МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Пятигорский институт (филиал) СКФУ
Колледж Пятигорского института (филиал) СКФУ

МОДЕЛИРОВАНИЕ И АНАЛИЗ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

Специальности СПО
09.02.07 Информационные системы и программирование
Квалификация специалист по информационным системам

Пятигорск 2025

Методические указания для практических занятий по дисциплине «Моделирование и анализ программного обеспечения» составлены в соответствии с ФГОС СПО. Предназначены для студентов, обучающихся по специальности 09.02.07 Информационные системы и программирование

Пояснительная записка

Выполнение студентами практических работ направленно на:

- обобщение, систематизацию, углубление, закрепление полученных теоретических знаний по конкретным темам;
- формирование умений применять полученные знания на практике и реализацию единства интеллектуальной и практической деятельности;
- развитие интеллектуальных умений у будущих специалистов: аналитических, проектировочных, конструктивных и др.;
- выработку при решении поставленных задач таких профессиональных качеств, как самостоятельность, ответственность, точность, творческая инициатива.

В результате изучения учебной дисциплины «Моделирование и анализ программного обеспечения» обучающийся должен**уметь**:

-работать с проектной документацией;

- разработанной с использованием графических языков спецификаций;
- выполнять оптимизацию программного кода с использованием специализированных программных средств;
- использовать методы и технологии тестирования и ревьюирования кода и проектной документации;
- применять стандартные метрики по прогнозированию затрат, сроков и качества.

В результате освоения учебной дисциплины обучающийся должен **знать:**

- задачи планирования и контроля развития проекта;
- принципы построения системы деятельностей программного проекта;
- современные стандарты качества программного продукта и процессов его обеспечения.

Практическая работа № 1

Тема 1.1: Методы организации работы в команде разработчиков. Системы контроля версий.

1.Системы контроля версий.

Цель:Научиться определять порядок работы с системой.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

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*).

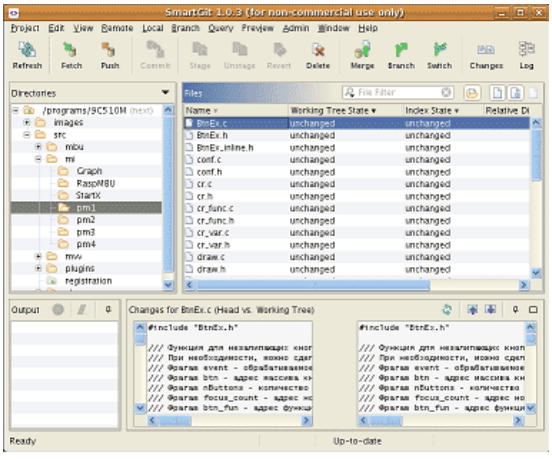
При разработке программного обеспечения рано или поздно приходится вносить сложные исправления(изменения), в которых, с большой вероятностью, могут содержаться ошибки. Если проект - небольшой, то, обычно, делается его резервная копия. Но что делать, если вы сопровождаете огромный проект, состоящий из сотен и даже тысяч файлов?

Копировать каждый раз весь проект, вручную описывать версию - долго и трудоемко. Мало того, что это будет занимать слишком много места на жестком диске, так еще немудрено запутаться в версиях и использовать, вместо сохраненной правильной версии, промежуточную недоделанную версию с массой ошибок. Хорошо, если потом можно быстро откатиться на правильную версию, но ведь бывают случаи, когда, наводя порядок в архиве, удаляются единственно верные копии с последними наработками.

Все это значительно усложняется, когда проект ведут несколько человек, иногда территориально удаленных друг от друга на сотни и тысячи километров. В этом случае у каждого будут образовываться свои архивы версий, и начнется настоящий кошмар сопровождения разработанного программного обеспечения.

Естественным путем, сообщество разработчиков пришло к выводу о необходимости специализированного программного обеспечения, позволяющего просто и надежно сопровождать большие программные проекты. И в скором времени появилось множество программ для контроля версий (Git, CVS, Subversion, Bazaar, Monotone, Aegis и др.), позволяющих удовлетворять любые, даже самые изощренные пожелания.

Эти программы – системы контроля версий, позволяют:



Интерфейс одной из графических оболочек системы контроля версий Git

1. Сохранять все этапы разработки. Внеся изменения в один или несколько файлов проекта, программист записывает изменения в репозиторий – хранилище всех версий и изменений проекта. Стоит отметить, что сохраняется не весь проект целиком, а, в целях экономии места и времени сохранения изменений (сервер с репозиторием может быть удаленным, и, если проект - очень большой, то требуется достаточно большое время для передачи всех его файлов по сети), в репозиторий добавляются только файлы, претерпевшие изменения.

2. Обновляется до последней версии разработанного программного обеспечения. Так как, обычно, над разработкой проектов трудится целая команда специалистов, то они постоянно добавляют в репозиторий измененные файлы. Поэтому одной из основных задач системы контроля версий является возможность отслеживать все эти изменения и быстро обновлять программное обеспечение клиентов до актуальной версии.

3. Объединять изменения. Часто несколько программистов одновременно изменяют одни и те же файлы. Если изменения не пересекаются, то системы контроля версий позволяют легко и просто объединить эти изменения.

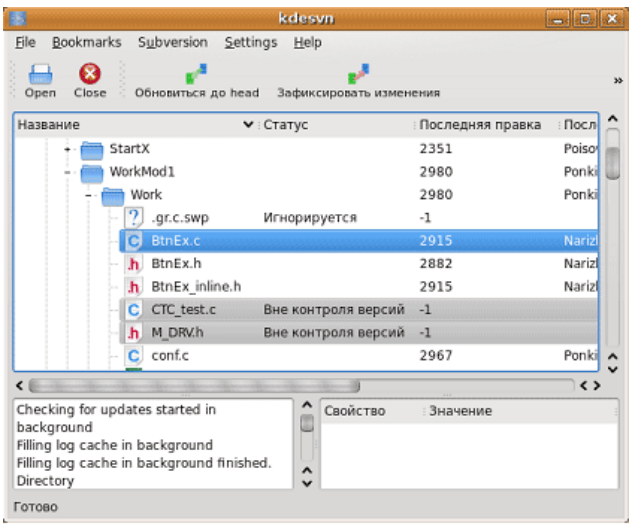
4. Решать конфликты. Если несколько человек изменили один и тот же участок кода, то, автоматически, объединить такие изменения - невозможно. Обычно, системы контроля версий предоставляют собой инструменты, позволяющие вручную внести необходимые правки в тест программ, чтобы объединить конфликтующие части кода.

5. Откатываться к предыдущим версиям. Если выбранное направление в развитии проекта оказалось тупиковым или содержащим ошибки, то системы контроля версий позволяют вернуть разработанное программное обеспечение к одной из последней рабочей версии, просто, скопировав из репозитория нужную версию программного обеспечение, либо отдельные файлы.

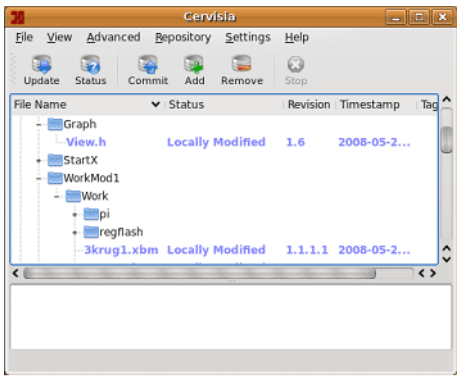
6. Сопровождение нескольких направлений развития программного обеспечения. Не всегда можно сразу сохранять внесенные изменения. Часто приходится достаточно долго разрабатывать и отлаживать отдельные правки прежде, чем их можно объединить с основным программным обеспечением. В этом случае многие системы контроля версий позволяют организовывать параллельные ветки по контролю нескольких направлений развития программного обеспечения, быстро переключаться между ними, а затем объединять их в единое целое.

Кроме этого, системы контроля версий обладают удобным интерфейсом для быстрого и простого выполнения перечисленных выше действий и надежного контроля версий разрабатываемого программного обеспечения.

Каждой системе контроля версий присуще свои достоинства и недостатки, но, в общем, все они основываются на одном и том же принципе: регистрации изменений в программном коде и сохранение каждого нового обнаруженного изменения в новой версии, к которой можно вернуться в любой момент времени.



Интерфейс одной из графических оболочек системы контроля версий Subversion



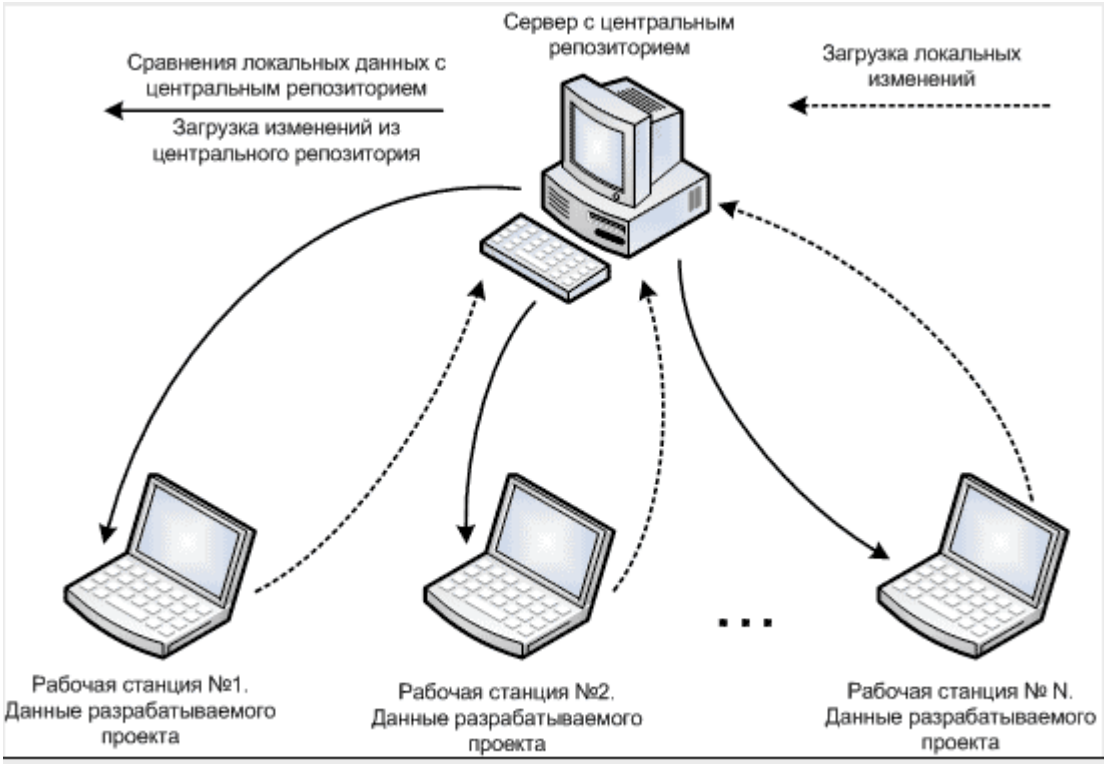
Интерфейс графической оболочки системы контроля версий CVS (Concurrent Versions System - система одновременных версий)

Несмотря на единый принцип работы, системы контроля версий можно разделить на две группы. Это - централизованные системы контроля версий и распределенные. У них много общего, но есть и принципиальные отличия.

Централизованные системы, такие как CVS и Subversion, для сохранения всех рабочих файлов контролируемого проекта использует репозиторий, размещенный на отдельном сервере.

В этом случае для работы с системой контроля версий каждый участник проекта сначала скачивает с сервера последнюю версию программного продукта, вносит в нее свои изменения и загружает на сервер полученный результат. При этом работа ведется с последней версией программного продукта, а, если необходимо вернуться к одной из предыдущих версий разработки, что бывает достаточно часто, приходится каждый раз запрашивать сервер и скачивать необходимую версию.

После внесения всех корректировок в скаченную версию, она заливается на сервер, в качестве последней версии разрабатываемого продукта. При этом, скорее всего, придется решать множество конфликтов, что бывает очень сложным, так как откат на предыдущую версию может затронуть большое количество уже сохраненных изменений.



Принцип работы централизованной системы контроля версий

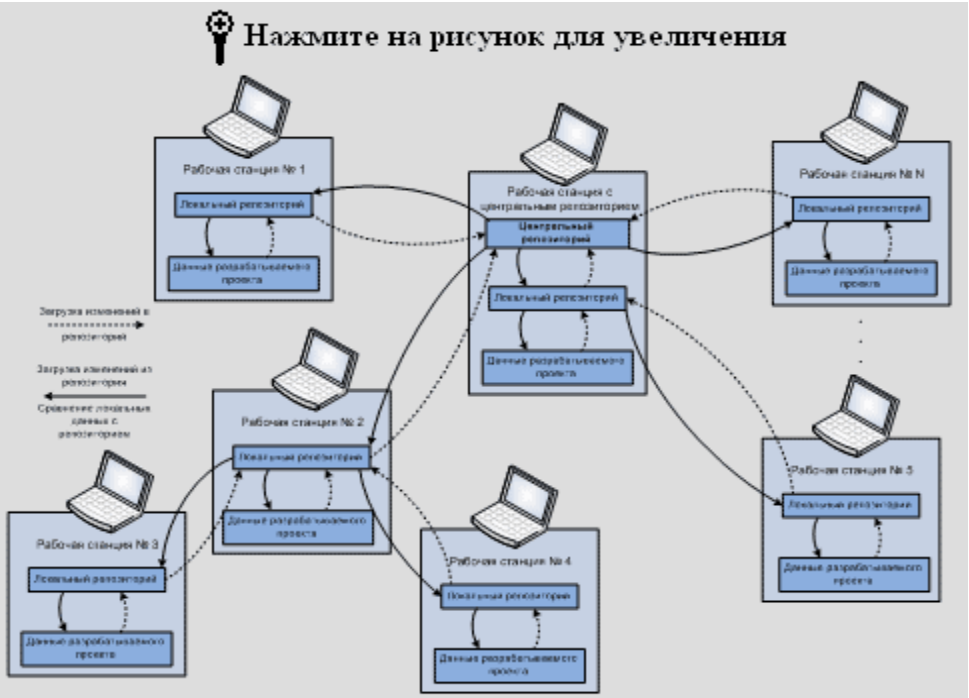
В распределенных системах, таких как Git, когда пользователи загружают данные из репозитория сервера, скачиваются все сохраненные изменения, а не только последняя версия. Естественно, каждый раз скачивать весь репозиторий не нужно, так как достаточно скопировать только те изменения, которых нет в локальном проекте пользователя. Даже, несмотря на это, операции копирования данных из репозитория могут быть достаточно долгими, но это с лихвой окупается при дальнейшем использовании распределенной системы контроля версий.

По сути, используя распределенные системы контроля, каждый пользователь имеет свой личный репозиторий, в который он локально сохраняет все свои изменения. При необходимости создает параллельные ветки контроля версий проекта,

отслеживающие сложные изменения, которые пока что нельзя сохранять в основной версии разрабатываемого проекта.

В любой момент откатиться на нужную версию проекта, переключиться между ветками или объединить несколько параллельных веток в единую, содержащую все внесенные изменения. Все это делается без обращения к основному серверу.

Загрузка изменений в основной репозиторий будет производиться, когда все они отлажены и избавлены от ошибок.



Принцип работы децентрализованной системы контроля версий

При этом может показаться, что использование распределенных систем - затруднительно в проектах, требующих жесткого централизованного управления. Однако, это - не так. Наличие у каждого пользователя своего репозитория со всеми изменениями, не отменяет важность основного репозитория. Это просто дает возможность пользователям более гибко работать с программным продуктом, что ускоряет и упрощает разработку.

Стоит отметить, что назначение системы контроля версий не ограничивается сопровождением разработки программных продуктов. Их можно использовать и для ведения документации, и для управления почтой, и для синхронизации данных на нескольких компьютерах в сети, и для многих других задач.

На данный момент системы контроля версий все более прочно входят в мир высоких технологий, и уже сейчас без них не мыслима разработка серьезных проектов

Контрольные вопросы:

1.Что такое системы управления версиями?

2. Что позволяют системы контроля версий?

Практическое занятие № 2

Тема 1.1:Методы организации работы в команде разработчиков. Системы контроля версий.

2.Элементы контроля версий.

Цель: Научиться определять структуру контроля версий.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Чаще всего в качестве элементов для контроля версий выступают:

s файлы;

s директории;

s hard - и softlinks.

Внутри системы контроля сами элементы могут размещаться по-разному - это зависит от архитектуры СКВ. Пользователю важно лишь знать, что элемент помещается внутри хранилища и работа с ним идет с помощью команд выбранного инструментария. Как уже было сказано, системы контроля должны предоставлять структуры для хранения версий. Самым распространенным представлением подобной структуры является дерево версий. Это такая организация версий элемента, при которой на основе любой версии элемента конфигурации может быть создано несколько наборов последовательностей его версий. При этом отдельный набор версий, происходящий из произвольной версии, называется веткой. И поскольку ветка содержит версии, то каждая из версий может быть источником для создания других веток.

Типовые примеры веток таковы:

контроль версия программный продукт

s ветка для запроса на изменения - заводится для версий, создаваемых в ходе работы по запросу на изменение ("девелоперская", или "сиарная", ветка);

s интеграционная ветка - служит промежуточным хранилищем для процесса стабилизации;

s релизная ветка - для выкладывания версий при стабилизации конфигурации (см. соответствующий раздел первой части статьи). Какие-то версии на ветке могут быть в дальнейшем объявлены частью базовой конфигурации;

s отладочная ("дебажная") ветка - для кратковременного хранения версий, в основном, для целей проверки каких-либо решений.

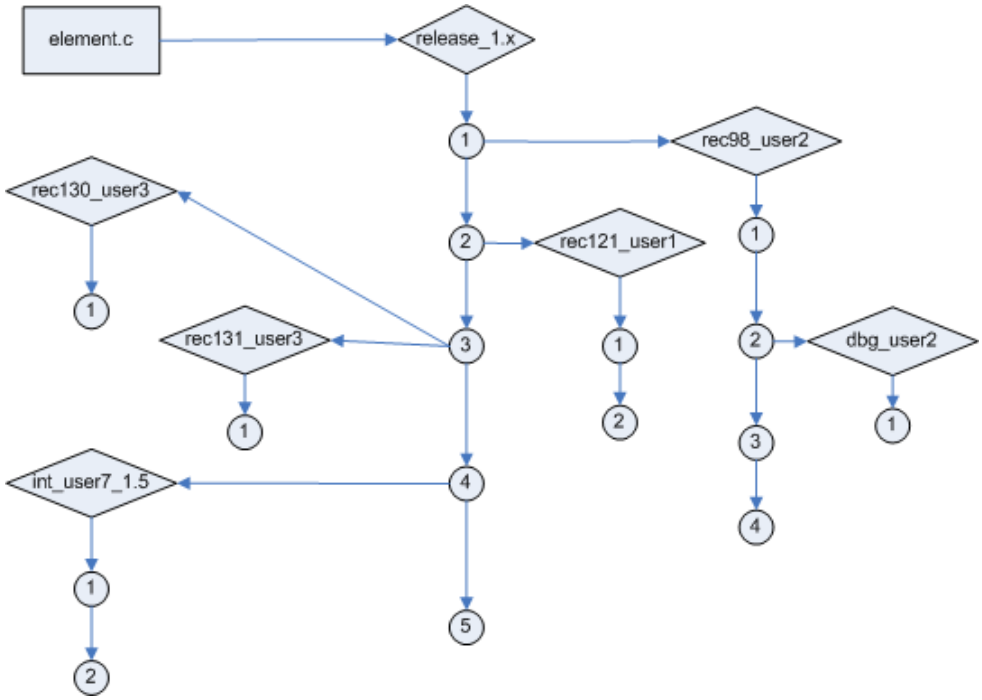


Рисунок 1 - Дерево версий элемента element. c

На рисунке 1 изображен пример дерева версий. У файла element. c создана релизная ветка release_1. x, куда складываются стабилизированные версии этого элемента (1-5). Для сохранения дельты по каждому запросу на изменения заводится отдельная ветка

со специальным форматом имени. В нашем случае формат имеет вид rec<номер_записи>_<имя_пользователя>, где номер_записи - это ID запроса на изменение в системе отслеживания. Для объединения дельты от разных разработчиков создаются интеграционные ветки с названиями вида int_<имя_пользователя>_<суффикс>, где суффикс хранит описание интеграции или номер стабилизируемой конфигурации. Также можно увидеть ветку для отладки, чаще всего они именуются как dbg_<имя пользователя>_<произвольный_комментарий> - на неё выкладываются проверочные варианты изменений.

