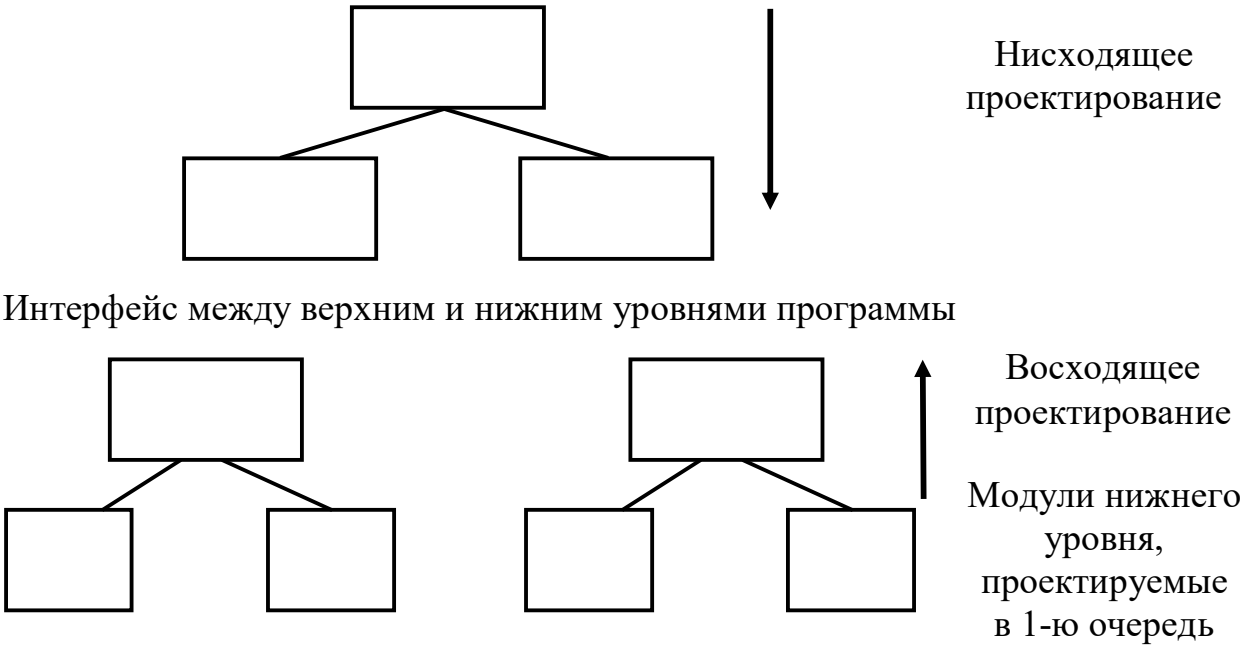

Рис. 4.8. Проектирование программы нисходящим и восходящим методами на базе модулей нижнего уровня

4.5. Методы расширения ядра

При использовании данных методов в первую очередь создается ядро (основная часть) программы. Затем данное ядро постепенно расширяется, пока не будет полностью сформирована управляющая структура разрабатываемой программы.

Существует два основных подхода к реализации методов расширения ядра.

Первый подход основан на методах проектирования *структур данных*, используемых при иерархическом проектировании модулей. Данный подход применяется в методах JSP и JSD, разработанных Майклом Джексоном [20, 29].

Второй подход основан на определении областей хранения данных с последующим анализом связанных с ними функций. Данный подход использует метод определения спецификаций модуля, разработанный Парнасом [18].

В данном разделе рассмотрен метод JSP, реализующий первый подход. Метод JSD, являющийся развитием метода JSP, кратко рассмотрен в п. 5.5.1.

4.6. Метод JSP Джексона

Метод структурного программирования JSP (Jackson Structured Programming) разработан М. Джексоном в 70-х гг. XX в. Данный метод наиболее эффективен в случае высокой степени структуризации данных. Это характерно, например, для класса планово-экономических задач.

Метод JSP (называемый также методом Джексона) базируется на *исходном положении*, состоящем в том, что структура программы зависит от структуры подлежащих обработке данных. Поэтому *структура данных может использоваться для формирования структуры программы*.

4.6.1. Основные конструкции данных

Метод JSP основывается на возможности представления структур данных и структур программ единым набором основных конструкций. М. Джексоном предложены *четыре основные конструкции данных* [20].

1. Конструкция последовательности данных

Эта конструкция используется, когда два или более компонента данных следуют друг за другом строго последовательным образом и образуют единый компонент данных.

Графически конструкция последовательности данных (последовательность данных) изображается в соответствии с рис. 4.9. На данном рисунке компоненты **B, C, D, E** объединяются в указанном порядке и образуют последовательность **A**.

В конструкции последовательности данных должно быть не менее двух подкомпонентов, причем каждый из них должен встречаться строго один раз и обязательно в предписанном порядке.

Рис. 4.10 представляет пример последовательности данных – запись даты **D**, состоящая из трех последовательных частей – поля **N** числа, поля **M** месяца и поля **Y** года.

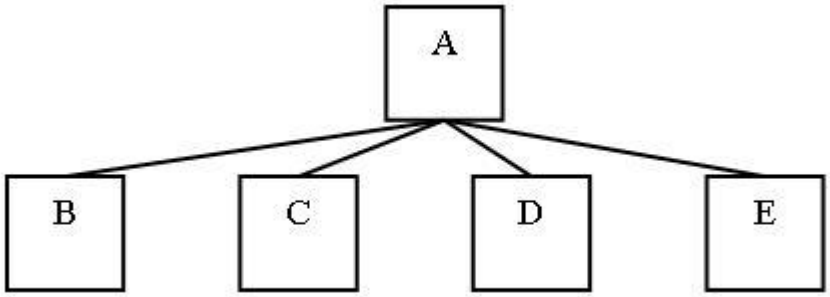

Рис. 4.9. Конструкция последовательности данных

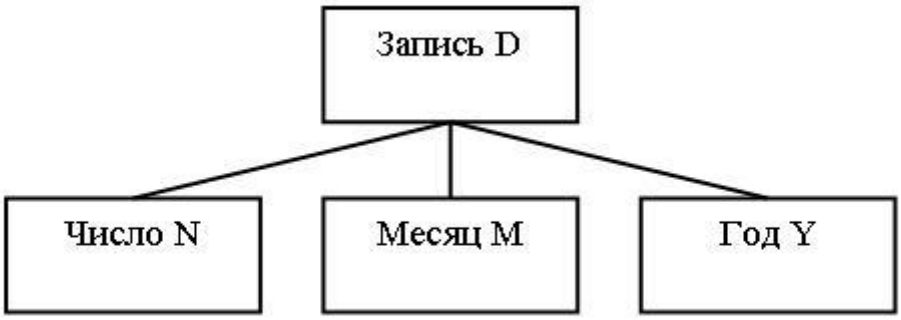

Рис. 4.10. Пример последовательности данных

2. Конструкция выбора данных

Конструкцией выбора данных (выбором данных) называется конструкция сведения результирующего компонента данных к одному из двух или более выбираемых подкомпонентов.

На рис. 4.11 представлена конструкция выбора данных, результирующий компонент **S** которой сводится к подкомпоненту **P**, **Q** или **R**. Внешне конструкция выбора данных отличается от конструкции последовательности наличием символа «**o**» в верхнем правом углу каждого из выбираемых подкомпонентов.

Очевидно, что в конструкции выбора должно быть не менее двух подкомпонентов.

На практике выбор может заключаться в том, что подкомпонент выбора либо присутствует, либо отсутствует. Рис. 4.12 иллюстрирует некорректное изображение такой конструкции выбора. Данная конструкция должна быть представлена в соответствии с рис. 4.13.

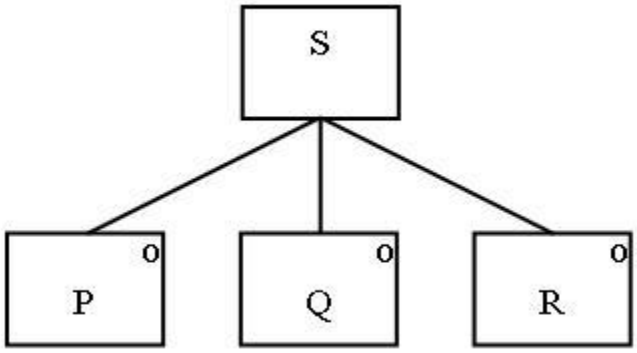

Рис. 4.11. Конструкция выбора данных

Рис. 4.12. Пример некорректного
изображения конструкции выбора данных

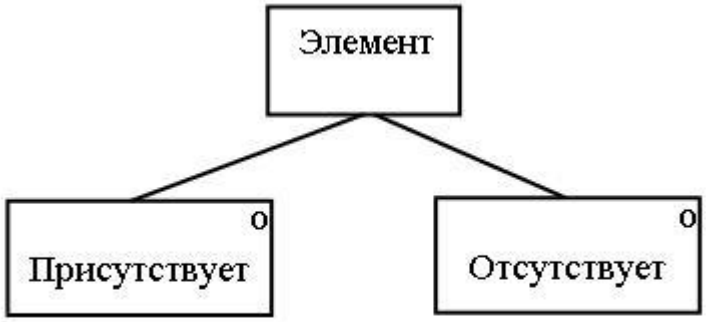

Рис. 4.13. Пример правильного представления
конструкции выбора данных

3. Конструкция повторения данных

Данная конструкция применяется тогда, когда конкретный элемент данных может

повторяться от нуля до неограниченного числа раз.

Представление конструкции повторения данных (повторения данных) в нотации структур Джексона иллюстрирует рис. 4.14. На данном рисунке компонент **I** состоит из повторяющихся подкомпонентов **X**.

У конструкции повторения только один подкомпонент. Признаком повторяемой части конструкции является символ ***** в верхнем правом углу.

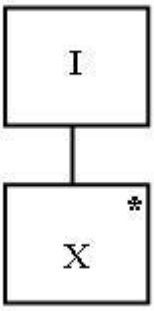

Рис. 4.14. Конструкция повторения данных

Пример конструкции повторения данных представлен на рис. 4.15. На данном рисунке файл **F** состоит из нуля или более записей **D**.

Если некоторый компонент должен включать одно или более появлений повторяемого подкомпонента (ситуация нуля повторений подкомпонента возникнуть не может), то используется конструкция, представленная на рис. 4.16.

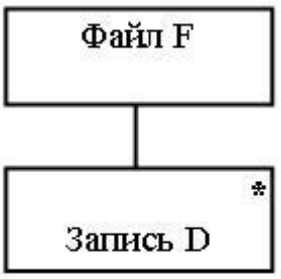

Рис. 4.15. Пример конструкции
повторения данных

Рис. 4.16. Пример конструкции повторения данных
с не менее чем одним появлением

В данном случае файл **F** изображается последовательностью из двух подкомпонентов. Подкомпонент «Остаток файла» представляет собой повторение записи **D**.

При необходимости конкретное количество повторений может быть указано в круглых скобках рядом с повторяемой частью конструкции (рис. 4.17). Указание числа повторений является расширением нотации метода JSP.

4. Элементарная конструкция

Элементарными являются те компоненты, которые не разбиваются далее на подкомпоненты. Примерами элементарных конструкций являются, например, первая запись **D** и запись **D** на рис. 4.16, компоненты число **N**, месяц **M**, год **Y** на рис. 4.10.

Компонент может являться элементарным, потому что его нельзя разложить дальше или потому, что с практической точки зрения отсутствует необходимость в его дальнейшем разбиении.

Резюме

Метод JSP основывается на возможности представления структур данных и структур программ единым набором основных конструкций. Существует четыре основных конструкции данных: конструкция последовательности, конструкция выбора, конструкция повторения и элементарная конструкция.

