МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Пятигорский институт (филиал) СКФУ

Колледж Пятигорского института (филиала) СКФУ

ПМ.05 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНФОРМАЦИОННЫХ СИСТЕМ

ПМ.05.03 ТЕСТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

Специальности СПО

09.02.07 Информационные системы и программирование

(ЭЛЕКТРОННЫЙ ДОКУМЕНТ)

Пятигорск 2021

Методические указания для выполнения практических занятий по дисциплине ПМ.05.03 Тестирование информационных систем составлены в соответствии с требованиями ФГОС СПО. Предназначены для студентов, обучающихся по специальности 09.02.07 Информационные системы и программирование.

Рассмотрено на заседании ПЦК Колледжа Пятигорского института (филиала) СКФУ

Протокол № 8 от «22» марта 2021г.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Программа дисциплины Тестирование информационных систем предусматривает изучение тестирования информационных систем.

При изучении предмета следует соблюдать единство терминологии и обозначения в соответствии с действующими стандартами, Международной системной единицы (СИ). В результате изучения дисциплины Тестирование информационных систем студенты *должны* **знать**:

- основные виды и процедуры обработки информации, модели и методы решения задач обработки информации;
- основные платформы для создания, исполнения и управления информационной системой;
- основные процессы управления проектом разработки;
- основные модели построения информационных систем, их структуру, особенности и области применения;
- методы и средства проектирования, разработки и тестирования информационных систем;
- систему стандартизации, сертификации и систему обеспечения качества продукции. **уметь**:
- осуществлять постановку задач по обработке информации;
- проводить анализ предметной области;
- осуществлять выбор модели и средства построения информационной системы и программных средств;
- использовать алгоритмы обработки информации для различных приложений;
- решать прикладные вопросы программирования и языка сценариев для создания программ;
- разрабатывать графический интерфейс приложения;
- создавать и управлять проектом по разработке приложения;
- проектировать и разрабатывать систему по заданным требованиям и спецификациям. **иметь практический опыт в**:
- управлении процессом разработки приложений с использованием инструментальных средств;
- обеспечении сбора данных для анализа использования и функционирования информационной системы;
- программировании в соответствии с требованиями технического задания;
- использовании критериев оценки качества и надежности функционирования информационной системы;
- применении методики тестирования разрабатываемых приложений;
- определении состава оборудования и программных средств разработки информационной системы;
- разработке документации по эксплуатации информационной системы;
- проведении оценки качества и экономической эффективности информационной системы в рамках своей компетенции;
- модификации отдельных модулей информационной системы.

В результате освоения учебной дисциплины студент должен овладевать:

Общими компетенциями:

ОК 01. Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам.

ОК 02. Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности.

ОК 03. Планировать и реализовывать собственное профессиональное и личностное развитие.

ОК 04. Работать в коллективе и команде, эффективно взаимодействовать с коллегами, руководством, клиентами.

ОК 05. Осуществлять устную и письменную коммуникацию на государственном языке с учетом особенностей социального и культурного контекста.

ОК 07. Содействовать сохранению окружающей среды, ресурсосбережению, эффективно действовать в чрезвычайных ситуациях

ОК 09. Использовать информационные технологии в профессиональной деятельности.

ОК 10. Пользоваться профессиональной документацией на государственном и иностранном языке.

Профессиональными компетенциями:

ПК 5.1. Собирать исходные данные для разработки проектной документации на информационную систему.

ПК 5.2. Разрабатывать проектную документацию на разработку информационной системы в соответствии с требованиями заказчика.

ПК 5.3. Разрабатывать подсистемы безопасности информационной системы в соответствии с техническим заданием.

ПК 5.4. Производить разработку модулей информационной системы в соответствии с техническим заданием.

ПК 5.5. Осуществлять тестирование информационной системы на этапе опытной эксплуатации с фиксацией выявленных ошибок кодирования в разрабатываемых модулях информационной системы.

ПК 5.6. Разрабатывать техническую документацию на эксплуатацию информационной системы.

ПК 5.7. Производить оценку информационной системы для выявления возможности ее модернизации.

ОБЩИЕ МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

По дисциплине Тестирование информационных систем практические работы содержат задачи и теоретические вопросы. Варианты для каждого обучающегося – индивидуальные.

Задачи и ответы на вопросы, выполненные не по своему варианту, не засчитываются.

Практическая работа выполняется в отдельной тетради. Условия задачи и формулировки вопросов переписываются полностью. Формулы, расчеты, ответы на вопросы пишутся ручкой, а чертежи, схемы и рисунки выполняются карандашом, на графиках и диаграммах указывается масштаб. Вначале задача решается в общем виде, затем делаются расчёты по условию задания. Решение задач обязательно ведется в Международной системе единиц (СИ).

При выполнении практической работы необходимо следовать методическим указаниям: повторить краткое содержание теории, запомнить основные формулы и законы, проанализировать пример выполнения аналогичного задания, затем преступить

непосредственно к решению задачи. К зачету допускаются студенты, получившие положительные оценки по всем практическим работам.

Правила выполнения практических работ.

1. Студент должен прийти на практическое занятие подготовленным к выполнению практической работы.
2. Каждый студент после проведения работы должен представить отчет о проделанной работе с анализом полученных результатов и выводом по работе.
3. Таблицы и рисунки следует выполнять с помощью чертежных инструментов (линейки, циркуля, и.т.д.) карандашом с соблюдением ЕСКД.
4. Расчет следует проводить с точностью до двух значащих цифр.
5. Исправления проводить на обратной стороне листа. При мелких исправлениях неправильное слово (буква, число и т.п.) аккуратно зачеркивается и над ним пишут правильное пропущенное слово (букву, число и т.п.).
6. Вспомогательные расчеты можно выполнять на отдельных листах, а при необходимости на листах отчета.
7. Если студент не выполнит практическую работу или часть работы, то он выполнит ее во внеурочное время, согласованное с преподавателем.
8. Оценку по практической работе студент получает с учетом срока выполнения работы, если;

- расчеты выполнены правильно и в полном объеме;
- сделан анализ проделанной работы и вывод по результатам работы;
- студент может пояснить выполнение любого этапа работы;
- отчет выполнен в соответствии с требованиями к выполнению работы.

Практическая работа №1. Описание тестируемой системы.

Практикум базируется на тестировании модели реальной системы управления автоматизированным комплексом хранения подшипников. Она обеспечивает прием подшипников на склад, сохранение характеристик поступивших подшипников в базе данных (БД), а при поступлении заявки на подшипники вместе с параметрами оси - подбор подходящих подшипников и их выдачу. У каждого из элементов комплекса (склада, терминала подшипника и терминала оси) существует программа низкоуровневого управления, реализованная в виде динамически подключаемой библиотеки (dll), принимающая на вход высокоуровневые команды, и преобразующая их в управляющие воздействия на данный элемент тестируемой системы. Таким образом, есть реальное **окружение** - аппаратура и dll, которые осуществляют связь с аппаратурой. Система вызывает следующие функции из dll для своих элементов (см. рис. 1.1):

GetStoreStat, GetStoreMessage, SendStoreCom (Store.dll) для склада.

GetAxlePar (Axle.dll) для терминала оси.

GetRollerPar (Bearing.dll) для терминала подшипника.

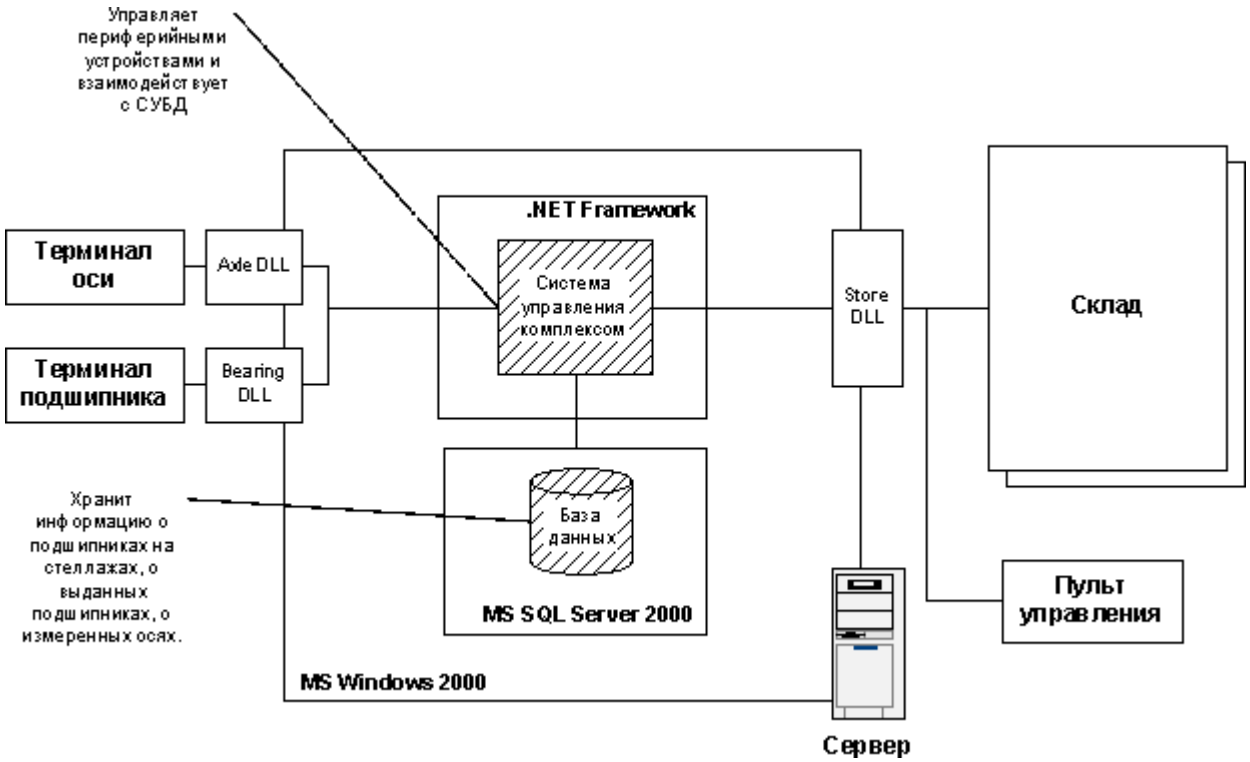


Рис. 1.1. Система и ее окружение

Поскольку для тестирования используется модель системы, в ее составе реальное окружение заменено на модельное, обеспечиваемое специальной библиотекой dll-функций окружения.

Тестируемая система реализована как многопоточное приложение. Многопоточность порождает недетерминированность поведения системы во времени. Поэтому при тестировании необходимо учитывать, что возможны различные варианты допустимых временных последовательностей событий системы. Кроме того, совсем не просто точно воспроизвести прогон конкретного теста, когда система содержит много параллельных потоков, так как планировщик операционной системы сам определяет порядок событий. Изменения, вносимые в программы, не связанные с тестируемой системой, могут повлиять на порядок, в котором будут выполняться (воспроизводиться) потоки событий тестируемой системы. Это может привести к тому, что после выявления и исправления дефекта при проведении повторного тестирования далеко не всегда удастся убедиться в том, что дефект действительно устранен, если ошибка не обнаружена во время прогона.

При системном тестировании мы рассматриваем систему как черный ящик. **Тестовый случай (test case)** представляет собой пару **(входные данные, ожидаемый результат)**, в которой входные данные - это описание данных, подаваемых на вход нашей системы, а ожидаемый результат - это описание **выходных данных**, которые система должна предъявить в ответ на соответствующий ввод. **Выполнение (прогон) тестового случая** - это сеанс работы системы, в рамках которого на вход системы подаются наборы данных, предусмотренные спецификацией тестового случая, и фиксируются результаты их обработки, которые затем сравниваются с ожидаемыми результатами, указанными в тестовом случае. Если фактический результат отличается от ожидаемого, значит, обнаружен отказ, т.е. тестируемая система не прошла испытание на заданном **тестовом случае**. Если полученный результат совпадает с ожидаемым, значит, тестируемая система прошла испытание на заданном тестовом случае. Из тестовых случаев формируются **тестовые наборы (test suits)**. **Тестовые наборы** организованы в определенном порядке, отражающем свойства **тестовых случаев**. Если система успешно

справилась со всеми **тестовыми случаями из набора**, то она успешно прошла испытания на **тестовом наборе**.

Для нашей системы входными данными является состояние окружения ее компонентов:

Склад. Состояние склада будет характеризоваться следующими параметрами:

Статус склада (StoreStat).

Сообщение от склада о результатах выполнения команды (StoreMessage).

Сообщение от склада о результатах получения команды - статус команды (CommandStatus).

Терминал подшипника. Состояние терминала подшипника задается следующими параметрами:

Статус обмена с терминалом подшипника.

Характеристики (параметры) подшипника (RollerPar):

ФИО мастера, производившего измерения.

Название депо.

Номер рабочей смены.

Номер подшипника.

Номер группы подшипника.

Тип сепаратора подшипника.

Терминал оси. Состояние терминала оси задается следующими параметрами:

Статус обмена с терминалом оси.

Характеристики (параметры) оси (AxlePar):

ФИО мастера, производившего измерения.

Название депо.

Номер оси.

Сторона оси: правая или левая.

Посадочный диаметр задний.

Посадочный диаметр передний.

База данных (БД). В БД хранятся характеристики поступивших на склад подшипников. При выборе подходящего для оси подшипника система обращается за этой информацией к БД. Поэтому имеет смысл предварительно очистить БД или заполнить ее определенными данными.

В спецификации **тестового случая** должны быть заданы состояние окружения (**входные данные**) и ожидаемая последовательность событий в системе (**ожидаемый результат**). После прогона **тестового случая** мы получим реальную последовательность событий в системе (**выходные данные**) при заданном состоянии окружения. Сравнивая фактический результат и ожидаемый, можно сделать вывод о том, прошла ли тестируемая система испытание на заданном **тестовом случае**. В качестве **ожидаемого результата** будем использовать пошаговое описание **случая использования (use case)**, так как оно определяет, как при заданном состоянии окружения система должна функционировать. Задавая ожидаемый результат, очень важно помнить о том, что при заданном состоянии окружения возможны различные варианты последовательности событий системы, которые все являются правильными.

В процессе работы последовательность событий (команд) системы, или история системы, записывается в **журнал (log) системы**. Вы можете использовать SystemLogAnimator (см. п.14 SysLog Animator Manual) для визуализации журнала системы. Выбирая различные log-файлы системы для визуализации, можно получить наглядное и достаточно полное представление о

функционировании системы и о том, какие события и в каком порядке могут происходить в системе.

Практическая работа №2. Описание окружения тестируемой системы.

Планирование тестирования

Процесс тестирования

Процесс тестирования находится в прямой зависимости от процесса разработки программного обеспечения, но при этом сильно отличается от него, поскольку преследует другие цели. Разработка ориентирована на построение программного продукта, тогда как тестирование отвечает на вопрос, соответствует ли разрабатываемый программный продукт требованиям, в которых зафиксирован первоначальный замысел изделия (т.е. то, что заказал заказчик).

Вместе оба процесса охватывают виды деятельности, необходимые для получения качественного продукта. Ошибки могут быть привнесены на каждой стадии разработки. Следовательно, каждому этапу разработки должен соответствовать этап тестирования. Отношения между этими процессами таковы, что если что-то разрабатывается, то оно подвергается тестированию, а результаты тестирования используются для определения, соответствует ли это "что-то" набору предъявляемых требований. Процесс тестирования возвращает выявленные им ошибки в процесс разработки. Процесс разработки передает процессу тестирования новые и исправленные проектные версии.

Планирование тестирования

Как было отмечено выше, процесс тестирования тесно связан с процессом разработки. Соответственно планирование тестирования тоже зависит от выбранной модели разработки. Однако вне зависимости от модели разработки при планировании тестирования необходимо ответить на пять вопросов, определяющих этот процесс:

Кто будет тестировать и на каких этапах?

Разработчики продукта, независимая группа тестировщиков или совместно?

Какие компоненты надо тестировать?

Будут ли подвергнуты тестированию все компоненты программного продукта или только компоненты, которые угрожают наибольшими потерями для всего проекта?

Когда надо тестировать?

Будет ли это непрерывный процесс, вид деятельности, выполняемый в специальных контрольных точках, или вид деятельности, выполняемый на завершающей стадии разработки?

Как надо тестировать?

Будет ли тестирование сосредоточено только на проверке того, что данный продукт должен выполнять, или также на том, как это реализовано?

В каком объеме тестировать?

Как определить, в достаточном ли объеме выполнено тестирование, или как распределить ограниченные ресурсы, выделенные под тестирование?

Кто будет тестировать?

Разработчик - это роль, для которой характерны виды деятельности, ориентированные на создание программного продукта (ПП). Тестировщик - это роль, для которой характерны виды деятельности, ориентированные на улучшение/обеспечение качества программного продукта. Эта роль предусматривает выбор тестов, необходимых для конкретных целей, построение тестов, выполнение тестов и оценку результатов. Конкретный исполнитель проекта

может выступать как в роли разработчика, так и в роли тестировщика. Момент начала тестирования в проекте можно регулировать (рис. 1.2).

Рис. 1.2. Кто тестирует

В рамках данного практикума студенту предназначена роль тестировщика.

Какие компоненты надо тестировать?

Могут быть варианты, когда ничего не надо тестировать (поскольку все компоненты были протестированы ранее), а может потребоваться тестировать каждый компонент с точностью до строки кода. В объектно- ориентированном программировании базовым компонентом является класс. В этом случае область тестирования определяется классами. Область тестирования на уровне классов подлежит выбору (рис. 1.3). Классы, заимствованные из других проектов или взятые из библиотек, чаще всего в повторном тестировании не нуждаются. Существуют различные стратегии по выбору подмножества классов для тестирования.

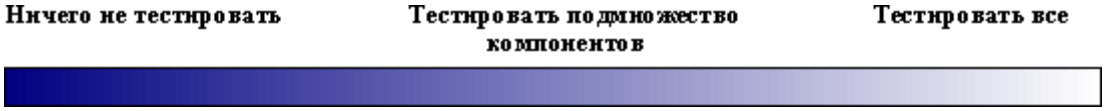


Рис. 1.3. Что тестировать

В нашем случае будет производиться тестирование всех классов приложения.

Когда надо тестировать?

Компоненты можно тестировать на завершающем этапе, когда они будут интегрированы в единый выполняемый модуль. Частота тестирования определяется различными соображениями. Можно проводить тестирование каждый день, учитывая тот факт, что чем раньше выявлена проблема, тем легче и дешевле ее решение. Можно тестировать программный компонент по мере завершения его разработки (рис. 1.4). Частое тестирование компонентов несколько замедляет ранние этапы разработки, однако сопряженные с этим потери с лихвой восполняются за счет меньшего числа проблем на более поздних этапах разработки проекта, когда отдельные модули объединяются в более крупные компоненты системы.

В случае, когда компоненты системы не отличаются большой сложностью, можно сначала осуществлять интегрирование не подвергавшихся автономному тестированию компонентов, а затем тестировать объединенный код как единое целое. Такой подход полезен при тестировании компонентов, для которых реализация тестовых драйверов требует существенных усилий. Тестовый драйвер представляет собой программу, которая выполняет прогон тестовых случаев и сбор полученных при этом результатов.

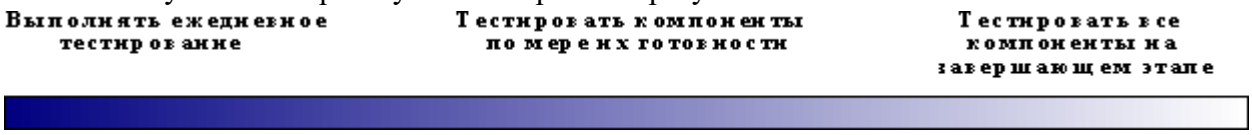


Рис. 1.4. Когда тестировать

Студентам предлагается тестировать компоненты по мере их готовности.

Как надо тестировать?

Основные подходы к тестированию ПО основаны на **спецификации** и **реализации** (рис. 1.5).

Спецификация модуля (или класса) ПП определяет, что этот модуль должен делать, т.е. она описывает допустимые наборы входных данных, подаваемых на вход модуля, включая ограничения на то, как многократные вводы данных должны соотноситься друг с другом, и какие выходные данные соответствуют различным наборам входных данных.

Реализация модуля ПП есть выражение алгоритма, порождающего выходные результаты для различных наборов входных данных с соблюдением требований спецификации. **Спецификация** указывает, что делает модуль ПП, а **реализация** показывает, **как** модуль ПП это делает. Полный учет требований спецификации дает гарантию того, что ПП выполняет все, что от него требуется. Полный учет требований к реализации дает гарантию того, что ПП не будет делать того, что от него не требуется.

Спецификация играет важную роль в тестировании. Обычно для множества компонентов ПП создаются спецификации, обеспечивающие разработку и тестирование, включая спецификации систем, подсистем и классов.

Наряду с автономным тестированием компонентов (классов) системы (**модульным уровнем тестирования**), необходимо тестировать взаимодействие между различными компонентами (**интеграционный уровень тестирования**). Цель **интеграционного тестирования** заключается в обнаружении отказов, возникающих вследствие ошибок в интерфейсах или в силу неверных предположений относительно интерфейсов. После интеграционного тестирования проводится **системное тестирование** ПП (**системный уровень**). На этом уровне тестированию подвергается система как единое целое.

Тестирование следует осуществлять в достаточных объемах, чтобы быть более-менее уверенным в том, что ПП функционирует в соответствии с предъявленными к нему требованиями, т.е. выполняется принцип **адекватности** тестирования ПП. **Адекватность** можно измерить, используя понятие **покрытия**. **Покрытие** можно измерить двумя способами. Первый заключается в подсчете количества требований, сформулированных в спецификации, которые подверглись тестированию. Второй способ заключается в подсчете выполненных компонентов ПП в результате прогона тестового набора. Набор тестов можно считать адекватным, если определенная часть строк исходного кода или исполняемых ветвей исходного кода была выполнена, по крайней мере, один раз во время прогона тестового набора. Эти два способа измерения отражают два базовых подхода к тестированию:

- при использовании первого подхода проверяется, что должен выполнять ПП;
- при использовании второго подхода проверяется, как фактически работает ПП.

При **тестировании в соответствии со спецификацией (функциональном тестировании или тестировании "черного ящика")** построение тестовых случаев производится в соответствии со спецификацией и не зависит от того, как реализован ПП. Эффективность зависит от качества спецификации и способности тестировщика корректно ее интерпретировать.

При **структурном тестировании (тестировании в соответствии с реализацией или тестировании "белого ящика")** построение тестовых случаев производится на основе программного кода, представляющего собой реализацию ПП. Входные данные каждого тестового случая должны быть определены спецификацией ПП, однако они могут быть выбраны на основе анализа самого программного кода для прохождения той или иной ветви программы. При этом покрытие увеличивается.

Рис. 1.5. Как тестировать

Практическая работа №3. Планирование тестирования.

Мы будем использовать оба подхода. При тестировании классов мы будем стремиться покрыть как спецификации классов, так и код их реализации. При тестировании взаимодействий будем покрывать спецификацию. При системном тестировании также будем стремиться покрыть спецификацию системы.

В каком объеме тестировать?

Различные уровни адекватного тестирования изображены на рис. 1.6, который охватывает случаи от отсутствия тестирования до исчерпывающего тестирования, когда выполняется прогон всех возможных тестовых случаев. Объем необходимого тестирования следует определять исходя из краткосрочных и долгосрочных целей проекта и в соответствии с особенностями разрабатываемого ПП. **Покрытие** - это мера полноты использования возможностей программного компонента тестовым набором.

Например, одна из мер - задействована ли каждая строка программного кода продукта хотя бы один раз при прогоне данного тестового набора. Другая мера - количество требований спецификации, проверенных данным тестовым набором. Если требования сформулированы в терминах **случаев использования**, то **покрытие** измеряется количеством случаев использования и числом сценариев, построенных для каждого случая использования.

Анализ рисков в процессе тестирования применяется для определения уровня детализации и времени, затрачиваемого на тестирование конкретного компонента. Например, на тестирование классов, более важных для приложения, отводится больше времени.

Рис. 1.6. В каком объеме тестировать Мы будем использовать все перечисленные меры.

Ответы на поставленные вопросы и, возможно, на многие другие, оформляются в виде набора документов, принятого в компании. Например, **тестовый план** может содержать следующую информацию:

1. Перечень тестовых ресурсов.
2. Перечень функций и подсистем, подлежащих тестированию.
3. Тестовую стратегию:

- Анализ функций и подсистем с целью определения слабых мест, требующих исчерпывающего тестирования, то есть участков функциональности, где появление дефектов наиболее вероятно.
- Определение стратегии выбора входных данных для тестирования. Поскольку в реальных применениях множество входных данных программного продукта практически бесконечно, выбор конечного подмножества для проведения тестирования является сложной задачей. Для ее решения могут быть применены методы покрытия классов входных и выходных данных, анализ крайних значений, покрытие случаев использования и тому подобное. Выбранная стратегия должна быть обоснована и задокументирована.
- Определение потребности автоматизации процесса тестирования. При этом решение об использовании существующей, либо о создании новой автоматизированной

системы тестирования должно быть обосновано, а также продемонстрирована оценка затрат на создание новой системы или на внедрение уже существующей.

4. График (расписание) тестовых циклов.
5. Указание конкретных параметров аппаратуры и программного окружения.
6. Определение тестовых метрик, которые необходимо собирать и анализировать, таких как покрытие набора требований, покрытие кода, количество и уровень серьезности дефектов, объем тестового кода и т.п.

Практическая работа №4. Модульное тестирование на примере классов. Тестовый прогон.

Цель тестирования программных модулей состоит в том, чтобы удостовериться, что каждый модуль соответствует своей спецификации. Если это так, то причиной любых ошибок, которые возникают при их объединении, является неправильная стыковка модулей. В процедурно- ориентированном программировании модулем называется процедура или функция, иногда группа процедур, которая реализует абстрактный тип данных. Тестирование модулей обычно представляет собой некоторое сочетание проверок и **прогонов тестовых случаев**. Можно составить план тестирования модуля, в котором учесть тестовые случаи и построение **тестового драйвера**.

Тестирование классов аналогично тестированию модулей. Основным элементом объектно-ориентированной программы является класс. Рассмотрим методику тестирования отдельного класса. Тестирование классов охватывает виды деятельности, ассоциированные с проверкой реализации класса на точное соответствие спецификации класса. Если реализация корректна, то каждый экземпляр этого класса ведет себя подобающим образом.

Эффективного тестирования классов можно достичь при помощи **ревью** и **тестовых прогонов**. **Ревью** представляет собой просмотр исходного кода ПО с целью обнаружения ошибок и дефектов, возможно, до того, как это ПО заработает. **Ревьюирование** предназначено для выявления таких ошибок, как неспособность выполнять то или иное **требование спецификации** или ее неправильное понимание, а также алгоритмических ошибок в реализации. **Тестовый прогон** обеспечивает тестирование ПО в процессе выполнения программы. Осуществляя прогон программы, тестировщик стремится определить, способна ли программа вести себя в соответствии со спецификацией. Тестировщик должен выбрать наборы входных данных, определить соответствующие им правильные наборы выходных данных и сопоставить их с реально получаемыми выходными данными.

Рассмотрим тестирование классов в режиме **прогона тестовых случаев**. После идентификации **тестовых случаев** для класса нужно реализовать **тестовый драйвер**, обеспечивающий прогон каждого тестового случая, и запротоколировать результаты каждого прогона. При тестировании классов **тестовый драйвер** создает один или большее число экземпляров тестируемого класса и осуществляет прогон тестовых случаев. **Тестовый драйвер** может быть реализован как автономный тестирующий класс.

Кто, что, когда, как и в каком объеме?

Рассмотрим эти вопросы в контексте тестирования классов.

Кто выполняет тестирование? Обычно тестирование классов выполняют их разработчики. В этом случае время на изучение спецификации и реализации сводится к минимуму. Недостатком подхода является то, что если разработчик неправильно понял спецификации, то он для своей неправильной реализации разработает и "ошибочные" тестовые наборы.

Что тестировать? Необходимо удостовериться, что программный код класса в точности отвечает требованиям, сформулированным в его спецификации, и что он не делает ничего более.

В какой момент следует выполнять тестирование? План тестирования или хотя бы тестовые случаи должны разрабатываться после составления полной спецификации класса. Разработка тестовых случаев по мере реализации класса помогает разработчику лучше понять спецификацию. Тестирование класса должно проводиться до того, как возникнет необходимость использовать этот класс в других компонентах ПО. Регрессионное тестирование класса должно выполняться всякий раз, когда меняется реализация класса. Регрессионное тестирование позволяет убедиться в том, что разработанные и оттестированные функции продолжают удовлетворять спецификации после выполнения модификации ПО.

Как будет выполняться тестирование? Тестирование классов обычно выполняется путем разработки тестового драйвера, который создает экземпляры классов и окружает эти экземпляры соответствующей средой (тестовым окружением), чтобы стал возможен прогон соответствующего тестового случая. Драйвер посылает сообщения экземпляру класса в соответствии со спецификацией тестового случая, а затем проверяет исход этих сообщений. Тестовый драйвер должен удалять созданные им экземпляры тестируемого класса. Статические элементы данных класса также необходимо тестировать.