Каждый проект может иметь свои способы создания и именования веток, однако основные из них были перечислены выше. Дерево версий растет и ширится, и рано или поздно надо результаты работы объединять. Например, разработчик отрастил ветку у одного из элементов для работы над запросом на изменение. На неё он положил несколько версий, и последняя является той, которая содержит отлаженный и протестированный код. В то же время есть релизная ветка, где лежат версии, выпускаемые в рамках базовых конфигураций и стабильных релизов. Необходимо соединить результаты.

Для этого используется механизм слияния версий. Как правило, он подразумевает создание новой версии элемента, для которой в качестве основы берется базовая версия на выбранной ветке (база), и к ней применяются изменения, содержащиеся в выбранной сторонней версии (источнике). В англоязычных источниках используется термин merge ("мёрж"). Ветка с версией-источником может быть отращена как от версии-источника, так и от его более ранних предков. Существующие СКВ позволяют делать слияние как вручную, так и автоматически. Причем второй способ является основным. Ручное слияние запрашивается лишь в случае конфликтов.

Конфликты слияния возникают в случае, если в обоих версиях элемента меняется один и тот же фрагмент. Такая ситуация возникает когда предок версии-источника не является версией, от которой будет расти новая версия. Типичным примером такого конфликта может служить история изменений (revision history), которая добавляется в начало файла исходников, чтобы в каждой версии можно было сразу видеть, кто последним менял и что было сделано. В случае слияния версий, отращенных от разных источников, эта строчка точно будет вызывать конфликт, и решается он лишь вставкой обеих строчек в историю. Когда возникает более сложный случай - разработчик или эксперт в затронутом коде должен внимательно вручную произвести нужные изменения.

К вопросу об общих предках и о слиянии изменений: кроме ручного и автоматического, слияние может быть произведено двухпозиционным и трёхпозиционным способом. Двухпозиционное слияние производится простым сравнением двух версий и сложением их дельты (разницы между версиями элемента). Алгоритм работает по принципу diff'а или приближенно к нему: взять дельту и вставить/удалить/изменить нужные строки.

Трехпозиционное слияние учитывает "общего предка" обеих версий и высчитывает дельту исходя из истории изменения элемента в соответствующих ветках. Соответственно, при возникновении конфликта слияния разработчику предлагается 3 версии элемента - общий предок и 2 варианта, что с этим предком стало с течением времени и изменений. Такой подход помогает оценить степень и важность дельты на обеих ветках и принять решение о необходимости интеграции конфликтного куска часто даже без участия авторов изменений.

После того как слияние проведено, информация о нём должна быть сохранена, если это возможно. Как правило, большинство зрелых VCS имеют возможность сохранить "стрелки слияния" - метаинформацию о том, откуда, куда и в каком момент времени сливались изменения и кто это делал.

Основная идея систем контроля версий - запоминать все внесенные изменения, а так же комментарии пользователей, их вносивших. После этого становится понятно кто когда и что менял, зачем и почему. И, что немаловажно, все эти изменения можно потом откатить на любой момент времени. Ну и, помимо всего этого, систему контроля версий можно ещё дополнительно расширить различными плагинами.

Самые известные системы контроля версий, которые чаще всего упоминаются - CVS, Subversion, Git, Mercurial, Bazaar и Monotone.

Все эти системы объединяет то, что это системы с одним, выделенным сервером, на котором и находится репозиторий с кодом.

Контрольные вопросы:

1. Типовые примеры веток?
2. Причины возникновения конфликтов слияния?

Практическое занятие № 3

Тема 1.2: Цели, задачи, этапы и объекты ревьюирования. Планирование ревьюирования.

Цель: Научиться определять задачи и этапы ревьюирования.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Теоретическая часть.

Ревьюирование (инспекция) программного кода.

Инспекция кода (Code review) – систематический и периодический анализ программного кода, направленный на поиск необнаруженных на ранних стадиях разработки программного продукта ошибок, а также, на выявление некачественных архитектурных решений и критических мест в программе.

Задачи и цели проведения формальных инспекций.

Не всегда возможна разработка автоматических или хотя бы четко формализованных ручных тестов для проверки функциональности программной системы. В некоторых случаях выполнение тестируемого программного кода невозможно в условиях, создаваемых тестовым окружением. Такая ситуация возможна во встроенных системах, если программный код предназначен для обработки исключительных ситуаций, создаваемых только на реальном оборудование.

В тех случаях, когда верифицируется не программный код, а проектная документация на систему, которую нельзя "выполнить" или создать для нее отдельные тестовые примеры, также обычно прибегают к методу экспертных исследований программного кода или документации на корректность или непротиворечивость.

Такие экспертные исследования называют инспекциями или просмотрами. Существует два типа инспекций - неформальные и формальные.

При неформальной инспекции автор некоторого документа или части программной системы передает его эксперту, а тот, ознакомившись с документом, передает автору список замечаний, которые тот исправляет. Сам факт проведения инспекции и замечания нигде отдельно не сохраняются, состояние исправлений по замечаниям также нигде не отслеживается.

Формальная инспекция является четко управляемым процессом, структура которого обычно четко определяется соответствующим стандартом проекта. Таким образом, все формальные инспекции имеют одинаковую структуру и одинаковые выходные документы, которые затем используются при разработке.

Факт начала формальной инспекции четко фиксируется в общей базе данных проекта. Также фиксируются документы, подвергаемые инспекции, и списки замечаний, отслеживаются внесенные по замечаниям изменения. Этим формальная инспекция похожа на автоматизированное тестирование: списки замечаний имеют много общего с отчетами о выполнении тестовых примеров.

В ходе формальной инспекции группой специалистов осуществляется независимая проверка соответствия инспектируемых документов исходным документам. Независимость проверки обеспечивается тем, что она осуществляется инспекторами, не участвовавшими в разработке инспектируемого документа.

Входами процесса формальной инспекции являются инспектируемые документы и исходные документы, а выходами - материалы инспекции, включающие список обнаруженных несоответствий и решение об изменении статуса инспектируемых документов.

Этапы формальной инспекции и роли ее участников

Процесс формальной инспекции состоит из пяти фаз: инициализация, планирование, подготовка (экспертиза), обсуждение, завершение. В некоторых случаях подготовку и обсуждение целесообразно рассматривать не как последовательные этапы, а как параллельные подпроцессы. В частности, такая ситуация может сложиться при использовании автоматизированной системы поддержки проведения формальных инспекций.

Процедура формальной инспекции проекта должна точно описывать порядок проведения формальных инспекций в данном проекте.

После устранения обнаруженных в ходе формальной инспекции несоответствий процесс формальной инспекции повторяется, возможно, в другой форме и с другим составом участников. Процедура формальной инспекции должна регламентировать возможные формы проведения повторной инспекции в зависимости от объема и характера изменений, внесенных в объект инспекции. Как правило, допускается упрощение процесса повторной инспекции (проведение инспекции одним инспектором, отсутствие фазы обсуждения) при внесении в объект инспекции незначительных изменений относительно ранее инспектировавшейся версии.

Инициализация

Руководитель проекта или его заместитель запрашивает из базы, хранящей все данные проекта (например, из системы конфигурационного управления), список объектов, готовых к инспекции, выбирает объект инспекции, затем назначает участников формальной инспекции: автора, ведущего и одного или нескольких инспекторов.

Ведущий также может выполнять роль инспектора; остальные участники выполняют только одну роль. На роль ведущего или инспектора не допускается назначать сотрудников, участвовавших в разработке объекта инспекции.

В роли автора выступает один из разработчиков объекта инспекции, но возможны ситуации, когда разработчик недоступен - например, переведен в другой проект или находится в отпуске. Тогда на роль автора назначается сотрудник, который будет исправлять обнаруженные несоответствия в инспектируемых документах. При инспектировании документов, разработанных заказчиком, автор может не назначаться.

Рекомендуется назначать не менее двух инспекторов. Их количество может быть увеличено, если инспектируются документы особой сложности или новизны понятий, а также, если в качестве инспекторов привлекаются сотрудники с недостаточным опытом. Рекомендуемое общее число участников инспекции не должно превышать пяти.

В обоснованных случаях процедура формальной инспекции проекта может допускать проведение инспекции единственным инспектором, например, когда объект инспекции отличается особой простотой и оцениваемые характеристики такого объекта инспекции тривиальны.

В случае, если проводится повторная инспекция по сокращенной форме, ведущий самостоятельно инициирует процесс повторной инспекции без участия руководителя проекта. Процедура формальной инспекции проекта может разрешать ведущему самостоятельно инициировать процесс повторной инспекции (в том же составе участников), даже когда она проводится в полной форме, если это диктуется спецификой проекта.

Планирование

По завершению процесса инициализации ведущий проверяет, что инспектируемые документы размещены в базе данных проекта, а их статус соответствует готовности к формальной инспекции. Если это не так, инспекция откладывается.

Затем ведущий должен изменить статус инспектируемых документов так, чтобы отметить факт начала инспекции и ограничить доступ к инспектируемой документации. Во время инспекции изменение документов невозможно, а соответствующий статус сохраняется до конца инспекции. Этот статус называется Review.

После этого ведущий должен скопировать из базы данных проекта бланк инспекции и занести в него идентификаторы инспектируемых и исходных документов и номера их версий, список участников с указанием их ролей и дату фактического начала процесса инспекции, т.е. того момента, когда инспектируемые документы были переведены в состояние Review

Ведущий должен оценить время, необходимое инспекторам для подготовки, и продолжительность обсуждения. Время, отводимое на этап подготовки, не может быть менее одного часа. Также ведущий должен определить дату, время и место обсуждения, если оно будет проходить в форме собрания. При этом может потребоваться согласование с другими участниками инспекции. Если оценка продолжительности обсуждения в форме собрания превышает 2 часа, то необходимо запланировать несколько собраний, каждое из которых будет длиться не более двух часов.

Процедура формальной инспекции проекта может допускать проведение повторной инспекции без собрания, если итогом предыдущей инспекции было решение о проведении повторной инспекции в сокращенной форме. Также допускается не проводить собрание, если результаты формальной инспекции ведутся и хранятся в электронном виде.

В этом случае процедура формальной инспекции проекта должна регламентировать взаимодействия участников формальной инспекции между собой. Кроме того, процедура формальной инспекции проекта должна определять механизм подготовки, проведения обсуждения и принятия решения.

- Подготовив бланк инспекции и определив время и место собрания, ведущий должен известить участников инспекции о времени и месте проведения собрания и разослать им подготовленный бланк инспекции.
- Процедура формальной инспекции проекта может предусматривать использование бланка, заполненного в ходе предыдущей инспекции, если проводится повторная инспекция в сокращенной форме и при этом ведущий является единственным инспектором.

Контрольные вопросы:

1. Понятие ревьюирования.
2. Цели, задачи, этапы ревьюирования.

Практическое занятие № 4

Тема 1.3:Цели, корректность и направления анализа программных продуктов. Выбор критериев сравнения. Представление результатов сравнения.

Цель: Научиться выбирать критерии сравнения программных продуктов.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Быстрое увеличение сложности и размеров современных комплексов программ при одновременном росте ответственности выполняемых функций резко повысило требования со стороны заказчиков и пользователей к их качеству и безопасности применения. Испытанным средством обеспечения высокой эффективности и качества функционирования программ и программных комплексов являются международные стандарты, разработанные при участии представителей ведущих компаний отрасли.

По мере расширения применения и увеличения сложности информационных систем выделились области, в которых ошибки или недостаточное качество программ либо данных могут нанести ущерб, значительно превышающий положительный эффект от их использования.

Во многих случаях контракты и предварительные планы на создание сложных программных средств и баз данных для информационных систем подготавливаются и оцениваются неквалифицированно, на основе неформализованных представлений заказчиков и разработчиков о требуемых функциях и характеристиках качества информационных систем. Значительные системные ошибки при определении требуемых показателей качества, оценке трудоемкости, стоимости и длительности создания программных средств - явление достаточно массовое. Многие информационные системы не способны выполнять полностью требуемые функциональные задачи с гарантированным качеством, и их приходится долго и иногда безуспешно дорабатывать для достижения необходимого качества и надежности функционирования, затрачивая дополнительно большие средства и время. В результате часто проекты информационных систем не соответствуют исходному, декларированному назначению и требованиям к характеристикам качества, не укладываются в графики и бюджет разработки.

В технических заданиях и реализованных проектах информационных систем часто обходятся молчанием или недостаточно формализуются сведения о понятиях и значениях качества программного продукта, о том, какими характеристиками они описываются, как их следует измерять и сравнивать с требованиями, отраженными в контракте, техническом задании или спецификациях. Кроме того, некоторые из характеристик часто отсутствуют в требованиях на программные средства, что приводит к произвольному их учету или к пропуску при испытаниях. Нечеткое декларирование в документах понятий и требуемых значений характеристик качества программных средств вызывает конфликты между заказчиками-пользователями и разработчиками-поставщиками из-за разной трактовки одних и тех же характеристик. В связи с этим стратегической задачей в жизненном цикле современных информационных систем стало обеспечение требуемого качества программных средств и баз данных.

За последние несколько лет создано множество международных стандартов, регламентирующих процессы и продукты жизненного цикла программных средств и баз данных. Применение этих стандартов может служить основой для систем обеспечения качества программных средств, однако требуется корректировка, адаптация или исключение некоторых положений стандартов применительно к принципиальным особенностям технологий и характеристик этого вида продукции. При этом многие клиенты требуют соответствия технологии проектирования, производства и качества продукции современным международным стандартам, которые необходимо осваивать и применять для обеспечения конкурентоспособности продукции на мировом рынке.

Стандартизация характеристик качества

Одной из важнейших проблем обеспечения качества программных средств является формализация характеристик качества и методология их оценки. Для определения адекватности качества функционирования, наличия технических возможностей программных средств к взаимодействию, совершенствованию и развитию

необходимо использовать стандарты в области оценки характеристик их качества. Основой регламентирования показателей качества программных средств ранее являлся международный стандарт ISO 9126:1991 (ГОСТ Р ИСО / МЭК 9126-93) "Информационная технология. Оценка программного продукта. Характеристики качества и руководство по их применению". В России в области обеспечения жизненного цикла и качества сложных комплексов программ в основном применяется группа устаревших ГОСТов, которые отстают от мирового уровня на 5-10 лет.

Первая часть стандарта - ISO 9126-1 - распределяет атрибуты качества программных средств по шести характеристикам, используемым в остальных частях стандарта. Исходя из принципиальных возможностей их измерения, все характеристики могут быть объединены в три группы, к которым применимы разные категории метрик:

- категорийным, или описательным (номинальным) метрикам наиболее адекватны функциональные возможности программных средств;
- количественные метрики применимы для измерения надежности и эффективности сложных комплексов программ;
- качественные метрики в наибольшей степени соответствуют практичности, сопровождаемости и мобильности программных средств.

В части стандарта ISO 9126-1 даются определения с уточнениями из остальных его частей для каждой характеристики программного средства, а также для субхарактеристик качества.

За последние несколько лет создано множество стандартов ISO, регламентирующих процессы и продукты жизненного цикла программных средств и баз данных, которые могут служить основой для систем обеспечения качества программных продуктов.

Вторая и третья части стандарта - ISO 9126-2 и ISO 9126-3 - посвящены формализации соответственно внешних и внутренних метрик характеристик качества сложных программных средств. Все таблицы содержат унифицированную рубрикацию, где отражены имя и назначение метрики; метод ее применения; способ измерения, тип шкалы метрики; тип измеряемой величины; исходные данные для измерения и сравнения; а также этапы жизненного цикла программного средства (по ISO 12207), к которым применима метрика.

Четвертая часть стандарта - ISO 9126-4 - предназначена для покупателей, поставщиков, разработчиков, сопровождающих, пользователей и менеджеров качества программных средств. В ней обосновываются и комментируются выделенные показатели сферы (контекста) использования программных средств и группы выбранных метрик для пользователей.

Выбор показателей качества

Исходными данными и высшим приоритетом при выборе показателей качества в большинстве случаев являются назначение, функции и функциональная пригодность соответствующего программного средства. Достаточно полное и корректное описание этих свойств должно служить базой для определения значений большинства остальных характеристик и атрибутов качества. Принципиальные и технические возможности и точность измерения значений атрибутов характеристик качества всегда ограничены в соответствии с их содержанием. Это определяет рациональные диапазоны значений каждого атрибута, которые могут быть выбраны на основе здравого смысла, а также путем анализа прецедентов в спецификациях требований реальных проектов.

Процессы выбора и установления метрик и шкал для описания характеристик качества программных средств можно разделить на два этапа:

- выбор и обоснование набора исходных данных, отражающих общие особенности и этапы жизненного цикла проекта программного средства и его потребителей, каждый из которых влияет на определенные характеристики качества комплекса программ;

- выбор, установление и утверждение конкретных метрик и шкал измерения характеристик и атрибутов качества проекта для их последующей оценки и сопоставления с требованиями спецификаций в процессе квалификационных испытаний или сертификации на определенных этапах жизненного цикла программного средства.

На первом этапе за основу следует брать всю базовую номенклатуру характеристик, субхарактеристик и атрибутов, стандартизированных в ISO 9126. Их описания желательно предварительно упорядочить по приоритетам с учетом назначения и сферы применения конкретного проекта программного средства. Далее необходимо выделить и ранжировать по приоритетам потребителей, которым необходимы определенные показатели качества проекта программного средства с учетом их специализации и профессиональных интересов. Подготовка исходных данных завершается выделением номенклатуры базовых, приоритетных показателей качества, определяющих функциональную пригодность программного средства для определенных потребителей.

На втором этапе, после фиксирования исходных данных, которое должен выполнить потребитель оценок качества, процессы выбора номенклатуры и метрик начинаются с ранжирования характеристик и субхарактеристик для конкретного проекта и их потребителя. Далее этими специалистами для каждого из отобранных показателей должна быть установлена и согласована метрика и шкала оценок субхарактеристик и их атрибутов для проекта и потребителя результатов анализа. Для показателей, представляемых качественными признаками, желательно определить и зафиксировать в спецификациях описания условий, при которых следует считать, что данная характеристика реализуется в программном средстве. Выбранные значения характеристик качества и их атрибутов должны быть предварительно проверены разработчиками на их реализуемость с учетом доступных ресурсов конкретного проекта и при необходимости откорректированы.

Оценка качества

Методологии и стандартизации оценки характеристик качества готовых программных средств и их компонентов (программного продукта) на различных этапах жизненного цикла посвящен международный стандарт ISO 14598, состоящий из шести частей. Рекомендуется следующая общая схема процессов оценки характеристик качества программ:

- установка исходных требований для оценки - определение целей испытаний, идентификация типа метрик программного средства, выделение адекватных показателей и требуемых значений атрибутов качества;
- селекция метрик качества, установление рейтингов и уровней приоритета метрик субхарактеристик и атрибутов, выделение критериев для проведения экспертиз и измерений;
- планирование и проектирование процессов оценки характеристик и атрибутов качества в жизненном цикле программного средства;
- выполнение измерений для оценки, сравнение результатов с критериями и требованиями, обобщение и оценка результатов.