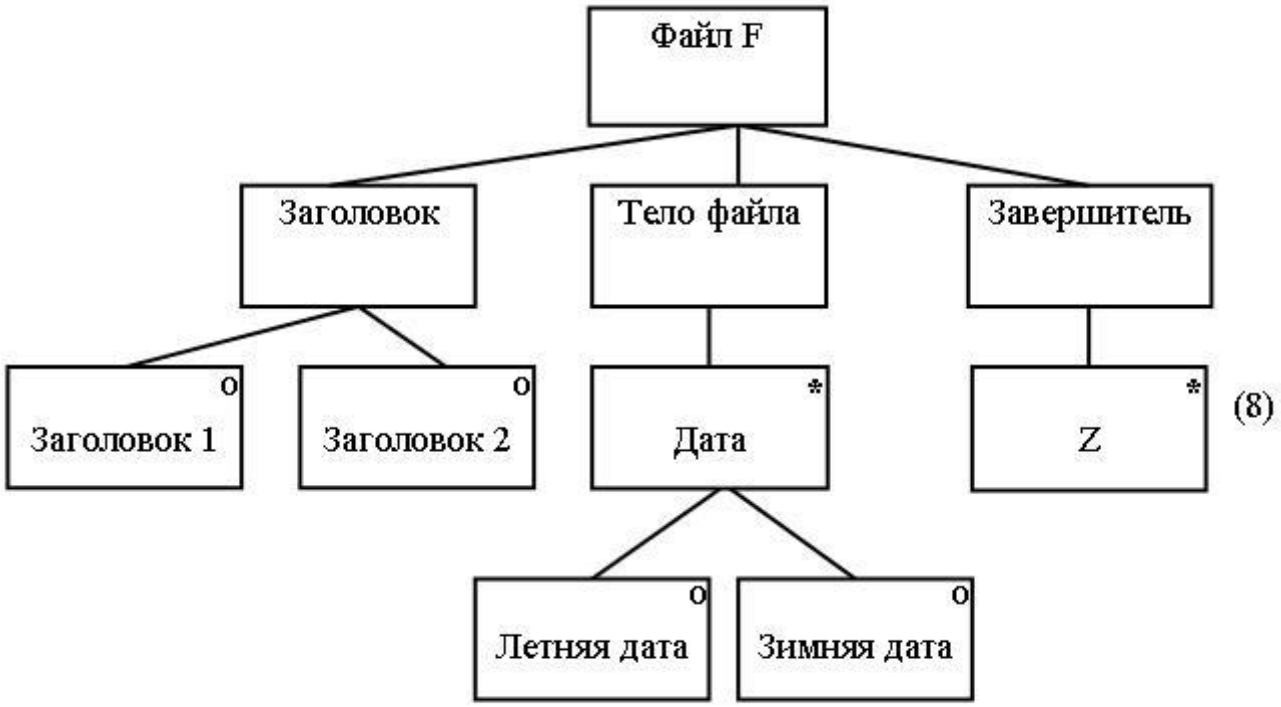

Рис. 4.17. Иерархическая структура данных

4.6.2. Построение структур данных

Рассмотрим возможность формирования сложных структур данных на основе комбинации четырех описанных в п. 4.6.1 конструкций данных.

Пусть имеется некоторый файл **F**, состоящий из заголовка, за которым следует совокупность дат, завершающаяся признаком окончания. Заголовок может иметь один из двух типов, каждая дата представляет собой летнюю или зимнюю дату, завершитель состоит из восьми символов **Z**. Структуру файла **F** можно представить в виде *иерархической структуры данных*, базируясь на основных конструкциях данных Джексона (см. рис. 4.17).

На данном рисунке файл **F** представлен в виде конструкции последовательности данных, заголовок – в виде конструкции выбора, тело файла и завершитель – конструкции повторения, дата – конструкции выбора.

На первый взгляд кажется, что в вышеприведенной структуре компонент «Тело файла» является излишним (см. рис. 4.17). Однако если данный компонент убрать, то последовательность «Файл F» будет состоять из трех подкомпонентов, вторым среди

которых будет являться «Дата». Подкомпонент «Дата» представляет собой повторяемый подкомпонент. Следовательно, «Дата» может присутствовать в файле любое число раз. Но в конструкции последовательности данных любой подкомпонент должен встретиться ровно один раз. Поэтому наличие компонента «Тело файла» в структуре данных, представленной на рис. 4.17, является обязательным.

Из вышеприведенного примера видно, что если данные можно представить в виде иерархической структуры, то их можно изобразить с помощью набора из четырех основных конструкций Джексона.

Помимо иерархической структуры данных широко используется сетевая и реляционная структуры данных.

В *сетевой структуре данных* имеются связи между отдельными компонентами, включая компоненты самых разных уровней структуры. Проектирование программы для обработки таких данных связано со значительными трудностями. Поэтому сетевые структуры данных обычно преобразуются к иерархическому виду. В этом случае один компонент сетевой структуры обычно принимается в качестве основного (ключевого). Для преобразования сетевой структуры в иерархическую упрощают сложные взаимосвязи между данными, не существенные для конкретного вида данных.

Пример сетевой структуры данных иллюстрирует рис. 4.18 [20]. На рис. 4.19 показан возможный вариант преобразования этих данных к иерархическому виду.

При *реляционном подходе* структура данных представляется в виде совокупности таблиц. На рис. 4.20 представлена таблица, состоящая из пяти строк и шести столбцов. Очевидно, что сами таблицы имеют иерархическую структуру. На рис. 4.21 показаны два способа иерархического представления таблицы, состоящей из пяти строк и шести столбцов.

Таким образом, реальные наборы данных, как правило, могут быть представлены в виде иерархических структур.

Представление структур данных в виде иерархий является первым этапом проектирования программы по методу Джексона (методу JSP). На следующих этапах описывается взаимосвязь между структурами данных и программой.

Резюме

Большинство структур (иерархическая, сетевая, реляционная) реальных наборов

данных может быть сведено к иерархическим структурам, которые могут быть представлены в нотации структур метода JSP Джексона.

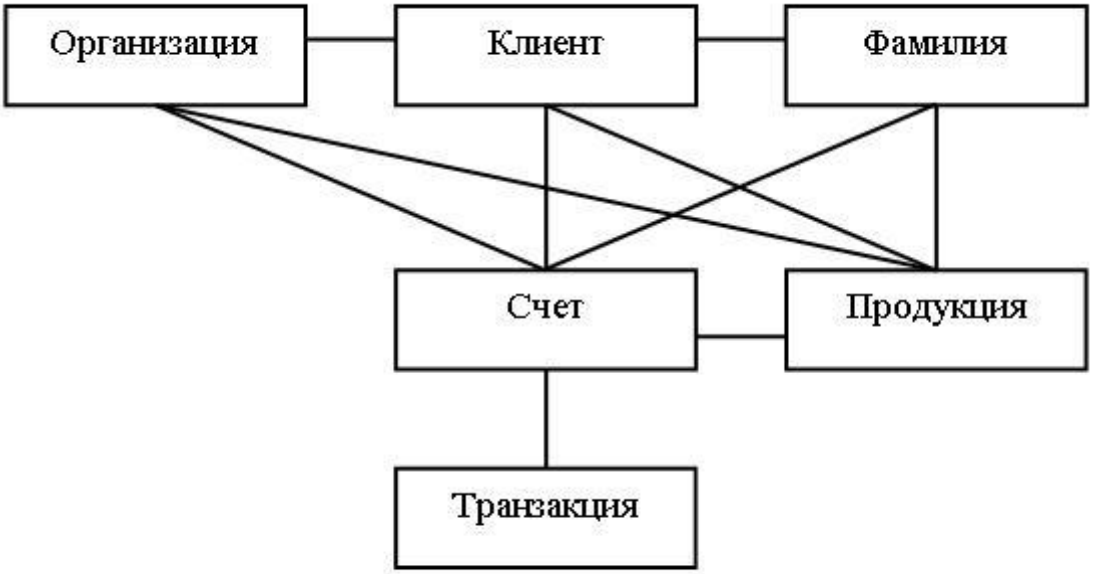

Рис. 4.18. Пример сетевой структуры данных

Рис. 4.19. Иерархический вид сетевой структуры данных

Столбцы

Строки	1	2	3	4	5	6
1						
2						
3						
4						
5						

Рис. 4.20. Пример представления структуры данных
при использовании реляционного подхода

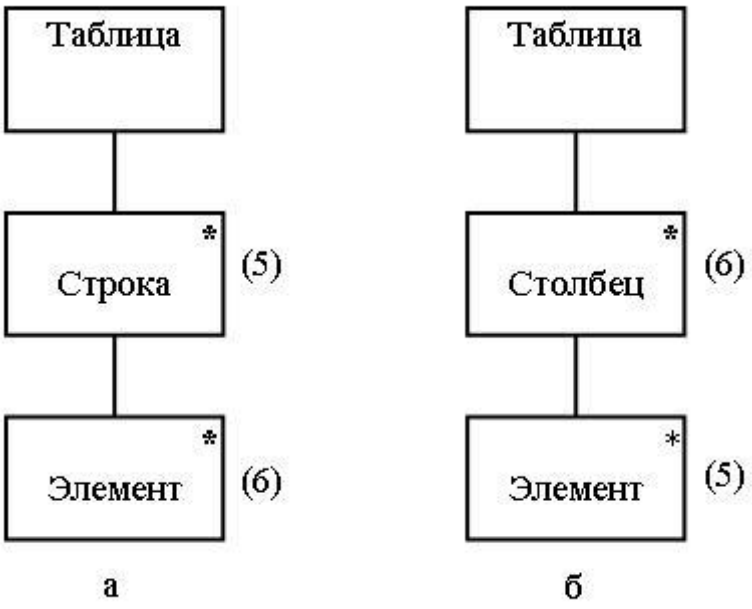

Рис. 4.21. Иерархическая структура таблиц:
а – представление таблицы в виде набора строк;
б – представление таблицы в виде набора столбцов

4.6.3. Проектирование структур программ

В соответствии с методом JSP конструкции, используемые для построения структур данных, применяются и для построения структур программ. Так же, как и данные, программы могут быть составлены из конструкций последовательности, выбора и повторения.

Большинство ПС предназначено для обработки некоторых входных данных и получения некоторых выходных данных. Структура выходных данных формируется программой в результате некоторого преобразования структуры входных данных. Таким образом, для проектирования структуры программы необходимо определить взаимосвязь между входными данными, выходными данными и процессом преобразования [20].

Пример 4.6

Рассмотрим простейший пример [20]. Пусть необходимо найти суммы элементов строк в массиве, состоящем из 15 строк и 10 столбцов.

Рис. 4.22 иллюстрирует структуры данных, представляющие входные и выходные данные разрабатываемой программы.

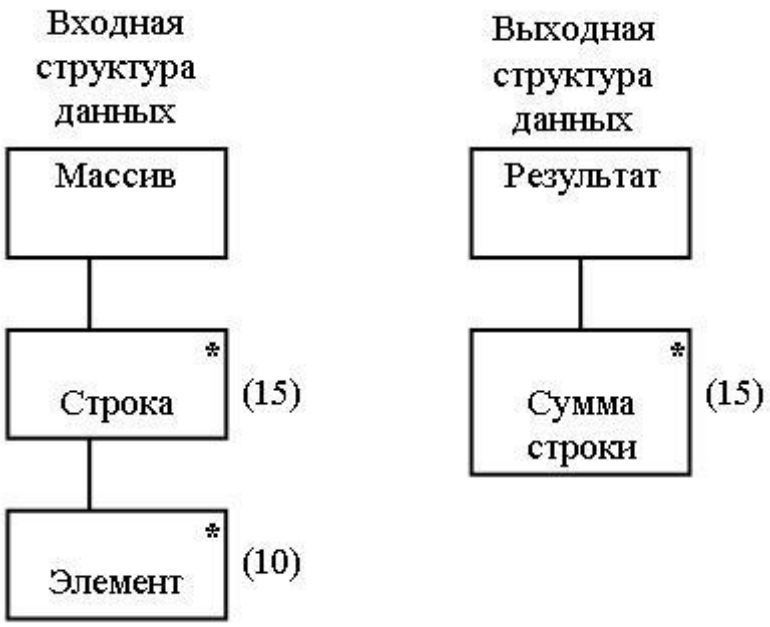

Рис. 4.22. Структуры входных и выходных данных

Очевидно, что между обеими структурами имеется непосредственная взаимосвязь. Компонент «Сумма строки» в структуре «Результат» появляется столько раз, сколько раз компонент «Строка» появляется во входной структуре «Массив», и в том же порядке.

