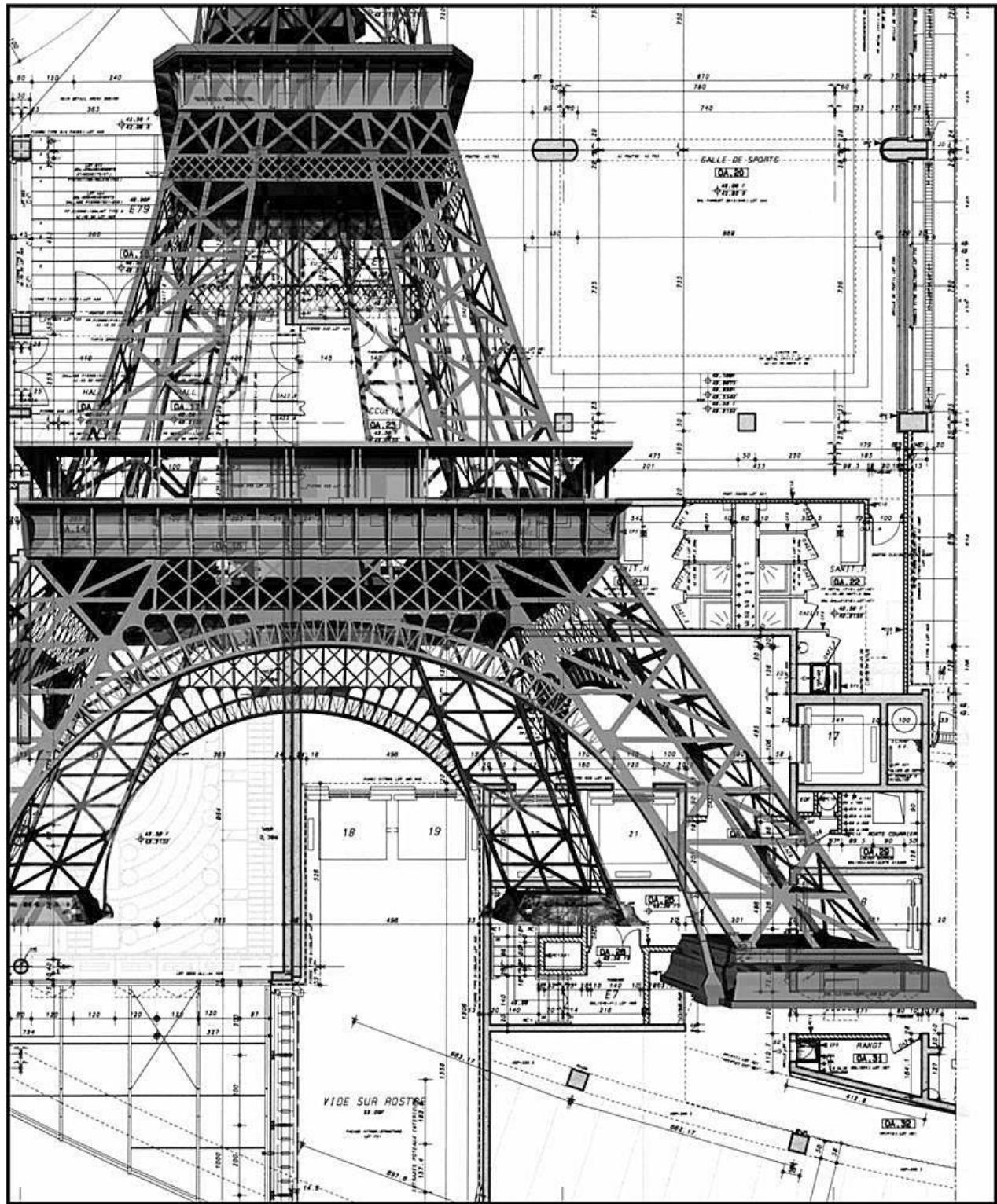


МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ
ПО КУРСУ

«Технології розробки програмного забезпечення - 2»



Зміст

Зміст

| | |
|---|----|
| ОБЩИЕ ПОЛОЖЕНИЯ..... | 5 |
| Лабораторная работа №1 | 6 |
| История развития систем контроля версий | 6 |
| Темная эпоха | 6 |
| Ренессанс..... | 8 |
| Работа с Mercurial | 8 |
| Лабораторная работа №2..... | 11 |
| Основы отладки программных продуктов..... | 11 |
| Устранение неисправностей..... | 12 |
| Средства отладки веб-сайтов | 15 |
| Средства отладки Visual Studio..... | 16 |
| ЛАБОРАТОРНАЯ РАБОТА №3 | 18 |
| Управление проектами | 18 |
| Лабораторная Работа № 4 | 24 |
| Диаграмма вариантов использования (Use-Cases Diagram) | 25 |
| Актеры (actor)..... | 26 |
| Варианты использования (use case)..... | 26 |
| Отношения на диаграмме вариантов использования | 26 |
| Сценарии использования..... | 30 |
| Лабораторная Работа №5 | 32 |
| Инструменты прототипирования | 32 |
| Прототипирование при помощи SketchFlow..... | 33 |
| ЛАБОРАТОРНАЯ РАБОТА №6 | 39 |
| Представление классов..... | 39 |
| Отношения | 40 |
| Применение диаграмм классов | 42 |
| Логическая структура базы данных | 42 |
| Нормальные формы | 42 |
| Проектирование БД..... | 43 |
| Диаграмма развертывания (Deployment Diagram) | 45 |
| Windows Presentation Foundation (WPF) | 47 |
| XAML | 48 |
| Способы организации элементов интерфейса в окне WPF | 49 |
| Основные элементы управления | 55 |

| | |
|---|-----|
| Лабораторная Работа № 8 | 55 |
| Диаграмма компонентов | 56 |
| Рекомендации по построению диаграммы компонентов..... | 58 |
| События WPF..... | 59 |
| Стили WPF | 60 |
| Шаблоны WPF | 61 |
| Триггеры WPF..... | 62 |
| Лабораторная Работа № 9 | 64 |
| Диаграмма деятельности | 64 |
| Механизм привязок WPF | 67 |
| Шаблоны данных в WPF..... | 70 |
| Механизм команд WPF..... | 72 |
| Шаблон MVVM | 73 |
| Лабораторная работа №10..... | 75 |
| Диграмма состояний UML..... | 75 |
| Анимации WPF..... | 80 |
| Элементы управления WPF | 82 |
| Лабораторная работа № 11 | 85 |
| Диаграмма последовательностей..... | 85 |
| Windows Communication Foundation..... | 90 |
| Лабораторная работа № 12 | 94 |
| Стандарты стилистического оформления кода | 94 |
| Stylecop | 95 |
| Лабораторная работа № 14 | 100 |
| FxCop | 100 |
| ЛАБОРАТОРНАЯ РАБОТА № 15 | 103 |
| Лабораторная работа № 16 Документирование исходного кода..... | 111 |
| Doxygen..... | 112 |

ОБЩИЕ ПОЛОЖЕНИЯ

Требования к оформлению отчетов

Отчет должен содержать: титульный лист с фамилией и инициалами студента, номером и названием лабораторной работы; задание на лабораторную работу; ход работы, где освещены шаги выполнения работы; выводы.

ЛАБОРАТОРНАЯ РАБОТА №1

Системы контроля версий. Децентрализованная система контроля версий Mercurial

Задание

1. Ознакомиться с краткими теоретическими сведениями.
2. Создать Mercurial репозиторий.
3. Клонировать Mercurial репозиторий одного из open-source проектов (например, на bitbucket.org или codeplex.com)
4. Продемонстрировать базовую работу с репозиторием: создание версий, добавление тегов, работа с ветками (создания и слияние), работа с удаленным репозиторием.

Краткие теоретические сведения

Что такое системы контроля версий и зачем они нужны ?

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

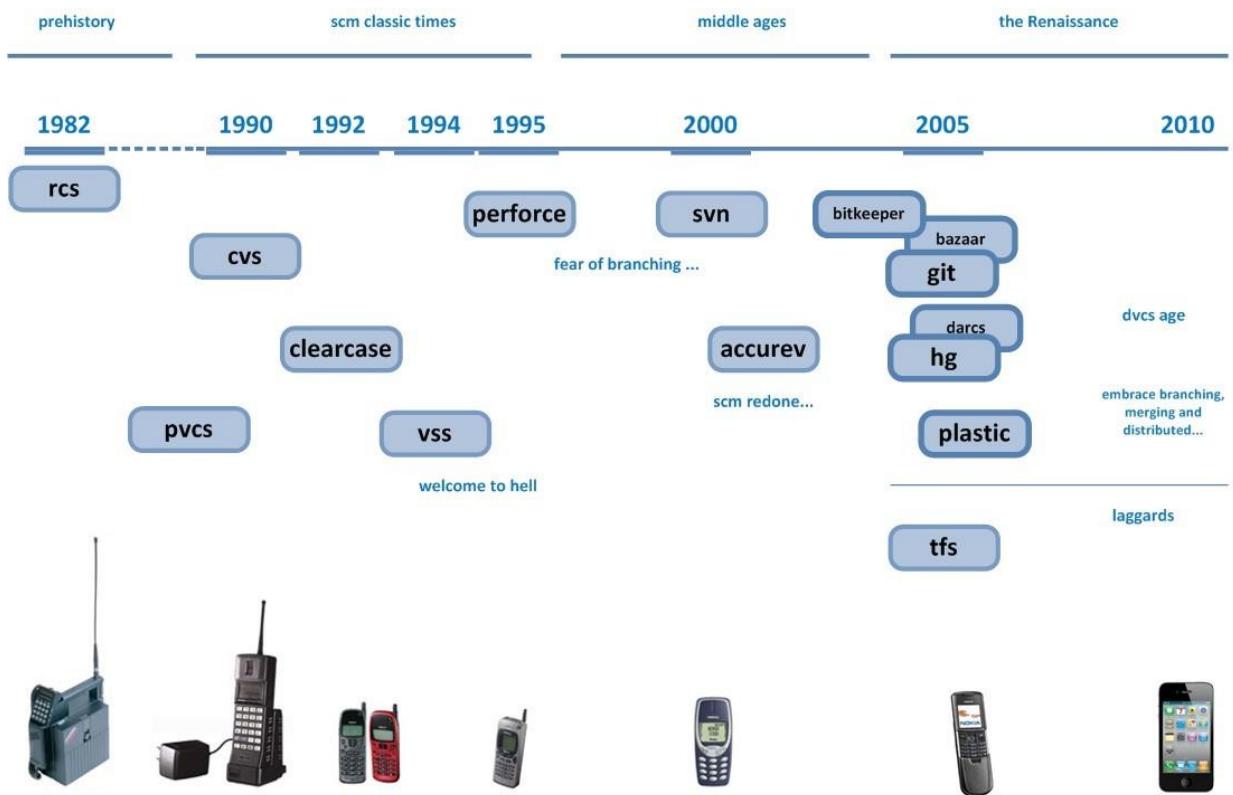
Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы управления версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM). Управление версиями используется в инструментах конфигурационного управления (Software Configuration Management Tools).

История развития систем контроля версий

Условно, развитие систем контроля версий можно разбить на три этапа:
темная эпоха, средневековье, ренессанс.

Темная эпоха

Самой первой системой контроля версий была система «скопировать и вставить», когда большинство проектов просто копировалось с места на место с изменением название (проект_1; проект_новый; проект_новее и т.д.), как правило в виде zip архива или подобных (arj, tar). Естественно, такие манипуляции над файловой системой едва ли можно назвать хоть скольконибудь полноценной системой контроля версий (или системой вообще). Для решения этих проблем в 1982 году появляется RCS.



RCS

Одной из основных нововведений RCS было использование дельт для хранения изменений (т.е. хранятся только те строки, которые изменились, а не весь файл). Однако он имел ряд недостатков.

Прежде всего, он был только для текстовых файлов. Не было центрального репозитория; каждый версионированный файл имел собственный репозиторий в виде rcs файла рядом с самим файлом. Т.е., если на проекте было 100 файлов, рядом ложилось 100 rcs файлов. В лучшем случае, эти 100 файлов образовывались в директории RCS (при правильной настройке). Именование версий и веток было невозможным.

Классическая Эра

В начале 90х началась т.н. классическая эра систем контроля версий. Самый яркий (и поныне используемый) представитель – CVS. Классическую эпоху можно охарактеризовать достаточно сформированным представлением о системах контроля версий, их возможностях, появлением центральных репозиториев (и синхронизации действий команды).

CVS

CVS (Concurrent Version System) была создана в 1990. Она умела работать с множеством версий, разрабатываемых параллельно на различных машинах и содержала данные на центральном сервере.

CVS умела обрабатывать версии достаточно хорошо – включая ветки (branching) и слияния (merging).

CVS не хранила изменений в названиях файлов или папок и работала на основе блокировок всего репозитория (не отдельных файлов).

ClearCase

В 1992 появился один из основных представителей мира систем контроля версий. ClearCase был однозначно впереди своего времени и для многих он до сих пор является одной из самых мощных систем контроля версий когда-либо созданных.

Данная система позволяла пользоваться виртуальной файловой системой для хранения и получения изменений; имела широкий спектр возможностей по конфигурации, внедрению в процесс разработки (аудит сборок продукта, версии, слияния изменений, динамические представления); запускалась на множестве различных систем.

SVN

SVN – надежная и быстродействующая система контроля версий, на данный момент разрабатываемая в рамках проекта Apache Software Foundation. Она реализована по технологии клиент-сервер и отличается невероятной простотой – две кнопки (*commit*, *update*).

Несмотря на это, SVN очень плохо умеет создавать и сливать ветки и плохо решает конфликтные ситуации с версиями.

Ренессанс

Git

Линус Торвальдс, т.н. отец Линукса, разработал и внедрил первую версию Гит для предоставления возможности разработчикам ядра линукс проводить контроль версий не только в BitKeeper.

Гит представляет собой систему *распределенного контроля версий*, когда каждый разработчик имеет собственный репозиторий, куда он вносит изменения. Далее система гит синхронизирует репозитории с центральным репозиторием. Это позволяет проводить работу независимо от центрального репозитория (в отличие от svn, когда версионирование предполагало наличие связи с центральным сервером), перекладывает сложности ведения веток и склеивания изменений больше на плечи системы, чем разработчиков и др.

Изменения считаются в виде наборов изменений (*changeset*), который получает уникальный идентификатор (хеш-сумма на основе самих изменений).

Mercurial

Mercurial был создан также как и гит после объявления о том, что BitKeeper более не будет бесплатным для всех. Во многом похожий на гит, меркуриал также использует идею наборов изменений, но в отличие от гит, хранит их не в виде узла в графе, а в виде плоского набора файлов и папок, называемых *revlog*.

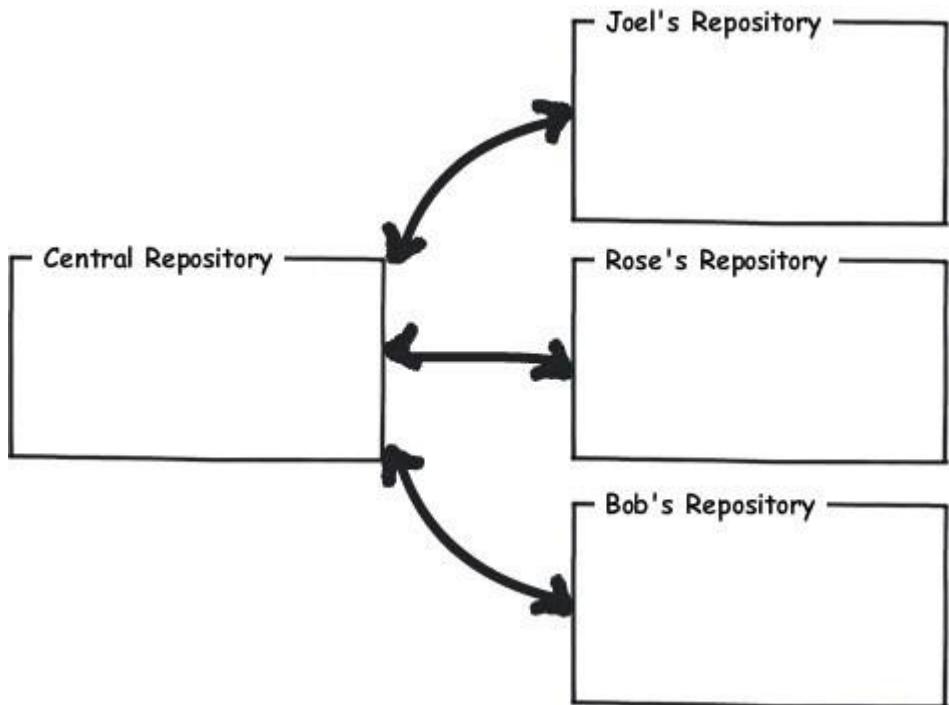
Работа с Mercurial

Подробнее о работе с Mercurial можно прочитать по следующим ссылкам:

<http://hginit.com/> <http://hgbook.red-bean.com/>

Для работы с Mercurial крайне рекомендуется установить TortoiseHg – визуальную оболочку для командной строки меркуриала. Дальнейшие инструкции покажут, как работать именно с визуальной оболочкой.

Основная идея меркуриала – каждый разработчик имеет собственный репозиторий, куда складываются изменения (версии) файлов, и синхронизация между разработчиками идет посредством синхронизации репозиториев. Процесс работы выглядит следующим образом:

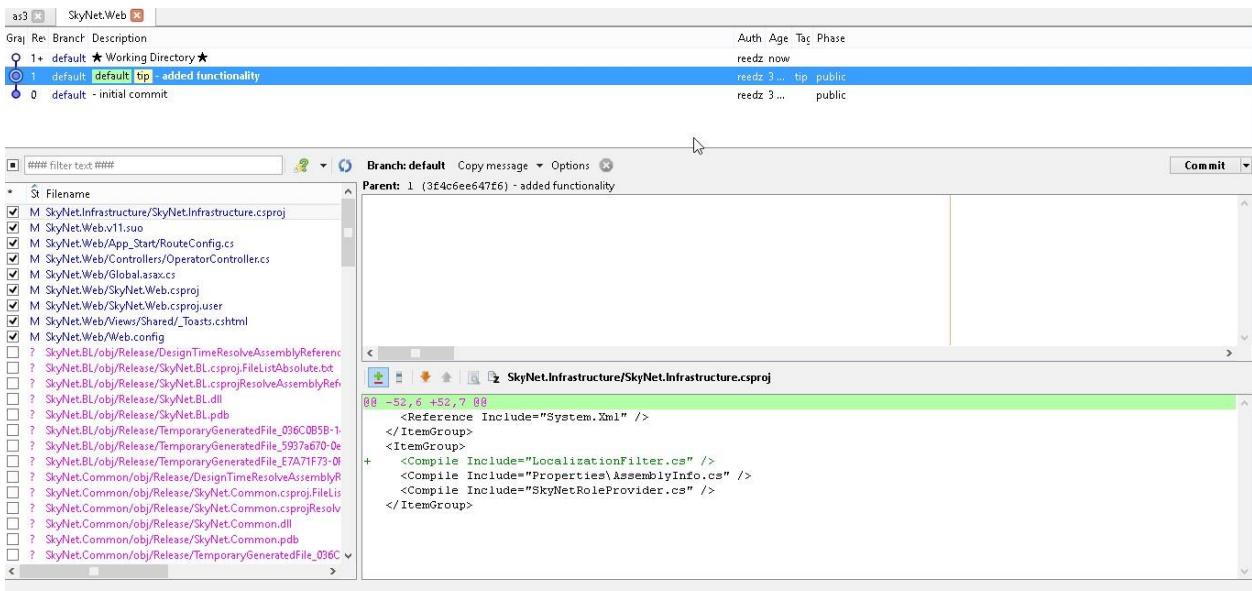


Соответственно, есть 5 основных команд для работы:

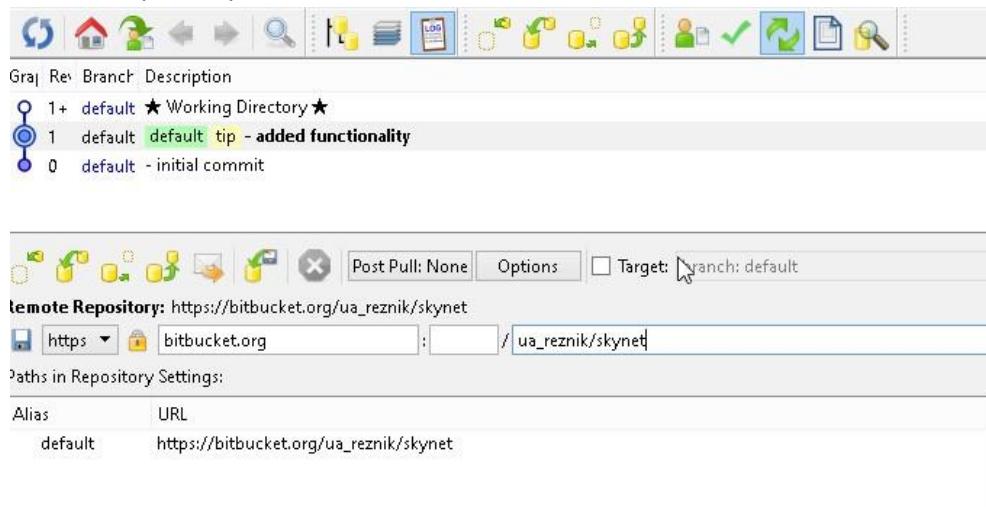
- Клонировать репозиторий – получить копию репозитория на локальную машину для последующей работы с ним;
- Синхронизация репозиториев – получить изменения – pull – получение изменений из удаленного (исходного, центрального, либо любого другого такого же) репозитория
- Синхронизация репозиториев – переслать изменения – push – передача собственных изменений в удаленный репозиторий - Записать изменения – commit – создание новой версии
- Обновиться до версии – update – обновиться до определенной версии, присутствующей в репозитории.

В TortoiseHG это выглядит следующим образом:

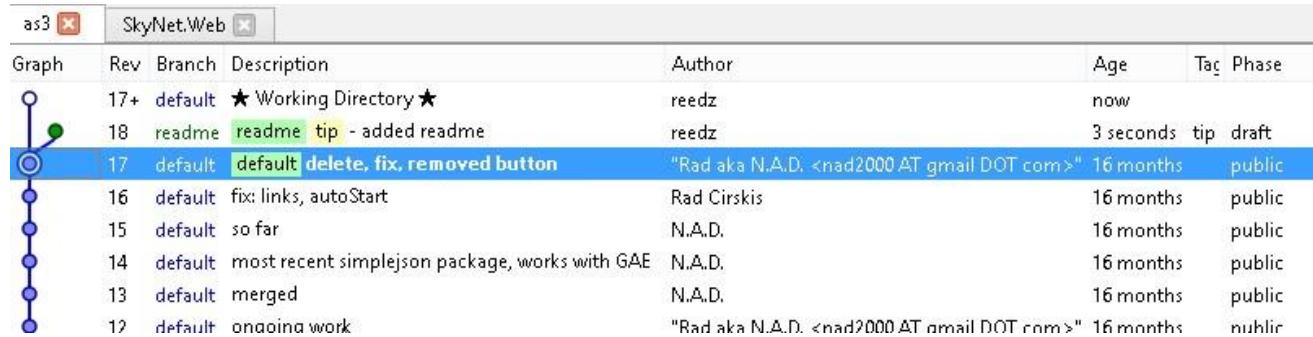
Это основное рабочее окно (TortoiseHG Workbench) предоставляющее все возможности работы с репозиториями. Кнопки (1) и (2) позволяют выполнить команды push и pull соответственно. Кнопка (3) позволяет выполнить команду commit, но для этого необходимо будет выбрать передаваемые изменения и ввести текст коммита:



Кнопка (4) позволяет посмотреть, с какими репозиториями будет выполняться синхронизация, и выполнить команды push и pull:



Окно (5) представляет собой т.н. revision log – набор версий данного репозитория. Тут выводятся данные о номере ревизии, все описания (commit message), автор изменений и дата изменений. Также выводится удобный график изменений, позволяющий визуализировать ветки и склеивания:



Окно(6) представляет собой детали конкретной ревизии(версии) – какие файлы были изменены, добавлены, удалены, переименованы. Окно (7) представляет собой блок для вывода commit message.

Окно (8) показывает изменения по текущему выбранному файлу в данной ревизии в режимах – только изменения, текст и изменения, весь текст.

ЛАБОРАТОРНАЯ РАБОТА №2.

Основы отладки программных продуктов

Задание

1. Ознакомиться с краткими теоретическими сведениями.
2. Запустить любой из open source проектов на отладку, проверить различные методы и способы отладки.
3. Подготовить отчет о выполнении лабораторной работы.

Краткие теоретические сведения

Не смотря на некоторую непривлекательность факта, следует отметить — отладка программного обеспечения занимает около 80% времени программистов. На стадии поддержки программного продукта эта цифра может доходить до 100%, ведь известен факт — программный продукт может разрабатываться до года, а применяться — в течении пяти и более лет.

Однако программисты редко уделяют достаточно внимания принципам и технике отладки программного обеспечения, в основном оттачивая собственные навыки написания кода. В данной лабораторной работе рассмотрим общие принципы отладки, действенные способы и средства отладки, а также поближе познакомимся со встроенными средствами отладки Visual Studio.

Основы отладки программных продуктов

Вопрос о необходимости отладки возникает лишь тогда, когда в программный продукт закрадывается уязвимость/ошибка/проблема, которую сложно выявить без отладчика. Более того, для большинства программистов, отладка — процесс работы с отладчиком либо процесс написания решения (fix) для проблемной ситуации.

На самом деле отладка — понятие более обширное. Процесс отладки можно разделить на следующие этапы:

1. Процесс определения причин неисправности;
2. Устранение неисправности;
3. Проверка, что во время устранения неисправности не было задето ничегобольше;
4. Поддержания общего качества кода на соответствующем уровне
(читаемость, поддерживаемость, поддержка архитектурных решений); 5. Убеждение в том, что данная проблема более нигде не появится вновь.

Процесс определения причин неисправности является наиболее главным во всем процессе отладки. На данном этапе необходимо выявить, каковы симптомы неисправности и каковы реальные ее причины. Часто возникает ситуация, когда программисты тратят уйму времени, устраняя симптомы неисправности; это приводит к различным результатам — ошибка может появиться вновь в другом месте, либо ошибка демонстрирует новые симптомы.

Процесс устранения неисправности достаточно прямолинейный — необходимо определить изменения, которые необходимо внести чтобы устранить неисправность. Однако, при этом

необходимо помнить о всех последующих этапах отладки — изменения в исходный код программного продукта должны лишь устранить неисправность и не повлиять на общее функционирование программы. Об этом, к сожалению, разработчики часто забывают, что приводит к кошмарным последствиям — количество ошибок после отладки растет, а не уменьшается.

Также необходимо помнить, что устранение неисправностей — это написание исходных кодов того же качества, как и написание «нового» кода. Часто возникает желание не тратить длительное время на обдумывание «правильных» реализаций устранения неисправности, а просто «залатать» ошибку и продолжить разработку. Данный подход приведет к тому, что заметно упадет качество программного продукта; последующая поддержка и доработка системы станет весьма сложной; падение качества программного продукта отрицательно скажется на моральном благосостоянии разработчиков.

И самое главное при устранении неисправностей — избегание повторного появления ошибок. В индустрии разработки программного обеспечения это называют термином «регрессия» - повторное появление ранее устранивших ошибок. На первый взгляд может показаться, что нет ничего дурного в том, чтобы ошибки (по-крайней мере какая-то их часть) возникали вновь; однако при возникающих вновь ошибках наблюдаются следующие негативные факторы: время, потраченное программистом однажды, приходится тратить вновь; неконтролируемый рост числа ошибок программного продукта. Для избежания регрессии рекомендуемой практикой является написание регрессионных тестов.

Устранение неисправностей

Непосредственный процесс поиска и устранения неисправности можно разделить на 4 шага:

1. Воспроизведение ошибки;
2. Диагностика ошибки;
3. Устранение ошибки;
4. Рефлексия.

Любой процесс отладки начинается с воспроизведения ошибки. Это закономерно, поскольку ошибку тяжело починить, не имея возможности проверить представленное решение (или вообще найти решение). Потому, к воспроизведению ошибки ставятся специальные требования.

Прежде всего, воспроизведение ошибки должно быть гарантированным. Должна быть выработана такая последовательность действий, в результате которой абсолютно однозначно возникает ошибка. Если ошибка возникает лишь в части случае, это означает, что не все параметры были определены верно (параметры окружения или входные данные).

Воспроизведение также должно быть настолько быстрым, насколько возможно. Это необходимо для эффективного и быстрого поиска решений. В случае, если обратная связь будет длительной, т. е. разработчику необходимо ждать час, прежде чем подтвердится или опровергнется факт разрешения уязвимости, такая ошибка может еще долго не устранена.

Воспроизведение, по возможности, должно быть автоматизировано — это уменьшит время и ресурсы, необходимые для воспроизведения ошибки, а таким образом увеличит скорость устранения неисправности.

Для воспроизведения ошибки всегда необходимо соблюдать следующие правила:

1. Необходимо пользоваться той же версией программного обеспечения, в которой была выявлена неисправность — в противном случае можно долго гоняться за «phantomной» неисправностью (которая могла быть уже решена или еще не введена, в зависимости от используемой версии программного продукта);
2. Необходимо пользоваться тем же окружением, в котором наблюдалась переменная — например, операционной системой либо браузером.
3. Необходимо передавать именно те параметры, при которых наблюдалась неисправность (любые входные данные — нажатия на клавиши в определенных последовательностях и т. п.)

Диагностика — процесс нахождения рационального объяснения причин возникновения ошибки после ее воспроизведения. Поскольку компьютер представляет собой полностью детерминированную машину (каждое приложение имеет фиксированное состояние в фиксированный момент времени в зависимости от входных и выходных величин), у каждого происшествия имеется конкретное объяснение.

Процесс диагностики представляет собой процесс постановки таких экспериментов над исходным кодом, при которых становится однозначно понятно, какие участки кода «виноваты» за неисправность. Для этого:

1. После воспроизведения ошибки и изучения релевантных участков кода сначала строится гипотеза — например, за ошибку виноват модуль просчета числа дней в месяце, поскольку он не учитывает наличие високосных лет
2. Для подтверждения или опровержения гипотезы ставится эксперимент — исходный код модифицируется таким образом, чтобы воспроизвести эксперимент — модуль просчета числа дней в месяце возвращает правильное значение (пускай не считает, а просто возвращает, без разницы на данном этапе).
3. В случае, если гипотеза подтверждена — можно считать корень проблемы найденным. В обратном случае необходимо начать сначала.

Естественно, гипотезы и эксперименты должны доказывать определенные вещи; целью является выявление причин неисправности, а не постановка экспериментов.

При постановке экспериментов одно из наиболее базовых правил — применение не более одного набора изменений за раз. Если у вас есть две гипотезы по поводу ошибки, не надо применять их обе, иначе в случае разрешения неисправности тяжело сказать, что конкретно помогло — первая или вторая гипотеза.

В случае, если было перепробовано много гипотез, можно потеряться в том, что уже было испытано. Для этого можно вести специальный дневник (или просто черновик) испробованных гипотез. Туда же можно дописывать идеи для последующих гипотез. Такой дневник следует периодически просматривать — вдруг, придут новые мысли, которые помогут решить текущую проблему.

Во время процесса построения гипотез очень важно владеть необходимым инструментарием и средствами отладки.

Прежде всего, к таким средствам следует отнести традиционный отладчик. Отладчик позволяет просматривать состояния всех переменных в фиксированный момент времени выполнения программы. Об этом будет сказано чуть позднее. Данная возможность позволяет отыскать неверные результаты обработок и калькуляций внутри программного продукта; определять неверность входных и выходных параметров.

В случае, если по каким-то причинам отладчик недоступен, есть более дешевые средства отладки. Наиболее традиционным является отладка посредством вывода в консоль либо в виде всплывающих окон (MessageBox), отображающих значения текущих переменных.

Также для отладки можно использовать инструменты логирования. По сути, это тот же вывод, только в текстовый файл. Обработки и анализ лог файл часто могут дать общее представление о том, в каком месте программы что-то пошло не так.

Однако, включение последних двух вариантов требует внесения изменений в программный код, которые многие забывают удалить в последствии.

Еще одной важной техникой отладки является «разделяй и властвуй». Для определения проблемного участка кода необходимо определить, какие компоненты/участки кода **не влияют** на воспроизведение ошибки. Таким образом значительно сужается круг поиска. Для этого можно отключить внешние компоненты, которые интегрируются с программой, определить, после выхода какой функции появляется ложный результат. В основу идеи положена идея бинарного поиска: вместо того, чтобы перебирать весь набор элементов от начала до конца, набор делится на две части, и поиск продолжается лишь в той части, где может находиться число.

Необходимо заранее также отбросить все, даже наименее вероятные, гипотезы, тем самым еще сужая область поиска ошибки.

Часто в отладке также помогают такие средства разработки программного обеспечения, как системы контроля версий. В случае, если ошибка была внедрена в последней версии, а в версии до этого ее не было, можно просмотреть изменения и определить проблемные участки. Этот вариант работает лишь, если поддерживается грамотность работы с репозиториями — все наборы изменения заливаются мелкими, рабочими порциями.

Существует также набор часто допускаемых разработчиками ошибок во время диагностики. К таким относят:

1. Изменение не тех участков кода/проектов — иногда возникает ситуация, что разработчик не понимает, почему не видит собственных изменений; оказывается, он вносит изменения в другой продукт либо забывает их применить (если это веб-сайт, то часто разработчики забывают обновить css стили или javascript файлы, кешируемые браузером);
2. Проверка предположений — каждое предположение (сколько бы невероятным оно неказалось) необходимо тщательно проверять;
3. Наличие множества симптомов — необходимо изолировать симптомы и искать причины каждого симптома по-отдельности;
4. Непредвиденные изменения во время отладки — например, функция, работавшая вчера, сегодня не работает; необходимо определить и изолировать те переменные окружения, которые

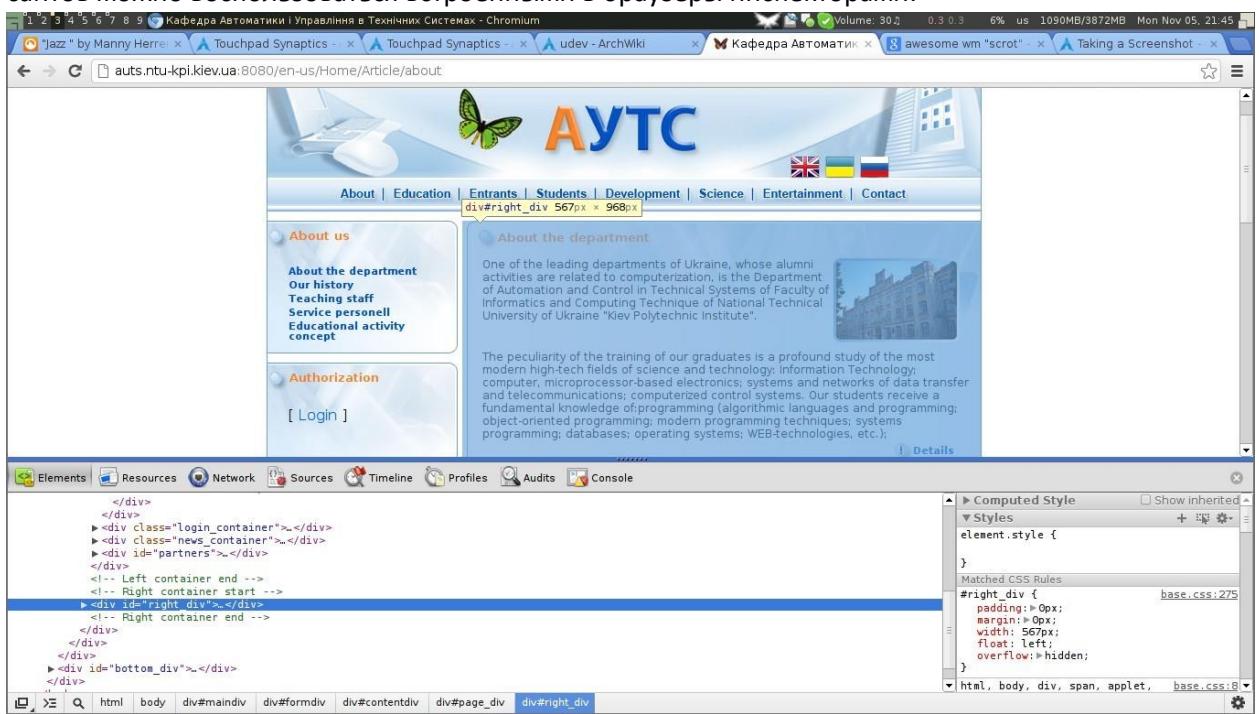
влияют на правильную работу системы (например, если система настроена работать определенным образом до 12 дня и после 12 дня).

