3.1. Задачи и цели тестирования программного кода

Тестирование программного кода - процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под дефектом здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям). Неожиданное поведение системы может приводить к сбоям в ее работе и отказам, в этом случае говорят о существенных дефектах программного кода. Некоторые дефекты вызывают незначительные проблемы, не нарушающие процесс функционирования системы, но несколько затрудняющие работу с ней. В этом случае говорят о средних или малозначительных дефектах.

Задача тестирования при таком подходе - *определение* условий, при которых проявляются дефекты системы, и *протоколирование* этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных участков программного кода и никогда не входит исправление дефектов - это задача отладки, которая выполняется по результатам тестирования системы.

Цель применения процедуры тестирования программного кода - *минимизация* количества дефектов (в особенности существенных) в конечном продукте. Тестирование само по себе не может гарантировать полного отсутствия дефектов в программном коде системы. Однако, в сочетании с процессами верификации и валидации, направленными на устранение противоречивости и неполноты проектной документации (в частности - требований на систему), грамотно организованное тестирование дает гарантию того, что система удовлетворяет требованиям и ведет себя в соответствии с ними во всех предусмотренных ситуациях.

При разработке систем повышенной надежности, например, авиационных, гарантии надежности достигаются с помощью четкой организации процесса тестирования, определения его связи с остальными процессами жизненного *цикла*, введения количественных характеристик, позволяющих оценивать успешность тестирования. При этом чем выше требования к надежности системы (ее уровень критичности), тем более жесткие требования предъявляются.

Таким образом, в первую *очередь* мы рассматриваем не конкретные результаты тестирования конкретной системы, а общую организацию процесса тестирования, используя подход "хорошо организованный процесс дает качественный результат". Такой подход является общим для многих международных и отраслевых стандартов качества, о которых более подробно будет рассказано в конце данного курса. Качество разрабатываемой системы при таком подходе является следствием организованного процесса разработки и тестирования, а не самостоятельным неуправляемым результатом.

Поскольку современные программные системы имеют весьма значительные размеры, при тестировании их программного кода используется метод функциональной декомпозиции. Система разбивается на отдельные модули (классы, пространства имен и т.п.), имеющие определенную требованиями функциональность и интерфейсы. После этого по отдельности тестируется каждый *модуль* - выполняется *модульное тестирование*. Затем происходит *сборка* отдельных модулей в более крупные конфигурации - выполняется *интеграционное тестирование*, и наконец, тестируется система в целом - выполняется *системное тестирование*.

С точки зрения программного кода, модульное, интеграционное и *системное тестирование* имеют много общего, поэтому пока основное внимание будет уделено модульному тестированию, особенности интеграционного и системного тестирования будут рассмотрены позднее.

В ходе модульного тестирования каждый *модуль* тестируется как на соответствие требованиям, так и на отсутствие проблемных участков программного кода, которые могут вызвать отказы и сбои в работе системы. Как правило, модули не работают вне системы - они принимают данные от других модулей, перерабатывают их и передают дальше. Для того, чтобы с одной стороны, изолировать *модуль* от системы и исключить влияние потенциальных ошибок системы, а с другой стороны - обеспечить *модуль* всеми необходимыми данными, используется тестовое окружение.

Задача тестового окружения - создать среду выполнения для модуля, эмулировать все внешние интерфейсы, к которым обращается *модуль*. Об особенностях организации тестового окружения пойдет речь далее в данной лекции.

Типичная процедура тестирования состоит в подготовке и выполнении тестовых примеров (также называемых просто тестами). Каждый тестовый пример проверяет одну "ситуацию" в поведении модуля и состоит из списка значений, передаваемых на вход модуля, описания запуска и выполнения переработки данных - тестового сценария и списка значений, которые ожидаются на выходе модуля в случае его корректного поведения. Тестовые сценарии составляются таким образом, чтобы исключить обращения к внутренним данным модуля, все взаимодействие должно происходить только через его внешние интерфейсы.