Соответствия между компонентами входной и выходной структур данных принято изображать жирными линиями с двусторонними стрелками. Соответствие между структурами входных и выходных данных для рассматриваемого примера представлено на рис. 4.23.

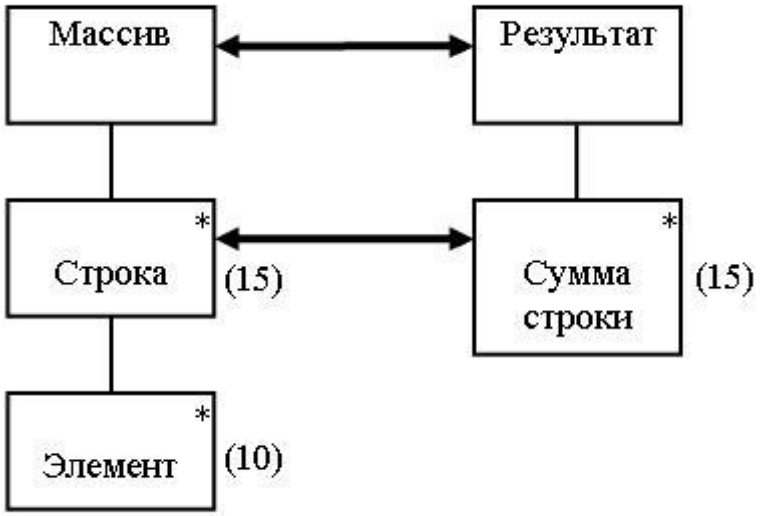

Рис. 4.23. Соответствие между структурами
входных и выходных данных

Каждая пара соответствующих компонентов входных и выходных данных образует основу одного компонента программы. Поэтому проектирование упрощенной структуры программы (ее ядра) выполняется путем «слияния» соответствующих компонентов структур входных и выходных данных. Это значит, что на месте линий соответствия размещаются компоненты программы, которым присваивается соответствующее имя (рис. 4.24). Слияние двух повторяемых **N** раз компонентов данных сформирует один повторяющийся **N** раз компонент программы.

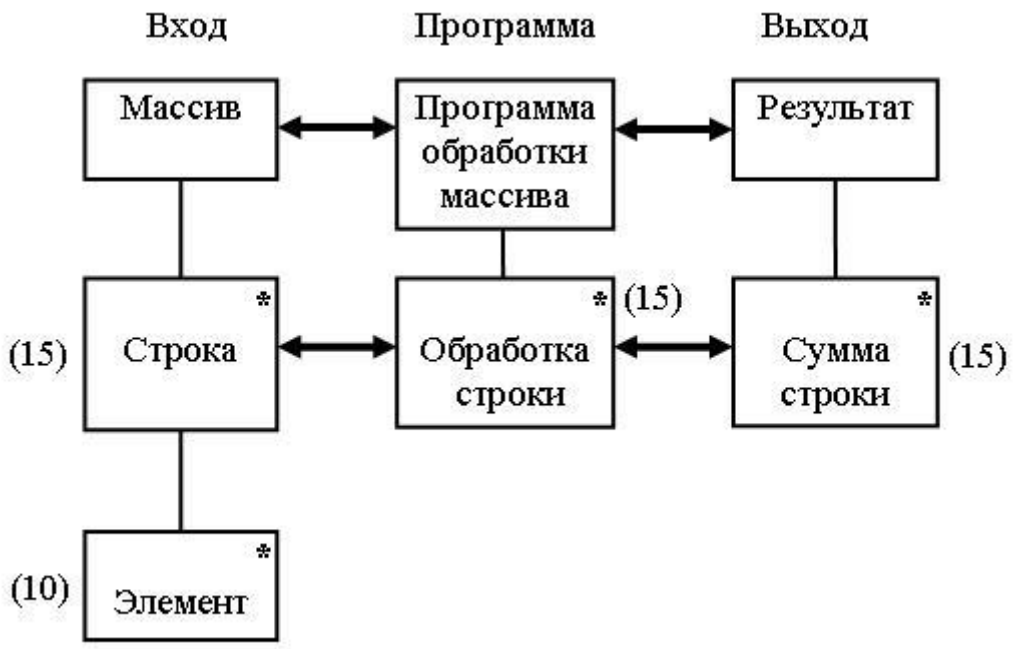

Рис. 4.24. Соответствие данных и программы

Пример 4.7

Рассмотрим более сложный пример [20]. Пусть результат предыдущего примера должен сопровождаться некоторым заголовком и некоторым завершителем. Структуры входных и выходных данных и соответствия между ними для данного случая представлены на рис. 4.25.

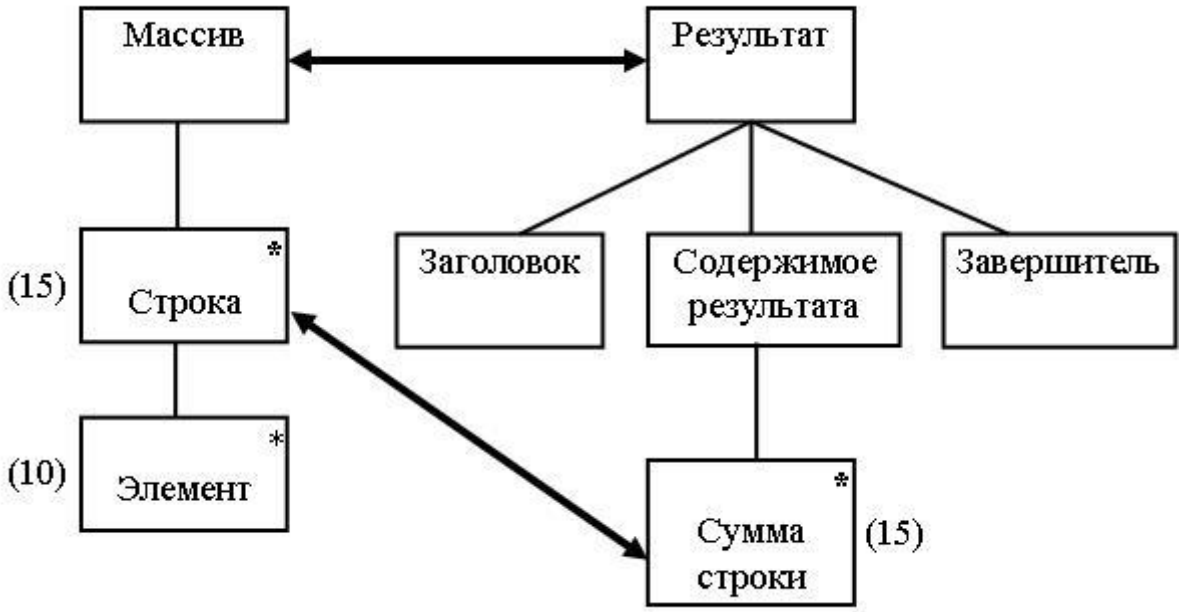

Рис. 4.25. Расширенные структуры входных и выходных данных

На первом подэтапе формирования структуры программы на основании линий соответствия формируется упрощенный вариант структуры (ядро программы, рис. 4.26).

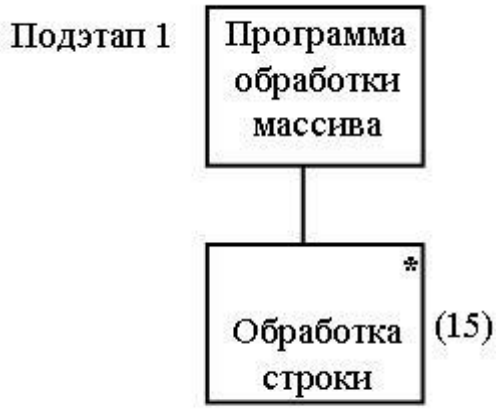

Рис. 4.26. Первый подэтап формирования структуры программы

На следующих подэтапах ядро программы расширяется. При этом рассматриваются те компоненты структур данных, к которым не подходят линии соответствия.

На втором подэтапе анализируются компоненты во входной структуре данных, к которым не подходят линии соответствия.

Таким компонентом во входной структуре данных является «Элемент». Ему не соответствует ни один компонент в структуре выходных данных.

Но «Элемент» образует подкомпоненты, составляющие компонент «Строка». А «Строка» сливается с компонентом «Сумма строки», формируя компонент «Обработка строки». Поэтому компонент «Элемент» можно трактовать как повторяемый подкомпонент компонента «Обработка строки» и переименовать его в компонент «Обработка элемента» (рис. 4.27).

Рис. 4.27. Второй подэтап формирования структуры программы

На третьем подэтапе рассматриваются компоненты в выходной структуре данных, к которым не подходят линии соответствия. Такими компонентами являются «Заголовок», «Содержимое результата», «Завершитель» (см. рис. 4.25).

По аналогии с компонентом «Элемент» каждый из этих компонентов помещается в то относительное положение в структуре программы, в котором он появляется в структурах данных. И соответственно меняется имя данных компонентов (добавляется слово «Получение», «Формирование», «Обработка» и т.п.). Рис. 4.28 иллюстрирует полученную в результате структуру программы.

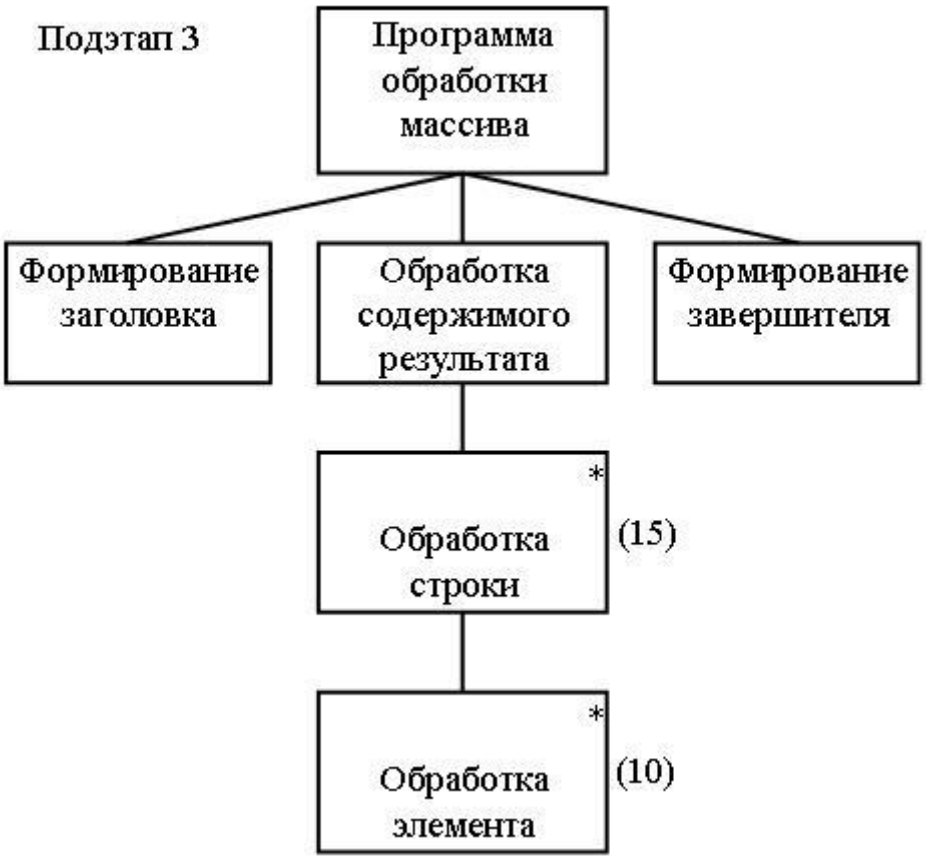

Рис. 4.28. Формирование полной структуры программы

Таким образом, выше на простых примерах показан процесс генерации структуры программы из структур входных и выходных данных на основе метода Джексона. Следующий подраздел посвящен *формализованному* описанию данного процесса.