Какие объемы тестирования следует считать адекватными? Адекватность может быть измерена полнотой охвата тестами спецификации или реализации. Будем использовать оба способа.

Практическая работа №5. Типы классов.

Что тестировать?

Можно выделить два типа классов с точки зрения их взаимодействия с другими классами:

- примитивные классы;
- непримитивные классы.

Примитивный класс может порождать экземпляры, и эти экземпляры можно использовать без необходимости создания экземпляров каких-либо других классов, в том числе и данного класса. Такие объекты представляют собой простейшие компоненты системы и, несомненно, играют важную роль при выполнении любой программы. Тем не менее, в объектно-ориентированной программе существует сравнительно небольшое количество примитивных классов, которые реалистично моделируют объекты задачи и все отношения между этими объектами. Обычным явлением для хорошо спроектированных объектноориентированных программ является использование непримитивных классов. Основываясь на этой информации, определим, к какому типу относится каждый класс в нашем приложении
(табл. 2.1).

Таблица 2.1. Типы Классов

Класс	**Тип**
TBearingParam	Примитивный
TAxleParam	Примитивный
TCommand	Примитивный
TLog	Примитивный

TCommandQueue	Непримитивный
TStore	Непримитивный
TTerminalBearing	Непримитивный
TTerminalAxle	Непримитивный
TModel	Непримитивный
MainForm	Непримитивный

В большинстве объектно-ориентированных языков члены класса имеют один из трех уровней доступа:

Public. Члены с доступом **public** доступны из любых классов. Они образуют интерфейс класса, которым будет пользоваться любой разработчик, использующий данный класс в своем приложении.

Private. Члены с доступом **private** доступны только внутри самого класса, то есть из его методов. Они являются частью внутренней реализации класса и недоступны стороннему разработчику.

Protected. Члены с доступом **protected** доступны из самого класса и из классов, являющихся его потомками, но недоступны извне. Использование этих методов возможно только при создании класса- потомка, расширяющего функциональность базового класса.

Таким образом, необходимость тестирования функциональности класса зависит от того, предоставляется ли им возможность наследования. Если класс является законченным (**final**) и не предполагает наследования, необходимо тестирование его **public** части (впрочем, классы **final** не содержат **protected** членов). Если же класс рассчитан на расширение за счет наследования, необходимо тестирование также его **protected** части.

Кроме того, во многих языках класс может содержать статические (**static**) члены, которые принадлежат классу в целом, а не его конкретным экземплярам. При наличии **public static** или **protected static** членов, кроме тестирования объектов класса, должно отдельно выполняться тестирование статической части класса.

Как тестировать?

Как уже упоминалось, для тестирования классов применяются **тестовые драйверы**. Существует несколько способов реализации тестового драйвера:

Тестовый драйвер реализуется в виде отдельного класса. Методы этого класса создают объекты тестируемого класса и вызывают их методы, в том числе статические методы класса. Таким способом можно тестировать public часть класса.

Тестовый драйвер реализуется в виде класса, наследуемого от тестируемого. В отличие от предыдущего способа, такому тестовому драйверу доступна не только public, но и protected часть.

Тестовый драйвер реализуется непосредственно внутри тестируемого класса (в класс добавляются диагностические методы). Такой тестовый драйвер имеет доступ ко всей реализации класса, включая private члены. В этом случае в методы класса включаются вызовы отладочных функций и агенты, отслеживающие некоторые события при тестировании.

В дальнейшем мы будем использовать первый способ при реализации драйверов.

При разработке **спецификации** класса можно задействовать один из следующих подходов:

Контрактный подход. Интерфейс определяется в виде обязательств отправителя и получателя, вступивших во взаимодействие. Операция определяется как набор обязательств каждой стороны, причем ответственность по отношению друг к другу соблюдается как отправителем, так и получателем.

Подход защитного программирования. Интерфейс определяется главным образом в терминах получателя. Операция возвращает результат запроса - успешное или неудачное выполнение по конкретной причине (например, по недопустимому входному значению). Другими словами, соответствующий получатель следит за тем, чтобы на вход не попали некорректные данные, т.е. проверяет правильность и допустимость входных данных, и после получения запроса сообщает отправителю результат обработки запроса.

Различие между **контрактным** и **защитным методами проектирования** распространяется и на тестирование. **Контрактное проектирование** возлагает большую ответственность на проектировщика, чем на программы поиска ошибок. Основное внимание во время тестирования взаимодействий в условиях контактного подхода уделяется проверке того, выполнены ли объектом-отправителем предусловия методов получающего объекта. Не допускается построение тестовых случаев, нарушающих эти предусловия. Обычно практикуется перевод объекта- получателя в некоторое заданное состояние, после чего инициируется выполнение тестового драйвера, по условиям которого объект- отправитель требует, чтобы объект-получатель находился в другом состоянии. Смысл подобной проверки заключается в том, чтобы установить, выполняет ли объект-отправитель проверку предусловий объекта-получателя, прежде чем отправить заранее неприемлемое сообщение, и корректно ли он прекращает свою работу.

Подробное описание тестового случая

Рассматривается пример тестов на C# для класса TCommand (приложение 3 (HLD)). При выполнении заданий необходимо будет самостоятельно написать тесты для других классов приложения. Параллельно с изучением этого раздела полезно открыть проект ModuleTesting\ModuleTests.sln.

Рассмотрим тестирование класса TCommand. Этот класс реализует единственную операцию GetFullName(), которая возвращает полное название команды в виде строки. Разработаем **спецификацию тестового случая** для тестирования метода GetFullName на основе спецификации этого класса (приложение 3):

Название класса: TCommand	Название тестового случая: TCommandTest1

Описание тестового случая: Тест проверяет правильность работы метода GetFullName - получения полного названия команды на основе кода команды. В тесте подаются следующие значения кодов команд (входные значения): -1, 1, 2, 4, 6, 20, где -1 - запрещенное значение Начальные условия: Нет

Ожидаемый результат:

Перечисленным входным значениям должны соответствовать следующие выходные:

Коду команды -1 должно соответствовать сообщение "ОШИБКА: Неверный код команды"

Коду команды 1 должно соответствовать полное название команды "ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ"

Коду команды 2 должно соответствовать полное название команды "ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ"

Коду команды 4 должно соответствовать полное название команды "ПОЛОЖИТЬ В РЕЗЕРВ"

Коду команды 6 должно соответствовать полное название команды "ПРОИЗВЕСТИ ЗАНУЛЕНИЕ"

Коду команды 20 должно соответствовать полное название команды "ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ"

На основе спецификации был создан тестовый драйвер - класс TCommandTester, наследующий функциональность абстрактного класса Tester.

```
public class Log
{
 static private StreamWriter log=new
   StreamWriter("log.log"); //Создание лог файла  static
public void Add(string msg)
   //Добавление сообщения в лог файл
 {
   log.WriteLine(msg);
 }
 static public void Close() //Закрыть лог файл
 {
   log.Close();
 }
}
abstract class Tester
{
protected void LogMessage(string s)
//Добавление сообщения в лог-файл
 {
   Log.Add(s);
 }
}
class TCommandTester:Tester // Тестовый драйвер {
```

```
 TCommand OUT;    public
TCommandTester()
 {
  OUT=new TCommand();
  Run();
 }
 private void Run()
 {
  TCommandTest1();
 }
 private void TCommandTest1()
 {
  int[] commands = {-1, 1, 2, 4, 6, 20};    for(int
i=0;i<=5;i++)
                {
                 OUT.NameCommand=commands[i];
                 LogMessage(commands[i].ToString()+" :
                  "+OUT.GetFullName());
                }
 }
 [STAThread]
 static void Main()
 {
  TCommandTester CommandTester = new TCommandTester();
  Log.Close();
 }
}
```

Листинг 2.1. Тестовый драйвер

Класс TCommandTester содержит метод TCommandTest1(), в котором реализована вся функциональность теста. В данном случае для покрытия спецификации достаточно перебрать следующие значения кодов команд: -1, 1, 2, 4, 6, 20, где -1 - запрещенное значение, и получить соответствующие им полное название команды с помощью метода GetFullName(). Пары соответствующих значений заносятся в log-файл для последующей проверки на соответствие спецификации.

Таким образом, для тестирования любого метода класса необходимо:

- Определить, какая часть функциональности метода должна быть протестирована, то есть при каких условиях он должен вызываться. Под условиями здесь понимаются параметры вызова методов, значения полей и свойств объектов, наличие и содержимое используемых файлов и т. д.
- Создать тестовое окружение, обеспечивающее требуемые условия.
- Запустить тестовое окружение на выполнение.
- Обеспечить сохранение результатов в файл для их последующей проверки.
- После завершения выполнения сравнить полученные результаты со спецификацией.

Практическая работа №6. Модульное тестирование на примере классов. Описание тестовых процедур.

Описание тестовых процедур

Как запустить тест?

Для того чтобы запустить тест, нужно:

- В методе Run тестового драйвера TCommandTester вызвать метод

TCommandTest1, реализующий тест.

- Собрать и запустить приложение.

Проверка результатов выполнения тестов (сравнение с ожидаемым результатом)

После завершения теста следует просмотреть текстовый журнал теста (..\ModuleTesting\bin\Debug\log.log), чтобы сравнить полученные результаты с ожидаемыми результатами, заданными в спецификации тестового случая TCommandTest1. Журнал теста: -1 : ОШИБКА : Неверный код команды

1 : ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ

2 : ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ

4 : ПОЛОЖИТЬ В РЕЗЕРВ

6 : ПРОИЗВЕСТИ ЗАНУЛЕНИЕ

20 : ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ

Задание 1

Для остальных примитивных классов (табл. 2.1) в соответствии с приведенным примером необходимо самостоятельно разработать спецификации тестовых случаев, соответствующие тесты и провести тестирование. Отчет требуется составить в следующей форме (табл. 2.2):

Таблица 2.2. Тестовый отчет

Название тестового случая:

Тестировщик:

Тест пройден: Да/Нет (PASS/FAIL)

Степень важности ошибки:
Фатальная (3 уровень - crash)
Серьезная (2 уровень - расхождение в спецификации)

Незначительная (1 уровень - незначительная ошибка) Описание

проблемы:

Как воспроизвести ошибку:

Предлагаемое исправление (необязательно):

Комментарий тестировщика (необязательно):

Практическая работа №7.

Интеграционное тестирование. Идентификация взаимодействий.

Основное назначение тестирования взаимодействий состоит в том, чтобы убедиться, что происходит правильный обмен сообщениями между объектами, классы которых уже прошли тестирование в автономном режиме (на модульном уровне тестирования).

Тестирование взаимодействия или интеграционное тестирование представляет собой тестирование собранных вместе, взаимодействующих модулей (объектов). В интеграционном тестировании можно объединять разное количество объектов - от двух до всех объектов тестируемой системы. Интеграционное тестирование отличается от системного тем, что:

- при интеграционном тестировании используется подход "белого ящика", а при системном - "черного ящика";
- целью интеграционного тестирования является только проверка правильности взаимодействия объектов, тогда как целью системного - проверка правильности функционирования системы в целом.

Идентификация взаимодействий

Взаимодействие объектов представляет собой просто запрос одного объекта (**отправителя**) на выполнение другим объектом (**получателем**) одной из операций получателя и всех видов обработки, необходимых для завершения этого запроса.

В ситуациях, когда в качестве основы тестирования взаимодействий объектов выбраны только спецификации общедоступных операций, тестирование намного проще, чем когда такой основой служит реализация. Мы ограничимся тестированием общедоступного интерфейса. Такой подход вполне оправдан, поскольку мы полагаем, что классы уже успешно прошли модульное тестирование. Тем не менее, выбор такого подхода отнюдь не означает, что не нужно возвращаться к спецификациям классов, дабы убедиться в том, что тот или иной метод выполнил все необходимые вычисления. Это обусловливает необходимость проверки значений атрибутов внутреннего состояния получателя, в том числе любых агрегированных атрибутов, т.е. атрибутов, которые сами являются объектами. Основное внимание уделяется отбору тестов на основе спецификации каждой операции из общедоступного интерфейса класса.

Взаимодействия неявно предполагаются в спецификации класса, в которой установлены ссылки на другие объекты. В разделе 4 рассматривалось тестирование примитивных классов. Такие объекты представляют собой простейшие компоненты системы и, несомненно, играют важную роль при выполнении любой программы. Тем не менее, в объектно-ориентированной программе существует сравнительно небольшое количество примитивных классов, которые реалистично моделируют объекты задачи и все отношения между этими объектами. Обычным явлением для хорошо спроектированных объектно- ориентированных программ является использование непримитивных классов; в этих программах им отводится главенствующая роль.

Выявить такие взаимодействующие классы можно, используя отношения ассоциации (в том числе отношения агрегирования и композиции), представленные на диаграмме классов. Ассоциации такого рода преобразуются в интерфейсы класса, а тот или иной класс взаимодействует с другими классами посредством одного или нескольких способов:

Тип 1. Общедоступная операция имеет один или большее число формальных параметров объектного типа. Сообщение устанавливает ассоциацию между получателем и параметром, которая позволяет получателю взаимодействовать с этим параметрическим объектом.

Тип 2. Общедоступная операция возвращает значения объектного типа. На класс может быть возложена задача создания возвращаемого объекта, либо он может возвращать модифицированный параметр.

Тип 3. Метод одного класса создает экземпляр другого класса как часть своей реализации.

Тип 4. Метод одного класса ссылается на глобальный экземпляр некоторого другого класса. Разумеется, принципы хорошего тона в проектировании рекомендуют минимальное использование глобальных объектов. Если реализация какого-либо класса ссылается на некоторый глобальный объект, рассматривайте его как неявный параметр в методах, которые на него ссылаются.

Приведем еще раз таблицу разделения классов на примитивные и непримитивные типы (табл. 3.1):

Таблица 3.1. Типы классов

Класс	Тип
TBearingParam	Примитивный
TAxleParam	Примитивный
TCommand	Примитивный
TLog	Примитивный
TCommandQueue	Непримитивный
Класс	Непримитивный
TStore	Непримитивный
TTerminalBearing	Непримитивный
TTerminalAxle	Непримитивный
TModel	Непримитивный
MainForm	Непримитивный

Таблица 3.2. Типы взаимодействия классов

Непримитивные типы	Tbearing Param	Taxle Param	Tcommand	TCommand Queue	Store	TTerminal Bearing	TTerminal Axle	Log	model
TCommandQueue	1	1	3			1			
TStore	3	1	1						
TTerminalBearing	3								

TTerminalAxle	3				
TModel		3	3	3	
MainForm					

Таким образом, интеграционному тестированию будут подвергнуты взаимодействия перечисленных непримитивных классов.

Практическая работа №8. Выбор тестовых случаев.

Выбор тестовых случаев

Исчерпывающее тестирование, другими словами, прогон каждого возможного тестового случая, покрывающего каждое сочетание значений - это, вне всяких сомнений, надежный подход к тестированию. Однако во многих ситуациях количество тестовых случаев достигает таких больших значений, что обычными методами с ними справиться попросту невозможно. Если имеется принципиальная возможность построения такого большого количества тестовых случаев, на построение и выполнение которых не хватит никакого времени, должен быть разработан систематический метод определения, какими из тестовых случаев следует воспользоваться. Если есть выбор, то мы отдаем предпочтение таким тестовым случаям, которые позволяют найти ошибки, в обнаружении которых мы заинтересованы больше всего.

Существуют различные способы определения, какое подмножество из множества всех возможных тестовых случаев следует выбирать. При любом подходе мы заинтересованы в том, чтобы систематически повышать уровень покрытия.

Подробное описание тестового случая

Продемонстрируем тестирование взаимодействий на примере класса TCommandQueue. Из табл. 3.2, которая была составлена на основе спецификаций классов, описанных в приложении 2, видно, что класс очереди команд взаимодействует со следующими классами: TBearingParam,

TAxleParam,

TCommand,

TStore,

TterminalBearing.

С объектом TCommand осуществляется взаимодействие третьего типа, т. е. TCommandQueue создает объекты класса TCommand как часть своей внутренней реализации.

С остальными классами осуществляется взаимодействие первого типа: ссылки на объекты классов TStore и TTerminalBearing передаются как параметры в конструктор TCommandQueue, а ссылки на объекты классов TBearingParam и TAxleParam передаются в метод TCommandQueue.AddCommand.

Одновременно с изучением этого раздела можно открыть проект IntegrationTesting\IntegrationTests.sln.

Для тестирования взаимодействия класса TCommandQueue и класса TCommand, так же, как и при модульном тестировании, разработаем спецификацию тестового случая:

Таблица 3.3. Спецификация тестового случая

Названия взаимодействующих классов: Название теста: TCommandQueue, TCommand TCommandQueueTest1

Описание теста: тест проверяет возможность создания объекта типа TCommand и добавления его в очередь при вызове метода AddCommand Начальные условия: очередь команд пуста

Ожидаемый результат: в очередь будет добавлена одна команда

На основе этой спецификации был разработан тестовый драйвер - класс TCommandQueueTester, который наследуется от класса Tester. Этот класс содержит:

- Метод Init, в котором создаются объекты классов TStore, TterminalBearing и объект типа TcommandQueue. Этот метод необходимо вызывать в начале каждого теста, чтобы тестируемые объекты создавались вновь:

```
• private void Init()
• {
• TB = new TTerminalBearing();
• S = new TStore();
• CommandQueue=new TCommandQueue(S,TB);
• S.CommandQueue=CommandQueue;
}
```

Пример 3.1. Метод Init

- Методы, реализующие тесты. Каждый тест реализован в отдельном методе.
- Метод Run, в котором вызываются методы тестов.
- Метод dump, который сохраняет в log-файле теста информацию обо всех командах, находящихся в очереди в формате - номер позиции в очереди: полное название команды.
- Точку входа в программу - метод Main, в котором происходит создание экземпляра класса TCommandQueueTester и запуск метода Run.

Сначала создадим тест, который проверяет, создается ли объект типа TCommand, и добавляется ли команда в конец очереди.

```
private void TCommandQueueTest1()
{
Init();
LogMessage("/////////// TCommandQueue Test1 //////////////");
LogMessage("Проверяем, создается ли объект типа TCommand");
// В очереди нет команд dump();
// Добавляем команду
// параметр = -1 означает, что команда должна быть добавлена
//в конец очереди
CommandQueue.AddCommand(TCommand.GetR,0,0,0,new
TBearingParam(),new TAxleParam(),-1);
LogMessage("Command added");
// В очереди одна команда
dump(); }
```

Пример 3.2. Тест, проверяющий создание объекта типа TCommand В этот класс включены еще два разработанных теста.

Практическая работа №9.
Интеграционное тестирование. Описание тестовых процедур.

Описание тестовых процедур

Как запустить тест?

Для выполнения этого теста в методе Run необходимо вызвать метод TCommandQueueTest1() и запустить программу на выполнение:

```
private void Run()
{
  TCommandQueueTest1();
}
```

Пример 3.3. Метод Run

Проверка результатов выполнения тестов (сравнение с ожидаемым результатом) После завершения теста следует просмотреть текстовый журнал теста (..\IntegrationTesting\bin\Debug\test.log), чтобы сравнить полученные результаты с ожидаемыми результатами, заданными в спецификации тестового случая TCommandQueueTest1.

```
////////////////// TCommandQueue Test1 /////////////////
Проверяем, создается ли объект типа TCommand
0 commands in command queue
Command added
1 commands in command queue
0 : ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ
```

Пример 3.4. Журнал теста

Задание 2

Для тестирования взаимодействия остальных непримитивных классов (табл. 2.1) по аналогии с приведенным примером требуется самостоятельно разработать спецификации тестовых случаев, соответствующие тесты, провести тестирование и составить тестовые отчеты (табл. 2.2).

Практическая работа №10. Системное тестирование

Системное тестирование качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает систему в целом и применяется на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей, хотя набор модулей может быть аналогичным. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы, а также отслеживать правильность работы конкретных функций. Основной задачей системного тестирования является выявление дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в использовании и тому подобное.

Поскольку системное тестирование проводится на уровне пользовательских интерфейсов, то построение специальной тестовой системы становится технически необязательным. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная автоматизация тестирования, что может потребовать создания тестовой системы намного более сложной, чем система тестирования на уровне модулей или их комбинаций.

Необходимо подчеркнуть, что существует два принципиально разных подхода к системному тестированию.

В первом варианте для построения тестов используются требования к системе, например, для каждого требования строится тест, который проверяет выполнение данного требования в системе. Этот подход особенно широко применяется при разработке военных и научных систем, когда заказчик вполне осознает, какая функциональность ему нужна, и составляет полный набор **формальных требований**. Тестировщик в данном случае только проверяет, соответствует ли разработанная система этому набору. Такой подход предполагает длинную и дорогостоящую фазу сбора требований, выполняемую до начала собственно проекта. В этом случае для определения требований обычно разрабатывается прототип будущей системы.

Во втором подходе основой для построения тестов служит представление о способах использования продукта и о задачах, которые он решает. На основе более или менее формальной модели пользователя создаются **случаи использования** системы, по которым затем строятся собственно **тестовые случаи**. **Случай использования (use case)** описывает, как субъект использует систему, чтобы выполнить ту или иную задачу. **Субъекты** или **актеры (actors)** могут исполнять различные роли при работе с системой. **Случаи использования** могут описываться с различной степенью абстракции. **Случаи использования** не обязательно охватывают каждое требование. Можно конкретизировать **случаи использования** и расширять их в наборы более **специфических случаев использования (пошаговое описание случая использования)**. В контексте конкретного **случая использования** можно определить один или большее число **сценариев**. **Сценарий** представляет конкретный экземпляр **случая использования - путь в пошаговом описании случая использования**. Каждый **путь (сценарий)** в **случае использования** должен быть протестирован (рис. 4.1).

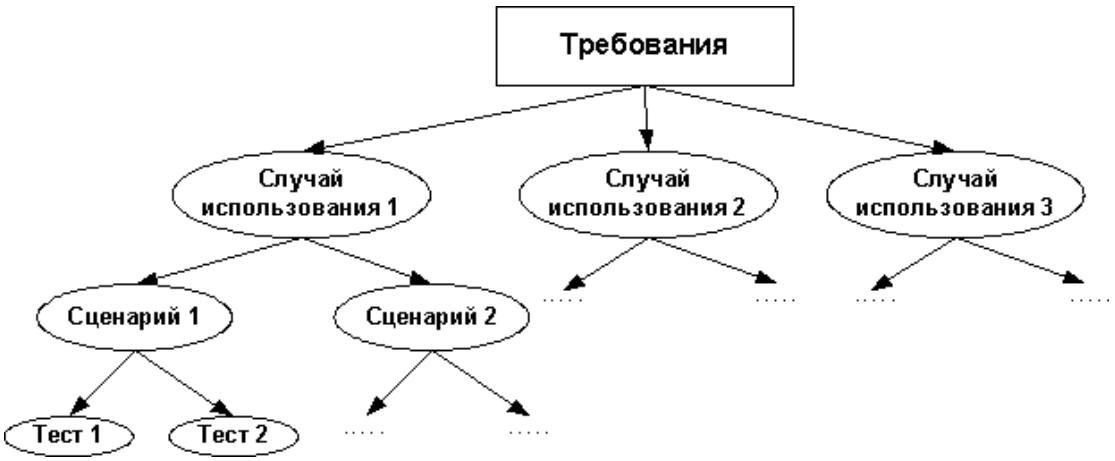


Рис. 4.1. Тестирование случаев использования

Входные данные для каждого **сценария** надо выбирать следующим образом:

- Идентифицировать все значения (входные данные), которые могут задавать субъекты для **случая использования**.
- Определить **классы эквивалентности** для каждого типа входных данных.

- Построить таблицу со списком значений из различных классов эквивалентности.
- Построить **тестовые случаи** на базе таблицы с учетом внешних ограничений.

Далее при построении **тестовых случаев** применялись оба подхода и при выполнении заданий необходимо действовать следующим образом:

- На основе требований определить случаи использования (use case)
- На основе каждого случая использования (use case) построить сценарии. Для каждого сценария разработать тестовые случаи (набор тестов).

Практическая работа №11. Случаи использования системного тестирования.

Случаи использования (use cases)

Описание случая использования (use case) "подбор подшипников для оси"

Последовательно приходят два подшипника, поступает запрос от оси. При поступлении запроса от оси система подбирает два подшипника из имеющихся на складе и выдает их в выходную ячейку.

Рассмотрим этот **случай использования** подробнее. Согласно спецификации, система постоянно опрашивает склад и терминал оси. При поступлении подшипника (статус склада 32) система опрашивает терминал подшипника, формирует и посылает команду складу "принять подшипник" и получает ответ от склада о результатах выполнения команды. При поступлении оси (поступлении параметров оси при опросе терминала оси) система должна подобрать подшипники из имеющихся на складе, сформировать команды для их выдачи, послать их складу и получить ответ о результате выполнения команд.

Далее приводится пошаговое описание этого случая использования:

Приняли на склад первый подшипник (1-10)

Приняли на склад второй подшипник (11-20)

Поступила ось (21-26)

Подбираем первый подшипник для оси (27-30) Подбираем второй подшипник для оси (31-34) Завершение выдачи команд (35-39).

Пошаговое описание случая использования

Пошаговое описание приведено на рис. 4.2.

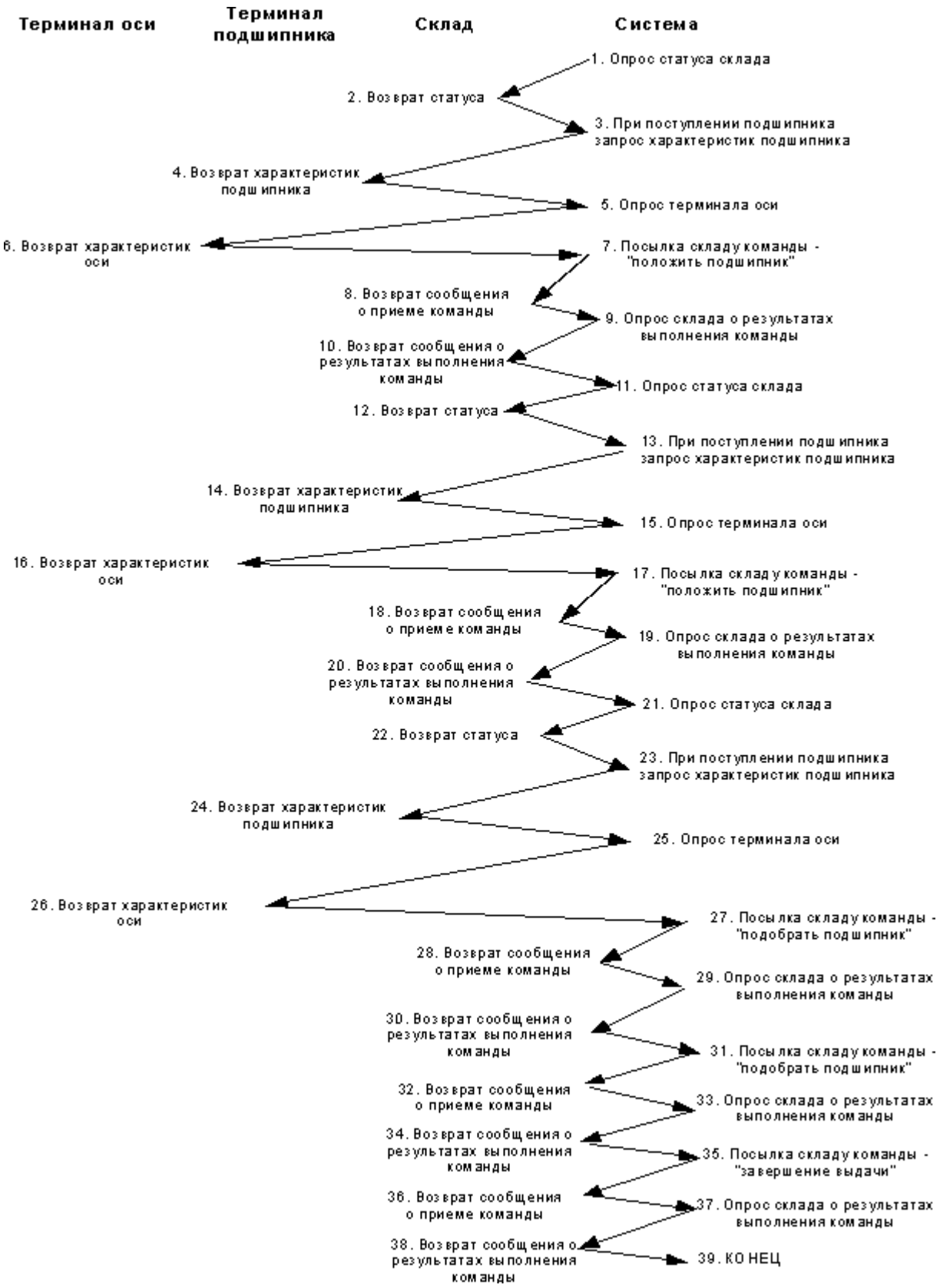


Рис. 4.2. Пример use case

Список альтернативных путей

В пошаговом описании случая использования рассматривался оптимистический сценарий, например при анализе статуса склада в пунктах 3 и 11 считалось, что поступил подшипник. Но необходимо рассмотреть и возможные альтернативные сценарии:

3, 11 - анализ статуса склада:

3.а, 11.а - подшипник в манипуляторе.