Для каждой характеристики качества рекомендуется формировать меры и шкалу измерений с выделением требуемых, допустимых и неудовлетворительных значений. Реализация процессов оценки должна коррелировать с этапами жизненного цикла конкретного проекта программного средства в соответствии с применяемой, адаптированной версией стандарта ISO 12207.

Функциональная пригодность - наиболее неопределенная и объективно трудно оцениваемая субхарактеристика программного средства. Области применения, номенклатура и функции комплексов программ охватывают столь разнообразные сферы деятельности человека, что невозможно выделить и унифицировать небольшое число

атрибутов для оценки и сравнения этой субхарактеристики в различных комплексах программ.

Оценка корректности программных средств состоит в формальном определении степени соответствия комплекса реализованных программ исходным требованиям контракта, технического задания и спецификаций на программное средство и его компоненты. Путем верификации должно быть определено соответствие исходным требованиям всей совокупности к компонентов комплекса программ, вплоть до модулей и текстов программ и описаний данных.

Оценка способности к взаимодействию состоит в определении качества совместной работы компонентов программных средств и баз данных с другими прикладными системами и компонентами на различных вычислительных платформах, а также взаимодействия с пользователями в стиле, удобном для перехода от одной вычислительной системы к другой с подобными функциями.

Оценка защищенности программных средств включает определение полноты использования доступных методов и средств защиты программного средства от потенциальных угроз и достигнутой при этом безопасности функционирования информационной системы. Наиболее широко и детально методологические и системные задачи оценки комплексной защиты информационных систем изложены в трех частях стандарта ISO 15408:1999-1--3 "Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий".

Оценка надежности - измерение количественных метрик атрибутов субхарактеристик в использовании: завершенности, устойчивости к дефектам, восстанавливаемости и доступности/готовности.

Потребность в ресурсах памяти и производительности компьютера в процессе решения задач значительно изменяется в зависимости от состава и объема исходных данных. Для корректного определения предельной пропускной способности информационной системы с данным программным средством нужно измерить экстремальные и средние значения длительностей исполнения функциональных групп программ и маршруты, на которых они достигаются. Если предварительно в процессе проектирования производительность компьютера не оценивалась, то, скорее всего, понадобится большая доработка или даже замена компьютера на более быстродействующий.

Оценка практичности программных средств проводится экспертами и включает определение понятности, простоты использования, изучаемости и привлекательности программного средства. В основном это качественная (и субъективная) оценка в баллах, однако некоторые атрибуты можно оценить количественно по трудоемкости и длительности выполнения операций при использовании программного средства, а также по объему документации, необходимой для их изучения.

Сопровождаемость можно оценивать полнотой и достоверностью документации о состояниях программного средства и его компонентов, всех предполагаемых и выполненных изменениях, позволяющей установить текущее состояние версий программ в любой момент времени и историю их развития. Она должна определять стратегию, стандарты, процедуры, распределение ресурсов и планы создания, изменения и применения документов на программы и данные.

Оценка мобильности - качественное определение экспертами адаптируемости, простоты установки, совместимости и замещаемости программ, выражаемое в баллах. Количественно эту характеристику программного средства и совокупность ее атрибутов можно (и целесообразно) оценить в экономических показателях: стоимости, трудоемкости и длительности реализации процедур переноса на иные платформы определенной совокупности программ и данных.

Система управления качеством

Выбор характеристик и оценка качества программных средств - лишь одна из задач в области обеспечения качества продукции, выпускаемой компаниями - разработчиками ПО. Комплексное решение задач обеспечения качества программных средств предполагает разработку и внедрение той или иной системы управления качеством. В мировой практике наибольшее распространение получила система, основанная на международных стандартах серии ISO 9000, включающей десяток с лишним документов, в том числе стандарт, регламентирующий обеспечение качества ПО (ISO 9000/3). Эти стандарты должны служить руководством для ведущих специалистов компаний, разрабатывающих ПО на заказ.

Определения характеристик и субхарактеристик качества (ISO 9126-1)

Функциональные возможности - способность программного средства обеспечивать решение задач, удовлетворяющих сформулированные потребности заказчиков и пользователей при применении комплекса программ в заданных условиях.

Функциональная пригодность - набор и описания субхарактеристики и ее атрибутов, определяющие назначение, номенклатуру, основные, необходимые и достаточные функции программного средства, соответствующие техническому заданию и спецификациям требований заказчика или потенциального пользователя.

Правильность (корректность) - способность программного средства обеспечивать правильные или приемлемые для пользователя результаты и внешние эффекты.

Способность к взаимодействию - свойство программных средств и их компонентов взаимодействовать с одной или большим числом компонентов внутренней и внешней среды.

Защищенность - способность компонентов программного средства защищать программы и информацию от любых негативных воздействий.

Надежность - обеспечение комплексом программ достаточно низкой вероятности отказа в процессе функционирования программного средства в реальном времени.

Эффективность - свойства программного средства, обеспечивающие требуемую производительность решения функциональных задач, с учетом количества используемых вычислительных ресурсов в установленных условиях.

Практичность (применимость) - свойства программного средства, обусловливающие сложность его понимания, изучения и использования, а также привлекательность для квалифицированных пользователей при применении в указанных условиях.

Сопровождаемость - приспособленность программного средства к модификации и изменению конфигурации и функций.

Мобильность - подготовленность программного средства к переносу из одной аппаратно-операционной среды в другую.

Домашнее задание:Провести сравнительный анализ любых двух программных продуктов.

Практическое занятие № 5

Тема 1.4: Примеры сравнительного анализа программных продуктов.

Цель: Научиться проводить сравнительный анализ программных продуктов на примере программных продуктов CASE-технологий.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Сравнительный анализ программных продуктов CASE-технологий

Примерами программных продуктов CASE-технологий являются ERwin, Rational Rose, Методология ARIS.

ERwin - средство концептуального моделирования БД, использующее стандарт IDEF1X. ERwin реализует проектирование схемы БД, генерацию ее описания на языке целевой СУБД (ORACLE, Informix, Ingres, Sybase, DB/2, Microsoft SQL Server, Progress и др.) и реинжиниринг существующей БД. ERwin выпускается в нескольких различных конфигурациях, ориентированных на наиболее распространенные средства разработки приложений 4GL. Версия ERwin/OPEN полностью совместима со средствами разработки приложений PowerBuilder и SQLWindows и позволяет экспортировать описание спроектированной БД непосредственно в репозитории данных средств. Для ряда средств разработки приложений (PowerBuilder, SQLWindows, Delphi, Visual Basic) выполняется генерация форм и прототипов приложений. Сетевая версия Erwin ModelMart обеспечивает согласованное проектирование БД и приложений в рамках рабочей группы. BPwin - средство функционального моделирования, реализующее методологию IDEF0-IDEF3. Методология IDEF0, являющаяся официальным федеральным стандартом США, представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель IDEF0 отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Методология IDEF может использоваться для моделирования широкого круга систем и определения требований и функций, а затем для разработки системы, которая удовлетворяет этим требованиям и реализует эти функции. Для уже существующих систем IDEF может быть использована для анализа функций, выполняемых системой, а также для указания механизмов, посредством которых они осуществляются.

Rational Rose - предназначено для автоматизации этапов анализа и проектирования ПО, а также для генерации кодов на различных языках и выпуска проектной документации. Rational Rose использует синтез-методологию объектно-ориентированного анализа и проектирования, основанную на подходах трех ведущих специалистов в данной области: Буча, Рамбо и Джекобсона. Разработанная ими универсальная нотация для моделирования объектов (UML - Unified Modeling Language) претендует на роль стандарта в области объектно-ориентированного анализа и проектирования. Конкретный вариант Rational Rose определяется языком, на котором генерируются коды программ (Smalltalk, PowerBuilder, Ada, SQLWindows и ObjectPro). Rational Rose позволяет разрабатывать проектную документацию в виде диаграмм и спецификаций, а также генерировать программные коды на C++. Кроме того, Rational Rose содержит средства реинжиниринга программ, обеспечивающие повторное использование программных компонент в новых проектах.

Методология ARIS рассматривает предприятие как совокупность четырех взглядов: взгляд на организационную структуру, взгляд на структуру функций, взгляд на структуру данных, взгляд на структуру процессов. При этом каждый из этих взглядов разделяется еще на при подуровня: описание требований, описание спецификации, описание внедрения. Таким образом, ARIS предлагает рассматривать организацию с позиции 12 аспектов, отображающих разные взгляды на предприятие, а также разную глубину этих взглядов. Для описания бизнес-процессов предлагается использовать 85 типов моделей, каждая из которых принадлежит тому или иному аспекту. Среди большого количества возможных методов описания можно выделить следующие: EPC (event-driven process chain) - метод описания процессов, нашедший применение для описания процессов системы SAP R/3; ERM (Entity Relationship Model) - модель сущностей-связей для описания структуры данных; UML (Unified Modeling Language) - объектно-ориентированный язык моделирования. ARIS Toolset (ARIS Easy Design) - единая среда моделирования, которая представляет собой совокупность четырех основных компонентов - Explorer (Проводник), Designer (средство для графического описания моделей), Таблиц (для ввода различных параметров и атрибутов) и Мастеров (Wizards). Различия двух продуктов заключается не в методологической части (ARIS Easy Design

входит в ARIS Toolset), а лишь в функционале. ARIS Easy Design ориентирован на сбор информации и документирование, когда ARIS Toolset позволяет еще и проводить комплексный анализ, семантические проверки информации. Кроме того, только ARIS Toolset позволяет создавать скрипты (шаблоны) для отчетов, анализа и семантических проверок.

Таблица 2

Сравнительный анализ и выбор средств инструментальной поддержки организационного проектирования и реинжиниринга бизнес процессов.

Программные продукты	Достоинства	Недостатки
ARIS	1. "Могучая" репрезентативная графика. 2. Наличие большого числа стандартных объектов для описание бизнес процессов. 3. Наличие инструмента имитационного моделирования. 4. Наличие внутреннего языка управления ARIS-Basic. 5. Возможность тестирования проекта на соответствие требования стандарта качества ISO 9000.	1. Невозможность генерации каких-либо кодов или баз данных. 2. Большое количество времени (возможно до 5 мес.) на обучение персонала.
ERwin/BPwin	1. Авторитетность (множество положительных отзывов). 2. Распространенность (99,9%) проектов организационного реинжиниринга исполняются с использованием стандарта IDEF). 3. Возможность генерации исполняемого кода по разработанной модели информационной системы. 4. ERwin 7-й год подряд выбирается лучшим продуктом года по результатам опроса «DBMS Reader's Choice Award!». 5. Относительно низкая стоимость продукта.	1. Репрезентативные свойства низки. 2. Отсутствие стандартных объектов для описания бизнес процессов. 3. Довольно узкие возможности для проведения экономического анализа.
Rational Rose	1. В наибольшей степени подходит для разработки крупных информационных систем. 2. Реализует большую часть функций ARIS и ERwin/BPwin. 3.	1. Отсутствие стандартных объектов для описания бизнес процессов. 2. Цена не соответствует потенциальному риску.

	Мощные функциональные возможности по генерации исполняемых кодов.	

ERwin является средством концептуального моделирования БД, использующее стандарт IDEF1X. ERwin реализует проектирование схемы БД, генерацию ее описания на языке целевой СУБД и реинжиниринг существующей БД. BPwin является средством функционального моделирования, реализующее методологию IDEF0-IDEF3, которая представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Недостатком данной программы являются довольно узкие возможности для проведения экономического анализа, а также отсутствие стандартных объектов для описания бизнес процессов. Методология ARIS позволяет, рассматривает предприятие как совокупность четырех взглядов: взгляд на организационную структуру, взгляд на структуру функций, взгляд на структуру данных, взгляд на структуру процессов. Например, ARIS Easy Design ориентирован на сбор информации и документирование, когда ARIS Toolset позволяет еще и проводить комплексный анализ, семантические проверки информации. Недостатком данной программы является невозможность генерации каких-либо кодов или баз данных и большое количество времени на обучение персонала. Rational Rose предназначено для автоматизации этапов анализа и проектирования ПО, а также для генерации кодов на различных языках и выпуска проектной документации. Недостатком данной программы является отсутствие стандартных объектов для описания бизнес процессов, программа не поддерживает функционально-стоимостной анализ.

Таблица 3

Сравнительный функциональный анализ

Сравнительный функциональный анализ	Функции, свойства	ARIS	ERwin/ BPwin	Rational Rose
1	Моделирование организационных функций и процессов	+	+	+
2	Разработка технического задания	+	+/-	+/-
3	Функционально-стоимостной анализ	+	+	+/-
4	Оптимизация бизнес процессов	+	-	-
5	Имитационное моделирование, событийно-управляемое моделирование	+	+/-	-
6	Генерация кода	-	+	+/-

	приложения			
7	Оформление проектной документации; генерация технологических инструкций для рабочих мест	+	+/-	+
8	Хранение моделей деятельности предприятий	+	+/-	+/-
9	Создание концептуальных и физических моделей структуры базы данных	+/-	+	+
10	Генерация программного кода, SQL-сценариев для создания структуры базы данных.	-	+	+/-
11	Стандартное представление основных бизнес процессов (более 100 типов)	+	-	-
12	Ведение библиотеки типовых бизнес моделей	+	+/-	+/-
13	Групповая работа над проектом	+	+	+
14	Выдача встроенных отчетов по стандарту ISO9000	+	-	-
Ценовые различия	$31 740	$23 685	$40 520	
«+» - да«+/-» - частичная реализация, требующая доработки иными				

инструментальными средствами«-» - нет				

[http://or-rsv.com/Aris-IDEF.htm]

Больше преимуществ в данной таблице выделяется у программного продукта ARIS. Продукт поддерживает имитационное моделирование, моделирование организационных функций и процессов, оптимизацию бизнес - процессов, функционально - стоимостной анализ и другие свойства и функции. Единственным недостатком является отсутствие генерации кода предложения и SQL-сценариев для создания структуры базы данных. На втором месте по функциональности стоит программный продукт ERwin/ BPwin. Здесь в основном наблюдается частичная реализация данных функций. Отсутствует оптимизация бизнес - процессов и стандартное представление основных бизнес процессов. И последнее место занимает программный продукт Rational Rose. В котором очень мало функций и свойств: моделирование организационных функций и процессов, оформление проектной документации, создание концептуальных и физических моделей структуры базы данных, групповая работа над проектом. Остальные функции отсутствуют, либо находятся в доработке.

На основе вышесказанного можно сделать вывод, что у каждой программы есть свои достоинства и недостатки и нельзя сказать, что какая-то программа лучше или хуже, всё зависит от того, какие именно вы хотите внести изменения в свою организацию. Если учитывать цену, то самая дешёвая из предложенных программ это ERwin/ BPwin но у неё больше недостатков, чем у других программ. Самая дорогая обойдётся в 40520$ это Rational Rose у неё меньше возможностей и она в наибольшей степени подходит для разработки крупных информационных систем.

Практическое занятие № 6

Тема 1.5: Цели, задачи и методы исследования программного кода

Цель: Изучить методы исследования программного кода.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Два основных способа исследования программного кода — это дизассемблирование и отладка.

Используя дизассемблер, можно посмотреть, как устроена программа, какие команды и в какой последовательности должны выполняться, к каким функциям идет обращение и т.д. В общем случае дизассемблер не способен восстановить исходный текст программы, написанной на языке высокого уровня, таком как C или Pascal. Результатом работы дизассемблера является (как можно догадаться из названия) эквивалентный текст на языке ассемблера. Для осмысления ассемблерного текста аналитик, разумеется, должен быть хорошо знаком с языком ассемблера и с особенностями той среды, в которой должна выполняться дизассемблируемая программа. Дизассемблер является пассивным инструментом — он никак не воздействует на программу. Самым мощным дизассемблером из существующих на сегодняшний день' можно смело назвать дизассемблер IDA Pro (Interactive DisAssembler), разработанный компанией DataRescue.

Для защиты от дизассемблеров применяются различные методы. Например, если код программы запакован или зашифрован, дизассемблер не сможет увидеть в исследуемом файле настоящие инструкции и окажется бесполезен. Но защищенную таким образом программу можно сначала расшифровать и распаковать, а потом воспользоваться дизассемблером.

Для большинства популярных средств упаковки и шифрования кода исполняемых модулей давно разработаны автоматические или полуавтоматические распаковщики. А для того чтобы узнать, чем именно запакован тот или иной модуль, можно

воспользоваться специальными программами-идентификаторами, которые по некоторым характерным признакам способны опознавать название и версию используемого средства защиты, а также версию компилятора, применявшегося при разработке программы.

Так что реальную сложность для дизассемблирования представляют только программы, которые расшифровывают фрагменты кода динамически, не допуская единовременного присутствия в памяти расшифрованного кода целиком.

В некоторых случаях дизассемблер отказывается работать с исполняемым файлом, если какие-то заголовки файла сформированы с нарушением спецификации, но данный способ также не является надежным.

Иногда код программы модифицируется таким образом, чтобы дизассемблированную последовательность команд было очень трудно анализировать. Например, соседние команды разносятся в разные места, а правильность выполнения организуется за счет большого числа безусловных переходов. Или между командами вставляются произвольные фрагменты кода, не влияющие на результаты вычислений, но отнимающие у человека, выполняющего анализ, уйму времени.

Правда, стоит отметить, что, например, дизассемблер IDA Pro имеет весьма мощные средства расширения (подключаемые модули и язык сценариев), предоставляя тем самым возможность нейтрализовать все попытки противодействия дизассемблированию и последующему анализу.

Отладчик, в отличие от дизассемблера, является активным инструментом и позволяет проследить процесс выполнения по шагам, получая в любой момент всю информацию о текущем состоянии программы или вносить изменения в порядок ее выполнения. Разумеется, отладчик способен показывать дизассемблированные инструкции, состояния регистров, памяти и многое другое. Но наличие отладчика, в силу его активности, может быть обнаружено программой или той ее частью, которая отвечает за защиту. И программа может предпринять ответные действия.

Отладчики бывают трех основных типов: уровня пользователя, уровня ядра и эмулирующие.

Отладчики пользовательского уровня (User-level Debuggers) имеют практически те же возможности, что и отлаживаемая программа. Они используют Debugging API, входящий в состав операционной системы и с его помощью осуществляют контроль над объектом отладки. Отладчики пользовательского уровня входят в состав многих сред разработки, таких как Visual Studio. Они пригодны для исследования незащищенных программ, но могут быть легко обнаружены.

Отладчики уровня ядра (Kernel-mode Debuggers) встраиваются внутрь операционной системы и имеют гораздо больше возможностей, чем отладчики пользовательского уровня. Из ядра операционной системы можно контролировать многие процессы, не доступные другими способами. Одним из самых мощных и часто используемых отладчиков уровня ядра является Softlce, разработанный в компании NuMega Labs (Compuware Corporation). Но и отладчики уровня ядра почти всегда могут быть обнаружены из программы, не имеющей доступа к ядру. Хотя для Softlce, например, был разработан модуль расширения IceExt, позволяющий, среди прочего, неплохо скрывать наличие отладчика в памяти.

Эмулирующие отладчики, пожалуй, являются самым мощным средством исследования кода программ. Такие отладчики эмулируют выполнение всех потенциально опасных действий, которые программа может использовать для выхода из-под контроля исследователя. Однако основная проблема создания эмулирующих отладчиков заключается в том, что иногда им приходится эмулировать реальное периферийное оборудование, а это чрезвычайно сложная задача. Возможно поэтому сейчас нет доступных широкой аудитории эмулирующих отладчиков, хотя существует как минимум два пакета для создания виртуальных компьютеров: VMware, разработанный

одноименной компанией, и VirtualPC, созданный в Connectix Corp. и недавно перешедший в собственность корпорации Microsoft.

Для защиты от отладки программа должна уметь определять наличие отладчика. Для обнаружения того же SoftIce разработано более десяти способов. Но в некоторых случаях можно определить, что программа исследуется при помощи отладчика по косвенным признакам, таким как время выполнения.