Резюме

Для проектирования структуры программы по методу Джексона необходимо разработать структуры ее входных и выходных данных, определить взаимосвязь между данными структурами и процессом преобразования входных данных в выходные, сформировать ядро программы с учетом соответствий между входной и выходной структурами данных, поместить не нашедшие соответствия компоненты входных и выходных данных в нужные места структуры программы.

4.6.4. Этапы проектирования программного средства

Метод JSP реализуется *пятью этапами* [20]:

1. Проектирование структур входных и выходных данных.
2. Идентификация соответствий между структурами данных.
3. Проектирование структуры программы.
4. Перечисление и распределение выполняемых операций.
5. Создание текста программы на метаязыке структурированного описания.

Рассмотрим правила выполнения данных этапов на конкретном примере.

Пример 4.8

Пусть имеется входной файл, состоящий из заголовка, за которым следует совокупность дат, завершающаяся признаком окончания (завершителем). Заголовок начинается символом **H** (Heading), за которым следует содержимое заголовка, состоящее из типа файла и даты создания файла. Тип файла может иметь одно из двух значений – «Тип 1» или «Тип 2». Каждая дата представляет собой летнюю или зимнюю дату. Завершитель состоит из восьми символов **Z**.

Необходимо получить выходной файл, состоящий из заголовка и тела файла. Заголовок состоит из типа файла и даты создания файла, причем данные компоненты должны извлекаться из входного файла. Тело файла должно содержать результат подсчета количества летних дат и количества зимних дат, имеющихся во входном файле.

Этап 1. Проектирование структур входных и выходных данных

Данный этап является самым важным этапом метода JSP, так как структуры данных образуют основу формируемой структуры программы.

При создании структур данных обычно принято использовать *три вида документов*:

А. Таблицы для идентификации компонентов данных.

Б. Графическое представление структур данных.

В. Контрольные перечни вопросов для анализа структур данных.

А. Таблицы для идентификации компонентов данных

Формат таблицы для идентификации компонентов входных данных иллюстрирует табл. 4.2. Эта таблица содержит структуру входных данных, соответствующую примеру 4.8. Данная структура представлена в иерархическом виде и построена на базе конструкций Джексона. Аналогичная таблица заполняется для полного набора выходных данных (табл. 4.3).

Таблица 4.2

Структура входных компонентов данных

Ссылочный номер	Тип компонента	Ссылочный номер старшего компонента	Имя компонента данных (условие появления)	Ссылочные номера составляющих компонентов			
				Повторение	Последовательность	Выбор	Элементарная
1	Последовательность	—	Входной файл	1.2, 1.3	1.1	—	—
1.1	Последовательность	1	Заголовок	—	1.1.2	—	1.1.1
1.2	Повторение	1	Тело файла	—	—	1.2.1	—
1.3	Повторение	1	Завершитель	—	—	—	1.3.1
1.1.1	Элементарная	1.1	Символ «H»	—	—	—	—
1.1.2	Последовательность	1.1	Содержимое заголовка	—	—	1.1.2.1	1.1.2.2
1.2.1	Выбор	1.2	Дата (пока не завершитель)	—	—	—	1.2.1.1, 1.2.1.2
1.3.1.	Элементарная	1.3	«Z» (пока <= 8)	—	—	—	—
1.1.2.1	Выбор	1.1.2	Тип файла	—	—	—	1.1.2.1.1,

Ссылочный номер	Тип компонента	Ссылочный номер старшего компонента	Имя компонента данных (условие появления)	Ссылочные номера составляющих компонентов			
				Повторение	Последовательность	Выбор	Элементарная
							1.1.2.1.2
1.1.2.2	Элементарная	1.1.2	Дата создания файла	—	—	—	—
1.2.1.1	Элементарная	1.2.1	Летняя дата	—	—	—	—
1.2.1.2	Элементарная	1.2.1	Зимняя дата	—	—	—	—
1.1.2.1.1	Элементарная	1.1.2.1	Тип 1	—	—	—	—
1.1.2.1.2	Элементарная	1.1.2.1	Тип 2	—	—	—	—

Таблица 4.3

Структура выходных компонентов данных

Ссылочный номер	Тип компонента	Ссылочный номер старшего компонента	Имя компонента данных (условие появления)	Ссылочные номера составляющих компонентов			
				Повторение	Последовательность	Выбор	Элементарная
1	Последовательность	—	Выходной файл		1.1, 1.2	—	—
1.1	Последовательность	1	Заголовок	—		—	1.1.1, 1.1.2
1.2	Последователь-	1	Тело файла	—	—	—	1.2.1,

Ссылочный номер	Тип компонента	Ссылочный номер старшего компонента	Имя компонента данных (условие появления)	Ссылочные номера составляющих компонентов			
				Повторение	Последовательность	Выбор	Элементарная
	ность						1.2.2
1.1.1	Элементарная	1.1	Тип файла	—	—	—	—
1.1.2	Элементарная	1.1	Дата создания файла	—	—	—	—
1.2.1	Элементарная	1.2	Количество летних дат	—	—	—	—
1.2.2	Элементарная	1.2	Количество зимних дат	—	—	—	—

Для заполнения таблиц необходимо выполнить следующие действия.

1-й шаг. Представить совокупность всех входных и выходных данных в виде компонентов самого высокого уровня (например, «Файл», «Отчет» и т.д.).

2-й шаг. Перечислить подкомпоненты данных, которые содержит компонент из 1-го или 4-го шагов. Этот компонент должен быть либо последовательностью, либо повторением, либо выбором.

3-й шаг. Снабдить иерархическими номерами все подкомпоненты. Указать имя, тип, номер и условие появления.

4-й шаг. Для каждого подкомпонента определить, можно ли его обрабатывать при каждом появлении одним и тем же набором действий независимо от условий. Если да, то компонент можно считать элементарным и исследовать следующий подкомпонент компонента более высокого уровня. Если нет, то необходим возврат ко 2-му шагу с целью дальнейшего разложения этого подкомпонента.

Б. Графическое представление структур данных

На основе разработанных таблиц, идентифицирующих компоненты входных и выходных данных, выполняется графическое построение их структур.

На рис. 4.29 содержится результат графического представления структуры входных данных. Основу для данного представления составляет табл. 4.2.

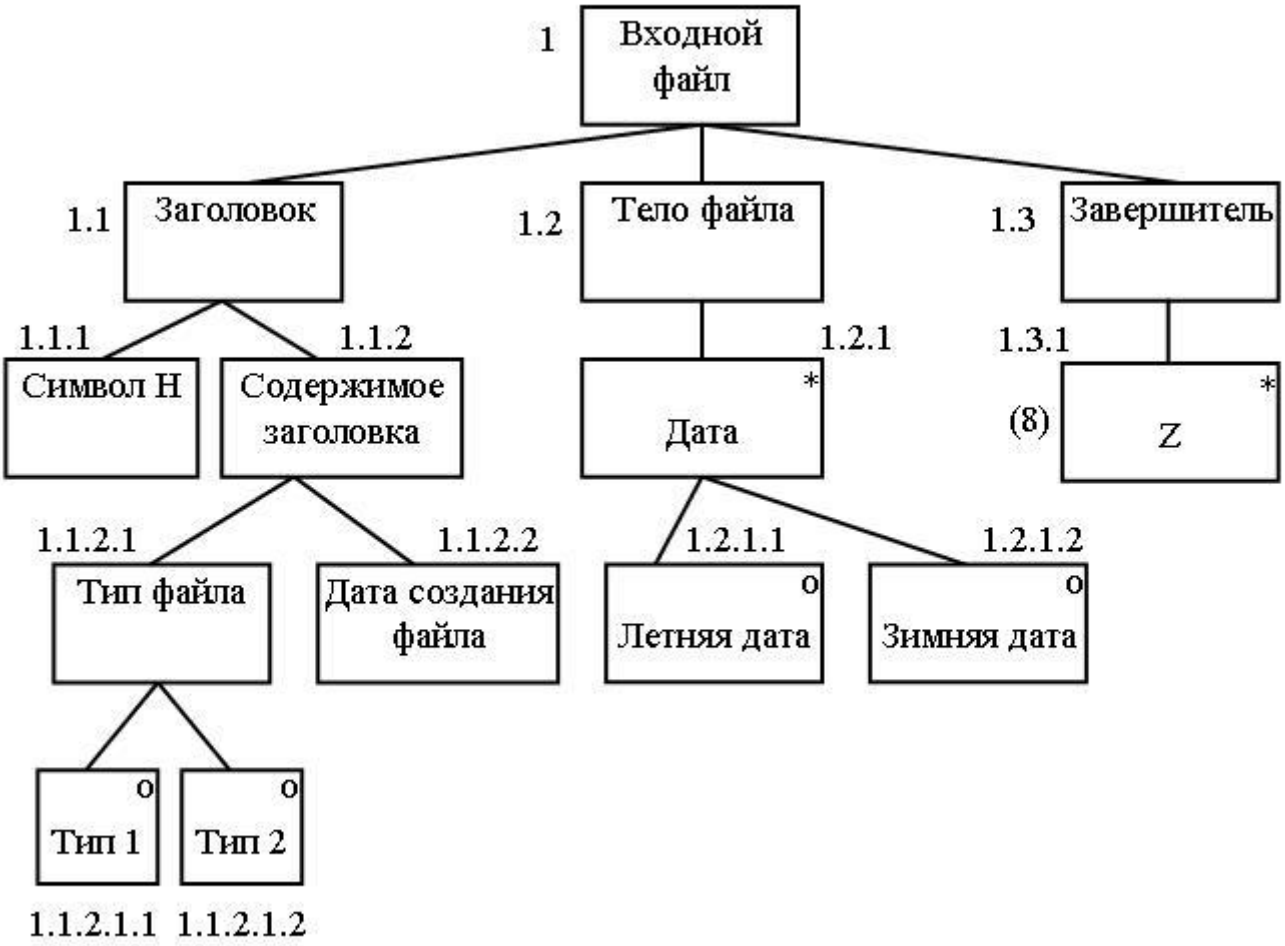

Рис. 4.29. Графическое представление структуры входного файла

Рис. 4.30 содержит графическое представление структуры выходного файла. Данное представление получено на основе табл. 4.3.