3.b, 11.b - склад свободен.

3.c, 11.c - ошибочное состояние.

При получении сообщения от склада о выполнении команды считалось, что команда выполнена без ошибки, но здесь тоже существуют альтернативные варианты:

8, 16, 22, 24, 26 - получение сообщения о выполнении команды:

8.a, 16.a, 22.a, 24.a, 26.a - нет склада.

8.b, 16.b, 22.b, 24.b, 26.b - нет сообщения.

8.c, 16.c, 22.c, 24.c, 26.c - команда выполнена с ошибкой.

Список альтернативных вариантов для рассмотренного случая использования можно продолжить.

Практическая работа №12. Процесс тестирования.

Спецификация тестового случая №1

Состояние окружения (входные данные):

Статус склада (StoreStat=32). Пришел подшипник.

Статус обмена с терминалом подшипника (0 - есть подшипник) и его параметры (RollerPar="0 NewUser Depot1 123456 1 12 1 1").

Статус обмена с терминалом оси (1 - нет оси) и ее параметры (AxlePar="1 NewUser Depot1 123456 1 0 12 12").

Статус команды (CommandStatus=0). Команда успешно принята.

Сообщение от склада (StoreMessage=1). Команда успешно выполнена.

Ожидаемая последовательность событий (выходные данные):

Система запрашивает статус склада (вызов функции GetStoreStat) и получает 32.

Система запрашивает параметры подшипника (вызов функции GetRollerPar) и получает 0 NewUser Depot1 123456 1 12 1 1.

Система запрашивает параметры оси (вызов функции GetAxlePar) и получает 1 NewUser Depot1 123456 1 0 12 12.

Система добавляет в очередь команд склада на последнее место команду SendR (получить из приемника в ячейку) (вызов функции SendStoreCom) и получает сообщение о том, что команда успешно принята - 0.

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage) и получает сообщение о том, что команда успешно выполнена - 1.

Система запрашивает статус склада (вызов функции GetStoreStat) и получает 32.

Система запрашивает параметры подшипника (вызов функции GetRollerPar) и получает 0 NewUser Depot1 123456 1 12 1 1.

Система запрашивает параметры оси (вызов функции GetAxlePar) и получает 1 NewUser Depot1 123456 1 0 12 12.

Система добавляет в очередь команд склада на первое место команду GetR (получить из приемника в ячейку) (вызов функции SendStoreCom) и получает сообщение о том, что команда успешно принята - 0.

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage) и получает сообщение о том, что команда успешно выполнена - 1.

Изменяем состояние окружения (входные данные):

Статус обмена с терминалом подшипника (1 - нет подшипника) и его параметры (RollerPar="1 NewUser Depot1 123456 1 12 1 1").

Статус обмена с терминалом оси (0 - есть ось) и ее параметры (AxlePar="0 NewUser Depot1 123456 1 0 12 12").

Ожидаемая последовательность событий (выходные данные):

Система запрашивает статус склада (вызов функции GetStoreStat) и получает 32.

Система запрашивает параметры подшипника (вызов функции GetRollerPar) и получает 1 NewUser Depot1 123456 1 12 1 1.

Система запрашивает параметры оси (вызов функции GetAxlePar) и получает 0 NewUser Depot1 123456 1 0 12 12.

Система добавляет в очередь команд склада на последнее место команду SendR (вызов функции SendStoreCom) и получает сообщение о том, что команда успешно принята - 0.

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage) и получает сообщение о том, что команда успешно выполнена - 1.

Система добавляет в очередь команд склада на последнее место команду SendR (вызов функции SendStoreCom) и получает сообщение о том, что команда успешно принята - 0.

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage) и получает сообщение о том, что команда успешно выполнена - 1.

Система добавляет в очередь команд склада на последнее место команду Term (вызов функции SendStoreCom) и получает сообщение о том, что команда успешно принята - 0.

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage) и получает сообщение о том, что команда успешно выполнена - 1.

Во всех последующих разделах будет подробно рассматриваться именно этот **тестовый случай**!

Описание процесса системного тестирования Рассмотрим
процесс системного тестирования:

Анализ. Тестируемая система анализируется (проверяется) на наличие определенных свойств, которым надо уделить особое внимание, и определяются соответствующие **тестовые случаи**.

Построение. Выбранные на стадии анализа **тестовые случаи** переводятся на язык программирования.

Выполнение и анализ результатов. Производится выполнение **тестовых случаев**. Полученные результаты анализируются, чтобы определить, успешно ли прошла система испытания на **тестовом наборе**.

Процесс запуска тестовых случаев и анализа полученных результатов должен быть подробно описан в **тестовых процедурах**.

Далее мы рассмотрим три различных подхода, которые используются при системном тестировании:

- Ручное тестирование.
- Автоматизация выполнения и проверки результатов тестирования с помощью скриптов.
- Автоматическая генерация тестов на основе формального описания.

Практическая работа №13. Ручное тестирование

Наиболее распространенным способом разработки тестов является создание тестового кода вручную. Такой способ создания тестов является наиболее гибким, однако производительность тестировщиков при создании тестового кода соизмерима с производительностью разработчиков при создании кода продукта, а объемы тестового кода часто бывают в 1-10 раз больше объема самого продукта.

В этом случае запуск тестов осуществляется вручную. Проверку, прошла ли тестируемая система испытания на заданном **тестовом случае**, тестировщик также осуществляет вручную, сравнивая фактические результаты журнала теста с **ожидаемыми результатами**, описанными в спецификации **тестового случая**.

Функции dll-библиотеки обеспечивают обращение к серверу для получения информации о состоянии элементов комплекса и возвращают серверу информацию о функционировании системы. Значит, для моделирования состояния окружения (**входных данных**) необходимо создать специальный сервер.

Кроме того, необходимо сохранять получаемую от сервера информацию о функционировании системы (**выходные данные**) в журнале (рис. 5.1).

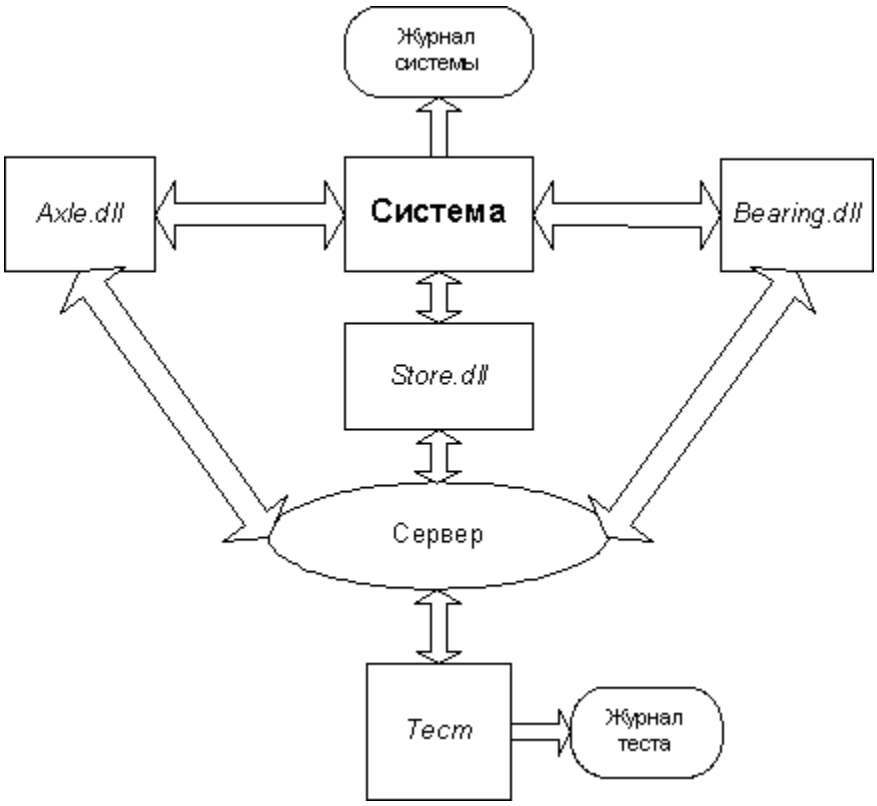


Рис. 5.1. Система и ее окружение (ручное тестирование)

При разработке тестов был использован следующий подход:

- состояние окружения задается в тесте (входные данные);
- в тесте создается сервер:
  - сервер по запросу от dll передает информацию о заданном состоянии окружения;
  - сервер получает от dll информацию о функционировании системы;
- получаемая информация сохраняется в журнале теста.

Подробное описание тестового случая №1

Ознакомление с настоящим пунктом полезно предварить изучением п. 7, содержащего описание ручного тестирования. Здесь рассматривается та часть теста на C#, которую вам придется написать самостоятельно при выполнении заданий. Приведенный пример был

разработан в соответствии со спецификацией тестового случая N1. Для простоты будем считать, что события происходят последовательно в строго заданном порядке. Реально наша система представляет собой многопоточное приложение, поэтому мы не можем это гарантировать.

```
class Test1:Test { override
public void start()
{//Задаем состояние окружения (входные данные) StoreStat="32";
//Поступил подшипник
RollerPar="0 NewUser Depot1 123456 1 12 1 1";
//статус обмена с терминалом подшипника (0 - есть подшипник)
//и его параметры
AxlePar="1 NewUser Depot1 123456 1 0 12 12";
//статус обмена с терминалом оси (1 - нет оси) и ее параметры
CommandStatus="0"; //команда успешно принята
StoreMessage="1"; //успешно выполнена
//Получаем информацию о функционировании системы
wait("GetStoreStat"); //опрос статуса склада wait("GetRollerPar");
//Получение информации о подшипнике с терминала подшипника wait("GetAxlePar");
//Получение информации об оси с терминала оси wait("SendStoreCom");
//добавление в очередь команд склада на первое место
//команды GetR (получить из приемника в ячейку) wait("GetStoreMessage");
//Получение сообщения от склада о результатах выполнения команды
//В результате первый подшипник должен быть принят
wait("GetStoreStat"); //опрос статуса склада wait("GetRollerPar");
//Получение информации о подшипнике с терминала подшипника wait("GetAxlePar");
//Получение информации об оси с терминала оси wait("SendStoreCom");
//добавление в очередь команд склада на первое место
//команды GetR (получить из приемника в ячейку) wait("GetStoreMessage");
//Получение сообщения от склада о результатах выполнения
//команды. В результате второй подшипник должен быть принят
//Задаем новое состояние окружения (входные данные)
RollerPar="1 NewUser Depot1 123456 1 12 1 1";
//статус обмена с терминалом подшипника (1 - нет подшипника)
//и его параметры
AxlePar="0 NewUser Depot1 123456 1 0 12 12";
//статус обмена с терминалом оси (0 - есть ось) и ее параметры //Получаем
информацию о функционировании системы
wait("GetStoreStat"); //опрос статуса склада
wait("GetRollerPar");
//Получение информации о подшипнике с терминала подшипника wait("GetAxlePar");
//Получение информации об оси с терминала оси
wait("SendStoreCom");//Добавление в очередь команд склада на последнее
место //команды SendR (ячейку на выход) wait("GetStoreMessage");
//Получение сообщения от склада о результатах выполнения //команды
```

```
//В результате первый подшипник для оси должен быть выдан wait("SendStoreCom");
//Добавление в очередь команд склада на последнее место
//команды SendR (ячейку на выход) wait("GetStoreMessage");
//Получение сообщения от склада о результатах выполнения //команды.
//В результате второй подшипник для оси должен быть выдан wait("SendStoreCom");
//Добавление в очередь команд склада на последнее место
//команды Term (завершение команд выдачи) wait("GetStoreMessage");
//Получение сообщения от склада о результатах выполнения
//команды
finish();
}
```

} Пример 5.1. Пример фрагмента теста (вариант 1)

При разработке тестов не обязательно дожидаться каждого события, которое должно происходить в соответствии со случаем использования. Достаточно вызвать wait для событий, после наступления которых надо менять состояние окружения. В период ожидания наступления события, заданного в wait, может происходить любое количество других событий. Все эти события будут занесены в журнал. При необходимости ждать не первого, а n-го вызова можно вызывать wait с одним и тем же параметром n раз (например, в цикле). При таком подходе тест будет гораздо короче, например приведенный выше тест будет выглядеть следующим образом:

```
class Test1:Test
{
override public void start()
{
StoreStat="32";//Пришел подшипник
RollerPar="0 NewUser Depot1 123456 1 12 1 1";//его параметры
AxlePar="1 NewUser Depot1 123456 1 0 12 12";//нет оси
CommandStatus="0";//команда успешно принята
StoreMessage="1";//команда успешно выполнена
wait("SendStoreCom");//первый подшипник принят
wait("SendStoreCom");//второй подшипник принят
RollerPar="1 NewUser Depot1 123456 1 12 1 1"; //больше нет
подшипников
AxlePar="0 NewUser Depot1 123456 1 0 12 12";//есть ось
wait("SendStoreCom");//выдача подшипника
wait("SendStoreCom");//выдача подшипника
wait("SendStoreCom");//завершение
выдачи finish();} }
```

Пример 5.2. Пример фрагмента теста (вариант 2)

Практическая работа №14. Описание тестовых процедур.

Описание тестовых процедур

Тестовые процедуры - это формальный документ, содержащий описание необходимых шагов для выполнения тестового набора. В случае ручных тестов тестовые процедуры содержат полное описание всех шагов и проверок, позволяющих протестировать продукт и вынести вердикт PASS/FAIL. Процедуры должны быть составлены таким образом, чтобы любой инженер, не связанный с данным проектом, был способен адекватно провести цикл тестирования, обладая только самыми базовыми знаниями о применяющемся инструментарии.

Как запустить тест

Для того чтобы запустить тест, нужно:

В методе Class1.Main создать экземпляр нового класса и вызвать его метод start. Так как метод start является виртуальным, то можно обращаться к тестам, используя ссылку на тип Test. Это делается так:

Test t = new <ваш класс-потомок Test>;

t.start();

Собрать и запустить приложение.

Проверка результатов выполнения тестов (сравнение с ожидаемым результатом)

После завершения теста следует просмотреть текстовый журнал теста (..\SystemTesting\ManualTests\Tests\bin\Debug\log.log), чтобы выяснить, какая последовательность событий в системе была реально зафиксирована (выходные данные) и сравнить их с ожидаемыми результатами, заданными в спецификации тестового случая №1.

Как уже упоминалось, кроме журнала теста, создается еще и соответствующий **журнал системы**. В обоих журналах содержится информация о происходивших в системе событиях. До того, как вручную сравнивать журнал теста с ожидаемыми результатами, заданными в спецификации тестового случая, вы можете использовать SystemLogAnimator (п. 14) для визуализации соответствующего журнала системы и получить наглядное (в том числе ретроспективное) представление о том, как функционировала система. Это важно при разработке тестовых случаев, потому что, не зная в деталях, как работает система, можно написать неправильный тест. Необходимо научиться отличать, что вы в действительности обнаружили - ошибку в системе или результат работы неправильного теста.

Пример неправильного теста

Предположим, что вы не поняли Приложение 2 (FS) во всех деталях и неправильно составили спецификацию тестового случая №1 и тест. Например:

```
//Задаем состояние окружения (входные данные) StoreStat="32";
//Поступил подшипник
...
//Получаем информацию о функционировании системы
wait("GetStoreStat"); //опрос статуса склада
//Вместо того, чтобы получить информацию о подшипнике с
//терминала подшипника, мы хотим получить информацию
//об оси с терминала оси
wait("GetAxlePar");
...
```

В журнале теста мы увидим следующую информацию:

```
CALL: GetStoreStat 0
RETURN: 32
CALL: GetRollerPar
RETURN: 0 NewUser Depot1 123456 1 12 1 1
```

CALL: GetAxlePar
RETURN: 1 NewUser Depot1 123456 1 0 12 12
...

Главное - суметь разобраться и исправить спецификацию тестового случая и тест, если вы сами допустили ошибку.

Практическая работа №15. Выполнение тестирования.

Задание 3

Для тестового случая №1 необходимо составить полный список всех возможных альтернативных путей (см. подраздел "Список альтернативных путей") и разработать соответствующие тесты.

Кроме того, необходимо:

- выбрать **случай использования** на основании дерева решений (..\SystemTesting\Decision Tree.vsd);
- составить пошаговое описание выбранного случая использования;
- учесть все альтернативные пути;
- составить спецификации тестовых случаев;
- разработать соответствующие **тестовые случаи (тесты)** ; составить **тестовые отчеты** (табл. 2.2).

Практическая работа №16.
Автоматизация тестирования с помощью скриптов

Общая тенденция последнего времени предусматривает максимальную автоматизацию тестирования, которая позволяет справляться с большими объемами данных и тестов, необходимых для современных продуктов.

В этом случае запуск тестов и проверка того, что тестируемая система прошла испытания на заданном **тестовом случае**, будет осуществляться автоматически.

Здесь еще раз повторим, что функции в dll были переписаны так, что они обращаются к серверу для получения информации о состоянии элементов комплекса и возвращают серверу информацию о функционировании системы. Для задания состояния окружения (**входных данных**) необходимо обратиться к серверу и передавать ему необходимую информацию. В данном случае был разработан сервер, который кроме приема и передачи информации еще осуществляет проверку правильности поведения системы (рис. 6.1). Он представляет собой модель тестируемой системы. В рамках модели заданы ожидаемые результаты и осуществляется сравнение **выходных данных** с **ожидаемыми результатами**. Хотя считалось, что разработанная модель является корректной, полной и непротиворечивой, у вас есть реальная возможность найти ошибки и в самой модели.

При разработке тестов был использован следующий подход: состояние окружения задается в тесте (**входные данные**); разработанный сервер:

- передает информацию о заданном состоянии окружении по запросу от dll; ○ получает от dll информацию о функционировании системы (выходные данные);
- сравнивает выходные данные с ожидаемым результатом.

• получаемая информация сохраняется в журнале теста;

• строится таблица покрытия FS (..\SystemTesting\ScriptTests\Logs\summary.html).

Подробное описание тестового случая №1

Изучение настоящего пункта полезно предварить ознакомлением с п.8, в котором описан подход к автоматизации тестирования с помощью языка скриптов. Здесь рассматривается тест на tcl, подобные тесты необходимо будет писать самостоятельно при выполнении заданий. Приведенный пример был разработан в соответствии со спецификацией тестового случая №1. Для простоты будем считать, что события происходят последовательно в строго заданном порядке. Реально наша система - многопоточное приложение, поэтому такое условие далеко не всегда можно гарантировать.

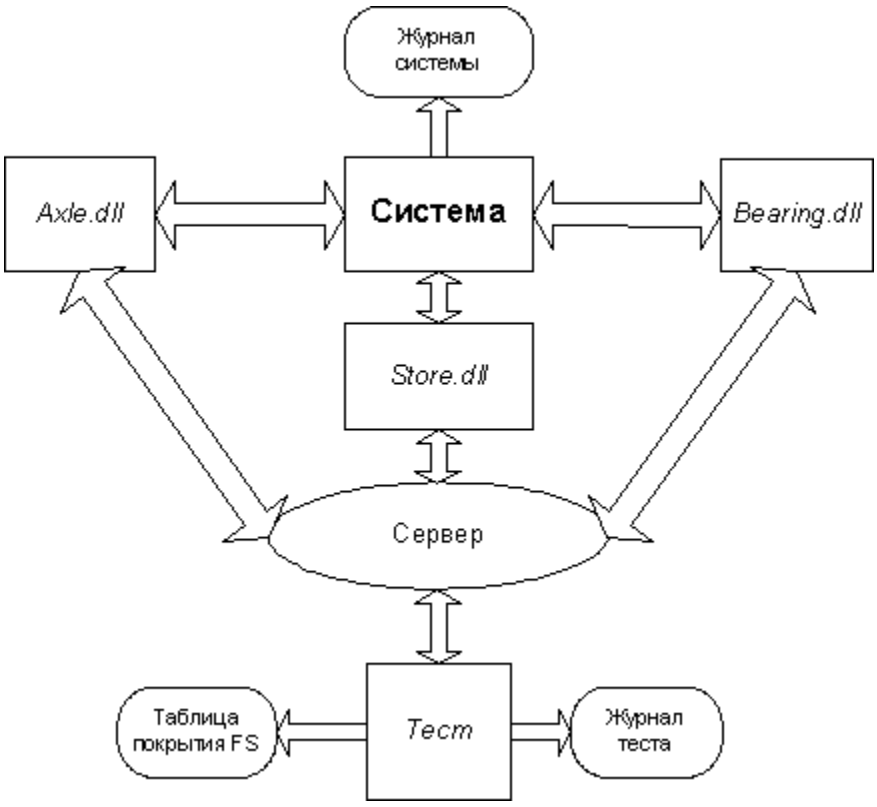


Рис. 6.1. Система и ее окружение (скрипты) source bin\\srv.tcl
//запуск сервера global StoreStat //статус склада global
RollerPar //терминал подшипника global AxlePar
//терминал оси global CommandStatus //возвращаемое
значение функции
//SendStoreCom о результатах получения команды global
StoreMessage
//сообщение от склада о результатах выполнения команды global
rollers_found //1 (можно подобрать подходящий
//подшипник или 0 (нельзя) global fds //строка для графы
"Покрытие FS" в итоговой //таблице результатов тестирования
(по умолчанию - Default) global last_command //последняя

```
команда тестируемой системы global allowed //список
разрешенных команд set fds "1.a.1; 1.a.4; 2.a; 2.c; 3.a; 4.c"
//покрывает заданные пункты FS
StartTest Warehousetest0001 //запуск теста Timeout 30
// процесс тестирования будет прерван через 30 сек
//Задаем состояние окружения (входные
данные) set StoreStat 32 //Поступил подшипник
set RollerPar "0 NewUser Depot1 123456 1 12 1 1"
//статус обмена с терминалом подшипника
//(0 - есть подшипник) и его параметры set
AxlePar "1 NewUser Depot1 123456 1 0 12 12"
//статус обмена с терминалом оси (1 - нет оси)
//и ее параметры
set CommandStatus 0 //команда успешно принята set
StoreMessage 1 //команда успешно выполнена set
rollers_found 1 //можно подобрать подходящий подшипник
//Получаем информацию о функционировании системы Wait
"GetStoreStat *" 0 1
//Неограниченное время (0) ждем получения команды ("опрос
//статуса склада") и если команда не получена, то будет
//зафиксирована ошибка (1)
Wait "GetRollerPar" 0 1
//Неограниченное время (0) ждем получения команды
//("получить информацию о подшипнике с терминала подшипника")
//и если команда не получена, то будет зафиксирована ошибка (1) set
allowed [list "GetAxlePar"]
//Получение команды "получить информацию об оси с терминала
//оси" разрешено и не должно вызвать ошибку
Wait "SendStoreCom 1 *" 10 1
//В течение 10 секунд ждем получения команды ("добавить в
//очередь команд склада на первое место команду GetR (1 -
//получить из приемника в ячейку)" и если команда за это
//время не получена, то будет зафиксирована ошибка (1) Wait
GetStoreMessage 0 1
//Неограниченное время (0) ждем получения команды
//("получить сообщение от склада о результатах выполнения
//команды") и если команда не получена, то будет
//зафиксирована ошибка (1)
//В результате первый подшипник должен быть
принят set allowed [list] Wait "GetStoreStat *" 0 1
//Неограниченное время (0) ждем получения команды ("опрос
//статуса склада") и если команда не получена, то будет
//зафиксирована ошибка (1)
```

```
Wait GetRollerPar 0 1
//Неограниченное время (0) ждем получения команды
//("получить информацию о подшипнике с терминала
//подшипника") и если команда не получена, то будет
//зафиксирована ошибка (1) set
allowed [list "GetAxlePar"]
//Получение команды "получить информацию об оси с терминала
//оси" разрешено и не должно вызвать ошибку
Wait "SendStoreCom 1 *" 10 1
//В течение 10 секунд ждем получения команды ("добавить в
//очередь команд склада на первое место команду GetR
//(1 - получить из приемника в ячейку)") и если команда за
//это время не получена, то будет зафиксирована ошибка (1) Wait
"GetStoreMessage" 0 1
//Неограниченное время (0) ждем получения команды
//("получить сообщение от склада о результатах выполнения //команды") и если команда
не получена, то будет
//зафиксирована ошибка (1)
//В результате второй подшипник должен быть принят
//Задаем новое состояние окружения (входные
данные) set StoreStat 32 //Поступил подшипник set
RollerPar {1 NA NA 0 0 0 0 0}
//статус обмена с терминалом подшипника (1 - нет подшипник)
//и его параметры set AxlePar "0 NewUser Depot1
123456 1 0 12 12"
//статус обмена с терминалом оси (0 - есть ось) и
//ее параметры
//Получаем информацию о функционировании системы Wait
"GetStoreStat *" 0 1
//Неограниченное время (0) ждем получения команды ("опрос
//статуса склада") и если команда не получена, то будет
//зафиксирована ошибка (1)
Wait "GetRollerPar" 0 1
//Неограниченное время (0) ждем получения команды
//("получить информацию о подшипнике с терминала
//подшипника") и если команда не получена, то будет
//зафиксирована ошибка (1)
Wait GetAxlePar 0 1
//Неограниченное время (0) ждем получения команды
//("получить информацию о оси с терминала оси") и если
//команда не получена, //то будет зафиксирована ошибка (1)
//В результате должна прийти ось
Wait "SendStoreCom 2 *" 10 1
```

```
//В течение 10 секунд ждем получения команды ("добавить в
//очередь команд склада на последнее место команду SendR
//(2 - ячейку на выход)") и если команда за это время не
//получена, то будет зафиксирована ошибка (1) if {![string
compare $last_command "SendStoreCom 2 9 9 9 0 0 1"]} {
//была послана команда выдать первый подшипник для оси Wait
"GetStoreMessage" 0 1
//Неограниченное время (0) ждем получения команды
//("получить сообщение от склада о результатах выполнения
//команды") и если команда не получена, то будет
//зафиксирована ошибка (1)
Wait "SendStoreCom 2 9 9 9 0 1 1" 0 1
//Неограниченное время (0) ждем получения команды //("добавить
в очередь команд склада на последнее место
//команду SendR (2 - ячейку на выход)") и если команда за
//это время не получена, то будет зафиксирована ошибка (1)
//послали команду выдать второй подшипник
}
if {![string compare $last_command "SendStoreCom 2 9 9 9 0 1 1"]} {
//если была послана команда выдать второй подшипник для оси
Wait "GetStoreMessage" 0 1
//Неограниченное время (0) ждем получения команды
//("получить сообщение от склада о результатах выполнения
//команды") и если команда не получена, то будет
//зафиксирована ошибка (1)
Wait "SendStoreCom 2 9 9 9 0 0 1" 0 1
//Неограниченное время (0) ждем получения команды
//("добавить в очередь команд склада на последнее место
//команду SendR (2 - ячейку на выход)") и если команда за
//это время не получена, то будет зафиксирована ошибка (1)
//послали команду выдать первый подшипник
}
Wait "GetStoreMessage" 0 1
//Неограниченное время (0) ждем получения команды ("получить
//сообщение от склада о результатах выполнения команды") и //если
команда не получена, то будет зафиксирована ошибка (1)
Wait "SendStoreCom 20 *" 0 1
//Неограниченное время (0) ждем получения команды
//("добавить в //очередь команд склада на последнее место
//команду Term (20 - //завершение команд выдачи)") и если
//команда не получена, то будет зафиксирована ошибка (1)
Wait "GetStoreMessage" 0 1
//Неограниченное время (0) ждем получения команды
```

//("получить сообщение от склада о результатах выполнения

//команды") и если команда не получена, то будет

//зафиксирована ошибка (1) EndTest

Листинг 6.1. Тест на tcl/tk

Практическая работа №17.

Автоматизация тестирования с помощью скриптов. Описание тестовых процедур.

Описание тестовых процедур

Как запустить тест

Запустить run.bat. В файле run.bat запускается tests.bat. Файл tests.bat содержит команды запуска тестов и установки необходимого состояния базы данных:

osql -H <Host > -S <Server > -d WarehouseTCL -U sa -P sa -i sql\ClearDB.sql

//Вызывается скрипт ClearDB.sql из подкаталога sql.

//Предполагается, что SQL Server выполняется на машине //<Host>
и называется <Server>.

//Эти параметры были настроены автоматически при //установке
практикума.

//База данных называется WarehouseTCL. Если названия отличаются,

//необходимо заменить параметры командной строки утилиты OSQL:

// -h <host> - задает имя хоста, на котором выполняется SQL Server;

// -s <server> - имя сервера;

// -d <db> - имя базы данных;

// -u <username> - имя пользователя; //
-p <password> - пароль;

// -i <script> - имя скрипта SQL. bin\launcher
tests\warehousetest0001.tcl //Вызывается
тест warehousetest0001.tcl copy log.txt
logs\warehousetest0001.txt

//Файл log.txt (лог тестируемой системы) копируется в файл

//warehousetest0001.txt. Для исключения части тестов из набора

//достаточно закомментировать соответствующие строки (команда
//REM).