Выполнение тестового примера поддерживается тестовым окружением, которое включает в себя программную реализацию тестового сценария. Выполнение начинается с передачи модулю входных данных и запуска сценария. Реальные *выходные данные*, полученные от модуля в результате выполнения сценария, сохраняются и сравниваются с ожидаемыми. В случае их совпадения тест считается пройденным, в противном случае - не пройденным. Каждый не пройденный тест указывает на дефект либо в тестируемом модуле, либо в тестовом окружении, либо в описании теста.

Совокупность описаний тестовых примеров составляет тест-план - основной документ, определяющий процедуру тестирования программного модуля. Тест-план задает не только сами тестовые примеры, но и порядок их следования, который также может быть важен. Структура и особенности тест-планов, а также проблемы, связанные с порядком следования тестовых примеров, будут рассмотрены в следующих лекциях.

При тестировании часто бывает необходимо учитывать не только требования к системе, но и структуру программного кода тестируемого модуля. В этом случае тесты составляются таким образом, чтобы детектировать типичные ошибки программистов, вызванные неверной интерпретацией требований. Применяются проверки граничных условий, проверки классов эквивалентности. Отсутствие в системе возможностей, не заданных требованиями, гарантировано различными оценками покрытия программного кода тестами, т.е. оценками того, какой *процент* тех или иных языковых конструкций выполнен в результате выполнения всех тестовых примеров. Обо всем этом пойдет речь в завершение рассмотрения процесса тестирования программного кода.

3.2. Методы тестирования

3.2.1. Черный ящик

Основная идея в тестировании системы как черного ящика состоит в том, что все материалы, которые доступны тестировщику, - требования на систему, описывающие ее поведение, и сама система, работать с которой он может, только подавая на ее входы некоторые внешние воздействия и наблюдая на выходах некоторый результат. Все внутренние особенности реализации системы скрыты от тестировщика, - таким образом, система представляет собой "черный ящик", правильность поведения которого по отношению к требованиям и предстоит проверить.

С точки зрения программного кода черный ящик может представлять с собой набор классов (или модулей) с известными внешними интерфейсами, но недоступными исходными текстами.

Основная задача тестировщика для данного метода тестирования состоит в последовательной проверке соответствия поведения системы требованиям. Кроме того, тестировщик должен проверить работу системы в критических ситуациях - что происходит в случае подачи неверных входных значений. В идеальной ситуации все варианты критических ситуаций должны быть описаны в требованиях на систему и тестировщику остается только придумывать конкретные проверки этих требований. Однако в реальности в результате тестирования обычно выявляется два типа проблем системы.

1. Несоответствие поведения системы требованиям
2. Неадекватное поведение системы в ситуациях, не предусмотренных требованиями.

Отчеты об обоих типах проблем документируются и передаются разработчикам. При этом проблемы первого типа обычно вызывают изменение программного кода, гораздо реже - изменение требований. Изменение требований в данном случае может потребоваться из-за их противоречивости (несколько разных требований описывают разные модели поведения системы в одной и той же самой ситуации) или некорректности (требования не соответствуют действительности).

Проблемы второго типа однозначно требуют изменения требований ввиду их неполноты - в требованиях явно пропущена ситуация, приводящая к неадекватному поведению системы. При этом под неадекватным поведением может пониматься как полный крах системы, так и вообще любое поведение, не описанное в требованиях.

Тестирование черного ящика называют также тестированием по требованиям, т.к. это единственный источник информации для построения тест-плана.

3.2.2. Стеклянный (белый) ящик

При тестировании системы как стеклянного ящика тестировщик имеет доступ не только к требованиям к системе, ее входам и выходам, но и к ее внутренней структуре - видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и определять тем самым, на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, не покрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы - часто одна проблема нейтрализует другую, и они никогда не возникают одновременно.

3.2.3. Тестирование моделей

Тестирование моделей находится несколько в стороне от классических методов верификации программного обеспечения. Причина прежде всего в том, что объект тестирования - не сама система, а ее модель, спроектированная формальными средствами. Если оставить в стороне вопросы проверки корректности и применимости самой модели (считается, что ее корректность и соответствие исходной системе могут быть доказаны формальными средствами), то тестировщик получает в свое распоряжение достаточно мощный инструмент анализа общей целостности системы. На модели можно создать такие ситуации, которые невозможно создать в тестовой лаборатории для реальной системы. Работая с моделью программного кода системы, можно анализировать его свойства и такие параметры системы, как оптимальность алгоритмов или ее устойчивость.