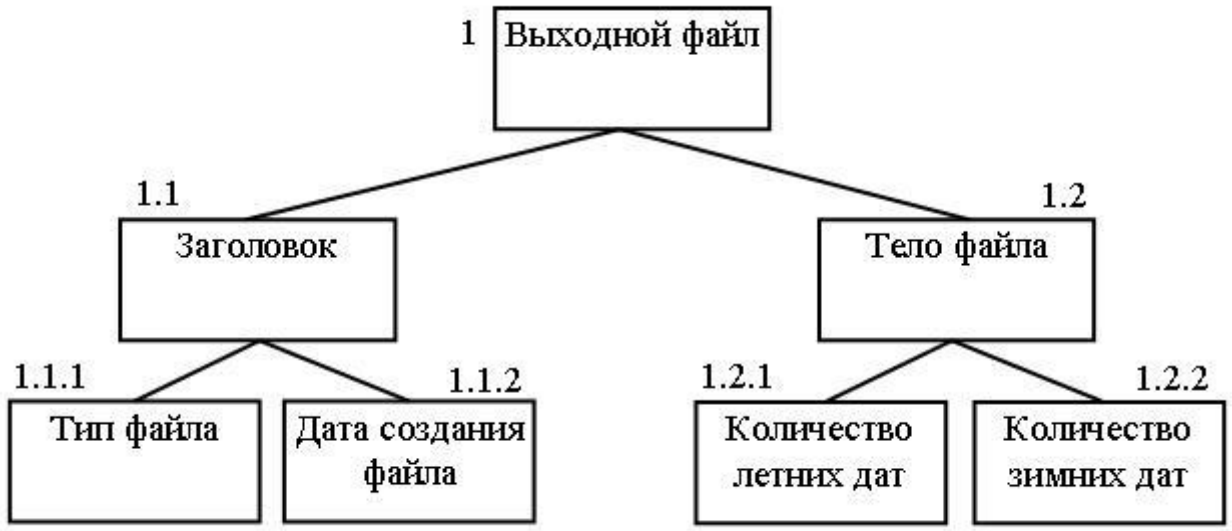

Рис. 4.30. Графическое представление структуры выходного файла

В. Контрольный перечень вопросов для анализа структур данных

Когда проектирование структур данных завершено, возникает необходимость в проверке их правильности. С этой целью используется *контрольный перечень*, представляющий собой достаточно большой стандартный набор вопросов для оценки полноты и корректности структур данных. Контрольный перечень состоит из двух частей.

Первая часть перечня предназначена для проверки полноты структуры данных. Примеры вопросов, относящихся к данной части:

- идентифицирован ли каждый вход и выход?
- представляет ли компонент данных самого высокого уровня весь вход или выход?
- идентифицирован ли каждый компонент данных именем?
- идентифицированы ли все объекты прикладного уровня (заголовки, типы строк отчетов и т.п.)?
- можно ли идентифицировать дополнительные структуры данных?

Вторая часть перечня предназначена для оценки корректности структур данных. Примеры вопросов, относящихся к данной части:

- является ли каждый подкомпонент конструкции выбора взаимоисключающим среди других соответствующих подкомпонентов?
- изображена ли структура данных сверху вниз и слева направо с точки зрения появления компонентов данных?
- является ли каждый компонент данных корректной последовательностью, выбором, повторением или элементарным компонентом?

Этап 2. Идентификация соответствий между структурами данных

Второй этап метода JSP Джексона заключается в идентификации соответствий между структурами входных и выходных данных, созданными на первом этапе. *Общие правила* установления соответствий между компонентами входной и выходной структур данных:

1) должно совпадать количество соответствующих друг другу компонентов данных;
2) соответствующие компоненты должны появляться в одинаковом порядке;
3) должна быть возможной совместная обработка каждого набора соответствующих компонентов.

Наиболее эффективный способ идентификации всех соответствий состоит в том, чтобы начать со структуры данных с наименьшим числом компонентов. В рассматриваемом примере такой структурой является структура выходных данных (сравните рис. 4.29 и 4.30). Используя три приведенных выше правила, необходимо начинать с вершины структуры и при работе продвигаться вниз.

На рис. 4.31 представлен результат идентификации соответствий рассмотренных структур входных и выходных данных.

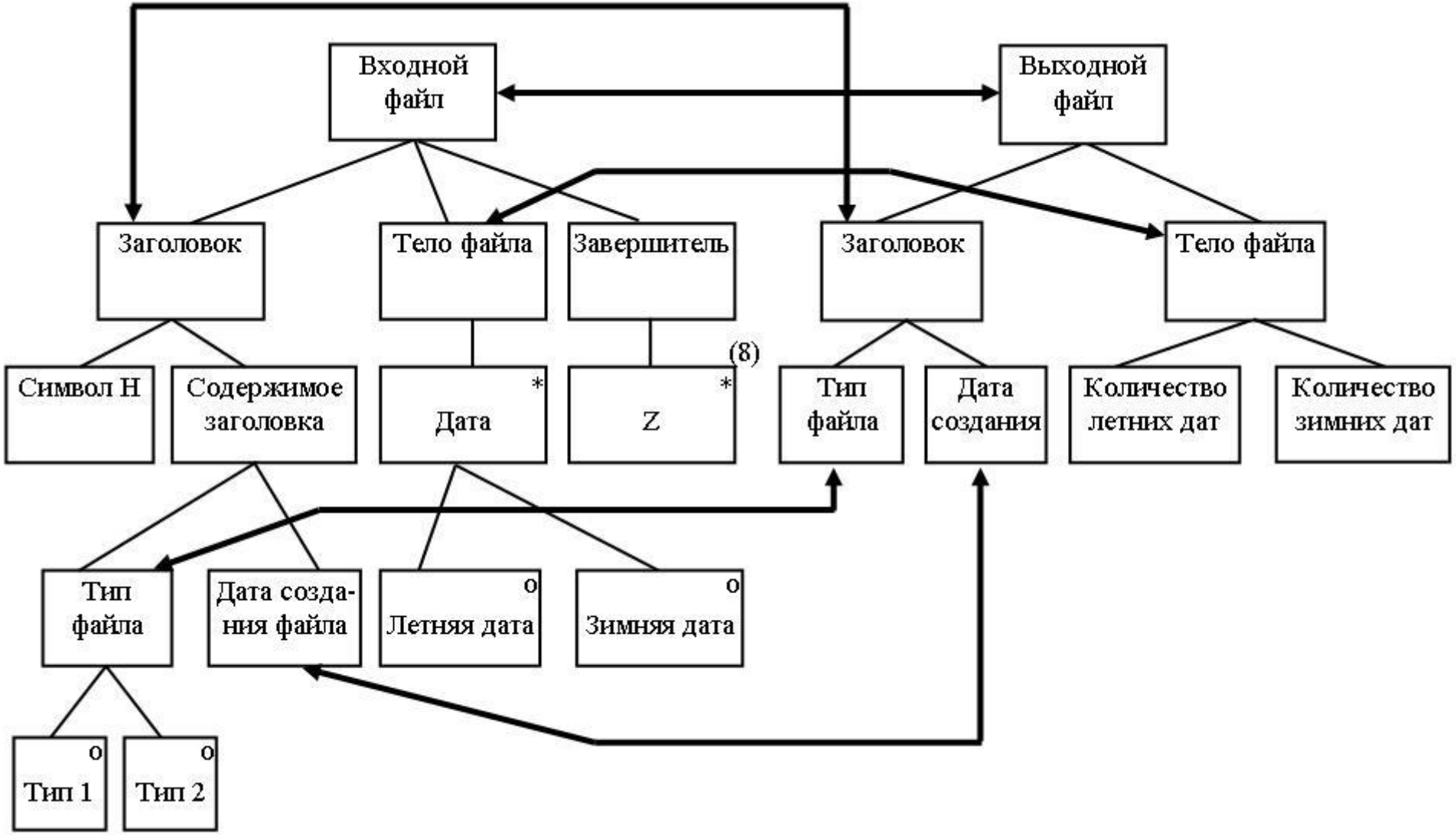

Рис. 4.31. Идентификация соответствий между входной и выходной структурами данных

Этап 3. Создание структуры программы

Можно выделить *три подэтапа* создания структур программ.

Подэтап 1. Слияние соответствующих компонентов входных и выходных данных для формирования компонентов программы.

В результате получается упрощенная структура (ядро) программы. Каждый из ее сформированных компонентов снабжается соответствующим именем (например, вида «Обработка X для создания Y»).

На рис. 4.32 показан результат данного подэтапа для рассматриваемого примера. В этом случае имеется пять линий соответствия между входными и выходными данными (см. рис. 4.31). Каждая линия заменяется соответствующим компонентом программы. Поэтому упрощенная структура программы также состоит из пяти компонентов.

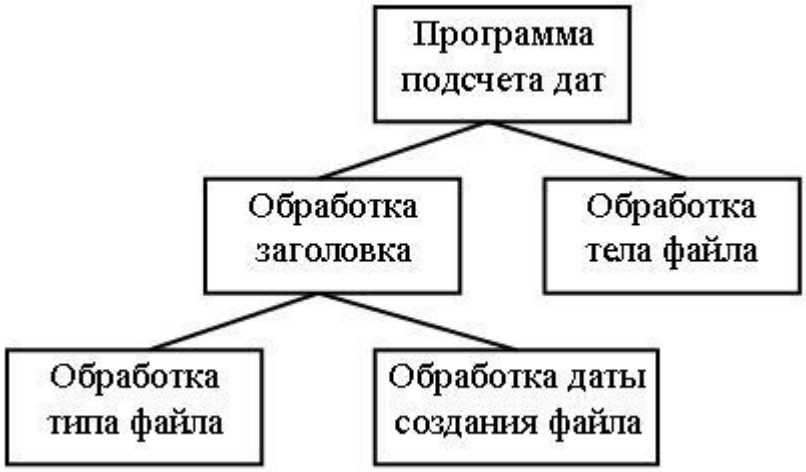

Рис. 4.32. Упрощенная структура программы

Подэтап 2. Включение не имеющих соответствия компонентов структуры входных данных в формируемую структуру программы на те же относительные иерархические места и присвоение им соответствующих имен.

На рис. 4.33 содержится результат выполнения данного подэтапа. На данном рисунке двойной линией справа выделены добавленные на данном подэтапе компоненты программы. Повторяемые компоненты входной структуры стали повторяемыми

компонентами программы. Выбираемые компоненты входной структуры стали выбираемыми компонентами программы. Однако, как будет видно при рассмотрении подэтапа 3, такое непосредственное преобразование возможно не всегда.

Подэтап 3. Включение не имеющих соответствия компонентов структуры выходных данных в формируемую структуру программы на те же относительные иерархические места и присвоение им соответствующих имен.

На рис. 4.34 представлен результат выполнения данного подэтапа.