Практическая работа №18.

Проверка результатов выполнения тестов.

Проверка результатов выполнения тестов (сравнение с ожидаемым результатом)

В этом случае запуск тестов и проверка того, что тестируемая система прошла испытания на заданном **тестовом случае**, осуществляется автоматически и результаты представляются в

виде таблицы; здесь, как и в предыдущем случае создается журнал теста, а также можно использовать SystemLogAnimator для визуализации журнала системы.

Пример неправильного теста

Рассмотрим тот же пример неправильного теста, как и в случае ручного тестирования:

При поступлении подшипника вместо того, чтобы получить информацию о подшипнике с терминала подшипника, мы хотим получить информацию об оси с терминала оси.

Убедитесь, что система функционирует по-другому.

Задание 4

Нужно выполнить те же задания, что и для ручного тестирования.

Практическая работа №19.

Автоматическая генерация тестов на основе формального описания.

Тесты составляются на основе спецификации требований. При формулировании требований на естественном языке существует проблема их различных толкований. Одним из способов избежать этого является применение формальных языков для описания структуры и поведения системы (UML, SDL, MSC). Кроме того, описание требований на формальном языке является формальным описанием тестовых случаев, на основе которого можно генерировать тестовый код. В практикуме для создания тестов будет использоваться язык диаграмм взаимодействия (Message Sequence Charts, MSC - п.11). В этом случае под тестом мы будем понимать его представление в виде MSC- диаграммы.

В Практикуме для реализации тестирования используется учебная система автоматизации тестирования TAT - Test Automation Training. На вход система принимает формальное описание тестов в виде MSC диаграмм (в текстовом формате MSC PR). На основе этих MSC диаграмм и конфигурационного файла (в формате XML), который описывает интерфейс тестируемой системы, генерируется тест на C#. (Интерфейс тестируемого приложения (Application Under Test - AUT) содержит сигналы, сообщения, транзакции, которые система может посылать тестовому окружению или может принимать от тестового окружения). Для запуска системы с этим тестом необходимо написать Wrapper, который транслирует сигналы от теста к системе и наоборот.

Таким образом, методика тестирования системы с помощью TAT выглядит следующим образом:

- написать Wrapper для тестируемой системы;
- создать файл конфигурации;
- создать формальное описание тестов в виде MSC-диаграмм;
- нарисовать MSC-диаграммы в MS Visio;
- сгенерировать с помощью макроса тестовый файл в формате MPR;
- настроить в ConfigTAT проект теста (указать пути) или набора тестов;
- запустить тест или набор тестов;
- проанализировать получаемые log-файлы.

В данном случае wrapper и файл конфигурации (первые два пункта методики) уже созданы, поэтому вам необходимо будет выполнить только п. 3-6.

В рассматриваемом подходе не только запуск тестов и проверка результатов прогона **тестового случая** будут осуществляться автоматически, но и сам тестовый код будет генерироваться автоматически на основе MSC-диаграммы (рис. 7.1).

При разработке тестов был использован следующий подход:

Когда реализуется определенное событие, модель посылает сигнал запроса состояния к тестовому окружению.

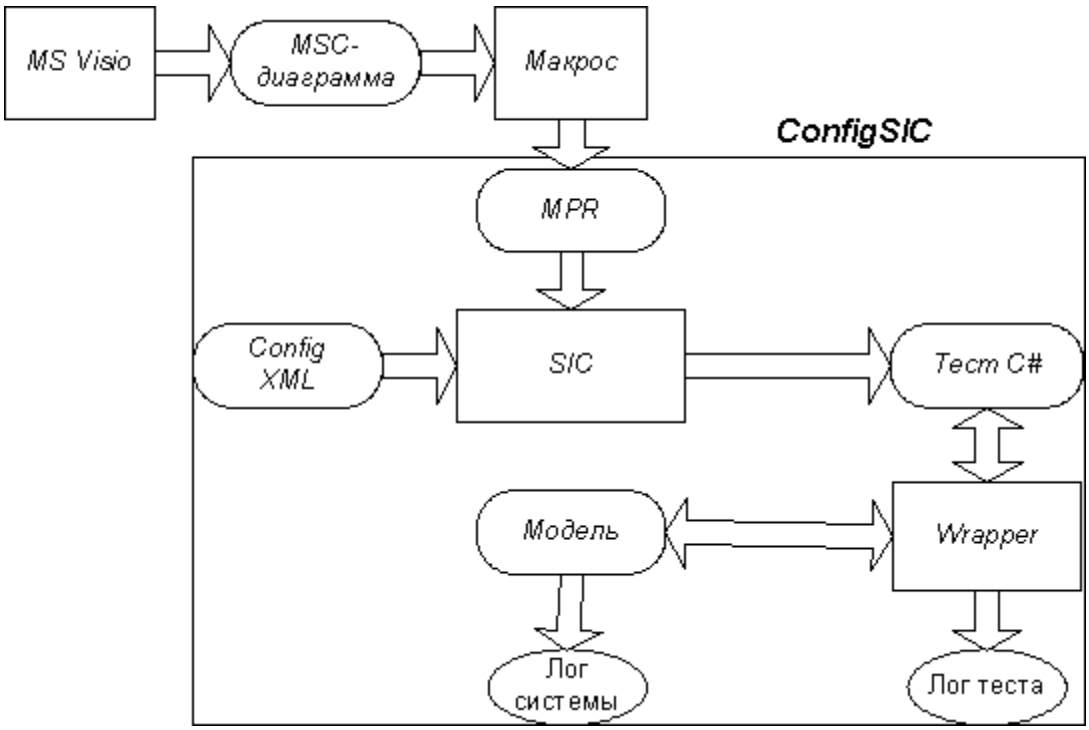


Рис. 7.1. Система и ее окружение (автоматическая генерация)

Состояние окружения задается в тесте (**входные данные**) в виде параметров сигналов. Тест возвращает состояние окружения (StoreStat, AxlePar, RollerPar, StoreMessage, CommandStatus), посылая модели сигнал с параметрами в соответствии с запросом.

Получаемая информация сохраняется в журнале теста.

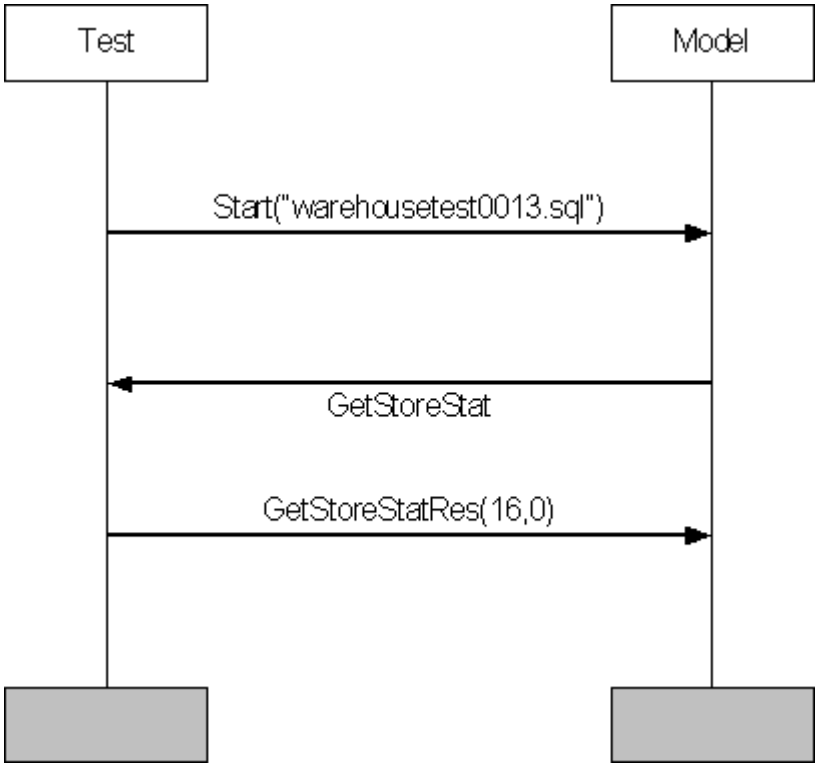


Рис. 7.2. Взаимодействие теста и модели (MSC-диаграмма)

Практическая работа №20. Описание тестовых процедур.

Подробное описание тестового случая №1

Изучение материала настоящего пункта полезно предварить ознакомлением с п. 9, содержащим описание подхода к автоматической генерации тестов на основе MSC. Здесь рассматривается тест, представляющий собой MSC-диаграмму, созданную в Visio. Подобные тесты необходимо будет разработать (нарисовать) самостоятельно при выполнении заданий. Приведенный пример был разработан в соответствии со спецификацией тестового случая №1 (рис. 7.3).

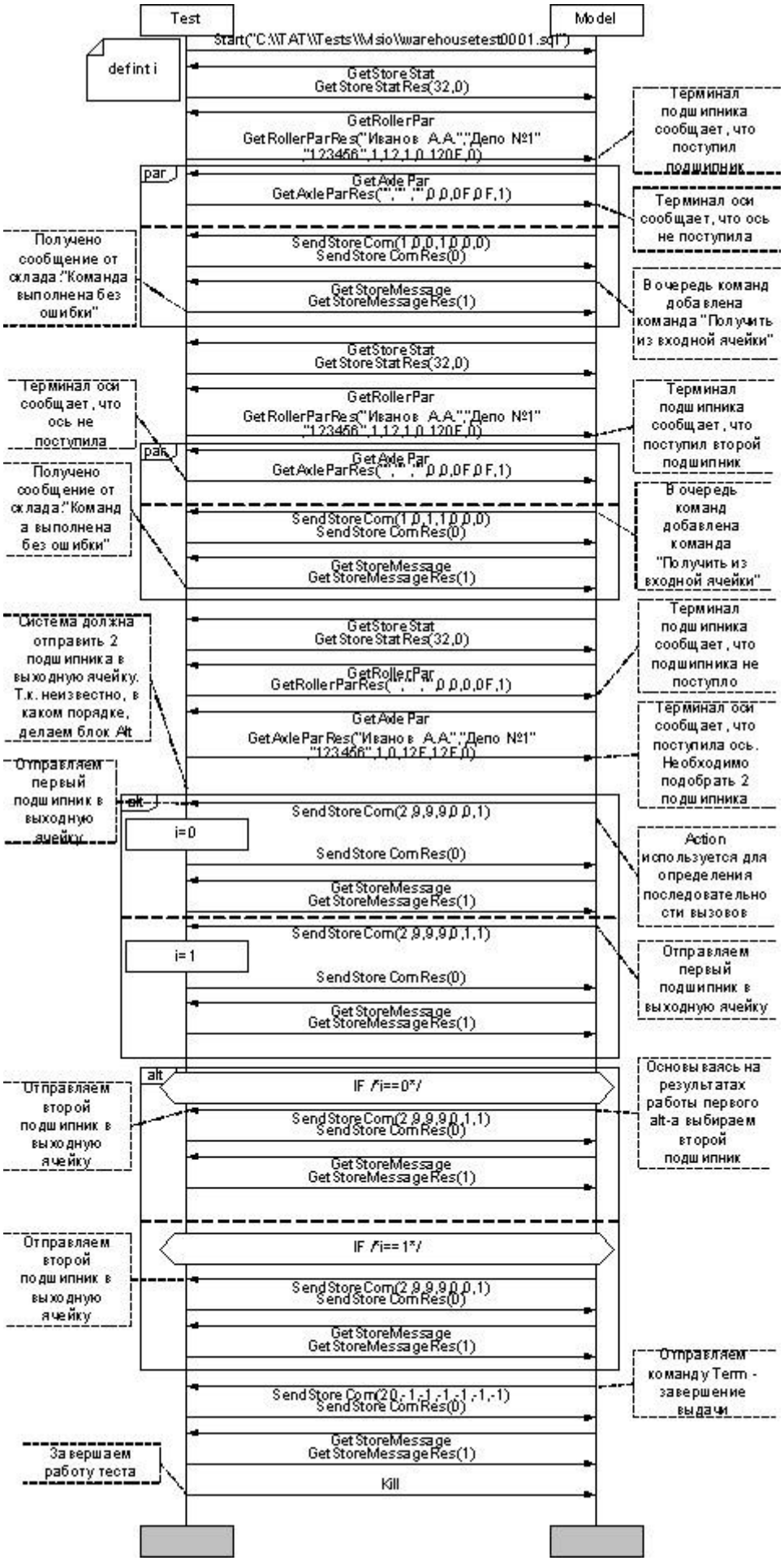


Рис. 7.3. Тестовый случай №1

Описание тестовых процедур

Как сгенерировать и запустить тест

Изучение материала настоящего пункта полезно предварить ознакомлением с п. 12, 13, содержащими описание использования MS Visio для генерации MPR файлов и описание конфигурирования - ConfigTAT.

На данном этапе используется тест, представляющий собой MSC- диаграмму, созданную в Visio. Дальнейшие действия описываются следующей методикой:

1. Запустить Microsoft Visio.
2. Загрузить Stensil (File->Open->MSC.VSS или File->Open Stensil ->MSC.VSS).

Visio выдаст предупреждение о том, что данный stensil содержит макросы. На предупреждение следует ответить "Enable macros".

3. Открыть существующий тестовый случай №1 - Warehousetest0001 (..\SystemTesting\TATTests\Tests\Tests\Tests.vsd).
4. Для генерации MPR вызвать следующий макрос: Tools->Macros ->MSC>Module1->Parse. В указанной папке будет создан MPR-файл с именем, соответствующим имени текущей страницы в Visio (..\SystemTesting\TATTests\Tests\WarehouseTest1\warehousetest0001.mpr).
5. Запустить ConfigTAT.
6. В меню File -> Open выбрать тестовый случай №1.
7. Выбрать настройки - установить по умолчанию (Set ALL to default).
8. Запуск - генерация и запуск теста (Run ALL).

Практическая работа №21. Проверка результатов выполнения тестов.

Проверка результатов выполнения тестов (сравнение с ожидаемым результатом)

В этом случае запуск тестов и проверка того, что тестируемая система прошла испытания на заданном **тестовом случае**, осуществляется автоматически, как и в предыдущем случае создается журнал теста, а также можно использовать SystemLogAnimator (п.14) для визуализации журнала системы.

Для просмотра протоколов тестирования надо использовать группу "Test Logs" ConfigTAT и можно просматривать:

- протокол тестирования в виде html-страницы (HTML log).
- протокол тестирования в виде txt файла (Text-log).
- протоколы в формате mpr (отдельный протокол для каждого testcase- а и каждой итерации теста), которые можно открыть в программе Telelogic нажатием кнопки "View" (MPR logs).

Пример теста с ошибкой

На рис. 7.4 представлена диаграмма теста с ошибкой. Используя FS, необходимо объяснить причину некорректности тестового случая.

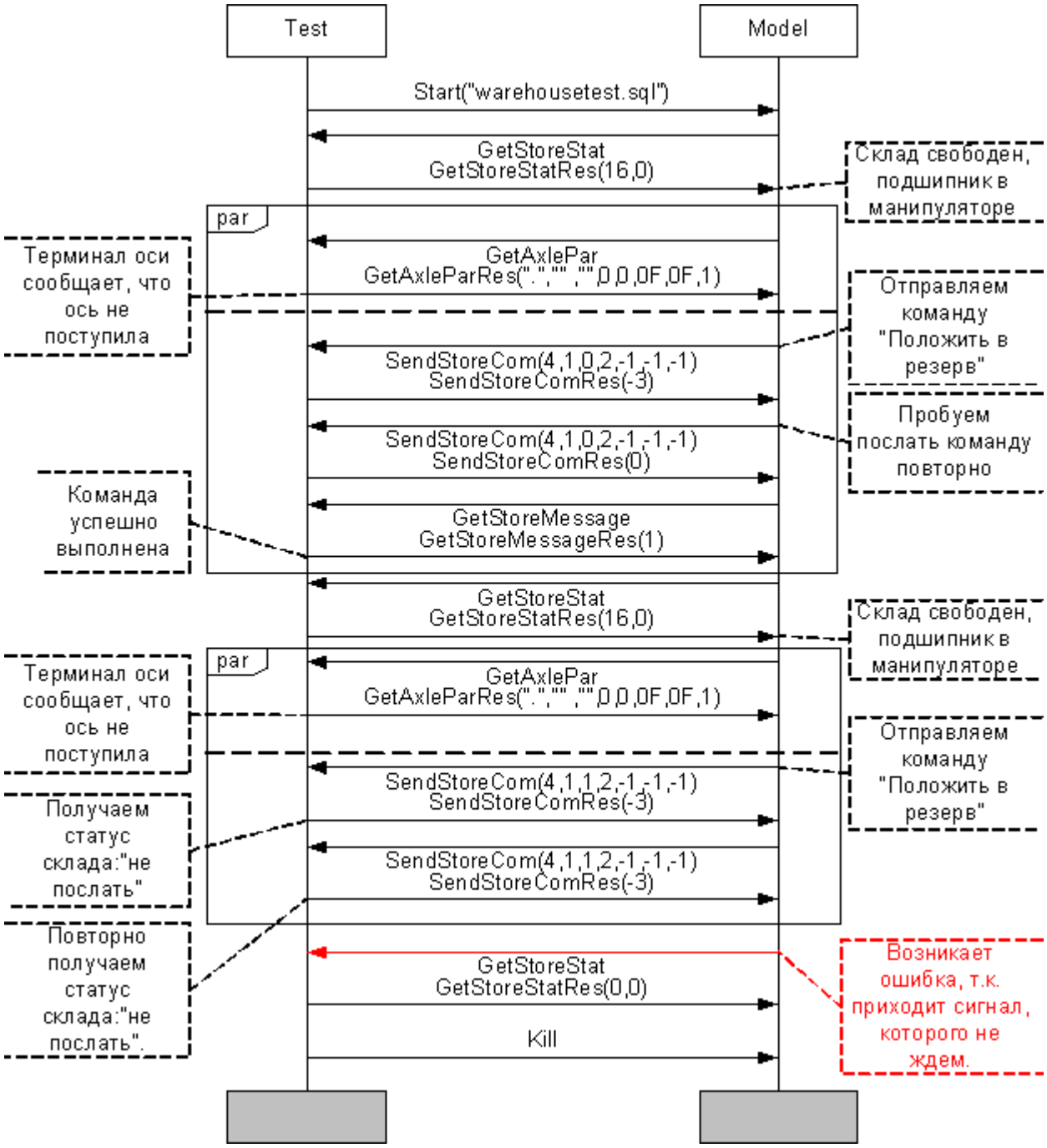


Рис. 7.4. Тест с ошибкой **Задание 5**

Для случая автоматического тестирования на основе MSC-диаграмм нужно повторить те же задания, что и для ручного тестирования.

Практическая работа №22. Описание ручного тестирования.

Реализация выбранного подхода для ручного тестирования приводится в Class1.cs (..\SystemTesting\ManualTests\Tests\Class1.cs).

Все классы входят в пространство имен Tests. Это пространство имен содержит следующие классы:

- Class1 - главный класс приложения. Содержит статический метод Main, вызываемый при запуске;
- Test - абстрактный (abstract) класс, реализующий общую для всех тестов функциональность. Содержит следующие методы:
  - public Test() - конструктор. Создает серверный сокет и запускает сервер; ○ protected void wait(string st) - ожидает получения от dll вызова, начинающегося со строки st ;
  - protected void finish() - обрабатывает последний запрос от dll и закрывает серверный сокет; ○ virtual public void start() - запускает тест. В каждом конкретном тесте переопределяется.

Кроме того, класс Test содержит пять protected полей типа string:

- StoreStat - статус склада;
- AxlePar - терминал оси;
- RollerPar - терминал подшипника;
- CommandStatus - возвращаемое значение функции SendStoreCom ; □ StoreMessage - сообщение от склада.

Как создать свой тест?

Для создания нового теста необходимо выполнить следующие действия:

- создать новый класс, являющийся потомком Test ;
- переопределить в нем метод start(), чтобы реализовать функциональность теста:
  - задать состояние окружения (StoreStat, AxlePar, RollerPar, StoreMessage, CommandStatus); ○ ждать, когда произойдет определенное событие (вызов wait, в котором надо задать строку для выхода из состояния ожидания); ○ задать новое состояние окружения и т.д.
- □ тест должен завершаться вызовом finish(). В

общем виде тест выглядит так:

```
override public void start()
{ <задание переменных> wait(<строка>);
<задание переменных>
wait(<строка>);
....
finish();
}
```

Практическая работа №23. Автоматизация тестирования с помощью скриптов. Создание своих тестов.

Как создать свой тест?

В данном случае используется тот же принцип, что и при ручном тестировании:

- задать состояние окружения (StoreStat, AxlePar, RollerPar, StoreMessage, CommandStatus);
- ждать, когда в системе произойдет определенное событие;

- задать новое состояние окружения; и т.д.

Каждый тест должен быть представлен в следующем виде:

ЗАГОЛОВОК

БЛОК

Wait <условие>

БЛОК

Wait <условие>

.....

EndTest

Опишем эти элементы более подробно.

Описание заголовка

Заголовок теста состоит из:

- команды запуска сервера;
- команд global, устанавливающих доступ к глобальным переменным состояния окружения;
- команды StartTest, задающей имя теста (оно не обязательно должно совпадать с именем файла).

```
source bin\\srv.tcl   // запуск сервера global
StoreStat      // статус склада global AxlePar
// терминал оси global RollerPar      //
терминал подшипника
global CommandStatus  // возвращаемое значение функции
                // SendStoreCom
global StoreMessage   // сообщение от склада
global rollers_found  // 1 (можно подобрать подходящий
                // подшипник) или 0 (нельзя)
global fds            // строка для графы "Покрытие FDS"
// в итоговой таблице результатов
                // тестирование (по умолчанию - Default) global
last_command   // последняя команда тестируемой системы
global allowed        // список разрешенных команд, их
                // получение должно вызывать ошибку
StartTest <имя>       // запуск теста
```

Описание блока

Каждый блок устанавливает значения одной или более переменных окружения. Например:

```
Set StoreStat 32
...
```

Практическая работа №24. Описание команд.

Описание Wait

Команда Wait используется для изменения состояния тестового окружения в ответ на действия системы.

При вызове процедуры Wait скрипт входит в состояние ожидания. Обращение со стороны системы приводит к завершению состояния ожидания. Если условие выполнено, то ответ на это обращение будет сформирован на основе следующего блока, иначе - на основе предыдущего.

Wait <команда> <время> <обязательно>

<команда> определяет команду системы, которая приведет к переходу в следующий блок. Можно использовать стандартные символы подстановки *, ?, [];

<время> определяет время ожидания. Если задать 0, то время не ограничено;

<обязательно> дает возможность не формировать ошибку, если за заданное время требуемая команда не была получена. Если задана 1, то в случае неполучения команды будет сформирована ошибка, а если 0, то ошибки не будет.

Примеры использования:

Wait ZZZZZZZ 10 0 - ждать 10 секунд (команда ZZZZZZZ является недопустимой и не может быть получена)

Wait "GetStoreStat *" 0 1 - ждать команды GetStoreStat с любыми параметрами неограниченное время

Wait "SendStoreCom 2 *" 10 1 - ждать команды SendStoreCom с первым параметром 2 и произвольными остальными параметрами в течение 10 секунд, если она не получена, будет зафиксирована ошибка

Wait ZZZZZZZ 0 0 - бесконечное ожидание.

Так как ожидание в процедуре Wait прерывается только при поступлении запроса от системы, то в случае отсутствия запросов (система зависла и ничего не делает) тест также останавливается. Для предотвращения такой ситуации используется следующая команда: Timeout <время>.

<время> определяет число секунд, после которого процесс тестирования будет прерван. При этом в журнал теста будет занесено сообщение TIMEOUT. Срабатывание таймаута приводит к завершению теста. Команда Timeout 0 отменяет ограничение времени.

Описание allowed

Рассмотрим переменную allowed более подробно. Она содержит список тех команд, получение которых разрешено и не должно вызывать ошибку. Например, в следующем примере вызов GetAxlePar не спровоцирует ошибку:

```
Set StoreStat 32
Wait "GetStoreStat *" 0 1
Wait "GetRollerPar" 0 1
Set allowed [list "GetAxlePar"]
Wait "SendStoreCom 1 0 0 1 0 0 0" 0 1
```

Однако ошибка все же возникнет, если данная команда в данной ситуации является недопустимой. Несмотря на то, что проверка реализована в скрипте srv.tcl, использование команды Wait и списка allowed могут ужесточить проверку. Значение по умолчанию для allowed - " * ". В этом случае проверку проходят все команды. Вход в процедуру Wait вызывает расширение списка allowed до момента выхода из нее, так что ожидаемая команда системы всегда проходит проверку на принадлежность списку allowed. В списке allowed можно задействовать символы подстановки.

Практическая работа №25. Описание автоматической генерации MSC тестов

Как создать свой тест?

В данном случае под тестом мы будем понимать его представление в виде MSCдиаграммы. В качестве объектов мы будем рассматривать тест (Test) и тестируемую систему (Model). Обмен сообщениями между тестом и моделью показан на рис. 10.1.

Когда в системе происходит определенное событие, она запрашивает состояние окружения, посылая сигнал тестовому окружению.

Тест возвращает состояние окружения (StoreStat, AxlePar, RollerPar, StoreMessage, CommandStatus), посылая модели сигнал с параметрами в соответствии с запросом.

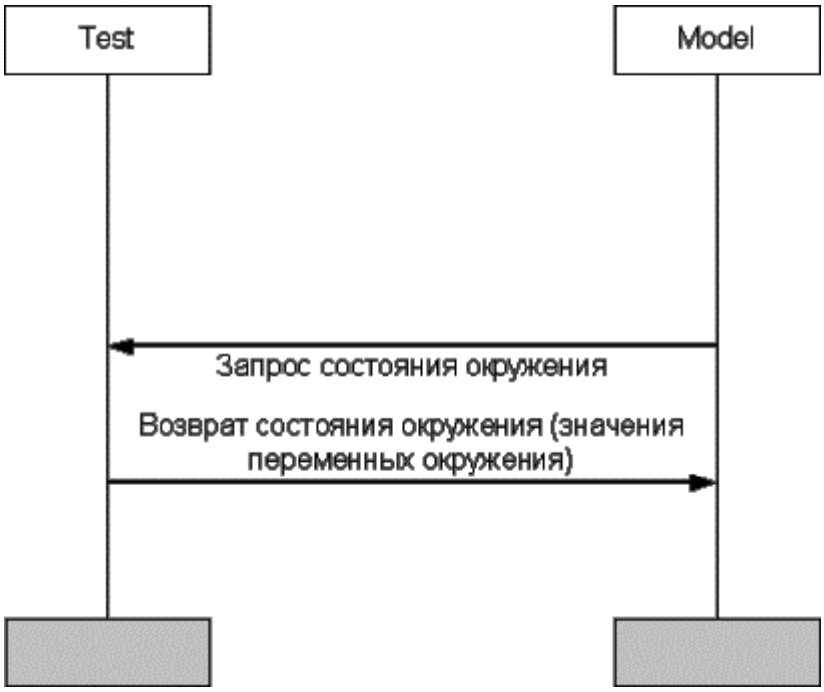
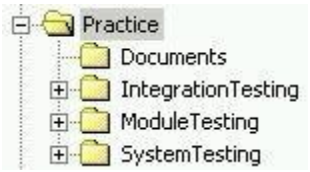


Рис. 10.1. Взаимодействие теста и модели

Структура и описание содержимого каталогов

В папке Documents находится:

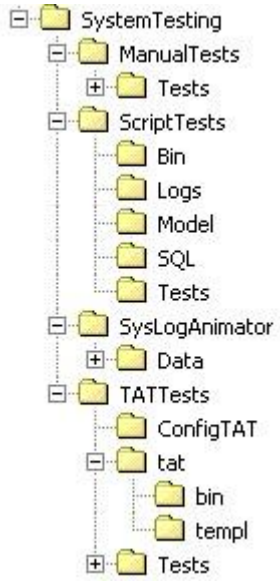
- FDS;
- HLD;
- Практикум (этот документ).



Папка IntegrationTesting содержит проект Visual Studio .NET с примером интеграционного теста.

Папка ModuleTesting содержит проект Visual Studio .NET с примером модульного теста.

В папке SystemTesting\ManualTests содержится проект Visual Studio .NET с примерами системных ручных тестов.



В папке SystemTesting\ScriptsTests содержатся примеры тестов с использованием скриптов:

\bin - содержит программу launcher.exe, файл srv.tcl и вспомогательные скрипты header.tcl и footer.tcl, используемые для формирования структуры html-отчета. Программа launcher.exe запускает тестируемую систему и тестовый скрипт на выполнение, а после завершения теста завершает выполнение системы.

\logs - log-файлы пройденных тестов (*.log), log-файлы тестируемой системы (*.txt) и общий отчет summary.html.