Во время устранения неисправности необходимо выполнять следующее:

1. Придерживаться следующих целей — устраниить неисправность; избежать регрессии; поддерживать код на должном уровне качества. Для этого можно использовать модульные тесты для покрытия ошибки и использовать рефакторинг для улучшения исправленного кода.
2. Начинать с чистой ревизии исходного кода (без лишних изменений).
3. Проверить, что все тесты (если имеются), проходят, до того как приступить к исправлению неисправности.
4. Исправлять причину, а не симптомы.
5. Для каждого логического изменения делать один коммит в репозиторий.

Средства отладки веб-сайтов

Со времен первых html страниц значительно продвинулись не только средства разработки сайтов, но и средства их отладки. Для отладки клиентской части (javascript, html верстка) современных сайтов можно воспользоваться встроенными в браузеры инспекторами:



Данные инспекторы имеют следующую функциональность:

1. Отображение дерева DOM с возможностью их изменения, просмотр применения CSS-стилей (и их изменения), а также с отображением «просчитанного стиля» (т. е. сумма всех примененных стилей);

2. Просмотр загруженных ресурсов (картинки, css и js файлы и т. п.);
3. Просмотр длительности и последовательности загрузки ресурсов(отображение фактических http-запросов на сервер и ответов на них);
4. Просмотр и отладка javascript файлов при помощи точек останова;
5. Интерактивная консоль для выполнения javascript кода.

Данной функциональности абсолютно достаточно для разработки и проверки правильности верстки, наложения css стилей и разработки и отладки javascript функций.

Средства отладки Visual Studio

Встроенный отладчик Visual studio имеет следующие функциональные возможности:

1. Отображение используемых текущих переменных в окне «locals»;
2. Отображение и уведомление об изменении значений функций либо переменных, определенных пользователем, в окне «watch» - сюда можно вписывать произвольные функции (например, DateTime.Now) либо некоторые переменные (MyService.CurrentTime); при изменении значения во время прохода значение подсветится красным цветом;
3. Возможность установки точек останова, при попадании на которые программа остановит процесс выполнения и перейдет в область разработки Visual Studio;
4. Создание дампов памяти, регистров;
5. Возможность перехода назад по пути исполнения программы (путем перетягивания курсора, указывающего текущую выполняющуюся строчку);
6. Возможность ведения лога выполненных функций и произошедших событий, и переход между ними (IntelliTrace);
7. Настраиваемый механизм точек останова;
8. Интерактивная подсказка (значения переменных отображаются при наведении на них курсора мыши);
9. Многопоточная отладка — отладчик способен воспринимать точки останова различных потоках;
10. Интерактивная консоль для выполнения инструкций;

Основным инструментом разработчика при отладке в Visual Studio является механизм точек останова, позволяющих «заморозить» приложение в определенный момент исполнения. Точки останова можно настраивать следующим образом:

- Условие остановки — можно задавать некоторое логическое условие остановки при помощи стандартных логических операторов;
- Остановку по проходу n-ый раз — можно задать, что точка останова срабатывает лишь после 10го прохода по данному участку кода;
- Вывод в интерактивную консоль — настроить вывод определенных данных в интерактивную консоль по проходу по точке останова;
- Отключить часть точек останова вручную либо по указанному фильтру.

ЛАБОРАТОРНАЯ РАБОТА №3

Управление проектами. Системы управления проектами

Задание

1. Ознакомиться с краткими теоретическими сведениями.
2. Завести проект в системе управления проектами.
4. Завести в системе управления проектами задачи, предварительно разбив их на следующие категории: неисправность, улучшение, функциональная возможность, другое.
5. Завести в системе набор компонентов системы.
6. Завести в системе несколько версий системы.
7. Распланировать даты релизов и соответствующие задачи по версиям.

Краткие теоретические сведения

Управление проектами

Управление проектами (project management) — область деятельности, в ходе которой определяются и достигаются четкие цели проекта при балансировании между объемом работ, ресурсами (такими как деньги, труд, материалы, энергия, пространство и др.), временем, качеством и рисками. Управление проектами — сочетание науки и искусства, которые используются в профессиональных сферах проекта, чтобы создать продукт проекта, который бы удовлетворил миссию проекта, путем организации надежной команды проекта, эффективно сочетающей технические и управленческие методы, создает наибольшую ценность и демонстрирует эффективные результаты работы.

Управление проектом — это нахождение такого баланса между ограничениями, рисками и ресурсами, который бы удовлетворял большинство требований проекта. Управление проектов традиционно рассматривается в простой модели «треугольника», называемой тройственной ограниченностью.



Тройственная ограниченность описывает баланс между содержанием проекта, стоимостью, временем и качеством. Качество было добавлено позже, поэтому изначально именовалась как тройственная ограниченность.

Как того требует любое начинание проект должен протекать и достигать финала с учетом определенных ограничений. Классически эти ограничения определены как содержание проекта, время и стоимость. Они также относятся к Треугольнику Управления проектами, где каждая его сторона представляет ограничение. Изменение одной стороны треугольника влияет на другие стороны. Дальнейшее уточнение ограничений выделило из содержания качество и действие, превратив качество в четвертое ограничение.

Ограниченностремени определяется количеством доступного времени для завершения проекта. Ограниченностистоимости определяется бюджетом, выделенным для осуществления проекта. Ограниченностисодержания определяется набором действий, необходимых для достижения конечного результата проекта. Эти три ограниченности часто соперничают между собой. Изменение содержания проекта обычно приводит к изменению сроков (времени) и стоимости. Сжатые сроки (время) могут вызвать увеличение стоимости и уменьшение содержания. Небольшой бюджет (стоимость) может вызвать увеличение сроков (времени) и уменьшение содержания.

Иной подход к управлению проектами рассматривает следующие три ограниченности: финансы, время и человеческие ресурсы. При необходимости сократить сроки (время) можно увеличить количество занятых людей для решения проблемы, что непременно приведет к увеличению бюджета (стоимость). За счет того, что эта задача будет решаться быстрее, можно избежать роста бюджета, уменьшая затраты на равную величину в любом другом сегменте проекта.

Существует множество подходов к управлению проектами в зависимости от типа проекта:

- Предположение о неограниченности ресурсов, критичен только срок выполнения и качество. Метод PERT, Метод критического пути;
- Предположение о критичности качества, при этом требования к сроку и ресурсам достаточно гибки (под качеством здесь понимается полнота удовлетворения потребностей, как известных, так и неизвестных заранее, часто создаваемых выходом нового продукта). Гибкая методология разработки;
- Предположение о неизменности требований, низких рисках, жесткий срок. Водопадная методология разработки;
- Предположение о высоких рисках проекта (инновационные проекты, стартапы). Lean методология разработки.

Необходимо понимать, что управление проектами и методология разработки — две разные вещи. Методология описывает характер процесса разработки, т. е. технической стороны реализации проекта. Управление проектом в себя включает значительно больше дисциплин — тут также подразумевается взаимодействие с потенциальными клиентами, решение юридических, финансовых и кадровых вопросов, маркетинг, планирование и общее видение проекта.

С точки зрения разработчиков, основная задача управляемцев — отгородить разработчиков от «неважных» деталей проекта (любые юридические проблемы и подобное) и предоставить грамотный и как можно более точный список требований к программному обеспечению.

С точки зрения менеджеров, основная задача управления — формулировать видение проекта, координировать работу различных отделов (например, риска-аналитики, бизнес-аналитики, разработчики и тестировщики), планировать и приоритезировать работу и следить за ходом ее выполнения.

Для поддержания работ на всех уровнях управления проектами были разработано т. н. программное обеспечение для управления проектами.

Программное обеспечение для управления проектами — определение для комплексного программного обеспечения, включающее в себя приложения для планирования задач, составления расписания, контроля цены и управления бюджетом, распределения ресурсов, совместной работы, общения, быстрого управления, документирования и администрирования системы, которое используются совместно для управления крупными проектами.

В задачи таких комплексов входит планирование, информирование и др.

Планирование

Одной из наиболее распространенных возможностей является возможность планирования событий и управления задачами. Требования могут различаться в зависимости от того, как используется инструмент. Наиболее распространенными являются:

- планирование различных событий, зависящих друг от друга;
- планирование расписания работы сотрудников и управление ресурсами;
- расчет времени, необходимого на решение каждой из задач; •сортировка задач в зависимости от сроков их завершения;
- управление нескольким проектами одновременно.

Информирование

Программное обеспечение для управления проектами предоставляет большое количество требуемой информации, такой как:

- список задач для сотрудников и информацию распределения ресурсов;
- обзор информации о сроках выполнения задач;
- ранние предупреждения о возможных рисках, связанных с проектом; •информации о рабочей нагрузке;
- информация о ходе проекта, показатели и их прогнозирование.

Другое

Программное обеспечение для управления проектами дополнительно может предоставлять следующие функциональные возможности:

- Ведение базы знаний проекта (wiki)
- Ведение учета потребителей и их жалоб (CRM)
- Ведение деловых переговоров и обсуждений(между отделами, с клиентами и др.)
- Совместная работа над документами

Ход работы

1. Зайти по адресу системы управления проектами: <http://5.175.130.122/redmine>



2. Завести проект

The screenshot shows the Project Name dashboard with the 'Overview' tab selected. The main content area displays a brief project description and a 'Issue tracking' section with a summary of open bugs, features, and support requests. On the right, there's a 'Spent time' panel showing 0.00 hours and links to 'Details' and 'Report'. The top navigation bar includes links for Home, My page, Projects, Administration, Help, and a search bar.

3. Завести задачи

The screenshot shows the 'New issue' creation form. The 'Tracker' is set to 'Feature'. The 'Subject' is 'First Task'. The 'Description' field contains sample text about a task. Below the description, there are fields for 'Status' (New), 'Priority' (Normal), 'Assigned to' (empty), and 'Start' date (2013-02-17). There are also fields for 'Due date' (2013-02-22), 'Estimated time' (1 Hours), and '% Done' (0 %). A 'Files' section with a 'Choose File' button is present, along with an 'Optional description' field and a 'Watchers' section. At the bottom are 'Create' and 'Create and continue' buttons.

4. Указать milestone'ы к задачам — версии ПО, к которым будут реализованы соответствующие задачи

The screenshot shows the 'New issue' creation form again, but now with a 'Milestone' field at the bottom containing the value '0.1'. The rest of the form fields are identical to the previous screenshot.

5. Сохранять диаграммы (.mdl файлы) как документацию в Documents

New document

Category [Technical documentation](#)

Title *

Description

[B](#) [I](#) [U](#) [S](#) [C](#) [H1](#) [H2](#) [H3](#) [List](#) [Table](#) [Image](#) [Pre](#) [File](#)

First try

Files smt.gaphor

Add another file (Maximum size: 5 MB)

[Create](#) [Cancel](#)

6. Создать Wiki систему для хранения знаний о проекте — принимаемые решения, спорные моменты, детали реализации, ссылки на похожие проекты и подобное.

[Overview](#) [Activity](#) [Issues](#) [New issue](#) [Gantt](#) [Calendar](#) [News](#) [Documents](#) [Wiki](#) [Files](#) [Repository](#) [Settings](#)

Wiki

B I U S C H1 H2 H3 List Table Image Pre

Text formatting: Help

h1. Wiki

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam elementum nibh sed sem dapibus vitae accumsan orci ultrices. Vivamus ac ante est. Nulla volutpat fermentum enim, vel dignissim tellus imperdiet non. Cras in libero ut tellus consequat tempor sit amet ut purus. Nunc purus nisl, accumsan in mattis sed, mollis sit amet eros. Cras purus turpis, posuere rhoncus imperdiet mollis, placerat scelerisque velit. Nam accumsan semper volutpat.

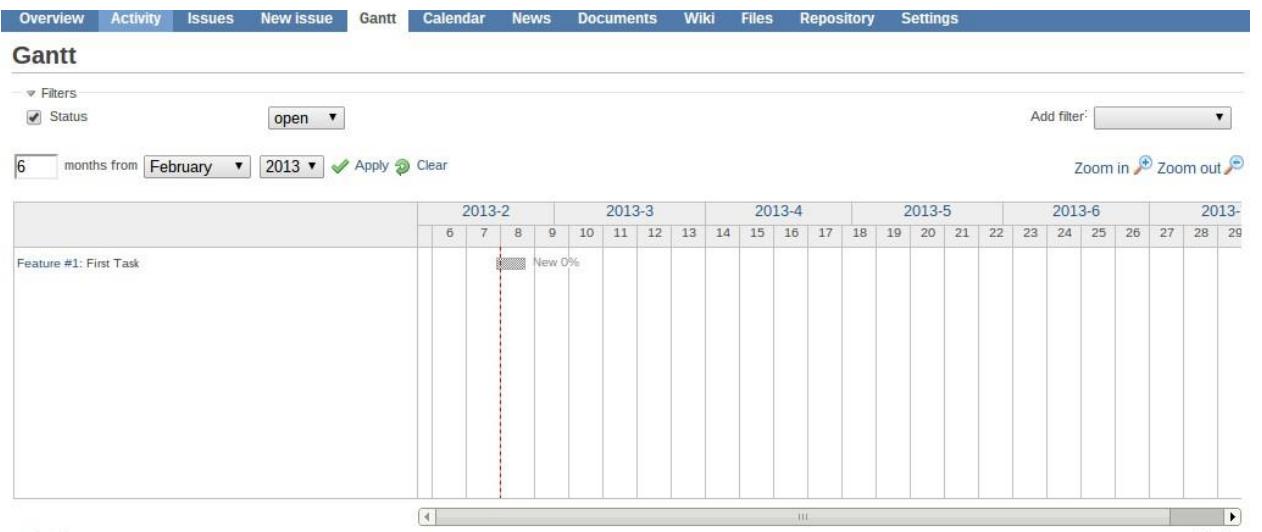
Comment

Files No file chosen

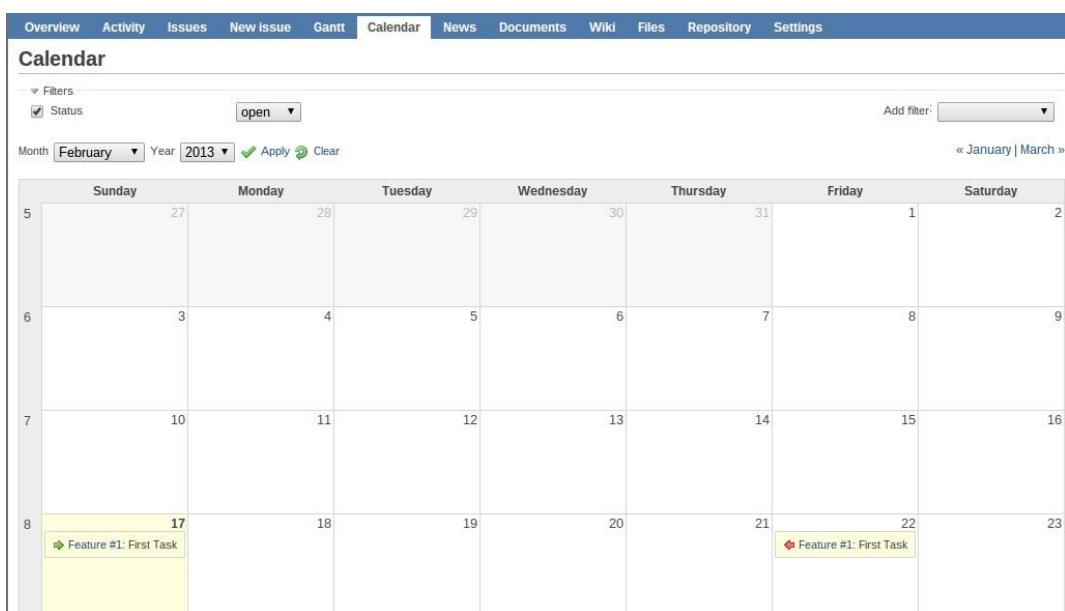
Add another file (Maximum size: 5 MB)

[Save](#) [Preview](#)

7. Рассмотреть диаграмму Ганнта и календарь задач.



Also available in: PDF



Вопросы для самопроверки.

1. Что такое системы управления задачами ?
2. Кто использует системы управления задачами ?
3. Как используют системы управления задачами разработчики ? Менеджеры ?
4. Что представляет собой «проект» в системе управления задачами ?
5. Что представляет собой «тикет» в системе управления задачами ?

ЛАБОРАТОРНАЯ РАБОТА № 4.

Диаграмма вариантов использования. Сценарии вариантов использования.

Задание

1. Ознакомиться с краткими теоретическими сведениями.
2. Проанализировать тему и нарисовать диаграмму вариантов использования согласно выбранной теме лабораторного цикла.
3. Выбрать 3 варианта использования и написать по ним сценарии использования.
4. Подготовить отчет о выполнении лабораторной работы.

Краткие теоретические сведения

Язык UML представляет собой общецелевой язык *визуального моделирования*, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем. Язык UML является достаточно строгим и мощным средством моделирования, которое может быть эффективно использовано для построения концептуальных, логических и графических *моделей* сложных систем различного целевого назначения. Этот язык вобрал в себя наилучшие качества и опыт методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

С точки зрения методологии ООАП (объектно-ориентированного анализа и проектирования) достаточно полная *модель* сложной системы представляет собой определенное число взаимосвязанных представлений (*views*), каждое из которых отражает аспект поведения или структуры системы. При этом наиболее общими представлениями сложной системы принято считать статическое и динамическое, которые в свою очередь могут подразделяться на другие более частные.

Принцип иерархического построения *моделей* сложных систем предписывает рассматривать процесс построения *моделей* на разных уровнях абстрагирования или детализации в рамках фиксированных представлений.

Уровень представления (*layer*) — способ организации и рассмотрения *модели* на одном уровне абстракции, который представляет горизонтальный срез архитектуры *модели*, в то время как разбиение представляет ее вертикальный срез.

При этом исходная или первоначальная *модель* сложной системы имеет наиболее общее представление и относится к концептуальному уровню. Такая *модель*, получившая название концептуальной, строится на начальном этапе проектирования и может не содержать многих деталей и аспектов моделируемой системы. Последующие *модели* конкретизируют концептуальную *модель*, дополняя ее представлениями логического и физического уровня. В целом же процесс ООАП можно рассматривать как последовательный переход от разработки наиболее общих *моделей* и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня. При этом на каждом этапе ООАП данные *модели* последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы.

В рамках языка UML все представления о *модели* сложной системы фиксируются в виде специальных графических конструкций, получивших название *диаграмм*.

Диаграмма (diagram) — графическое представление совокупности элементов *модели* в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика.

Нотация **канонических диаграмм** основное средство разработки *моделей* на языке UML.

В нотации языка UML определены следующие виды *канонических диаграмм*:

- вариантов использования (*use case diagram*)
- классов (*class diagram*)
- кооперации (*collaboration diagram*)
- последовательности (*sequence diagram*)
- состояний (*statechart diagram*)
- деятельности (*activity diagram*)
- компонентов (*component diagram*)
- развертывания (*deployment diagram*)

Перечень этих *диаграмм* и их названия являются *каноническими* в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения этих *диаграмм*. При этом совокупность построенных таким образом *диаграмм* является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы.

Каждая из этих *диаграмм* детализирует и конкретизирует различные представления о *модели* сложной системы в терминах языка UML. При этом *диаграмма* вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех остальных *диаграмм*. *Диаграмма* классов, по своей сути, логическая *модель*, отражающая статические аспекты структурного построения сложной системы.

Диаграммы кооперации и последовательностей представляют собой разновидности логической *модели*, которые отражают динамические аспекты функционирования сложной системы.

Диаграммы состояний и деятельности предназначены для моделирования поведения системы. И, наконец, *диаграммы* компонентов и развертывания служат для представления физических компонентов сложной системы и поэтому относятся к ее физической *модели*.

Диаграмма вариантов использования (Use-Cases Diagram)

Диаграмма вариантов использования (Use-Cases Diagram) - это UML *диаграмма* с помощью которой в графическом виде можно изобразить требования к разрабатываемой системе. *Диаграмма* вариантов использования – это исходная концептуальная модель проектируемой системы, она не описывает внутреннее устройство системы.

Диаграммы вариантов использования предназначены для:

- 1.Определение общей границы функциональности проектируемой системы; 2.Сформулировать общие требования к функциональному поведению проектируемой системы
- 3.Разработка исходной концептуальной модели системы;
- 4.Создание основы для выполнения анализа, проектирования, разработки и тестирования.

Диаграммы вариантов использования являются отправной точкой при сборе требований к программному продукту и его реализации. Данная модель строится на аналитическом этапе построения программного продукта (сбор и анализ требований) и позволяет бизнес-аналитикам получить более полное представление о необходимом программном обеспечении и документировать его.

Диаграмма вариантов использования состоит из ряда элементов. Основными элементами являются: *варианты использования* или прецеденты (use case), *актер* или действующее лицо (actor) и *отношения* между актерами и вариантами использования (relationship).

Актеры (actor)

Актером называется любой объект, субъект или система, взаимодействующая с моделируемой бизнес-системой извне для достижения своих целей или решения определенных задач. Это может быть человек, техническое устройство, программа или любая другая система, которая служит источником воздействия на моделируемую систему.



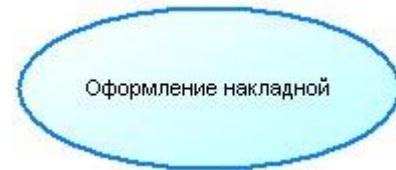
Изображение актеров на диаграммах UML

Имя актера должно быть достаточно информативным с точки зрения разрабатываемого программного обеспечения или предметной области. Для этой цели подходят наименования должностей в компании (например, продавец, кассир, менеджер, президент).

Варианты использования (use case)

Вариант использования служит для описания служб, которые система предоставляет актеру. Другими словами каждый вариант использования определяет набор действий, совершаемый системой при диалоге с актером. Каждый вариант использования представляет собой последовательность действий, который должен быть выполнен проектируемой системой при взаимодействии её с соответствующим актером, сами эти действия не отображаются на диаграмме.

Вариант использования отображается эллипсом, внутри которого содержится его краткое имя с заглавной буквы в форме существительного или глагола.



Обозначение варианта использования

Примеры вариантов использования: регистрация, авторизация, оформление заказа, просмотреть заказ, проверка состояния текущего счета и т.д.

Отношения на диаграмме вариантов использования

Отношение (relationship) — семантическая связь между отдельными элементами модели.

Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким службам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляемыми для всех них свой функционал.

Существуют следующие отношения: *ассоциации*, *обобщение*, *зависимость* (состоит из включение и расширение).

Ассоциация (association) – обобщенное, неизвестное отношение между актером и вариантом использования. Обозначается сплошной линией между актером и вариантом использования.



Направленная ассоциация (directed association) – то же, что и простая ассоциация, но показывает, что вариант использования инициализируется актером. Обозначается стрелкой.



Направленная ассоциация позволяет ввести понятие основного актера (он является инициатором ассоциации) и второстепенного актера (вариант использования является инициатором, т.е. передает актеру справочные сведения или отчет о выполненной работе).



Особенности использования отношения ассоциации:

1. Один вариант использования может иметь несколько ассоциаций с различными актерами.
2. Два варианта использования, относящиеся к одному и тому же актеру, не могут быть ассоциированы, т.к. каждый из них описывает законченный фрагмент функциональности актера.

Отношение обобщение (generalization) – показывает, что потомок наследует атрибуты и поведение своего прямого предка, т.е. один элемент модели является специальным или частным случаем другого элемента модели. Может применяться как для актеров, так для вариантов использования.

Графически отношение обобщения обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительский вариант использования.

Ниже на рисунке показано отношение обобщение. Вариант использования А называется предком (или родителем), а вариант использования Б – потомком (или дочерним) по отношению к А.

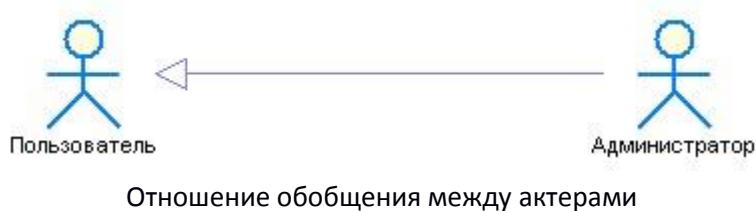


Потомки участвуют во всех отношениях родительских вариантов использования. В свою очередь, потомки могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от родителей свойства поведения.

Особенности использования отношения обобщения:

1. Главной особенностью отношения обобщения является то, что оно может связывать между собой только элементы одного типа.
2. Один вариант использования может иметь несколько родительских вариантов использования (множественное наследование).
3. Один вариант использования может быть предком для нескольких дочерних вариантов использования (таксономический характер отношений).

Также существует обобщение между актерами.

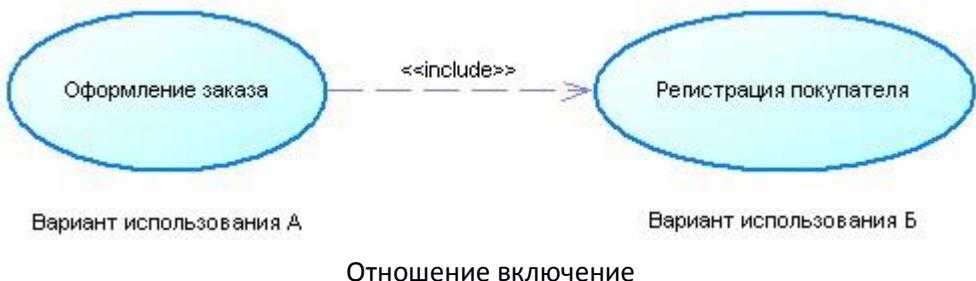


В приведенном примере Администратор наследует все атрибуты своего предка Пользователь, но может иметь свои индивидуальные, которые не изображены на рисунке.

Отношение зависимости (dependency) определяется как форма взаимосвязи между двумя элементами модели, предназначенная для спецификации того обстоятельства, что изменение одного элемента модели приводит к изменению некоторого другого элемента. В общем случае зависимость является направленным бинарным отношением, которое связывает между собой два элемента модели: независимый и зависимый.

Отношение включение (include) - частный случай общего отношения зависимости между двумя вариантами использования, при котором некоторый вариант использования содержит поведение, определенное в другом варианте использования.

Графическое изображение пунктирная стрелка с ключевым словом
<<include>>



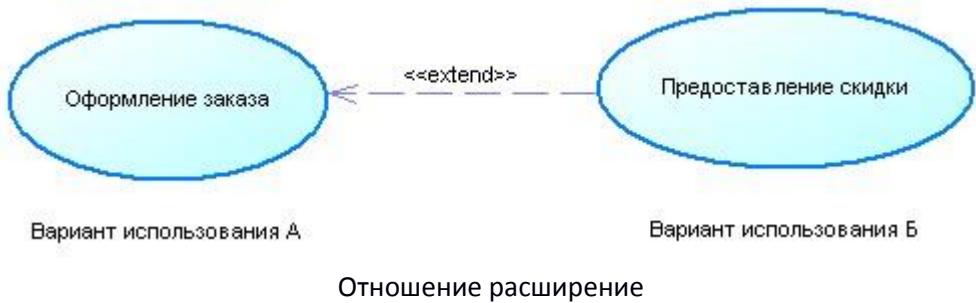
Зависимый вариант использования (Б) называют базовым, а независимый – включаемым (А). На рисунке включение означает, что каждое выполнение варианта использования А всегда будет включать в себя выполнение варианта использования Б. На практике отношение включения используется для моделирования ситуации, когда существует общая часть поведения двух или более вариантов использования. Общая часть выносится в отдельный вариант использования, т.е. типичный пример повторного использования функциональности.

Особенности использования отношения включения:

1. Один базовый вариант использования может быть связан отношением включения с несколькими включаемыми вариантами использования.
2. Один вариант использования может быть включен в другие варианты использования.
3. На одной диаграмме вариантов использования не может быть замкнутого пути по отношению включения.

Отношение расширение (extend) – показывает, что вариант использования расширяет базовую последовательность действий и вставляет собственную последовательность. При этом в отличие от типа отношений “включение” расширенная последовательность может осуществляться в зависимости от определенных условий.

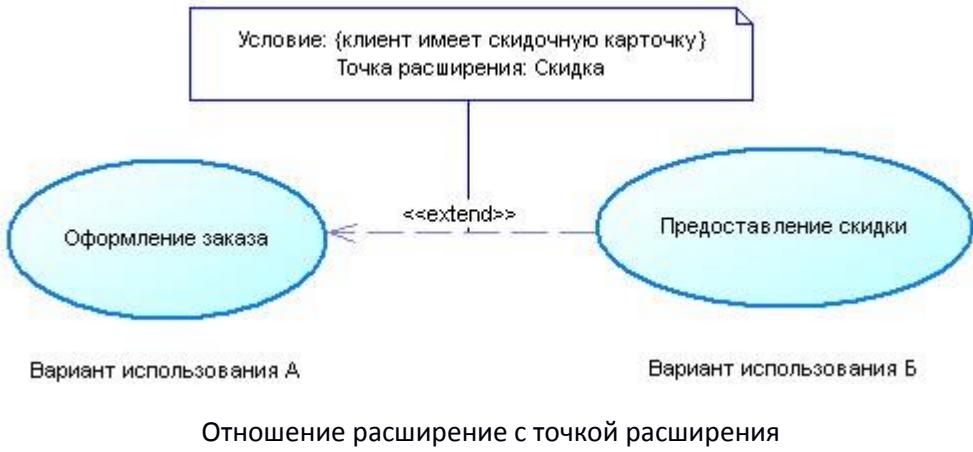
Графически изображение - пунктирная стрелка направленная от зависимого варианта (расширяющего) к независимому варианту (базовому) с ключевым словом `<<extend>>`.



Вариант использования А “Оформление заказа” является базовым и может быть расширен вариантом использования Б “Предоставление скидки”, например, при наличие у покупателя скидочной карточки.

Отношение расширение позволяют моделировать тот факт, что базовый вариант использования может присоединять к своему поведению некоторые дополнительные поведения за счет расширения в варианте другом варианте использования

Наличие такого отношения всегда предполагает проверку условия в точке расширения (extension point) в базовом варианте использования. Точка расширения может иметь некоторое имя и изображена с помощью примечания.



Особенности использования отношения расширения:

1. Один базовый вариант использования может иметь несколько точек расширения, с каждой из которых должен быть связан расширяющий вариант использования.
2. Один расширяющий вариант использования может быть связан отношением расширения с несколькими базовыми вариантами использования.
3. Расширяющий вариант использования может, в свою очередь, иметь собственные расширяющие варианты использования.
4. На одной диаграмме вариантов использования не может быть замкнутого пути по отношению расширения.

Сценарии использования

Диаграмма вариантов использования предоставляет знание о необходимой функциональности конечно системы в интуитивно-понятном виде, однако не несет сведений о фактическом способе ее реализации. Конкретные варианты использований могут звучать слишком общо и расплывчато и не являются пригодными для программистов.

Для документации вариантов использования в виде некоторой спецификации и для устранения неточностей и недоразумения диаграмм вариантов использований, как часть процесса сбора и анализа требований составляются так называемые сценарии использования.

Сценарии использования — это текстовые представления тех процессов, которые происходят при взаимодействии пользователей системы и самой системы. Они являются четко формализованными, пошаговыми инструкциями описывающими тот или иной процесс в терминах шагов достижения цели. Сценарии использования однозначно определяют конечный результат.

Сценарии использования описывают варианты использования на естественном языке. Они не имеют общего, шаблонного вида написания, однако рекомендуется следующий вид:

1. Предусловия — условия, которые должны быть выполнены для выполнения данного варианта использования;
2. Постусловия — что получается в результате выполнения данного варианта использования; 3. Взаимодействующие стороны;
4. Краткое описание;
5. Основной ход событий;
6. Исключения;
7. Примечания.

Рассмотрим пример входа студента в электронный кампус (вариант использования «получение электронных материалов с кампуса», один из подпроцессов).

Предусловия. Отсутствуют.

Постусловия. В случае успешного выполнения, пользователь входит в систему. В обратном случае, состояние системы не изменяется.

Взаимодействующие стороны. Студент, кампус.

Краткое описание. Данный вариант использования описывает вход пользователя в систему регистрации курсов.

Основной поток событий

Данный вариант использования начинает выполняться, когда пользователь хочет войти в систему регистрации курсов.

- 1.Система запрашивает имя пользователя и пароль.
- 2.Пользователь вводит имя и пароль.
- 3.Система проверяет имя и пароль, после чего открывается доступ в систему. В случае, если имя и пароль неверны, Исключение №1.

Исключения

Исключение №1

Неправильное имя/пароль. Если во время выполнения Основного потока обнаружится, что пользователь ввел неправильное имя и/или пароль, система выводит сообщение об ошибке.

Пользователь может вернуться к началу Основного потока или отказаться от входа в систему, при этом выполнение варианта использования завершается.

Примечания

Отсутствуют.

Вопросы для самопроверки.

1. Что такое UML ?
2. Какие диаграммы UML называют каноническими ?
3. Что собой представляет диаграмма вариантов использования ?

4.

ЛАБОРАТОРНАЯ РАБОТА №5

ОСНОВЫ ПРОTOTИПИРОВАНИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА Задание.

1. Ознакомиться с краткими теоретическими сведениями.
2. Реализовать прототипы пользовательского интерфейса для не менее чем 3х форм (и продемонстрировать Application Flow).
3. Реализовать обработку переходов между элементами пользовательского интерфейса (события переходов между формами и события на соответствующих элементах интерфейса) и добавить заполнение элементов тестовыми данными.

Краткие теоретические сведения.

Зачем прототипировать пользовательский интерфейс?

Может показаться несколько странным необходимость наличия опыта и умений прототипирования для разработчика. Разработчики, как правило, заняты созданием или изменением исходного кода, а прототипирование – ближе к работе дизайнера. Однак прототипирование несет массу выгод для разработчика:

- Позволяет быстро увидеть желаемый заказчиком результат;
- Позволяет более активно и предметно общаться по поводу приложения;- Наблюдать общую картину составления окон в проекте и соответствующих потоков данных;
- Позволяет увидеть взаимодействие элементов окон между собой (исоответствующий функционал);
- Позволяет более точно оценить объем работ;
- Позволяет закрепиться на достижении конкретного, осозаемого результата.

Нельзя сказать, что это основной инструмент разработчика, однако на первых этапах проекта (сбор и анализ требований, проектирование) – практически незаменимый. Известна простая истина, что чем позже найдена ошибка первых двух этапов – тем дороже ее исправить. Прототипирование позволяет справиться с этой проблемой, представив заказчику на утверждение конкретную концепцию пользовательского интерфейса; позднее эту концепцию можно использовать во время акта сдачи-приемки работы (чтобы указать, что все сделано согласно разработанной спецификации).

Прототипы сами по себе являются также одним из основных средств проектирования т.н. Visual Experience. Поскольку программы сами по себе не существуют, а разработаны для конкретных пользователей, крайне важно составить такой интерфейс, которым удобно пользоваться и выполнять соответствующие задачи. Поскольку пользователи, как правило, мало сведущи в компьютерных науках, самый простой способ проверить и разработать грамотный visual experience – предоставить прототипы на выбор и дать пользователям ими воспользоваться.

Инструменты прототипирования

Не будем останавливаться на огромной спецификации инструментов (критериев довольно много), в общем и целом их можно разделить на два класса: интерактивные и неинтерактивные.