Однако тестирование моделей не получило широкого распространения именно из-за трудностей, возникающих при разработке формального описания поведения системы. Одно из немногих исключений - системы связи, алгоритмический и математический аппарат которых достаточно хорошо проработан.

3.2.4. Анализ программного кода (инспекции)

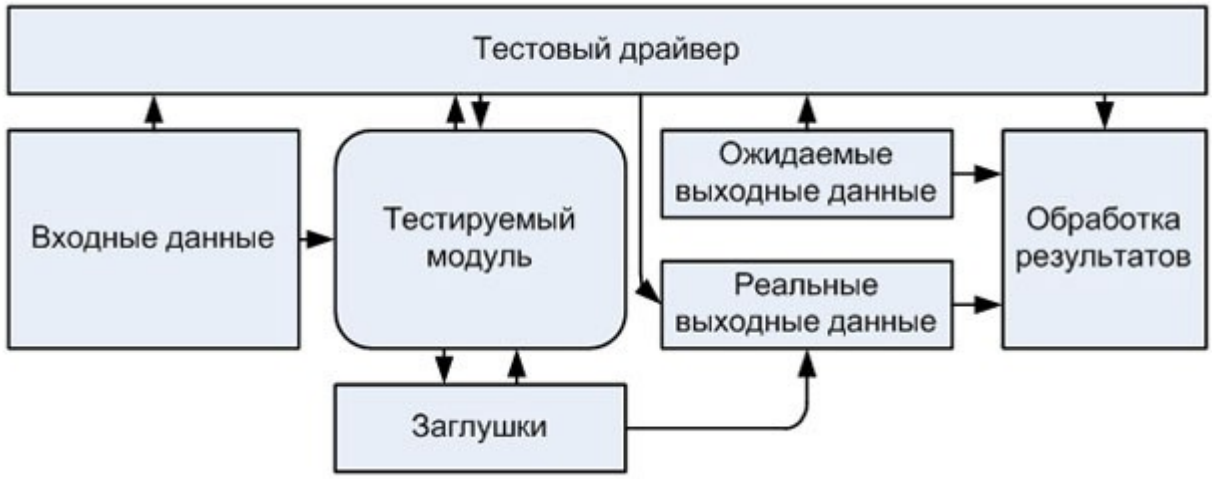
Во многих ситуациях тестирование поведения системы в целом невозможно - отдельные участки программного кода могут никогда не выполняться, при этом они будут покрыты требованиями. Примером таких участков кода могут служить обработчики исключительных ситуаций. Если, например, два модуля передают друг другу числовые значения и функции проверки корректности значений работают в обоих модулях, то функция проверки модуля-приемника никогда не будет активизирована, т.к. все ошибочные значения будут отсечены еще в передатчике.

В этом случае выполняется ручной анализ программного кода на корректность, называемый также просмотрами или инспекциями кода. Если в результате инспекции выявляются проблемные участки, то информация об этом передается разработчикам для исправления наравне с результатами обычных тестов.

3.3. Тестовое окружение

Основной объем тестирования практически любой сложной системы обычно выполняется в автоматическом режиме. Кроме того, тестируемая система обычно разбивается на отдельные модули, каждый из которых тестируется вначале отдельно от других, затем в комплексе.

Это означает, что для выполнения тестирования необходимо создать некоторую среду, которая обеспечит *запуск* и выполнение тестируемого модуля, передаст ему *входные данные*, соберет реальные *выходные данные*, полученные в результате работы системы на заданных входных данных. После этого среда должна сравнить реальные *выходные данные* с ожидаемыми и на основании данного сравнения сделать *вывод* о соответствии поведения модуля заданному (Рис 3.1).



увеличить изображение
Рис. 3.1. Обобщенная схема среды тестирования

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый *модуль*.