\model - исполнимые файлы тестируемой системы.

\sql - SQL скрипты для установки заданного состояния базы данных.

\tests - тестовые скрипты на языке TCL.

Папка SystemTesting\TATTests содержит:

\ConfigTAT - программа ConfigTAT для настройки и запуска тестов TAT:

\tat - система автоматизации тестирования TAT.

\Tests\config\config.xml - xml файл, описывающий тестируемую систему (AUT), тестовое окружение и сигналы между ними.

\Tests\Model - тестируемая система application under test.

\Tests\mpr - mpr-файлы, описывающие тесты.

\Tests\SQLScripts - sql-скрипты для подготовки базы данных к конкретному тесту.
\Tests\Tests - stencil для Visio и MSC-диаграммы тестов в Visio.

\Tests\WareHouseTest№ - рабочие папки тестов.

\Tests\ *.tcf, tests.ltc - сохраненные конфигурации тестов и Testsuite- а для ConfigTAT.

Папка SysLogAnimator содержит основные файлы: библиотека DirectX, запускаемый файл LogAnimator.exe и конфигурационный файл LogAnimator.cfg.

Папка SysLogAnimator\Data содержит необходимые для работы аниматора графические файлы.

Практическая работа №26. Язык диаграмм взаимодействия.

Описание MSC

Язык диаграмм взаимодействия (Message Sequence Charts, MSC) - это язык описания поведения системы в виде последовательности событий. События могут относиться к отдельным компонентам системы, к взаимодействиям между компонентами системы либо к взаимодействию между системой и ее окружением. Основное назначение диаграмм взаимодействия - описание последовательностей допустимых взаимодействий между компонентами системы и системой и ее окружением. Диаграммы изображаются в графическом виде, но существует также текстовая форма MSC-диаграмм. Обе формы переводятся взаимно однозначно друг в друга.

Разновидности диаграмм взаимодействия используются при разработке систем реального времени с 1960-х годов. Особое распространение диаграммы взаимодействия получили в области разработки телекоммуникационных систем. Язык диаграмм взаимодействия стандартизован в 1992 году Международным Телекоммуникационным Союзом (ITU-T) (Рекомендация Z.120 1992). В настоящее время принята новая, значительно расширенная версия стандарта (Рекомендация Z.120 1996.).

Основные понятия

Диаграмма взаимодействия описывает последовательности событий, происходящих с набором объектов (системой взаимосвязанных компонентов). Дополнительно каждая система рассматривается как открытая, т.е. подразумевается наличие некоторого окружения системы, с которым система взаимодействует. Окружение также может задаваться в виде отдельного объекта.

Основным понятием диаграммы взаимодействий является трасса объекта. Для каждого объекта на диаграмме имеется отдельная вертикальная ось. На этой оси откладываются события, имеющие отношение к данному объекту. Считается, что все объекты существуют одновременно, и последовательности событий объектов развиваются параллельно. При описании объекта используются стартовый (прозрачный прямоугольник) и конечный (черный прямоугольник) символы объекта, обозначающие соответственно начало и конец описания объекта в данной MSC-диаграмме.

Взаимодействие между объектами (а также между объектом и окружением системы) осуществляется только при помощи обмена сообщениями (рис. 10.2).

Сообщение моделирует взаимодействие (т.е. обмен информацией) между двумя объектами системы или между объектом и окружением системы. С точки зрения системы, взаимодействие между двумя объектами разбивается на два сопряженных события: посылка сообщения одним объектом и прием сообщения другим объектом (рис. 10.2). Сообщения, приходящие из окружения системы, моделируются одним событием приема сообщения, а события, посылаемые в окружение, моделируются одним событием посылки сообщения. Сообщение имеет имя. Имя сообщения задает тип взаимодействия. Диаграмма может описывать несколько обменов сообщениями с одинаковым именем. Для уникальной идентификации конкретного обмена предусмотрен так называемый уникальный идентификатор обмена (message instance name), однако он используется только в текстовом представлении для снятия неоднозначности в описании сопряженных событий у различных объектов. В графическом представлении такой проблемы не возникает, так как сопряженные события представляются различными концами одного и того же графического объекта (стрелки от трассы одного объекта к трассе другого).

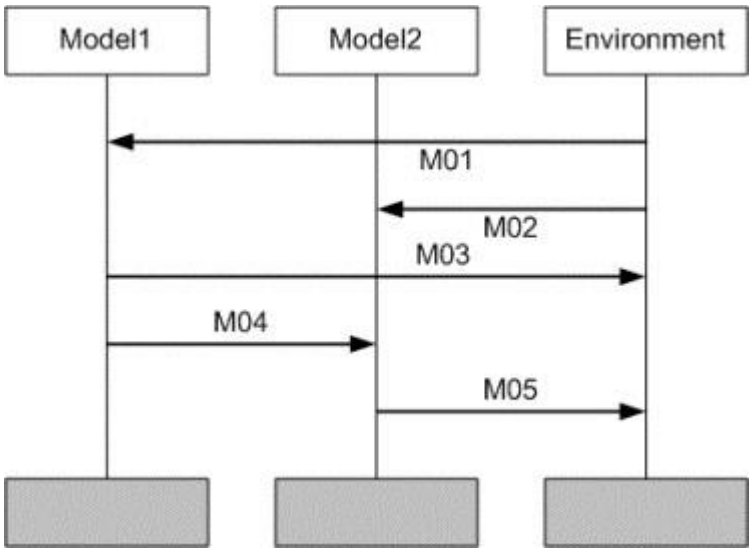


Рис. 10.2. Пример взаимодействия между объектами в MSC

Практическая работа №27.
Функции языка диаграмм взаимодействия.

Дополнительно, язык диаграмм взаимодействия позволяет описывать передачу информации в сообщении (рис. 10.3). С сообщением может быть связан список параметров. Каждый параметр моделирует передачу конкретной информации от одного объекта к другому. Язык диаграмм взаимодействия не определяет семантику параметров сообщения.

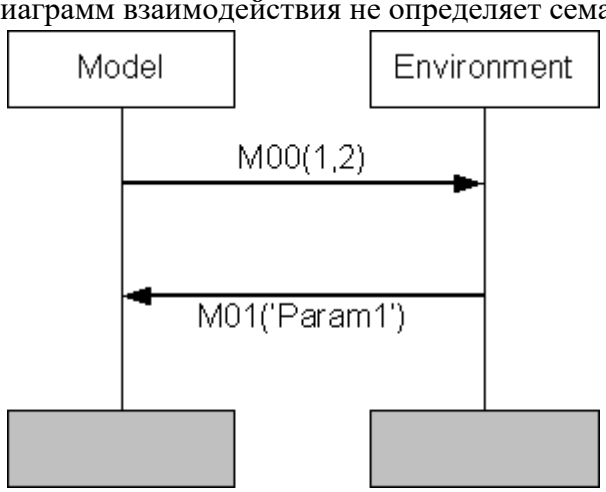


Рис. 10.3. Передача параметров сообщений

MSC-диаграммы позволяют создавать более сложные описания поведения системы с помощью специальных операторов. В MSC'96 используется четыре типа операторов: alt - альтернативный оператор, par - параллельный оператор, loop - итерация, opt - опциональная область.

Графически операторные конструкции изображаются в виде прямоугольника с пунктирными линиями в качестве разделителей. Ключевое слово оператора располагается в левом верхнем углу.

Альтернативная композиция (рис. 10.4) позволяет задавать альтернативное выполнение секций MSC-диаграммы. Только одна альтернатива может быть реализована.

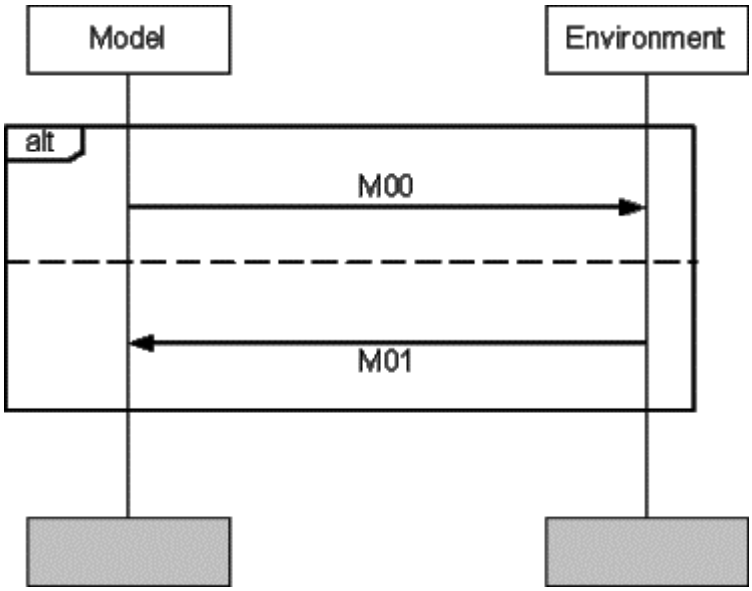


Рис. 10.4. Альтернатива

Операция par имеет структуру, аналогичную конструкции alt (рис. 10.3), и определяет параллельное выполнение секций. Это означает, что все события внутри параллельных секций будут выполнены. Единственным ограничением является то, что порядок событий в каждой секции будет сохранен.

Конструкция loop (рис. 10.5) имеет несколько форм. Наиболее общая форма - loop <n, m>, где n и m - натуральные числа. Это означает, что конструкция может быть выполнена от n до m раз. Вместо натурального числа может использоваться ключевое слово inf, обозначающее бесконечность.

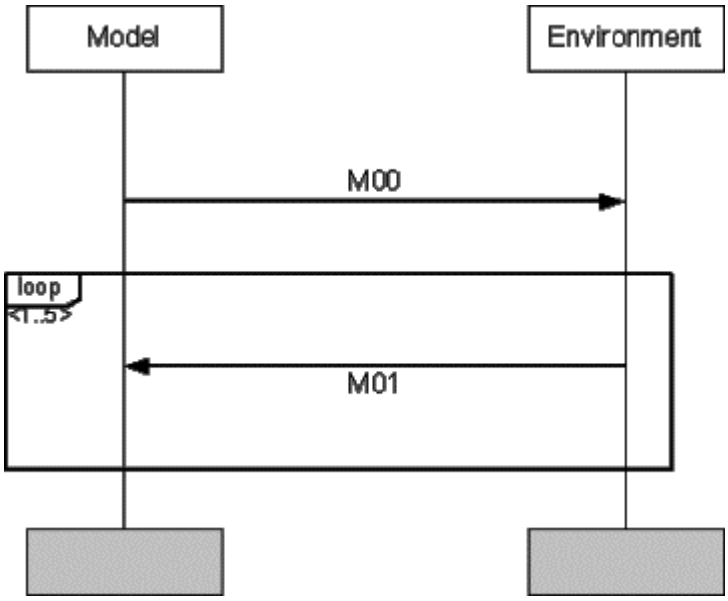


Рис. 10.5. Цикл

Оператор opt имеет структуру, аналогичную loop, но без операндов, и обозначает то же, что и оператор alt с пустой MSC в качестве второго операнда.

Одним из важных понятий в MSC является условие или состояние. Состояние - это особое событие на трассе объекта. В отличие от прочих событий, одно и то же состояние может разделяться одним, двумя и более объектами. По числу объектов на диаграмме, разделяющих некоторое состояние, различают глобальные состояния (общее для всех объектов), разделяемые состояния (разделяемые несколькими, но не всеми объектами) и локальные состояния

(разделяемые единственным объектом). Если два объекта разделяют одно и то же состояние, то сопряженные события приема и посылки сообщений должны происходить либо оба до соответствующего состояния, либо оба после соответствующего состояния.

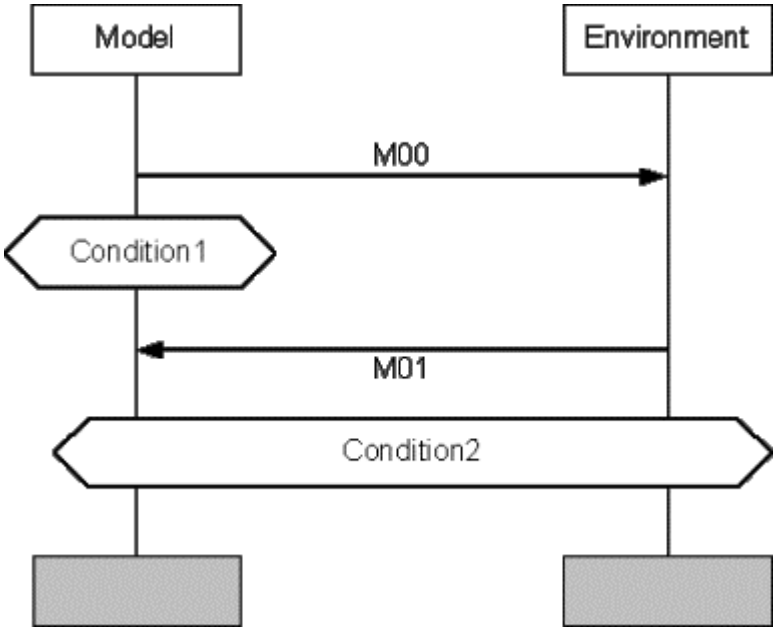


Рис. 10.6. Состояние (условие)

Основной недостаток стандартной MSC - невозможность описать отношения данных в параметрах сообщения. Эта проблема решается с использованием некоторого расширения стандартного MSC - mMSC (macro MSC). Основные из этих расширений включают в себя:

Практическая работа №28.
Работа с макроподстановками и диаграммами.

Макроподстановки (рис. 10.7). Они позволяют создавать множество MSC-диаграмм с одинаковой структурой и разными параметрами сообщения, циклами и т. п. Макроподстановки начинаются с символа # и могут быть константами или функциями.

Функции, кроме названия, содержат параметры, заключенные в скобки.

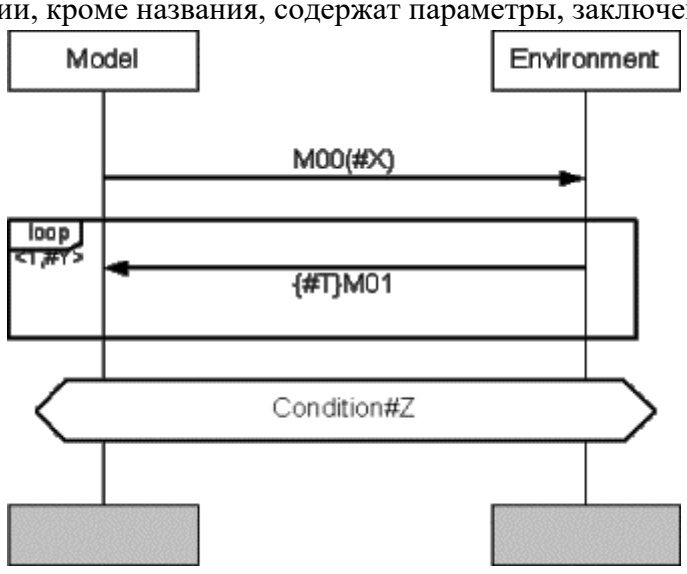


Рис. 10.7. Макроподстановки

Типы макроподстановок описываются в определенном файле, и на основе этой информации вместо них подставляются конкретные значения.

Временные ограничения служат для указания времени посылки/приема сообщения, его длительности и типа (синхронное с явно заданным моментом выполнения, асинхронное). Время может задаваться относительно начала работы (абсолютное), относительно предыдущего сообщения (относительное) или относительно метки. На рис. 10.8 сообщение M01 должно отправиться через 5 единиц времени, в течение 5 единиц после получения сообщения M00 (указано с помощью метки).

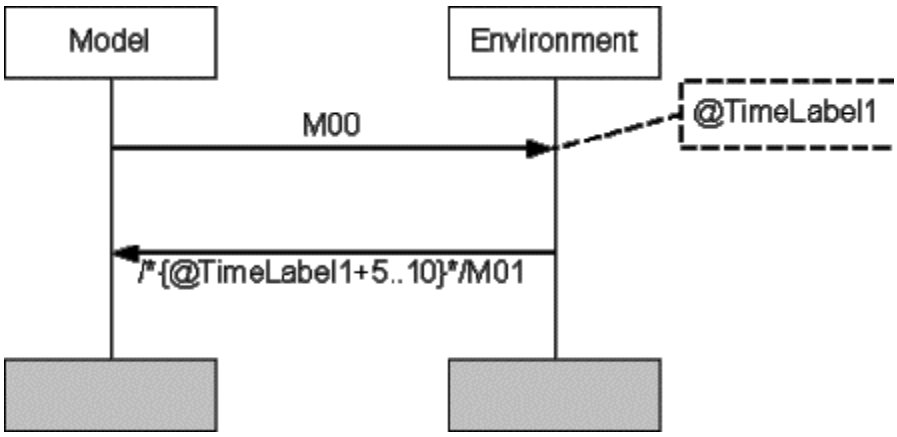


Рис. 10.8. Использование времени

Применение MSC-диаграмм для описания поведения моделей

Благодаря своему главному преимуществу - ясному графическому представлению, которое дает интуитивное понимание поведения описываемой системы, MSC-диаграммы широко применяются для различных целей:

- для определения требований;
- как спецификация интерфейсов;
- как спецификация взаимодействия процессов;
- как базис для генерации тестов;
- для документации;
- для объектно-ориентированного анализа и разработки.

Хотя изначально MSC-диаграммы предназначались для описания телекоммуникационных систем, сейчас они с успехом применяются и в других областях.
Обработка MSC-диаграмм

В случае достаточно сложных систем могут быть обнаружены ошибки, возникшие как при создании диаграммы, так и при проектировании тестируемой системы. Исправление тех же ошибок, если они будут обнаружены на более позднем этапе жизненного цикла продукта, потребует гораздо больших затрат.

Проверка MSC-диаграммы на полноту

Большинство систем или их частей можно в том или ином виде представить с помощью механизма state-машин. Тогда каждому состоянию системы на диаграмме будет соответствовать условие - конструкция condition. Для каждого события диаграммы (под событием будем понимать сообщение или действие - action) можно составить множество всех состояний, которые могут непосредственно предшествовать ему (предусловия). В этом случае проверка на полноту MSC-диаграммы будет заключаться в проверке того, все ли возможные случаи предусловий для каждого события представлены в ней. Если это не так, то, возможно,

диаграмма не описывает полностью все возможные сценарии работы системы, и множество тестов, сгенерированных по этой диаграмме, будет неполным.

Практическая работа №29. Генерация MPR-файлов

Разработанный набор утилит предназначен для:

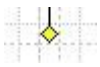
- преобразования MSC-диаграмм в формат MPR;
- проверки правильности подключения сигналов;
- загрузки комментариев.

Утилита поддерживает следующие типы конструкций:

Instance
Instance End
Message
Action
Comment
Coregion
Text
Condition
Reference
Block(Alt, Par, Loop, Opt).

При разработке диаграмм соединение объектов осуществляется при помощи Connector Point. Если какая-либо конструкция языка не будет присоединена, то будет выдано соответствующее сообщение об ошибке, "неправильный" элемент будет выделен красным цветом, а MPR-файл генериться не будет.

Исключение составляет элементы: Block, Condition, Reference и Text. **Описание элементов**



Instance представляет собой один из взаимодействующих объектов. Необходимо задать имя данного элемента. После завершения приема/выдачи всех сигналов к Instance присоединяется блок Instance End. Для увеличения длины Instance TimeLine, его необходимо выделить и увеличить длину, используя Control.

Instance End самостоятельно не используется, применяется только совместно с конструкцией Instance. Используется для сигнализации того, что данный объект закончил принимать/посылать сигналы.

Message (событие/сообщение) представляет собой взаимодействие между объектами (Instances). Необходимо задать имя сообщения и параметры (если они необходимы). Messages должны быть присоединены к Instance с помощью Connector Points. Messages без имени не обрабатываются, и выдается сообщение об ошибке.

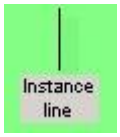
Action используется для отражения события, выполненного в рамках одного Instance. Блок необходимо присоединить с помощью Connector Point-а. Action без имени не обрабатывается, и выдается соответствующее сообщение об ошибке.

Comment используется для написания комментариев. Его требуется присоединить к Connector Point на Instance. Для увеличения длины необходимо использовать Control.

Coregion показывает, что сигналы, поступившие в рамках данного элемента, могут приходить в разном порядке. Coregion должен быть присоединен с использованием Connector Points, в противном случае возникает ошибка.

Text - это блок текстовых комментариев или описаний. Может располагаться в любой части рабочего листа.

Condition используется для объявления события, которое распространяется на несколько объектов (Instance). Может использоваться в качестве точки синхронизации. Необходимо задать имя или другие параметры. При использовании требуется "растянуть" на используемые оси Instance. Если условие Condition не распространяется на один из объектов (Instance), то поверх блока Condition задается Instance Line (рис. 11.1 и 11.2).



Instance Line самостоятельно не применяется, а только совместно с блоком Condition. Для ее присоединения используем Connector Points.

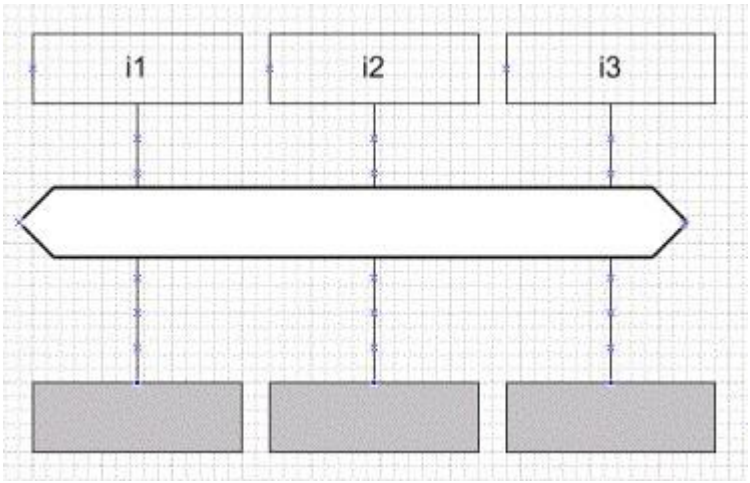


Рис. 11.1. Изображение Condition без Instance Line

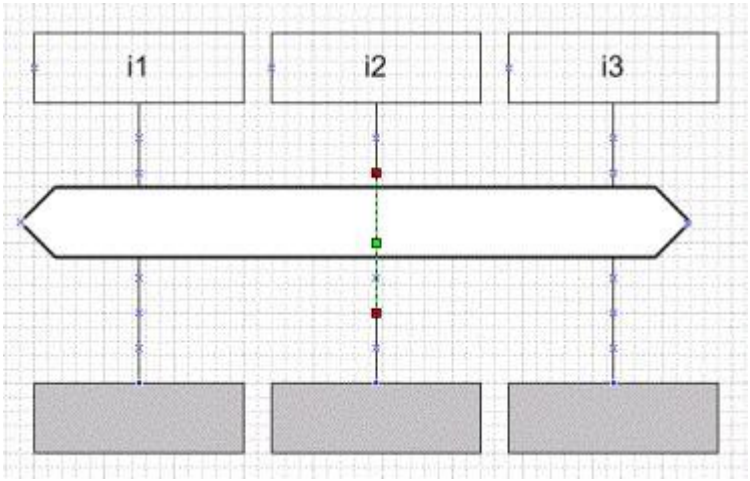


Рис. 11.2. Изображение Condition с Instance Line

Reference применяется для определения ссылки на другую диаграмму (MPR-файл). Необходимо указать путь к используемой диаграмме. При отсутствии имени (пути) возникает ошибка.

Block позволяет задавать имя блока, используется совместно с элементом Separator, который разделяет Block на 2 и более фреймов. В заглавии блока необходимо указать его название и параметры (если необходимо):

Alt - указывает, что может выполняться один из фреймов в определенной последовательности.

Par - указывает, что сообщения, которые были объявлены в рамках данного блока, будут выполняться параллельно.

Opt - указывает на то, что данный фрейм может использоваться опционально.

Loop - указывает на то, что данный блок будет повторяться в цикле указанное число раз (указывается в параметрах).

Separator самостоятельно не используется, а применяется только совместно с элементом Block. При присоединении необходимо задействовать Connector Point.

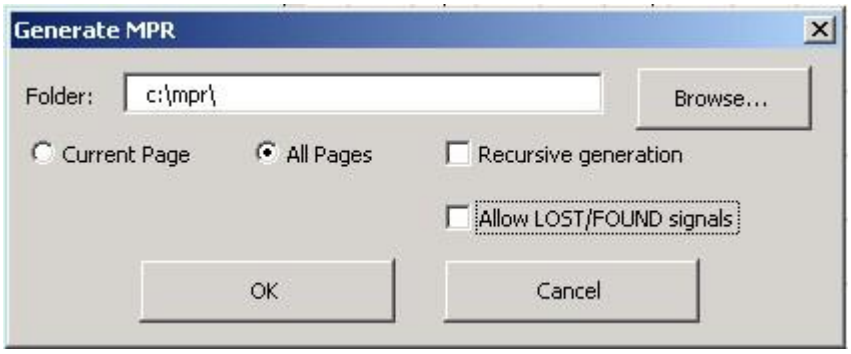
Практическая работа №30.
Использование MS Visio для генерации MPR-файлов. Генерация MPR.

Генерация MPR

Для генерации требуется выполнить следующие действия:

1. Установить Microsoft Visio.
2. Создать новый документ (File->New->New Drawing).
3. Загрузить Stensil для генерации mpr файлов(File->Open ->MSC.VSS или File>Open Stensil ->MSC.VSS). Visio выдаст предупреждение о том, что данный stensil содержит макросы. На предупреждение следует ответить "Enable macros".
4. Нарисовать MSC-диаграмму, используя технологию Drag-And- Drop.
5. Для генерации MSC вызвать следующий макрос: Tools->Macros ->MSC->Module1->Parse.

Возможны следующие опции генерации:



Current Page: При использовании данной опции генерируются только текущая страница. В указанной папке будет создан MPR-файл с именем, соответствующим имени текущей страницы в Visio.

All Pages: При использовании указанной опции генерируются все страницы активного документа.

Recursive Generation: При использовании данной опции имеется возможность генерации нескольких диаграмм на одной странице. Для этого необходимо нарисовать диаграмму, сгруппировать ее (Shape->Group). При этом сохраняется возможность рисовать несколько диаграмм на одной странице. По умолчанию - отключена.

Allow LOST/FOUND signals: При включении данной опции разрешается использование LOST/FOUND сигналов при проектировании MSC-диаграмм.

При отключении данной опции использование LOST/FOUND запрещается, при отсутствии connect на каких-либо сигналах выдается ошибка и MPR-файл **не генерируется**.

После генерации MPR-файла вызывается программа проверки синтаксиса сгенерированного MPR-файла. Для этого используется внешняя подпрограмма MSCJUST. Для ее использования необходимо настроить в Environmental Variables переменную SIC_PATH, которая соответствовала бы пути к соответствующей программе в пакете TAT.

Для проверки правильности подключения сигналов необходимо вызвать макрос Tools>Macros->MSC->Module1->Check. В случае, если какие-либо сигналы являются неподключенными, они закрашиваются красным цветом.

Используется возможность загрузки комментариев, полученных на выходе тестирующей программы. Для этого необходимо вызвать следующий макрос: Tools->Macros>MSC->Module2->OpenMPRFile. Требуется указать путь к файлу с "трассой" тестирования.

После загрузки комментариев с ошибкой они выделяются красным цветом и располагаются в месте предполагаемой ошибки.

ConfigTAT

Программа ConfigTAT предназначена для управления процессом генерации и выполнения тестов на основе MSC-диаграмм.

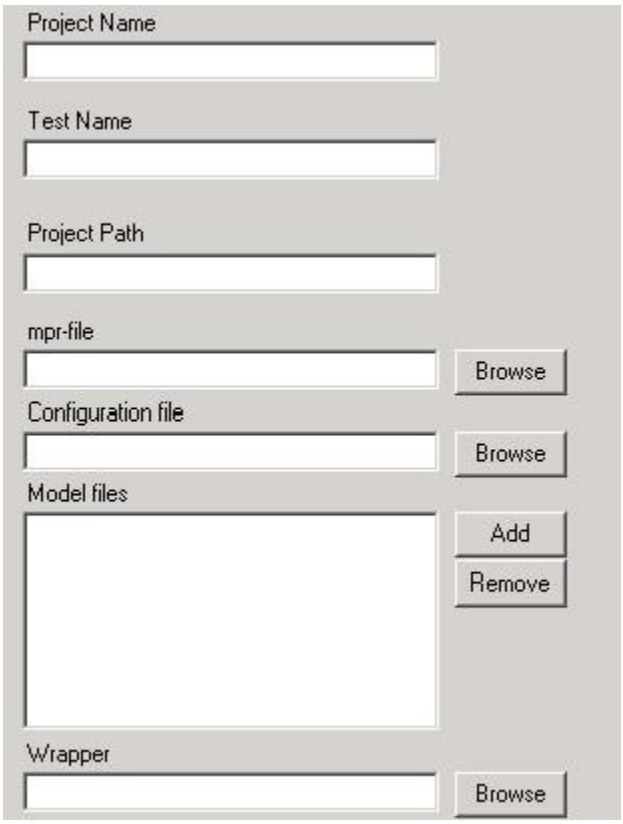
При активной вкладке "Test" осуществляется настройка и запуск одного теста.