Неинтерактивные прототипы (например, дизайны созданные в средстве Adobe Photoshop) позволяют увидеть «снимок» экрана работы программы. Такие прототипы на данный момент

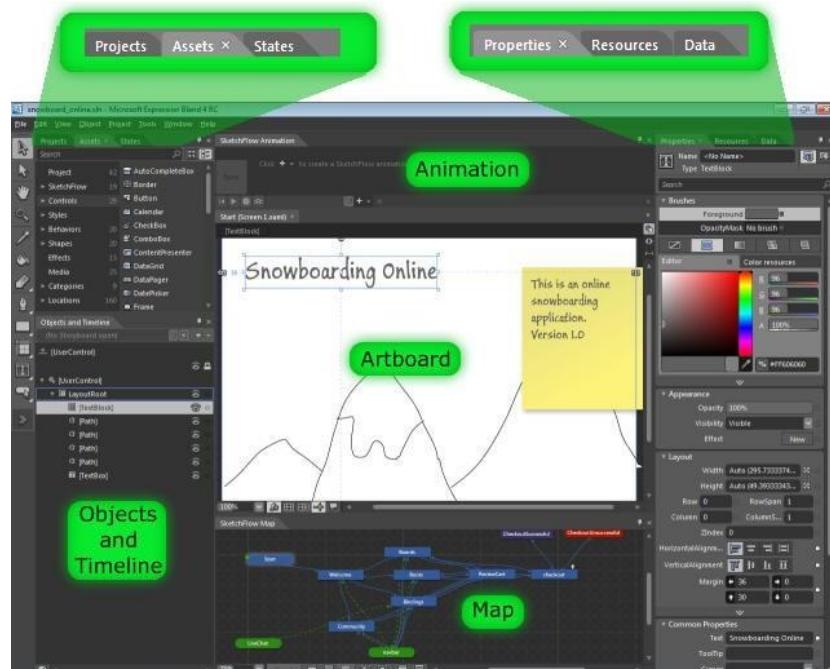
распространены. Однако они имеют определенные недостатки: отсутствие просмотра возможных действий пользователя (их можно реализовать в виде набора изображений, но это крайне трудоемко); отсутствует представление о переходах между формами (и способы перехода – анимированное переключение и др.); отсутствует возможность «щупать».

Интерактивные прототипы избавлены от таких недостатков, однако имеют собственную специфику: как правило разработка подобных прототипов более трудоемка и требует большего количества знаний из области программирования. Зато на выходе – полностью осозаемый продукт, который можно использовать для утверждения заказчиком или для проведения маркетинговых исследований (на предмет удобства использования целевой аудиторией). Интерактивные прототипы как правило компилируются либо в отдельное приложение, либо в набор интерактивных HTML страниц.

Прототипирование при помощи SketchFlow

Программный продукт SketchFlow представляет собой средство визуального проектирования интерактивных интерфейсов при помощи набора средств Blend и технологии .NET (WPF/Silverlight). Рассмотрим процесс создания простейшего прототипа при помощи SketchFlow.

Общий интерфейс программы выглядит следующим образом:



Основной элемент прототипирования – область «Мап» - карта взаимодействия экранов. Экраны можно связывать между собой и организовывать переходы между ними посредством нажатия на кнопку и пр. Также существует возможность создания т.н. «композиционных» экранов – набора элементов, которые повторяются на множестве экранов (как шапка на сайте).

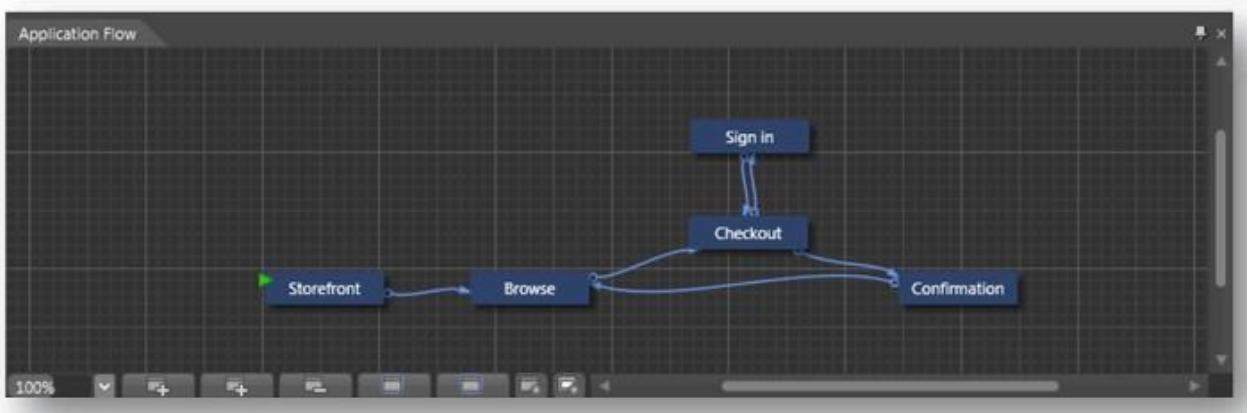
В левом верхнем углу находятся элементы, которые можно перетягивать и использовать при прототипировании экранов. Это те же элементы, что и используются в WPF – кнопки, ссылки, текст, поля ввода и др. Существуют также развитые элементы интерфейса – такие как диалог, который содержит три кнопки, текст и заголовок. Там же есть генерируемый «проект» прототипирования, который очень сильно напоминает проект в Visual Studio.

В левом нижнем углу показаны объекты на экране.

В правой части экрана размещаются свойства выделенных в данный момент объектов.

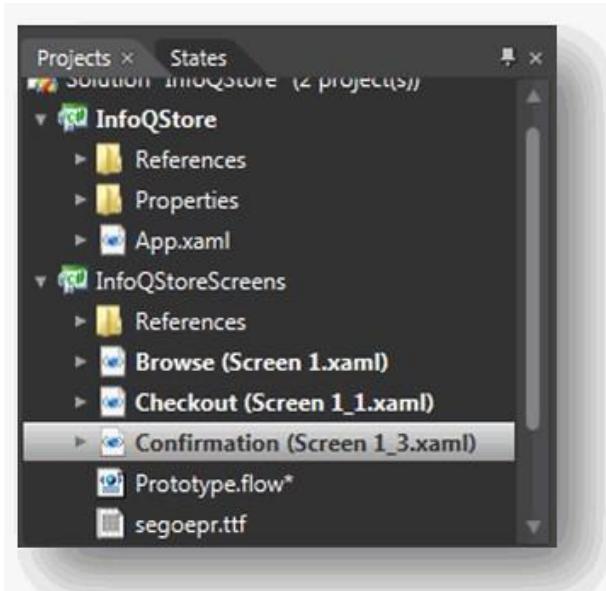
В верхней части экрана показываются добавленные на экран анимации. По центру – область прототипирования. Здесь отображается экран в том виде, в котором он будет отображаться пользователю.

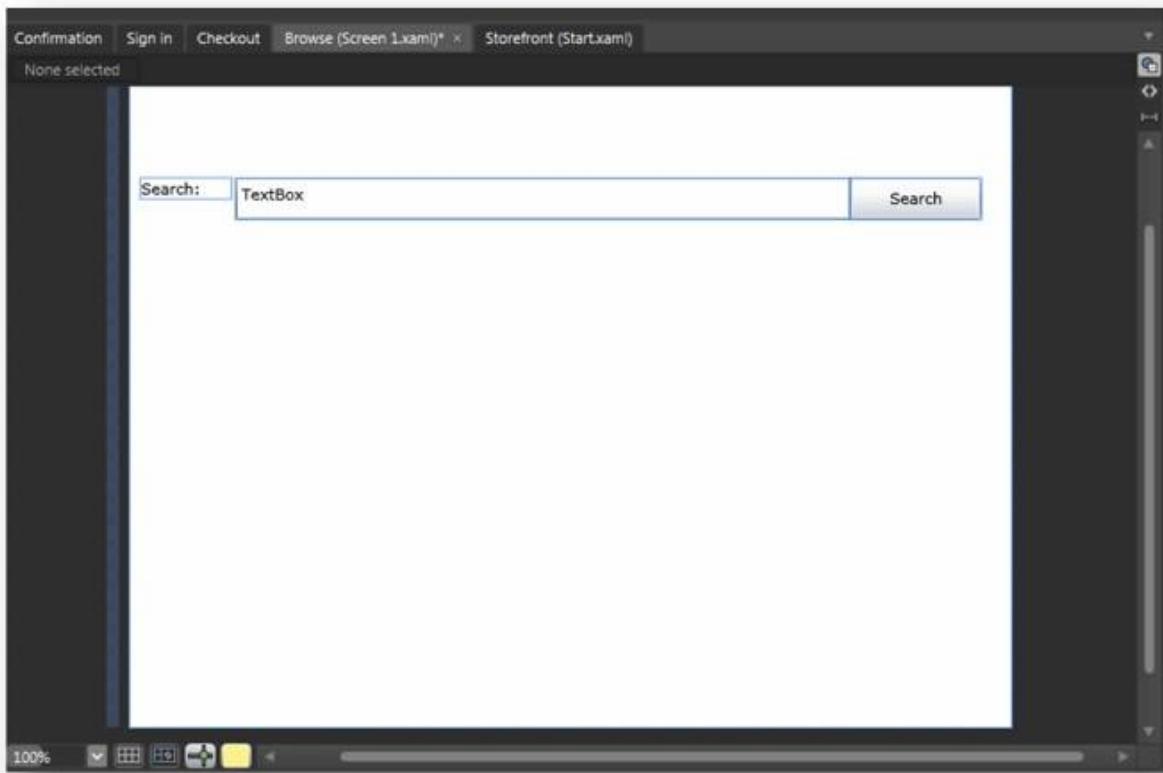
Начинать прототипирование следует с определения экранов и переходов между ними:



Создание этих переходов сгенерирует следующую структуру проекта:

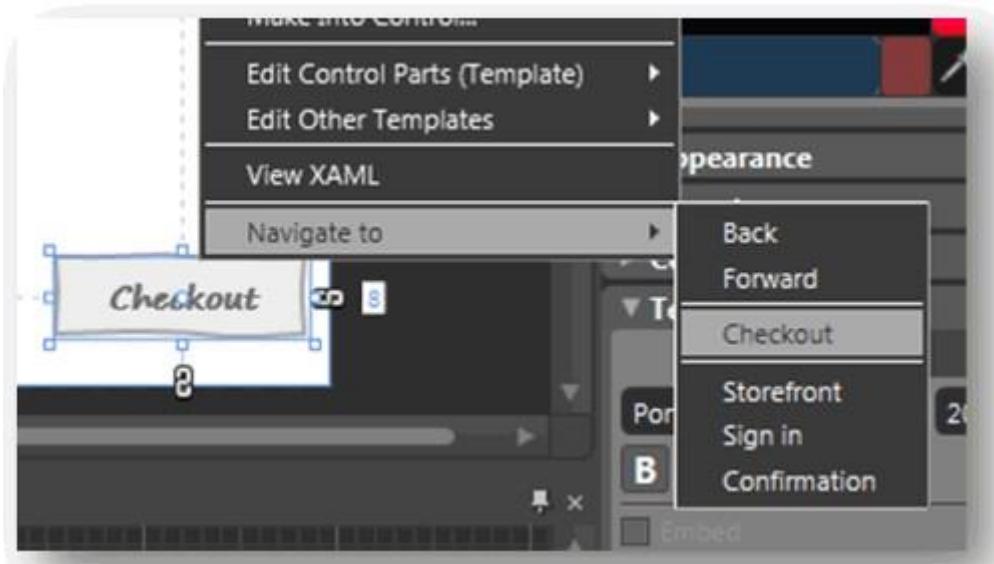
После выполнения данных действий можно непосредственно открывать экраны и начать их редактирование:



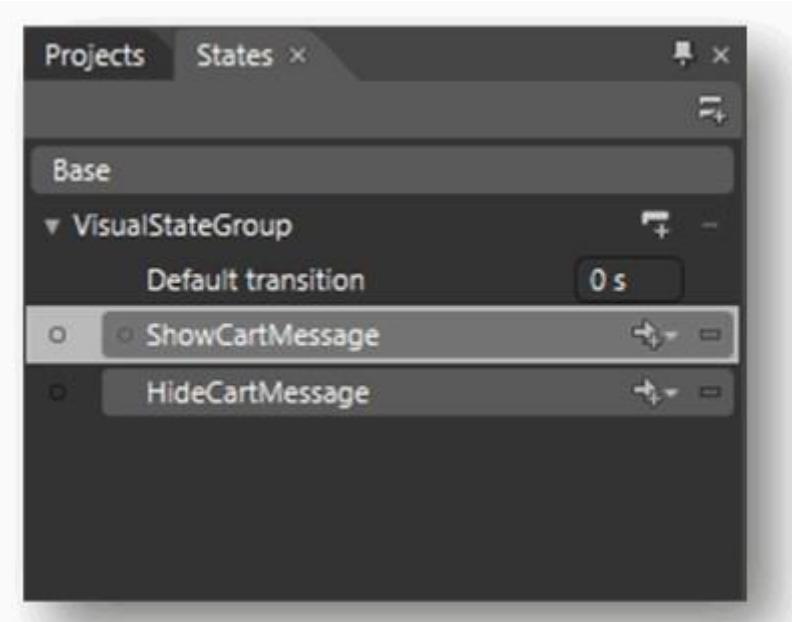


Следует отметить, что SketchFlow как правило использует собственный стиль элементов (Sketchy), который напоминает нарисованные от руки элементы. Это удобно, поскольку в таком случае у заказчика не будет мыслей о том, что данный вид проекта – оконченный. Он будет понимать, что это прототип, и не будет предлагать «сделать эту кнопку на 2 пикселя меньше» и другое; внимание будет концентрироваться на концепции пользовательского интерфейса.

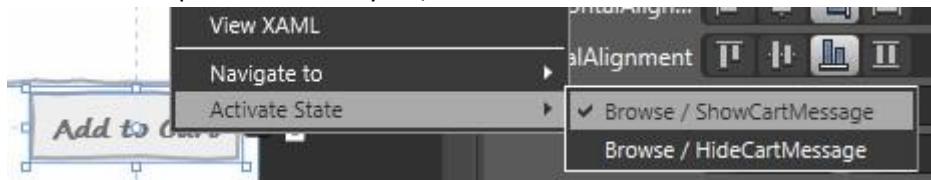
После набрасывания элементов управления на экраны необходимо завести переходы между экранами. Для этого на релевантных элементах необходимо нажать правой кнопкой мыши и выбрать соответствующее меню:



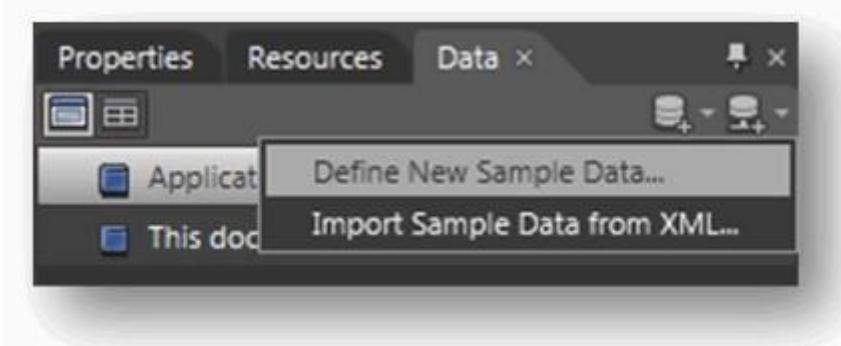
В случае необходимости отображения дополнительных диалоговых окон без смены экрана можно воспользоваться механизмом состояний в Expression Blend. Состояния определяют вид одного и того же экрана в различных случаях.

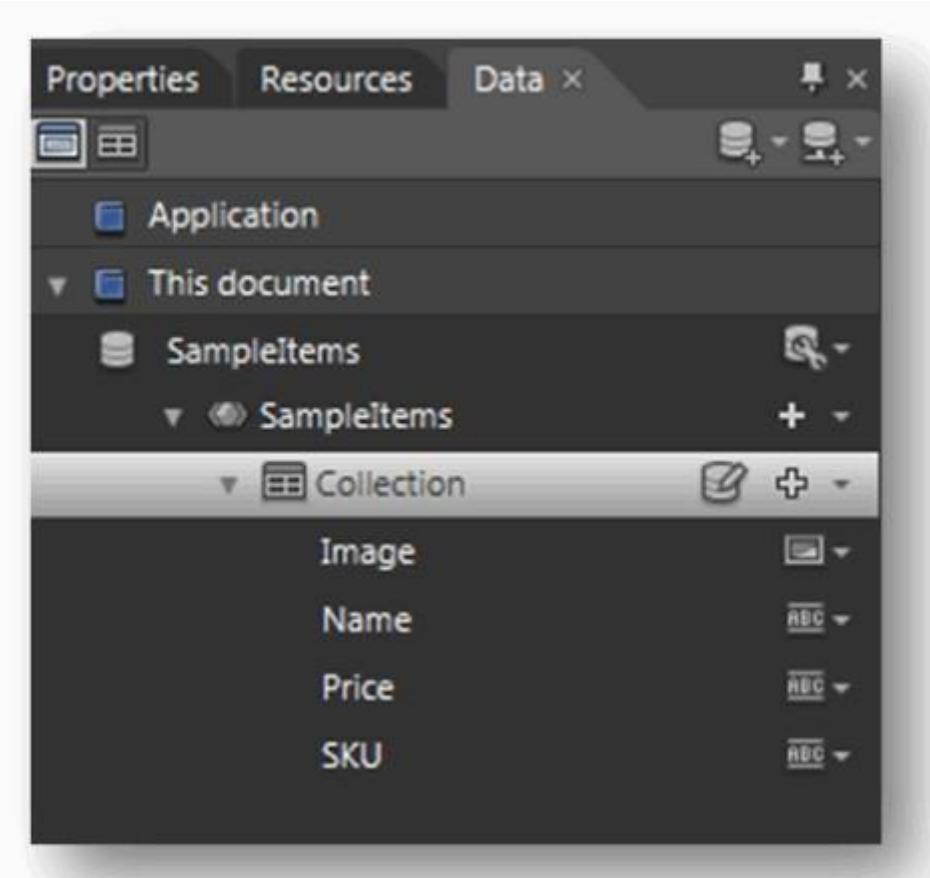


Изменение состояния происходит тем же образом, как и навигация, с помощью правой кнопки мыши и выбором соответствующего контекстного меню:



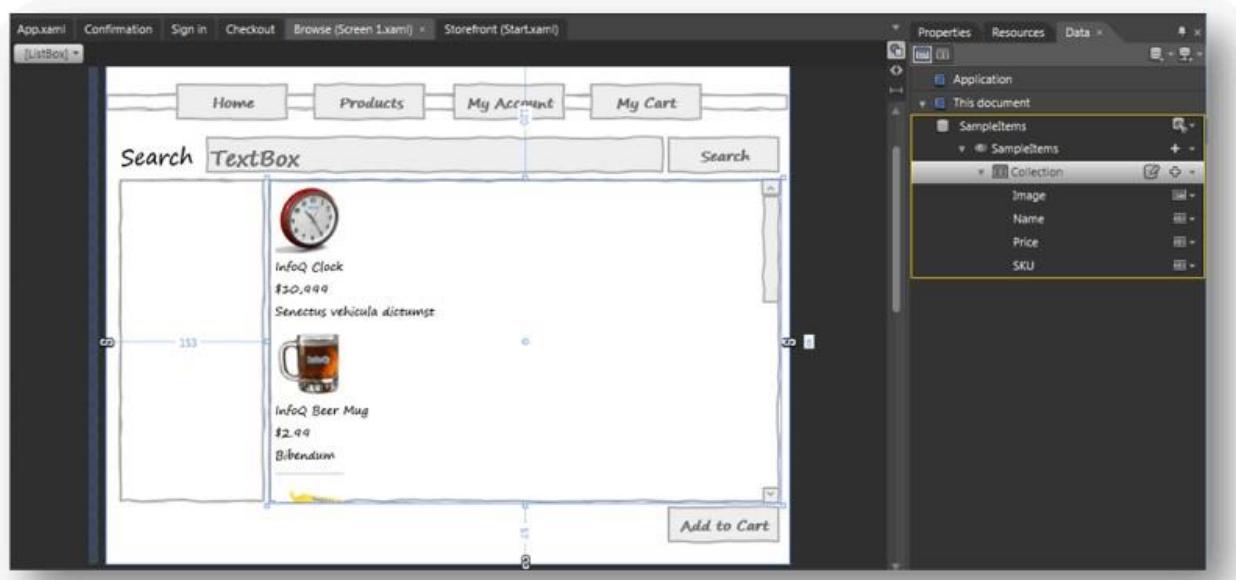
Для определения тестовых данных для заполнения элементов экрана можно воспользоваться кнопкой создания тестовых данных:





Как видим, можно также подключать собственные источники данных либо вбивать данные самостоятельно.

Далее для заполнения соответствующих элементов представления достаточно просто перебросить данные из окна данных на экран в элемент управления, и при запуске можно будет увидеть заполненный элемент управления:



Естественно, доступны и более сложные сценарии заполнения данными.

Ход работы.

1. Создать проект в SketchFlow (WPF Application).
2. Создать карту экранов и связать их между собой. Создать хотя бы 1 композиционный экран (например для нижней части окна – информация о разработчике).
3. Заполнить все экраны и реализовать переходы между ними.

4. Заполнить хотя бы 1 экран тестовыми данными и реализовать некоторые возможности сложного взаимодействия (нажатия на кнопки, заполнение текстовых полей и др.).

ЛАБОРАТОРНАЯ РАБОТА №6

Диаграммы UML. Диаграммы классов. Концептуальная модель системы

Задание.

1. Ознакомиться с краткими теоретическими сведениями.
2. Нарисовать диаграмму классов для реализованной части системы.
3. Разработать основные классы и структуру базы данных системы. Классы данных должны реализовать шаблон Active Record для взаимодействия с базой данных.
4. Подготовить и оформить отчет о выполнении лабораторной работы. Представленный отчет должен содержать диаграмму классов системы, исходные коды классов системы а также изображение структуры базы данных.

Краткие теоретические сведения.

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Представление классов

Класс – это основной строительный блок ПС. Это понятие присутствует и в ОО языках программирования, то есть между классами UML и программными классами есть соответствие, являющееся основой для автоматической генерации программных кодов или для выполнения реинжиниринга. Каждый класс имеет название, атрибуты и операции. Класс на диаграмме показывается в виде прямоугольника, разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия операций – услуг, предоставляемых объектами этого класса.



Изображение класса в нотации UML

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

Для каждого атрибута класса можно задать видимость (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов:

- Открытый (public) – атрибут виден для любого другого класса (объекта);
- Защищенный (protected) – атрибут виден для потомков данного класса;
- Закрытый (private) – атрибут не виден внешними классами (объектами) и может использоваться только объектом, его содержащим.

Последнее значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части ПС.

Класс содержит объявления операций, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Каждая операция имеет сигнатуру, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу. Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в ПС.

Отношения

На диаграммах классов обычно показываются ассоциации и обобщения.

Каждая ассоциация несет информацию о связях между объектами внутри ПС. Наиболее часто используются бинарные ассоциации, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи. Помимо названия, ассоциация может иметь такую характеристику, как множественность. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации (полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль). Связаны между собой могут быть и объекты одного класса, поэтому ассоциация может связывать класс с самим собой. Например, для класса «Житель города» можно ввести ассоциацию «Соседство», которая позволит находить всех соседей конкретного жителя.



Ассоциация «включает» показывает, что набор может включать несколько различных товаров. В данном случае направленная ассоциация позволяет найти все виды товаров, входящие в набор, но не дает ответа на вопрос, входит ли товар данного вида в какой-либо набор.

Ассоциация сама может обладать свойствами класса, то есть, иметь атрибуты и операции. В этом случае она называется класс-ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса.

Ассоциация — наиболее общий вид связи между двумя классами системы. Как правило, она отображает использование одного класса другим посредством некоторого свойства или поля.

Обобщение(наследование) на диаграммах классов используется, чтобы показать связь между классом-родителем и классом-потомком. Оно вводится на диаграмму, когда возникает разновидность какого-либо класса, а также в тех случаях, когда в системе обнаруживаются несколько классов, обладающих сходным поведением (в этом случае общие элементы поведения выносятся на более высокий уровень, образуя класс-родитель).



Наследуются атрибуты и операции

Агрегацией обозначается отношение «has-a», когда объекты одного класса входят в объект другого класса. Типичным примером такого отношения являются списки объектов. В данном случае список будет выступать агрегатом, а объекты, входящие в список, агрегируемыми элементами.



Агрегация

Композицией обозначается отношение «owns-a». По своей сути оно напоминает агрегацию, однако обозначает более тесную связь между агрегатом и агрегируемыми элементами. Примером композиции может служить связь между машиной и карбюратором: машина не будет функционировать без карбюратора (потому отношение композиции). Список, в свою очередь, не теряет своих функций без отдельных элементов списка (потому отношение агрегации).



Композиция

Применение диаграмм классов

Диаграммы классов создаются при логическом моделировании ПС и служат для следующих целей:

1. Для моделирования данных. Анализ предметной области позволяет выявить основные характерные для нее сущности и связи между ними. Это удобно моделируется с помощью диаграмм классов. Эти диаграммы являются основой для построения концептуальной схемы базы данных.
2. Для представления архитектуры ПС. Можно выделить архитектурно значимые классы и показать их на диаграммах, описывающих архитектуру ПС.
3. Для моделирования навигации экранов. На таких диаграммах показываются пограничные классы и их логическая взаимосвязь. Информационные поля моделируются как атрибуты классов, а управляющие кнопки – как операции и отношения.
4. Для моделирования логики программных компонент.
5. Для моделирования логики обработки данных.

Логическая структура базы данных

Различают две модели базы данных — логическую и физическую. Физическая модель базы данных представляет собор набор бинарных данных в виде файлов, структурированных и сгруппированных согласно назначению (сегменты, экстенты и др.), используемая для быстрого и эффективного получения информации с жесткого диска, а также для компактного хранения и размещения данных на жестком диске.

Логическая модель базы данных представляет собой структуру таблиц, представлений, индексов и других логических элементов базы данных, позволяющих собственно программирование и использование базы данных.

Процесс создания логической модели базы данных носит название проектирования базы данных (*database design*). Проектирование происходит в тесной связи с проработкой архитектуры программной системы, поскольку база данных создается для хранения данных, получаемых из программных классов.

Соответственно можно различить несколько подходов к связыванию программных классов и таблиц: одна таблица — один класс, одна таблица — несколько классов, один класс — несколько таблиц. В зависимости от выбранного подхода, будет либо усложняться (и замедляться) работа с базой данных при добавлении данных, либо получение данных из БД и парсинг ее в соответствующие классы.

С другой стороны, если программные классы представляют собой сущности проектируемой системы (элементы предметной области), то таблицы отображают их техническую реализацию и способ хранения и связи.

Основным руководством при проектировании таблиц являются т. н. нормальные формы базы данных.

Нормальные формы

Нормальная форма — свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности, потенциально приводящей к логически ошибочным результатам

выборки или изменения данных. Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение.

Процесс преобразования отношений базы данных (БД) к виду, отвечающему нормальным формам, называется нормализацией. Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объема базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации. Как отмечает К. Дейт, общее назначение процесса нормализации заключается в следующем:

- исключение некоторых типов избыточности;
- устранение некоторых аномалий обновления;
- разработка проекта базы данных, который является достаточно «качественным» представлением реального мира, интуитивно понятен и может служить хорошей основой для последующего расширения;
- упрощение процедуры применения необходимых ограничений целостности. Устранение избыточности производится, как правило, за счет декомпозиции отношений таким образом, чтобы в каждом отношении хранились только первичные факты (то есть факты, не выводимые из других хранимых фактов).

Переменная отношения находится в первой нормальной форме (1НФ) тогда и только тогда, когда в любом допустимом значении отношения каждый его кортеж содержит только одно значение для каждого из атрибутов.

Переменная отношения находится во второй нормальной форме тогда и только тогда, когда она находится в первой нормальной форме, и каждый неключевой атрибут неприводимо (функционально полно) зависит от ее потенциального ключа.

Переменная отношения находится в третьей нормальной форме тогда и только тогда, когда она находится во второй нормальной форме, и отсутствуют транзитивные функциональные зависимости неключевых атрибутов от ключевых.

Переменная отношения находится в нормальной форме Бойса — Кодда (иначе — в усиленной третьей нормальной форме) тогда и только тогда, когда каждая ее нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ.

Проектирование БД

Для проектирования базы данных можно воспользоваться Schema View в соответствующих средствах работы с СУБД (Microsoft Sql Server Management Studio, PL/SQL Developer и другие) либо встроенное средство Microsoft Office Access.

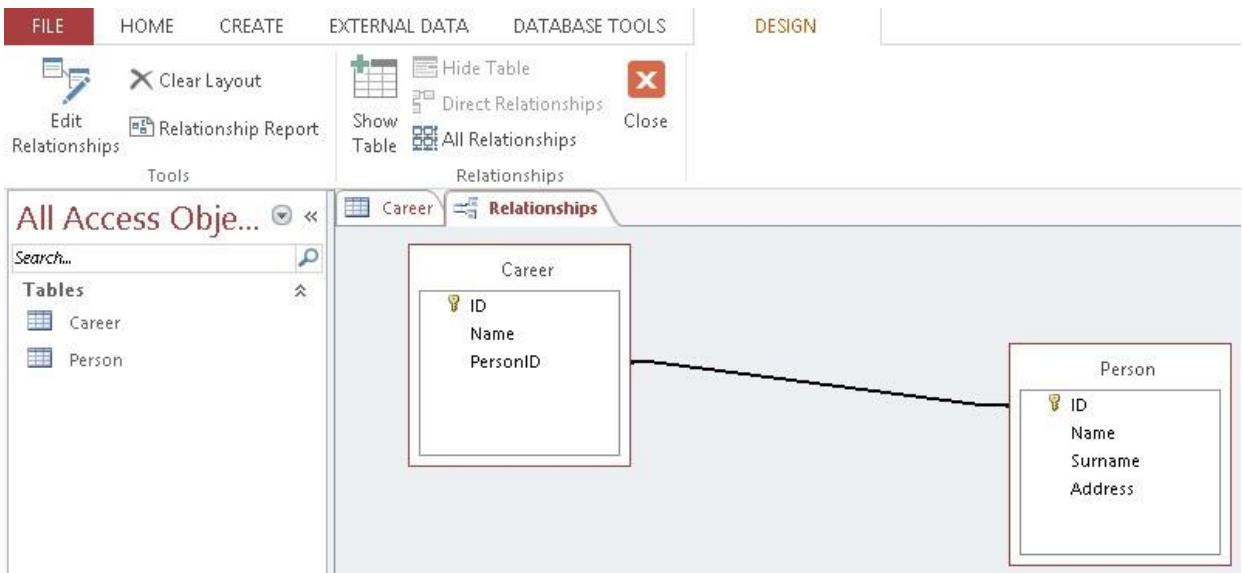
Для создания таблиц необходимо нажать кнопку «Создать»(CREATE):

| FILE | HOME | CREATE | EXTERNAL DATA | DATABASE TOOLS | DESIGN |
|--|--|--|---|--|--------|
| View Primary Key Test Validation Rules Views | Insert Rows Delete Rows Modify Lookups | Property Sheet Indexes Show/Hide | Create Data Macros Field, Record & Table Events | Rename/ Delete Macro Relationships Object Dependencies Relationships | |

All Access Obj...

| Field Name | Data Type |
|------------|------------|
| ID | AutoNumber |
| Name | Short Text |
| PersonID | Number |
| | |

После заполнения полей и их типов, необходимо назначить первичный ключ. После чего можно выставить связи между таблицами в окне «Связи»(Relationships):



Вопросы для самопроверки

- Что представляет собой диаграмма классов UML ?
- Чем отличается связи типа агрегации от связей композиции на диаграммах классов ?
- Что представляют собой нормальные формы баз данных ?
- Что такая физическая модель базы данных ? Логическая ?
- Какая взаимосвязь между таблицами БД и программными классами ?

Лабораторная Работа № 7

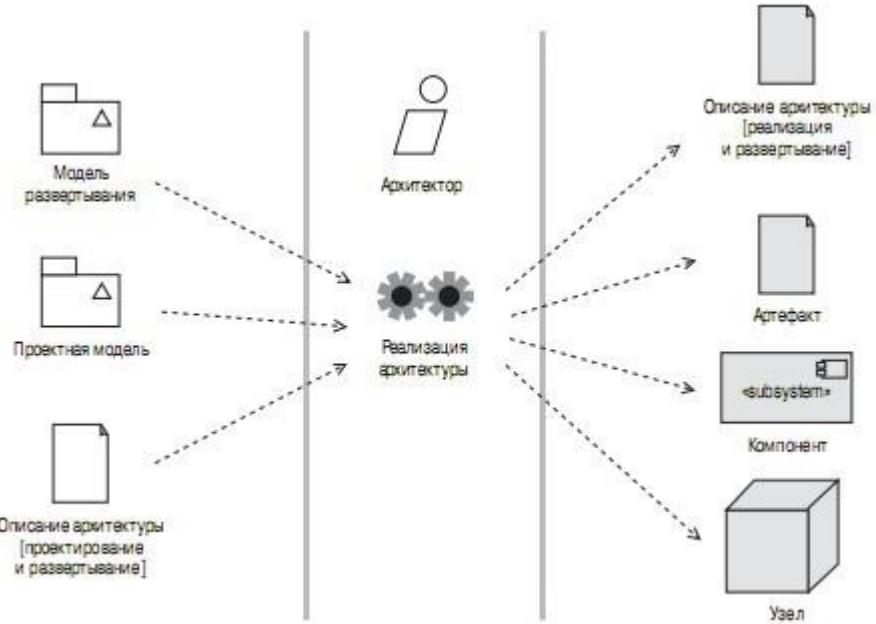
Диаграмма развертывания. Создание графического интерфейса.

Задание

- Ознакомиться с краткими теоретическими сведениями.
- Изобразить диаграмму развертывания для своей системы.
- Разработать визуальный интерфейс при помощи технологии WPF согласно ранее спроектированному. Не менее ¾ всех форм должны быть разработаны полностью (визуально) — без отработки логики.
- Оформить отчет о выполненной работе.

Краткие теоретические сведения

Диаграмма развертывания (Deployment Diagram)



Диаграммы развертывания представляют физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения.

Главными элементами диаграммы являются узлы, связанные информационными путями. Узел (node) – это то, что может содержать программное обеспечение. Узлы бывают двух типов. Устройство (device) – это физическое оборудование: компьютер или устройство, связанное с системой. Среда выполнения (execution environment) – это программное обеспечение, которое само может включать другое программное обеспечение, например операционную систему или процесс-контейнер (например, веб-сервер).

Между узлами могут стоять связи, обычно изображаемые в виде прямой линии. Как и на других диаграммах, у связей могут быть атрибуты множественности (для показания, например, подключения 2x и более клиентов к одному серверу) и название. В названии, как правило, содержится способ связи между двумя узлами — это может быть название протокола (http, IPC) либо используемая технология для обеспечения взаимодействия узлов (.net Remoting, WCF).

Узлы могут содержать артефакты (artifacts), которые являются физическим олицетворением программного обеспечения; обычно это файлы. Такими файлами могут быть исполняемые файлы (такие как файлы .exe, двоичные файлы, файлы DLL, файлы JAR, сборки или сценарии) или файлы данных, конфигурационные файлы, HTML-документы и т. д. Перечень артефактов внутри узла указывает на то, что на данном узле артефакт разворачивается в запускаемую систему.

Артефакты можно изображать в виде прямоугольников классов или перечислять их имена внутри узла. Если вы показываете эти элементы в виде прямоугольников классов, то можете добавить значок документа или ключевое слово «artifact». Можно сопровождать узлы или артефакты значениями в виде меток, чтобы указать различную интересную информацию об узле, например поставщика, операционную систему, местоположение – в общем, все, что придет вам в голову.

Часто у вас будет множество физических узлов для решения одной и той же логической задачи. Можно отобразить этот факт, нарисовав множество прямоугольников узлов или поставив число в виде значения-метки.

Артефакты часто являются реализацией компонентов. Это можно показать, задав значения-метки внутри прямоугольников артефактов.

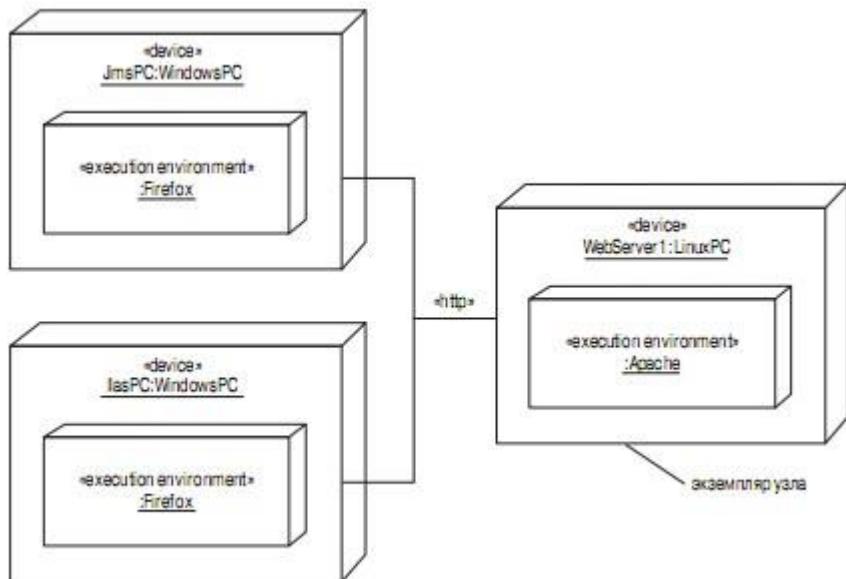
Основные виды артефактов:

- исходные файлы;
- исполняемые файлы;
- сценарии;
- таблицы баз данных;
- документы;
- результаты процесса разработки, UML-модели.