Поскольку тестовое окружение само является программой (причем зачастую реализованной не на том языке программирования, на котором написана система), оно само должно быть протестировано. Целью тестирования тестового окружения является *доказательство* того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

3.3.1. Драйверы и заглушки

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов - драйвера, который обеспечивает запуск и выполнение тестируемого модуля, и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, дабы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования. Так, драйвер может выполнять следующие функции:

1. Вызов тестируемого модуля
2. 1 + передача в тестируемый модуль входных значений и прием результатов
3. 2 + вывод выходных значений
4. 3 + протоколирование процесса тестирования и ключевых точек программы

Заглушки могут выполнять следующие функции:

1. Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля)
2. Выводить сообщения о том, что заглушка была вызвана

3. 1 + выводить сообщения со значениями параметров, переданных в функцию
4. 2 + возвращать значение, заранее заданное во входных параметрах теста
5. 3 + выводить значение, заранее заданное во входных параметрах теста
6. 3 + принимать от тестируемого ПО значения и передавать их в драйвер [10].

Для тестирования программного кода, написанного на процедурном языке программирования, используются драйверы, представляющие собой программу с точкой входа (например, функцией main()), функциями запуска тестируемого модуля и функциями сбора результатов. Обычно драйвер имеет как минимум одну функцию - точку входа, которой передается управление при его вызове.

Функции-заглушки могут помещаться в тот же файл исходного кода, что и основной текст драйвера. Имена и параметры заглушек должны совпадать с именами и параметрами "заглушаемых" функций реальной системы. Это требование важно не столько с точки зрения корректной сборки системы (при сборке тестового драйвера и тестируемого ПО может использоваться приведение типов), сколько для того, чтобы максимально точно моделировать поведение реальной системы по передаче данных. Так, например, если в реальной системе присутствует функция вычисления квадратного корня

```
double sqrt(double value);
```

то, с точки зрения сборки системы, вместо типа double может использоваться и float, но снижение точности может вызвать непредсказуемые результаты в тестируемом модуле.

В качестве примера драйвера и заглушек рассмотрим реализацию стека на языке С, причем значения, помещаемые в стек, хранятся не в оперативной памяти, а помещаются в ППЗУ при помощи отдельного модуля, содержащего две функции - записи данных в ППЗУ по адресу и чтения данных по адресу.

Формат этих функций следующий:

```
void NV_Read(char *destination, long length, long offset);
void NV_Write(char *source, long length, long offset);
```

Здесь destination - адрес области памяти, в которую записывается значение, считанное из ППЗУ, source - адрес области памяти, из которой записывается значение в ППЗУ, length - длина записываемой области памяти, offset - смещение относительно начального адреса ППЗУ.

Реализация стека с использованием этих функций выглядит следующим образом:

```
long currentOffset;

void initStack()
{
 currentOffset=0;
}

void push(int value)
{
 NV_Write((int*)&value,sizeof(int),currentOffset);
 currentOffset+=sizeof(int);
```

```
}

int pop()
{
 int value;
 if (currentOffset>0)
 {
  NV_Read((int*)&value,sizeof(int),currentOffset;
  currentOffset-=sizeof(int);
 }
}
```

При выполнении этого кода на реальной системе происходит запись в ППЗУ, однако, если мы хотим протестировать только реализацию стека, изолировав ее от реализации модуля работы с ППЗУ, необходимо использовать заглушки вместо реальных функций. Для имитации работы ППЗУ можно выделить достаточно большой участок оперативной памяти, в которой и будет производиться запись данных, получаемых заглушкой.