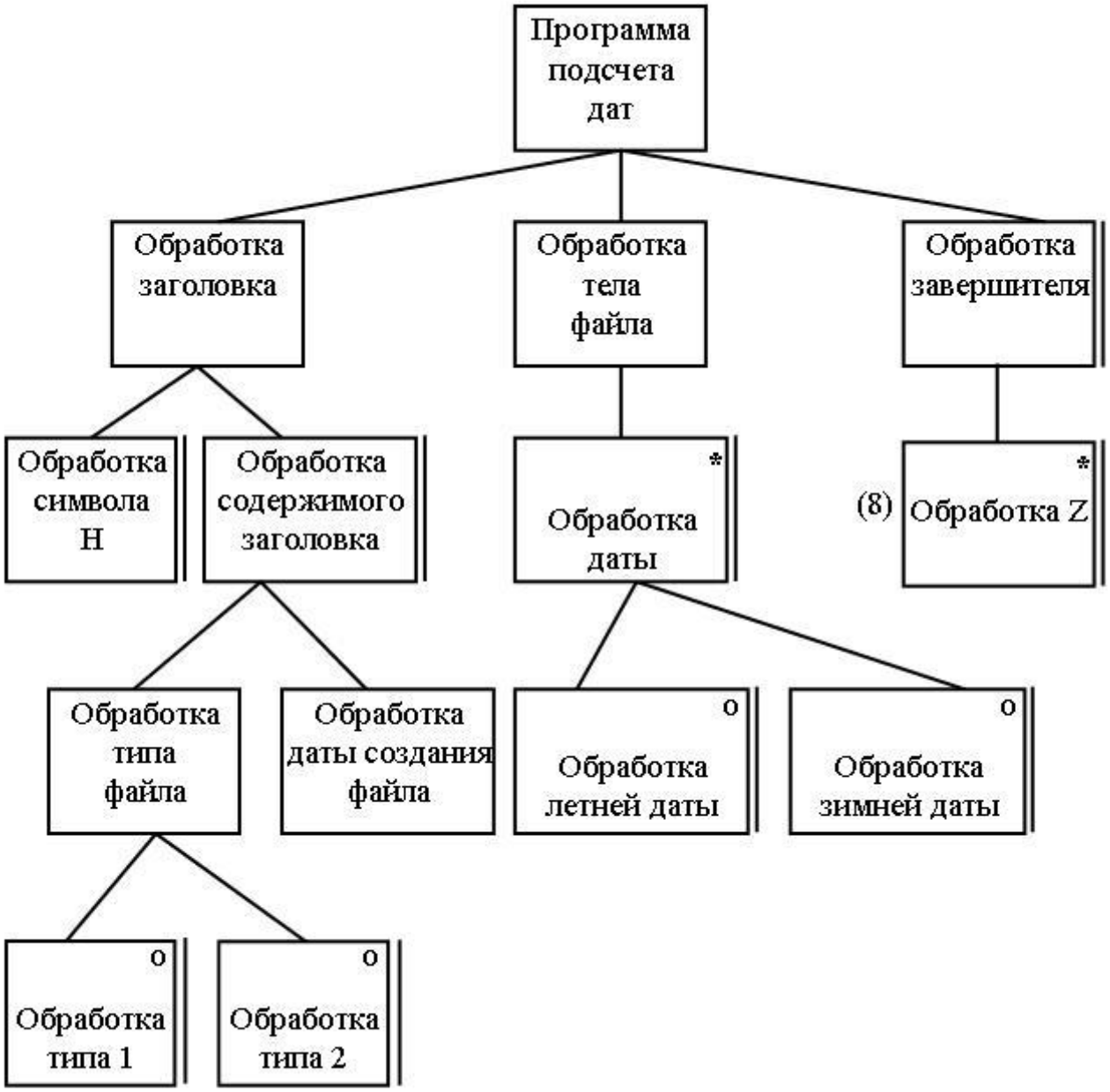

Рис. 4.33. Добавление не имеющих соответствия компонентов структуры входных данных

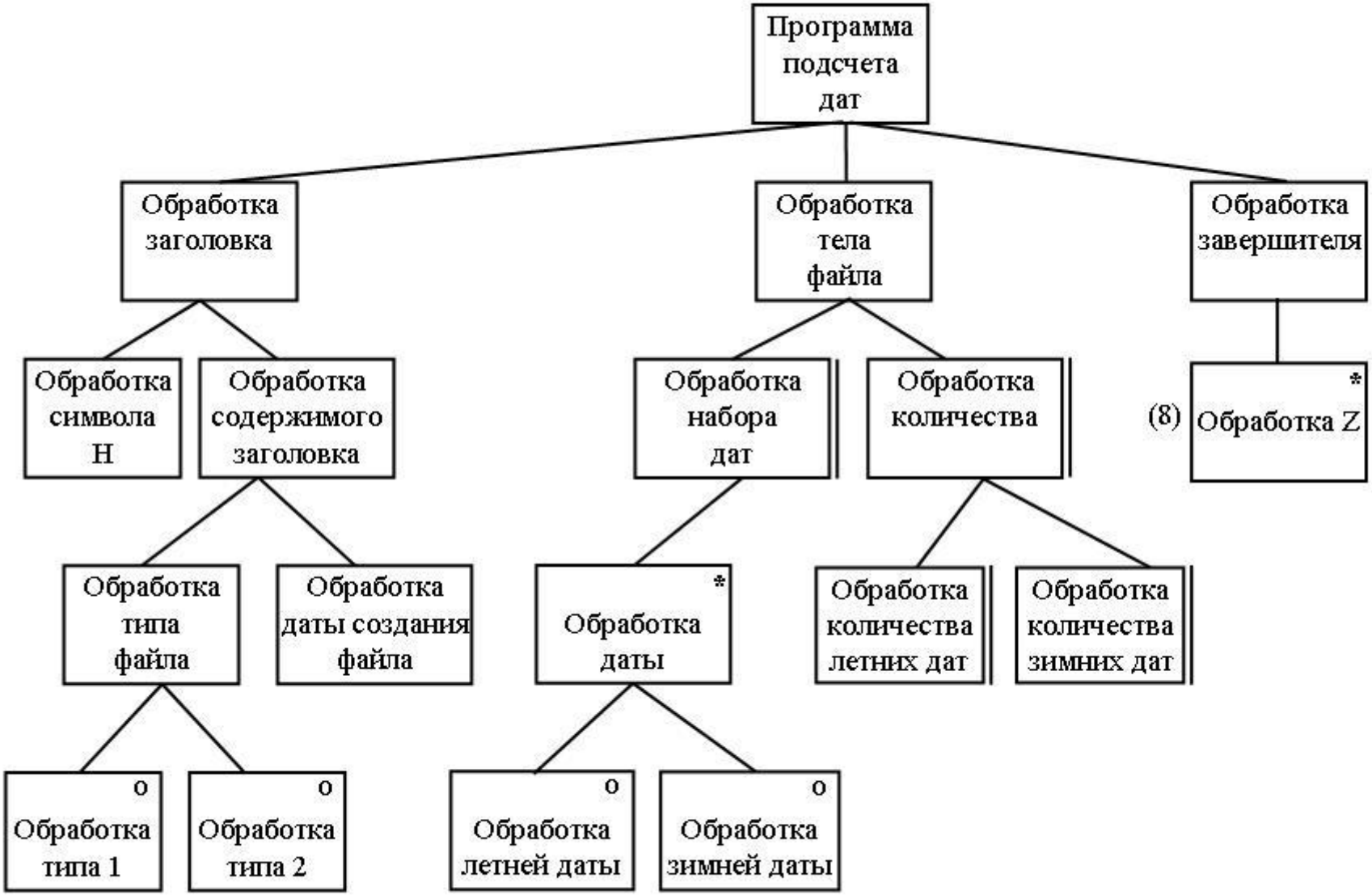

Рис. 4.34. Добавление не имеющих соответствия компонентов структуры выходных данных

На данном рисунке двойной линией справа выделены добавленные на данном подэтапе компоненты программы.

В структуре выходных данных не имеют соответствия компоненты «Количество летних дат» и «Количество зимних дат». Это подкомпоненты компонента «Тело файла» выходных данных (см. рис. 4.30 и 4.31). Следовательно, в проектируемой структуре программы компоненты, обрабатывающие «Количество летних дат» и «Количество зимних дат», должны стать подкомпонентами компонента «Обработка тела файла». Однако они не должны содержаться в подкомпоненте «Обработка даты» последнего (см. рис. 4.31 и 4.33). Поэтому их нужно поместить вне этого подкомпонента. Так как подкомпоненты «Количество летних дат» и «Количество зимних дат» могут возникнуть только после обработки всего набора дат, то соответствующие компоненты программы нужно поместить за компонентом «Обработка даты». Поэтому в это место структуры программы добавляется компонент «Обработка количества».

Добавление изменяет структуру компонента «Обработка тела файла». Ранее этот компонент был повторением с повторяемой частью «Обработка даты». Теперь он стал последовательностью. Напомним, что среди последовательных подкомпонентов не должно быть повторяемых. Поэтому в структуру компонента «Обработка тела файла» добавляется подкомпонент «Обработка набора дат».

Таким образом, в результате трех подэтапов третьего этапа сформирована управляющая структура программы (см. рис. 4.34).

На следующем этапе необходимо снабдить управляющую структуру программы выполняемыми операциями.

Этап 4. Перечисление и распределение выполняемых операций

Для составления точного списка операций, которые должна выполнять проектируемая программа, необходимо знать:

- спецификацию того, что должно делать программное средство;
- язык программирования, на котором должна быть реализована программа (для определения уровня детализации описания операций).

Для составления списка операций рекомендуется пользоваться *контрольным перечнем операций*. В состав данного перечня входят следующие *группы операций*.

I. Операции завершения, служащие для прекращения работы программы – по одной на программу (например «Стоп», «Конец» и т.п.).

II. Операции открытия и закрытия (например для файлов).

III. Операции вывода результатов (например «Писать»).

IV. Вычисления.

V. Операции ввода входных данных (например «Читать»).

VI. Управление внутренними переменными (например запоминание, инициализация и т.п.).

Ниже описан перечень операций для программы, управляющая структура которой представлена на рис. 4.34. При этом используется один из метаязыков, называемый *метаязыком структурированного изложения*.

I. *Операции завершения*

Как уже отмечалось, для проектируемой программы необходима одна операция данной группы:

1. Стоп.

II. *Операции открытия и закрытия*

В рассматриваемой программе используется два файла – входной и выходной. Поэтому из операций данной группы для проектируемой программы необходимы следующие операции:

2. Открыть входной файл.

3. Открыть выходной файл.

4. Закрыть входной файл.

5. Закрыть выходной файл.

III. *Операции вывода результатов*

Для определения операций вывода исследуется структура выходных данных (см. рис. 4.30). Каждому элементарному компоненту данной структуры соответствует своя операция вывода:

6. Писать тип файла.

7. Писать дату создания файла.

8. Писать количество летних дат.

9. Писать количество зимних дат.

IV. *Вычисления*

Вычисления требуются для формирования выходных данных. Поэтому анализируется каждая из операций вывода и определяется, какие вычисления или обработка нужны для получения соответствующих выходных данных.

В рассматриваемом примере для операций 6 и 7 вычислений не нужно – информация по условию задачи должна просто переписываться из входного файла в выходной (см. условие примера 4.8 на с. 36).

Для операций 8, 9 необходим подсчет количества летних (Кл) и зимних (Кз) дат. Поэтому появляются операции:

10. Кл := Кл + 1.

11. Кз := Кз + 1.

V. *Операции ввода входных данных*

В рассматриваемом примере используется один входной файл. Поэтому необходима одна операция ввода:

12. Читать из входного файла.

VI. *Управление внутренними переменными*

Внутренние переменные используются, как правило, в вычислениях. Переменными, участвующими в вычислениях, в рассматриваемом примере являются счетчики Кл и Кз (см. операции 10, 11). Вначале их нужно обнулить:

13. Кл := 0.

14. Кз := 0.

Итак, полный набор выполняемых операций в рассматриваемом примере содержит 14 операций.

Для размещения данных операций в нужные места структуры программы (см. рис. 4.34) по каждой из операций необходимо определить следующую информацию:

1) когда и какую часть данных обрабатывает операция (например один раз на файл, один раз на запись и т.п.);

2) где эта часть данных обрабатывается в структуре программы. В результате идентифицируется компонент программы, в который вносится анализируемая операция. Эта операция вносится как его последовательный подкомпонент;

3) на каком последовательном месте должна появиться операция в компоненте программы (слева или справа – в начале или в конце компонента).

Результат размещения выполняемых операций в структуре программы представлен на рис. 4.35. На данном рисунке двойной линией справа выделены добавленные на данном этапе компоненты программы.

Операция 1 (Стоп) встречается один раз на файл. Файл обрабатывается компонентом «Программа подсчета дат». Поэтому операция «Стоп» должна являться последовательным подкомпонентом этого компонента. Данная операция встречается в самом конце программы. Поэтому размещается на правом краю компонента «Программа подсчета дат».