Можно также детализировать артефакты, входящие в узел; например, дополнительно внутри развертываемого файла указать, какие туда входят компоненты либо классы. Такая детализация, как правило, не имеет смысла на диаграммах развертывания, поскольку может смещать фокус внимания от модели развертывания программного обеспечения к его внутреннему устройству, однако иногда может быть полезной. При этом, возможно устанавливать связи между компонентами/классами в пределах разных узлов.

Диаграммы развертываний различают двух видов: описательные и экземплярные. На диаграммах описательной формы указываются узлы, артефакты и связи между узлами без указания конкретного оборудования либо программного обеспечения, необходимого для развертывания. Такой вид диаграмм полезен на ранних этапах разработки для понимания, какие в принципе физические устройства необходимы для функционирования системы или для описания процесса развертывания в общем ключе.

Диаграммы экземплярной формы несут в себе экземпляры оборудования, артефактов и связей между ними. Под экземплярами понимаются конкретные элементы — ПК с соответствующим набором характеристик и установленным ПО; вполне может быть, в пределах одной организации это может быть какойто конкретный узел (например, ПК тестировщика Василия). Диаграммы экземплярной формы разрабатываются на завершальных стадиях разработки ПО — когда уже известны и сформулированы требования к программному комплексу, оборудование закуплено и все готово к развертыванию. Диаграммы такой формы представляют собой скорее план развертывания в графическом виде, чем модель развертывания.



Windows Presentation Foundation (WPF)

Windows Presentation Foundation — современный фреймворк пользовательского интерфейса для создания красивых приложений в среде .NET. На данный момент этот фреймворк может использоваться для создания десктоп-приложений (WPF), мобильных приложений (WP7, WP8) и веб приложений (Silverlight).

К созданию нового движка пользовательского интерфейса команду Microsoft подтолкнула масса проблем с существующим способом создания приложений (WinForms):

- Огромные объемы устаревшего (legacy) кода - отрисовка в WinForms происходит посредством использования родных функций Windows (WinApi) через GDI/GDI+ и библиотеку User32; помимо того, что библиотека присутствует со времен Windows 3.1, это накладывает существенные ограничения по возможностям движка;
- Устаревший подход к организации пользовательского интерфейса — интерфейс строится без оглядки на используемое разрешение экрана и монитора (dpi); компоненты формы как правило имеют фиксированные размеры, поскольку разработка форм поддерживающих увеличение очень утомительна и не всегда приносит желаемый результат;
- Программная прорисовка — не используются ресурсы видеокарты;
- Медлительность;
- Интерфейс определяется в терминах программного кода, что ограничивает вовлеченность дизайнеров в процесс проектирования визуального интерфейса;
- Другие.

Фреймворк WPF призван улучшить способ разработки визуального представления приложений следующим образом:

- Использует DirectX API для прорисовки окна и его элементов — автоматически поддерживается аппаратное ускорение с откатом на программное в некоторых случаях;
- Не завязан на программный код — дизайнеры могут использовать такие средства визуального проектирования интерфейса как Expression Blend и передавать полученный исходный код разработчикам для дальнейшего внедрения и реализации;
- Упрощена работа с анимацией — анимация является неотъемлемой частью платформы;
- Упрощена работа с примитивами — 2d и 3d графикой;

- Доступна возможность изменения внешнего вида абсолютного любого элемента окна;
- Элементы автоматически поддерживают разные плотности экранов (dpi) а также могут изменять свой размер динамически в зависимости от настройки визуального представления при изменении размеров окна и сохранять приличный внешний вид;
- Окно не отрисовывается постоянно, как в WinForms (по событию WM_PAINT), а лишь когда есть изменения в визуальном представлении — значительно уменьшает объем потребляемых ресурсов и ускоряет отображение; - Множество других.

XAML

Для разработки визуального интерфейса в WPF используется специальный синтаксис — eXtensible Application Markup Language (Расширяемый язык разметки приложений). Он представляет собой модифицированный XML. Рассмотрим простейший код окна WPF:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

    <!-- Add button to window -->
    <Button Name="button">Click Me!</Button>

</Window>
```

Как видно, есть корневой элемент `Window`, описывающий окно. При помощи атрибутов `Title`, `Width` и `Height` можно выставить заголовок, ширину и высоту окна соответственно. Список доступных атрибутов можно посмотреть при помощи Intellisense в среде разработки Microsoft Visual Studio; среди прочих можно задать минимальный и максимальный размер окна, его исходное размещение на экране.

Комментарии в XAML имеют тот же синтаксис, что и в XML - `<!-- Комментарий -->`.

Другие элементы представляют собой теги XML с соответствующим названием. Например, для кнопки определен тег `Button` и свой набор свойств. Важно отметить, что в WPF все элементы тем или иным образом входят как дочерние в другие элементы. Например, если необходимо отобразить картинку на кнопке вместо текста «Click me!» достаточно вставить внутрь ее объект `Image`:

```
<Button Name="button">
    <Image Source="fun.png" />
</Button>
```

В XAML принято описывать лишь визуальную часть интерфейса. Вся логика взаимодействия, точно также как и в WinForms, может быть описана в codebehind файлах. У большинства элементов имеется известных из WinForms набор событий (event) для отработки тех или иных действий. Например, для кнопки событие `Click` (аналог `OnClick` в WinForms):

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"      Title="Window
with Button"
    Width="250" Height="100">

    <!-- Add button to window -->
```

```

<Button Name="button" Click="button_Click">Click Me!</Button>

</Window> using

System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window
    {
        public
        AWindow()
        {
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}

```

Для взаимодействия с элементами представления по-прежнему можно использовать их имена (в данном примере название кнопки — button). Однако, необходимо отметить, что имена являются опциональными (в отличие от Windows Forms) и как правило не выставляются, поскольку для взаимодействия с элементами визуального интерфейса используется механизм привязок (binding), который будет рассмотрен в следующих лабораторных работах.

Способы организации элементов интерфейса в окне WPF

В предыдущем примере видно, что кнопка занимает весь размер окна, хотя нигде это не указано. Это происходит потому, что кнопка является единственным дочерним элементом окна, и ей не указаны никакие ограничения по размерам — она автоматически расширяется для заполнения объема пространства родительского элемента.

В случае, если попытаться добавить две кнопки в окно — возникнет ошибка. Это связано с тем, что в WPF все элементы делятся на три группы: элементы, которые могут содержать множество других элементов (контейнеры); элементы, которые могут содержать один другой элемент (content control); элементы, не содержащие дочерних элементов.

Для организации элементов на экране используются т.н. контейнеры. Вот перечень основных контейнеров: StackPanel, WrapPanel, DockPanel, Grid, Canvas.

Canvas

Канва (Canvas) — наиболее простой элемент организации элементов WPF, представляет собой доску для рисования. Каждый элемент, добавляемый на канву, определяет свое положение относительно левого верхнего угла канвы при помощи координат X и Y. Ввиду своей простоты является наиболее быстродействующим элементом организации контента в приложениях WPF и наиболее гибким, однако не умеет автоматически подстраиваться под изменение размеров окна (ввиду абсолютного позиционирования). Пример:

```

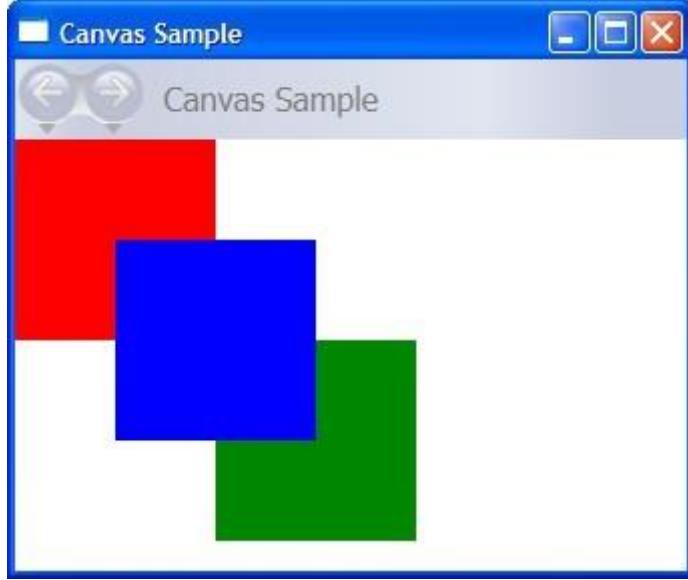
<Page WindowTitle="Canvas Sample"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Canvas Height="400" Width="400">

```

```

<Canvas Height="100" Width="100" Top="0" Left="0" Background="Red"/>
<Canvas Height="100" Width="100" Top="100" Left="100" Background="Green"/>
<Canvas Height="100" Width="100" Top="50" Left="50" Background="Blue"/>
</Canvas>
</Page>

```



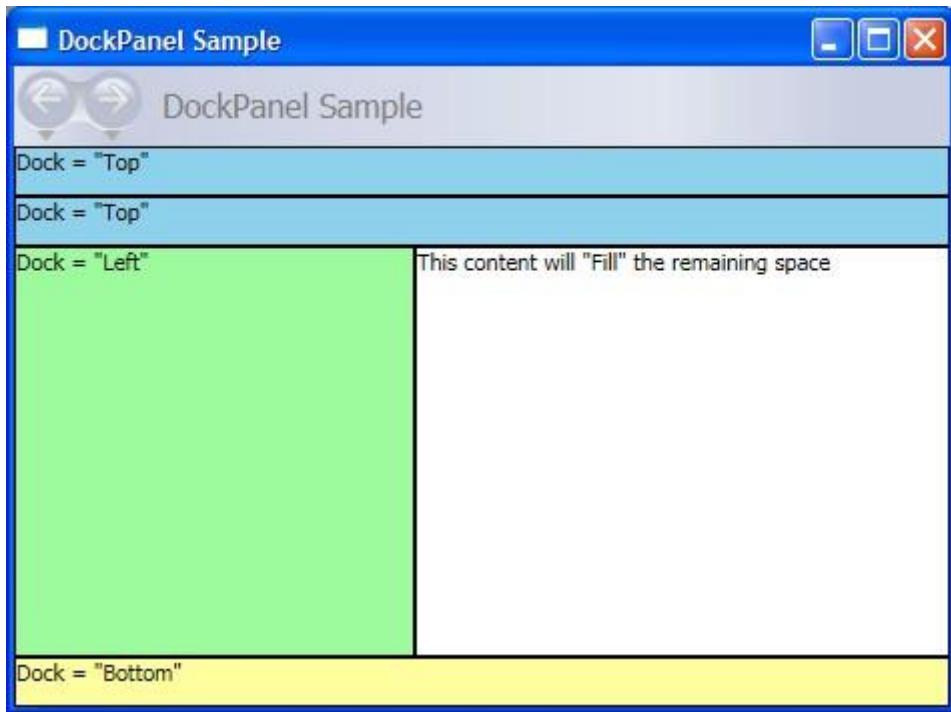
DockPanel

Панель стыковки (DockPanel) может быть известна WinForms разработчикам — позволяет «прикреплять» элементы управления к одному из краев родительского элемента (аналог свойства Dock в WinForms приложениях). Внутренние элементы занимают либо лишь необходимую часть пространства, либо последний элемент занимает все оставшее пространство. Пример:

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="DockPanel Sample">
    <DockPanel LastChildFill="True">
        <Border Height="25" Background="SkyBlue" BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Top">
            <TextBlock Foreground="Black">Dock = "Top"</TextBlock>
        </Border>
        <Border Height="25" Background="SkyBlue" BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Top">
            <TextBlock Foreground="Black">Dock = "Top"</TextBlock>
        </Border>
        <Border Height="25" Background="LemonChiffon" BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Bottom">
            <TextBlock Foreground="Black">Dock = "Bottom"</TextBlock>
        </Border>
        <Border Width="200" Background="PaleGreen" BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Left">
            <TextBlock Foreground="Black">Dock = "Left"</TextBlock>
        </Border>
        <Border Background="White" BorderBrush="Black" BorderThickness="1">
            <TextBlock Foreground="Black">This content will "Fill" the remaining space</TextBlock>
        </Border>
    </DockPanel>
</Page>

```



Grid

Сетка (Grid) позволяет позиционировать элементы при помощи таблицы, при этом с возможностью элементам занимать одновременно несколько клеток такой сетки. Сетка определяется при помощи строк (Grid.RowDefinitions) и колонок (Grid.ColumnDefinitions), но не обязательно имеет и строки и колонки. Строки и колонки сетки могут иметь различный размер, при этом можно указать сетке распределять размер между колонками равномерно (занимать оставшийся размер родительского контейнера).

Дочерние элементы указывают номер строки и колонки, в которой хотят находиться. Пример:

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      WindowTitle="Grid Run Dialog Sample"
      WindowWidth="425"
      WindowHeight="225">
    <Grid Background="#DCDCDC"
          Width="425"
          Height="165"
          HorizontalAlignment="Left"
          VerticalAlignment="Top"
          ShowGridLines="True">
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto" />
          <ColumnDefinition Width="*" />
          <ColumnDefinition Width="*"/>           <ColumnDefinition
Width="*"/>
          <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
          <RowDefinition Height="*"/>
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
```

```

<Image Grid.Column="0" Grid.Row="0" Source="RunIcon.png" />
<TextBlock Grid.Column="1" Grid.ColumnSpan="4" Grid.Row="0" TextWrapping="Wrap"> Type the name of
a program, folder, document, or
Internet resource, and Windows will open it for you.
</TextBlock>
<TextBlock Grid.Column="0" Grid.Row="1">Open:</TextBlock>
<TextBox Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="5" />
<Button Margin="10, 0, 10, 15" Grid.Row="3" Grid.Column="2">OK</Button>
<Button Margin="10, 0, 10, 15" Grid.Row="3" Grid.Column="3">Cancel</Button>
<Button Margin="10, 0, 10, 15" Grid.Row="3" Grid.Column="4">Browse ...</Button> </Grid>
</Page>

```



Обратим внимание: задать ширину колонки или высоту строки можно тремя способами: auto, * или ввести число. Auto означает, что строка будет занимать ровно столько, сколько необходимо ее контенту (дочерним элементам). Звездочка означает использовать оставшееся место равномерно; т. е. если указать четыре колонки с размером *, то они будут иметь одинаковый размер, вычисляемый по формуле: [свободная ширина] / 4. Более того, можно использовать множители вместе с «*», например 3*, или 2*. Это означает, что определенная колонка будет в 3 раза шире, чем колонки размера *, но при этом размер все равно считается исходя из оставшегося места. Числовое значение определяет абсолютную ширину колонки.

Правилом хорошего тона считается задавание размера при помощи оператора «*» либо Auto, поскольку при изменении размера окна контейнер будет автоматически подстраиваться и увеличивать/уменьшать размеры строк/колонок.

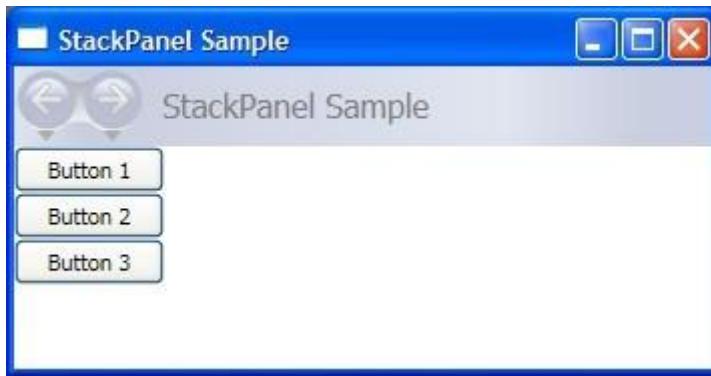
StackPanel

Стековая панель позволяет организовывать элементы плоским списком, один за другим, в любом из двух направлений — вертикально либо горизонтально. При этом, если элементы выходят за границы панели (им необходимо больше места, чем есть в наличии у стековой панели) — они будут обрезаны по доступное пространство. Пример:

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="StackPanel Sample">
    <StackPanel HorizontalAlignment="Left"
                VerticalAlignment="Top">
        <Button>Button 1</Button>
        <Button>Button 2</Button>
        <Button>Button 3</Button>
    </StackPanel>
</Page>

```



WrapPanel

Сверточная панель используется для заполнения свободного пространства элементами последовательно, слева направо с использованием переноса строки, подобно тому как текст заполняет страничку — по окончанию строки текст переходит на следующую. Пример:

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="WrapPanel Sample">
    <Border HorizontalAlignment="Left" VerticalAlignment="Top" BorderBrush="Black"
    BorderThickness="2">
        <WrapPanel Background="LightBlue" Width="200" Height="100">
            <Button Width="200">Button 1</Button>
            <Button>Button 2</Button>
            <Button>Button 3</Button>
            <Button>Button 4</Button>
        </WrapPanel>
    </Border>
</Page>
```



Использование нескольких панелей одновременно

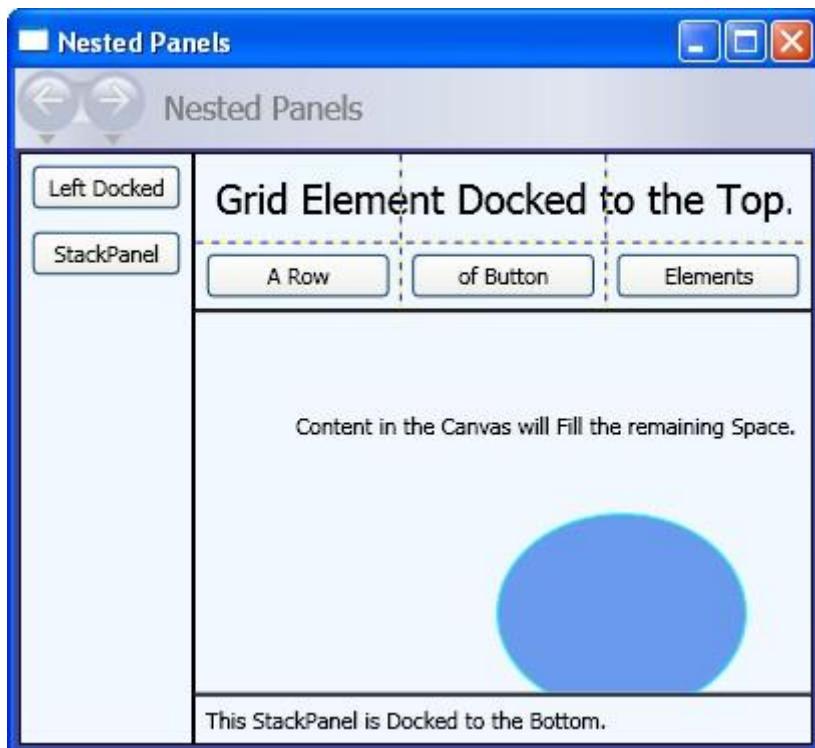
Естественно, панели можно объединять для получения более точного контроля за позиционированием элементов в окне. Часто употребляется связка Grid + StackPanel — стековые панели размещаются в ячейках сетки и содержат в себе сразу несколько элементов, один за другим (например, label + textbox). Пример такого симбиоза:

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="Nested Panels">
    <Border Background="AliceBlue"
    Width="400"
    Height="300"
    BorderBrush="DarkSlateBlue"
    BorderThickness="2"
    HorizontalAlignment="Left"
    VerticalAlignment="Top">
```

```

<DockPanel>
    <Border BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Left">
        <StackPanel>
            <Button Margin="5">Left Docked</Button>
            <Button Margin="5">StackPanel</Button>
        </StackPanel>
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Top">
        <Grid ShowGridLines="True">
            <Grid.RowDefinitions>
                <RowDefinition/>
            <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <TextBlock FontSize="20" Margin="10" Grid.ColumnSpan="3" Grid.Row="0">Grid Element Docked to the Top.</TextBlock>
            <Button Grid.Row="1" Grid.Column="0" Margin="5">A Row</Button>
            <Button Grid.Row="1" Grid.Column="1" Margin="5">of Button</Button>
            <Button Grid.Row="1" Grid.Column="2" Margin="5">Elements</Button>
        </Grid>
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" DockPanel.Dock="Bottom">
        <StackPanel Orientation="Horizontal">
            <TextBlock Margin="5">This StackPanel is Docked to the Bottom.</TextBlock>
        </StackPanel>
    </Border>
    <Border BorderBrush="Black" BorderThickness="1">
        <Canvas ClipToBounds="True">
            <TextBlock Canvas.Top="50" Canvas.Left="50">
                Content in the Canvas will Fill the remaining Space.
            </TextBlock>
            <Ellipse Height="100" Width="125" Fill="CornflowerBlue" Stroke="Aqua" Canvas.Top="100" Canvas.Left="150"/>
        </Canvas>
    </Border>
</DockPanel>
</Border>
</Page>

```



ОСНОВНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Основные элементы управления в WPF весьма похожи на WinForms:

- Button — кнопка;
- ToggleButton — кнопка с двумя состояниями, определяемых свойством IsChecked;
- TextBlock — легковесный аналог Label;
- Label — подпись;
- ListBox — список элементов, основные свойства — Items, ItemsSource;
- TabPanel — панель с вкладками (табами);
- Image — изображение, устанавливается при помощи свойства Source;
- TextBox — поле для ввода текста;
- Border — граница, рамка;
- Expander — выпадающий элемент;
- ContextMenu, Menu — контекстное меню и меню;
- CheckBox - «галочка»;
- ComboBox — выпадающий список;

Контрольные вопросы

1. Что представляет собой диаграмма развертываний ?
2. Какие бывают виды узлов на диаграмме развертываний ?
3. Какие бывают связи на диаграмме развертываний ?
4. Что представляет собой платформа WPF ?
5. Какие проблемы платформы WinForms решает WPF ?

ЛАБОРАТОРНАЯ РАБОТА № 8

Диаграмма компонентов. События, шаблоны, стили и триггеры WPF.

Задание

1. Ознакомиться с краткими теоретическими сведениями.
2. Разработать диаграмму компонентов для проектируемой системы.
3. Реализовать все формы приложения визуально с использованием шаблонов стилей.
4. Реализовать логику работы одной формы полностью от изначальной загрузки данных из БД до сохранения при помощи событий WPF.
5. Оформить отчет о проделанной работе.

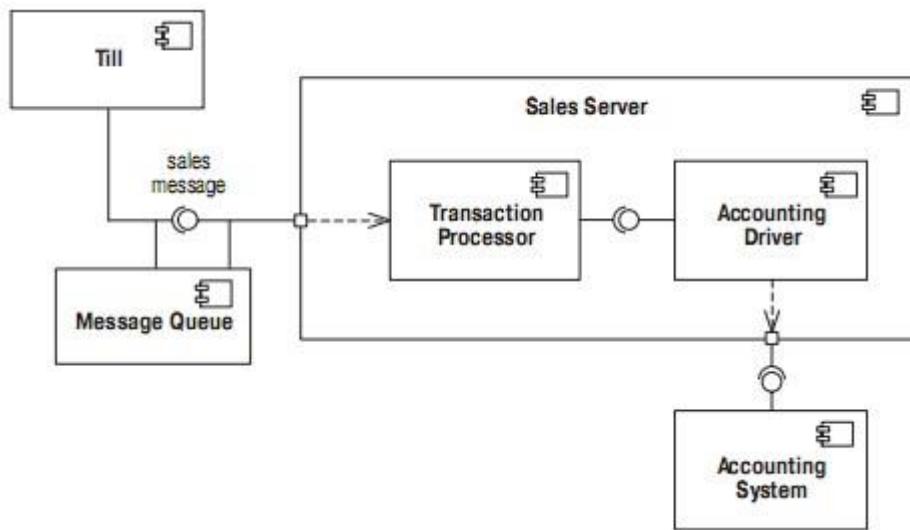
Краткие теоретические сведения

Диаграмма компонентов

Диаграмма компонентов UML является представлением проектируемой системы разбитой на отдельные модули. В зависимости от способа разделения на модули различают три вида диаграмм компонентов:

- Логические
- Физические
- Исполняемые

Наиболее часто используется логическое разбиение на компоненты — в таком случае проектируемая система виртуально представляется как набор самостоятельных, автономных модулей (компонентов), взаимодействующих между собой. Пример такого разбиения приведен на рисунке ниже:

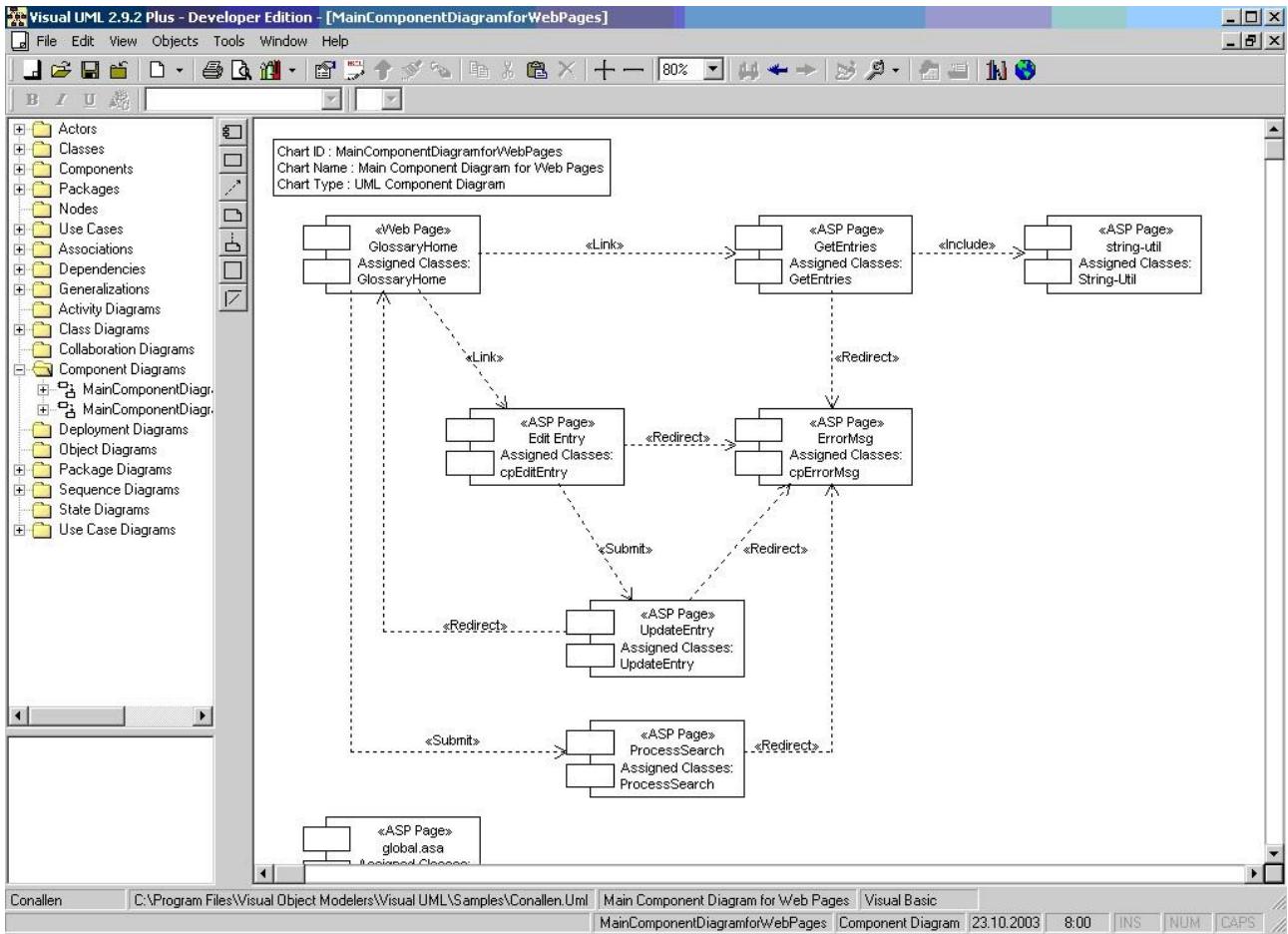


В данном случае система по продаже продуктов использует четыре компонента: кассу, очередь сообщений, сервер продаж и систему учета продаж. Каждый компонент — взаимозаменяемый элемент такой системы, при этом не обязательно находится в пределах одной физической единицы (компьютера) или даже в пределах одного исполняемого процесса. В данном случае компонент — скорее маркетинговая единица, чем техническая, поскольку как правило компонентно-ориентированные системы собираются из компонент под каждого клиента.

Компоненты могут разделяться по физическим единицам — отдельные узлы распределенной системы — набор компьютеров и серверов; на каждом из узлов могут быть установлены различные

исполняемые компоненты. Такой вид диаграмм компонентов устарел и как правило вместо него используется диаграмма развертываний.

На диаграммах компонентов с исполняемым разделением компонентов каждый компонент представляет собой некоторый файл — исполняемые файлы (.exe), файлы исходных кодов, страницы html, базы данных и таблицы и подобное. Пример такой диаграммы приведен ниже:



В данном случае диаграмма походит на диаграмму классов, но на более верхнем уровне — уровне исполняемых файлов или процессов. Такой подход дает другой разрез представления системы, однако как правило не имеет смысла ввиду наличия диаграммы развертываний.

Для полноты обзора диаграммы компонентов приведем цитату Ральфа Джонсона:

«Компоненты – это не технология. Технические специалисты считают их трудными для понимания. Компоненты – это скорее стиль отношения клиентов к программному обеспечению. Они хотят иметь возможность покупать необходимое им программное обеспечение частями, а также иметь возможность обновлять его, как они обновляют свою стереосистему. Они хотят, чтобы новые компоненты работали так же, как и прежние, и обновлять их согласно своим планам, а не по указанию производителей. Они хотят, чтобы системы различных производителей могли работать вместе и были взаимозаменяемыми. Это очень разумные требования. Одна загвоздка: их трудно выполнить. »

Ральф Джонсон (Ralph Johnson),
<http://www.c2.com/cgi/wiki?DoComponentsExist>

Диаграмма компонентов разрабатывается для следующих целей:

- визуализации общей структуры исходного кода программной системы;

- спецификации исполняемого варианта программной системы;
- обеспечения многократного использования отдельных фрагментов программного кода;
- представления концептуальной и физической схем баз данных.

Рекомендации по построению диаграммы компонентов

Разработка диаграммы компонентов предполагает использование информации как о логическом представлении модели системы, так и об особенностях ее физической реализации. До начала разработки необходимо принять решения о выборе вычислительных платформ и операционных систем, на которых предполагается реализовывать систему, а также о выборе конкретных баз данных и языков программирования.

После этого можно приступить к общей структуризации диаграммы компонентов. В первую очередь, необходимо решить, из каких физических частей (файлов) будет состоять программная система. На этом этапе следует обратить внимание на такую реализацию системы, которая обеспечивала бы не только возможность повторного использования кода за счет рациональной декомпозиции компонентов, но и создание объектов только при их необходимости.

Речь идет о том, что общая производительность программной системы существенно зависит от рационального использования вычислительных ресурсов. Для этой цели необходимо большую часть описаний классов, их операций и методов вынести в динамические библиотеки, оставив в исполняемых компонентах только самые необходимые для инициализации программы фрагменты программного кода.

После общей структуризации физического представления системы необходимо дополнить модель интерфейсами и схемами базы данных. При разработке интерфейсов следует обращать внимание на согласование (стыковку) различных частей программной системы. Включение в модель схемы базы данных предполагает спецификацию отдельных таблиц и установление информационных связей между таблицами.

Завершающий этап построения диаграммы компонентов связан с установлением и нанесением на диаграмму взаимных связей между компонентами, а также отношений реализации. Эти отношения должны иллюстрировать все важнейшие аспекты физической реализации системы, начиная с особенностей компиляции исходных текстов программ и заканчивая исполнением отдельных частей программы на этапе ее выполнения. Для этой цели можно использовать различные виды графического изображения компонентов.

При разработке диаграммы компонентов следует придерживаться общих принципов создания моделей на языке UML. В частности, необходимо использовать уже имеющиеся в языке UML компоненты и стереотипы. Для большинства типовых проектов этого набора элементов может оказаться достаточно для представления компонентов и зависимостей между ними.

Если проект содержит некоторые физические элементы, описание которых отсутствует в языке UML, то следует воспользоваться механизмом расширения и использовать дополнительные стереотипы для отдельных нетиповых компонентов или помеченные значения для уточнения их отдельных характеристик.

Следует обратить внимание, что диаграмма компонентов, как правило, разрабатывается совместно с диаграммой развертывания, на которой представляется информация о физическом размещении компонентов программной системы по ее отдельным узлам.

События WPF

События в WPF напоминают события WinForms и события .NET вообще (events) — события определяются в классах элементов управления и вызываются по выполнения некоторого действия (например, нажатия на кнопку). Программисты подписываются на события при помощи оператора «+=» и регистрируют таким образом обработчик события. Однако, в отличие от обычных событий WinForms, события WPF являются переадресуемыми (routed).

Переадресуемые события отличаются от обычных тем, что они вызываются не только в тех местах, где возникли события (например, на кнопке), но и на остальных связанных элементах управления (дочерних либо родительских).

В WPF различают три стратегии адресации событий:

- Нисходящая маршрутизация (tunnelling)
- Восходящая маршрутизация (bubbling)
- Прямой вызов

Все стратегии иллюстрируются следующим рисунком:



В случае туннелирования (нисходящая маршрутизация) события, произошедшие на элементах верхнего уровня (родительских) вызываются и для элементов нижнего уровня.

В случае всплытия (восходящая маршрутизация) события, произошедшие на элементах нижнего уровня (дочерних) вызываются и у элементов верхнего уровня.

С точки зрения WPF это события с префиксом Preview (туннелирование) и без него. Основная идея заключается в том, чтобы дать необходимую гибкость при обработке различных событий. Например, есть некоторая канва с набором различных элементов. Необходимо логгировать каждый щелчок по канве.

В традиционной ситуации для выполнения такого требования пришлось бы подписаться на событие щелчка мышки на канве и на каждом из элементов, принадлежащих канве (поскольку щелчок может выполниться и на них, но это тоже считается как щелчок по канве).

В WPF можно сделать проще — зарегистрироваться на PreviewClick событие у канвы; в таком случае перед каждым щелчком по отдельному элементу сначала будет срабатывать событие у канвы.

Иногда возникает ситуация, когда необходимо не допустить адресацию события дальше по цепочке родительских или дочерних элементов — для этого существует булевское свойство Handled. Устанавливая это свойство в true можно принудительно прервать маршрутизацию события.

Общий синтаксис событий обычновенный:

```
<Button Click="b1SetColor">button</Button>

void b1SetColor(object sender, RoutedEventArgs args) {
    //logic to handle the Click event
    // для прерывания маршрутизации события
    // args.Handled = true;
}
```

Стили WPF

Одна из особенностей WPF — возможность изменять внешний вид элементов управления посредством стилей. Стиль определяет значения свойств элемента управления — ширина рамки, фоновый цвет, цвет текста, отступы, размер шрифта и другие.

По сути, все свойства, выставляемые при помощи XAML напрямую могут быть выставлены в отдельном стиле. Это удобно, когда необходимо переиспользовать одни и те же наборы настроек — например, каждая кнопка должна содержать текст кеглем 15 пунктов.

Рассмотрим простейший пример стиля:

```
<Style TargetType="TextBlock">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14" />

```

У стиля прежде всего указывается тип, к которому он применяется. Если не указать стиль, будет считаться, что он применяется к базовому для всех элементов управления типу FrameworkElement. В таком случае набор устанавливаемых свойств будет ограничен доступными FrameworkElement.

Далее следует набор «установщиков» - каждый указывается значение конкретного свойства. В данном примере устанавливается центрирование по горизонтали и 14 кегль шрифта Comic Sans для всех текстовых подписей.

Стили как правило помещаются в т. н. ресурсы. Ресурсы — это специальное хранилище объектов внутри элементов управления (наподобие tag элемента в Windows Forms). В ресурсы могут помещаться изображения, коллекции объектов, стили, шаблоны и другие элементы. Ключевой особенностью ресурсов является то, что среда выполнения WPF производит поиск по ресурсам элементов управления при необходимости подключить тот или иной объект.

Рассмотрим пример:

```
<Window.Resources>
    <!--A Style that extends the previous TextBlock Style-->
```

```

<!--This is a "named style" with an x:Key of TitleText-->
<Style BasedOn="{StaticResource {x:Type TextBlock}}"
    TargetType="TextBlock"
x:Key="TitleText">
    <Setter Property="FontSize" Value="26"/>
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Offset="0.0" Color="#90DDDD" />
                    <GradientStop Offset="1.0" Color="#5BFFFF" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>

...
</Window.Resources>

<TextBlock Style="{StaticResource TitleText}" Name="textblock1">My Pictures</TextBlock><TextBlock>Check out my new pictures!</TextBlock>

```

Прежде всего, для применения ресурса к элементу управления необходимо дать ему ключ в словаре ресурсов посредством установки значения x:Key. У каждого элемента управления есть свойство Style, в котором указывается ключ стиля.

В предыдущем примере ключ указан не был — так тоже можно объявлять ресурс, тогда он станет стилем по-умолчанию (базовым) для всех элементов управления указанного типа.

Стили имеют также систему наследования — указываемую при помощи атрибута BasedOn. Отметим также, что в случае сложных значений свойств (например, фон указываемый при помощи градиента с несколькими точками), можно использовать Setter.Value.

Шаблоны WPF

```

<ControlTemplate TargetType="Button">    <Border
    Name="RootElement">

        <!--Create the SolidColorBrush for the Background
        as an object element and give it a name so          it can be
        referred to elsewhere in the          control template.-->
        <Border.Background>
            <SolidColorBrush x:Name="BorderBrush" Color="Black"/>      </Border.Background>

        <!--Create a border that has a different color
        by adding smaller grid. The background of          this grid
        is specified by the button's
        Background property.-->
        <Grid Margin="4" Background="{TemplateBinding Background}">

            <!--Use a ContentPresenter to display the Content of
            the Button.-->      <ContentPresenter
                HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
                VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
                Margin="4,5,4,4" />

```

```
</Grid>  
</Border>  
</ControlTemplate>
```

В данном случае кнопка представляет собой рамку черного цвета, в которой находится сетка с отступом в 4 платформо-независимых единицы (в общем случае, 4 пикселя) со всех сторон. Внутри сетки находится ContentPresenter — специальный элемент, который отображает то значение, которое будет написано разработчиков в Button.

Шаблон можно переопределить при помощи стиля и установщика значения свойства Template, при помощи непосредственного применения шаблона из словаря ресурсов либо непосредственно. Примеры:

```
<Window.Resources>  
  
<Style TargetType="TextBlock" x:Key="TemplatedTextBlock">  
    <Setter Property="Template">  
        <Setter.Value>  
            <ControlTemplate>  
                ...  
            </ControlTemplate>  
        </Setter.Value>  
    </Setter> </Style>  
  
...  
  