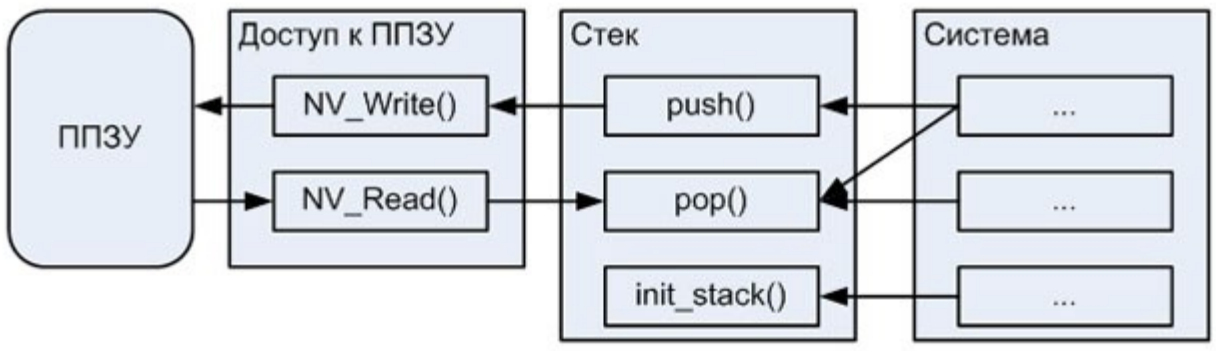
Заглушки для функций могут выглядеть следующим образом:

```
char nvrom[1024];

void NV_Read(char *destination, long length, long offset)
{
 printf("NV_Read called\n");
 memcpy(destination, nvrom+offset, length);
}
void NV_Write(char *source, long length, long offset);
{
 printf("NV_Write called\n");
 memcpy(nvrom+offset, source, length);
}
```

Каждая из заглушек выводит трассировочное сообщение и перемещает переданное значение в память, эмулирующую ППЗУ (функция NV_Write), или возвращает по ссылке значение, которое хранится в памяти, эмулирующей ППЗУ (функция NV_Read).

Схема взаимодействия тестируемого ПО (функций работы со стеком) с реальным окружением (основной частью системы и модулем работы с ППЗУ) и тестовым окружением (драйвером и заглушками функций работы с ППЗУ) показана на Рис 3.2 и Рис 3.3.



увеличить изображение

Рис. 3.2. Схема взаимодействия частей реальной системы

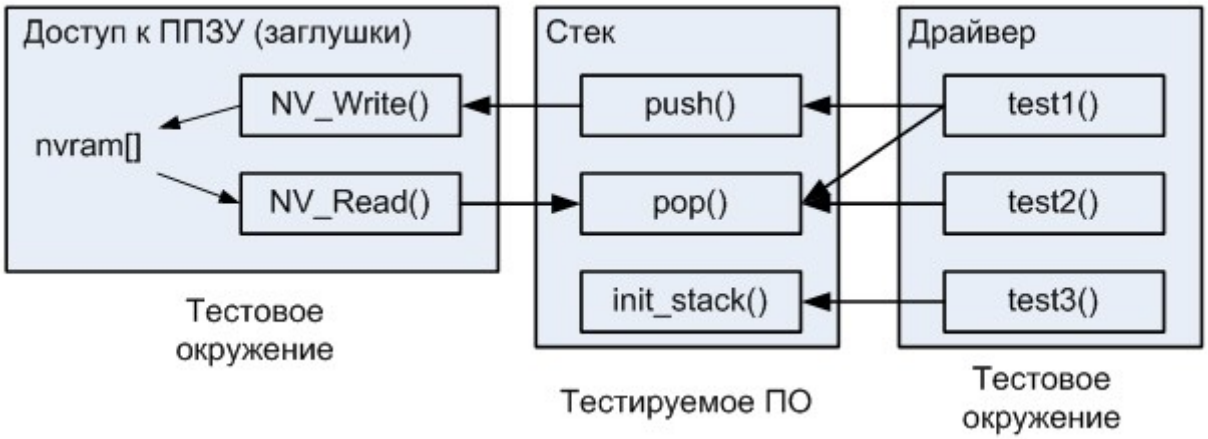


Рис. 3.3. Схема взаимодействия тестового окружения и тестируемого ПО

3.3.2. Тестовые классы

Тестовое окружение для объектно-ориентированного ПО выполняет те же самые функции, что и для структурных программ (на процедурных языках). Однако, оно имеет некоторые особенности, связанные с применением наследования и инкапсуляции.

Если при тестировании структурных программ минимальным тестируемым объектом является функция, то в объектно-ориентированным ПО минимальным объектом является класс. При применении принципа инкапсуляции все внутренние данные класса и некоторая часть его методов недоступна извне. В этом случае тестировщик лишен возможности обращаться в своих тестах к данным класса и произвольным образом вызывать методы; единственное, что ему доступно - вызывать методы внешнего интерфейса класса.