В современных процессорах с архитектурой x86 реализована команда RDTSC (Read Time-Stamp Counter). Эта команда позволяет получить количество тактов процессора, прошедших с момента включения питания или последнего сброса. Очевидно, что отладчик тоже является программой. Следовательно, когда защищенная программа исследуется отладчиком, изрядная часть тактов процессора расходуется на выполнение его кода. И если программа знает приблизительное количество тактов, необходимое для выполнения определенного фрагмента кода, то, измерив реально затраченное число тактов, легко обнаружить значительное увеличение времени выполнения, затраченного на отладку.

Для программ, компилируемых в псевдокод, также существуют и отладчики, и декомпиляторы, выдающие исходный текст не на ассемблере, а в некотором ином представлении, пригодном для анализа.

Контрольные вопросы:

1. Основные способы исследования программного кода?
2. Отличие дизассемблирование и отладки?

Практическое занятие № 7

Тема 1.6: Механизмы и контроль внесения изменений в код

1. Управление конфигурацией

Цель: Изучение процесса управления программным кодом и документацией модифицируемых программных систем. Понимать значение управления конфигурацией ПО.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Управление конфигурацией – это процесс разработки и применения стандартов и правил по управлению эволюцией программных продуктов. Эволюционирующие системы нуждаются в управлении по той простой причине, что в процессе их эволюции создается несколько версий одних и тех же программ. В эти версии обязательно вносятся некоторые изменения, исправляются ошибки предыдущих версий; кроме того, версии могут адаптироваться к новым аппаратным средствам и операционным системам. При этом в разработке и эксплуатации могут одновременно находиться сразу несколько версий. Поэтому нужно четко отслеживать все вносимые в систему изменения. Процедуры управления конфигурацией регулируют процессы регистрации и внесения изменений в систему с указанием измененных компонентов, а также способы идентификации различных версий системы. Средства управления конфигурацией применяют для хранения всех версий системных компонентов, для компоновки из этих компонентов системы и для отслеживания поставки заказчикам разных версий системы. Управление конфигурацией нередко рассматривается как часть общего процесса управления качеством. Поэтому иногда одно и то же лицо может отвечать как за управление качеством, так и за управление конфигурацией. Но обычно разрабатываемая программная система сначала контролируется командой по управлению качеством, которая проверяет ПО на соответствие определенным стандартам качества. Далее ПО передается команде по управлению конфигурацией, которая контролирует изменения, вносимые в систему.

Существует много причин, объясняющих наличие разных конфигураций одной и той же системы. Различные версии создаются для разных компьютеров или операционных систем, включающих специальные функции, нужные заказчикам, и т.д. (рис. 1). Менеджеры по управлению конфигурацией обязаны следить за различиями между разными версиями, чтобы обеспечить возможность выпуска следующих вариантов системы и своевременную поставку нужных версий соответствующим заказчикам.

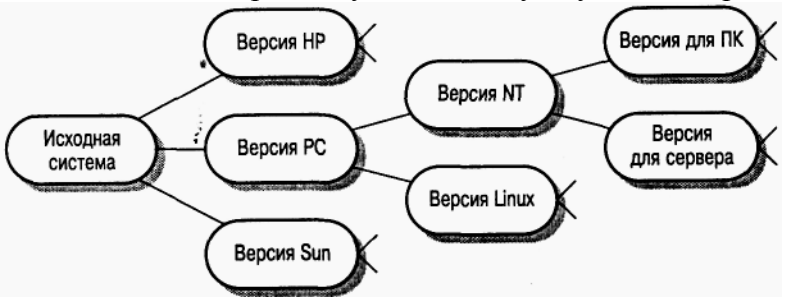


Рис.1. Семейство версий системы

Процесс управления конфигурацией и связанная с ним документация должны подчиняться определенным стандартам. В качестве примера можно привести стандарт IEEE 828-1983, определяющий составление планов управления конфигурацией. Каждая организация должна иметь справочник, в котором указаны эти стандарты, либо они должны входить в общий справочник стандартов качества. Общенациональные или международные стандарты могут быть также использованы как основа для разработки детализированных специальных норм и стандартов для конкретных организаций. За основу можно взять любой тип стандарта, поскольку все они содержат описания однотипных процессов. Для сертификации качества своих программных продуктов организация должна придерживаться официальных стандартов управления конфигурацией, которые приведены в стандартах ISO 9000 и в модели оценки уровня развития SEI.

При традиционной разработке ПО в соответствии с каскадной моделью разрабатываемая система попадает в группу по управлению конфигурацией уже после полного завершения разработки и тестирования ПО. Именно такой подход лежит в основе стандартов управления конфигурацией, которые, в свою очередь, обусловливают необходимость использования для разработки систем моделей, подобных каскадной. Поэтому упомянутые стандарты не в полной мере подходят при использовании таких методов разработки ПО, как эволюционное прототипирование и пошаговая разработка. В этой ситуации некоторые организации изменили подход к управлению конфигурацией, сделав возможным параллельную разработку и тестирование системы. Такой подход основан на регулярной (иногда ежедневной) сборке системы из ее компонентов.

1. Устанавливается время, к которому должна быть завершена поставка компонентов системы (например, к 14.00). Программисты, работающие над новыми версиями компонентов, должны предоставить их к указанному времени. Работу над компонентами не обязательно завершать, достаточно представить основные рабочие функции для проведения тестирования.

2. Создается новая версия системы с новыми компонентами, которые компилируются и связываются в единую систему.

3. После этого система попадает к группе тестирования. В то же время разработчики продолжают работу над компонентами, добавляя новые функции и исправляя ошибки, обнаруженные в ходе предыдущего тестирования.

4. Дефекты, замеченные при тестировании, регистрируются, соответствующий документ пересылается разработчикам. В следующей версии компонента эти дефекты будут учтены и исправлены.

Основным преимуществом ежедневной сборки системы является возможность выявления ошибок во взаимодействиях между компонентами, которые в противном случае могут накапливаться. Более того, ежедневная сборка системы поощряет тщательную проверку компонентов. Разработчики работают под давлением: нельзя прерывать сборку систем и поставлять неисправные версии компонентов. Поэтому программисты неохотно поставляют новые версии компонентов, если они не были предварительно тщательно проверены. Таким образом, на тестирование и исправление ошибок ПО уходит меньше времени.

Для ежедневных сборок системы требуется достаточно строгое управление процессом изменений, позволяющее отслеживать проблемы, которые выявляются и исправляются в ходе тестирования. Кроме того, в результате возникает множество версий компонентов системы, для управления которыми необходимы средства управления конфигурацией.

Планирование управления конфигурацией

В плане управления конфигурацией представлены стандарты, процедуры и мероприятия, необходимые для управления. Отправной точкой создания такого плана является набор общих стандартов по управлению конфигурацией, применяемых в организации-разработчике ПО, которые адаптируются к каждому отдельному проекту. Обычно план управления конфигурацией имеет несколько разделов.

1. Определение контролируемых объектов, подпадающих под управление конфигурацией, а также формальная схема определения этих объектов.
2. Перечень лиц, ответственных за управление конфигурацией и за поставку контролируемых объектов в команду по управлению конфигурацией.
3. Политика ведения управления конфигурацией, т.е. процедуры управления изменениями и версиями.
4. Описание форм записей о самом процессе управления конфигурацией.
5. Описание средств поддержки процесса управления конфигурацией и способов их использования.
6. Определение базы данных конфигураций, применяемой для хранения всей информации о конфигурациях системы.

Распределение обязанностей по конкретным исполнителям является важной частью плана. Необходимо четко определить ответственных за поставку каждого документа или компонента ПО для команд по управлению качеством и конфигурацией. Лицо, отвечающее за поставку какого-либо документа или компонента, должно отвечать и за их разработку. Для упрощения процедур согласования удобно назначать менеджеров проекта или ведущих специалистов команды разработчиков ответственными за все документы, созданные под их руководством.

Определение конфигурационных объектов.

В процессе разработки больших систем создаются тысячи различных документов. Большинство из них – это текущие рабочие документы, связанные с различными этапами разработки ПО. Есть также внутренние записки, протоколы заседания рабочих групп, проекты планов и предложений и т.п. Такие документы представляют разве что исторический интерес и не нужны для дальнейшего сопровождения системы. Для планирования процесса управления конфигурацией необходимо точно определить, какие проектные элементы (или классы элементов) будут объектами управления. Такие элементы называются *конфигурационными элементами.* Как правило, они представляют собой официальные документы. Конфигурационными элементами обычно являются планы проектов, спецификации, схемы системной архитектуры, программы и наборы тестовых данных. Кроме того, управлению подлежат все документы, необходимые для будущего сопровождения системы.

В процессе управления конфигурацией каждому документу необходимо присвоить уникальное имя, причем отображающее связи с другими документами. Для этого используется иерархическая система имен, где они имеют, например, такой вид:
PLC-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ/КОД
PLC-TOOLS/ПРАВКА/СПРАВКА/ЗАПРОС/ОКНО_СПРАВКИ/FR-l
Начальная часть имени – это название проекта PLC-TOOLS. В проекте разрабатываются четыре отдельных средства (рис.2). Имя средства используется в следующей части имени. Каждое средство создается из именованных модулей. Такое разбиение продолжается до тех пор, пока не появится ссылка на официальный документ базового уровня. Листья дерева иерархии документов являются официальными документами проекта. На рис. 2 показано, что для каждого объекта требуется три формальных документа. Это описание объектов (документ ОБЪЕКТЫ), код компонента (документ КОД) и набор тестов для этого кода (документ ТЕСТЫ).

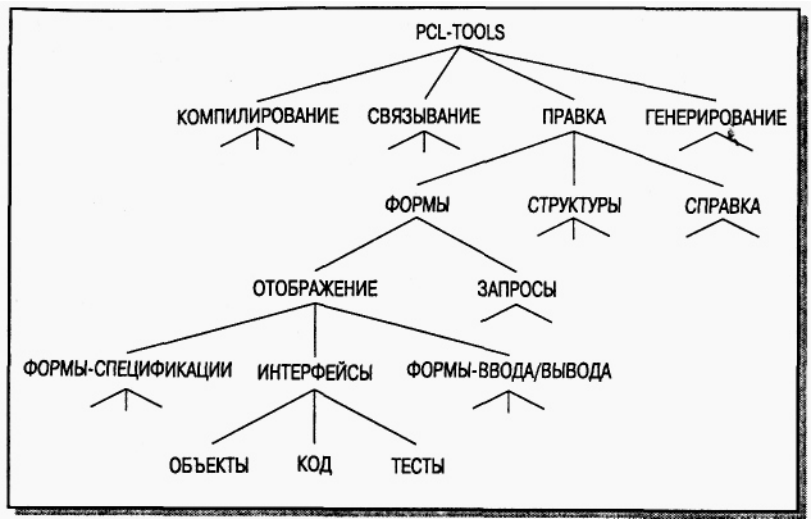


Рис.2. Иерархия конфигурации

Подобные схемы имен основаны на структуре проекта, когда имена соотносятся с соответствующими проектными компонентами. Такой подход к именованию документов порождает определенные проблемы. Например, снижается возможность повторного использования компонентов. Обычно в таких случаях из схемы берутся копии компонентов, которые можно повторно использовать, и переименовываются в соответствии с новой областью применения. Другие проблемы могут появиться, если эта схема именования документов используется как основа структуры хранения компонентов. Тогда пользователь должен знать названия документов, чтобы найти нужные компоненты, при этом не все документы одного типа (например, по проектированию) хранятся в одном месте. Также могут встретиться трудности при установлении соответствия между схемой имен и схемой идентификации, используемой в системе управления версиями.

База данных конфигураций

Такая база данных используется для хранения всей информации о системных конфигурациях. Основными функциями базы данных конфигураций являются поддержка оценивания влияния планируемых изменений в системе и предоставление информации о процессе управления конфигурацией. Задание структуры базы данных конфигураций, определение процедур записи и поиска информации в этой базе данных – все это является частью процесса планирования управления конфигурацией.

Информация, заключенная в базе данных конфигураций, должна помочь ответить на ряд вопросов, среди которых основными и часто запрашиваемыми будут следующие.

• Каким заказчикам поставлена определенная версия системы?

• Какие аппаратные средства и какая операционная система необходимы для работы данной версии системы?

• Сколько было выпущено версий данной системы и когда?

• На какие версии системы повлияют изменения, вносимые в определенный компонент?

• Сколько запросов на изменения было реализовано в данной версии?

• Какое количество ошибок было зарегистрировано в данной версии системы? В идеале база данных конфигураций должна быть объединена с системой управления версиями, которая создается для хранения и управления формальными проектными документами. Такой подход, к тому же поддерживаемый некоторыми интегрированными CASE-средствами, предоставляет возможность связать изменения, вносимые в систему, и с документами, и с теми компонентами, которые подверглись изменениям. В этом случае упрощается поиск измененных компонентов, *поскольку* установлены связи между документами (например, между документами по системной архитектуре и кодом программ) и этими связями можно управлять.

Однако многие организации вместо использования интегрированных CASE-средств для управления конфигурацией рассматривают базу данных конфигураций как отдельную систему. Конфигурационные элементы могут храниться в отдельных файлах или в системе управления версиями, например RCS (известная система управления версиями для Unix). В этом случае в базе данных конфигураций хранится информация о конфигурационных элементах и ссылки на имена соответствующих файлов в системе управления версиями. Несмотря на относительную дешевизну и гибкость такого подхода, основным недостатком его является то, что конфигурационные элементы могут быть изменены без внесения необходимых записей в базу данных. Поэтому нельзя гарантировать, что в базе данных конфигураций содержится обновленная и корректная информация о состоянии системы.

Практическое занятие № 8

Тема 1.6: Механизмы и контроль внесения изменений в код

2.Управление изменениями

Цель: Изучение процесса управления программным кодом и документацией модифицируемых программных систем. Понимать значение управления конфигурацией ПО.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Управление изменениями

Изменения в больших программных системах неизбежны. Как отмечалось в предыдущих главах, в течение жизненного цикла системы изменяются пользовательские и системные требования, а также приоритеты и запросы организаций. Процесс управления изменениями и соответствующие CASE-средства предназначены для того, чтобы зарегистрировать изменения и внести их в систему наиболее эффективным способом.

Процесс управления изменениями начинается после того, как программное обеспечение или соответствующая документация передается команде по управлению конфигурацией. Он может начаться во время тестирования системы или даже после ее поставки заказчику. Процедуры управления изменениями создаются для обеспечения корректного анализа необходимости изменений и их стоимости, а также для контроля за вносимыми изменения.

Процесс управления изменениями

Запрос на изменение, заполнение формы запроса

Анализ запроса

if изменение допустимо **then**

Оценка способа внесения изменения

Оценка стоимости изменения

Запись запроса в базу данных

Передача запроса группе контроля за изменениями

if запрос принят **then**

repeatвсение изменений в ПО регистрация изменениймпередача измененного ПО группе управления качеством

until качество ПО соответствует нормам создание ноной версии системы

elseзапрос на изменение отвергнут

elseзапрос на изменение отвергнут

Первым этапом в процессе управления изменениями является заполнение формы запроса на изменения, в которой указываются те изменения, которые планируется внести в систему. В форме запроса также приводятся рекомендации относительно изменений, предварительная оценка затрат и даты запроса, его утверждения, внедрения и проверки. Также форма может включать раздел, в котором указывается способ выполнения изменения. Запросы на изменения регистрируются в базе данных конфигураций. Таким образом, команда управления конфигурациями может следить за выполнением изменений, а также контролировать изменения определенных программных компонентов.

Сразу после представления заполненной формы запроса проводится проверка необходимости и допустимости изменения. Это объясняется тем, что некоторые изменения вызваны не ошибками в программе, а неправильным пониманием требований, другие могут дублировать исправление ранее обнаруженных ошибок. Если в процессе проверки выявляется, что изменение недопустимо, повторяется или уже было рассмотрено, то изменение отклоняется. Лицу, представившему запрос на изменение, объясняется причина отказа.

Для принятых изменений начинается вторая стадия – оценка изменений и предварительное определение стоимости. Сначала следует проверить влияние изменения на всю систему. Для этого делается технический анализ способа внесения изменения. Затем определяется стоимость внесения изменения в определенные компоненты, что регистрируется в форме запроса. В процессе оценивания полезна база данных конфигураций с информацией о взаимосвязях между компонентами, благодаря чему есть возможность оценить влияние изменений на другие компоненты системы.

Все изменения, кроме тех, которые относятся к исправлению мелких недоработок, должны быть переданы в группу контроля за изменениями, где принимается решение о принятии изменения либо отказе. Эта группа оценивает воздействие изменения не с технической, а скорее с организационной или стратегической точек зрения. Во внимание принимаются такие соображения, как экономическая выгодность изменения и организационные факторы, которые оправдывают необходимость изменения.

Группа контроля за изменениями состоит из лиц, на которых возлагается ответственность за решения о внесении изменений. Такие группы со структурой, включающей старшего менеджера компании-заказчика и сотрудников фирмы-разработчика, обязательны при выполнении военных проектов. Для небольших или среднего размера проектов в эту группу может входить только менеджер проекта и один-два инженера, которые не занимались разработкой данного ПО. В отдельных случаях допускается участие аналитика по изменениям, который дает рекомендации относительно того, оправданы эти изменения либо нет.

После принятия решения о внесении изменений программная система для внесения изменений передается разработчикам или команде по сопровождению системы. По окончании этой процедуры система обязательно должна пройти проверку на правильность внесения изменений. После этого именно команда по управлению конфигурацией, а не разработчики, займется выпуском новой версии.

Изменение каждого компонента системы должно регистрироваться. Таким образом создается история компонента. Самый лучший способ для этого – создавать

стандартизированные комментарии в начале кода компонента, где содержатся ссылки на запросы изменений данного компонента. Для составления отчетов об изменениях компонента и обработки их историй используются специальные средства.

Заголовок компонента

```
// Проект PROTEUS (ESPRIT 6087)
//
// PCL-TOOLS/ПРАВКА/ФОРМЫ/ОТОБРАЖЕНИЕ/ИНТЕРФЕЙСЫ
//
// Объект: PCL-Tool-jbesc
// Автор: Г.Дин
// Дата создания: 10 ноября 1998 г.
//
// © Lancaster University 1998
//
//История изменений
// Версия Кто внес изменения Дата Изменение Причина
// 1.0. Дж. Джонс 1/12/1998 Добавление Предложена
// заголовка группой по
//управлению
//конфигурацией
// 1.1. Г. Дин 9/4/1999 Новое поле Запрос R07/99
```

Практическое занятие № 9

Тема 1.6: Механизмы и контроль внесения изменений в код

3. Управление версиями и выпусками

Цель: Изучение процесса управления программным кодом и документацией модифицируемых программных систем. Понимать значение управления конфигурацией ПО.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Управление версиями и выпусками

Управление версиями и выпусками ПО необходимо для идентификации и слежения за всеми версиями и выпусками системы. Менеджеры, отвечающие за управление версиями и выпусками ПО, разрабатывают процедуры поиска нужных версий системы и следят за тем, чтобы изменения не осуществлялись произвольно. Они также работают с заказчиками и планируют время выпуска следующих версий системы. Над новыми версиями системы должна работать команда по управлению конфигурацией, а не разработчики, даже если новые версии предназначены только для внутреннего использования. Только в том случае, если информация об изменениях в версиях вносится исключительно командой по управлению конфигурацией, можно гарантировать согласованность версий.

Версией системы называют экземпляр системы, имеющий определенные отличия от других экземпляров этой же системы. Новые версии могут отличаться функциональными возможностями, эффективностью или исправлениями ошибок. Некоторые версии имеют одинаковую функциональность, однако разработаны под различные конфигурации аппаратного или программного обеспечения. Если отличия между версиями незначительны, они называются *вариантами* одной версии.