По аналогии *операции 2 – 5 (Открытие и закрытие файлов)* также являются последовательными подкомпонентами компонента «Программа подсчета дат». Но операции 2, 3 (Открытие входного и выходного файлов) необходимо поместить вначале (слева) программы, а операции 4, 5 (Закрытие файлов) – справа, но перед операцией 1 (Стоп).

Операция 6 (Писать тип файла) выполняется один раз на тип файла. Тип файла обрабатывается компонентом «Обработка типа файла». Но у него уже есть два выбираемых подкомпонента «Обработка типа 1», «Обработка типа 2». Операция 6 должна появляться после любого из этих подкомпонентов. Таким образом, компонент «Обработка типа файла» должен превратиться из выбора в последовательность двух подкомпонентов: сначала выбор, затем операция 6. Поэтому в программу вводится дополнительный компонент «Обработка содержимого типа».

Очевидно, что *операция 7 (Писать дату создания файла)* является подкомпонентом компонента «Обработка даты создания файла».

Аналогично, *операции 8, 9 (Писать количество летних дат, Писать количество зимних дат)* распределяются по компонентам «Обработка количества летних дат», «Обработка количества зимних дат».

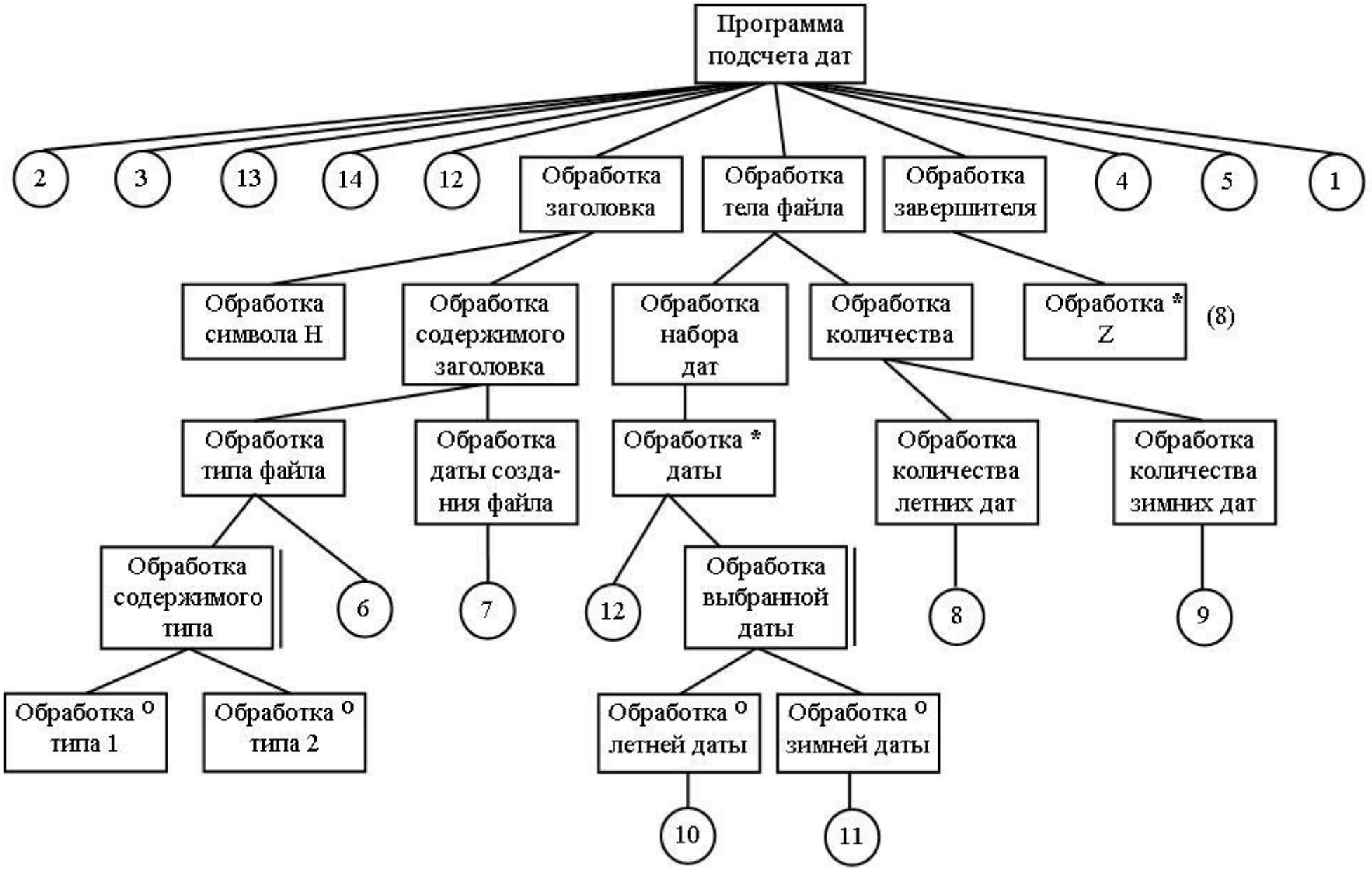

Рис. 4.35. Размещение выполняемых операций

Точно также *операции 10, 11 (Кл := Кл + 1, Кз := Кз + 1)* являются подкомпонентами компонентов «Обработка летней даты», «Обработка зимней даты».

Операции 13, 14 (Кл := 0, Кз := 0) выполняются один раз на всю программу. Поэтому они являются подкомпонентами компонента «Программа подсчета дат» и должны располагаться вначале (слева) программы.

Операция 12 (Читать из входного файла) должна появиться первый раз перед обработкой заголовка (один раз на всю программу). Поэтому операция 12 подключается к компоненту «Программа подсчета дат» перед подкомпонентом «Обработка заголовка» (слева от него). Затем операция 12 должна появляться перед обработкой каждой даты. Поэтому ее нужно подключить как последовательный подкомпонент компонента «Обработка даты». Таким образом, компонент «Обработка даты» из выбора превращается в последовательность и к нему добавляется подкомпонент «Обработка выбранной даты» (см. рис. 35).

В результате описанных выше действий в управляющей структуре проектируемой программы размещены все выполняемые операции. На следующем этапе разрабатывается текст программы. При этом используется один из метаязыков структурированного описания программ.

Этап 5. Создание текста программы на метаязыке структурированного описания

Каждая из основных конструкций, используемых в методе JSP (последовательность, выбор, повторение), может быть записана на метаязыке структурированного описания, который является одной из разновидностей словесного описания алгоритма [20].

Рис. 4.36 иллюстрирует представление конструкции последовательности при использовании структурированного описания. Данное представление состоит из списка подкомпонентов последовательности в порядке слева направо с меткой посл (последовательность) в начале и с меткой конец в конце.

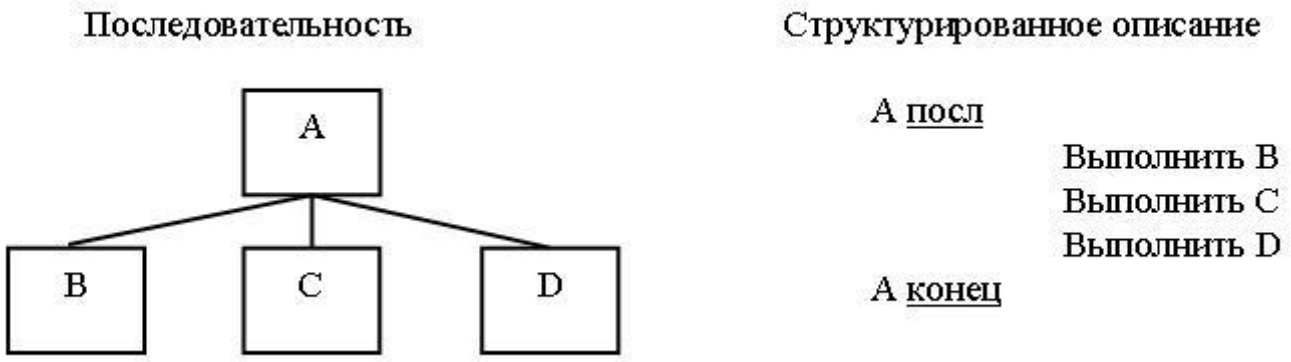

Рис. 4.36. Конструкция последовательности и ее структурированное описание

Рис. 4.37 содержит представление конструкции выбора в структурированном описании. Представление выбора начинается меткой выб (выбор) и заканчивается меткой конец. «Условие P» – это условие, при истинности которого выполняется процесс **P**. Аналогичное назначение имеют «Условие Q», «(Условие R)». Последнее условие берется в скобки, так как в программе, написанной на конкретном языке программирования, оно, как правило, не пишется (соответствует ветви «иначе»).

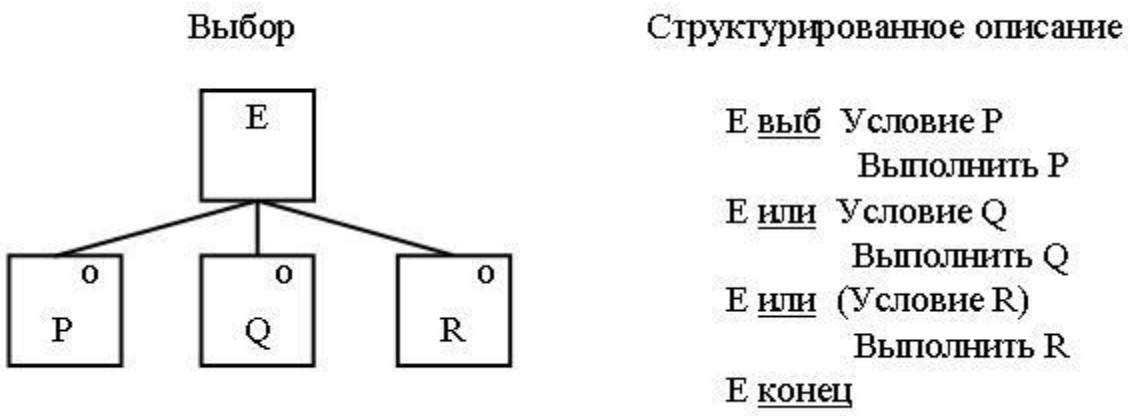

Рис. 4.37. Конструкция выбора и ее структурированное описание

Рис. 4.38 иллюстрирует представление конструкции повторения в структурированном описании. Представление начинается меткой повт (повторение) и заканчивается меткой конец. «Пока условие X» описывает условие, при котором повторяется выполнение процесса **X**.

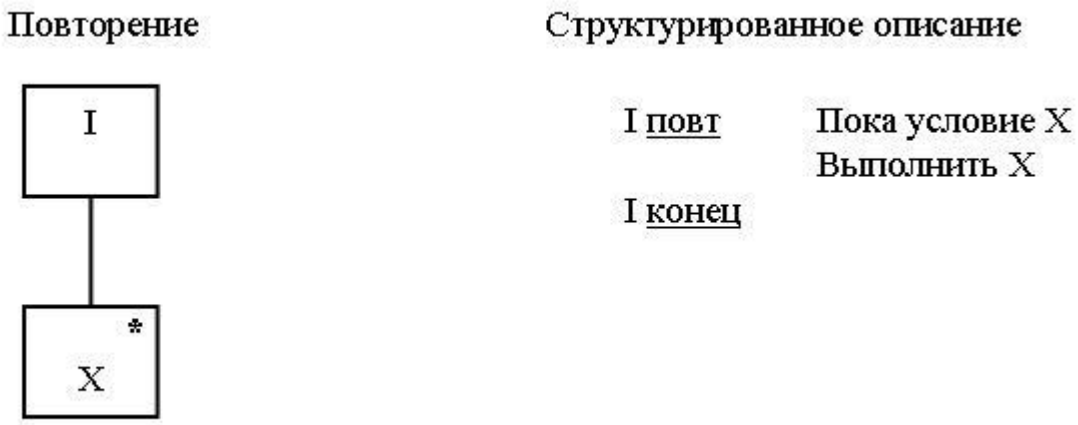

Рис. 4.38. Конструкция повторения и ее структурированное описание

Таким образом, чтобы перейти к структурированному описанию программы на базе ее структуры и размещенных выполняемых операций, необходимо определить условия для конструкций выбора и повторения.