Существует несколько подходов к тестированию классов, каждый из них накладывает свои ограничения на структуру драйвера и заглушек.

1. Драйвер создает один или больше объектов тестируемого класса, все обращения к объектам происходят только с использованием их внешнего интерфейса. Текст драйвера в этом случае представляет собой т.н. тестирующий класс, который содержит по одному методу для каждого тестового примера. Процесс тестирования заключается в последовательном вызове этих методов. Вместо заглушек в состав тестового окружения входит программный код реальной системы, соответственно, отсутствует изоляция тестируемого класса. Однако, именно такой подход к тестированию принят сейчас в большинстве методологий и сред разработки. Его классическое название - *unit testing* (тестирование модулей), более подробно он будет рассматриваться позднее.
2. Аналогично предыдущему подходу, но для всех классов, которые использует тестируемый класс, создаются заглушки
3. Программный код тестируемого класса модифицируется таким образом, чтобы открыть доступ ко всем его свойствам и методам. Строение тестового окружения в этом случае полностью аналогично окружению для тестирования структурных программ.

4. Используются специальные средства доступа к закрытым данным и методам класса на уровне объектного или исполняемого кода - скрипты отладчика или accessors в Visual Studio.

Основное достоинство первых двух методов: при их использовании класс работает точно таким же образом, как в реальной системе. Однако в этом случае нельзя гарантировать, что в процессе тестирования будет выполнен весь программный код класса и не останется непротестированных методов.

Основной недостаток 3-го метода: после изменения исходных текстов тестируемого модуля нельзя дать гарантии того, что класс будет вести себя таким же образом, как и исходный. В частности это связано с тем, что изменение защиты данных класса влияет на наследование данных и методов другими классами.

Тестирование наследования - отдельная сложная задача в объектно-ориентированных системах. После того, как протестирован базовый класс, необходимо тестировать классы-потомки. Однако, для базового класса нельзя создавать заглушки, т.к. в этом случае можно пропустить возможные проблемы полиморфизма. Если класс-потомок использует методы базового класса для обработки собственных данных, необходимо убедиться в том, что эти методы работают.

Таким образом, иерархия классов может тестироваться сверху вниз, начиная от базового класса. Тестовое окружение при этом может меняться для каждой тестируемой конфигурации классов.

3.3.3. Генераторы сигналов (событийно-управляемый код)

Значительная часть сложных программ в настоящее время использует ту или иную форму межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла следующие этапы [11].

1. Монолитные программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных. При запуске таких программ образуется один процесс, который выполняет всю необходимую работу.
2. Модульные программы, которые состоят из отдельных программных модулей с четко определенными интерфейсами вызовов. Объединение модулей в программу может происходить либо на этапе сборки исполняемого файла (статическая сборка или static linking), либо на этапе выполнения программы (динамическая сборка или dynamic linking). Преимущество модульных программ заключается в достижении некоторого уровня универсальности - один модуль может быть заменен другим. Однако, модульная программа все равно представляет собой один процесс, а данные, необходимые для решения задачи, передаются внутри процесса как параметры функций.
3. Программы, использующие межпроцессное взаимодействие. Такие программы образуют программный комплекс, предназначенный для решения общей задачи. Каждая запущенная программа образует один или более процессов. Каждый из процессов либо использует для решения задачи свои собственные данные и обменивается с другими процессами только результатом своей работы, либо работает с общей областью данных, разделяемых между разными процессами. Для решения

особо сложных задач процессы могут быть запущены на разных физических компьютерах и взаимодействовать через сеть. Преимущество использования межпроцессного взаимодействия заключается в еще большей универсальности - взаимодействующие процессы могут быть заменены независимо друг от друга при сохранении интерфейса взаимодействия. Другое преимущество состоит в том, что вычислительная нагрузка распределяется между процессами. Это позволяет операционной системе управлять приоритетами выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам.