Для генерации и запуска теста необходимо:

- указать название проекта (от него зависит одна из строк кода Wrapper);
- указать название теста (помогает различить тесты при запуске

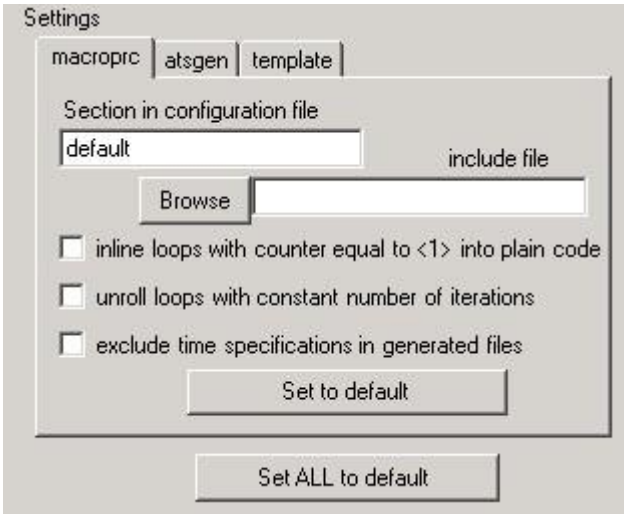
последовательности тестов);

- указать папку проекта (в нее будут помещены файлы, необходимые для генерации теста и протоколы тестирования);
- выбрать MPR-файл, задающий тест;
- выбрать файл конфигурации, описывающий методы тестируемой модели;
- выбрать файлы тестируемой модели;
- указать путь к Wrapper-у (интерфейс между моделью и тестом).



Практическая работа №31.
Настройки MS Visio для генерации MPR-файлов.

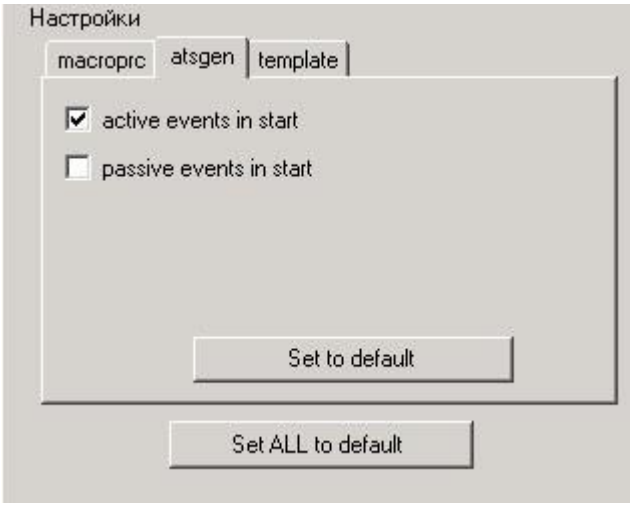
Указать следующие настройки. Для макропроцессора:



- раздел в файле конфигурации (конфигурации для нескольких тестов могут находиться в одном файле);
- inline loops with counter equal to <1> into plain code - преобразовывать циклы с одной итерацией в линейный код;
- unroll loops with constant number of iterations - разворачивать циклы с заданным числом итераций в линейный код;
- exclude time specifications in generated files - исключать временные спецификации из сгенерированных файлов.

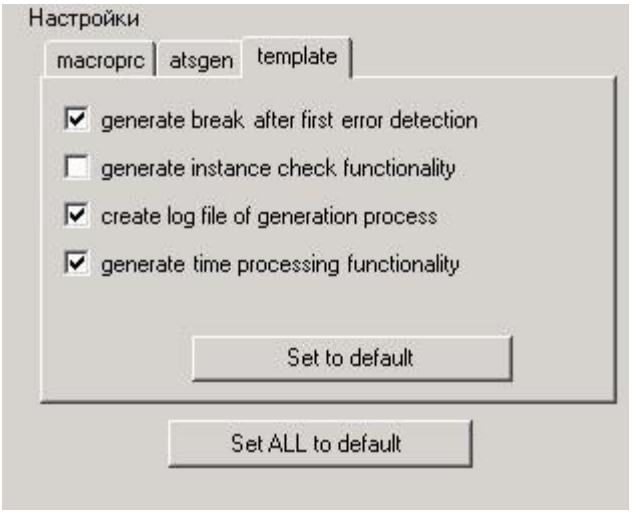
Для генератора Abstract Test Suite:

- active events in start - первый сигнал от теста к модели;
- passive events in start - первый сигнал от модели к тесту.

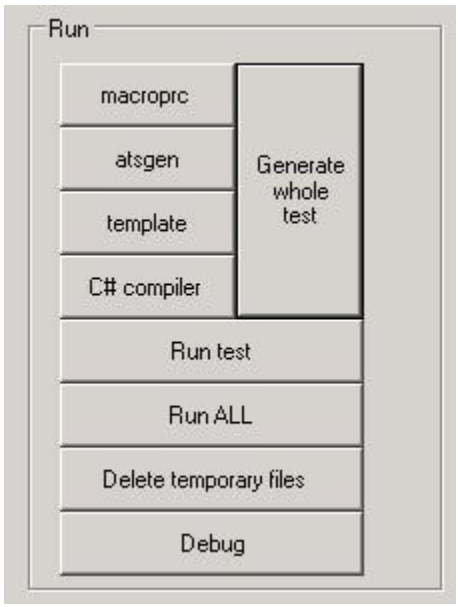


Для шаблона генерации теста на C#:

- generate break after first error detection - завершать тест после первой же ошибки;
- generate instance check functionality - проверять, соответствуют ли отправитель и получатель сигнала требуемым;
- create log file of generation process - протоколировать процесс генерации теста;
- generate time processing functionality - проверять временные требования к системе.



Группа кнопок "Run" служит для запуска по отдельности или целиком этапов генерации теста и выполнения сгенерированного теста:

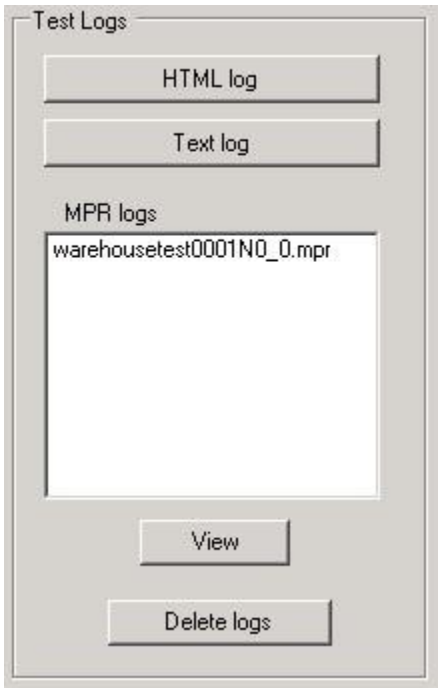


- кнопка "macroprc" запускает макропроцессор;
- кнопка "atsgen" запускает Abstract Test Suite генератор;
- кнопка "template" запускает шаблон генерации теста на языке C#;
- кнопка "C# compiler" запускает компилятор языка C# с получением на выходе файла test.exe, который представляет собой готовый к запуску тест;
- кнопка "Generate whole test" последовательно запускает макропроцессор, Abstract Test Suite генератор, шаблон генерации теста на языке C# и компилятор языка C#;
- кнопка "Run test" осуществляет запуск теста. Во время выполнения теста изменение всех настроек блокируется и отображается Progress Bar. Выполнение теста можно прервать нажатием кнопки "Cancel";

- кнопка "Run ALL" осуществляет последовательно генерацию и запуск теста;
- кнопка "Delete temporary files" удаляет промежуточные файлы, созданные в процессе генерации теста.

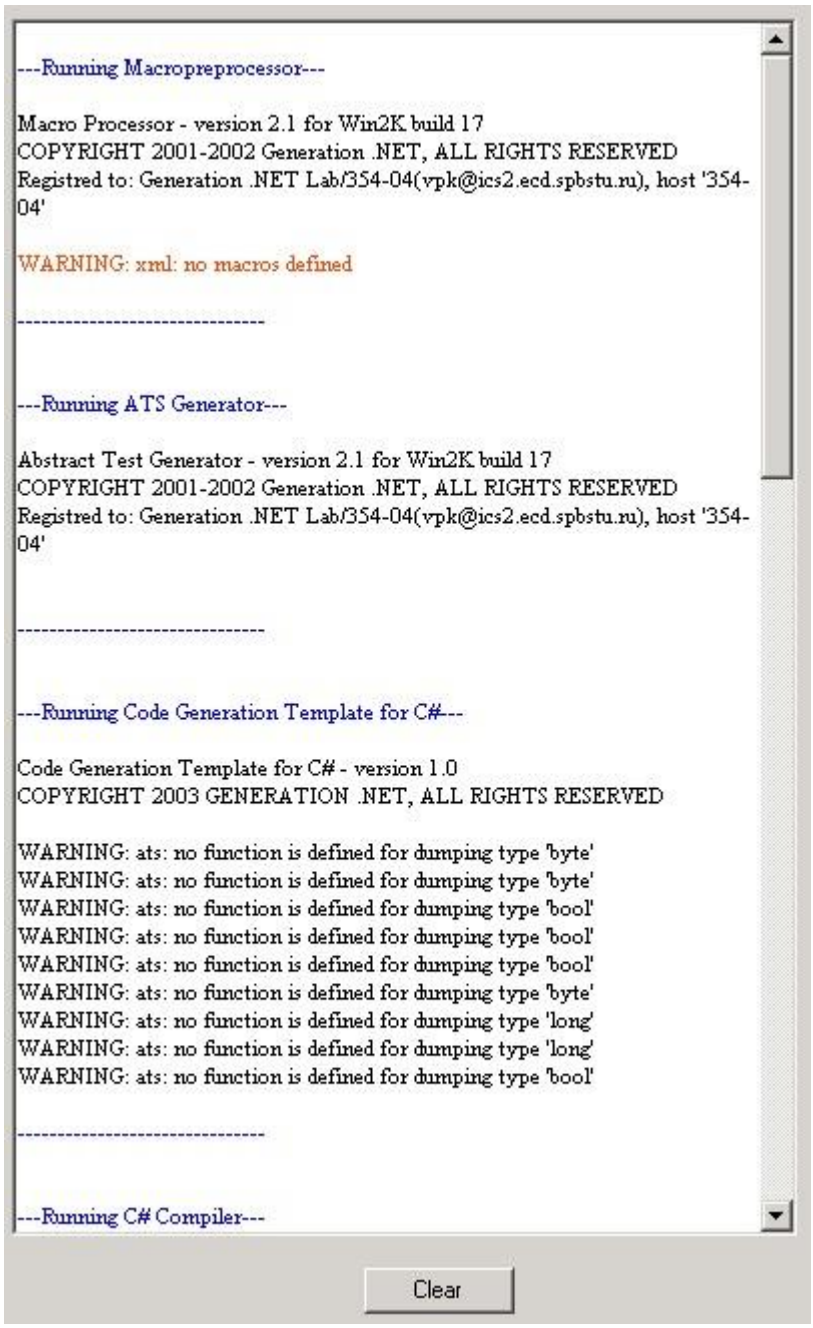
Практическая работа №32. Описание функционала.

Группа "Test Logs" позволяет просматривать протоколы тестирования:

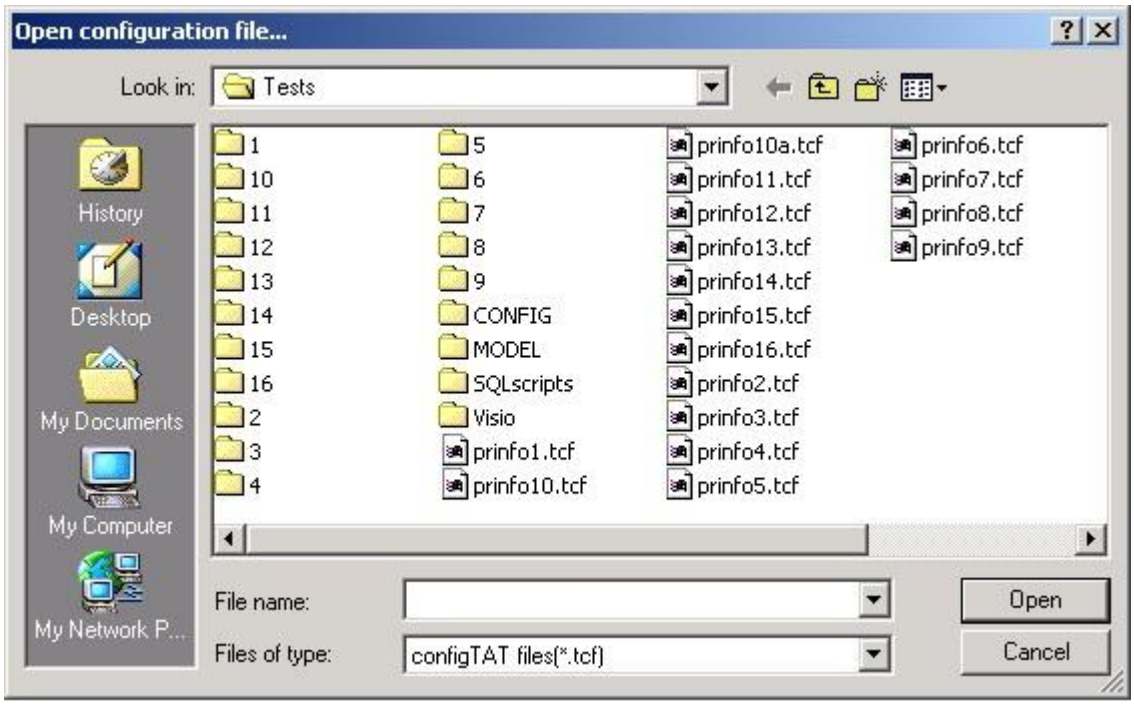


- при нажатии кнопки "HTML log" в Internet Explorer отображается протокол тестирования в виде html-страницы;
- при нажатии кнопки "Text log" в notepad отображается протокол тестирования в виде txt-файла;
- в listBox-е "MPR logs" отображается список протоколов в формате mpr (отдельный протокол для каждого testcase-а и каждой итерации теста), которые можно открыть в программе Telelogic нажатием кнопки "View";
- с помощью кнопки "Delete logs" можно удалить все протоколы тестирования.

В richTextBox-е в правой части формы отображается информация о процессе генерации и выполнении тестов. Очистить richTextBox можно с помощью кнопки "Clear":



Конфигурации тестов можно сохранять и открывать с помощью команд, соответственно, Save и Open меню File:

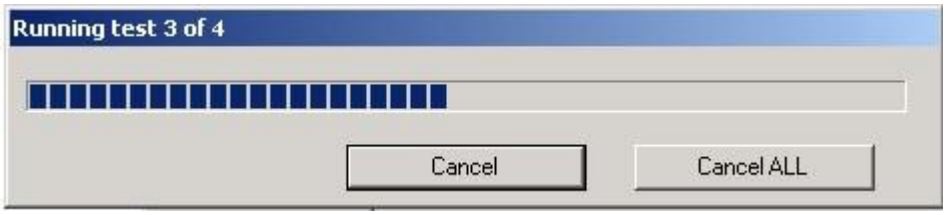


При активной вкладке "TestSuite" осуществляется настройка и запуск набора тестов.

Кнопки "Add" и "Remove" позволяют добавлять и убирать отдельные тесты (файлы конфигурации, созданные на вкладке "Test") из набора тестов.

Кнопка "Run" запускает на выполнение последовательность тестов. При этом все кнопки и меню блокируются и отображается Progress Bar.

Остановить выполнение данного теста или всей тестовой последовательности можно с помощью кнопок "Cancel" и "Cancell ALL" соответственно.



В richTextBox-е в правой части формы отображается краткая информация о результатах тестирования:

С помощью пунктов "Open" и "Save" меню "File" можно, соответственно, загрузить и сохранить список тестов тестового набора.

SysLog Animator Manual

Эта программа предназначена для визуализации журнала системы, полученного в результате тестирования системы).

Журнал системы представляет собой набор строчек следующего вида:

08.09.2003 15:01:29 : СИСТЕМА: Запрошен статус СКЛАДА

08.09.2003 15:01:29 : СКЛАД: Статус СКЛАДА = 16

08.09.2003 15:01:29 : СИСТЕМА: Запрошены параметры оси

08.09.2003 15:01:30 : СИСТЕМА: СКЛАДУ послана команда

ПОЛОЖИТЬ В РЕЗЕРВ: 4 1 0 2 -1 -1 -1

08.09.2003 15:01:30: ТЕРМИНАЛ ОСИ : 1 - нет данных

08.09.2003 15:01:34: СКЛАД : 0 - команда ПОЛОЖИТЬ В РЕЗЕРВ успешно принята

08.09.2003 15:01:38: СИСТЕМА : Запрошено сообщение от СКЛАДА

08.09.2003 15:01:38: СКЛАД : 1 - ПОДШИПНИК успешно ПРИНЯТ:
4 1 0 2 -1 -1 -1

0 8 . 0 9 . 2 0 0 3

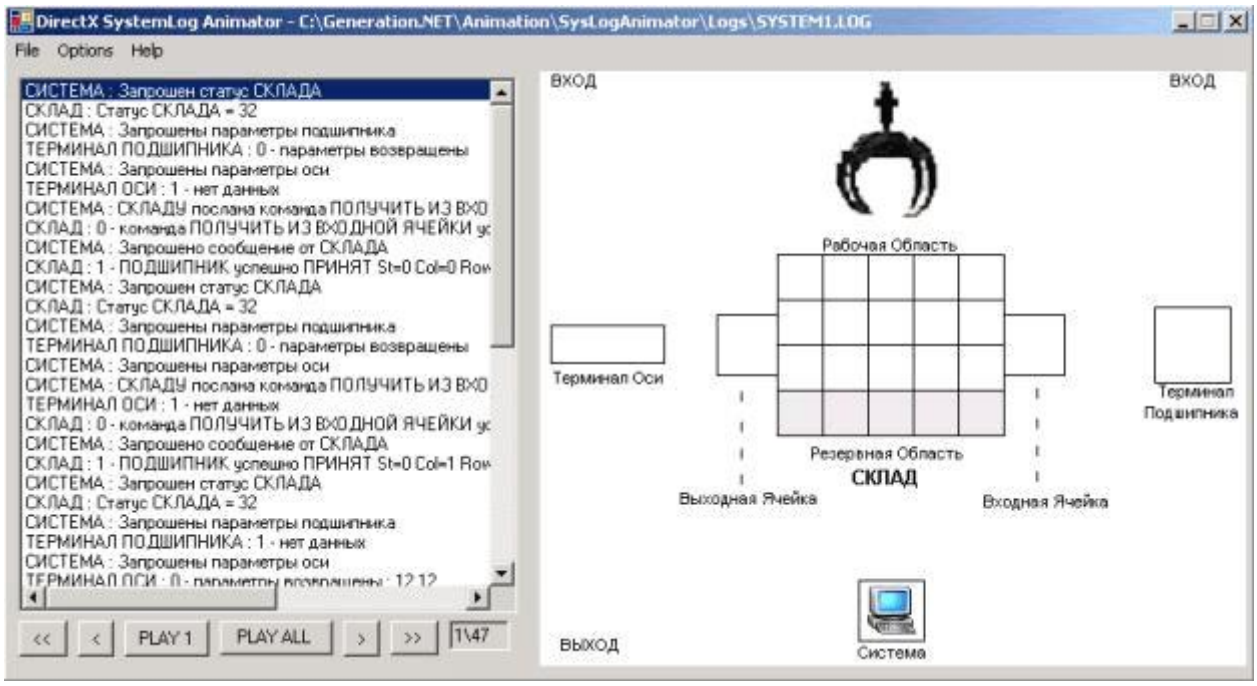
15:01:38: СИСТЕМА : Запрошен статус СКЛАДА

08.09.2003 15:01:38: СКЛАД : Статус СКЛАДА = 16

Каждая строчка такого вида формируется в кадр. Кадр - набор действий, происходящих в рамках одного события.

Практическая работа №33. Внешний вид приложения MS Visio.

Внешний Вид Приложения



Главное Меню

File - Состоит из двух подпунктов:

- Open - Открыть журнал сообщений системы.
- Exit - Выйти из приложения.

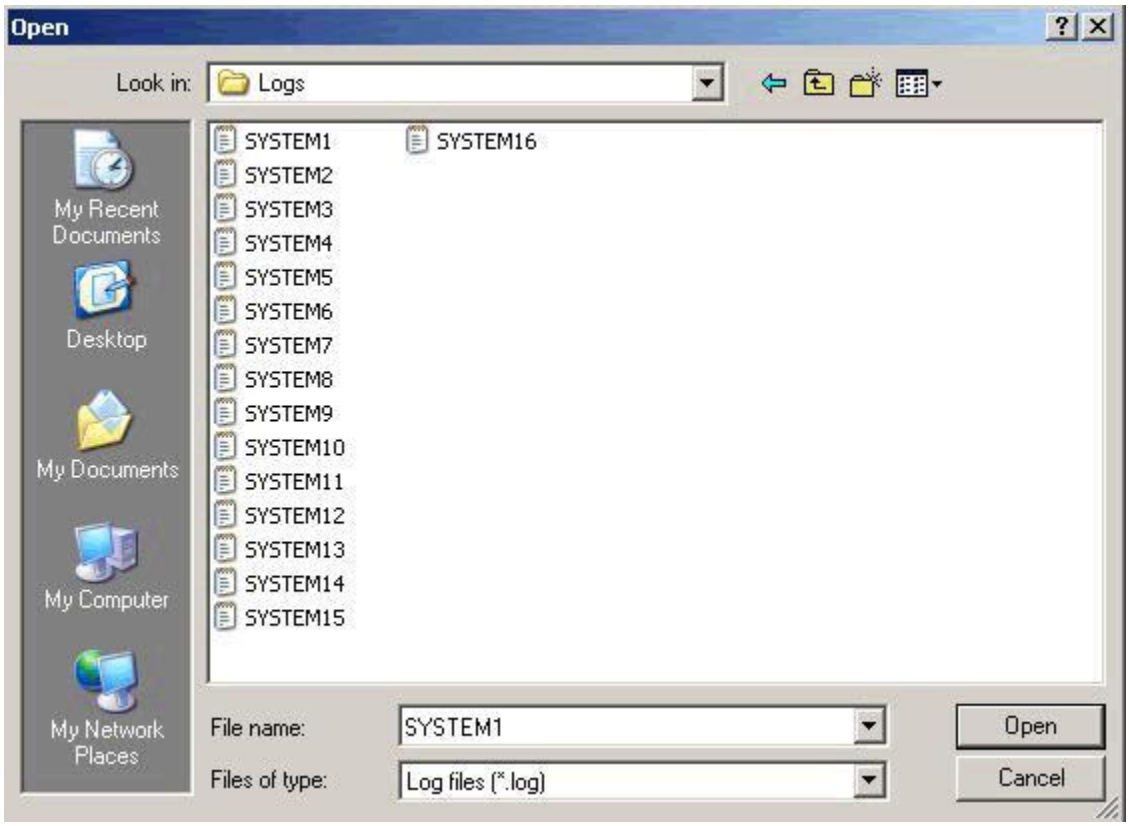
Options - Состоит из одного подпункта:

- Config - Позволяет настроить толщину линий, обозначающих сигналы и задержку при анимации сигналов

Help - Состоит из одного подпункта:

- About - Информация о текущей версии Log Animator-a

File->Open

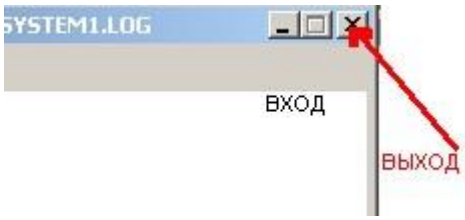


При выборе пункта File->Open откроется диалоговое окно, в котором можно выбрать файл, содержащий журнал сообщений системы, выделив необходимый файл среди имеющихся и нажав кнопку.

Если отказаться от открытия файла, необходимо нажать клавишу

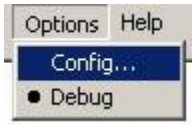
File->Exit

Для выхода из приложения можно выбрать пункт меню Exit или нажать кнопку "x" в правом верхнем углу окна.

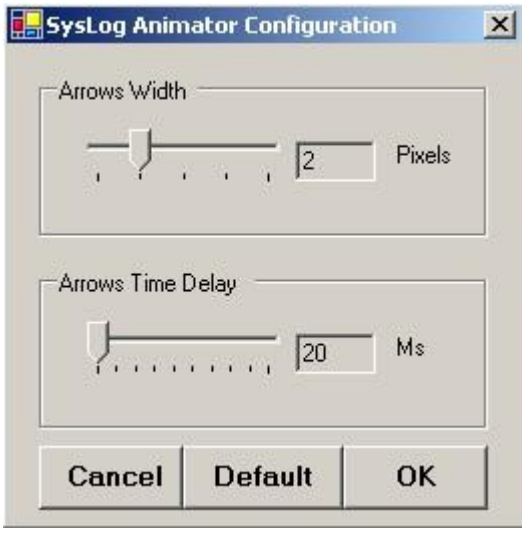


Options->Config...

Для настройки параметров визуализации нужно зайти в меню Options и выбрать подпункт Config...:



В результате появляется диалоговое окно вида:



Arrows Width - Устанавливает толщину линий, обозначающих сигнал.

Минимальное значение = 1. При этом линия выглядит следующим образом:

Максимальное значение = 5. При этом линия выглядит следующим образом:

Default значение = 2. При этом линия выглядит следующим образом:

Arrows Time Delay - Устанавливает время вывода сигналов. Чем больше значение параметра, тем дольше будет выводиться пунктирная линия, обозначающая сигнал.

Минимальное значение = 20.

Максимальное значение = 200. Default-значение
= 50.

Кнопка Cancel позволяет отказаться от изменения параметров и вернуться в визуализатор.

При нажатии кнопки Default - значения принимают величины, предопределенные заранее.

При нажатии кнопки OK система устанавливает новые значения.

Options->Debug

Выбрав этот пункт меню, можно настроить вариант работы приложения.

Если выбран режим Debug, то приложение будет выводить сообщения в отладочном режиме.

Если Debug не выбран, то сообщения будут выводиться в обычном формате: Help->About

Для просмотра информации о текущей версии проекта необходимо зайти в меню Help и выбрать подпункт About:

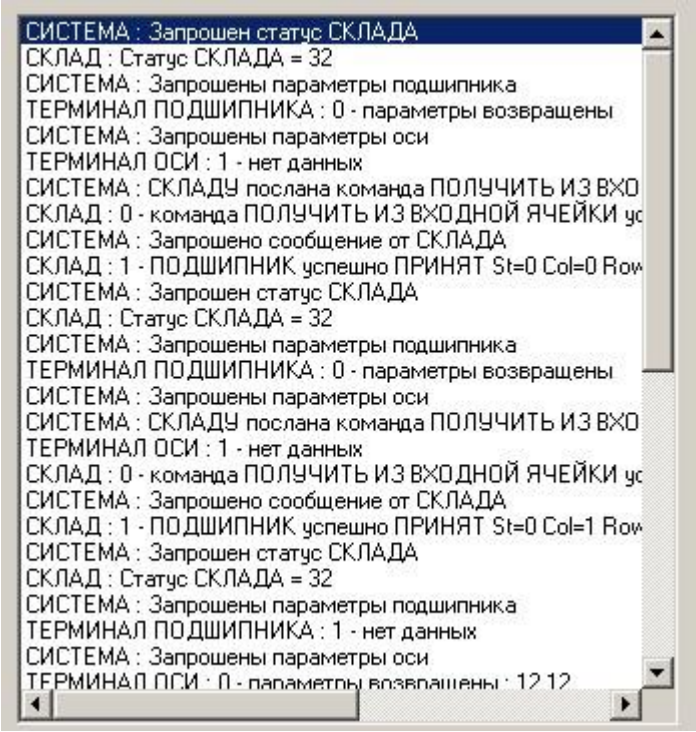
Появится диалоговое окно следующего вида:

Для выхода из диалогового окна необходимо нажать кнопку OK.