Для обеих конструкций выбора «Обработка содержимого типа» и «Обработка выбранной даты» (см. рис. 4.35) условия непосредственно связаны с проверкой текущего содержимого обрабатываемых данных.

Для повторения «Обработка набора дат» условием повторения является отсутствие завершителя (8 символов **Z**). Таким образом, с завершителем ничего не нужно делать, кроме распознавания его появления.

Еще одним компонентом, не требующим выполнения операций, является компонент «Обработка символа Н».

Таким образом, *структурированное описание программы подсчета дат*, схема которой представлена на рис. 4.35, имеет следующий вид:

Программа подсчета дат посл

Открыть входной файл

Открыть выходной файл

Кл := 0

Кз := 0

```
Читать из входного файла
Обработка заголовка посл
    Писать тип файла
    Писать дату создания файла
Обработка заголовка конец
Обработка тела файла посл
    Обработка набора дат повт пока не завершитель ((8) Z)
        Обработка даты посл
            Читать из входного файла
            Обработка выбранной даты выб условие летней даты
                Кл := Кл + 1
            Обработка выбранной даты или (условие зимней даты)
                Кз := Кз + 1
            Обработка выбранной даты конец
        Обработка даты конец
    Обработка набора дат конец
    Обработка количеств посл
        Писать количество летних дат
        Писать количество зимних дат
    Обработка количеств конец
Обработка тела файла конец
Закрыть входной файл
Закрыть выходной файл
Стоп
Программа подсчета дат конец.
```

Из приведенного примера видно, что в структурированном описании программы присутствуют только те компоненты ее структуры (сравните с рис. 4.35), которые содержат выполняемую операцию. Отступы в тексте структурированного описания указывают уровень вложенности соответствующей части программы. Например,

«Обработка количеств посл» находится на глубине второго уровня вложенности в программной структуре.

Структурированное описание легко преобразуется в код программы, написанной на любом языке программирования.

Очевидно, что применение метода JSP достаточно сложно и громоздко даже для программ небольшой сложности. Поэтому данный метод применяется, как правило, на нижних уровнях проектирования модульных ПС, а также при разработке программных модулей и программ невысокой сложности при условии высокой степени структуризации данных. Таким образом, при проектировании модульной структуры программного средства вначале обычно используются методы нисходящего проектирования, а затем может применяться метод JSP.

Развитием метода JSP является метод JSD (Jackson System Development), также разработанный М. Джексоном.

Резюме

Предложенный М. Джексоном метод JSP реализуется с помощью пяти этапов: проектирование структур входных и выходных данных, идентификация соответствий между структурами данных, проектирование структуры программы, перечисление и распределение выполняемых операций, создание текста программы на метаязыке структурированного описания.

4.7. Оценка структурного разбиения программы на модули

В предыдущих подразделах разд. 4 рассмотрены классические методы проектирования модульных ПС.

Для оценки корректности и эффективности структурного разбиения программы на модули необходимо оценить характеристики получившихся модулей. Существуют различные меры оценки характеристик модулей. Ниже рассматриваются две из них – связность и сцепление [17].

4.7.1. Связность модуля

Связность модуля определяется как мера независимости его частей. Чем выше связность модуля, тем больше отдельные части модуля зависят друг от друга и тем лучше результат проектирования. Для количественной оценки связности используется понятие ***силы связности модуля***. Типы связности модулей и соответствующие им силы связности представлены в табл. 4.4 [17].

Модуль с *функциональной связностью* выполняет единственную функцию и реализуется обычно последовательностью операций в виде единого цикла. Если модуль спроектирован так, чтобы изолировать некоторый алгоритм, он имеет функциональную связность. Он не может быть разбит на два других модуля, имеющих связность того же типа. Примером модуля с функциональной связностью является, например, модуль сортировки дат (см. пп. 4.3.2, 4.3.3). Другой пример – модуль, который может быть разбит только на исток, преобразователь и сток, так как он выполняет единую функцию (см. п. 4.3.4).

Таблица 4.4

Типы и силы связности модулей

Связность	Сила связности
1. Функциональная	10 (сильная связность)
2. Последовательная	9
3. Коммуникативная	7
4. Процедурная	5
5. Временная	3
6. Логическая	1
7. Связность по совпадению	0 (слабая связность)

Модуль, имеющий *последовательную связность*, может быть разбит на последовательные части, выполняющие независимые функции, совместно реализующие

единую функцию. Модуль с последовательной связностью реализуется обычно как последовательность циклов или операций.

Модуль, имеющий *коммуникативную связность*, может быть разбит на несколько функционально независимых модулей, использующих общую структуру данных. Общая структура данных является основой его организации как единого модуля. Если модуль спроектирован так, чтобы упростить работу со сложной структурой данных, изолировать эту структуру, он имеет коммуникативную связность. Такой модуль предназначен для выполнения нескольких различных и независимо используемых функций над структурой данных (например запоминание некоторых данных, их поиск и редактирование).

Процедурная связность характерна для модуля, управляющие конструкции которого организованы в соответствии со схемой алгоритма, но без выделения его функциональных частей. Такая структура модуля возникает, например, при расчленении длинной программы на части в соответствии с передачами управления, но без определения каких-либо функций при выборе разделительных точек; при группировании альтернативных частей программы; если для уменьшения размеров модуль с функциональной связностью делится на два модуля (например, исходный модуль содержит объявления, подпрограммы и раздел операторов для выполнения единой функции; после его разделения один модуль содержит объявления и подпрограммы, а другой – раздел операторов).

Модуль, содержащий функционально не связанные части, необходимые в один и то же момент обработки, имеет *временную связность* (*связность по классу*). Данный тип связности имеет, например, модуль инициализации, реализующий все требуемые в начале выполнения программы функции и начальные установки. Для увеличения силы связности модуля функции инициализации целесообразно разделить между другими модулями, выполняющими обработку соответствующих переменных или файлов или включить их выполнение в управляющий модуль, но не выделять в отдельный модуль.

Если в модуле объединены операторы только по принципу их функционального подобия (например, все они предназначены для проверки правильности данных), а для настройки модуля применяется алгоритм переключения, то модуль имеет *логическую связность*. Его части ничем не связаны, а лишь похожи. Например, модуль, состоящий из разнообразных подпрограмм обработки ошибок, имеет логическую связность. Однако

если с помощью этого модуля может быть получена вся выходная информация об ошибках, то он имеет коммуникативную связность, поскольку изолирует данные об ошибках.

Модуль имеет *связность по совпадению*, если его операторы объединяются произвольным образом.

Резюме

Модули верхних уровней иерархической структуры программы должны иметь функциональную или последовательную связность. Для модулей обслуживания предпочтительнее коммуникативная связность. Если модули имеют процедурную, временную, логическую связность или связность по совпадению, это свидетельствует о недостаточно продуманном их проектировании. Необходимо добиваться функциональной связности проектируемых модулей.

4.7.2. Сцепление модулей

Сцепление модулей – это мера относительной независимости модулей. Сцепление влияет на сохранность модулей при модификациях и на понятность их исходных текстов. Слабое сцепление определяет высокий уровень независимости модулей. Независимые модули могут быть модифицированы без переделки других модулей.

Два модуля являются полностью независимыми, если в каждом из них не используется никакая информация о другом модуле. Чем больше информации о другом модуле в них используется, тем менее они независимы и тем более сцеплены. Чем очевиднее взаимодействие двух связных друг с другом модулей, тем проще определить необходимую корректировку одного модуля, зависящую от изменений, производимых в другом.

В табл. 4.5 содержатся типы сцепления модулей и соответствующие им степени сцепления [17].

Независимое сцепление возможно только в том случае, если модули не вызывают друг друга и не обрабатывают одну и ту же информацию.

Модули сцеплены *по данным*, если они имеют общие простые элементы данных, передаваемые от одного модуля к другому как параметры. В вызывающем модуле определены только имя вызываемого модуля, типы и значения переменных,

передаваемых как параметры. Вызываемый модуль может не содержать никакой информации о вызывающем. В этом случае изменения в структуре данных в одном из модулей не влияют на другой модуль.

Таблица 4.5

Типы и степени сцепления модулей

Сцепление	Степень сцепления
1. Независимое	0
2. По данным	1 (слабое сцепление)
3. По образцу	3
4. По общей области	4
5. По управлению	5
6. По внешним ссылкам	**7**
7. По кодам	9 (сильное сцепление)

Например, в вызывающем модуле определена такая структура данных, как массив. В вызываемый модуль передается в качестве параметра элемент массива. При этом изменения в структуре данных вызывающего модуля не повлияют на вызываемый модуль.

Модули со сцеплением по данным не имеют общей области данных (общих глобальных переменных).

Если модули сцеплены по данным, то по изменениям, производимым в объявленных параметрах, легко можно определить модули, на которые эти изменения повлияют.

Модули сцеплены *по образцу*, если в качестве параметров используются структуры данных (например, в качестве параметра передается массив). Недостаток такого сцепления заключается в том, что в обоих модулях должна содержаться информация о внутренней структуре данных. Если модифицируется структура данных в одном из модулей, то необходимо корректировать и другой модуль. Следовательно, увеличивается вероятность появления ошибок при разработке и сопровождении ПС.

Модули сцеплены *по общей области*, если они имеют доступ к общей области памяти (например используют общие глобальные данные). В этом случае возможностей для появления ошибок при модификации структуры данных или одного из модулей намного больше, поскольку труднее определить модули, нуждающиеся в корректировке.

Модули сцеплены *по управлению*, если какой-либо из них управляет решениями внутри другого с помощью передачи флагов, переключателей или кодов, предназначенных для выполнения функций управления. Таким образом, в одном из модулей содержится информация о внутренних функциях другого.

Например, если модуль имеет логическую связность и при его вызове используется переключатель требующейся функции, то вызываемый и вызывающий модули сцеплены по управлению.

Модули сцеплены *по внешним ссылкам*, если у одного из них есть доступ к данным другого модуля через внешнюю точку входа. Таким путем осуществляется неявное влияние на функционирование другого модуля. Сцепление этого типа возникает, например, тогда, когда внутренние процедуры одного модуля оперируют с глобальными переменными другого модуля.

Модули сцеплены *по кодам*, если коды их команд объединены друг с другом. Например, для одного из модулей доступны внутренние области другого модуля без обращения к его точкам входа, то есть модули используют общий участок памяти с командами. Это сцепление возникает, когда модули проектируются как отдельные подпрограммы, путь через которые начинается в различных точках входа, но приводит к общему сегменту кодов. Например, некоторый модуль реализует функции синуса и косинуса с учетом того, что

$$Cos(x) = Sin(\pi/2 - x).$$

Путь через точки входа Sin и Cos ведет к общему участку команд модуля.

Следует иметь в виду, что если модули косвенно обращаются друг к другу (например, связь между ними осуществляется через промежуточные модули), то между ними также существует сцепление.

Резюме

Различают независимое сцепление модулей, сцепление по данным, по образцу, по

общей области, по управлению, по внешним ссылкам, по кодам. Сцепление модулей зависит от спроектированной структуры данных и способов взаимодействия между модулями. Необходимо использовать простые параметры и не применять общих областей памяти.