Выходная версия (release) системы – это та версия, которая поставляется заказчику. В каждой выходной версии либо обязательно присутствуют новые

функциональные возможности, либо она разработана под новую платформу. Количество версий обычно намного превышает количество выходных версий, поскольку версии создаются в основном для внутреннего пользования и не поставляются заказчику.

В настоящее время для поддержки управления версиями разработано много разнообразных CASE-средств. С помощью этих средств осуществляется управление хранением каждой версии и контроль за допуском к компонентам системы. Компоненты могут извлекаться из системы для внесения в них изменений. После введения в систему измененных компонентов получается новая версия, для которой с помощью системы управления версиями создается новое имя.

Идентификация версий

Любая большая программная система состоит из сотен компонентов, каждый из которых может иметь несколько версий. Процедуры управления версиями должны четко идентифицировать каждую версию компонента. Существует три основных способа идентификации версий.

1. *Нумерация версий.* Каждый компонент имеет уникальный и явный номер версии. Эта схема идентификации используется наиболее широко.

2. *Идентификация, основанная на знамениях атрибутов.* Каждый компонент идентифицируется именем, которое, однако, не является уникальным для разных версий, и набором значений атрибутов, разных для каждой версии компонента. Здесь версия компонента идентифицируется комбинацией имени и набора значений атрибутов.

3. *Идентификация на основе изменений.* Каждая версия системы именуется так же, как в способе идентификации, основанном на значениях атрибутов, плюс ссылки на запросы на изменения, которые реализованы в данной версии системы. Таким образом, версия системы идентифицируется именем и теми изменениями, которые реализованы в системных компонентах.

Нумерация версий

По самой простой схеме нумерации версий к имени компонента или системы добавляется номер версии. Например, Solaris 2.6 обозначает версию 2.6 системы Solaris. Первая версия обычно обозначается 1.0, последующими версиями будут 1.1, 1.2 и т.д. На каком-то этапе создается новая выходная версия – версия 2.0, нумерация этой версии начинается заново – 2.1, 2.2 и т.д. Эта линейная схема нумерации основана на предположении о последовательности создания версий. Подобный подход к идентификации версий поддерживается многими программными средствами управления версиями, например RCS .

На рис. 3 графически проиллюстрирован описанный способ нумерации версий. Стрелки на рисунке проведены от исходной версии к новой, которая создается на ее основе. Отметим, что последовательность версий не обязательно линейная – версии с последовательными номерами могут создаваться на основе разных базовых версий. Например, на рис. 29.3 видно, что версия 2.2 создана на основе версии 1.2, а не версии 2.1. В принципе каждая существующая версия может служить основой для создания новой версии системы.

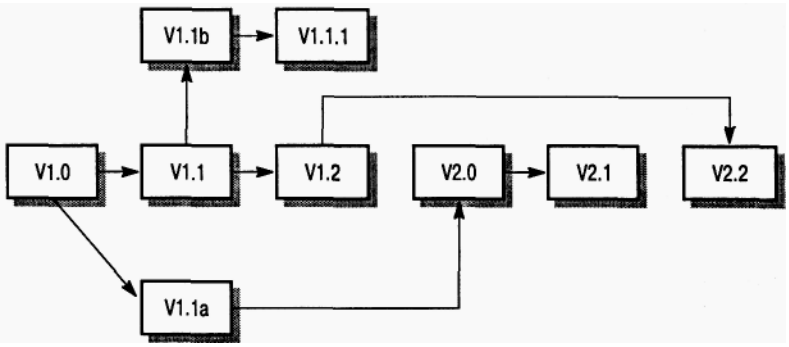


Рис. 3. Структура системных версий

Данная схема идентификации версий достаточно проста, однако она требует довольно большого количества информации для сопоставления версий, что позволяло бы отслеживать различия между версиями и связи между запросами на изменения и версиями. Поэтому поиск отдельной версии системы или компонента может быть достаточно трудным, особенно при отсутствии интеграции между базой данных конфигураций и системой хранения версий.

Идентификация, основанная на значениях атрибутов

Основная проблема схем явного именования версий заключается в том, что такие схемы не отображают тех признаков, которые можно использовать для идентификации версий, например:

- заказчик;
- язык программирования;
- состояние разработки;
- аппаратная платформа;
- дата создания.

Если каждая версия определяется единым набором атрибутов, нетрудно добавить новые версии, основанные на любой из существующих версий, поскольку они будут идентифицироваться единым набором значений атрибутов. При этом значения многих атрибутов новой версии будут совпадать со значениями атрибутов исходной версии; таким образом можно прослеживать взаимоотношения между версиями. Поиск версий осуществляется на основе значений атрибутов. При этом возможны такие запросы, как "самая последняя версия", "версия, созданная между определенными датами" и т.п. Например, обозначение для версии системы AC3D, разработанной на языке Java для использования под управлением Windows NT в январе 1999 года, будет выглядеть следующим образом:

AC3D (язык = Java, платформа = NT4, дата = январь 1999).

Идентификация, основанная на значениях атрибутов, системой управления версиями может применяться непосредственно. Однако более распространено использование только части имени версии, при этом база данных конфигураций поддерживает связь между значениями атрибутов и версиями системы и компонентов.

Идентификация на основе изменений

Идентификация, основанная на значениях атрибутов, устраняет проблему поиска версий, свойственную простым схемам нумерации, когда для поиска версии требуется знание ее атрибутов. Но в этом случае для регистрации взаимосвязей между версиями и изменениями необходимо использование отдельной системы управления изменениями.

Идентификация на основе изменений применяется скорее к системам, чем к системным компонентам; версии отдельных компонентов скрыты от пользователей системы управления конфигурацией. Каждое изменение в системе описывается *массивом изменений,* где указаны изменения в отдельных компонентах, реализующие данное системное изменение. Массивы изменений могут применяться последовательно таким образом, чтобы создать версию системы, в которой реализованы все необходимые изменения. В этом случае не требуется точного обозначения версии. Команда управления конфигурацией работает с системой управления версиями посредством системы управления изменениями.

Естественно, применение нескольких массивов изменений к системе должно быть согласовано, поскольку отдельные массивы изменений могут быть несовместимыми и их последовательное применение может привести к появлению неработоспособной системы. Кроме того, массивы изменений могут конфликтовать, если они предполагают разные изменения в одном компоненте. Для устранения этих проблем применяются средства управления версиями, поддерживающие идентификацию на основе изменений, что позволяет установить точные правила согласованности последовательности

системных версий и что, в свою очередь, ограничивает способы комбинирования массивов изменений.

Управление выходными версиями

Выходной версией системы называется версия, поставляемая заказчику. Менеджеры по выпуску выходных версий отвечают за решение о дате выпуска, за управление процессом создания выходной версии, а также за создание документации.

Выходная версия системы включает в себя не только системный код, но также ряд компонентов.

1. *Конфигурационные файлы,* определяющие способ конфигурирования системы для каждой инсталляции.

2. *Файлы данных,* необходимые для работы системы.

3. *Программа установки,* которая помогает инсталлировать систему.

4. *Документация* в электронном и печатном виде, описывающая систему.

5. *Упаковка и рекламные материалы,* разработанные специально для этой версии системы.

Менеджеры по выпуску выходных версий не могут быть уверены, что заказчики всегда будут заменять старые версии системы новыми. Некоторые пользователи вполне удовлетворены установленными у них версиями и считают, что установка новых версий не стоит затрат. Поэтому новые выходные версии системы не должны зависеть от предыдущих. Рассмотрим следующую ситуацию.

1. Версия 1 системы находится в эксплуатации.

2. Выпускается версия 2, требующая установки новых файлов данных. Однако некоторые пользователи не нуждаются в дополнительных возможностях версии 2 и продолжают использовать версию 1.

3. Версия 3 требует файлов, содержащихся в версии 2, но сама не содержит этих файлов.

Дистрибьютор ПО не может знать наверняка, что файлы данных, требующиеся для версии 3, уже установлены; некоторые пользователи будут переходить от версии 1 к версии 3, минуя версию 2. У других пользователей вследствие каких-либо обстоятельств файлы данных, связанные с версией 2, могут быть изменены. Отсюда следует простой вывод: версия 3 должна содержать все файлы данных.

Принятие решения о выпуске выходной версии

Подготовка и распространение программных систем требуют больших затрат, особенно это касается рынка массовых программных продуктов. Если выпуски выходных версий осуществляются слишком часто, пользователи не успеют осознать потребность в расширенных возможностях новых версий, а если выходные версии создаются редко, существует вероятность потери рынка сбыта, поскольку пользователи переходят к альтернативным системам. Это не относится к программным продуктам, созданным под заказ для определенной организации. Однако и тут редкие выходные версии могут привести к расхождению программной системы и тех бизнес-процессов, для поддержки которых система была разработана.

Принятие решения о том, когда именно должна выйти следующая выходная версия системы, существенно зависит от технических и общих организационных факторов, которые описаны в таблице 1.

Таблица 1. Факторы, влияющие на стратегию выпуска версий системы

Фактор	**Описание**
Техническое качество системы	Необходимость выпуска новой версии обусловлена зарегистрированными ошибками в существующей версии системы. Небольшие дефекты можно устранить с помощью заплат (patches),

	которые часто распространяются через Internet
Пятый закон Лемана	Этот закон постулирует постоянство приращения функциональных возможностей в каждой выходной версии по сравнению с предыдущей. Однако существуют и исключения, например за версией с достаточно большими изменениями следует версия с исправлением ошибок
Конкуренция	Необходимость новой версии объясняется наличием на рынке конкурирующих продуктов
Требования рынка	Отдел маркетинга компании может приурочить выход новой версии к определенной дате
Предложения заказчика об изменениях в системе	Для разработанных под заказ систем заказчик может предложить внести в систему ряд изменений, тогда новая версия выйдет сразу после реализации этих изменений

Создание выходной версии

Создание выходной версии – это процесс сбора всех необходимых файлов и документации, составляющих выходную версию системы. Требуется определить нужные исполняемые коды программ и файлы с данными. Конфигурация выходной версии должна определяться под конкретный тип аппаратных средств и операционной системы. Также нужно подготовить инструкции для пользователей по инсталляции системы, в том числе в электронном виде. Должны быть написаны сценарии для инсталляционной программы. В завершение создается инсталляционный диск, на котором будет распространяться система. В настоящее время в качестве носителей дистрибутивов наиболее широко распространены компакт-диски емкостью до 600 Мбайт.

Документирование выходной версии

Процесс создания выходной версии должен быть задокументирован, чтобы была возможность восстановить ее в будущем. Это особенно важно для больших систем с длинным жизненным циклом, разрабатываемых под заказ. Заказчики обычно используют одну версию системы на протяжении многих лет, и все необходимые изменения вносятся именно в эту версию через много лет после ее поставки.

Для документирования выходной версии прежде всего необходимо записать версии исходного кода компонентов, которые использованы для создания исполняемого кода. Также следует собрать и сохранить все копии исходных и исполняемых кодов, системных данных и конфигурационных файлов. Кроме того, должны быть записаны версии операционной системы, библиотеки, компиляторы и другие средства, применяемые для сборки системы.

Практическое занятие № 10

Тема 1.7: Обратное проектирование. Анализ потоков данных. Дизассемблирование

1.Обратное проектирование

Цель: Изучение классических методов проектирования, обратное проектирование.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Рассмотрим классические методы проектирования, ориентированные на процедурную реализацию программных систем (ПС). Повторим, что эти методы появились в период революции структурного программирования. Обсуждаются только два (наиболее популярных) метода: метод структурного проектирования и метод проектирования Майкла Джексона (этот Джексон не имеет никакого отношения к известному певцу). Зачем мы это делаем? Да чтобы знать исторические корни современных методов проектирования.

Метод структурного проектирования

Исходными данными для метода структурного проектирования являются компоненты модели анализа ПС, которая представляется иерархией диаграмм потоков данных. Результат структурного проектирования — иерархическая структура ПС. Действия структурного проектирования зависят от типа информационного потока в модели анализа.

Типы информационных потоков

Различают 2 типа информационных потоков:

1) поток преобразований;
2) поток запросов.

Как показано на рис. 1, в потоке преобразований выделяют 3 элемента: Входящий поток, Преобразуемый поток и Выходящий поток.

Потоки запросов имеют в своем составе особые элементы — запросы.

Назначение элемента-запроса состоит в том, чтобы запустить поток данных по одному из нескольких путей. Анализ запроса и переключение потока данных на один из путей действий происходит в центре запросов.

Структуру потока запроса иллюстрирует рис. 2.

Рис. 1. Элементы потока преобразований

Рис. 2. Структура потока запроса

Проектирование для потока данных типа «преобразование»

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДД0, словарь данных и спецификации процессов. Оценивается их согласованность с системной спецификацией.

Шаг 2. Проверки и уточнения диаграмм потоков данных уровней 1 и 2. Оценивается согласованность диаграмм, достаточность детализации преобразователей.

Шаг 3. Определение типа основного потока диаграммы потоков данных. Основной признак потока преобразований — отсутствие переключения по путям действий.

Шаг 4. Определение границ входящего и выходящего потоков, отделение центра преобразований. Входящий поток — отрезок, на котором информация преобразуется из внешнего во внутренний формат представления. Выходящий поток обеспечивает обратное преобразование — из внутреннего формата во внешний. Границы входящего и выходящего потоков достаточно условны. Вариация одного преобразователя на границе слабо влияет на конечную структуру ПС.

Шаг 5. Определение начальной структуры ПС. Иерархическая структура ПС формируется нисходящим распространением управления. В иерархической структуре:

- модули верхнего уровня принимают решения;
- модули нижнего уровня выполняют работу по вводу, обработке и выводу;
- модули среднего уровня реализуют как функции управления, так и функции обработки.

Начальная структура ПС (для потока преобразования) стандартна и включает *главный контроллер* (находится на вершине структуры) и три подчиненных контроллера:

1. *Контроллер входящего потока* (контролирует получение входных данных).
2. *Контроллер преобразуемого потока* (управляет операциями над данными во внутреннем формате).
3. *Контроллер выходящего потока* (управляет получением выходных данных).

Данный минимальный набор модулей покрывает все функции управления, обеспечивает хорошую связность и слабое сцепление структуры.

Начальная структура ПС представлена на рис. 3.

Диаграмма потоков данных ПДД

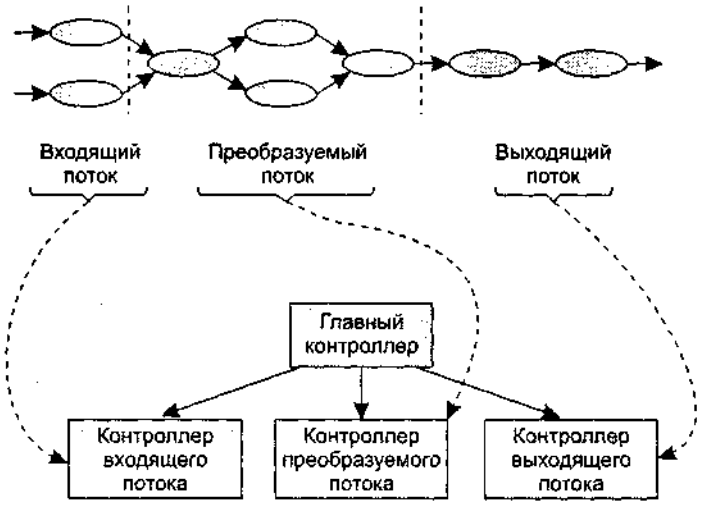


Рис. 3 Начальная структура ПС для потока «преобразование»

Шаг 6. Детализация структуры ПС. Выполняется отображение преобразователей ПДД в модули структуры ПС. Отображение выполняется движением по ПДД от границ

центра преобразования вдоль входящего и выходящего потоков. Входящий поток проходится от конца к началу, а выходящий поток — от начала к концу. В ходе движения преобразователи отображаются в модули подчиненных уровней структуры (рис. 4).

Центр преобразования ПДД отображается иначе (рис. 5). Каждый преобразователь отображается в модуль, непосредственно подчиненный контроллеру центра.

Проходится преобразуемый поток слева направо.

Возможны следующие варианты отображения:

- 1 преобразователь отображается в 1 модуль;
- 2-3 преобразователя отображаются в 1 модуль;
- 1 преобразователь отображается в 2-3 модуля.

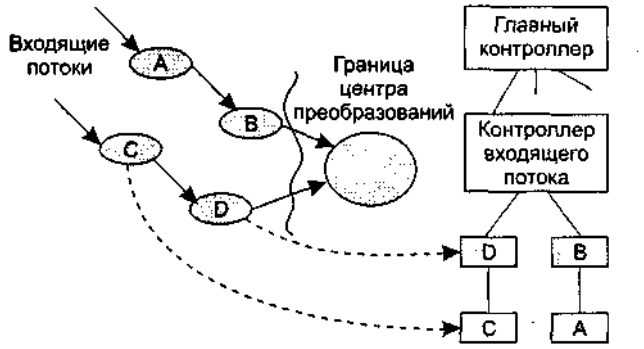


Рис. 4. Отображение преобразователей ПДД в модули структуры

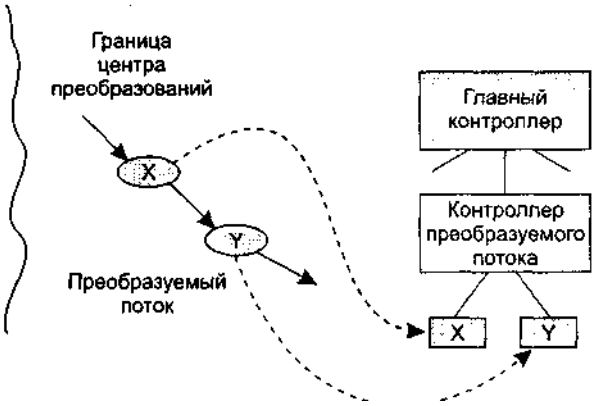


Рис. 5. Отображение центра преобразования ПДД

Для каждого модуля полученной структуры на базе спецификаций процессов модели анализа пишется сокращенное описание обработки.

Шаг 7. Уточнение иерархической структуры ПС. Модули разделяются и объединяются для:

1) повышения связности и уменьшения сцепления;
2) упрощения реализации;
3) упрощения тестирования;
4) повышения удобства сопровождения.

Проектирование для потока данных типа «запрос»

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДДО, словарь данных и спецификации процессов. Оценивается их согласованность с системной спецификацией.

Шаг 2. Проверки и уточнения диаграмм потоков данных уровней 1 и 2. Оценивается согласованность диаграмм, достаточность детализации преобразователей.

Шаг 3. Определение типа основного потока диаграммы потоков данных. Основной признак потоков запросов — явное переключение данных на один из путей действий.

Шаг 4. Определение центра запросов и типа для каждого из потоков действия. Если конкретный поток действия имеет тип «преобразование», то для него указываются границы входящего, преобразуемого и выходящего потоков.

Шаг 5. Определение начальной структуры ПС. В начальную структуру отображается та часть диаграммы потоков данных, в которой распространяется поток запросов. Начальная структура ПС для потока запросов стандартна и включает входящую ветвь и диспетчерскую ветвь.

Структура входящей ветви формируется так же, как и в предыдущей методике.

Диспетчерская ветвь включает диспетчер, находящийся на вершине ветви, и контроллеры потоков действия, подчиненные диспетчеру; их должно быть столько, сколько имеется потоков действий.