<TextBlock Style="{StaticResource TemplatetdTextBlock}" />  
  
<TextBlock Template="{StaticResource TextBlockTemplate}" />  
  
<TextBlock>  
    <TextBlock.Template>  
        <ControlTemplate>  
            ...  
        </ControlTemplate>  
    </TextBlock.Template>  
    </TextBlock>
```

Триггеры WPF

Триггеры — простая замена событийному движку WPF для выполнения некоторых действий или изменения внешнего вида компонента при изменении значений его свойств.

Простейший пример триггера — подсвечивание элемента при наведении мышки. Рассмотрим его:

```
<Style TargetType="ListBoxItem">  
    <Setter Property="Opacity" Value="0.5" />  
    <Setter Property="MaxHeight" Value="75" />  
    <Style.Triggers>  
        <Trigger Property="IsSelected" Value="True">  
            <Setter Property="Opacity" Value="1.0" />      </Trigger>
```

...

```
</Style.Triggers>  
</Style>
```

Триггеры определяются в стилях или шаблонах элементов. У триггера указывается свойство, к которому он привязывается, в данном случае — это свойство «выбран ли элемент» (`IsSelected`). Далее указывается значение свойства, по которому срабатывает триггер. И внутри — описание необходимых действий, как правило это установка свойств элемента — в данном случае выделенные элементы становятся непрозрачными. Вот примерный вид интерфейса:



Специально для поддержки механизма триггеров команда Microsoft создала набор базовых булевских свойств у элементов управления — `IsSelected`, `IsMouseOver`, `.IsEnabled` и другие.

Триггеры существуют следующих типов:

- Trigger — выполняется по изменению одного из свойств объекта;
- EventTrigger — выполняется по возникновению маршрутизируемого события;
- MultiTrigger — выполняется по выполнению множества условий (аналог нескольких последовательно связанных триггеров);
- DataTrigger — выполняется по выполнению условия не по свойству, а по данным, связанным с элементом управления;
- MultiDataTrigger — условия берутся из множества связанных данных; Триггеры в целом представляют очень удобный механизм добавления визуальной интерактивности элементов при помощи визуального оформления без использования логики отрисовки.

Контрольные вопросы

1. Какие элементы присутствуют на диаграмме компонентов ?
2. Что представляют собой связи на диаграмме компонентов ?
3. Что такое шаблоны элементов управления WPF ?
4. Что представляют собой триггеры в WPF ?
5. Как определяются стили в WPF ? Что они содержат ?

ЛАБОРАТОРНАЯ РАБОТА № 9.

Диаграмма деятельности. Шаблон MVVM, механизм привязок, команды WPF

Задание

1. Ознакомиться с краткими теоретическими сведениями.
2. Разработать диаграмму деятельности для проектируемой системы.
3. Реализовать логику работы форм, реализованных в предыдущей лабораторной работе, при помощи шаблона MVVM с использованием привязок по данным и по командам.
4. Оформить отчет о выполнении лабораторной работы.

Краткие теоретические сведения

Диаграмма деятельности

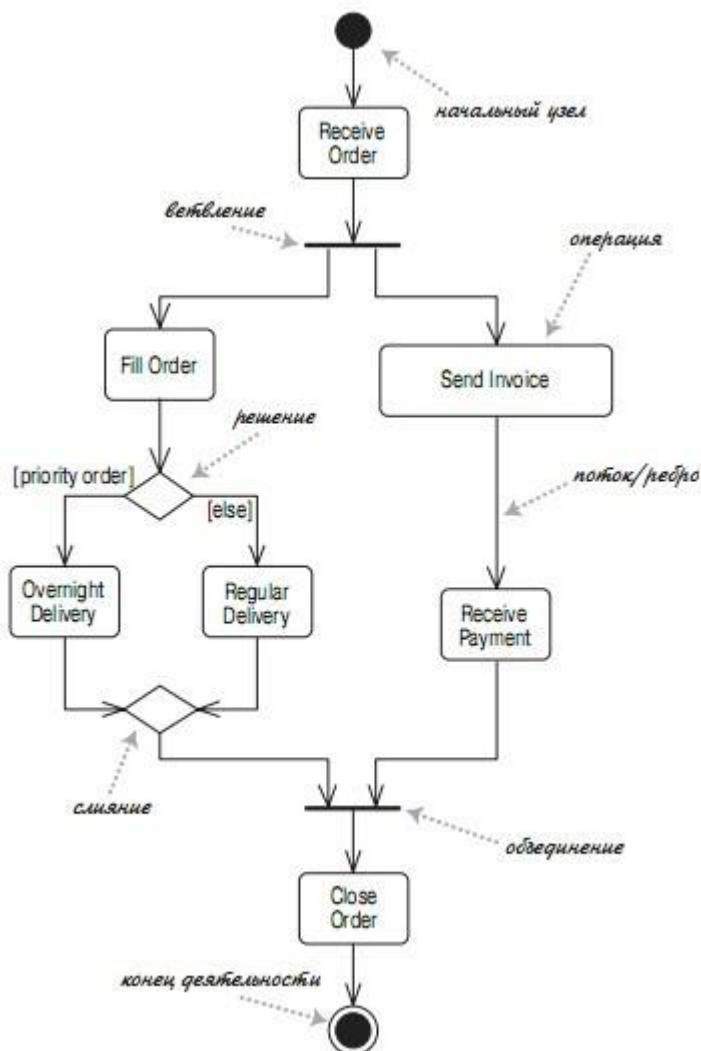
Диаграммы деятельности – это технология, позволяющая описывать логику процедур, бизнес-процессы и потоки работ. Во многих случаях они напоминают блок-схемы, но принципиальная разница между диаграммами деятельности и нотацией блок-схем заключается в том, что первые поддерживают параллельное процессы.

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций.

В контексте языка UML деятельность (activity) представляет собой совокупность отдельных вычислений, выполняемых автоматом, приводящих к некоторому результату или действию (action). На диаграмме деятельности отображается логика и последовательность переходов от одной деятельности к другой, а внимание аналитика фокусируется на результатах. Результат деятельности может привести к изменению состояния системы или возвращению некоторого значения.

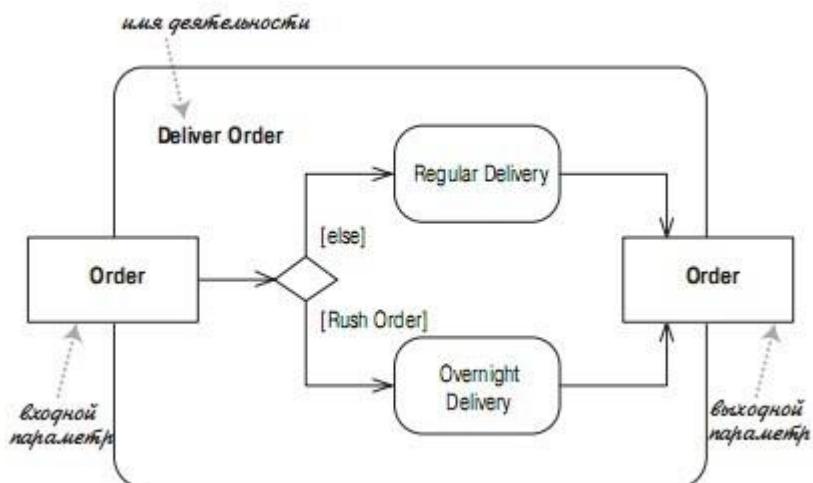
Графически состояние действия изображается прямоугольником с закругленными углами (рис. 5). Внутри этого изображения записывается выражение действия (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами. Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать конкретный проект.



Изображение состояния действия

Иногда возникает необходимость представить на диаграмме деятельности некоторое сложное действие, которое, в свою очередь, состоит из нескольких более простых действий. В этом случае можно использовать специальное обозначение состояния под-деятельности (subactivity state).



Изображение состояния под-деятельности

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния. Они имеют такие же обозначения, как и на диаграмме состояний. При этом каждая

деятельность начинается в начальном состоянии и заканчивается в конечном состоянии. Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз. В этом случае начальное состояние будет изображаться в верхней части диаграммы, а конечное в нижней.

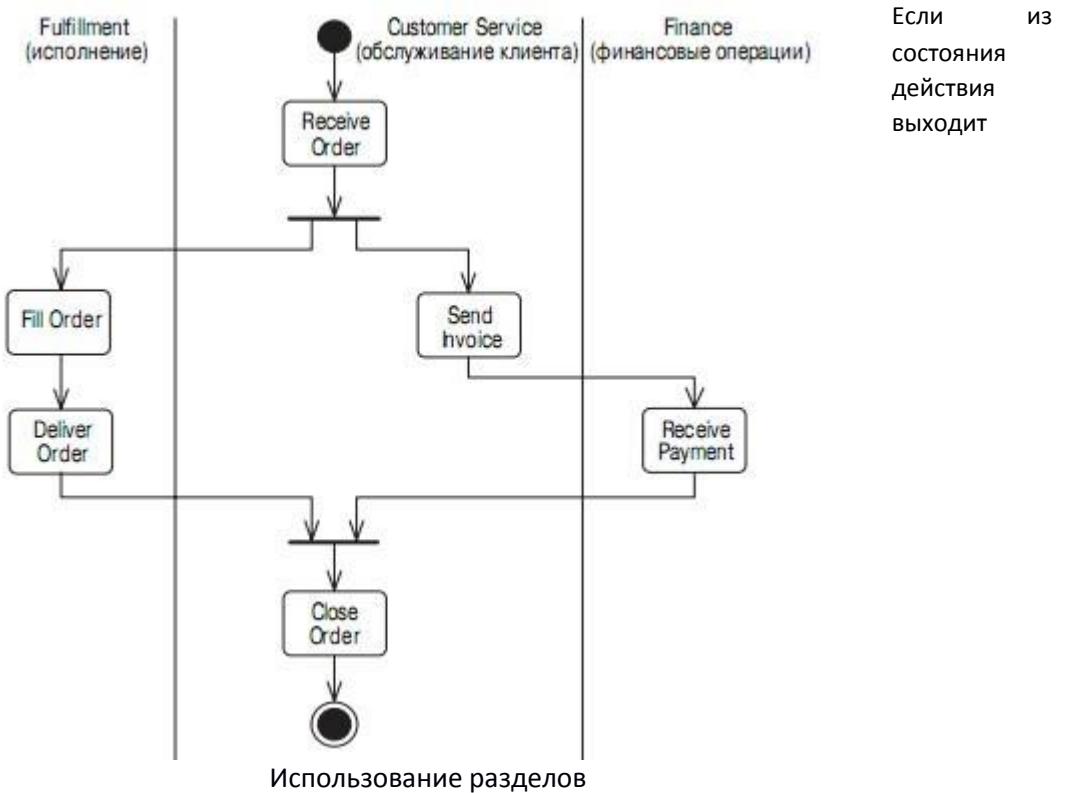
Переходы между действиями изображаются сплошной стрелкой. Переход между состояниями означает завершение одного действия и начало другого. Помимо простых переходов, существуют также элементы выбора - «решения». Решения — это аналог элемента «условие» из блок-схем; в случае истинного значения условия предпринимается одна цепочка действия, в обратном случае — другая. На диаграммах они, точно так же как и на блок-схемах, отображаются ромбом.

Помимо решений есть также операции ветвления и слияния. Это не условные операции; это распаралеливание процесса выполнения действий. При ветвлении после выполнения действия следующие действия выполняются параллельно и независимо друг от друга (если не указано иначе); при слиянии — восстановливается однопоточная обработка действий.

Диаграммы деятельности рассказывают о том, что происходит, но ничего не говорят о том, кто какие действия выполняет. В программировании это означает, что диаграмма не отражает, какой класс является ответственным за ту или иную операцию. В моделировании бизнес-процессов это означает, что не отражено распределение обязанностей между подразделениями фирмы. Это не всегда представляет собой трудность; часто имеет смысл сконцентрироваться на том, что происходит, а не на том, кто какую роль играет в данном сценарии.

Можно разбить диаграмму деятельности на разделы (partitions), чтобы показать, кто что делает, то есть какие операции выполняет тот или иной класс или подразделение предприятия. На рис.11.4 приведен простой пример, показывающий, как операции по обработке заказа могут быть распределены между различными подразделениями.

Разделы — исключительно логическая единица организации диаграммы. Если рассматривать с точки зрения исходного кода, создание разделов не несет никакого отражения на генерируемом программном коде. Возможен случай, когда разные разделы отражают разные исходные файлы, однако использование разделов таким образом встречается крайне редко — поскольку одно действие может выполняться в различных файлах с исходным кодом (классах).



Самое большое достоинство диаграмм деятельности заключается в том, что они поддерживают и стимулируют применение параллельных процессов. Именно благодаря этому они представляют собой мощное средство моделирования потоков работ. Множество импульсов к развитию UML 2 пришло от людей, вовлеченных в эти потоки работ.

Диаграммы деятельности являются ценным инструментом для представления логики поведения систем. Диаграммы деятельности применяются для описания прецедентов. Опасность такого подхода в том, что часто эксперты в предметной области с трудом могут им следовать. Если дело обстоит так, то лучше обойтись обычной текстовой формой.

Механизм привязок WPF

Механизм привязки данных в WPF предоставляет простой способ связывания визуального интерфейса и данных в виде объектов .NET. При этом, в случае правильной настройки привязки и реализации механизма нотификации (посредством интерфейса `INotifyPropertyChanged`) при изменении данных в визуальном интерфейсе (например, заполнение `TextBox`'а) привязанный класс данных тоже изменяется. Рассмотрим пример:

```
<TextBlock x:Name="horseName" Text="{Binding Path=Name}" />
```

```
public class Horse
{
    public string Name { get; set; }
}

public partial class AWindow : Window
{
    public AWindow()
    {
```

```

        InitializeComponent(); this.DataContext = new Horse { Name =
    «MyHorse» };
}
}

```

Механизм привязок работает следующим образом. Определяется класс данных, содержащий значение — в данном случае это Horse. На желаемое свойство элемента управления создается привязка и указывается, к какому свойству привязываться (Path = «Name»). TextBox знает, что необходимо обращаться к классу Horse, потому что он установлен как DataContext на окне. DataContext — это объект, связанные с элементом управления, по которому берется значение по привязке. В случае, если он не установлен — берется DataContext первого родительского элемента, у которого он установлен (в данном случае — окно).

В результате при изменении данных в окне, в экземпляре класса Horse значение Name также изменится — автоматически, без дополнительного кода подвязки.

Чтобы заработала обратная сторона интеграции — изменения в экземпляре класса Horse были видны в визуальном интерфейсе — необходимо реализовать интерфейс INotifyPropertyChanged:

```

public class Horse:INotifyPropertyChanged
{
    private string _name;

    #region INotifyPropertyChanged Members
    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    #endregion
    public string Name
    {
        get{ return _name; }      set
        {
            _name = value;
            OnPropertyChanged("Name");
        }
    }
}

```

Механизм привязок проверяет реализацию данного интерфейса и в случае успеха подписывается на событие изменения свойства. При всяком изменении свойства Name автоматически вызывается событие PropertyChanged, и механизм привязок считывает значение и устанавливает его в визуальном элементе.

Привязки также имеют некоторые настройки. По обновлению взаимодействующих сторон они бывают:

- Двусторонние (обновляется и визуальный интерфейс, и класс данных);
- Односторонние к визуальному интерфейсу;
- Односторонние к классу данных;
- Односторонние к визуальному интерфейсу 1 раз;



По способу получения данных можно привязки организовывать следующим образом:

- Из контекста данных (по-умолчанию) — через DataContext;
- Из свойств других элементов управления — при помощи ElementName;
- Из свойств других элементов — при помощи относительной привязки (RelativeSource);

Привязки можно также использовать вместе с триггерами. Например, можно настроить триггер данных, который будет изменять рамку кнопки в зависимости от значения в модели.

Также настраивается момент обновления источника/приемника привязки(UpdateSourceTrigger):

- LostFocus — значение в модели обновляется по потере фокуса элемента управления;
- PropertyChanged — при вводе (при нажатии каждой клавиши);
- Explicit — когда должно вызываться из программного кода.

Некоторые примеры привязок:

```

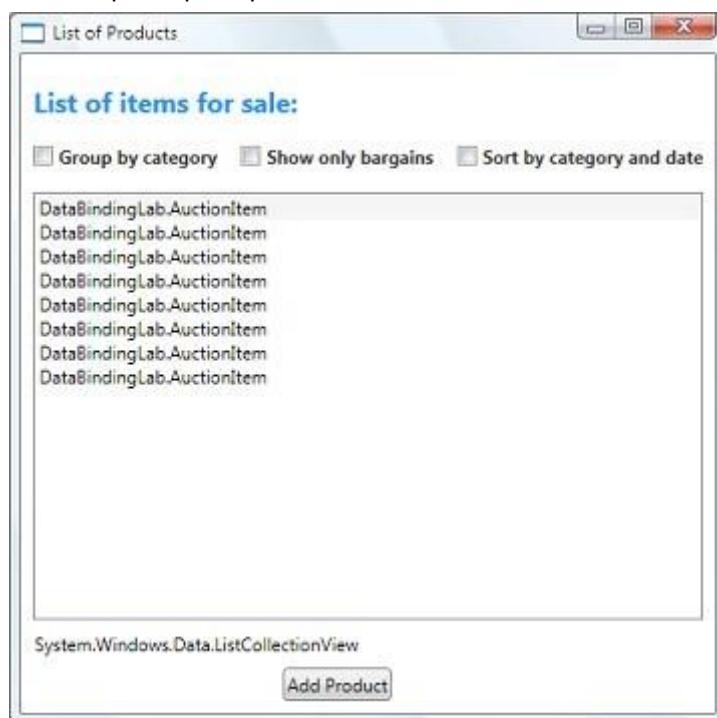
<DockPanel.Resources>
  <c:MyData x:Key="myDataSource"/>
</DockPanel.Resources>
<Button Width="150" Height="30"
       Background="{Binding Source={StaticResource myDataSource},
                           Path=ColorName}">I am bound to be RED!</Button>

<TextBox Name="StartPriceEntryForm" Grid.Row="2" Grid.Column="1"      Style="{StaticResource
textStyleTextBox}" Margin="8,5,0,5">
  <TextBox.Text>
    <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">
      <Binding.ValidationRules>
        <ExceptionValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
<Binding Path=PathToProperty, RelativeSource={RelativeSource AncestorType={x:Type typeOfAncestor}}>

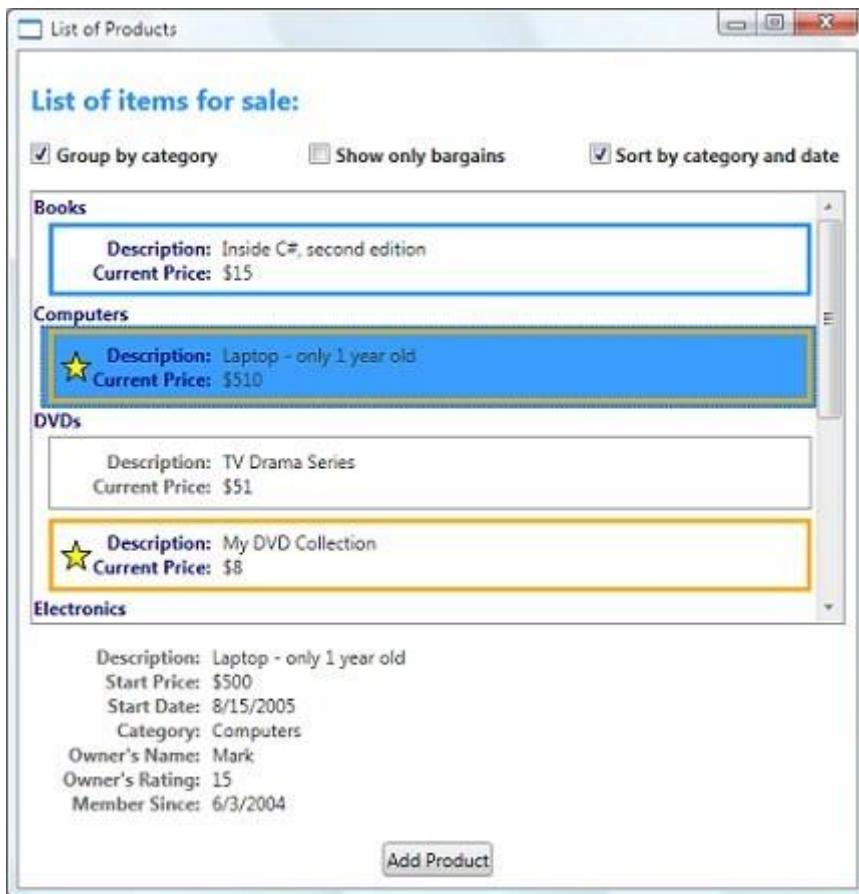
```

Шаблоны данных в WPF

Очень часто становится задача отобразить список объектов — например, список пациентов в больнице. При этом список пациентов — это набор .NET объектов — экземпляров класса Patient. Для красивого их отображения в представлении в WPF существует шаблоны данных — DataTemplate. Они указывают, какой внешний вид будут принимать объекты конкретного типа. Рассмотрим пример.



Без использования шаблонов данных, привязанные объекты AuctionItem будут отображаться в виде TextBlock-ов с текстом, определенным в методе ToString() объектов. В случае, если мы хотим отобразить их следующим образом:



Достаточно добавить следующий шаблон данных:

```
<DataTemplate DataType="{x:Type src:AuctionItem}">
    <Border BorderThickness="1" BorderBrush="Gray"
        Padding="7" Name="border" Margin="3" Width="500">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="20"/>
                <ColumnDefinition Width="86"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>

            <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
                Fill="Yellow" Stroke="Black" StrokeThickness="1"
                StrokeLineJoin="Round" Width="20" Height="20"
                Stretch="Fill"
                Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7 7,7"
                Visibility="Hidden" Name="star"/>

            <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
                Name="descriptionTitle"
                Style="{StaticResource smallTitleStyle}">Description:</TextBlock>
            <TextBlock Name="DescriptionDTData" Grid.Row="0" Grid.Column="2"
                Text="{Binding Path=Description}"
                Style="{StaticResource textStyleTextBlock}"/>
            <TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
                Name="currentPriceTitle"
                Style="{StaticResource smallTitleStyle}">Current Price:</TextBlock>
            <TextBlock Name="CurrentPriceDTData" Grid.Row="1" Grid.Column="2"
                Text="{Binding Path=CurrentPrice}"
                Style="{StaticResource textStyleTextBlock}"/>
        </Grid>
    </Border>
</DataTemplate>
```

```

        Name="currentPriceTitle"
        Style="{StaticResource smallTitleStyle}">Current Price:</TextBlock>
    <StackPanel Grid.Row="1" Grid.Column="2" Orientation="Horizontal">
        <TextBlock Text="$" Style="{StaticResource textStyleTextBlock}"/>
        <TextBlock Name="CurrentPriceDTData-Type"
            Text="{Binding Path=CurrentPrice}"
            Style="{StaticResource textStyleTextBlock}"/>
    </StackPanel>
</Grid>
</Border>
<DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
        <DataTrigger.Value>
            <src:SpecialFeatures>Color</src:SpecialFeatures>
        </DataTrigger.Value>
        <DataTrigger.Setters>
            <Setter Property="BorderBrush" Value="DodgerBlue" TargetName="border" />
            <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
            <Setter Property="BorderThickness" Value="3" TargetName="border" />
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger.Setters>
    </DataTrigger>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
        <DataTrigger.Value>
            <src:SpecialFeatures>Highlight</src:SpecialFeatures>
        </DataTrigger.Value>
        <Setter Property="BorderBrush" Value="Orange" TargetName="border" />
        <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
        <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
        <Setter Property="Visibility" Value="Visible" TargetName="star" />
        <Setter Property="BorderThickness" Value="3" TargetName="border" />
        <Setter Property="Padding" Value="5" TargetName="border" />
    </DataTrigger>
</DataTemplate.Triggers>
</DataTemplate>

```

У шаблона данных (DataTemplate) обязательно указывается тип данных, к которому он относится. Шаблоны данных находятся в ресурсах, и могут также иметь ключ — в таком случае, они будут подключаться только если принудительно указать ключ к шаблону данных. Без указания ключа (x:Key) они будут установлены как шаблоны по-умолчанию.

Важно отметить, что шаблон данных может иметь триггеры — изменять внешний вид в зависимости от определенных значений либо экземпляра класса либо собственных свойств (наведение мышки и др.).

Шаблон данных во многом напоминает шаблон элемента управления — только вместо элемента управления стилизуются обычные объекты, которые поставлены в качестве DataContext у элементов управления в свойство Content.

Механизм команд WPF

Команда в WPF — это действие, которое должно быть выполнено при каком-то событии. Команды заменяют собой событийную модель реализации логики представления, поскольку вместо подписки на событие — к элементу управления привязывается объект команды, у которого будет вызван соответствующий метод. Все команды реализуют интерфейс ICommand:

```
public interface ICommand
{
    public bool CanExecute(object parameter); public
    void Execute(object parameter); event EventHandler
    CanExecuteChanged;
}
```

При выполнении какого-то события (например, щелчок на кнопку), у команды вызывается метод Execute с параметром, который указан дополнительно в привязке. Метод CanExecute используется для отключения элементов управления (IsEnabled). Простейший пример привязки команды:

```
<Button Content="Click"
        Height="23"
        HorizontalAlignment="Left"
        Margin="77,45,0,0"
        Name="btnClick"
        VerticalAlignment="Top"
        Width="203"
        Command="{Binding ButtonCommand}"
        CommandParameter="Hai" />
```

Параметр команды может браться из класса данных, связанных с элементом управления при помощи привязки. Команды предоставляют простой способ привязки функций к соответствующим событиям элементов управления.

Шаблон MVVM

Шаблон Model-View-ViewModel (MVVM) применяется для разделения ответственности между компонентами программного обеспечения, облегчения разработки и поддержки приложений с насыщенной логикой визуального интерфейса, повышения тестируемости кода и другое. Данный шаблон получил широкое применение в приложения WPF ввиду простоты его внедрения и удобства использования.

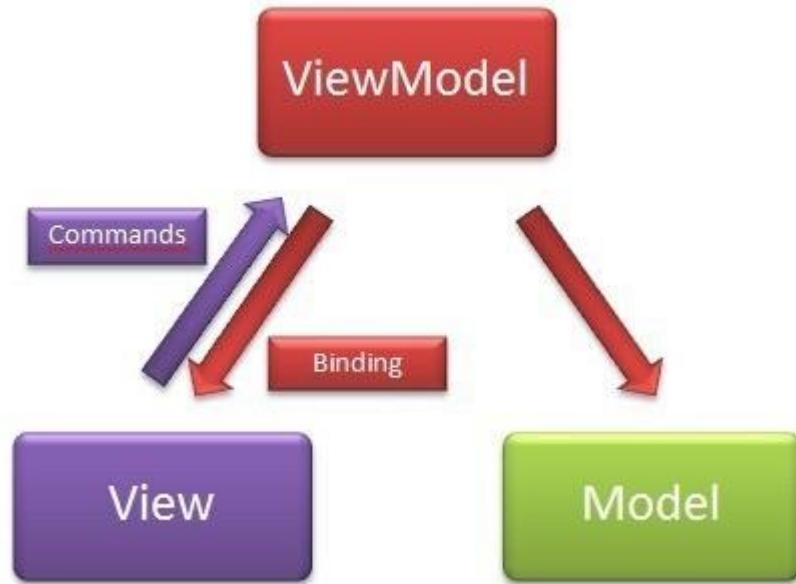
Основная проблема при написании приложений WinForms или WPF с применением событийной системы реализации бизнес-логики работы — огромные массы кода в code-behind классах представлений (форм, view), которые сложно отслеживать и поддерживать. Например, когда необходимо предпринимать некоторые действия при изменении галочки (снята/добавлена), приходится подписываться на множество событий, при этом не всегда тривиально и ясно, каким образом конкретно протекает ход событий. Также сложна логика заполнения формы значениями — как правило это выделенные методы InitForm, ClearForm(для очистки всех текстовых полей и галочек от значений) и прочее.

В замену этим трудностям рекомендуется использовать специальные классы, содержащие в себе полностью всю логику отработки представления — логика работы кнопок, наведения мышки и прочее. Этот же класс должен хранить состояния галочек, введенные текстовые значения и другое. Такой класс называется модель представления (view model) — он хранит все данные и логику представления (view). Необходимо сразу отметить, что он не имеет никакой логики, касающейся отрисовки — вся она должна остаться в представлении (view).

Третий элемент шаблона — модель — это чистые классы данных системы либо сервисы или слой доступа к данным, из которого можно получить необходимые данные для заполнения формы.

Например, если это форма редактирования товара, то моделью тут будет товар, его описание, связанные товары и прочее.

Модель представления (viewmodel) использует сервисы и данные, получаемые от модели (model) для заполнения представления (view). Для достижения этой цели.viewmodel использует механизм привязок и специальный интерфейс INotifyPropertyChanged.



Таким образом, для каждого представления создается специальный класс, содержащий всю логику работы представления без деталей визуального отображения. И механизм привязок (команд и свойств) позволяет легко и без усилий перенести всю эту логику на визуальный интерфейс.

Контрольные вопросы

1. Чем отличается диаграмма последовательностей от блок-схем ?
2. Что представляют собой разделы на диаграмме последовательностей ?
3. Что такое привязки в WPF ?
4. Как осуществляется привязка команд в WPF ?
5. Что такое MVVM ?

ЛАБОРАТОРНАЯ РАБОТА №10.

Диаграмма состояний. Анимации и пользовательские элементы управления WPF

Задание.

1. Разработать диаграмму состояний.
2. Реализовать анимированных переход между окнами в приложении/
3. Реализовать несколько видов анимаций по различным свойствам зависимостей.
4. Реализовать собственный элемент управления.

Краткие теоретические сведения.

Диаграмма состояний UML

Диаграммы состояний являются средством моделирования поведения системы в целом или отдельных сущностей системы, в зависимости от глубины рассмотрения системы на диаграмме. Диаграмма состояний (statechart diagram) используется для моделирования поведения объектов посредством конечного набора состояний и переходов между ними. В математике такие структуры обычно называют конечным автоматом (finite automaton).

Таким образом, поведение определяется как прохождение по графу состояний (узлы — состояния, ребра — переходы между ними). Переходы осуществляются при срабатывании некоторых событий (условий). При переходах возможно также выполнение некоторых действий.

Событие (event) – это спецификация существенного факта, который занимает некоторое положение во времени и в пространстве. В контексте диаграмм состояний событие – это спецификация факта, который может привести к смене состояния. События могут быть внутренними или внешними. Внешние события передаются между системой и актерами (например, нажатие кнопки или посылка сигнала от датчика передвижений). Внутренние события передаются между объектами внутри системы. В UML можно моделировать четыре вида событий:

- сигналы;
- вызовы;
- истечение промежутка времени;
- изменение состояния.

В UML различают 3 вида состояний:

- обычные состояния;
- составные состояния;
- подавтомат.

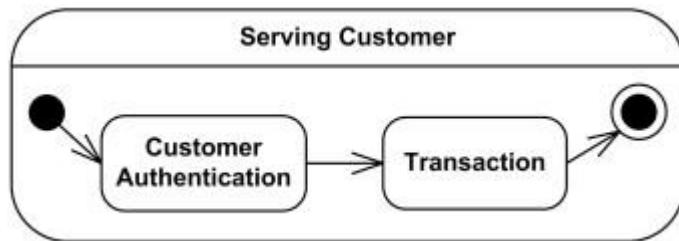
Обыкновенное состояние не имеет подсостояний — не несет подобластей и подавтоматов. Обыкновенные состояния содержат следующую информацию: название, внутренние действия, внутренние переходы.



Во внутренних действиях состояния указывается список действий или деятельностей, выполняемых в различные моменты времени в состоянии в зависимости от условий. Есть несколько зарезервированных условий:

- entry — действия, выполняемые при входе системы в это состояние;
- exit — действия, выполняемые при выходе системы из этого состояния;
- do — деятельность, выполняющаяся на протяжении всего периода пребывания системы в этом состоянии.

Составные состояния содержат внутренние состояния, точку входа и выхода из состояния. Состояния внутри подсостояний могут быть как последовательные, так и параллельные.



Псевдо-состояния

Спецификация UML определяет также набор псевдо-состояний — служебных состояний, представляющих различные способы управления переходами на диаграммах состояний.

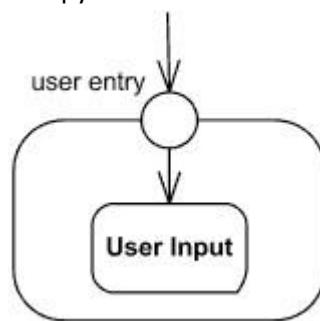
Исходное псевдо-состояние — точка входа на диаграмму состояний либо в конкретное состояние. Изображается в виде закрашенного круга.



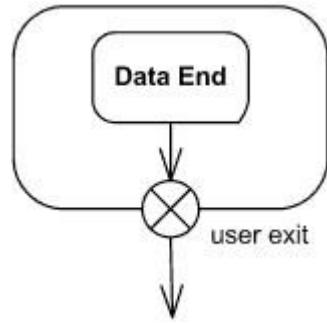
Терминирующее псевдо-состояние — отмечает окончание жизненного цикла объекта/контекста, который рассматривается на диаграмме состояний. Отмечается при помощи крестика.



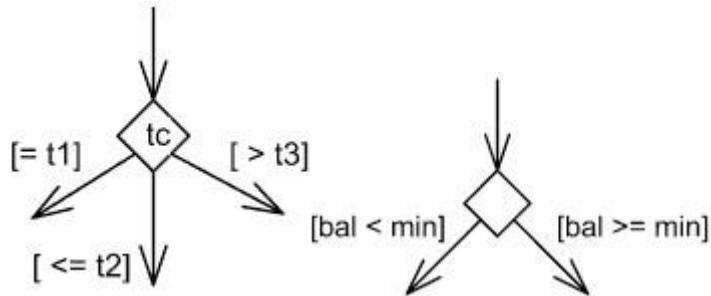
Псевдо-состояние точки входа — точка входа для состояния или подавтомата. Изображается в виде незакрашенного круга.



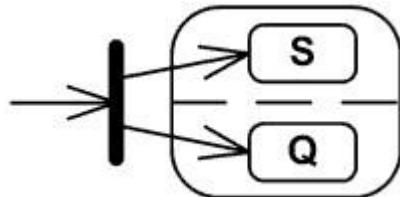
Точка выхода — выход из состояния или подавтомата; изображается в виде круга с крестиком внутри.



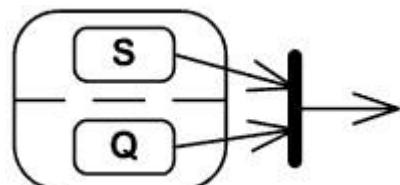
Условие (выбор) — состояние, являющиеся аналогом блока условия на диаграммах деятельности / блок-схемах. Используется для выбора следующего состояния в зависимости от определенных условий.



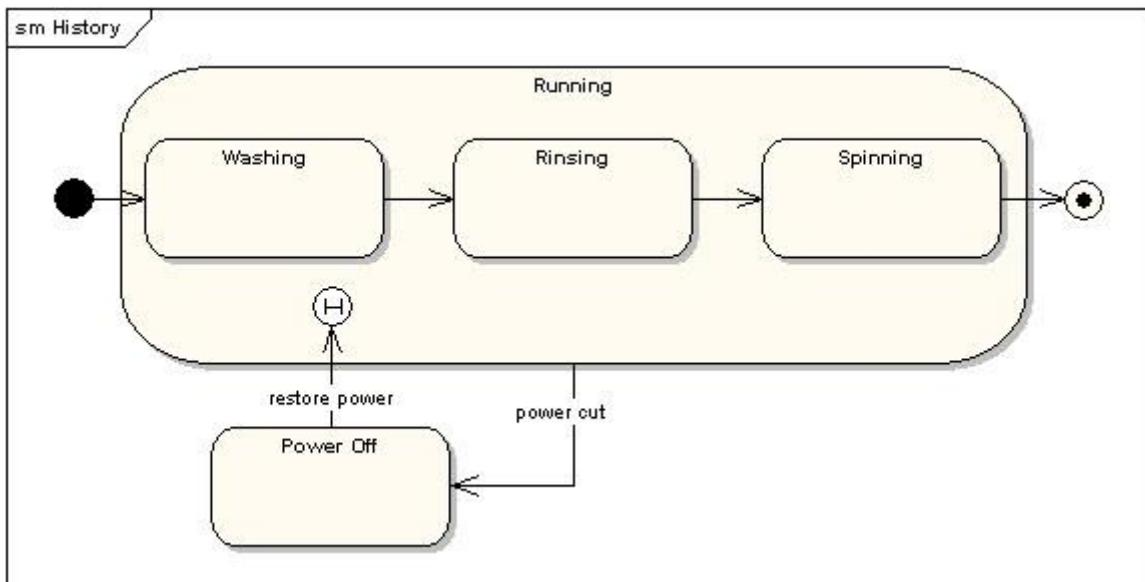
Развилка — состояние, определяющее распараллеливание процесса переходов по состояниям в системе. Используется для отображения нескольких параллельно выполняющихся состояний.



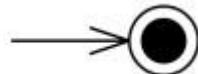
Слияние — состояние, обратное состоянию развлки.



Исторические события — события глубокой и неглубокой истории; отображают, что выполнение состояния должно продолжаться с того же момента, в котором оно и закончилось в случае непредвиденного завершения прохождения по графу состояний:

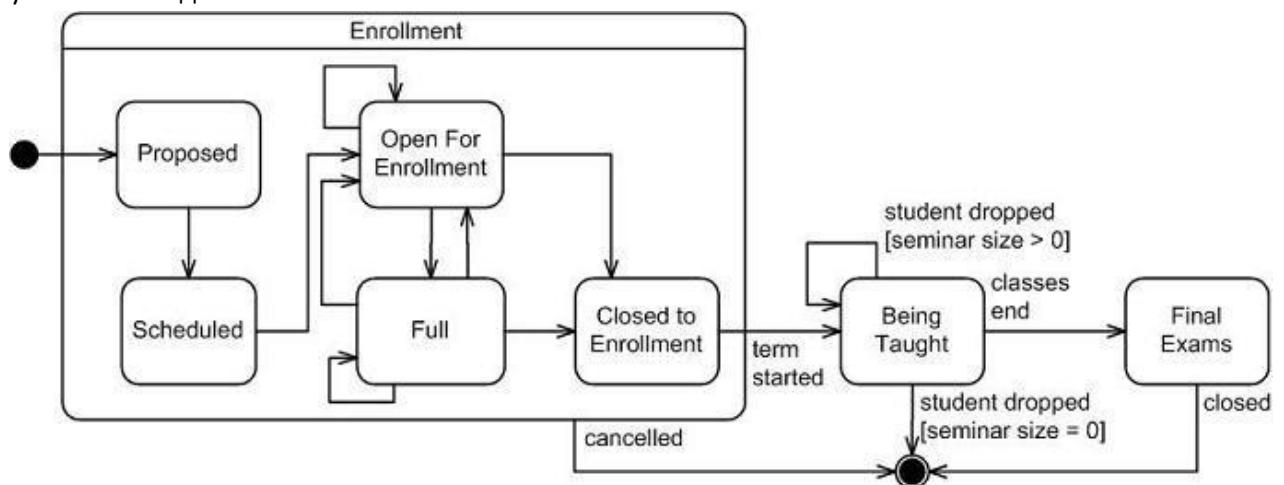


Конечное состояние — состояние, отмечающее завершение отработки подавтомата/состояния; конечное состояние автомата.

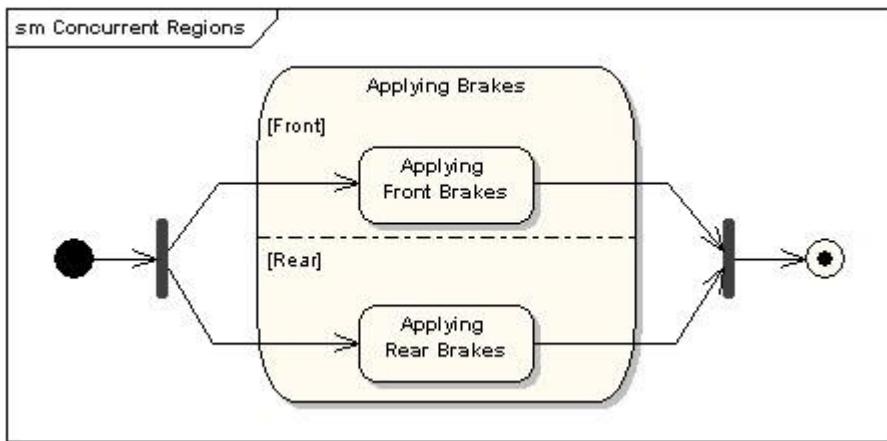


Примеры диаграмм состояния

Пример использования составных и простых состояний на одной диаграмме. При этом необходимо отметить, что переходы также могут иметь условия, при которых выполняется переход, и поясняющие комментарии. В данном случае отображен процесс обучения студентов в высшем учебном заведении.



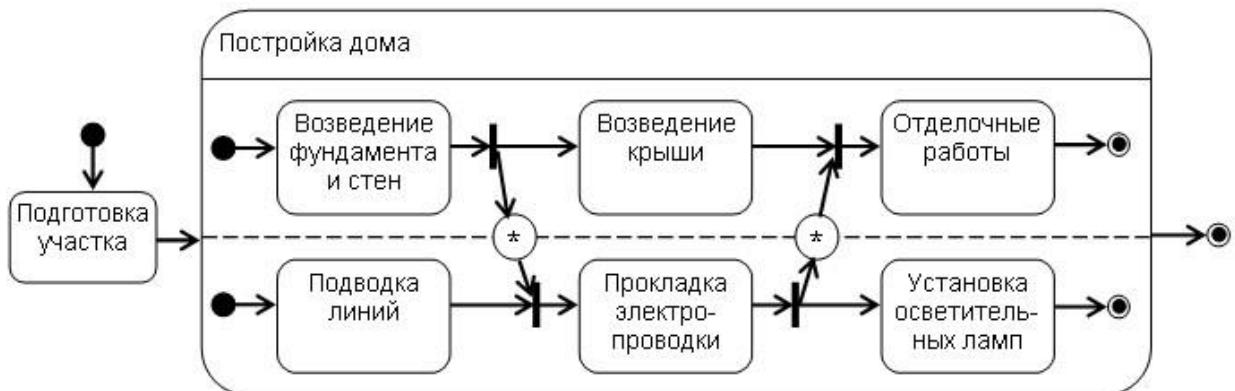
На данной диаграмме приведен автомат работы тормозной системы велосипеда в момент ее срабатывания. В данном случае показано, что задние и передние тормоза применяются к колесам параллельно, а не последовательно, после чего автомат заканчивает свою работу. Необходимо отметить, что части состояния могут также иметь заголовок.



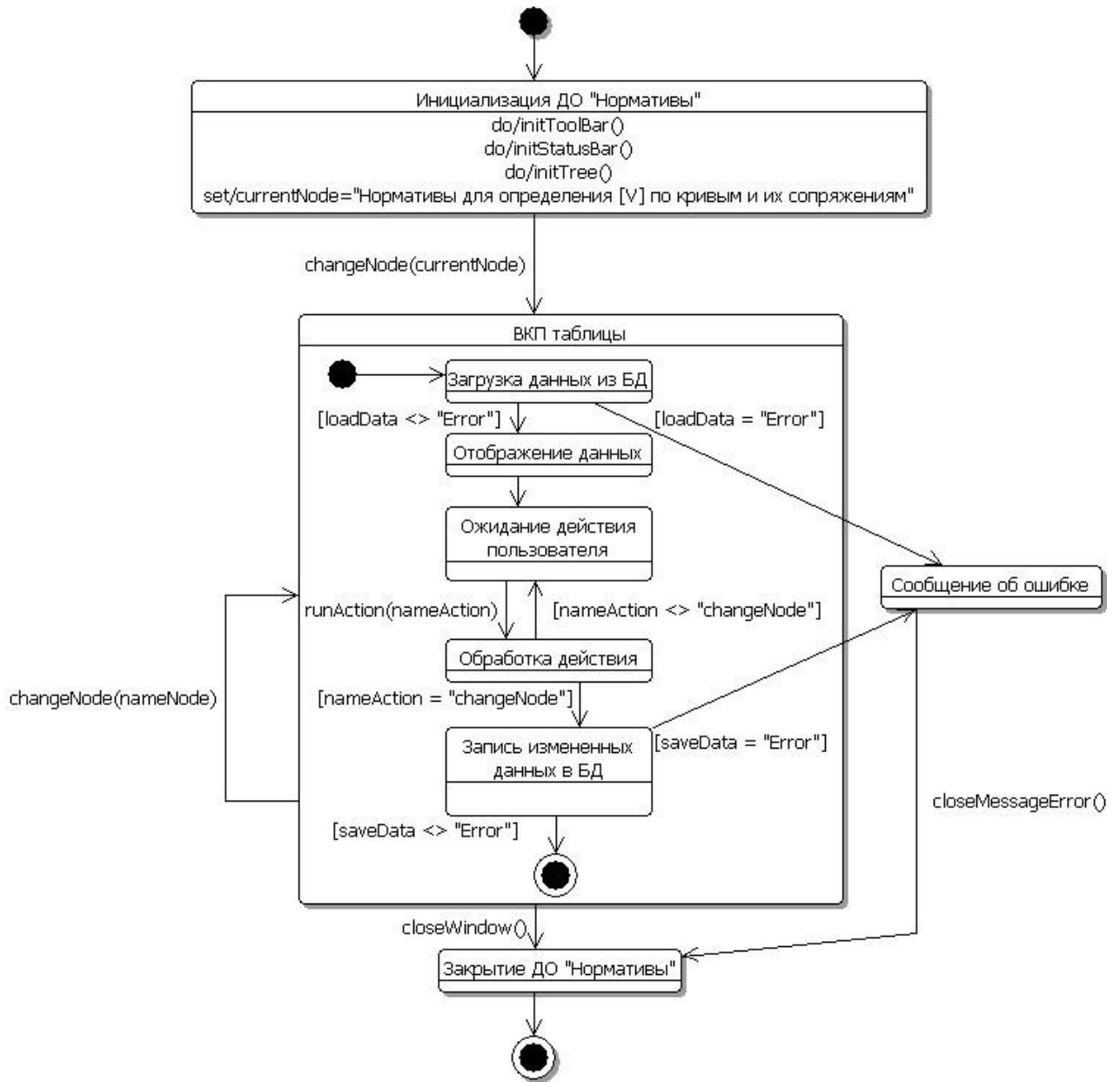
На следующей диаграмме изображен жизненный цикл заказа в интернетмагазине. Следует обратить внимание на то, что закрытие заказа приводит к обрыву жизненного цикла объекта заказа (о чем свидетельствует символ терминации) и к завершению отработки конечного автомата.



На приведенной ниже диаграмме представлен процесс постройки дома. В данном случае изображены два параллельно выполняющихся процесса, однако при этом дополнительно используются псевдо-состояния синхронизации взаимодействий частей составного состояния (круг со звездочкой).



Напоследок, приведем пример диаграммы состояний реального программного комплекса с определением переходов в соответствующие окна и выполнения действий по обработке данных, взаимодействием с БД и др. Важно отметить, что при помощи таких диаграмм состояний в промышленных пакетах моделирования имеется возможность воссоздания исходных кодов на различных языках программирования(включая популярные языки, такие как Java, C++, C#, ruby, python, objective-c и другие).



Анимации WPF

Анимация — оптическая иллюзия, которая достигается частой сменой статических изображений с целью создания иллюзии движения. В программировании ранее для реализации анимации необходимо было выполнить следующие действия для анимирования (создания динамики статических объектов):

1. Создать таймер
 2. Проверять значение таймера в дискретные интервалы времени
 3. Выполнять анимирующее действие в эти дискретные интервалы времени, возможно с вычислением нового значения того или иного свойства (положения относительно левого края экрана, прозрачности фона и т. п.)
 4. Перерисовывать поверхность с новыми значениями.

В технологии WPF анимация является «first-class citizen», т. е. одной из основных концепций, внедренных в платформу. При помощи технологии WPF относительно просто создавать сложные анимации с использованием множества разных свойств, триггеров и с подключением множества одновременно работающих таймеров. Самое главное преимущество для программиста в данном случае — декларативное описание анимаций; программист не задумывается о необходимых

программных служебных структурах для поддержки анимации, он лишь указывает, какую конкретно анимацию и каким образом ему необходимо достичь.

Анимация и свойства

Прежде всего, необходимо понимать, что анимация относится к свойства. Например, для увеличения объекта, можно использовать его height и width свойства. Для перемещения по экрану можно использовать его свойства Margin или Canvas.Left (в случае, если он нарисован на канве). Для поддержки анимации, свойства должны иметь следующие характеристики:

- Свойство должно быть зависимым (dependency property);
- Оно должно быть внутри класса, реализующего DependencyObject и IAnimatable;
- Должен существовать хотя бы один совместимый тип анимации (имеется возможность создавать собственные типы анимации).

Простейший пример анимации, использующий DoubleAnimation (линейно изменяющийся набор значений по времени):

```
<Window x:Class="WpfApplication1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
<Grid>
    <StackPanel Margin="10">
        <Rectangle
            Name="MyRectangle"
            Width="100"
            Height="100"
            Fill="Blue">
            <Rectangle.Triggers>
                <!-- Animates the rectangle's opacity. -->
                <EventTrigger RoutedEvent="Rectangle.Loaded">
                    <BeginStoryboard>
                        <Storyboard>
                            <DoubleAnimation
                                Storyboard.TargetName="MyRectangle"
                                Storyboard.TargetProperty="Opacity"
                                From="1.0" To="0.0" Duration="0:0:5"
                                AutoReverse="True" RepeatBehavior="Forever" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Rectangle.Triggers>
        </Rectangle>
    </StackPanel>
</Grid>
</Window>
```

В данном случае прямоугольник будет исчезать и появляться (при помощи изменения его прозрачности) бесконечно каждую секунду.

Типы анимаций

Анимация представляет собой просто набор значений, применяемых к свойству, сгенерированных на промежутке времени. Из-за того, что свойства имеют различные типы, существуют и различные типы анимаций. Например, для анимирования свойств, имеющих тип Double, необходимо использовать DoubleAnimation; для Point — PointAnimation. Помимо этого, существуют различные типы в зависимости от того, какой числовой ряд генерируется. Рассмотрим некоторые из них.

- <Тип>Animation — линейно-изменяющаяся анимация, характеризуется тремя свойствами: от, до, шаг. (From/To/By animation)
- <Тип>AnimationUsingKeyFrames — анимация, похожая на From/To/By анимацию, но имеющую несколько конечных точек. Т.е. вместо анимирования от конкретного значения до конкретного значения, анимирование происходит в несколько точек, например: от 0 до 50 за 3 секунды; от 50 до 20 за 2 секунды; от 20 до 100 за 4 секунды. Таким образом можно добиваться «прыгающих» или «скользящих» анимаций.
- <Тип>AnimationUsingPath — анимация, указываемая при помощи пути (Path).
- <Тип>AnimationBase — базовый класс для реализации классов <Тип>Animation <Тип>AnimationUsingKeyFrames. Используется только при реализации собственных анимаций.

Storyboard

Прежде всего, каждая анимация наследуется от объекта Timeline. Timeline представляет собой некоторый промежуток времени. Данные промежуток характеризуется длительностью, повторимостью и скоростью его прохождения (он не обязательно привязан к системным часам).

Всякая анимация применяется к свойству конкретного элемента управления. Для этого у класса StoryBoard (доска истории; агрегат анимаций) содержатся свойства TargetName и TargetProperty — целевой элемент управления и целевое свойство.

Сами по себе StoryBoard поддерживают также возможность контроля над анимацией — остановку, паузу, перемещение по временной оси (timeline) и другое. StoryBoard можно запустить при помощи EventTrigger'a (как показано в примере выше) или при помощи StoryBoard.Begin метода в коде. При этом у каждой StoryBoard есть возможность подписаться на события начала и окончания и ряд других.

При использовании анимаций с изменяемыми параметрами, задаваемыми при помощи привязок (Binding), анимации необходимо перезапускать — в силу того, что необходимо регенерировать числовой ряд.

Элементы управления WPF

Традиционно для множества технологий, используемых в программировании, существует два способа создания собственных элементов управления:

- Композитные элементы управления - UserControl;
- Собственные элементы управления — Custom Controls.

Композитные элементы управления представляют собой набор совмещенных существующих в платформе элементов управления, связанных логически одной целью. Создание таких элементов не отличается ничем от создания собственных приложений, поэтому этот способ рассматриваться не будет.

Выбор базового класса

При создании собственных элементов управления имеются следующие возможности:

- Наследование от `UserControl` — это способ создания композитного элемента управления; такой способ создания собственных является традиционно наиболее простым, однако наиболее ограниченным: отсутствует поддержка стилизации элемента управления при помощи `DataTemplate` или `ControlTemplate`, код нагроможден в одном месте и может конфликтовать с другими элементами управления и другое.
- Наследование от `Control` — данный способ избавлен от недостатков предыдущего и позволяет (и поощряет) использовать шаблоны элементов управления, стили для декорирования и поддержки различных тем оформления элементов управления.
- Наследование от `FrameworkElement` — самый низкоуровневый способ реализации собственного элемента управления; в данном случае возможен вариант собственной реализации отображения (рендеринга) элемента управления на экране, таким образом имея наибольшую гибкость (и наибольшую сложность) в реализации.

Основой для содержания логики настройки элементов управления являются зависимые свойства (*dependency properties*). Они предоставляют следующую функциональность:

- Поддержку механизма привязок;
- Поддержку стилей, триггеров;
- Поддержку механизмов валидации, обновления значений, установки пороговых значений и другое;
- Поддержку анимаций;
- Использование динамических ресурсов в качестве источников данных для этих свойств;
- И другое.

Для создания собственного зависимого свойства, выполните следующее:

```
/// <summary>
/// Identifies the Value dependency property.
/// </summary>
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
        "Value", typeof(decimal), typeof(NumericUpDown),
        new FrameworkPropertyMetadata(MinValue, new
            PropertyChangedCallback(OnValueChanged),
            new
            CoerceValueCallback(CoerceValue)));


/// <summary>
/// Gets or sets the value assigned to the control.
/// </summary> public
decimal Value
{
    get { return (decimal)GetValue(ValueProperty); }      set {
        SetValue(ValueProperty, value); }
}

private static object CoerceValue(DependencyObject element, object value)
{
    decimal newValue = (decimal)value;
    NumericUpDown control = (NumericUpDown)element;      newValue =
        Math.Max(MinValue, Math.Min(MaxValue, newValue));

    return newValue;
```

```

}

private static void OnValueChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args) {
    NumericUpDown control = (NumericUpDown)obj;

    RoutedPropertyChangedEventArgs<decimal> e = new RoutedPropertyChangedEventArgs<decimal>(
        (decimal)args.OldValue, (decimal)args.NewValue, ValueChangedEvent);
    control.OnValueChanged(e); }

```

Для каждого зависимого свойства определяется статический регистратор, само свойство, функция вызываемая при изменении значения и функция корректировки значения (для установки порогового).

Помимо зависимых свойств, ключевым также является использование переадресуемых событий (Routed events). Прежде всего, это необходимо для поддержания общей концепции обработки событий в WPF, а также для поддержки механизмов EventTrigger/EventSetter. Для создания переадресуемого события, выполните следующее:

```

/// <summary>
/// Identifies the ValueChanged routed event.
/// </summary>
public static readonly RoutedEvent ValueChangedEvent = EventManager.RegisterRoutedEvent(
    "ValueChanged", RoutingStrategy.Bubble,
    typeof(RoutedPropertyChangedEventHandler<decimal>), typeof(NumericUpDown));

/// <summary>
/// Occurs when the Value property changes.
/// </summary>
public event RoutedPropertyChangedEventHandler<decimal> ValueChanged
{
    add { AddHandler(ValueChangedEventArgs, value); }      remove {
    RemoveHandler(ValueChangedEventArgs, value); } }

/// <summary>
/// Raises the ValueChanged event.
/// </summary>
/// <param name="args">Arguments associated with the ValueChanged event.</param> protected virtual void
OnValueChanged(RoutedEventArgs<decimal> args) {
    RaiseEvent(args);
}

```

Дополнительная информация по созданию собственных элементов управления может быть найдена по следующим ссылкам:

<http://msdn.microsoft.com/en-us/library/ee330302.aspx> <http://msdn.microsoft.com/en-us/library/ms752339.aspx>

Контрольные вопросы.

1. Что такое состояния на диаграмме состояний ?
2. Чем отличается псевдо-состояние терминирования от конечного псевдостатуса ?
3. Чем определяются переходы на диаграммах состояний ?
4. Как поддерживается анимация в технологии WPF?
5. Какие существуют варианты создания собственных элементов управления в технологии WPF ?

ЛАБОРАТОРНАЯ РАБОТА № 11.

Диаграмма последовательностей. Реализация сетевого взаимодействия приложений

Задание.

1. Разработать диаграмму последовательностей.
2. Разработать интерфейс взаимодействий между клиентской и серверной частью приложения, разрабатываемого в пределах лабораторного цикла.
3. Разработать логику работы серверной части приложения и продемонстрировать успешную обработку клиентских запросов.

Краткие теоретические сведения.

Диаграмма последовательностей

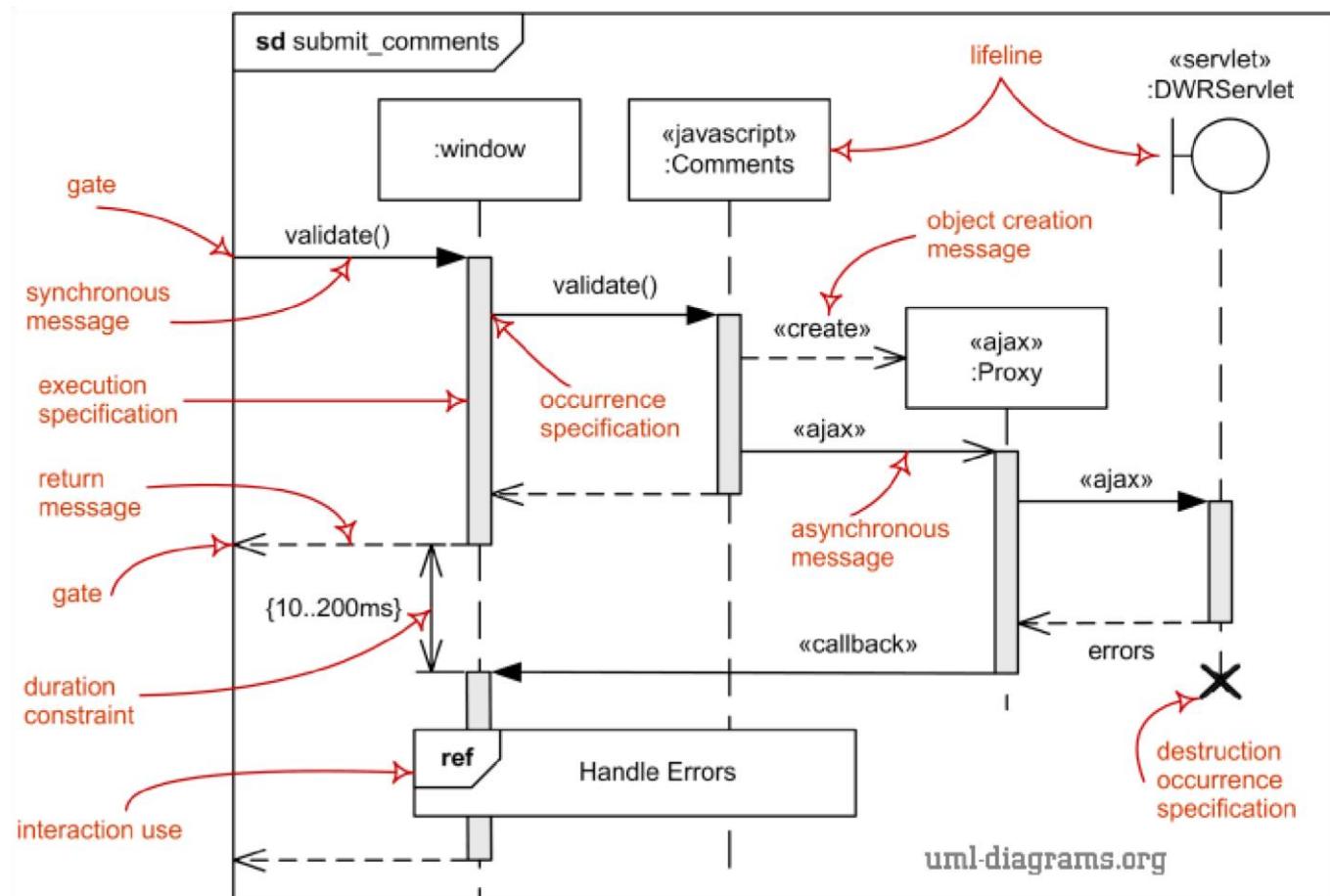


Диаграмма последовательностей — наиболее используемая из диаграмм взаимодействия. Она представляет взгляд на разрабатываемую систему с точки зрения взаимодействия ее компонентов при помощи сообщений в упорядоченной манере.

По обыкновению при помощи диаграммы последовательностей моделируют следующее:

1. Сценарии использования. Сценарий использования — потенциальный способ использования системы; возможно с указанием альтернативных путей прохода по системе. В данном случае диаграмма может быть связана с диаграммой вариантов использования (в том смысле, что описывает один или несколько вариантов использования одновременно).
2. Логику методов — описание принципа работы сложных методов и функций в виде последовательностей вызовов.

3. Логику сервисов/служб — описание протоколов взаимодействия сторонними системами/службами/сервисами.

Ключевые элементы диаграмм взаимодействия

Линия жизни (lifeline) — отображает длительность участия конкретного объекта или элемента диаграммы последовательностей в процессе взаимодействия с другими элементами.

При этом элементы могут быть следующих типов:

1. Действующие лица (actors) — в случае, если необходимо изобразить взаимодействие действующих лиц с системой; в особенности в тех случаях, когда действующее лицо инициирует некоторый процесс (путем нажатия кнопки на визуальной форме, например);
2. Компоненты — отдельные компоненты системы в случае, если показывается на достаточно высоком уровне процесс работы системы;
3. Объекты конкретных типов (классов) — в случае, если предоставляется низкоуровневая реализация отдельных методов или функций;
4. Интерфейсы и службы — в случае, если проектируется взаимодействие внешними компонентами.

Линия жизни состоит из трех частей: «шапки», собственно линии жизни и терминаатора. Шапка описывает, что за объект взаимодействует с другими объектами. Линия жизни представляет собой пунктирную линию и показывает длительность жизненного цикла объекта. Терминатор показывает, когда жизненный цикл объекта обрывается (происходит освобождение объекта из памяти, например).

На линии жизни также отражены периоды активности взаимодействующего объекта при помощи прямоугольника — execution specification. На протяжении этого участка объект активен и обменивается сообщениями с другими элементами диаграммы.

Ключевым понятием также является сообщение — передаваемый сигнал управления (вызов метода, передаваемое сообщение по сети, возникновение некоторого события и др.). Различают синхронные, асинхронные, возвратные, рефлексивные сообщения. Кроме того, различают узко-специализированные сообщения: создания элементов («create»), удаления элементов («destory»). Дополнительно, к сообщениям может добавляться произвольный комментарий.

Возвратные сообщения (return message) кодируются при помощи пунктирной стрелки. Возвратные сообщения используются для возврата значений из соответствующих методов или результатов выполнения действий/отработки событий.

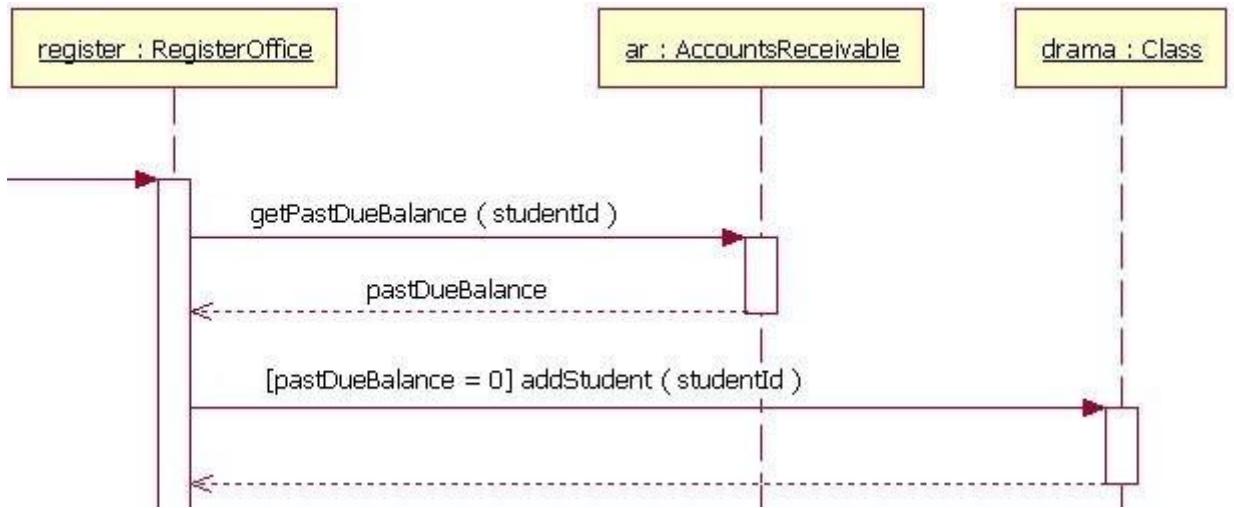
Синхронные сообщения используются для спецификации вызовов функций/методов/обработчиков у других элементов диаграммы, которые происходят последовательно по ходу протекания взаимодействия. Они кодируются при помощи сплошной линии с закрашенной стрелкой на конце.

Асинхронные сообщения используются для спецификации действий, вызываемых асинхронно на других элементах диаграммы. Они кодируются при помощи сплошной линии с незакрашенной стрелкой на конце.

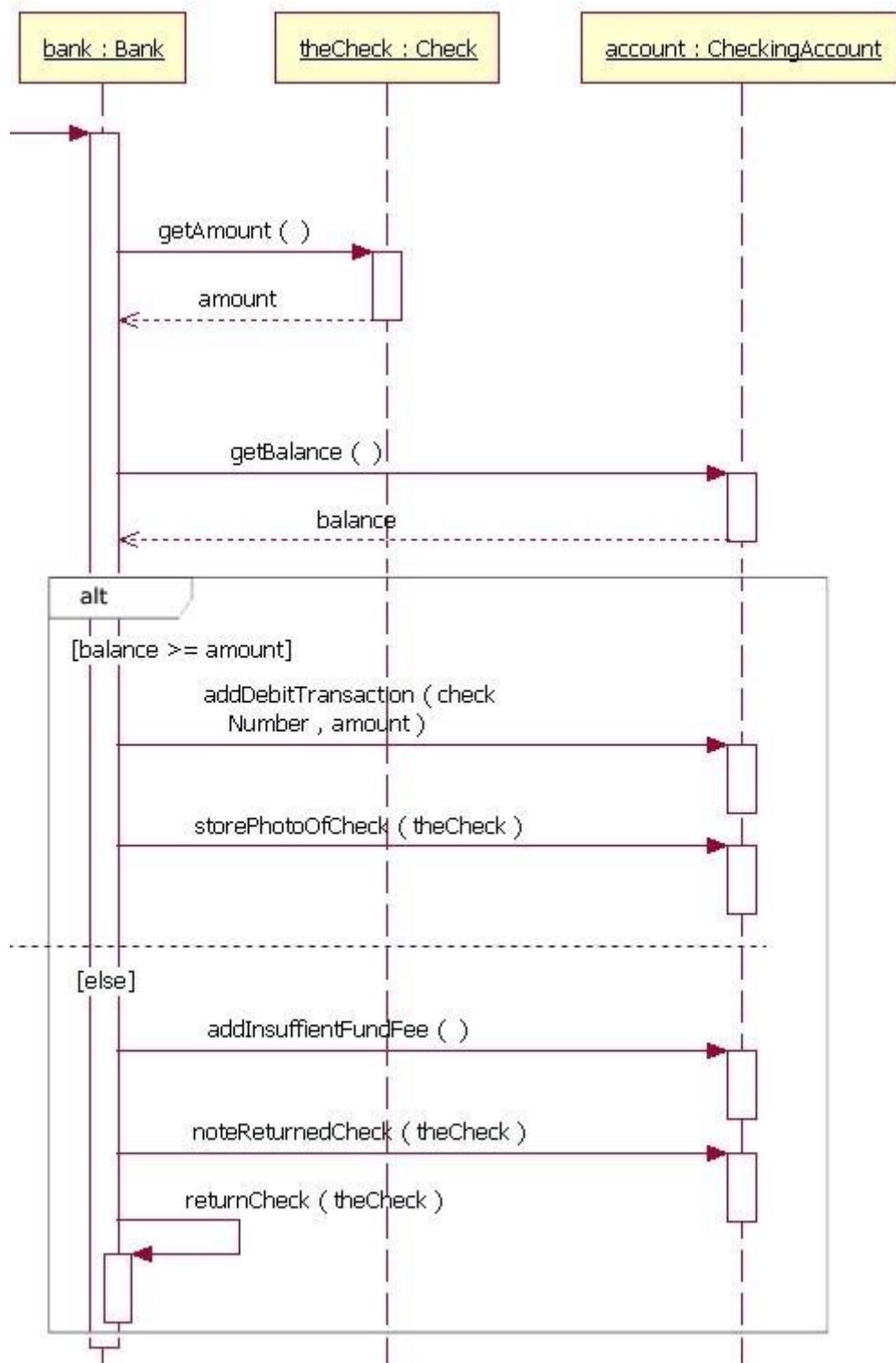
Рефлексивные сообщения — происходят от элемента и направлены в себя самого. Используются для спецификации, что конкретные методы/обработчики принадлежат этому же элементу и раскрывают внутренние детали реализации элемента диаграммы.

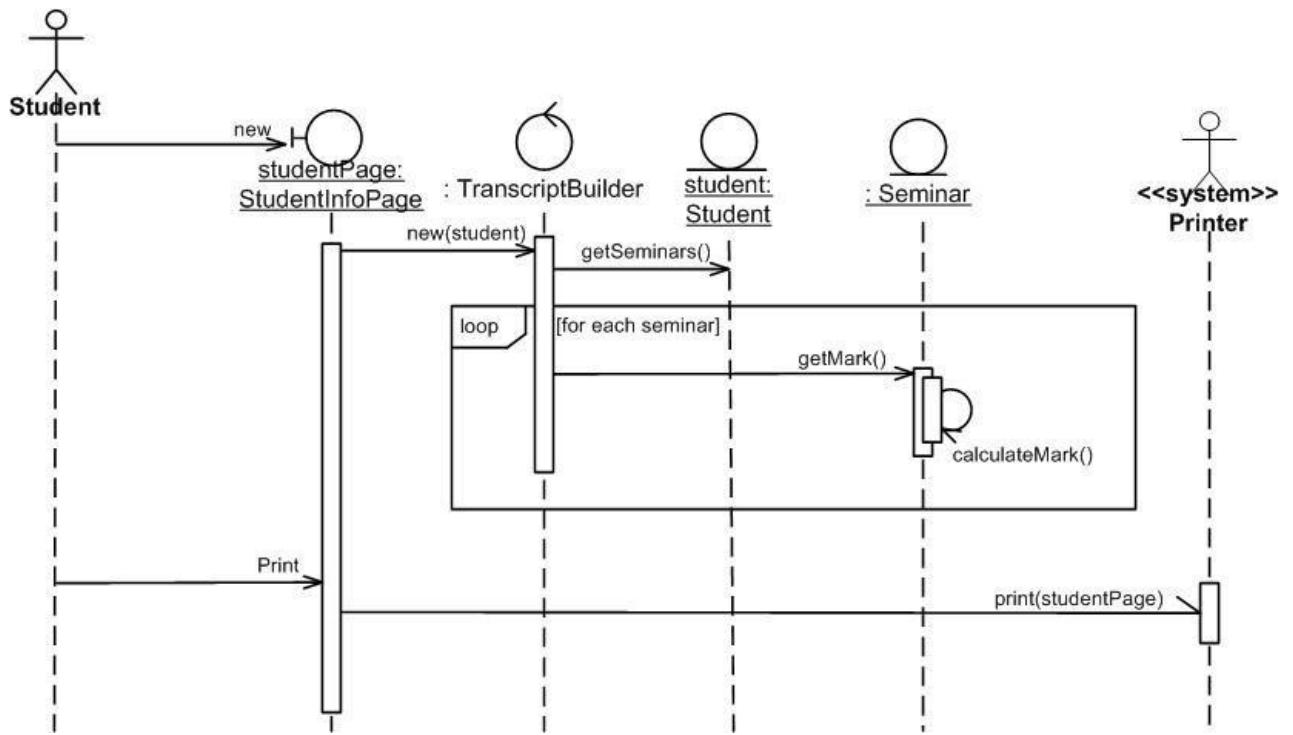
Обратные сообщение (callback message) используются для спецификации протокола взаимодействия между элементами.

Помимо этого, сообщения могут также сообщать о конкретных параметрах, передаваемых в методы, объектах, используемых во взаимодействия и другом. Дополнительно, могут указываться условия, при которых сообщение будет послано:



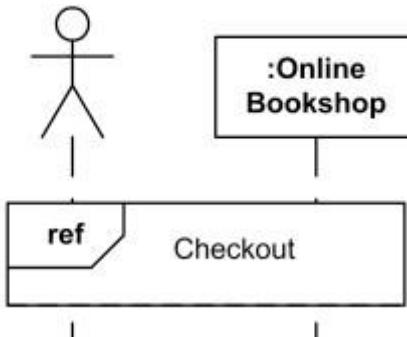
В случае, если необходимо изобразить более сложную условную логику на диаграммах взаимодействия, также доступны конструкции «если..то» (if..else) и циклический операторы.



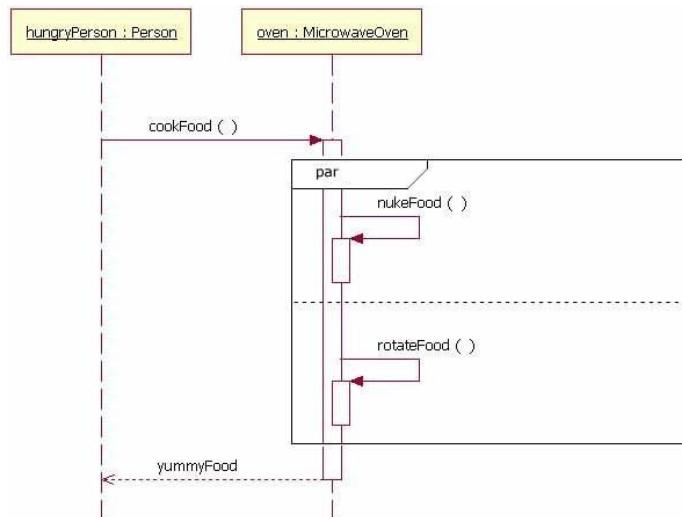


При необходимости можно также включать отдельные части других диаграмм для более компактного и удобного просмотра:

:Web Customer



Отображение параллельных процессов на диаграммах взаимодействий выполняется так же, как и на диаграммах состояний — при помощи пунктирной линии, отделяющей параллельные пути.



Windows Communication Foundation

Для разработки сетевого взаимодействия приложения в .NET Framework доступны следующие технологии:

- Sockets;
- TcpConnection/TcpListener;-
- .NET Remoting; - WCF.

Исторически первой и наиболее низкоуровневой технологией для реализации взаимодействия приложений по сети является технология с применением сокетов. Сокет — это сетевой интерфейс, состоящий из ip-адреса и порта. Данная технология позволяет подсоединяться к удаленным сокетам по стеку TCP/IP и передавать данные в виде массива байтов.

За ними появилась технология на основе TcpListener/TcpClient, которая содержала более высокоуровневые обвязки вокруг технологии сокетов. Данная технология позволяла различать между соединениями, использовать прокси-соединения и прочие дополнительные возможности. Недостатком их являлось использование «сырых» данных для передачи — массив байтов на передачу и прием.

Попыткой исправить эту ситуацию является появление на рынке технологии .NET Remoting, которая позволяла специфицировать объект взаимодействия (набор методов, вызываемых с клиента на сервере, вместе с возвращаемыми значениями). При этом экземпляр данного объекта хранился как на сервере (для обработки запросов), так и на клиенте (для вызова сервера). Все передаваемые данные сериализовались прозрачно для программиста и передавались по TCP каналам. Однако данная технология была достаточно громоздкой в настройке, потребляла слишком много ресурсов и была негибкой — невозможно было использовать многопоточное взаимодействие, обеспечивать безопасность канала связи и другое.

Для решения этих проблем команда разработчиков .NET Framework представила новый фреймворк для разработки сетевого взаимодействия приложения — WCF (Windows Communication Framework). Он был призван решить все проблемы предшественников и является универсальным фреймворком для построения сетевых приложений. Универсальность заключается в его гибкости и простоте использования.

Ключевые концепции, заложенные в WCF:

1. Ориентированность на сервисы — сервис как независимая единица обработки запросов;
2. Кросс-платформенное взаимодействие — поддержка множества различных стандартов (WS-* и другие);
3. Разнородные шаблоны сообщений — обмен сообщениями может происходить по-разному: запрос-ответ (request-reply), односторонние сообщения, асинхронные сообщения, дуплексные сообщения и др.
4. Поддержка метаданных для автоматического обнаружения и подключения к сервисам;
5. Безопасность — поддержка различных способов обеспечения безопасности канала связи — безопасность на уровне транспорта и на уровне сообщений;
6. Разнообразные способы транспортировки сообщений;
7. Надежная доставка;
8. Расширяемость.

Ключевые концепции WCF

Так называемая «азбука WCF» (WCF ABC's) — адреса (addresses), привязки (bindings), контракты (contracts).

Адреса представляют собой удаленные точки для подключения. Адрес представляет собой строку URI (Universal resource identifier), которая включает в себя протокол взаимодействия (http, https, tcp, ipc и другие), адрес ресурса (доменное имя, ip-адрес, название канала, др.), порт и название службы. Например:

HTTPS://cohownery:8005/ServiceModelSamples/CalculatorService

Контракты представляют собой спецификации или протоколы взаимодействия. Контракты описывают все возможные действия, которые могут быть выполнены над сервисом.

Привязки определяют способ взаимодействия сервиса с окружающим миром — стандарт общения, уровень безопасности, параметры передачи данных (максимальный размер пакета данных, максимальное количество передаваемых объектов, таймаут и другое).

Рассмотрим простейший пример.