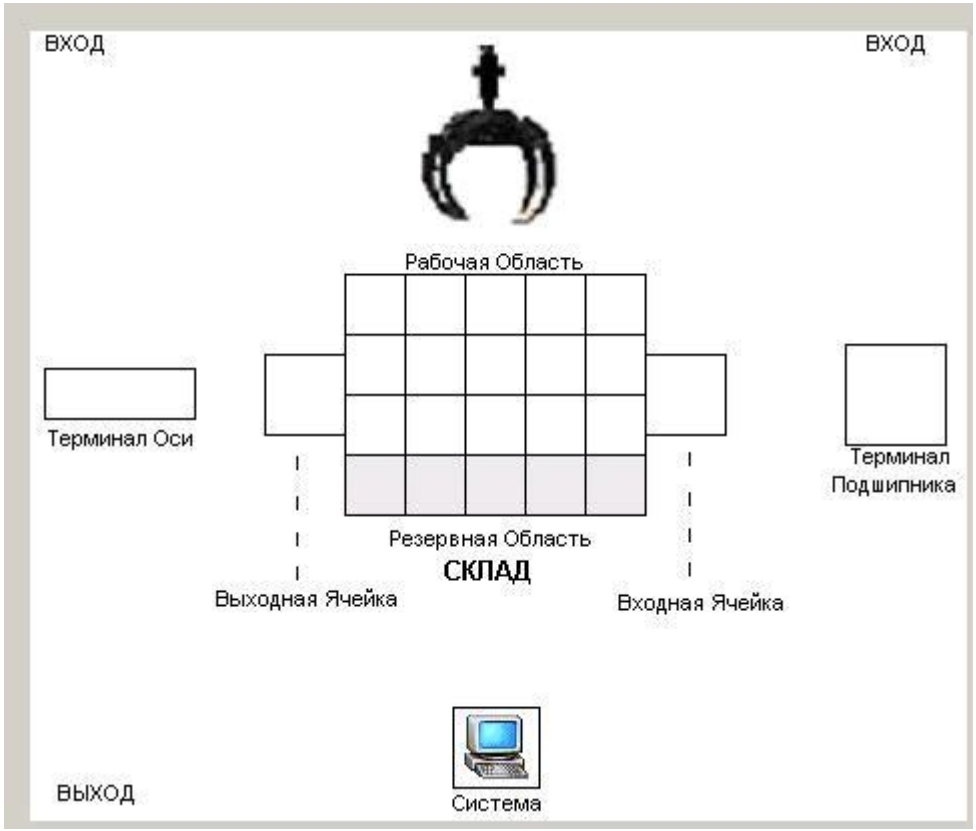
Практическая работа №34.
Анимирование кадров в приложении MS Visio.

Анимирование кадров

Если открыть файл, содержащий журнал сообщений системы, то появится разбитый на кадры журнал. Любой кадр может быть выбран путем нажатия двойного щелчка мышки на названии нужного кадра.



Изображение начального состояния выбранного кадра появится справа от списка кадров.



Для работы с кадрами существует панель управления:

Для выбора нужного кадра можно воспользоваться кнопками перемотки:

- выбрать первый кадр;

- выбрать предыдущий кадр;

- выбрать следующий кадр;

- выбрать последний кадр.

Для проигрывания кадра/кадров можно воспользоваться следующими двумя клавишами:

- проиграть 1 кадр;

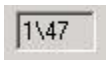
- непрерывно проигрывать все кадры, начиная с текущего и до финального. В любой момент можно прервать проигрывание,

нажав кнопку Stop, которая появится во время проигрывания вместо кнопки Play All.

Последним в списке кадров является финальное состояние (ФС).

Финальное состояние отражает состояние системы после выполнения последнего кадра.

ФС нельзя проиграть, т.к. ФС не является кадром, а лишь отражает состояние системы в результате произошедших в системе событий.



Здесь первая цифра обозначает номер текущего кадра, вторая - сколько всего имеется кадров.

Практическая работа №35. Графическая составляющая MS Visio.

Функционально-графическая составляющая

Сигналы

Сигналы в системе бывают четырех видов:

- сигнал синего цвета означает, что идет запрос от системы;
- сигнал красного цвета означает, что ответом на запрос системы является ошибка; сигнал желтого цвета означает, что идет ответ на запрос системы, не содержащий никаких данных;
- сигнал зеленого цвета означает, что запрос системы выполнен успешно. Ответ положительный.

Объекты

- ось;

- подшипник;

- манипулятор.

Ось

У оси существуют два параметра, по которым к ней подбираются подшипники: □ передний диаметр; □ задний диаметр.

- здесь параметры переднего и заднего диаметров равны 12.

- здесь значение переднего диаметра не соответствует допустимому по FS, а значение заднего диаметра не определено.

Подшипник

Подшипник может быть изображен в трех вариантах:

- означает, что у подшипника не определен номер группы.

- означает, что у подшипника 12 номер группы.

- означает, что номер группы у подшипника не соответствует допустимому по FS.

Дополнительный материал 1.

Назначение

Система предназначена для управления автоматизированным комплексом хранения подшипников. Она обеспечивает прием подшипников на склад, а также подбор и выдачу по запросу.

Данный документ описывает требования и ограничения на использование приложения.

Принятые сокращения

В настоящем документе приняты следующие определения и сокращения:

Сокращение	Определение
FS	Данный документ. Содержит технические требования, предъявляемые к программному продукту
HLD	High Level Design. Содержит описание модульной структуры проекта и взаимодействия его модулей
Проект	Проект системы управления автоматизированным комплексом хранения подшипников

В последующем тексте слово "должен" определяет необходимое требование к продукту. Слова "может", "предполагает" и "способен" определяют направление работ, которое подлежит дальнейшему уточнению.

Обзор

Введение

Комплекс хранения подшипников состоит из:

1. Склада (п. 1.2.2).
2. Терминала подшипника (п. 1.2.3).
3. Терминала оси (п. 1.2.4).

У каждого из элементов комплекса существует программа управления, реализованная в виде в dll, принимающая на вход высокоуровневые команды и преобразующая их в управляющие воздействия для данного элемента.

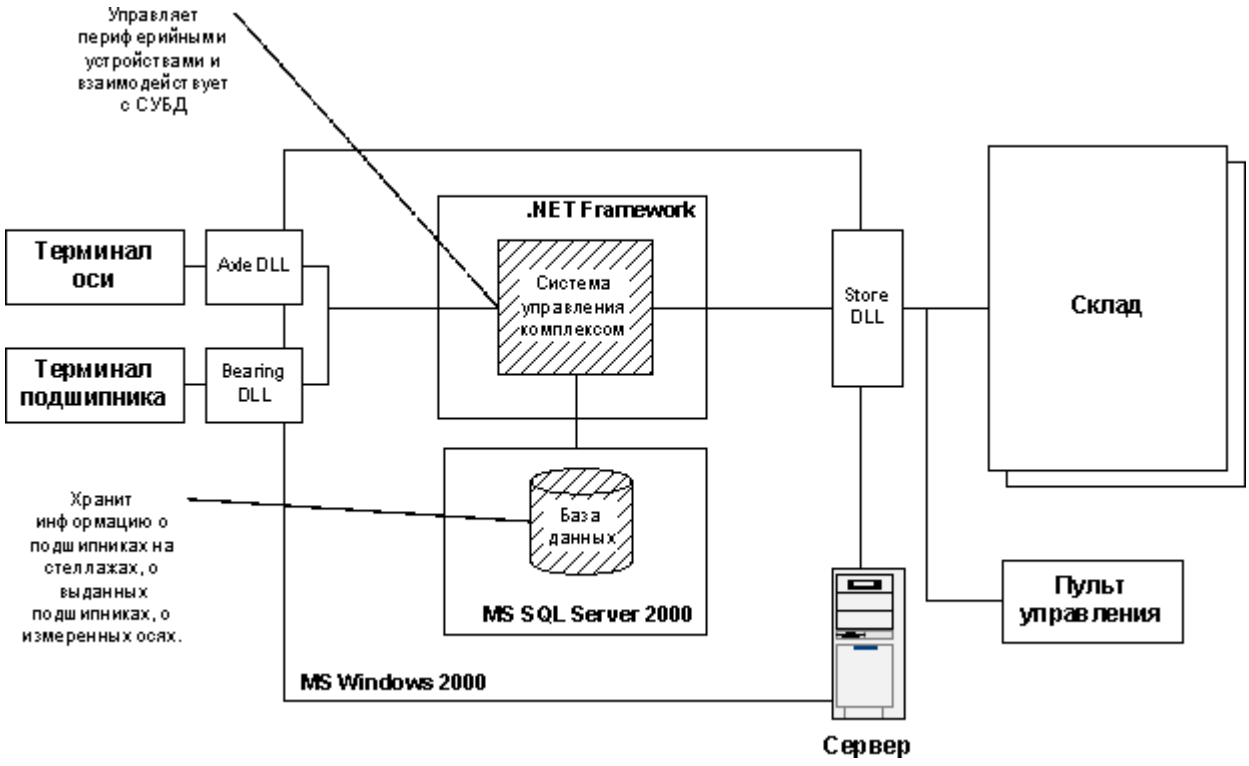


Рис. 13.1. Структура комплекса

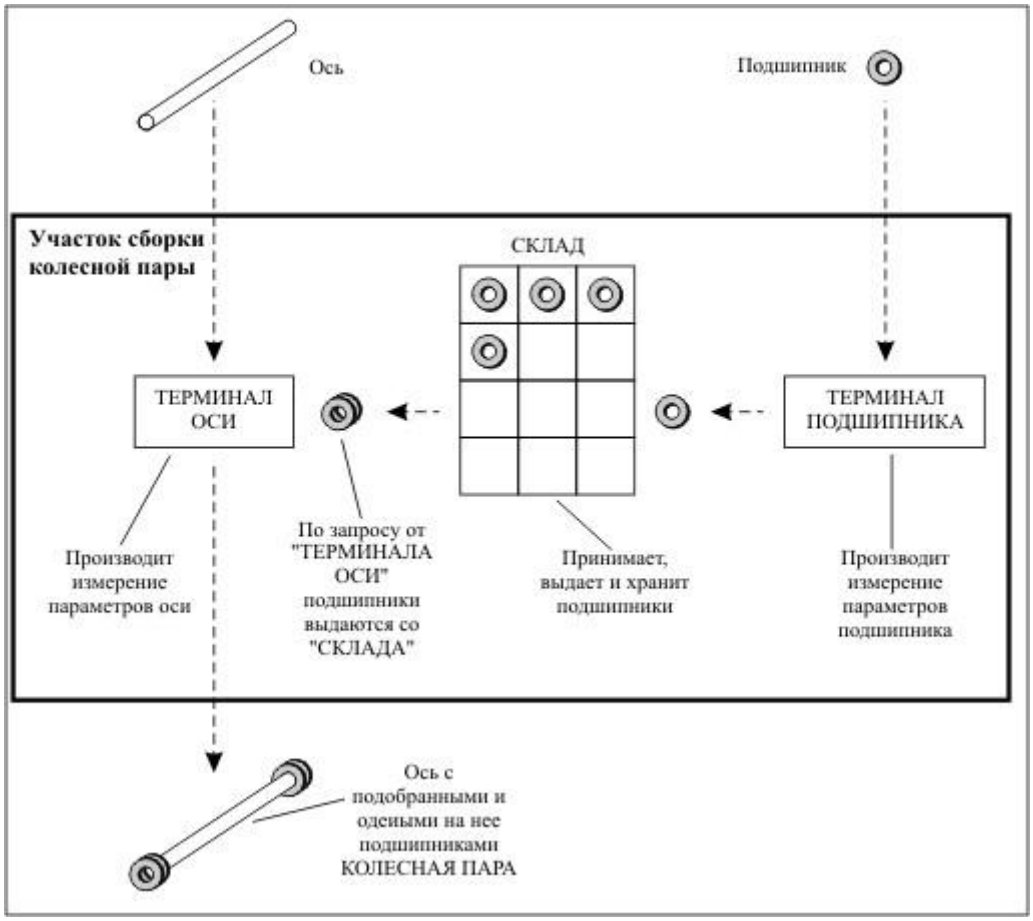


Рис. 13.2. Концептуальная схема стеллажа Склад

Склад предназначен для хранения подшипников. Он представляет собой стеллаж с ячейками, имеет **входную** и **выходную** ячейку и **робот-манипулятор**.

Все ячейки имеют координаты - сторона стеллажа (0 или 1), ряд (0 или 1 для каждой стороны), колонка (от 0 до 4 для каждого ряда), а также порядковый номер. Выходная ячейка имеет номер 999 и координаты (9,9,9), а приемная - 0 и координаты (0,0,0). Всего ячеек, кроме приемной и выходной, 20 штук.

Стеллаж имеет две области ячеек: рабочую и резервную (рис. 13.2). В рабочей области хранятся опознанные подшипники, т.е. подшипники, для которых были получены параметры с терминала подшипника и операция по перемещению в ячейку завершилась успешно. В резервную область попадают неопознанные подшипники, т.е. подшипники, при операциях с которыми произошел сбой и нельзя гарантировать достоверность полученных для них параметров.

Как в рабочей, так и в резервной областях, могут существовать сбойные ячейки. Ячейка помечается как сбойная, если робот-манипулятор не может взять из нее подшипник или если он не может положить подшипник в эту ячейку. Каждая ячейка может быть помечена либо как занятая, либо как свободная, либо как сбойная.

Таблица 13.1. Схема стеллажа

	колонка 0	колонка 1	колонка 2	колонка 3	колонка 4	
сторона 0	1	2	3	4	5	ряд 1
	6	7	8	9	10	ряд 2

	Рабочая зона					
сторона 1	11	12	13	14	15	ряд 3
	16	**17**	**18**	**19**	**20**	ряд 4
	Резервная зона					

Статус склада

Склад может иметь в каждый момент времени один из следующих статусов:

Таблица 13.2. Статус склада

Код	Значение
32	Подшипник во входной ячейке
16	Подшипник в манипуляторе
4	Нет нуля
0	Склад свободен

Рис. 13.3. Граф переходов статусов склада

Список команд складу

Склад принимает следующие команды:

Таблица 13.3. Список команд складу

Код	Название	Полное название команды
	GetR	ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ

SendR	ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ
PutR	ПОЛОЖИТЬ В РЕЗЕРВ
SetN	ПРОИЗВЕСТИ ЗАНУЛЕНИЕ
Term	ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ

0

Формат команд складу Команды имеют формат:

Таблица 13.4. Формат команд складу

Параметр	Тип	Диапазон допустимых значений	Описание
NameCommand	int	1:6,9,10,20	Название команды (табл. 13.2)
TagSt	int	0:1	Сторона результирующей ячейки
TagCol	int	0:4	Колонка результирующей ячейки
TagRow	int	1:2	Ряд результирующей ячейки
SourseSt	int	0:1	Сторона исходной ячейки
SourseCol	int	0:4	Колонка исходной ячейки
SourseRow	int	1:2	Ряд исходной ячейки

Таблица 13.5. Сообщения от склада в ответ на посылку команды

№	Код	Описание
1	-4	Нет свободных ячеек
2	-3	Не послать
3	-2	Тайм-аут
4	-1	Нет клиента
5	0	Успешное получение команды

6	1	Ошибка при получении команды
7	2	Склад не понял команду
8	3	Склад занят

Таблица 13.6. Сообщение от склада

Код	Описание
- 1	Нет склада
0	Нет сообщения
1	Команда выполнена без ошибки
2	Команда выполнена с ошибкой. Не удается взять подшипник из заданной ячейки
3	Команда выполнена с ошибкой. Не удается положить подшипник в заданную ячейку

Таблица 13.7. Статус обмена со складом

Код	Описание
- 3	Нет обмена
- 2	Тайм аут
- 1	Нет клиента
0	Возвращаемый параметр mParametr содержит статус склада
1	Нет данных

Терминал подшипника

Терминал подшипника производит измерение параметров подшипника, и на запрос системы возвращает одно из следующих сообщений.

Таблица 13.8. Статус обмена с терминалом подшипника

Код	Описание
- 3	Нет обмена

-	Тайм-аут
2	
-	Нет клиента
1	
0	Структура с измеренными параметрами подшипника в буфере
1	Нет данных

Если код сообщения = 0, то в буфере находится структура с параметрами:

Таблица 13.9. Параметры подшипника

Параметр	Тип	Диапазон допустимых значений	Описание
NameMaster	String	Любая строка длиной символов	ФИО мастера, до 255 производившего измерения
Factory	String	Любая строка длиной символов	Название депо до 255
ShiftNum	Byte	1...2	Номер рабочей смены
Number	String	Любая строка длиной до 255 символов	Номер подшипника
GroupNum	Int	12...20	Номер группы подшипника
Septype	Byte	0...1	Тип сепаратора подшипника
AShift	Float	0.01...1	Осевой сдвиг

Терминал оси

Терминал оси производит измерение ее параметров. В ответ на запрос системы он возвращает одно из следующих сообщений:

Таблица 13.10. Статус обмена с терминалом оси

Код	Описание
-3	Нет обмена
-2	Тайм аут
-1	Нет клиента
0	Структура с измеренными параметрами оси в буфере
1	Нет данных

Таблица 13.11. Параметры оси

Параметр	Тип	Диапазон допустимых значений	Описание
Name Master	String	Любая строка длиной до 255 символов	ФИО мастера, производившего измерения
ShiftNum	Byte	1...2	Номер рабочей смены
Factory	String	Любая строка длиной до 255 символов	Название депо
ANumber	Int	Любая строка длиной до 255 символов	Номер оси
Side	Byte	0...1	Сторона оси 0-правая, 1-левая
BackDiam	Float	12...20	Посадочный диаметр задний
FrontDiam	Float	12...20	Посадочный диаметр передний

Интерфейсы взаимодействия системы

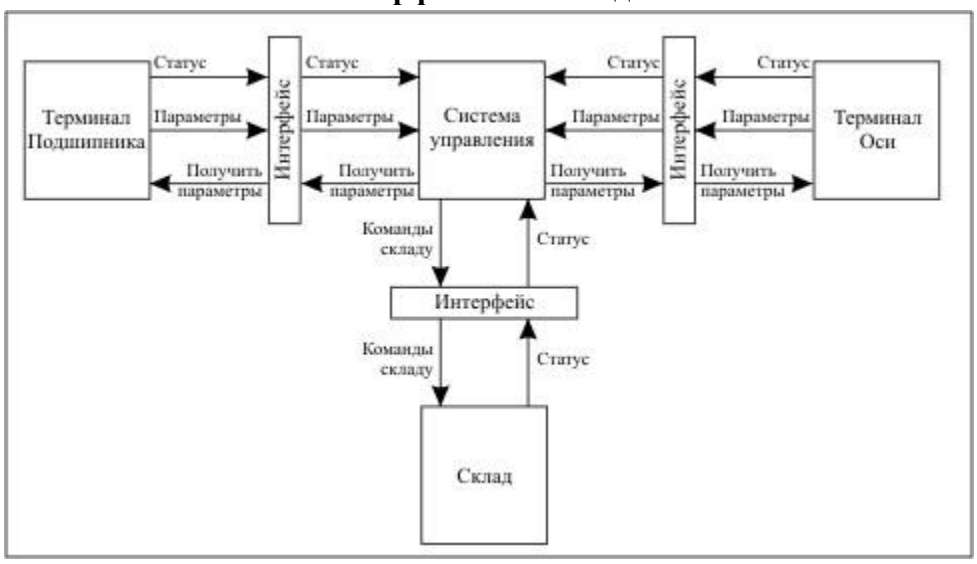


Рис. 13.4. Интерфейсы системы

Интерфейс со складом (Store.dll) Для получения сообщения от склада: static public long GetStoreMessage() - возвращаемые значения описаны в табл. 13.7. Для получения статуса склада: static public int GetStoreStat(out long mParametr) - возвращаемые значения описаны в табл. 13.6. Значения, принимаемые возвращаемым параметром mParametr, описаны в табл. 13.2.

Для посылки команды складу:

static public int SendStoreCom (int NameCommand, int TagSt, int TagCol, int TagRow, int SourseSt, int SourseCol, int SourseRow) - возвращаемые значения в передаваемые параметры описаны в табл. 13.4.

Интерфейс с терминалом подшипника (Bearing.dll) Для получения сообщения от терминала подшипника система должна вызвать следующую функцию модуля Bearing.dll:

static public int GetRollerPar(out string NameMaster, out string Factory, out string Number, out byte ShiftNum, out int GroupNum, out byte SepType, out float AShift). Функция должна возвращать одно из значений, перечисленных в табл. 13.87. Возвращаемые параметры описаны в табл. 13.9.

Интерфейс с терминалом оси (Axle.dll)

Для получения сообщения от терминала оси система должна вызывать следующую функцию модуля Axle.dll:

static public int GetRollerPar(out string NameMaster, out string Factory, out string Number, out byte ShiftNum, out int GroupNum, out byte SepType, out float AShift). Функция должна возвращать одно из значений, перечисленных в табл. 13.10. Возвращаемые параметры описаны в табл. 13.11.

Специфические требования Система управления комплексом должна:

Произвести опрос статуса склада (вызвать функцию etStoreStat).

Добавить в журнал сообщений запись "СИСТЕМА: Запрошен статус СКЛАДА". В зависимости от полученного значения произвести следующие действия:

Полученный статус склада = 32. В приемную ячейку склада поступил подшипник. Система должна:

Добавить в журнал сообщений запись "СКЛАД: Статус СКЛАДА = 32".

Получить параметры поступившего подшипника с терминала подшипника (должна быть вызвана функция GetRollerPar).

Добавить в журнал сообщений запись "СИСТЕМА: Запрошены параметры подшипника".

В зависимости от статуса терминала подшипника (возвращенного функцией GetRollerPar значения) должны быть выполнены действия, приведенные в табл.:

Полученный статус склада = 16. Склад свободен, т.е. не выполняет никаких команд, но при этом в манипуляторе находится подшипник. В этом случае система должна:

Добавить в журнал сообщений запись "СКЛАД: Статус СКЛАДА = 16".

Поставить на первое место в очереди команду PutR - "ПОЛОЖИТЬ В РЕЗЕРВ".

Полученный статус склада = 4. Нет нуля. В этом случае складу система должна:

Добавить в журнал сообщений запись "СКЛАД: Статус СКЛАДА = 4".

Поставить на первое место в очереди команду SetN - "ПРОИЗВЕСТИ ЗАНУЛЕНИЕ".

Полученный статус склада = 0. Склад свободен. Никаких действий в ответ на этот статус система предпринимать не должна.

Таблица 13.12. Действия в зависимости от статуса терминала подшипника

Статус терминала подшипника	Действие системы
-3	Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: -3 - нет обмена"

-2	Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: -2 - таймаут"
-1	Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: -1 - нет клиента"
0	Добавить на первое место команду GetR - "ПОЛУЧИТЬ ИЗ ПРИЕМНИКА В ЯЧЕЙКУ" Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: 0 - параметры возвращены <Номер_группы> "
1	Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: 1 - нет данных"
Другое	Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: ОШИБКА: Неопределенный статус"

При любом другом статусе в журнал должно быть добавлено сообщение "СКЛАД: ОШИБКА: Неопределенный статус".

Произвести опрос терминала оси (вызвать функцию получения сообщения от терминала - GetAxlePar). В журнал сообщений должно быть добавлено сообщение "СИСТЕМА: Запрошены параметры оси". В зависимости от статуса терминала оси (возвращенного функцией GetAxlePar значения) должны быть выполнены следующие действия:

При поступлении команды в очередь система должна отправить команду на выполнение складу (параллельно с продолжающимся опросом терминала оси) и в зависимости от возвращенного функцией посылки команды статуса команды, выполнить следующие действия (табл. 13.4):

Полученный статус: **0 - успешное получение команды**. В журнал сообщений дложно быть добавлено сообщение "СКЛАД: 0 - команда <Полное_название_команды> успешно принята". Команда должна быть удалена. Система должна получить сообщение от склада о результатах выполнения команды.

Полученный статус: **1 - при посылке команды произошла ошибка**. В журнал сообщений дложно быть добавлено сообщение "СКЛАД: 1 - ошибка при посылке команды <Полное_название_команды> ". Команда должна быть удалена.

Таблица 13.13. Действия, зависящие от статуса терминала оси

Статус терминала подшипника	Действие системы
-3	Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: -3 - нет обмена"

-2	Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: -2 - таймаут"
-1	Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: -1 - нет клиента"
0	Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: 0 - параметры возвращены <Передний_диаметр> < Задний_диаметр> " Подобрать два подшипника из имеющихся на складе в соответствии со следующими требованиями: Они должны находиться в ячейках с разными номерами Разность группы первого подшипника и группы переднего посадочного диаметра оси (FrontDiam) должна быть меньше либо равна 2 Разность группы второго подшипника и группы заднего посадочного диаметра оси (BackDiam) должна быть меньше либо равна 2 Разность разностей пункта ii и iii должна быть меньше либо равна 2 Выдаваемые подшипники должны иметь одинаковый тип сепаратора. В первую очередь должны выдаваться подшипники, которые находятся на складе дольше всего. При успешном подборе подшипников: В журнал сообщений должно быть добавлено сообщение "ТЕРМИНАЛ ОСИ: ПОДШИПНИКИ подобраны" В конец очереди команд должны быть добавлены две команды " SendR Отправить ячейку на выход" с параметрами подобранных подшипников, а также завершающая выдачу команда "Term Завершение команд выдачи" При отсутствии на складе подшипников, удовлетворяющих заданным параметрам, в журнал должно быть добавлено сообщение "ТЕРМИНАЛ ОСИ: Не подобрать ПОДШИПНИКОВ".
1	Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: 1 - нет данных"
Другое	Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: ОШИБКА: Неопределенный статус"

Полученный статус: **2 - склад не понял команду**. В журнал сообщений должно быть добавлено сообщение "СКЛАД: 2 - склад не понял команду <Полное_название_команды> ". Система должна попытаться послать команду повторно; после второй неудачной попытки в журнал должно быть добавлено сообщение "СКЛАД: Не понял команду после двух попыток", команда должна быть удалена, а система должна приступить к выполнению следующей в очереди команды, если очередь не пуста.

Полученный статус: **3 - склад занят**. Система должна посылать команду в течение 30 секунд, при истечении данного интервала в журнал должно быть добавлено сообщение об ошибке "СКЛАД: 3 - занят более 30 секунд", команда должна быть удалена, а система должна приступить к выполнению следующей в очереди команды, если очередь не пуста.

Полученный статус: **-1 - нет склада**. Система должна попытаться выполнить команду повторно; после второй неудачной попытки в журнал должно быть добавлено сообщение "СКЛАД:-1 - нет СКЛАДА" и система должна завершить работу.

Полученный статус: **-2 - таймаут**. Система должна попытаться выполнить команду повторно; после второй неудачной попытки в журнал должно быть добавлено сообщение "СКЛАД:-2 - таймаут при посылке команды" и система должна завершить работу.

Полученный статус: **-3 - не посылать**. Система должна попытаться выполнить команду повторно; после второй неудачной попытки в журнал должно быть добавлено сообщение "СКЛАД:-3 - не удается послать команду СКЛАДУ" и система должна завершить работу.

Полученный статус: **-4 - нет свободных ячеек**. В журнал должно быть добавлено сообщение "СКЛАД:-4 - нет свободных ячеек", команда должна быть удалена, а система должна приступить к выполнению следующей в очереди команды, если очередь не пуста.

При получении любого другого статуса в журнал должно быть добавлено сообщение "СКЛАД: ОШИБКА: Неопределенный статус при посылке команды".

После успешного получения складом команды (получения статуса команды = 0) система должна получить сообщение от склада и добавить в журнал сообщение "СИСТЕМА: Запрошено сообщение от СКЛАДА":

Полученное сообщение: **-1 - нет склада**. Система должна попытаться получить сообщение повторно, после второй неудачной попытки в журнал должно быть добавлено сообщение "СКЛАД:-1 - нет СКЛАДА" и система должна завершить работу.

Полученное сообщение: **0 - нет сообщения**. Это означает, что склад еще занят, следует продолжать опрос.

Полученное сообщение: **1 - Команда выполнена без ошибки**.

На первое место в очередь должна быть добавлена команда с параметрами удаленной команды, но номер результирующей ячейки в команде должен быть заменен номером другой свободной ячейки, соответственно алгоритму поиска свободной ячейки.

Полученное сообщение: **2 - Команда выполнена с ошибкой, не удается взять подшипник**. Если текущая выполняемая команда: GetR - "Получить из приемника в ячейку":

В журнал должно быть добавлено сообщение "СКЛАД: ОШИБКА: Не взять из входной ячейки".

Текущая команда должна быть удалена.

На первое место в очереди должна быть добавлена команда SetN - "Произвести зануление".

SendR - "Отправить ячейку на выход":

В журнал должно быть добавлено сообщение "СКЛАД: Не взять из ячейки <Номер_ячейки> ".

Текущая команда должна быть удалена.

На первое место в очереди должна быть добавлена команда SetN - "Произвести зануление".

На второе место в очереди должна быть добавлена команда SendR - "Отправить ячейку на выход" с параметрами удаленной в п. 2 команды.

При получении того же статуса при повторной попытке выполнения команды SendR:

Ячейка должна быть помечена как сбойная (не должны предприниматься дальнейшие попытки положить в нее подшипники)

В журнал должно быть добавлено сообщение "СИСТЕМА: Ячейка <Номер_Ячейки> <Номер_Стороны> <Номер_Колонки> <Номер_Ряда> помечена как сбойная" Текущая команда SendR должна быть удалена.

На первое место в очередь должна быть добавлена команда с параметрами удаленной команды, но номер результирующей ячейки в команде должен быть заменен номером другой свободной ячейки, соответственно алгоритму поиска свободной ячейки.