При выполнении многих процессов, решающих общую задачу, используются несколько типичных механизмов взаимодействия между ними, направленных на решение следующих задач:

- передача данных от одного процесса к другому;
- совместное использование одних и тех же данных несколькими процессами;
- извещения об изменении состояния процессов.

Во всех этих случаях типичная структура каждого процесса представляет собой конечный автомат с набором состояний и переходов между ними. Находясь в определенном состоянии, процесс выполняет обработку данных, при переходе между состояниями - пересылает данные другим процессам или принимает данные от них [12].

Для моделирования конечных автоматов используются stateflow [13] или SDL-диаграммы [13], акцент в которых делается соответственно на условиях перехода между состояниями и пересылаемыми данными.

Так, на Рис 3.4 показана схема процесса приема/передачи данных. Закругленными прямоугольниками указаны состояния процесса, тонкими стрелками - переходы между состояниями, большими стрелками - пересылаемые данные. Находясь в состоянии "Старт", процесс посылает во внешний мир (или процессу, с которым он обменивается данными) сообщение о своей готовности к началу сеанса передачи данных. После получения от второго процесса подтверждения о готовности начинается сеанс обмена данными. В случае поступления сообщения о конце данных происходит завершение сеанса и переход в стартовое состояние. В случае поступления неверных данных (например, неправильного формата или с неверной контрольной суммой) процесс переходит в состояние "Ошибка", выйти из которого возможно только завершением и перезапуском процесса.

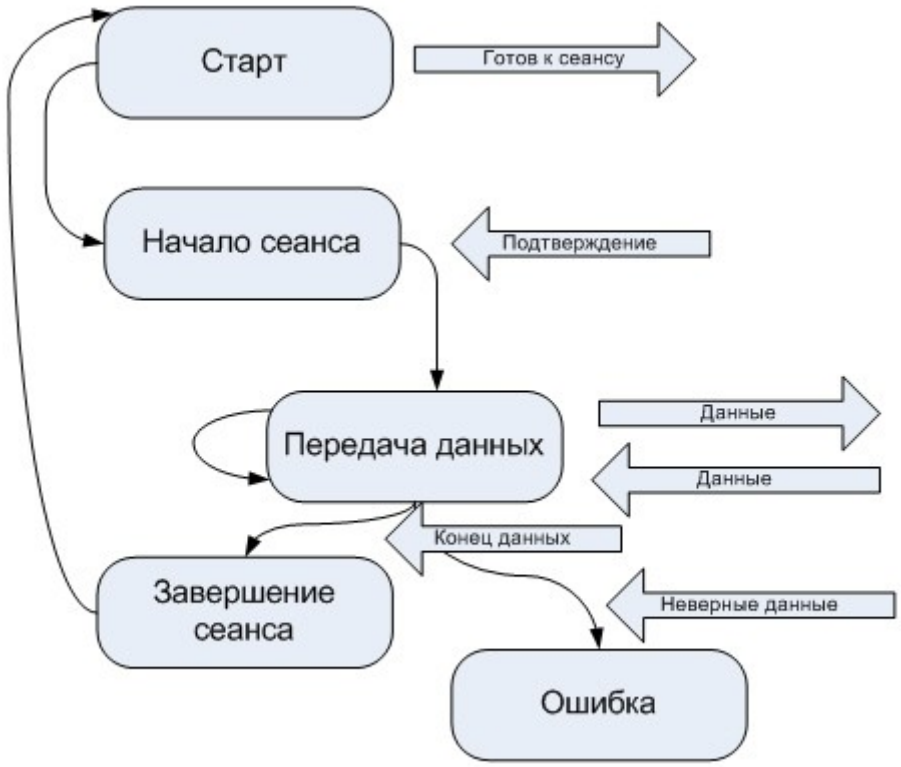


Рис. 3.4. Пример конечного автомата процесса приема-передачи данных

Тестовое окружение для такого процесса также должно иметь структуру конечного автомата и пересылать данные в том же формате, что и тестируемый процесс. Целью тестирования в данном случае будет показать, что процесс обрабатывает принимаемые данные в соответствии с требованиями, форматы передаваемых данных корректны, а также что процесс во время своей работы действительно проходит все состояния конечного автомата, моделирующего его поведение.