Диаграмма потоков данных

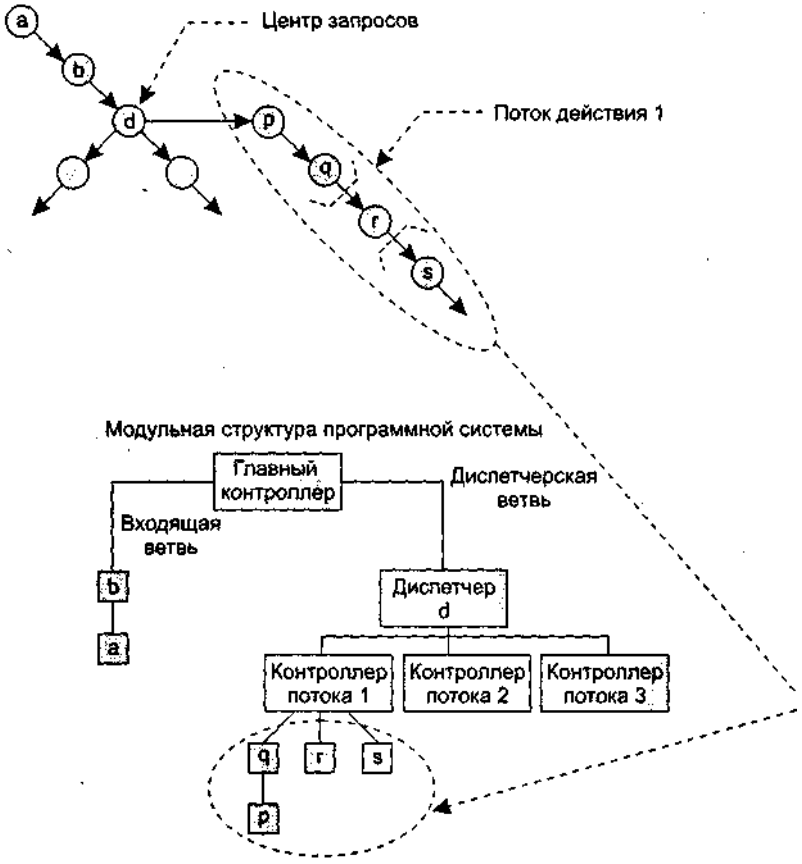


Рис. 6. Отображение в модульную структуру ПС потока действия 1

Шаг 6. Детализация структуры ПС. Производится отображение в структуру каждого потока действия. Каждый поток действия имеет свой тип. Могут встретиться поток-«преобразование» (отображается по предыдущей методике) и поток запросов. На рис. 6 приведен пример отображения потока действия 1. Подразумевается, что он является потоком преобразования.

Шаг 7. Уточнение иерархической структуры ПС. Уточнение выполняется для повышения качества системы. Как и при предыдущей методике, критериями уточнения служат: независимость модулей, эффективность реализации и тестирования, улучшение

сопровождаемости.

Метод проектирования Джексона

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

Метод Джексона включает шесть шагов. Три первых шага относятся к этапу анализа. Это шаги: *объект — действие, объект — структура, начальное моделирование.*

Доопределение функций

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

1. Встроенные функции (задаются командами, вставляемыми в структурный текст процесса-модели).

2. Функции впечатления (наблюдают вектор состояния процесса-модели и вырабатывают выходные результаты).

3. Функции диалога.

Они решают следующие задачи:

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

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

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

прибыть(1);
конец ТРАНСПОРТ-1;

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

Условия выработки этих команд.

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

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

Ясно, что процесс М-УПРАВЛЕНИЕ должен иметь внешние связи с моделями ТРАНСПОРТ-1 и КНОПКА. Соединение с моделью КНОПКА организуем через вектор состояния BV. Соединение с моделью ТРАНСПОРТ-1 организуем через поток данных S1D.

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

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

Очевидно, что при такой связи процессов необходимо гарантировать, что процесс ТРАНСПОРТ-1 выполняет операции опрос SV, а процесс М-УПРАВЛЕНИЕ читает

сообщения Прибытия в S1D с частотой, достаточной для своевременной остановки транспорта. Временные ограничения, планирование и реализация должны рассматриваться в последующих шагах проектирования.

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

Диаграмма дополнительного процесса КНОПКА-2, в котором учтено это уточнение, показана на рис. 7.

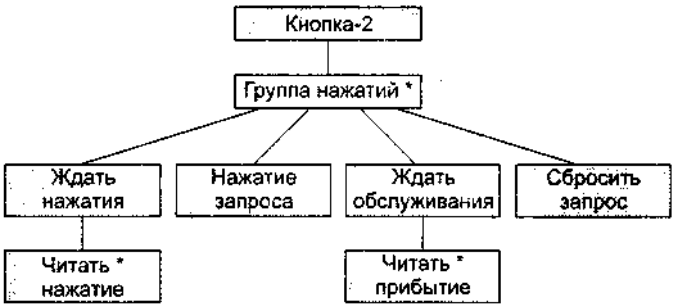


Рис. 7. Диаграмма дополнительного процесса КНОПКА-2

Внешние связи модели КНОПКА-2 должны включать:

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

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

Структурный текст модели КНОПКА-2 может иметь следующий вид:

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

Диаграмма системной спецификации, отражающая все изменения, представлена на рис. 8.

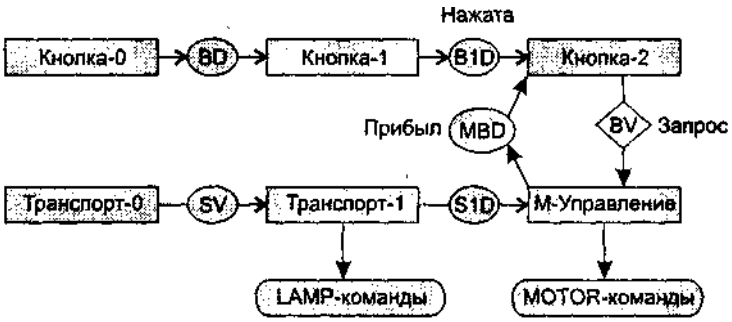


Рис. 8. Полная диаграмма системной спецификации

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

Учет системного времени

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

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

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

Контрольные вопросы

1. В чем состоит суть метода структурного проектирования?
2. Какие различают типы информационных потоков?
3. Что такое входящий поток?
4. Что такое выходящий поток?
5. Что такое центр преобразования?
6. Как производится отображение входящего потока?
7. Как производится отображение выходящего потока?
8. Как производится отображение центра преобразования?

Практическое занятие № 11

Тема 1.7: Обратное проектирование. Анализ потоков данных. Дизассемблирование

2. Анализ потоков данных. Дизассемблирование

Цель: Изучение классических методов проектирования, обратное проектирование.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Базовые понятия дизассемблирования.

Что такое дизассемблирование. Дизассемблирование – преобразование программы на машинном языке к ее ассемблерному представлению. Декомпиляция – получение кода языка высокого уровня из программы на машинном языке или ассемблере. Декомпиляция – достаточно сложный процесс. Это обусловлено следующими причинами: − Процесс компиляции происходит с потерями. В машинном языке нет имен переменных и функций, и тип данных может быть определен только по производимым над ними операциям. Наблюдая пересылку 32-х бит данных, требуется значительная работа, чтобы определить, являются ли эти данные целым числом, дробью или указателем. − Компиляция это операция типа множество-множество. Компиляция и декомпиляция могут быть выполнены множеством способов. Поэтому результат декомпиляции может значительно отличаться от исходного кода. − Декомпиляторы в значительной степени зависимы от конкретного языка и библиотек. Обрабатывая исполняемый файл, созданный компилятором Delphi, декомпилятором, разработанным для C, можно получить фантастический результат. − Необходимо точное дизассемблирование исполняемого файла. Любая ошибка или упущение на фазе дизассемблирования практически наверняка размножатся в результирующем коде. Прогресс средств декомпиляции происходит медленно, но верно. Наиболее сложный на сегодняшний день декомпилятор IDA, будет рассмотрен ниже. Зачем нужно дизассемблирование.

Цель инструментов дизассемблирования заключается в содействии исследованию функционирования программ, когда их исходные коды не доступны. Наиболее распространенные цели дизассемблирования: − анализ вредоносного программного обеспечения; − анализ уязвимостей программного обеспечения с закрытым исходным кодом; − анализ совместимости программного обеспечения с закрытым исходным кодом; − валидация компилятора; − отображение команд программы в процессе отладки. Анализ вредоносного программного обеспечения. Разработчики вредоносного программного обеспечения вряд ли предоставят исходный код своего детища. А без доступа к исходному коду, будет довольно сложно 6 определить поведение вредоносной программы. Существуют два основных метода исследования вредоносных программ - динамический и статический анализ. Динамический анализ заключается в исполнении программы в тщательно контролируемом окружении (песочнице) и записи каждого ее действия. Статический анализ основан на разборе исходного кода, который, в случае вредоносной программы, в основном состоит из дизассемблированных листингов. Анализ уязвимостей. Для простоты, можно разделить процесс аудита безопасности на три стадии: поиск уязвимостей, анализ уязвимостей, и разработка эксплойта. Одни и те же шаги предпринимаются вне зависимости от того, имеется ли у вас исходный код; однако, уровень трудоемкости резко возрастает, когда в вашем распоряжении есть лишь исполняемый файл. Первый шаг - исследование потенциально уязвимых условий в программе. Это зачастую достигается использованием динамических техник, таких как фаззинг, однако также может быть реализовано (обычно со значительно большими усилиями) посредством статического анализа. Как только проблема обнаружена, требуется определить, является ли она уязвимостью и если да, то при каких условиях. Листинг дизассемблирования помогает понять, каким образом компилятор расположил переменные в памяти. Например, может быть полезно узнать, что объявленный программистом 70-байтный массив символов при распределении памяти компилятором был округлен в сторону 80 байт. Листинги дизассемблирования также предоставляют единственный способ понять, объявлены ли переменные глобально или внутри функций.

Понимание реального расположения переменных в памяти жизненно важно при разработке эксплойтов. Анализ совместимости. Когда программы доступны только в виде исполняемых файлов, сторонним разработчикам крайне сложно обеспечить совместимость с ними своих программ, а также расширить их функциональность. Например, если производитель не предоставил драйвер для аппаратного устройства, то реверс инжиниринг – практически единственное средство для разработки альтернативных драйверов. Валидация компилятора. Дизассемблирование может быть средством для проверки соответствия работы компилятора его спецификации. Также исследователя может заинтересовать наличие дополнительных возможностей, оптимизирующих результат компиляции. С точки зрения безопасности важно быть уверенным, что код, генерируемый компилятором, не содержит черных ходов. Отладка. К сожалению, дизассемблеры, встроенные в отладчики, зачастую малоэффективны (OllyDbg ─ исключение). Они неспособны к серийному дизассемблированию и иногда отказываются дизассемблировать, не будучи в состоянии определить границы функции. Поэтому, для лучшего контроля над процессом отладки, лучше использовать отладчик в сочетании с хорошим дизассемблером. 7 Как дизассемблировать. Типичные задачи, с которыми сталкивается дизассемблер: взять 100 КБ из исходного файла, отделить код от данных, преобразовать код к языку ассемблера, и главное ничего не потерять. В этот список можно добавить дополнительные пожелания, например, определение границ функций, распознавание таблиц переходов, выделение локальных переменных... Это значительно усложнит его работу. Качество результирующих листингов дизассемблирования определяется свойствами алгоритмов, а также уместностью их применения в конкретной ситуации. 1.2. Базовый алгоритм дизассемблирования Шаг 1. Первым шагом в процессе дизассемблирования является идентификация кодового сегмента. Так как команды обычно смешаны с данными, то дизассемблеру необходимо их разграничить. Шаг 2. Получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле) и выполнить табличное преобразование двоичного кода операции в соответствую ему мнемонику языка ассемблера. Шаг 3. Как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу. После этого необходимо выбрать одну из разновидностей синтаксиса языка ассемблера. Шаг 4. Далее необходимо перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

Практическое занятие № 12

Тема 2.1: Утилиты для review: обзор

Цель: Разобрать понятие утилита, обзорутилиты для review.

Утилита (досл. от англ. "utility" – "полезность") – в контексте компьютерных технологий обычно небольшая программа, выполняющая одну, чаще всего, связанную с обслуживанием ПК или операционной системы, функцию. Они могут как изначально быть частью ОС (например, штатный дефрагментатор жёсткого диска или диспетчер задач), так и поставляться сторонними разработчиками.

Для своей работы утилиты могут использовать либо полностью собственные ресурсы, либо часть функционала системных инструментов ОС (например, Командную

строку). Последний факт роднит их с плагинами. Однако, в отличие от плагинов, утилиты являются в большинстве своём самостоятельными программами и, хотя бы для выполнения базовых задач, могут работать автономно.

В обзоре ниже предлагаем рассмотреть несколько интересных бесплатных комплектов утилит, которые позволят Вам значительно расширить функционал операционной системы, настроить её под свои нужды и получить ряд полезной информации о своём компьютере.

Code review - инженерная практика в терминах гибкой методологии разработки. Это анализ (инспекция) кода с целью выявить ошибки, недочеты, расхождения в стиле написания кода, в соответствии написанного кода и поставленной задачи.

К очевидным плюсам этой практики можно отнести:

- Улучшается качество кода
- Находятся «глупые» ошибки (опечатки) в реализации
- Повышается степень совместного владения кодом
- Код приводится к единому стилю написания
- Хорошо подходит для обучения «новичков», быстро набирается навык, происходит выравнивание опыта, обмен знаниями.

Что можно инспектировать.

Для ревью подходит любой код. Однако, review обязательно должно проводиться для критических мест в приложении (например: механизмы аутентификации, авторизации, передачи и обработки важной информации — обработка денежных транзакций и пр.).

Также для review подходят и юнит тесты, так как юнит тесты — это тот же самый код, который подвержен ошибкам, его нужно инспектировать также тщательно как и весь остальной код, потому что, неправильный тест может стоить очень дорого.

Как проводить review

Вообще, ревью кода должен проводиться в совокупности с другими гибкими инженерными практиками: парное программирование, TDD, CI. В этом случае достигается максимальная эффективность ревью. Если используется гибкая методология разработки, то этап code review можно внести в Definition of Done фичи.

Из чего состоит review

- Сначала design review — анализ будущего дизайна (архитектуры).Данный этап очень важен, так как без него ревью кода будет менее полезным или вообще бесполезным (если программист написал код, но этот код полностью неверен — не решает поставленную задачу, не удовлетворяет требованиям по памяти, времени). Пример: *программисту поставили задачу написать алгоритм сортировки массива. Программист реализовал алгоритм bogo-sort, причем с точки зрения качества кода — не придраться (стиль написания, проверка на ошибки), но этот алгоритм совершенно не подходит по времени работы. Поэтому ревью в данном случае бесполезно (конечно — это утрированный пример, но я думаю, суть ясна), здесь необходимо полностью переписывать алгоритм*.
- Собственно, сам code review — анализ написанного кода. На данном этапе автору кода отправляются замечания, пожелания по написанному коду.

Также очень важно определиться, за кем будет последнее слово в принятии финального решения в случае возникновения спора. Обычно, приоритет отдается тому кто будет реализовывать код (как в scrum при проведении planning poker), либо специальному человеку, который отвечает за этот код (как в google — code owner).

Как проводить design review

Design review можно проводить за столом, в кругу коллег, у маркерной доски, в корпоративной wiki. На design review тот, кто будет писать код, расскажет о выбранной стратегии (примерный алгоритм, требуемые инструменты, библиотеки) решения поставленной задачи. Вся прелесть этого этапа заключается в том, что ошибка проектирования будет стоить 1-2 часа времени (и будет устранена сразу на review).

Как проводить code review

Можно проводить code review разными способами — дистанционно, когда каждый разработчик сидит за своим рабочим местом, и совместно — сидя перед монитором одного из коллег, либо в специально выделенным для этого месте, например meeting room. В принципе существует много способов (можно даже распечатать исходный код и вносить изменения на бумаге).

Pre-commit review

Данный вид review проводится перед внесением изменений в VCS. Этот подход позволяет содержать в репозитории только проверенный код. В microsoft используется этот подход: всем участникам review рассылаются патчи с изменениями. После того как собран и обработан фидбэк, процесс повторяется до тех пор пока все ревьюверы не согласятся с изменениями.

Post-commit review

Данный вид review проводится после внесения изменений в VCS. При этом можно коммитить как в основную ветвь, так и во временную ветку (а в основную ветку вливать уже проверенные изменения).

Тематические review

Можно также проводить тематические code review — их можно использовать как переходный этап на пути к полноценному code review. Их можно проводить для критического участка кода, либо при поиске ошибок. Самое главное — это определить цель данного review, при этом цель должна быть обозримой и четкой:

- "*Давайте поищем ошибки в этом модуле*" — не подходит в качестве цели, так как она необозрима.
- "*Анализ алгоритма на соответствие спецификации RFC 1149*" — уже лучше.

Основное отличие тематических review от полноценного code review — это их узкая специализация. Если в code review мы смотрим на стиль кода, соответствие реализации и постановки задачи, поиск опасного кода, то в тематическом review мы смотрим обычно только один аспект (чаще всего — анализ алгоритма на соответствие ТЗ, обработка ошибок).

Преимущество такого подхода заключается в том, что команда постепенно привыкает к практике review (его можно использовать нерегулярно, по требованию). Получается некий аналог мозгового штурма. Мы использовали такой подход при поиске логических ошибок в нашем ПО: смотрели «старый» код, который был написан за несколько месяцев до review (это можно отнести тоже к отличиям от обычного review — где обычно смотрят свежий код).

Результаты review

Самое главное при проведении review — это использование полученного результата. В результате review могут появиться следующие артефакты:

- Описание способа решения задачи (design review)
- UML диаграммы (design review)
- Комментарии к стилю кода (code review)
- Более правильный вариант (быстрый, легкочитаемый) реализации (design review, code review)

- Указание на ошибки в коде (забытое условие в switch, и т.д.) (code review)
- Юниттесты (design review, code review)

При этом очень важно, чтобы все результаты не пропали, и были внесены в VCS, wiki. Этому могут препятствовать:

- Сроки проекта.
- Лень, забывчивость разработчиков
- Отсутствие удобного механизма внесения изменений review, а также контроль внесения этих изменений.

Для преодоления этих проблем частично может помочь:

- pre-commit hook в VCS
- Создание ветви в VCS, из которой изменения вливаются в основную ветвь только после review
- Запрет сборки дистрибутива на CI сервере без проведения review. Например, при сборке дистрибутива проверять специальные свойства (svn:properties), либо специальный файл с результатами review. И отказывать в сборке дистрибутива, если не все ревьюверы одобрили (approve) код.
- Использование методологии в разработке (в которой code review является неотъемлемой частью).

Сложности при проведении review (субъективное мнение)

Основная сложность, с которой мы столкнулись, когда внедряли review в первый раз: это невозможность контроля изменений по результатам review. Отчасти это связано с тем, что данная практика применялась без других практик — CI (это еще раз доказывает тот факт, что все инженерные практики должны применяться вместе).

Утилиты для review

Вообще, утилит для проведения review существует большое количество, как платных, так и бесплатных. Я не стал их приводить, чтобы не навязывать свою точку зрения, в интернете можно найти множество инструментов и подробное описание.

Практическое занятие № 13

Тема 2.2: Предпроцессинг кода. Интеграция в IDE

Цель: Изучить предпроцессинг кода. Интеграция в IDE

Предпроцессинг кода.

Совершенный код… многие ищут этот Грааль, но реальность всегда вносит коррективы, доказывая, что единого совершенства не бывает. Но все же бывают стандарты, стандарты кодирования, как минимум, которым весь код в проекте должен соответствовать. И такая, казалось бы, простая вещь порой все равно не соблюдается. Так почему бы не призвать возможные инструменты на помощь?

Что и чем же мы хотели бы проверять.

1. Проверка синтаксиса PHP, HTML, JS, CSS.

PHP — тут все просто, сам интерпретатор PHP неплохо справляется с этой задачей, php -l $file.

HTML, CSS — с этим немного сложнее, существует множество онлайн-сервисов, но большинство из нас пишет коммерческий код и светить им на открытых сервисах… в общем, это совсем не то. Можно посмотреть в сторону сервисов на http://w3.org/.