Полученное сообщение: 3 - команда выполнена с ошибкой, не удается положить подшипник. Если порядковый номер, результирующей ячейки - 999, т.е. это выходная ячейка, то:

В журнал должно быть добавлено сообщение "СКЛАД: Не могу положить подшипник в выходную ячейку".

Текущая команда должна быть отложена.

Должна быть выполнена команда SetN - "Произвести зануление" Должна быть предпринята попытка выполнить отложенную команду.

При получении того же статуса при повторной попытке выполнения команды:

Текущая команда должна быть удалена.

В журнал должно быть добавлено сообщение "СКЛАД: Не могу положить подшипник в выходную ячейку после второй попытки". Если порядковый номер, результирующей ячейки любой другой кроме 999 и 0, то:

В журнал должно быть добавлено сообщение "СКЛАД: Не могу положить подшипник в ячейку № <Номер_ячейки> ".

Текущая команда должна быть удалена.

На первое место в очереди должна быть добавлена команда SetN - "Произвести зануление".

На второе место в очереди должна быть добавлена команда с параметрами удаленной в п. 2 команды.

При получении того же статуса при повторной попытке выполнения команды:

Ячейка должна быть помечена как сбойная (не должны предприниматься дальнейшие попытки положить в нее подшипники).

В журнал должно быть добавлено сообщение "СИСТЕМА: Ячейка <Номер_Стороны> <Номер_Колонки> <Номер_Ряда> помечена как сбойная". Текущая команда должна быть удалена.

На первое место в очередь должна быть добавлена команда с параметрами удаленной команды, но номер результирующей ячейки в команде должен быть заменен номером другой свободной ячейки, соответственно алгоритму поиска свободной ячейки.

Дополнительный материал 2.

Назначение

Данный документ описывает внутреннюю структуру, взаимодействие с окружением и внешние интерфейсы приложения. Приводится описание классов, их взаимодействие, а также описание их внешних и внутренних интерфейсов.

Определения и принятые сокращения

В настоящем документе приняты следующие определения и сокращения:

Сокращение	Определение
FS	Документ содержит технические требования, предъявляемые к функционированию программного продукта
HLD	High Level Design. Содержит описание модульной структуры проекта и взаимодействия его модулей
Проект	Проект симулятора морского боя

Слово "должен" определяет необходимое требование к продукту. Слова "может", "предполагает", "способен" определяют направление работ, которое подлежит дальнейшему уточнению.

Описание структуры проекта

Пользовательский уровень представления системы изображен на рис. 14.1. Детальное описание пользовательского уровня приведено в FS. Представление тестируемой системы с точки зрения окружения приведено на рис. 14.2.

Взаимодействие системы со складом и терминалами должно осуществляться посредством вызова методов модулей системы.

Методы внешнего модуля Axle

//Получение сообщения от терминала оси static public
int GetAxlePar(out string NameMaster, out string
Factory, out string Number, out byte ShiftNum, out int
Side, out float FrontDiam, out float BackDiam)

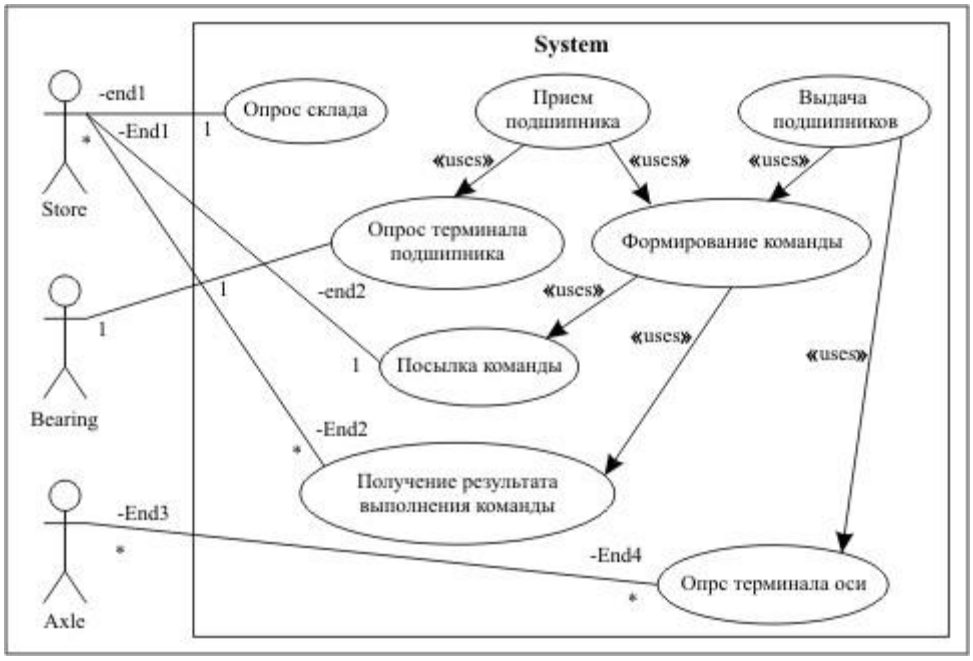


Рис. 14.1. Use case-диаграмма

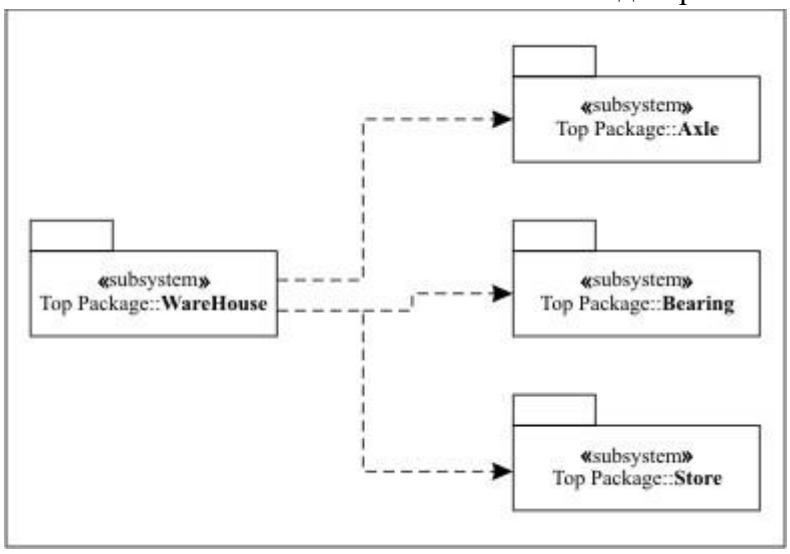


Рис. 14.2. Окружение системы

Методы внешнего модуля Bearing

```
//Получение сообщения от терминала подшипника
static public int GetRollerPar(out string NameMaster,
out string Factory, out string Number, out int ShiftNum,
out int Group, out int SepType, out float AShift)
```

Методы внешнего модуля Store

```
//Получение сообщения от СКЛАДА static
public long GetStoreMessage()
//Получение статуса СКЛАДА
static public int GetStoreStat(out long mParametr)
//Посылка команды СКЛАДУ
static public int SendStoreCom (int com, int s1, int y1, int
x1, int s2, int y2, int x2)
```

ОПИСАНИЕ КЛАССОВ

Класс TBearingParam

```
// Класс параметров подшипника public
class TBearingParam
{   public string Number;    // Номер подшипника
public int ShiftNum;     // Номер рабочей смены
public DateTime OutDateTime;
   // Дата и время выдачи подшипника   public
DateTime InDateTime;
   // Дата и время поступления подшипника   public
string Factory;   // Название депо   public string
NameMaster;// ФИО мастера   public int GroupNum;
// Номер группы подшипника   public int SepType;
// Тип сепаратора подшипника
  public float AShift;    // Осевой сдвиг   public
int Position;
  // Позиция на оси (0 - на заднем кольце (Back),
```

// 1 - на переднем кольце (Front))

// Конструктор public

TBearingParam() }

Класс реализует набор параметров подшипника.

Класс TTerminalBearing

```
// Класс терминала подшипника public
class TTerminalBearing
{
private TBearingParam BearingParam;//Структура параметров
                        //подшипника
public TCommandQueue CommandQueue; //Ссылка на очередь
                        //команд
public bool IsQuery;            //Флаг, разрешающий
                        //опрос терминала
// Конструктор public
TTerminalBearing()
// Опрашивает терминал
private long QueryTerminal()
// Запрашивает и обрабатывает статус терминала public void Process() }
```

Класс используется для взаимодействия с терминалом подшипника.

Операции:

- Конструктор TTerminalBearing() инициализирует поле BearingParam и устанавливает значение флага IsQuery в true.
- Метод QueryTerminal() вызывает функцию внешнего модуля IBearing.GetRollerPar(...), присваивает возвращенные значения полю BearingParam и возвращает значение статуса терминала.
- Метод Process() вызывает метод QueryTerminal(). Если статус терминала равен 0 (это означает, что параметры подшипника были успешно возвращены), то на первое место в очередь команд добавляется команда GetR - получить из входной ячейки: □ CommandQueue.AddCommand(TCommand.GetR,-1,0,-1, this.BearingParam, null,0);

Если получены другие значения статуса терминала, то в журнал сообщений добавляется запись в соответствии с FS пункт 1.a.iv.

Класс TAxleParam

```
// Класс параметров оси public
class TAxleParam
{ public byte ShiftNum;    //Номер рабочей
смены  public string NameMaster; //ФИО
мастера  public string Factory;   //Название депо
public string Number;    //Номер оси
 public int Side;         //Сторона оси 0 - правая,
                    //1 - левая
 public float BackDiam;   //Номер группы задний
public float FrontDiam;   //Номер группы передний }
```

Класс реализует набор параметров оси.

Класс TTerminalAxle

```
// Класс терминала оси public
class TTerminalAxle
{
private TAxleParam AxleParam; //Структура параметров
                        //подшипника        public
TStore Store;       //Ссылка на склад
 //Опрашивает            терминал
private long QueryTerminal()
 //Конструктор    public
TTerminalAxle()
 //Запрашивает и обрабатывает статус
терминала   public void Process() }
```

Класс используется для взаимодействия с терминалом оси. Операции:
Конструктор TTerminalAxle() инициализирует поле AxleParam.

- Метод QueryTerminal() вызывает функцию внешнего модуля IAxle.GetAxlePar(...), присваивает возвращенные значения полю AxleParam и возвращает значение статуса терминала.
- Метод Process() вызывает метод QueryTerminal(). Если статус терминала равен 0 (это означает что параметры оси были успешно возвращены), то вызывается метод Store.FindBearingInStore(...), подбирающий подшипники для данной оси. В зависимости от результатов подбора в журнал выводится сообщение (FS пункт 2).

Если получены другие значения статуса терминала, то в журнал сообщений добавляется запись в соответствии с FS пункт 2.

Класс TCommand

```
// Класс команды public
class TCommand
{  public const short GetR = 1;  //Получить из входной
                        //ячейки
  public const short SendR = 2; //Отправить из ячейки в
                        //выходную ячейку
  public const short MoveR = 3; //Переложить из ячейки
                        //в ячейку
  public const short PutR = 4;  //Положить в резерв
public const short PutR1 = 5; //Положить при выдаче
public const short SetN = 6;  //Произвести зануление
public const short CheckR = 9;//Проверить ячейку на
                        //занятость
  public const short PRoll = 10;//Получить параметры
                        //подшипника
  public const short Term = 20; //Завершение команд выдачи
public int NameCommand;       //Название команды   public int
NRetry;           //Число попыток выполнения команды   public int
CellSource;       //Порядковый номер исходной ячейки   public
int CellTarget;       //Порядковый номер
//результирующей ячейки   public int TagSt;           //Сторона
```

результирующей ячейки public int TagCol; //Колонка
результирующей ячейки public int TagRow; //Ряд
результирующей ячейки public int SourseSt; //Сторона
исходной ячейки public int SourseCol; //Колонка исходной
ячейки public int SourseRow; //Ряд исходной ячейки
public TBearingParam PR; //Структура характеристик
//подшипника
public TAxleParam PA; //Структура характеристик оси
public string GetFullName() }

Класс реализует команду складу. Команда складу описывает:

- код;
- название команды;
- полное название команды.

Код и название команды используются внутри системы и при взаимодействии со складом, а полное название команды используется при добавлении в журнал сообщений записей о работе системы.

Класс содержит описание всех допустимых команд (табл. 14.1).

Таблица 14.1. Список команд складу

Код	Название	Полное название
1	GetR	Получить из входной ячейки
2	SendR	Отправить из ячейки в выходную ячейку
4	PutR	Положить в резерв
6	SetN	Произвести зануление
20	Term	Завершение команд выдачи

Операции:

- Операция GetFullName() возвращает полное название команды, соответствующее коду команды (табл. 14.1), указанному в поле NameCommand, если он является допустимым кодом. В противном случае возвращается сообщение "ОШИБКА: Неверный код команды". Может применяться в любой момент.

Для выполнения конструктора не требуется никаких предварительных условий.

Для выполнения деструктора не требуется никаких предварительных условий.

Класс TComm andQueue

```
public class TCommandQueue : System.Windows.Forms.ListBox
{
 private TTerminalBearing TerminalBearing;
 //Терминал подшипника
 private TStore Store;                //Склад
 // Конструктор
public TCommandQueue(TStore store,       // Ссылка на экземпляр
//TStore
```

```
TTerminalBearing terminalBearing)        // Ссылка на экземпляр //TTerminalBearing
// Добавляет команду в очередь команд на указанную позицию   public
void AddCommand(int NameCommand, // Код команды
   int CntRoll,                  // Номер выдаваемого подшипника     int
CellSourse,                  // Порядковый номер исходной ячейки
 int CellTarget,                 // Порядковый номер результирующей
                              // ячейки
   TBearingParam PR,              // Параметры подшипника
   TAxleParam PA,                   // Параметры оси
int Position)                  // Позиция в очереди
 // Удаляет команду из очереди public
void DeleteCommand(int Position)
 // Выполняет первую команду в очереди public
void ProcessCommand()
 // Отправляет команду
складу private int
SendCommand() }
```

Класс реализует очередь FIFO объектов типа TCommand. Наследуется от System.Windows.Forms.ListBox библиотеки .NET. Количество команд в очереди не ограничено. Имеет ссылки на экземпляры классов TTerminalBearing и TStore. Операции:

- Конструктор TCommandQueue(...) создает экземпляр класса и инициализирует поля TerminalBearing и Store с помощью передаваемых указателей. Передаваемые указатели должны указывать на существующие объекты, внутри конструктора такой проверки не происходит.
- Операция AddCommand(...) создает объект типа TCommand, присваивает ему переданные параметры и добавляет в очередь команд на указанную позицию. При добавлении команды GetR (см. табл. 14.1 "Список команд складу") должен быть запрещен опрос терминала подшипника, т.е. полю TerminalBearing.IsQuery должно быть присвоено значение false. Позиции в очереди нумеруются, начиная с 0. Значение позиции в очереди = -1 означает, что команда будет добавлена в конец очереди. Можно также явно задавать позицию, на которую следует добавить команду.
- Операция DeleteCommand(...) удаляет команду из очереди на указанной позиции. При удалении команды GetR (см. табл. 14.1 "Список команд складу") должен быть разрешен опрос терминала подшипника, т.е. полю TerminalBearing.IsQuery должно быть присвоено значение true.
- Операция ProcessCommand() при наличии команд в очереди посылает первую команду из очереди складу при помощи метода SendCommand(). В зависимости от возвращенного этим методом значения - статуса команды - должны быть предприняты следующие действия, описанные в FS, глава 3 "Специфические требования", пункт 3.
- Операция SendCommand() посылает первую в очереди команду (с индексом 0) складу с помощью функции IStore.SendStoreCom() из Store.dll и возвращает код ответа склада. Перед отправкой команды устанавливаются следующие ее поля:
  - Для команды GetR "Получить из входной ячейки" координаты ячейки источника

- (SourseSt = 0, SourseCol = 0, SourseRow =0), а координаты результирующей ячейки возвращает метод TStore.FindFreeCell() с параметром, определяющим зону склада, для поиска установленным в true (поиск в рабочей зоне склада).

- Для команды SendR "Отправить из ячейки в выходную ячейку" координаты

результирующей ячейки - (TagSt = 9, TagCol = 9, TagRow = 9), а координаты ячейки источника возвращает метод TStore.GetCoord() на основе значения поля CellSourse посылаемой команды.

- Для команды PutR "Положить в резерв" координаты ячейки источника - (SourseSt = -1, SourseCol = -1, SourseRow = -1), а координаты результирующей ячейки возвращает метод TStore.FindFreeCell() с параметром, определяющим зону склада для поиска установленным в false (поиск в резервной зоне склада). Для выполнения деструктора не требуется никаких предварительных условий.

```
public class TStore
{
public TTerminalBearing TerminalBearing;    //Ссылка на
                                    //терминал подшипника
public TCommandQueue CommandQueue;          //Ссылка на очередь
                                    //команд
private static string ConnectionString;     //Строка
                                    //подключения к серверу
БД   private SqlConnection connFindFreeCell;   private
SqlConnection connFindBearing;   private
SqlConnection connAddBearing;   private SqlConnection
connMarkFree;   private SqlConnection
connRemoveBearing;   private SqlConnection
connGetCoord;
// Возвращает статус склада private
long GetStatus()
// Добавляет запись об обслуженной ОСИ в базу данных private
bool AddBearingAxle(TCommand Command)
// Добавляет запись о подшипнике в базу данных private
bool AddBearing(TCommand Command)
// Удаляет запись о подшипнике из базы данных private
bool RemoveBearing(TCommand Command)
// Помечает ячейку как проблемную private
bool MarkCellBad(int Cell)
// Возвращает сообщение склада о выполнении команды private
long GetMessage()
// Конструктор public
TStore()
// Запрашивает и обрабатывает статус склада public
void Process()
// Запрашивает и обрабатывает сообщение склада public
void ProcessMessage()
```

// Находит свободную ячейку
public bool FindFreeCell(ref int CNum, ref int TagSt, ref
int TagRow, ref int TagCol, bool IsReserve)
// Возвращает координаты ячейки по номеру
public bool GetCoord(int CNum, ref int Side,
ref int Row, ref int Col)
//Находит подшипники в складе на основании параметров
ОСИ public bool FindBearingInStore(TAxleParam AxleParam) }

Класс TStore Класс

реализует терминал склада.

Операции:

- Метод GetStatus() вызывает функцию внешнего модуля IStore.GetStoreStat(...) и в случае, если она вернула 0, GetStatus() возвращает значение статуса склада, в противном случае возвращается -1.
- Метод AddBearingAxle() добавляет запись об обслуженной оси в базу данных на основе переданной в качестве параметра команды.
- Метод AddBearing() добавляет запись о принятом подшипнике в базу данных на основе переданной в качестве параметра команды.
- Метод RemoveBearing() удаляет запись о выданном подшипнике из базы данных на основе переданной в качестве параметра команды.
- Метод MarkCellBad(...) помечает ячейку как проблемную, порядковый номер ячейки передается как параметр.
- Метод GetMessage() вызывает функцию внешнего модуля IStore.GetStoreMessage(...) и возвращает код сообщения склада
- Конструктор TStore() читает из конфигурационного файла имя сервера SQL , имя пользователя, пароль и инициализирует все поля типа SqlConnection.
- Метод Process() вызывает метод GetStatus(). В зависимости от полученного статуса склада он производит следующие действия:
  - Статус = 32. Добавляется сообщение в журнал и вызывается метод обработки ситуации прихода подшипника - TerminalBearing.Process().
  - Статус = 16. Добавляется сообщение в журнал и вызывается метод постановки команды "Положить в резерв" в очередь CommandQueue.AddCommand().
  - Статус = 8. Добавляется сообщение в журнал.
  - Статус = 4. Добавляется сообщение в журнал и вызывается метод постановки команды "Произвести зануление" в очередь CommandQueue.AddCommand().
  - Статус = 0. Добавляется сообщение в журнал.
  - При любом другом статусе в журнал добавляется сообщение об ошибке.
- Метод ProcessMessage() вызывает метод GetMessage() и в зависимости от полученного результата выполняет следующие действия:
  - Полученный результат - -1. Добавляется сообщение в журнал. Предпринимается повторная попытка получить сообщение, после второй неудачной попытки происходит выход из приложения.
  - Полученный результат - 0. Добавляется сообщение в журнал. Продолжается опрос (вызов метода GetMessage()).

- Полученный результат -1. Добавляется сообщение в журнал. Производит обновление базы данных по выданному или принятому подшипнику и удаляет команду из очереди.
- Полученный результат -2. Добавляется сообщение в журнал. Пытается выполнить команду повторно, после второй неудачной попытки удаляет команду из очереди.
- Полученный результат -3. Добавляется сообщение в журнал. Пытается выполнить команду повторно, после второй неудачной попытки удаляет команду из очереди.

Операция FindFreeCell(...) ищет свободную ячейку в резервной (параметр IsReserve установлен в true) или рабочей (параметр IsReserve установлен в false) области с наименьшим порядковым номером и возвращает ее координаты: порядковый номер - CNum, сторона - TagSt, ряд - TagRow, колонка - TagCol.

Операция GetCoord(...) возвращает для переданного порядкового номера ячейки (CNum) номер строки (Side), колонки (Col) и ряда (Row).

```
public class TLog
{
static private FileStream fs =  new
FileStream("system.log",
  FileMode.Create,FileAccess.Write,
FileShare.ReadWrite); static private StreamWriter srLog =
new StreamWriter(fs);
 // Деструктор
 ~ TLog()
 // Добавляет запись в журнал сообщений
системы  static public void AddToLog(string
LogMessage) }
```

Класс TLog Класс реализует журнал сообщений системы.

Операции:

- Метод AddToLog() сохраняет переданное ему в качестве параметра сообщение в файл.

```
public class TModel
{  private TLog Log;                  //Ссылка на журнал сообщений
 private TStore Store;                //Ссылка на склад  private
TTerminalAxle TerminalAxle;      //Ссылка на
                                     //терминал оси
 private TTerminalBearing TerminalBearing; //Ссылка на
                                     //терминал подшипника
 private TCommandQueue CommandQueue;      //Ссылка на очередь
                                     //команд
 public Thread QueryThread;           //Поток опроса
public Thread CommandThread;          //Поток команд
public bool QueryThreadTerminated;      //Флаг окончания
                                     //работы потока опроса
 public bool CommandThreadTerminated;    //Флаг окончания
                                     //работы потока команд
```

// Конструктор
public TModel()
// Метод, реализующий поток опроса public
void QueryThreadExecute()
// Метод, реализующий поток выполнения
команд public void CommandThreadExecute() }

Класс TModel Класс

реализует журнал сообщений системы.

Операции:

- Метод QueryThreadExecute() реализует поток опроса. Здесь последовательно вызываются методы TStore.Process() и TTerminalAxle.Process().
- Метод CommandThreadExecute() реализует поток выполнения команд. Здесь вызывается метод TcommandQueue.ProcessCommand().

ОПИСАНИЕ ВЗАИМОСВЯЗЕЙ КЛАССОВ

Взаимосвязи классов

Общая диаграмма классов, используемых в системе

Система должна быть реализована в виде набора классов, представленного на рис. 14.3.

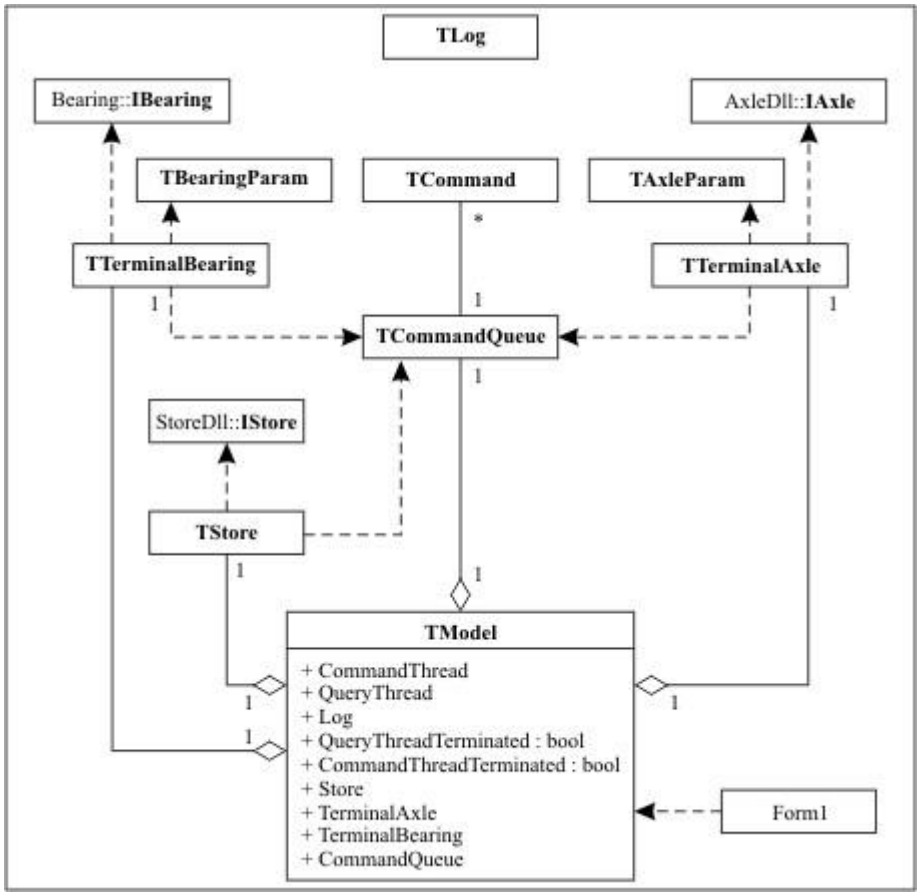


Рис. 14.3. Диаграмма классов

Описание интерфейсного взаимодействия

Общедоступные интерфейсы классов (подробное описание приведенных методов изложено в п. 2):

TStore

// Запрашивает и обрабатывает статус склада public
string Process()

// Запрашивает и обрабатывает сообщение склада public
string ProcessMessage()
// Находит свободную ячейку
public bool FindFreeCell(ref int CNum, ref int TagSt, ref
int TagRow, ref int TagCol, bool IsReserve)
// Возвращает координаты ячейки по номеру public
bool GetCoord(int CNum, ref int Side, ref int Row, ref
int Col)
// Находит подшипники в складе на основании параметров ОСИ public
bool FindBearingInStore(TAxleParam AxleParam) TTerminalBearing
// Конструктор public
TTerminalBearing()
// Запрашивает и обрабатывает статус
терминала public void Process() TTerminalAxle
// Конструктор public
TTerminalAxle() //
Запрашивает и
обрабатывает статус
терминала
public void Process()
TCommandQueue
// Конструктор
public TCommandQueue(TStore store, TTerminalBearing terminalBearing)
// Добавляет команду в очередь команд на указанную
позицию public void AddCommand(int NameCommand, int
CntRoll, int CellSourse, int CellTarget, TBearingParam PR,
TAxleParam PA, int Position)
// Удаляет команду из очереди public
void DeleteCommand(int Position)
// Выполняет первую команду в очереди
public void ProcessCommand()
TBearingParam
// Конструктор public
TBearingParam()
TAxleParam
// Конструктор public
TAxleParam()
TModel
// Конструктор public
TModel()
// Метод, реализующий поток опроса public
void QueryThreadExecute()
// Метод, реализующий поток выполнения команд
public void CommandThreadExecute() TCommand
// Возвращает полное название команды

public string GetFullName() TLog

// Добавляет запись в журнал сообщений системы static

public void AddToLog(string LogMessage) TMainForm

// Конструктор public

MainForm **Основная**

литература:

1. Мейер, Б. Объектно-ориентированное программирование и программная инженерия / Б. Мейер. - 2-е изд., испр. - Москва : Национальный Открытый Университет «ИНТУИТ», 2016. - 286 с. : ил. ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=429034

2. Лягинова, О.Ю. Разработка схем и диаграмм в Microsoft Visio 2010 / О.Ю. Лягинова. - 2-е изд., исправ. - Москва : Национальный Открытый Университет «ИНТУИТ», 2016. - 128 с. : схем., ил. ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=428810

3. Федорова Г.И. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности. Учебное пособие. Изд.: КУРС, Инфра-М. Среднее профессиональное образование. 2016 г. 336 стр.

4. Мякишев, Д.В. Принципы и методы создания надежного программного обеспечения АСУТП : методическое пособие / Д.В. Мякишев. - Москва ; Вологда : Инфра-Инженерия, 2017. - 115 с. : ил., схем., табл. - Библиогр. в кн. - ISBN 978-5-9729-0179-1 ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=466489

Дополнительная литература:

1. Долженко, А.И. Технологии командной разработки программного обеспечения информационных систем / А.И. Долженко. - 2-е изд., исправ. - М. : Национальный Открытый Университет «ИНТУИТ», 2016. - 301 с. : схем., ил. - Библиогр. в кн. ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=428801

2. Лоскутов, В.И. Разработка информационных систем для Windows Store / В.И. Лоскутов, И.Л. Коробова. - 2-е изд., исправ. - М. : Национальный Открытый Университет «ИНТУИТ», 2016. - 180 с. : ил. - Библиогр. в кн. ; То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=428809