```
[ServiceContract(Namespace = "http://Microsoft.ServiceModel.Samples")] public  
interface ICalculator  
{  
    [OperationContract]  
    double Add(double n1, double n2);  
    [OperationContract] double  
    Subtract(double n1, double n2);  
    [OperationContract] double  
    Multiply(double n1, double n2);  
    [OperationContract]  
    double Divide(double n1, double n2);  
}
```

Таким образом определяется контракт сервиса. Он помечается атрибутом ServiceContract, в простейшем случае без параметров. Дополнительно, сервисные контракты могут указывать следующее:

- Ответный контракт (callback service contract);- Поддержку сессий (Session mode);
- Уровень безопасности (Protection Level).

Реализация данного контракта представлена ниже:

```
public class CalculatorService : ICalculator  
{  
    public double Add(double n1, double n2)  
    {  
        double result =  
        n1 + n2;  
        Console.WriteLine("Received Add({0},{1})", n1, n2);  
        // Code added to write output to the console window.  
        Console.WriteLine("Return: {0}", result);  
        return result;  
    }  
}
```

```

public double Subtract(double n1, double n2)
{
    double result =
n1 - n2;
    Console.WriteLine("Received Subtract({0},{1})", n1, n2);
    Console.WriteLine("Return: {0}", result);
return result;
}

public double Multiply(double n1, double n2)
{
    double result =
n1 * n2;
    Console.WriteLine("Received Multiply({0},{1})", n1, n2);
    Console.WriteLine("Return: {0}", result);
return result;
}

public double Divide(double n1, double n2)
{
    double result =
n1 / n2;
    Console.WriteLine("Received Divide({0},{1})", n1, n2);
    Console.WriteLine("Return: {0}", result);
return result;
}

```

Как видим, для реализации контракта необходимо просто создать класс, реализующий интерфейс контракта. К нему дополнительно можно добавить ServiceBehavior атрибут, указывающий дополнительные настройки сервиса, такие как:

- Режим многопоточности;
- Количество объектов для клиентов (один на всех, по одному на каждую сессию, по одному на каждый запрос); - Уровень транзакций и другие.

Созданные сервисы могут быть запущены в трех местах: - В приложении локально;

- В службе Internet Information Services (IIS) Windows; - Как служба Windows.

Для запуска сервиса локально можно воспользоваться двумя путями: - Автоматический запуск при помощи конфигурационного файла; - Запуск из исходных кодов.

Для запуска из исходных кодов необходимо создать и наполнить объект ServiceHost, который будет поддерживать работу созданного сервиса:

```

using (ServiceHost host = new ServiceHost(typeof(HelloWorldService), baseAddress))
{
    // Enable metadata publishing.
    ServiceMetadataBehavior smb = new ServiceMetadataBehavior();      smb.HttpGetEnabled = true;
    smb.MetadataExporter.PolicyVersion = PolicyVersion.Policy15;      host.Description.Behaviors.Add(smb);

    // Open the ServiceHost to start listening for messages. Since

```

```

// no endpoints are explicitly configured, the runtime will create      // one endpoint
per base address for each service contract implemented      // by the service.
host.Open();

Console.WriteLine("The service is ready at {0}", baseAddress);
Console.WriteLine("Press <Enter> to stop the service.");      Console.ReadLine();

// Close the ServiceHost.
host.Close(); }

```

Для конфигурации при помощи конфигурационных файлов:

```

<?xml version="1.0" encoding="utf-8" ?>
<!-- Copyright © Microsoft Corporation. All Rights Reserved. -->
<configuration>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="CalculatorServiceBehavior">
                    <serviceMetadata httpGetEnabled="True"/>
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
            <service name="Microsoft.Samples.GettingStarted.CalculatorService"
behaviorConfiguration="CalculatorServiceBehavior">
                <endpoint address="" binding="basicHttpBinding" contract="ICalculator"/>
                <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
            </service>
        </services>
    </system.serviceModel>
</configuration>

```

Для создания клиента необходимо создать клиентский proxy объект. Это делается автоматически одним из двух способов:

1. Из командной строки при помощи утилиты svchostsvutil myservice.dll
2. Путем добавления сервисной ссылки в Microsoft Visual Studio

Контрольные вопросы:

1. Каковы сценарии использования диаграмм последовательностей ? 2. Какие элементы могут быть изображены на диаграмме последовательностей ?
3. Когда используется терминатор на диаграммах последовательностей ?
4. Что представляют собой сервисы WCF ?
5. Что представляют собой контракты WCF (данных, операций, ошибок, сервисов) ?

ЛАБОРАТОРНАЯ РАБОТА № 12.

Стилистическая проверка исходных кодов

Задание:

1. Сформировать набор правил стилистического анализа.
2. Произвести стилистический анализ исходных кодов.
3. Убрать все ошибки стилистического анализа.

Краткие теоретические сведения.

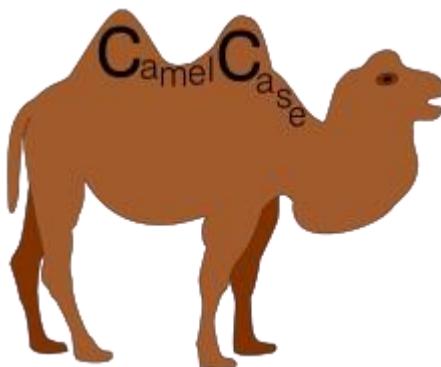
Стандарты стилистического оформления кода

Соглашения о стилистическом оформлении кода (coding conventions) — набор правил, определяющих стиль программирования, и по обыкновению касаются файловой организации исходного кода, стандартов именования, отступов, комментариев и многого другого.

Данные соглашения важны для поддержки структурного качества исходного кода. Стилистически однообразный код значительно легче поддерживать, читать, проводить поиск. Он также упрощает проведение смотров исходного кода (code review), поскольку весь код заранее приведен к конкретному стилю (уменьшение количества споров о том или ином стиле написания кода и перевод акцента на непосредственную функциональность кода).

Как правило, такие соглашения формализуются в виде документа в репозитории (readme, style, coding_guidelines и прочее), и им следуют все члены команды безоговорочно. Изначально может быть некоторое сопротивление в принятии этих соглашений — поскольку каждый программист имеет свой стиль программирования.

Исторически первым стандартом кодирования была т. н. «Венгерская Нотация» (Hungarian notation). Венгерская нотация предполагала именование переменных таким образом, чтобы в названии переменной присутствовал префикс, определяющий тип данных или некоторый специфический атрибут (например, префикс `t_` означал, что эта переменная является `private` переменной класса). С точки зрения именования появилась масса вариантов — кемел кейсинг (CamelCase), именование _с_нижними_подчеркиваниями, ИСПОЛЬЗОВАНИЕ-ВЕРХНЕГО-РЕГИСТРА и другие.



Stylecop

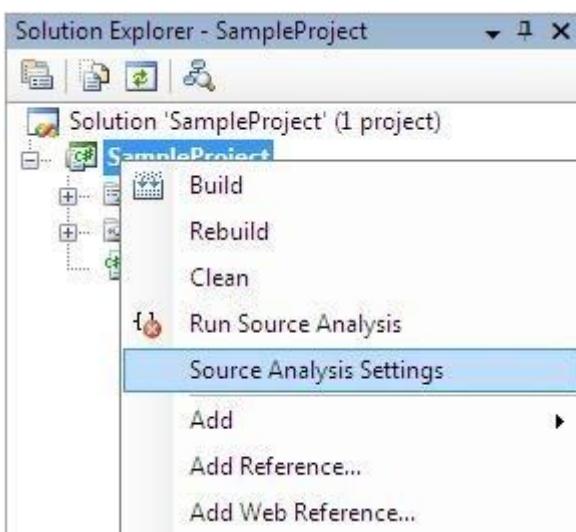
Stylecop представляет собой средство статического анализа исходного кода на предмет соответствия набору правил стилистического оформления кода. Сам по себе Stylecop содержит набор правил, которые отнесены в несколько категорий:

- Правила документирования кода (documentation)
- Правила расположения кода (layout)
- Правила, касающиеся поддерживаемости кода (maintainability)
- Правила именования (naming)
- Правила порядка размещения (ordering)
- Правила читаемости (readability)
- Правила выставления пробелов/отступов (spacing)

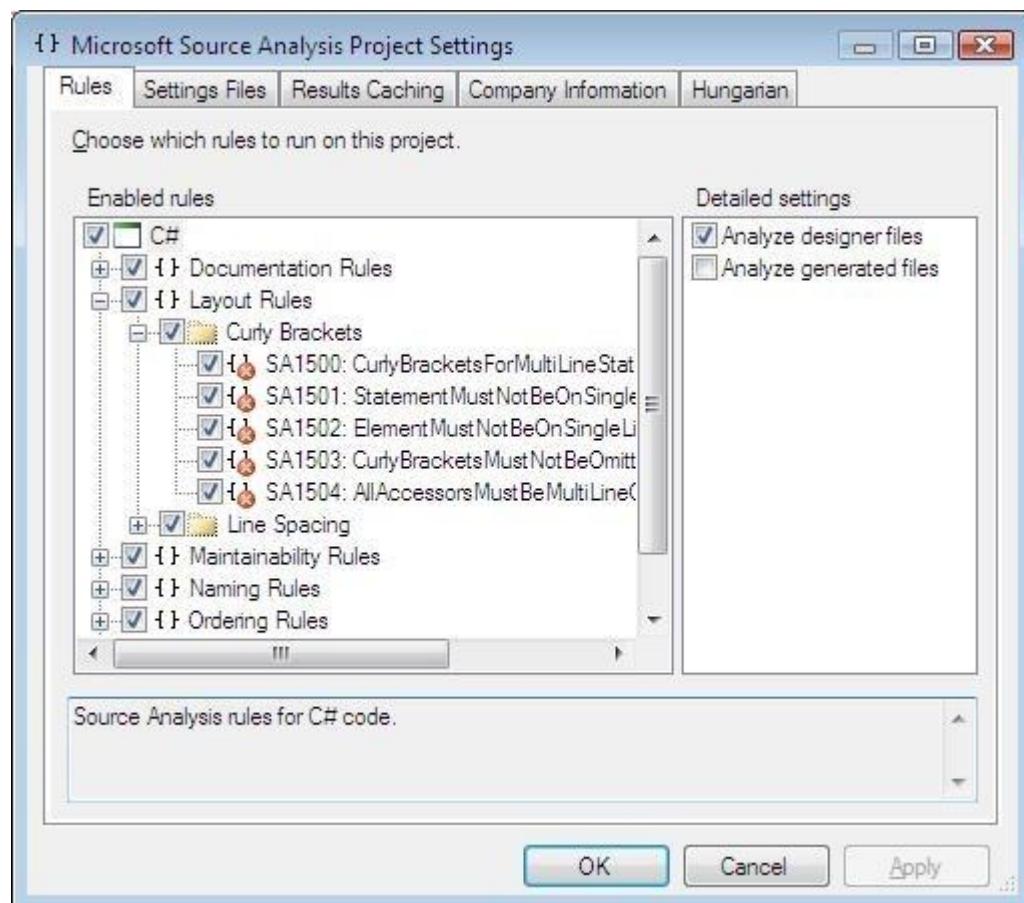
Также можно добавлять собственные правила, если набор правил, предоставляемый StyleCop, кажется недостаточным.

Например, удобными правилами для определения шаблонов именования является StyleCopPlus: <http://stylecopplus.codeplex.com/>

При работе с StyleCop необходимо прежде всего выбрать необходимый набор применимых правил. Stylecop изначально содержит набор правил, конфликтующих между собой — для поддержки различных стилей и команд разработки. Необходимо выбрать такой набор правил, который кажется наиболее приемлемым для команды. Правила определяются при помощи окна редактирования настроек StyleCop в Visual Studio. Там же можно посмотреть словесные описания правил.



Необходимо отметить, что StyleCop также поддерживает исключения именования (в случае, если есть зарезервированные слова в исходном коде, такие как ID и др.) и возможность отключения анализа генерируемых файлов (файлы .Designer.Cs или .g.cs генерируются автоматически и смысла их проверять нет).



Для проведения стилистического анализа необходимо нажать в контекстном меню кнопку Run Source Analysis (запустить анализ исходных кодов). Результаты анализа выводятся в качестве предупреждений (warning) либо ошибок (error) в окне Error List среды разработки Microsoft Visual Studio.

Контрольные вопросы.

1. Что представляет собой венгерская нотация ?
2. Какие преимущества использования соглашений о стиле исходного кода ?
3. Какие недостатки использования соглашений о стиле исходного кода ?
4. Какие правила именования Вам известны ?
5. Какие правила формирования отступов Вам известны ?

Лабораторная работа № 13.

Профилирование исходного кода

Задание.

1. Произвести инструментальное профилирование исходного кода.
2. Произвести сэмплированное профилирование исходного кода.
3. Найти узкое место системы и оптимизировать его.
4. Произвести профилирование взаимодействий с базой данных.
5. Произвести профилирование отрисовки визуального интерфейса WPF.

Краткие теоретические сведения.

Профилирование — процесс определения ресурсных показателей работы программного обеспечения, таких как объемы занимаемой памяти, загрузка ЦПУ, время обработки конкретных действий и другое. Профилирование как процесс исторически используется для выявления «узких мест» (bottlenecks) в программном обеспечении для повышения быстродействия либо для устранения утечек памяти (memory leaks).

Известная истина гласит, что в каждом приложении за 80% используемых ресурсов ответственны лишь 20% кода. Это свидетельствует против т. н. процесса микрооптимизации, когда при написании программы авторы сразу используют наиболее эффективные (и, как правило, наименее поддерживаемые) структуры/алгоритмы/подходы в программировании. Как правило, нет смысла оптимизировать на ранних этапах разработки программного обеспечения, поскольку характеристики нагрузок еще неизвестны.

Альтернативным способом написания программного обеспечения является оптимизация после первого (бета) запуска приложения и сбора данных о фактических параметрах нагрузок. В данном случае для оптимизации приложения необходимо использовать профилировщики.

Существует два класса профилировщиков:

- Профилировщики по процессорному времени;- Профилировщики по занимаемым объемам памяти.

Профилировщики также делятся, в зависимости от способа реализации, на следующие виды:

- Событийного типа; - Статистические; - Инструментальные.

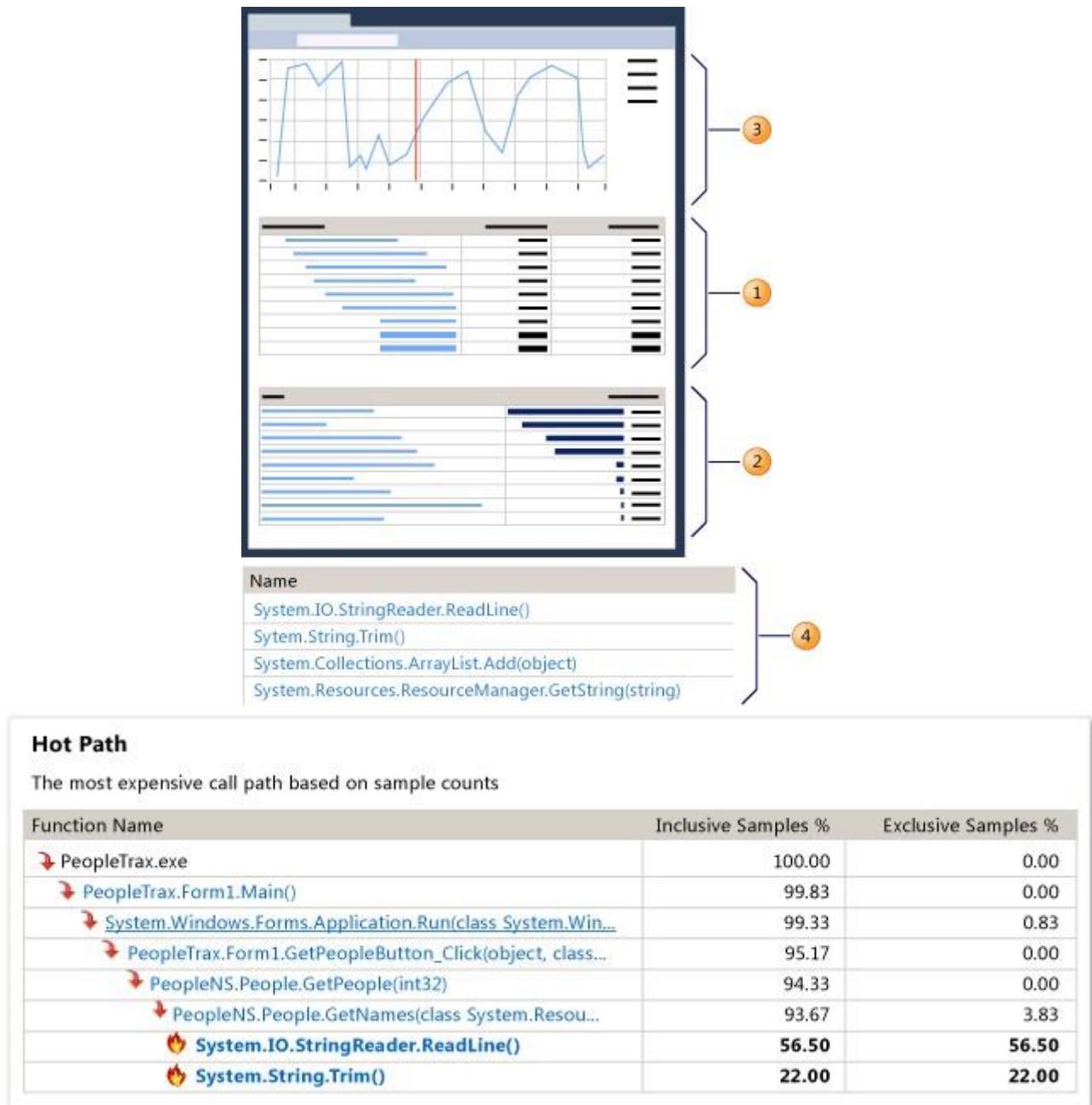
Профилировщики событийного типа внедряются в механизм запуска приложений (виртуальная машина Java, межязыковая среда выполнения CLR и пр.) и подписываются на разнообразные события уровня механизма выполнения программ (события создания/удаления потоков, события создания/удаления объектов, входа/выхода из контекста выполнения и др.). Такие профилировщики ограничиваются возможностями поддерживающего механизма.

Статистические профилировщики собирают данные о ходе выполнения программы благодаря программному счетчику (program counter) в определенные интервалы времени. Каждая единица данных называется sample. При этом такие профилировщики дают не точные данные (поскольку они собраны в результате проверки программного счетчика каждые n милисекунд), а скорее статистическую аппроксимацию. Однако, к преимуществам таких профилировщиков необходимо отнести практически ничтожное влияние на работу и быстродействие профилируемого приложения.

Инструментирующие профилировщики встраивают собственные участки кода для логирования некоторых параметров выполнения программного обеспечения, и на основе этих данных позднее строится график результатов. В случае проведения некачественного инструментирования программа

может перестать работать, и инструментирование наиболее сильно влияет на быстродействие приложения (замедляя его).

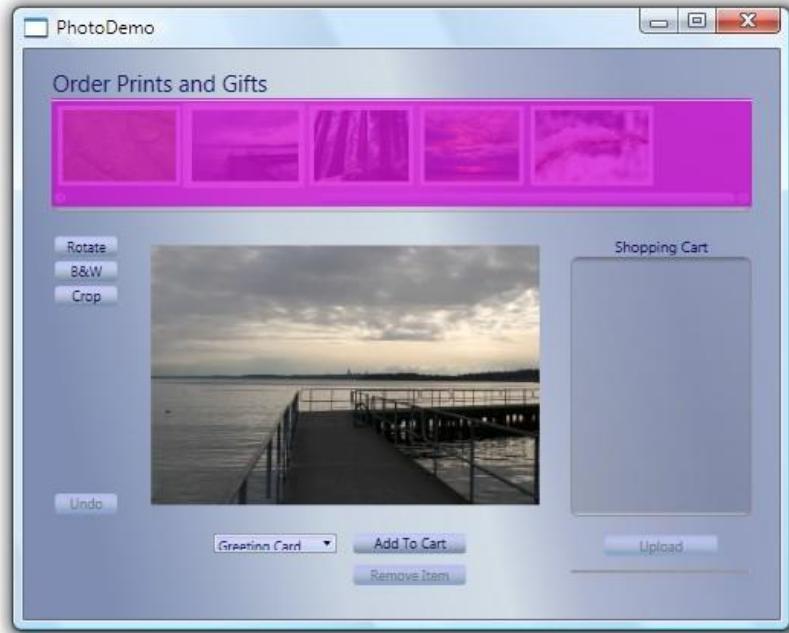
Результатом работы профилировщика является график использования различных ресурсов во времени, и дополнительная информация — построчный анализ исходного кода, какие участки занимали больше всего процессорного времени/ресурсов.



Для запуска профилирования в Microsoft Visual Studio необходимо выбрать пункт меню Tools → Performance Tools → Performance Wizard. Мастер настройки профилировщика позволит выбрать необходимые настройки профилировщика (какие приложения профилировать, какой тип профилирования использовать, какие данные собирать). После этого необходимо запустить сессию профилирования из вкладки Profiling Tools.

Дополнительно, в наборе Microsoft Visual Studio есть средство профилирования визуального интерфейса WPF. Поскольку WPF использует собственные технологии отрисовки (рендеринга)

приложений, для повышения быстродействия рекомендуется уменьшить количество прорисовок экрана. Средство Perforator позволяет просмотреть «грязные» участки приложения, которые были перерисованы с прошлого кадра, а также участки, которые постоянно перерисовываются либо отрисовываются без использования аппаратного ускорения. Данное средство позволяет выявить тормозящие участки визуального интерфейса и оптимизировать их для повышения быстродействия приложения.



Контрольные вопросы.

1. Что такое профилирование ?
2. Какие виды профилировщиков существуют ?
3. Как определить узкие места в приложении при помощи профилировщика ?
4. На что важно обращать внимание при профилировании визуального интерфейса WPF приложений ?

ЛАБОРАТОРНАЯ РАБОТА № 14

Статический анализ программного обеспечения

Задание:

1. Использовать максимальный набор правил статического анализа исходного кода.
2. Выполнить статический анализ исходного кода при помощи FxCop.
3. Исправить выявленные ошибки.

Краткие теоретические сведения.

Статический анализ программного обеспечения выполняется по скомпилированным исходным кодам в виде бинарных файлов (.dll, .exe) без непосредственного запуска приложений. Такой анализ производится при помощи автоматических средств анализа и, как правило, ставит своей целью одну из следующих:

- Выявление участков небезопасного кода — возможность встраивания SQLинъекций или других способов взлома;
- Выявление дефективных участков кода — участки кода, содержащие синтаксические либо логические ошибки (при помощи разнообразных эвристик), например, для выявления всех точек NullReferenceException;
- Выявление неоптимальных участков кода;
- Выявление несоответствия лучшим практикам программирования и многое другое.

Известны инструменты статического анализа программ, выявляющие дефекты в модулях и драйверах ядра Linux, прогнозирующие участки кода, которые в будущем будут содержать наибольшее количество дефектов, предлагающие альтернативные способы решения некоторых проблем (более оптимальные) и другие.

Проблема инструментов статического анализа состоит в том, что математически доказана невозможность выявления всех ошибок программного обеспечения для любого Тьюринг-полного языка программирования.

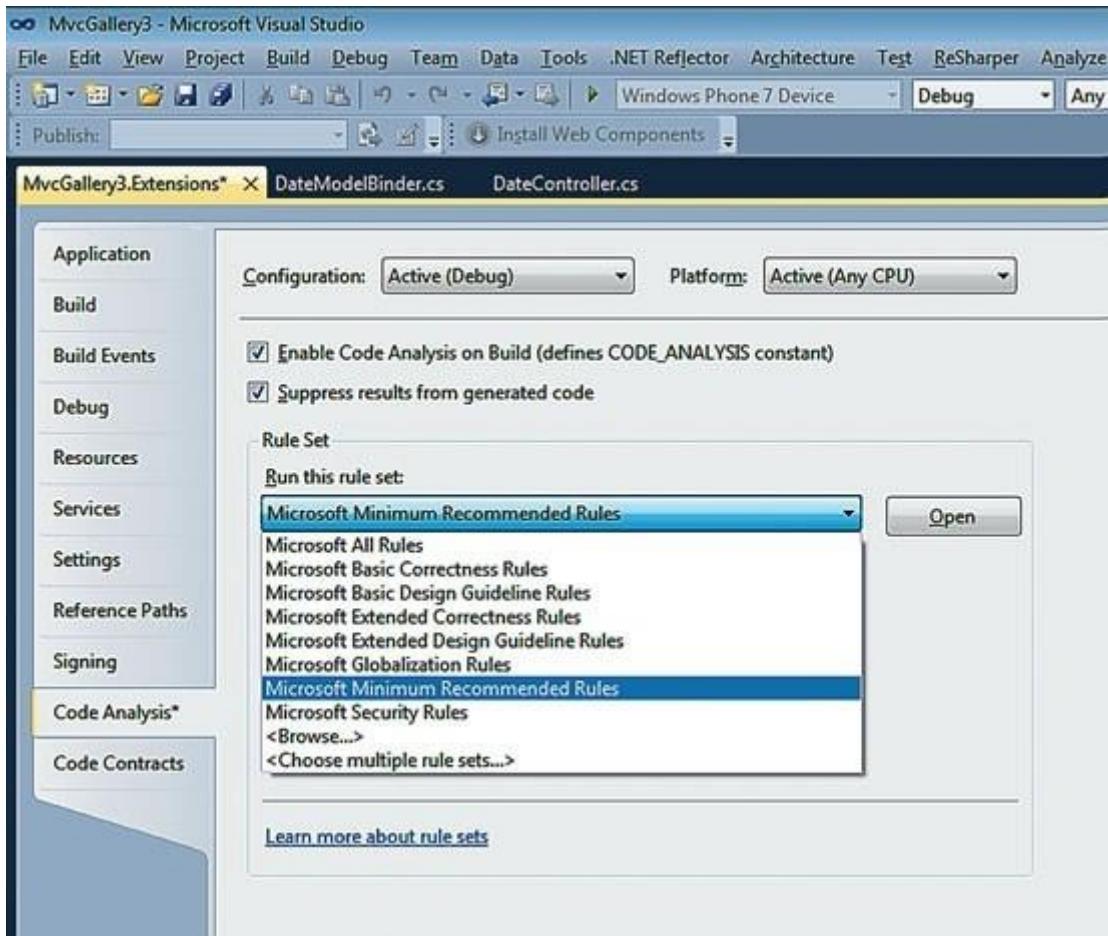
Существуют следующие способы реализации инструментов статического анализа:

- При помощи проверки моделей (model checking) — делается вывод о существовании конечного числа состояний у любой программы, и строится граф переходов между состояниями;
- При помощи анализа протекания данных (data flow analysis) — анализируется путь прохождения отдельных объектов по участкам исходного кода; - Абстрактная интерпретация — моделирование графа состояний, где каждый переход — это выполнение некоторого выражения в программном коде; - Использование шаблонов (templates);
- Использование проверок (assertions).

FxCop

FxCop — инструмент статического анализа программного обеспечения на соответствие лучшим практикам составления программного кода. Подобно StyleCop, он состоит из множества правил,

определяющие корректные пути написания программного обеспечения. Данное средство встроено в среду разработки Microsoft Visual Studio.



| Description | File | Line | Column | Project |
|---|-------------------------|------|--------|------------------------|
| CA1004 : Microsoft.Design : Change the type or parameter 'baseuri' or method 'PageHelper.Pager(this)' from string to System.Uri, or provide an overload of 'PageHelper.Pager(this HtmlHelper, string, int, int, string, int, object)', that allows 'baseUri' to be passed as a System.Uri object. | Pager.cs | 27 | | MvcGallery3.Extensions |
| CA1026 : Microsoft.Design : Replace method 'SimpleGridHelper.SimpleGrid<T>(this HtmlHelper, ICollection<T>, IList<IDataTransformer>)' with an overload that supplies all default arguments. | Grid.cs | 12 | | MvcGallery3.Extensions |
| CA1801 : Microsoft.Usage : Parameter 'helper' of 'SimpleGridHelper.SimpleGrid<T>(this HtmlHelper, ICollection<T>, IList<IDataTransformer>)' is never used. Remove the parameter or use it in the method body. | Grid.cs | 12 | | MvcGallery3.Extensions |
| CA1801 : Microsoft.Usage : Parameter 'transformers' of 'SimpleGridHelper.SimpleGrid<T>(this HtmlHelper, ICollection<T>, IList<IDataTransformer>)' is never used. Remove the parameter or use it in the method body. | Grid.cs | 12 | | MvcGallery3.Extensions |
| CA1062 : Microsoft.Design : In externally visible method 'SimpleGridHelper.SimpleGrid<T>(this HtmlHelper, ICollection<T>, IList<IDataTransformer>)', validate parameter 'dataSource' before using it. | Grid.cs | 25 | | MvcGallery3.Extensions |
| CA1801 : Microsoft.Usage : Parameter 'helper' of 'SubmitButtonHelper.Submit(this HtmlHelper, string, string, object)' is never used. Remove the parameter or use it in the method body. | Submit.cs | 22 | | MvcGallery3.Extensions |
| CA1019 : Microsoft.Design : Add a public read-only property accessor for positional argument 'transformerType' of Attribute 'TransformerAttribute'. | TransformerAttribute.cs | 6 | | MvcGallery3.Extensions |

Как и в StyleCop, возможен выбор правил проверки (в данном случае, однако, выбор лишь показывает степень жесткости проверки, нежели подбор желаемых параметров); ошибки анализа выводятся в окно списка ошибок Visual Studio. Каждая ошибка документирована и категоризирована на сайте MSDN — причина возникновения, способы устранения, примеры.

<http://msdn.microsoft.com/en-us/library/ee1hzekz.aspx>

Контрольные вопросы.

1. Зачем применяются инструменты статического анализа ?
2. Какие виды статического анализа известны ?

3. В чем заключается главная проблема инструментов статического анализа ?
4. Какие перспективы использования инструментов статического анализа ?
5. Какие категории встроенного анализатора Visual Studio вам известны ?

ЛАБОРАТОРНАЯ РАБОТА № 15.

Управление качеством программного продукта. Метрики программного обеспечения

Задание.

1. Ознакомиться с краткими теоретическими сведениями.
2. Составить метрики программного кода для собственного проекта при помощи продукта Ndepend.
3. Преобразовать наиболее сложные и громоздкие участки кода в более простую форму, повторно построить метрики программного кода и провести сравнение с предыдущими результатами.

Краткие теоретические сведения.

Метрики программного обеспечения — количественные показатели некоторой качественной стороны ПО. Различают множество различных метрик, в том числе и по виду: метрики качества ПО; метрики сложности программного кода; стоимостные метрики ПО; метрики производительности и др. Соответственно, метрики — попытка ввести количественные показатели для процесса разработки программ (точно так же, как такие показатели существуют в других сферах и отраслях).

Целью введения метрик разнообразны и коррелируют с видами метрик; метрики могут вводиться для вычисления конечной стоимости программного продукта (очевидно, по определенным критериям — трудозатратам, например), прогнозирования дат будущих релизов и вообще планирования работ, поднятия или поддержания качества программных кодов на должном уровне и др.

Необходимо отметить, что метрики программного кода не распространены в должной мере среди разработчиков программного обеспечения и множество из них остаются лишь попытками академических групп внести порядок в процесс разработки обеспечения. Однако результат разработки метрик нельзя назвать бесплодным — существуют метрики для проведения статического и динамического анализа программного кода (с целью выявления и исправления ошибок), метрики для прогнозирования ошибок и другие. Такие метрики используются в некоторых современных проектах, а также при разработке системных приложений (драйверов и системных модулей), поскольку стоимости ошибок в таких модулях — колоссальны.

В данной лабораторной работе мы ознакомимся с метриками, представляемыми проектом Ndepend.

Метрики приложений

В Ndepend предоставляется 12 метрик, высчитываемых по приложению: - Количество строк кода (Lines of code, LoC) — наиболее часто используется для оценки объема программного продукта; в более ранние годы использовалось для подсчета стоимости продукта либо калькуляции зарплаты программиста;

- Количество строк с комментариями;
- Процент покрытия кода — процент кода, покрытого модульными тестами; используется для оценки качества программного продукта; продукт, покрытый тестами полностью считается более стабильным;
- Количество инструкций IL;

- Количество сборок — подсчет количества физических файлов, занимаемых приложением;
- Количество областей именования;
- Количество типов;
- Количество методов;
- Количество полей;
- Процент комментариев — отношения количества комментариев к количеству строк исходного кода;
- Количество строк, покрытых тестами; - Количество строк, непокрытых тестами.

По названиям можно догадаться, что большинство метрик не имеет прямого применения в процессе разработки; основной упор в средстве NDepend делается на возможность построения собственных метрик из уже определенных; например, можно посчитать среднее количество типов на сборку или среднее количество методов на тип. В случае, если эти значения будут большими — можно думать о необходимости выделения более мелких подсистем или модулей, а также о том, что не соблюдаются принципы SOLID (а точнее, что каждый компонент должен решать лишь одну поставленную задачу).

По количеству строк кода, непокрытых тестами, можно подсчитывать общие объемы работ, которые еще предстоит выполнить, в то время как процент покрытия тестами представляет общую картину по проекту.

Также предоставляется 18 метрик по сборкам (представлены лишь неповторяющиеся):

- Внешняя связность — количество типов снаружи сборки, которые ссылаются на типы внутри сборки;
- Внутренняя связность — количество типов внутри сборки, которые ссылаются на типы снаружи сборки;
- Реляционная связность — показатель количества связей по типу; количество связей по типу должно быть достаточно высоким внутри сборки (поскольку типы выполняют одну функцию) и достаточно низко для связей с внешними компонентами (для выделения чистого интерфейса доступа к модулю); - Нестабильность — показатель склонности к изменению, в диапазоне от 0 до 1.
- Абстрактность — показатель количества абстрактных типов к общему количеству типов, в диапазоне от 0 до 1.
- Расстояние от основной последовательности — показатель баланса между абстрактностью и стабильностью. Идеальные сборки — полностью абстрактны и стабильны, либо полностью конкретны и нестабильны.

Метрики по сборкам позволяют выявить общие архитектурные сложности в продукте. Высокий показатель связности (внутренней и внешней) может служить признаком того, что код очень трудно изменять и поддерживать — любое изменение приводит к ошибкам во множестве частей продукта; в то же время слишком низкая связность сигнализирует, что модуль — излишний, не несет достаточной смысловой нагрузки и может быть влит в другой модуль.

Различают также 13 метрик по пространствам именования, которые во многом повторяют метрики по сборкам — с той лишь разницей, что считаются не для всей сборки, а для пространства имен.

В NDepend представлено 22 метрики по типам (представлены лишь новые): - Количество реализованных интерфейсов — большое количество реализованных интерфейсов как правило указывает, что модули более не выполняют одной конкретной функции и перегружены

функционалом; следует выделить либо отдельные типы либо абстрактный базовый класс с реализациями интерфейсов;

- Ранг — некоторый показатель количества связей с другими объектами; типы с большим значением ранга должны тестироваться более внимательно, поскольку ошибки в таких типах ведут к более катастрофическим неисправностям;
- Методы с недостаточной целенаправленностью — методы, которые используются не для достижения общей цели типа; принимает значения от 0 до 1; может использоваться для определения перегруженных или неудачных классов;
- Цикломатическая сложность — количество операторов ветвления и цикла в типе; определяет общую сложность типа и сложность чтения и поддержки;
- Цикломатическая сложность IL кодов;
- Ассоциация между классами — количество непосредственно определенных объектов других типов в теле этого типа;
- Количество дочерних классов;
- Глубина дерева наследования;

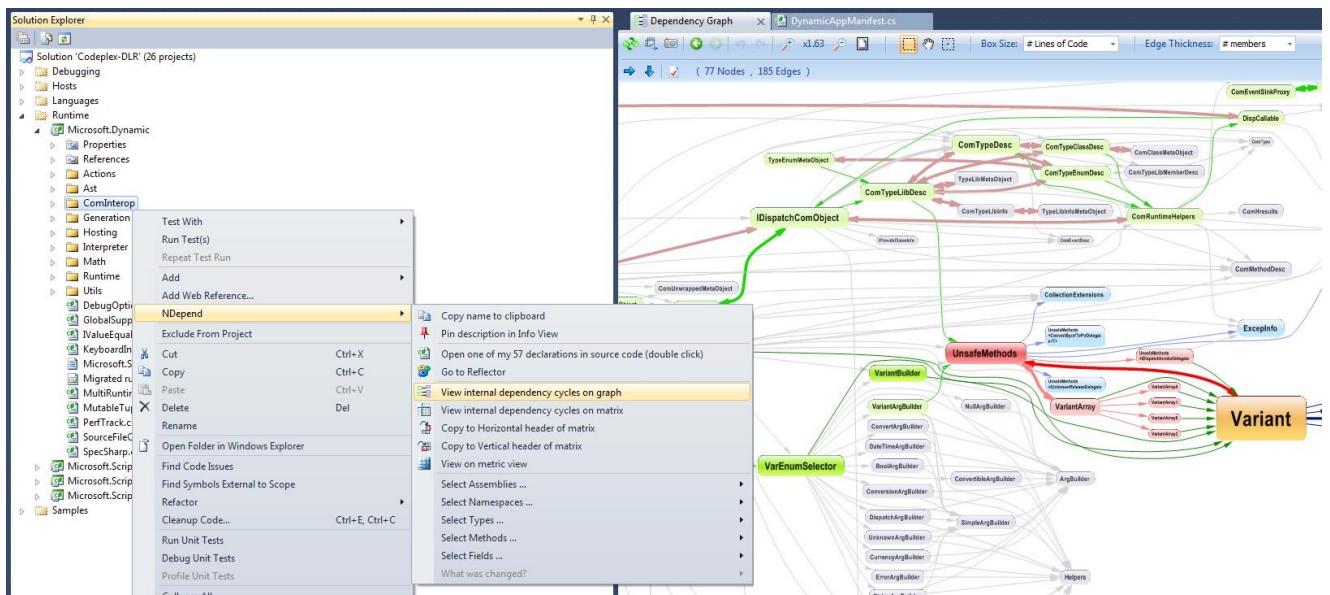
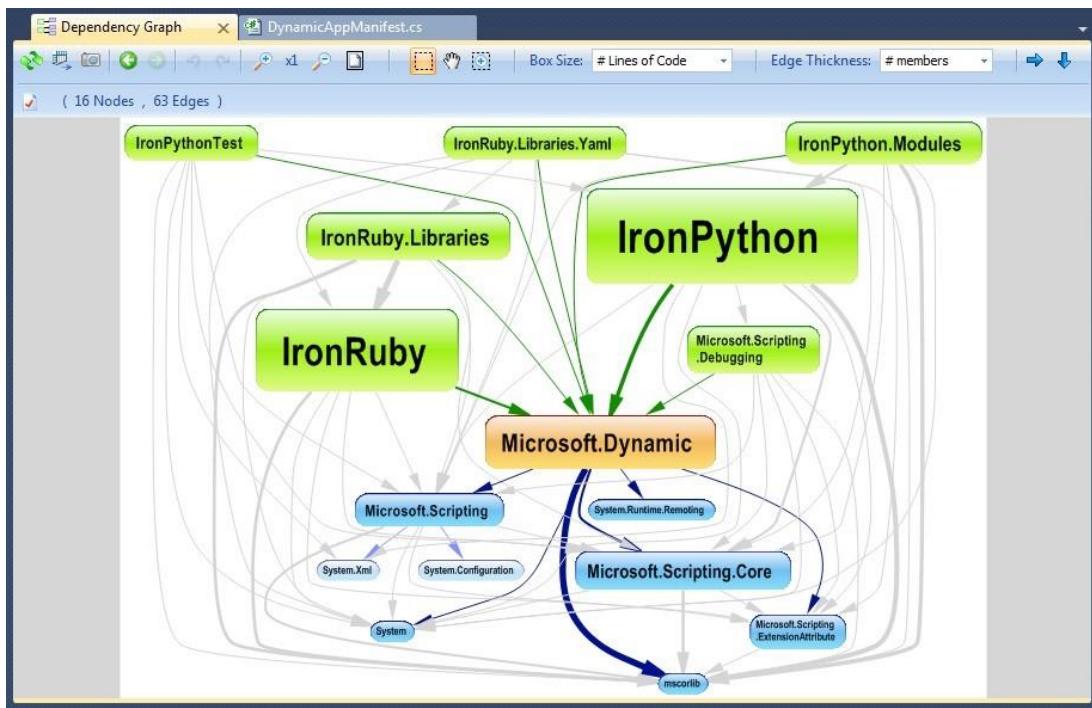
Метрики по классам позволяют оценить грамотность проектирования и реализации отдельных классов. Поскольку каждый класс, точно так же как и модуль, но на другом уровне (тактическом), должен выполнять лишь одну цель, необходимо следить за метриками связности и целенаправленности. Эти метрики также позволяют получить некоторое представление о простоте будущей поддержки данного кода и простоты его изменения.

Также в NDepend представлены 19 метрик по методам (представлены лишь новые): - Ранг метода;

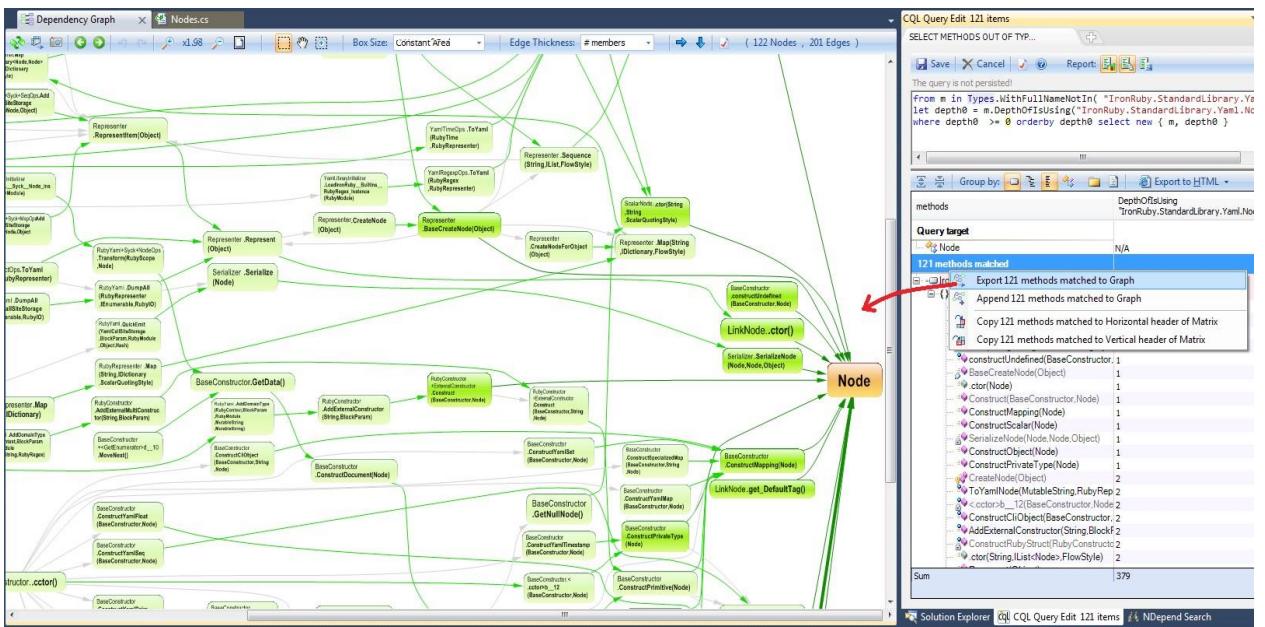
- Количество параметров;
- Количество переменных;- Количество перегрузок;
- Процент покрытия ветвлений.

Данные метрики помогают оценить простоту и понятность отдельных методов в классе. Большое количество параметров или переменных может означать, что метод выполняет очень много обязанностей. Большое количество перегрузок может значить либо слишком обобщенный метод (что редко необходимо, помимо случаев, когда проектируются общие библиотеки уровня framework'a), либо недостаточно грамотный подход к именованию методов. Все эти метрики также позволяют находить мертвые участки кода (не имеют внешних связей), перегруженные или необоснованно выделенные модули и другое. Помимо этих возможностей, NDepend имеет визуализационные средства.

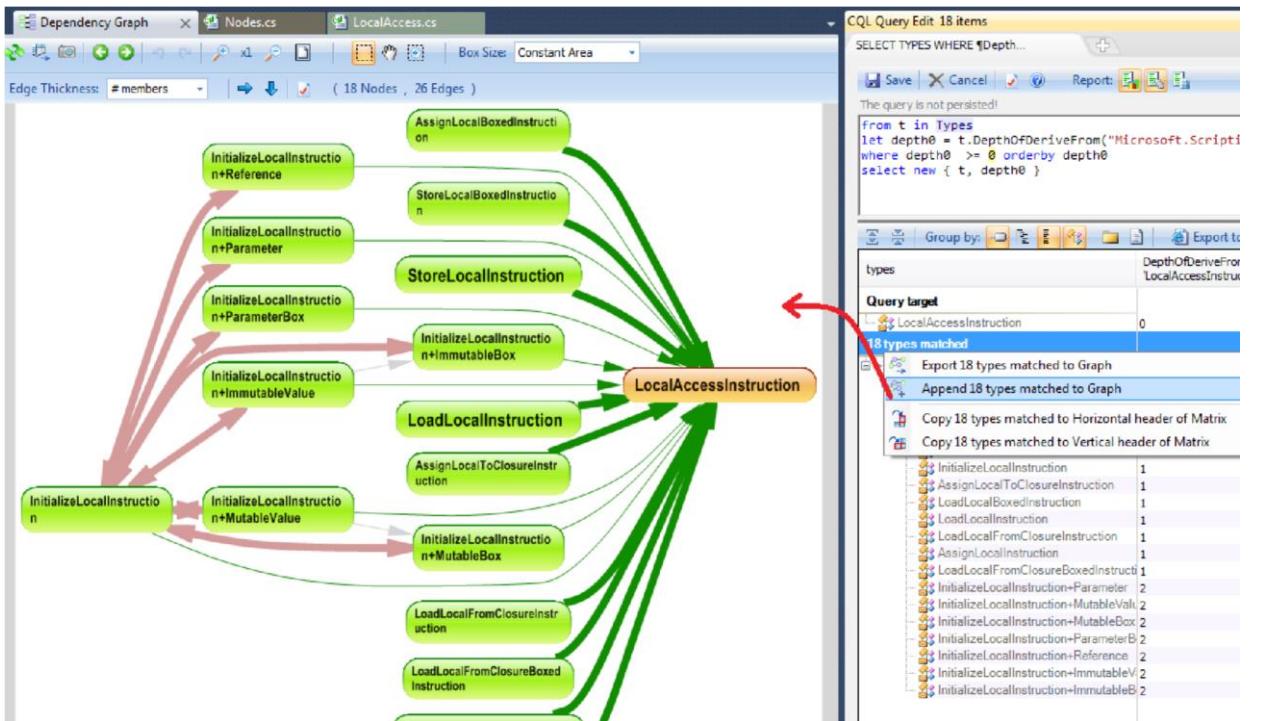
Визуализация зависимостей:



Дерево вызовов:



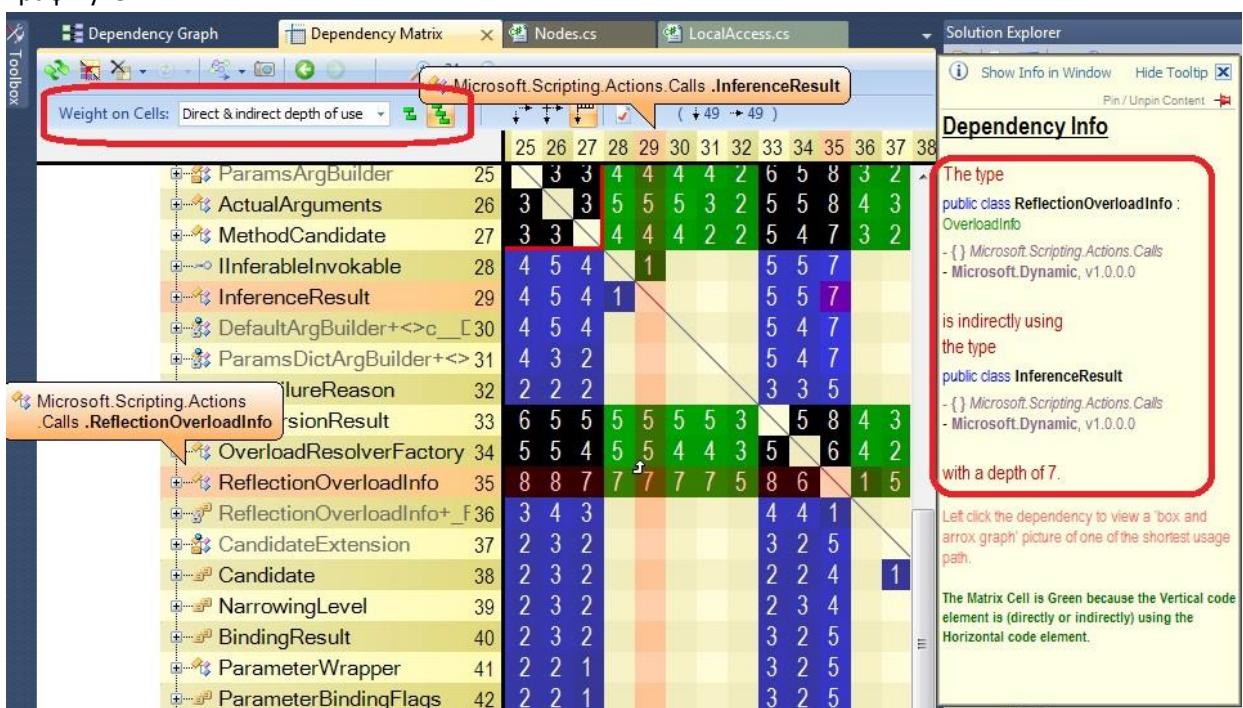
Глубина наследований



Связность компонентов:

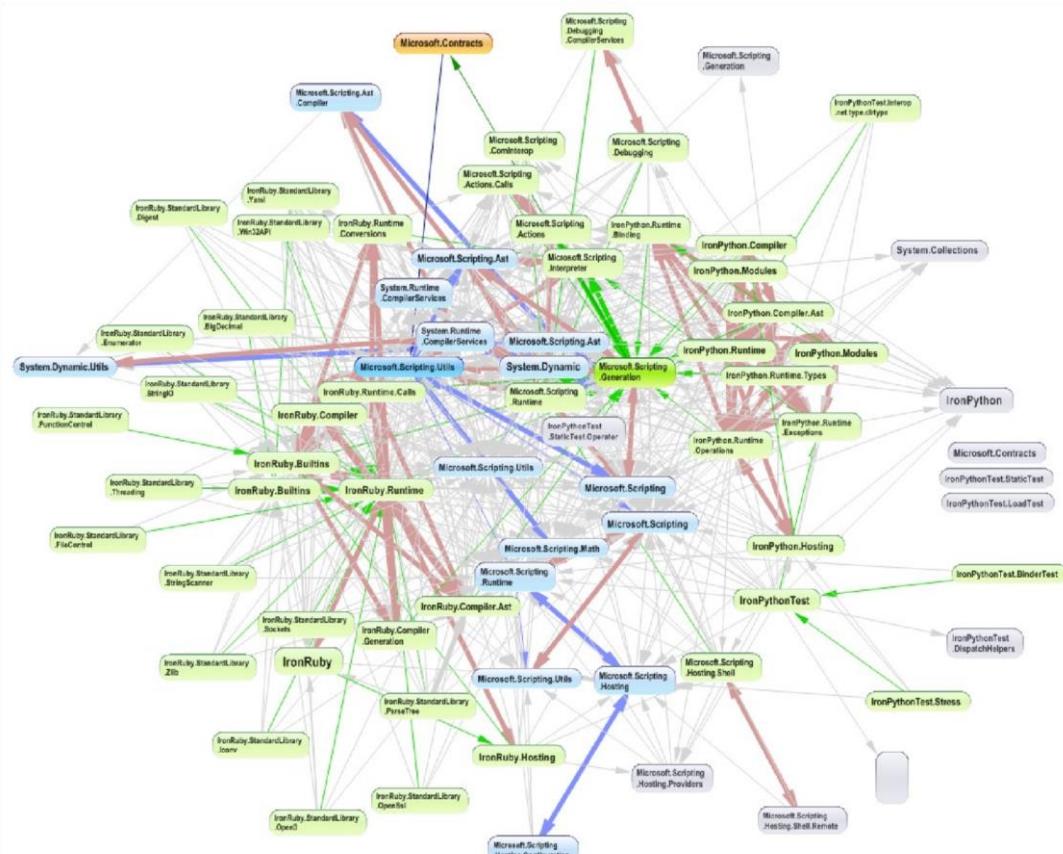


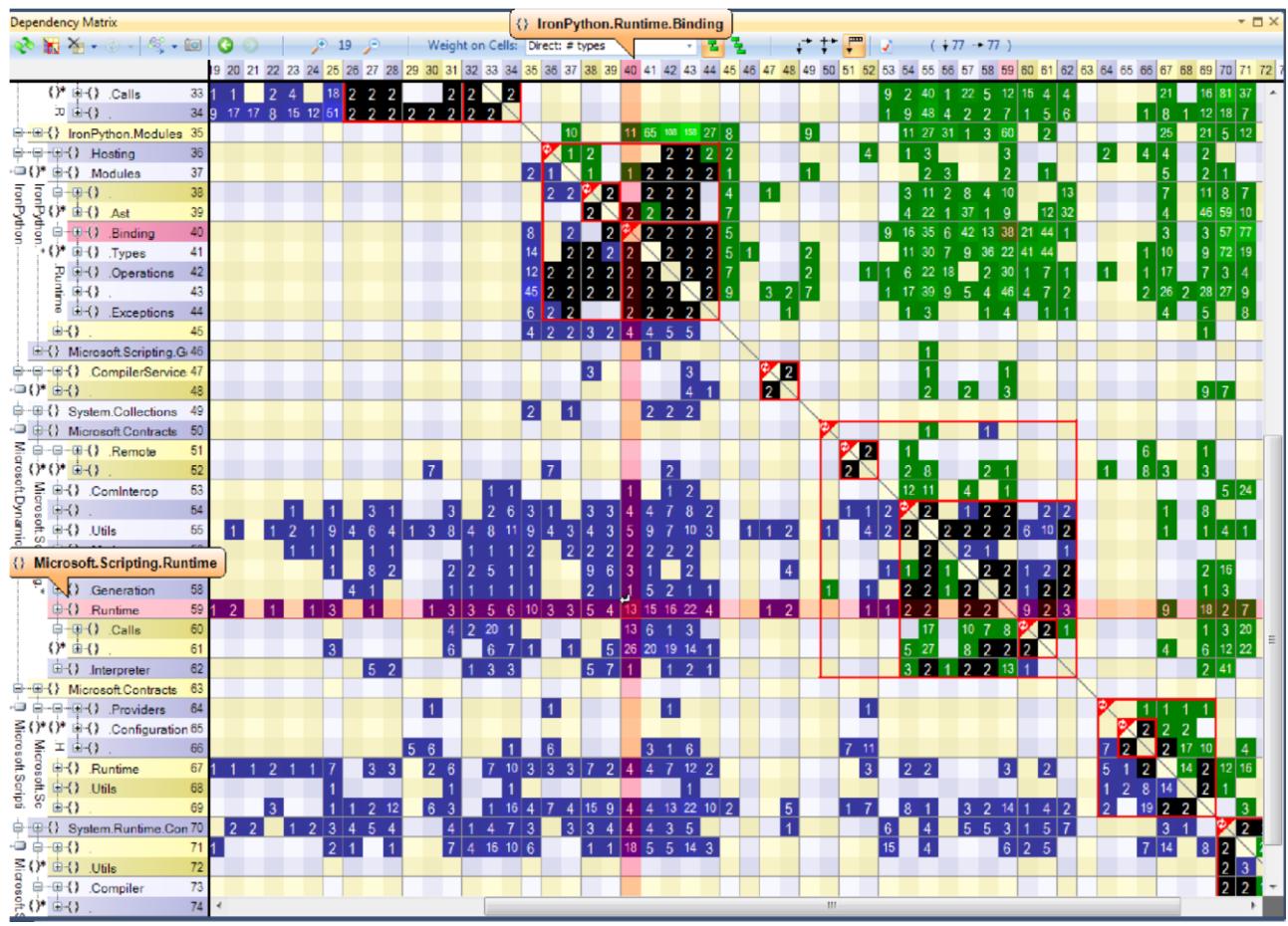
Граф путей:





Граф структуры зависимостей





ЛАБОРАТОРНАЯ РАБОТА № 16 ДОКУМЕНТИРОВАНИЕ ИСХОДНОГО КОДА

Задание.

1. Документировать все public классы, свойства, поля в лабораторном проекте.
2. Сгенерировать документацию при помощи любого средства генерирования документации по проекту.

Краткие теоретические сведения.

Программная документация — текст или документ, сопровождающий исходные коды программы либо бинарные файлы. Документация является важной частью процесса разработки программного обеспечения и бывает следующих типов:

1. Требования — документирование требований к программному обеспечению;
2. Архитектура системы — документирование архитектурных решений приложения;
3. Техническая — документирования набора классов, свойств, полей, интерфейса программирования приложений (API);
4. Пользовательская — инструкции по эксплуатации, администрированию, настройке и другие;
5. Маркетинговая — каким образом рекламировать продукт и на каких рынках.

В данной лабораторной работе рассматривается техническая документация. При создании программного обеспечения написания программного кода недостаточно, необходимо также дополнять его сопровождающим текстом, объясняющим те или иные участки кода или принимаемые решения. Данные участки текста называются документацией.

Документация должна быть полной и понятной, однако не должна быть большой — поскольку в таком случае ее становится тяжело поддерживать. Тяжело поддерживаемая документация приводит к ее рассинхронизации с реальным состоянием вещей (документация устаревает), и в результате вовсе перестает обновляться.

Данная документация впоследствии используется следующими лицами:

1. Новые разработчики — для ознакомления с проектом и понимания архитектуры и строения программного продукта;
2. Тестировщики — для составления корректных тест-кейсов и написания качественных баг-репортов, владения терминологией проекта;
3. Сторонние разработчики — для получения информации о предоставляемом API и способе его использования;

Для упрощения создания технической документации были созданы автоматизированные средства документирования продуктов. Как правило, такие средства генерируют html файлы (либо chm, pdf и любые другие), собранные по документации исходного кода и содержат структуру «namespace, class, members».

Для документирования кода в Microsoft Visual Studio используется специальный тип комментариев — три слеша («///»). Также доступен и особый синтаксис:

//<summary> - общее описание конкретного модуля
//<param> - параметры, принимаемые методом
//<return> - возвращаемые параметры
<seealso>, <see> - ссылки на другие участки кода/внешние ресурсы

Пример:

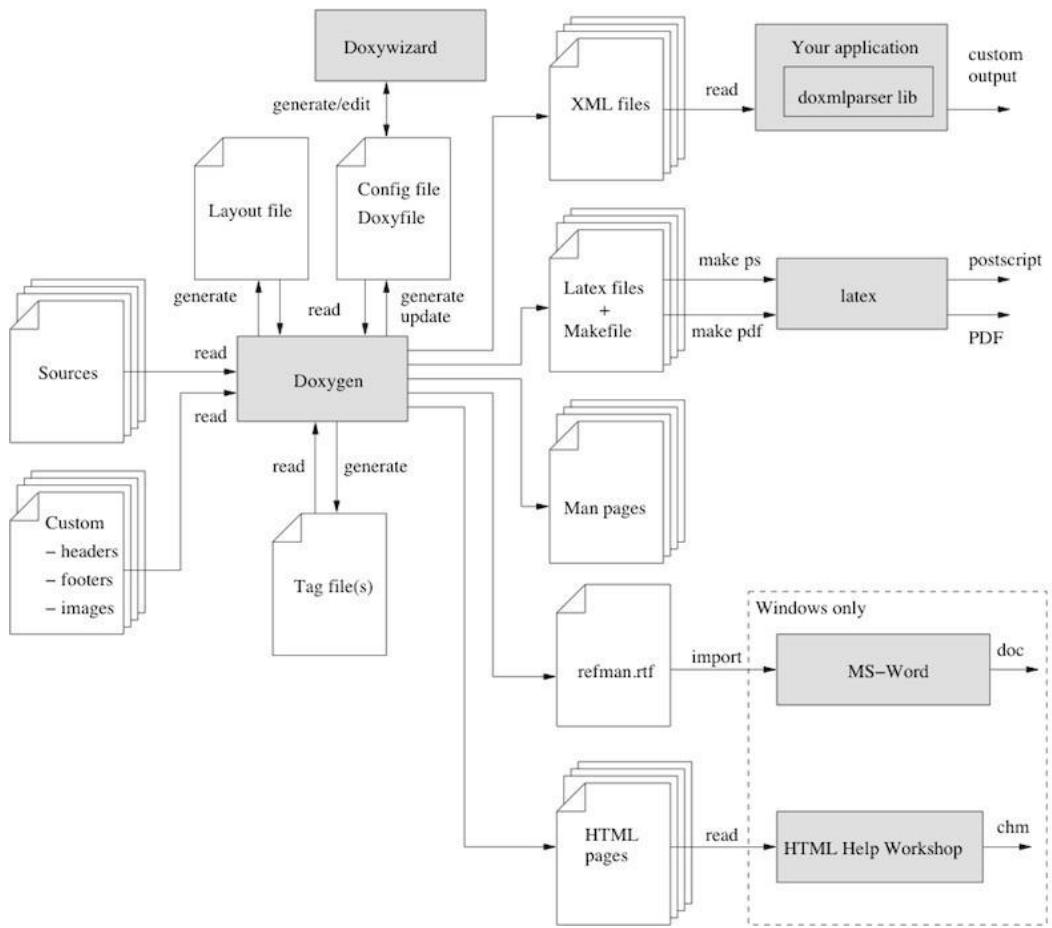
```
/// <summary>DoWork is a method in the TestClass class.  
/// <para>Here's how you could make a second paragraph in a description. <see  
/// cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>  
/// <seealso cref="TestClass.Main"/>  
/// </summary>  
public static void DoWork(int Int1)  
{  
}
```

Doxxygen

<http://doxygen.org/>

Для автоматизированного создания пакетов документации существует масса различных программ (NDoc, Sandcastle и другие). Для языка C# каждая из этих программ умеет читать и распознавать структуру файлов, проектов и сборок, находить участки комментариев и выделять их в отдельные файлы.

Общая структура работы программы Doxygen следующая. Прежде всего, генерируется конфигурационный файл, который указывает путь к сборке, тип документации для генерирования, исключения генерации, глубину вхождения и прочие параметры. Дополнительно можно задать шаблон документации (layout file) — внешний вид результата; или использовать шаблон поумолчанию. Шаблон представляет собой файл формата html со специальными вставками. Далее при запуске Doxygen считывает структуру и документацию по заданному пути и генерирует набор результирующих файлов в одном из следующих форматов: xml — для последующего преобразования собственными приложениями; latex + make — для генерации postscript и PDF файлов; rtf — для работы с Microsoft Word; html, chm — для выкладывания в сеть Internet (интерактивная документация).



Контрольные вопросы.

1. Зачем применяется документирование исходного кода ?
2. Какие виды документации Вам известны ?
3. К какому виду документации можно отнести UML диаграммы ?
4. Какие синтаксические конструкции Вам известны для документирования C# кода ?
5. Каким образом работают автоматические генераторы документации ?