Для HTML:

http://validator.w3.org/docs/install.html

Можно скачать исходные коды, развернуть сервис у себя на сервере, по ссылке выше довольно подробно описано, как это сделать. Исходные коды на Java.

http://www.bioinformatics.org/phplabware/internal_utilities/htmLawed/

Утилита, созданная компанией PHP Labware, вполне ожидаемо написана на PHP, последняя версия поддерживает HTML5-синтаксис (но требует доработки напильником). Также позволяет не только проводить валидацию, но и готовить чистый Html на выходе, что впоследствии может быть использовано для организации онлайн Html редакторов, прочего.

Для CSS:

Тут стоит упомянуть или даже настоятельно рекомендовать использовать препроцессоры CSS (SASS/SCSS, LESS), очень многие тренд-frontend-решения (Bootstrap, Foundation) используют препроцессоры кода. Они действительно позволяют задать стиль ваших CSS, помогают избежать банальных ошибок, дают возможность сократить количество кода, и еще много всяких полезных вещей. Попробуйте писать с использованием препроцессоров и поймете, насколько это упрощает работу.

Список инструментов, которые могут пригодиться:

http://jigsaw.w3.org/css-validator/DOWNLOAD.html

Доступен для загрузки, описание присутствует. Исходные коды на Java.

https://github.com/CSSLint/csslint

Для валидации CSS также можно использовать эту отдельную утилиту, хотя в последних версиях PHP_CodeSniffer для нее написан отдельный сниф. Довольно гибкая и хорошо настраиваемая. Кроме синтаксиса, может выводить стилевые ошибки. Исходные коды на JS.

http://code.w3.org/unicorn/wiki/Documentation/Install

Совмещенный валидатор (HTML, CSS).

http://csstidy.sourceforge.net/

Скорее, бьютифаер CSS, позволяет преобразовывать CSS, согласно набору стандартных правил, имеет возможность добавления своих правил, широкий спектр настроек, доступен в 2 вариантах C++ исполняемый файл и PHP библиотека.

2. Проверка стиля PHP, JS.

Для PHP:

PHP_CodeSniffer (вместе с PHP сейчас может проверять JS, CSS используя утилиты, описанные в обзоре). Заслуженно может занимать первое место по популярности, имеет обширный набор настроек — предподготовленных стандартов. Написан на PHP. Для реализации своего стандарта можно отнаследоваться от существующего и расширить своими настройками, или написать стандарт с нуля. В общем, вещь, достойная отдельной статьи.
PHPMD — собирает информацию о неиспользованном коде (еще для этого можно использовать PHPDCD), неоптимальном коде.

PHPCPD — любитель копипаста не пройдет. Утилита позволяет находить повторяющийся код в ваших PHP-файлах.

Для JS:

Closure Linter — валидатор Стиля JS от Google, написан на Python. Кроме валидации, может выдавать на выход исправленный файл.

JSHint — проверяет JS-файлы на наличие ошибок и потенциальных проблем, для установки требуются node.js и npm.

JSLint — еще один, и довольно неплохой валидатор JS с проверкой стиля, написанный на JS одним из идейных лидеров фронтенда и JS как языка в частности Дугласом Крокфордом.

JavaScript Lint — скорее, расширенная версия JSLint, от разработчиков Firefox. Написан на C, доступен и скомпилированным, и исходным кодом.

Так, с инструментами вроде разобрались.

Но почему же все-таки предпроцессинг, спросите вы, и как его готовить? Попробуем разобраться.

Применение, способы и практики:

Самым первым в очереди предпроцессинга кода у любого разработчика выступает IDE. У большинства IDE сегодня есть базовые анализаторы кода, которые, конечно, уже являются первым барьером (после прямоты рук разработчика), препятствующим банальным ляпам при разработке приложений.

Но не всегда в web-разработке (и не только) код пишется в IDE, которая предупредит об ошибке. Не всегда разработчик обращает внимания на некоторые предупреждения, да и IDE банально может не быть настроена на, к примеру, правильное количество пробелов или кодировку файлов, которые определяются в Code Convection проекта. Читать потом такой код — не самое приятное занятие, а привести код проектов «в возрасте» к какому-то стандарту вообще не представляется возможным. Для предотвращения этих досадных моментов стоит задуматься о внедрении практики предпроцессинга кода.

Выделим следующие разновидности:

1. Интеграция в IDE.

Имею в виду плагины или уже входящие в комплект IDE хендлеры утилит, описанных выше. Вариант, вполне подходящий для разработки в одиночку, и точно должен быть у любого. Главное — до начала работы с свежеустановленной IDE не забывать заглядывать в настройки и прописывать там пути к утилитам, выставлять необходимый код конвеншн. Ну и, конечно, в случае командной разработки, стоит сохранить экспорт настроек, наиболее используемых участниками проекта IDE, в репозиторий — сэкономим время других на настройку проектных практик оформления кода.

2. Pre-Commit (client side check)

Преимущество этого метода и следующих — возможность централизации процесса контроля качества кода. Т. е скрипты, которые выполняют проверки мо,гут лежать в отдельном общедоступном репозитории/ветке/папке, и тут уже не спишешь проблемы в коде на свежую IDE.

Итак, кто еще не знаком с системами хуков в системах контролей версий, — самое время познакомиться. Конечно, децентрализованные системы контроля версий имеют определенную специфику и в хуках (например, в GIT на клиентской стороне вы можете добавлять свою кастомную проверку и после каждого комита, и до операции Push, начиная с версии 1.8.2). Но попробуем немного обобщить.

Для начала стоит определиться, что уже реализовано в проверках на уровне 1. Хотя как раз для проверки на клиентской части на этапе хука вполне может перекрывать проверки на уровне IDE. И предположить, что будет лучше проверять на этапе Pre-Update.

Могу предложить следующие проверки на этом уровне:

- Валидация JS, CSS (если планируется использовать PHP_CodeSniffer, см. следующий этап). Т. е. если вы ошиблись в коде, JS, CSS, не будете тратить время и ресурсы сервера контроля версий на то чтобы поместить ваши ошибки в нем.
- Валидация синтаксиса PHP и проверка кода на наличие устаревшего и дублирующего кода, если используется Git, конечно, лучше вешать эту проверку на Pre-Push, т. к. она более актуальна для больших объемах кода, а все мы верим в «атомарность» коммитов.
- Если окружение и не очень большой объем репозитория позволяют, можно повесить на Pre-Push и запуск юнит-тестов.
- Валидация формата commit-сообщений (если про Git, у него есть отдельный hook-тип — “commit-msg”).
- Каждый может поиграть с уровнем строгости проверок. Разумеется, тут не стоит сильно увлекаться, но и особо «ленивыми» проверки тоже делать не стоит. Вообще, на данном этапе допускается пропуск проверок, но только посредством специальных

вставок в коде, которые поддерживают все вышеописанные утилиты, например в PHP_CodeSniffer это будет выглядеть так:

// *@codingStandardsIgnoreStart* class MyClassTest extends \PHPUnit_Framework_TestCase { // *@codingStandardsIgnoreEnd* // ... }

Т. е. все равно эти комментарии потом всплывут на этапе Review. И вот тогда, возможно, будет пересмотрен участок кода или, по крайней мере, будет напоминание, что тут есть отступление от правил, которое потом можно будет мониторить.

3. Pre-Update (server side check)

Добавляется на стороне сервера системы контроля версий, тут уже есть возможность запускать проверки, которые зависят от окружения (особенно если разработка ведется с участием удаленного Web сервера, такие, как запуск юнит-тестов и т. п. Но тут уже все зависит от объемов тестов: если времени на их обработку уходит много, стоит перенести проверки в Build). ОБЯЗАТЕЛЬНЫ проверки формата сообщений, ACL-права на возможность изменять определенные директории отдельными юзерами, просто на возможность записи в отдельно взятые ветки.

4. During Build check

Отдельно стоит вынести ряд проверок, запускаемый во время Build-процесса (Build как одна из составляющих CI-процесса): запуск автотестов (Unit, Functional, UI), анализаторы кода (PHPMD, PHPCPD, т. е. требующие большего количества кода, чем код на один коммит), автодокументирование и прочие полезности, которые занимают время и ресурсы сервера CI. Хорошо что Build можно настроить на запуск, к примеру, ночью.

Заключение

Это еще не выводы, а пока только заключение, — выводы можно будет найти во 2-й части статьи «Роль пред-подготовки кода в CI: часть 2. Доступные инструменты для автоматизации Review», которая пока в работе. Но искренне надеюсь, что хоть один из вышеперечисленных инструментов сможет сэкономить хоть чуть-чуть вашего бесценного времени.

Практическое занятие № 14

Тема 2.3: Валидация кода на стороне сервера и разработчика

Цель: Разобрать понятие валидация кода. Научиться писать собственные реализации провайдеров метаданных.

Валидация на стороне сервера должна осуществляться независимо от того, включена валидация на стороне клиента или нет. Пользователь может отключить JavaScript или совершить какие-нибудь другие непредвиденные действия, чтобы обойти валидацию на стороне клиента, и сервер останется последней линией защиты наших данных от некорректного ввода. Некоторые правила валидации требуют обработки данных на стороне сервера - топологией сети может быть установлено, что только сервер имеет доступ к внешним ресурсам, необходимым для проверки вводимых данных.

Мы рассмотрим два ключевых понятия. Сначала разберем самый распространенный способ валидации на стороне сервера с ASP.NET MVC, используя Data Annotations. Потом мы исследуем метаданные модели и научимся писать собственные реализации провайдеров метаданных.

Валидация с Data Annotations

Библиотека Data Annotations, впервые реализованная в пакете .NET 3.5 SP1, представляет собой набор атрибутов и классов, определенных в сборке System.ComponentModel.DataAnnotations, которые позволяют добавить метаданные к классам. Метаданные описывают набор правил, с помощью которых можно определить, как проводить проверку конкретных объектов.

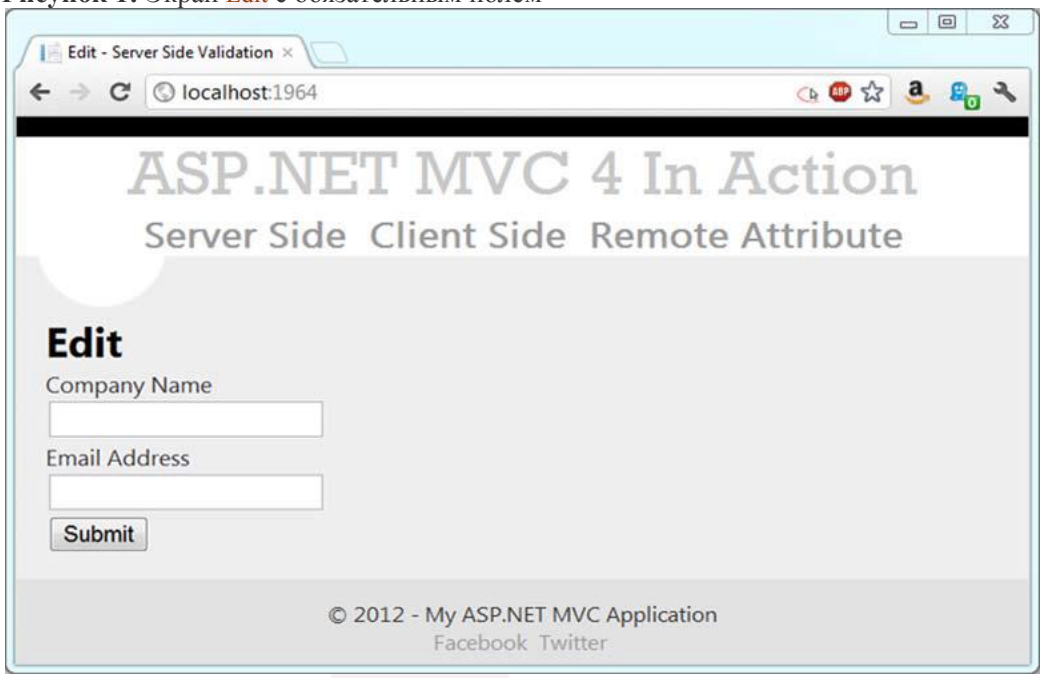
Кроме описания правил валидации, атрибуты DataAnnotations используются для реализации новых шаблонных функций. Специальные атрибуты для контроля валидации приведены в таблице 1.

ASP.NET MVC включает в себя набор классов резервной валидации для всех атрибутов, которые отвечают за выполнение текущей проверки данных. Изучим атрибуты валидации на примере интерфейса, которому необходима валидация. На рисунке 1 показана форма ввода Edit с полями Company Name и Email Address.

Таблица 1: Атрибуты Data Annotations для валидации

Атрибут	Описание
CompareAttribute	Сравнивает значения двух свойств модели. Если они равны, валидация успешно завершена
RemoteAttribute	Указывает JQuery Validate, библиотеке валидации на стороне клиента, что она должна вызвать действие для валидации на сервере, и выводит ее результат до отправки формы
RequiredAttribute	Указывает, что требуется значение поля данных
RangeAttribute	Устанавливает ограничения числового диапазона для значения поля данных
RegularExpressionAttribute	Указывает, что значение поля данных должно соответствовать заданному регулярному выражению
StringLengthAttribute	Задает максимальное число символов, которые разрешены в поле данных

Рисунок 1: Экран Edit с обязательным полем



В нашем приложении Company Name – обязательное для заполнения поле, Email Address - необязательное. Чтобы сделать поле Company Name обязательным, мы используем RequiredAttribute.

```
publicclass CompanyInput
{
```

```
  [Required]
publicstring CompanyName { get; set; }
  [DataType(DataType.EmailAddress)]
publicstring EmailAddress { get; set; }
}
```

Мы добавили RequiredAttribute к свойству CompanyName. Мы также добавили EmailAddress к атрибуту DataTypeAttribute, чтобы воспользоваться пользовательскими шаблонами для адресов электронной почты.

В нашем представлении нам нужно отображать потенциальные сообщения об ошибках валидации, и мы можем реализовать это несколькими способами. Мы можем использовать шаблоны, в которые уже включены сообщения валидации.

```
<h2>Edit</h2>
@using (Html.BeginForm("Edit", "Home")) {
  @Html.EditorForModel()
<button type="submit">Submit</button>
}
```

Шаблоны редактирования модели, используемые по умолчанию, создают пользовательский интерфейс, который включает в себя как элементы ввода, так и сообщения проверки.

Для более детального управления выводом, мы можем использовать методы расширения валидации HtmlHelper. Расширение ValidationSummaryпредоставляет сводный список ошибок валидации, который обычно отображается в верхней части формы. Чтобы выводить ошибки валидации для конкретных свойств модели, мы можем использовать метод ValidationMessage, а также ValidationMessageFor, основанный на выражениях.

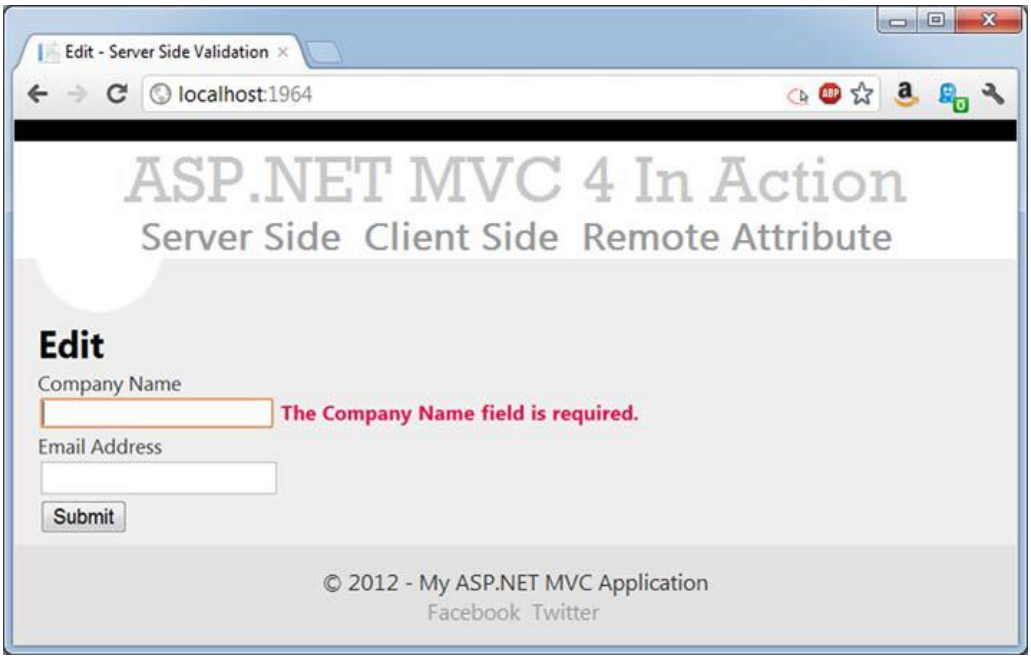
После вывода сообщения валидации мы должны убедиться, что наша модель действительна в результирующем методе контроллера метода POST. Мы можем добавить к модели какие угодно атрибуты валидации, но управлять ошибками валидации все равно придется в методе контроллера.

```
[HttpPost]
public ActionResult Edit(CompanyInput input)
{
if (ModelState.IsValid)
  {
return View("Success");
  }
return View(new CompanyInput());
}
```

Вдействии Edit метода POST мысначалапроверяемналичиеошибокв ModelState. Движок валидации MVC помещает ошибки валидации в ModelState, и их отсутствие отражается в свойстве IsValid. Если ошибок нет, мы выводим экран с сообщением об успешном заполнении формы. В противном случае мы отображаем исходный экран Edit, теперь с сообщением об ошибке валидации.

Чтобы продемонстрировать ошибку валидации в этом примере, просто отправим форму, не заполняя поле для названия компании. На этой странице поле Company name является обязательным. Результат показан на рисунке 2.

Рисунок 2: Ошибка валидации в результате отсутствия названия компании



Когда мы заполним форму, пропустив поле для названия компании, сообщение об ошибке валидации отображается корректно.

Тем не менее, на рисунке 2 показана проблема, связанная с сообщением об ошибке валидации и самим экраном. И сообщение об ошибке, и название поля отображаются как "CompanyName" без пробела. Но мы хотим всегда включать пробелы между словами в названиях полей. Исправить название поля можно с помощью DisplayNameAttribute (часть пространства имен System.ComponentModel). Так как принято отображать имена свойств с пробелами между словами, мы расширим встроенный класс ModelMetadataProvider с помощью метода, который будет автоматически добавлять пробелы.

Расширение ModelMetadataProvider

Как мы видели в предыдущей части, многие новые возможности в ASP.NET MVC используют метаданные модели. Шаблоны используют метаданные для отображения элементов ввода и текста, а провайдеры валидации используют метаданные для выполнения валидации.

Если мы хотим, чтобы метаданные для нашей модели извлекались из других источников, помимо Data Annotations, мы должны использовать ModelMetadataProvider.

Листинг1: Абстрактныйкласс ModelMetadataProvider

```
publicabstractclass ModelMetadataProvider
{
publicabstract IEnumerable<ModelMetadata> GetMetadataForProperties
   (object container, Type containerType);

publicabstract ModelMetadata GetMetadataForProperty
   (Func<object> modelAccessor, Type containerType, string propertyName);

publicabstract ModelMetadata GetMetadataForType
   (Func<object> modelAccessor, Type modelType);
}
```

Класс ModelMetadataProvider содержит методы, которые получают ModelMetadata для каждого члена типа, для конкретно свойства и для частного типа, что показано в листинге 1.

Чтобы изменить отображаемый текст для конкретного свойства, нам нужно переопределить поведение базового класса DataAnnotationsModelMetadataProvider. Класс AssociatedMetadataProvider предоставляет общие функции для осуществления сценариев,

в которых метаданные извлекаются из традиционных классов, свойств и атрибутов. Производным классам, таким как DataAnnotationsModelMetadataProvider, нужно всего лишь создать ModelMetadata уже уже существующих атрибутов.

В данном примере мы хотим изменить поведение DisplayName в модели метаданных. По умолчанию свойство DisplayName в ModelMetadata приходит от DisplayNameAttribute. Мы все еще хотим поддерживать значение DisplayName через атрибут.

В листинге 2 мы расширяем встроенный класс DataAnnotationsModelMetadataProvider создавая DisplayName из имени свойства, разделяя его пробелами.

Листинг 2: Пользовательский провайдер метаданных

```
publicclass ConventionProvider : DataAnnotationsModelMetadataProvider
{
protectedoverride ModelMetadata CreateMetadata(
    IEnumerable<Attribute> attributes,
    Type containerType,
    Func<object> modelAccessor,
    Type modelType,
string propertyName)
  {
var meta = base.CreateMetadata(attributes, containerType, modelAccessor, modelType, propertyName);
if (meta.DisplayName == null)
    meta.DisplayName = meta.PropertyName.ToSeparatedWords();
return meta;
  }
}
```

Строка **3**: Переопределяет CreateMetadata

Строка **10**: Вызывает базовый метод

Строка **12**: Разделяет слова в имени свойства пробелами

Чтобы создать соответствующую схему соглашения для отображения имен, мы создаем класс, который наследуется от класса DataAnnotationsModelMetadataProvider. Этот класс имеет довольно много встроенных возможностей, так что нам остается только переопределить метод CreateMetadata (строка 3). Базовый класс содержит поведение, которые мы хотим сохранить, поэтому мы сначала вызываем метод базового класса (строка 10) и сохраняем его результаты в локальной переменной. Так как мы могли поместить значение атрибута в DisplayName, теперь мы хотим изменить сценарий только в том случае, если значение DisplayName еще не было установлен. Итак, если оно не было установлено, мы хотим разделить имя свойства на отдельные слова с помощью расширенного метода ToSeparatedWords (строка 12). Наконец, мы возвращаем объект ModelMetadata, содержащий измененное имя.

Метод расширения ToSeparatedWords - довольно простое регулярное выражение для разделения идентификаторов на отдельные слова.

```
publicstaticclass StringExtensions
{
publicstaticstring ToSeparatedWords(thisstringvalue)
  {
if (value != null)
return Regex.Replace(value, "([A-Z][a-z]?)", " $1").Trim();
returnvalue;
  }
}
```

Когда мы создали пользовательский ModelMetadataProvider, мы должны настроить ASP.NET MVC, чтобы его использовать. Такаянастройкапроводитсявфайле Global.asax:

```
protectedvoidApplication_Start()
```

```
{
RegisterRoutes(RouteTable.Routes);
  ModelMetadataProviders.Current = new ConventionProvider();
}
```

Чтобы переопределить провайдер метаданных, мы записываем новый провайдер в свойстве ModelMetadataProviders.Current. Когда настройка проведена, сообщения валидации и названия полей отображаются корректно, как показано на рисунке3.

Рисунок 3: Экран Edit с корректно отображающимися названиями полей и сообщением об ошибке.

Используя соответствующую соглашениюмодификацию DataAnnotationsModelMetadataProvider, мы можем использовать имена свойств в названиях полей и сообщениях об ошибках. В противном случае мы должны были бы избегать шаблонов редактирования и отображения, или записывать отображаемые имена в атрибутах.

Практическое занятие № 15

Тема 2.4: Совместимость и использование инструментов ревьюирования в различных системах контроля версий

Цель: Изучить совместимость и использование инструментов ревьюирования в различных системах контроля версий

Основы VCS

Введение

Перед тем, как говорить про какую либо конкретную систему контроля версий необходимо понимать, что это такое, какими они бывают и зачем вообще они появились. Эта лекция предназначена для первоначального знакомства с системами контроля и управления версиями, и сначала я расскажу о происхождении инструментов для контроля версий, расскажу, какие системы управления версиями сейчас популярны и в чем у них основные различия.

О контроле версий

Что такое контроль версий, и зачем он вам нужен?

Наверное стоит начать с определения системы контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов.

В последнее время файлы являются конечным результатом для многих профессий (для примера, писательскую деятельность, научные работы и, конечно, разработку программного обеспечения). Тратится много времени и сил на разработку и поддержку этих файлов и никто не хочет, чтобы пришлось тратить еще больше времени и сил на восстановление данных потерянных в результате каких-либо изменений.

Представим, что программист разрабатывает проект состоящий из одного небольшого файла (кстати, пример вполне реальный, не синтетический, встречался в реальной жизни). После выпуска первой версии проекта перед ним встает непростой

выбор: необходимо исправлять проблемы о которых сообщают пользователи первой версии и, в тоже время, разрабатывать что-то новое для второй. Даже если надо просто исправлять возникающие проблемы, то велика вероятность, что после какого-либо изменения проект перестает работать, и надо определить, что было изменено, чтобы было проще локализовать проблему. Также желательно вести какой-то журнал внесенных изменений и исправлений, чтобы не делать несколько раз одну и ту же работу.

В простейшем случае вышеприведенную проблему можно решить хранением нескольких копий файлов, например, один для исправления ошибок в первой версии проекта и второй для новых изменений. Так как изменения обычно не очень большие по сравнению с размером файла, то можно хранить только измененные строки используя утилиту diff и позже объединять их с помощью утилиты patch. Но что если проект состоит из нескольких тысяч файлов и над ним работает сотня человек? Если в этом случае использовать метод с хранением отдельных копий файлов (или даже только изменений) то проект застопорится очень быстро. В последующих лекциях, для примеров я буду использовать исходные коды программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.

Если вы графический или веб-дизайнер и хотели бы хранить каждую версию изображения или макета — а этого вам наверняка хочется — то пользоваться системой контроля версий будет очень мудрым решением. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить. Вдобавок, накладные расходы за всё, что вы получаете, будут очень маленькими.

Локальные системы контроля версий

Как уже говорилось ранее - один из примеров локальной СУВ предельно прост: многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы. Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов

Одной из наиболее популярных СКВ такого типа является RCS (Revision Control System, Система контроля ревизий), которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. RCS была разработана в начале 1980-х годов Вальтером Тичи (Walter F. Tichy). Система позволяет хранить версии только одного файла, таким образом управлять несколькими файлами приходится вручную. Для каждого файла находящегося под контролем системы информация о версиях хранится в специальном файле с именем оригинального файла к которому в конце добавлены символы ',v'. Например для файла file.txt версии будут храниться в файле file.txt,v. Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами). Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи. Для хранения версий система использует утилиту diff. Хотя RCS соответствует минимальным требованиям к системе контроля версий она имеет следующие основные недостатки, которые также послужили стимулом для создания следующей рассматриваемой системы:

- Работа только с одним файлом, каждый файл должен контролироваться отдельно;

- Неудобный механизм одновременной работы нескольких пользователей с системой, хранилище просто блокируется пока заблокировавший его пользователь не разблокирует его;
- От бекапов вас никто не освобождает, вы рискуете потерять всё.

Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий.

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте. Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей.

CVS

CVS (Concurrent Versions System, Система совместных версий) пока остается самой широко используемой системой, но быстро теряет свою популярность из-за недостатков которые я рассмотрю ниже. Дик Грун (Dick Grune) разработал CVS в середине 1980-х. Для хранения индивидуальных файлов CVS (также как и RCS) использует файлы в RCS формате, но позволяет управлять группами файлов расположенных в директориях. Также CVS использует клиент-сервер архитектуру в которой вся информация о версиях хранится на сервере. Использование клиент-сервер архитектуры позволяет использовать CVS даже географически распределенным командами пользователей где каждый пользователь имеет свой рабочий директорий с копией проекта. Как следует из названия пользователи могут использовать систему совместно.

Возможные конфликты при изменении одного и того же файла разрешаются тем, что система позволяет вносить изменения только в самую последнюю версию файла. Таким образом всегда рекомендуется перед заливкой своих изменений обновлять свою рабочую копию файлов на случай возможных конфликтующих изменений. При обновлении система вносит изменения в рабочую копию автоматически и только в случае конфликтующих изменений в одном из мест файла требуется ручное исправление места конфликта.

CVS также позволяет вести несколько линий разработки проекта с помощью ветвей (branches) разработки. Таким образом, как уже упоминалось выше, можно исправлять ошибки в первой версии проекта и параллельно разрабатывать новую функциональность.

CVS использовалась большим количеством проектов, но конечно не была лишена недостатков которые позднее привели к появлению следующей рассматриваемой системы. Рассмотрим основные недостатки:

- Так как версии хранятся в файлах RCS нет возможности сохранять версии директорий. Стандартный способ обойти это препятствие - это сохранить какой-либо файл (например, README.txt) в директории;

- Перемещение, или переименование файлов не подвержено контролю версий. Стандартный способ сделать это: сначала скопировать файл, удалить старый с помощью команды cvs remove и затем добавить с его новым именем с помощью команды cvs add;

Subversion

Subversion (SVN) был разработан в 2000 году по инициативе фирмы CollabNet. SVN изначально разрабатывался как "лучший CVS" и основной задачей разработчиков было исправление ошибок допущенных в дизайне CVS при сохранении похожего интерфейса. SVN также как и CVS использует клиент-сервер архитектуру. Из наиболее значительных изменений по сравнению с CVS можно отметить:

- Атомарное внесение изменений (commit). В случае если обработка коммита была прервана не будет внесено никаких изменений.
- Переименование, копирование и перемещение файлов сохраняет всю историю изменений.
- Директории, символические ссылки и мета-данные подвержены контролю версий.
- Эффективное хранение изменений для бинарных файлов.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных.

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Зачем нужны распределенные системы?

Как следует из названия одна из основных идей распределенных систем — это отсутствие четко выделенного центрального хранилища версий - репозитория. В случае распределенных систем набор версий может быть полностью, или частично распределен между различными хранилищами, в том числе и удаленными. Такая модель отлично вписывается в работу распределенных команд, например, распределенной по всему миру команды разработчиков работающих над одним проектом с открытым исходным кодом. Разработчик такой команды может скачать себе всю информацию по версиям и после этого работать только на локальной машине. Как только будет достигнут результат одного из этапов работы, изменения могут быть залиты в один из центральных репозиториев или, опубликованы для просмотра на сайте разработчика, или в почтовой рассылке. Другие участники проекта, в свою очередь, смогут обновить свою копию хранилища версий новыми изменениями, или попробовать опубликованные изменения на отдельной, тестовой ветке разработки. К сожалению, без хорошей организации проекта отсутствие одного центрального хранилища может быть минусом распределенных систем. Если в случае централизованных систем всегда есть один общий репозиторий откуда можно получить последнюю версию проекта, то в случае распределенных систем нужно организационно решить какая из веток проекта будет основной. Почему распределенная система контроля версий может быть интересна кому-то, кто уже использует централизованную систему - такую как Subversion? Любая работа подразумевает принятие решений, и в большинстве случаев необходимо пробовать различные варианты: при работе с системами контроля версий для рассмотрения различных вариантов и работы над большими изменениями служат ветки разработки. И хотя это достаточно естественная

концепция, пользоваться ей в Subversion достаточно не просто. Тем более, все усложняется в случае множественных последовательных объединений с одной ветки на другую — в этом случае нужно безошибочно указывать начальные и конечные версии каждого изменения, что бы избежать конфликтов и ошибок. Для распределенных систем контроля версий ветки разработки являются одной из основополагающих концепций — в большинстве случаев каждая копия хранилища версий является веткой разработки. Таким образом, механизм объединения изменений с одной ветки на другую в случае распределенных систем является одним из основных, что позволяет пользователям прикладывать меньше усилий при пользовании системой.

Краткое описание популярных распределенных СУВ

- Git - распределенная система контроля версий, разработанная Линусом Торвальдсом. Изначально Git предназначалась для использования в процессе разработки ядра Linux, но позже стала использоваться и во многих других проектах — таких, как, например, X.org и Ruby on Rails, Drupal. На данный момент Git является самой быстрой распределенной системой, использующей самое компактное хранилище ревизий. Но в тоже время для пользователей, переходящих, например, с Subversion интерфейс Git может показаться сложным;
- Mercurial - распределенная система, написанная на языке Python с несколькими расширениями на C. Из использующих Mercurial проектов можно назвать, такие, как, Mozilla и MoinMoin.
- Bazaar - система разработка которой поддерживается компанией Canonical — известной своими дистрибутивом Ubuntu и сайтом https://launchpad.net/. Система в основном написана на языке Python и используется такими проектами, как, например, MySQL.
- Codeville - написанная на Python распределенная система использующая инновационный алгоритм объединения изменений (merge). Система используется, например, при разработке оригинального клиента BitTorrent.
- Darcs - распределенная система контроля версий написанная на Haskell используемая, например, проектом Buildbot.
- Monotone - система написанная на C++ и использующая SQLite как хранилище ревизий.

Практическое занятие № 16

Тема 2.5: Особенности ревьюирования в Linux. Настройки доступа

Цель:Изучить особенности ревьюирования в Linux. Настройки доступа

Теоретические сведения

Любой ресурс компьютера под управлением ОС Linux представляется как файл, поэтому мы будем говорить только о правах доступа к файлу. Как многопользовательская операционная система, ОС Linux содержит механизм разграничения доступа к данным, позволяющий как защитить данные одного пользователя от нежелательного вмешательства других, так и разрешить другим доступ к этим данным для совместной работы.

Говоря о правах доступа пользователя к файлам, стоит заметить, что в действительности манипулирует файлами не сам пользователь, а запущенный им процесс (например, утилита cat). Поскольку и файл, и процесс создаются и управляются системой, ей нетрудно организовать какую угодно политику доступа одних к другим, основываясь на любых свойствах процессов как субъектов и файлов как объектов системы.

В Linux, однако, используются не какие угодно свойства, а результат идентификации пользователя — его **UID**. Каждый процесс системы обязательно принадлежит какому-нибудь пользователю, и **идентификатор пользователя (UID)** — обязательное свойство любого процесса Linux. Когда программа login запускает

стартовый командный интерпретатор, она приписывает ему UID, полученный в результате диалога. Все процессы, запущенные пользователем во время терминальной сессии, будут иметь его идентификатор.

Пользователь может быть членом нескольких групп, равно как и несколько пользователей может быть членами одной и той же группы. Исторически сложилось так, что одна из групп — группа по умолчанию — является для пользователя основной: когда (не вполне точно) говорят о **«GID пользователя»**, имеют в виду именно идентификатор группы по умолчанию. Пользователь не может не быть членом как минимум одной группы, как снаряд не может не попасть в эпицентр взрыва! Часто процедуру создания пользователя проектируют так, что имя группы по умолчанию совпадает с входным именем пользователя, а GID пользователя — с его UID. Однако это совсем не обязательно: не всегда нужно заводить для пользователя отдельную группу, а если заводить, то не всегда удаётся сделать так,чтобы желаемый идентификатор группы совпадал с желаемым идентификатором пользователя.

При создании объектов файловой системы — файлов, каталогов и т. п. — каждому в обязательном порядке приписывается UID — идентификатор пользователя-владельца файла, GID — идентификатор группы, которой принадлежит файл, тип объекта и набор т. н. атрибутов, а также некоторую дополнительную информацию. Атрибуты определяют, кто и что с файлом имеет право делать, и описаны ниже.

В Linux существует выделенный пользователь системы, на которого не распространяются ограничения прав доступа - **суперпользователь (root)**. UID суперпользовательских процессов равен 0: так система отличает их от процессов других пользователей. Именно суперпользователь имеет возможность произвольно изменять владельца и группу файла. Ему открыт доступ на чтение и запись к любому файлу системы и доступ на чтение, запись и использование к любому каталогу. Наконец, суперпользовательский процесс может на время сменить свой собственный UID с нулевого на любой другой. Среди учётных записей Linux всегда есть запись по имени root («корень»), соответствующая нулевому идентификатору, поэтому вместо «суперпользователь» часто говорят «root». Множество системных файлов принадлежат root-у, множество файлов только ему доступны на чтение или запись. Пароль этой учётной записи — одна из самых больших драгоценностей системы. Свойство root иметь доступ ко всем ресурсам системы накладывает очень высокие требования на человека, знающего пароль root. Суперпользователь может всё — в том числе и всё поломать, поэтому любую работу стоит вести с правами обычного пользователя, а к правам root прибегать только по необходимости. Существует два различных способа получить **права суперпользователя**.

Первый — это зарегистрироваться в системе под этим именем, ввести пароль и получить стартовую оболочку, имеющую нулевой UID. Это — самый неправильный способ, пользоваться которым стоит, только если нельзя применить другие. Что в этом случае выдаст команда last ? Что тогда-то с такой-то консоли в систему вошёл неизвестно кто с правами суперпользователя и что-то там такое делал. С точки зрения системного администратора, это — очень подозрительное событие, особенно, если сам он в это время к указанной консоли не подходил... сами администраторы такой способ не любят.

Второй способ — это воспользоваться специальной утилитой su (shell of user), которая позволяет выполнить одну или несколько команд от лица другого пользователя. Отличие от предыдущего способа — в том, что всегда известно, кто именно запускал su, а значит, с кого спрашивать за последствия.

В некоторых случаях удобнее использовать не su, а утилиту sudo, которая позволяет выполнять только заранее заданные команды.

Утилита id(см. man id) выводит входное имя пользователя и соответствующий ему UID, а также группу по умолчанию и полный список групп, членом которых он является.

Вся информация о пользователях и группах, зарегестрированных в системеме, хранится в файлах /etc/passwd, /etc/shadow и /etc/groups

Таким образом, конкретный пользователь по отношению к его собственным файлам выступает как их владелец (u - user).

У каждого файла есть еще и группа (g - group), к которой принадлежит владелец файла.

Кроме владельца и группы в системе могут существовать и другие пользователи. Поэтому в атрибутах файла содержаться значения, запрещающие или разрешающие доступ всем другим (o - other), кто не вошел в группу и не является владельцем.

Возможные действия над файлом

Что можно делать с файлом после его создания? В первую очередь просматривать, или читать (r - read). Во вторую очередь, файл можно изменить (дописать, исправить, переименовать, переместить). Таким образом, мы можем говорить о возможности записи (w - write) в файл. Если файл является программой, то его содержимое представляет собой команды для процессора, выполнение которых приводит к тому или иному желаемому (мы надеемся) эффекту. Другими словами, некоторые файлы можно исполнять (x - execution).

Право чтения, записи и выполнения являются основными правами доступа к файлам, однако, существуют еще дополнительные атрибуты, с помощью которых можно управлять правами доступа к файлам (о них будет сказано ниже).

Представление прав доступа

Основные атрибуты прав доступа можно представить в виде двеннадцати битов двоичного числа, равных 1, если атрибут установлен, и 0, если нет. Порядок битов в числе следующий:

sU|sG|t|rU|wU|xU|rG|wG||xG|rO|wO|xO.

где sU — это SetUID, sG — это SetGID, t — это t-атрибут (sticky–бит), после чего следуют три тройки атрибутов доступа:

rU|wU|xU - права чтения (Read), записи (Write) и выполнения (eXecute) для владельца файла (User);

rG|wG|xG - права чтения (Read), записи (Write) и выполнения (eXecute) для группы файла (Group);

rO|wO|xO - права чтения (Read), записи (Write) и выполнения (eXecute) для всех остальных (Other).

Процессы с установленным битом sU выполняются с правами владельца. А с установленным битом sG – с правами группы.

В каталоге с установленным sticky–битом удалять файлы может только владелец или root. При том устанавливать этот бит может только root, а сбрасывать может владелец и root.

Например, команда ls -l выводит права доступа в таком формате:

«-rw-rw-rw-» (Первая черточка - обычный файл, и 9 прав доступа - все могут читать и изменять)

«drwx------» (Каталог, полный доступ (чтение, изменение, выполнение) имеет только владелец файла)

«-rw-r-----» (Обычный файл, владелец может читать и изменять, группа - читать, остальные - не имеют прав)

«drwxr-xr--» (Каталог, владелец имеет полный доступ, группа - чтение и выполнение, остальные - только чтение)

«drwxrwxrwt» (Каталог, все имеют полный доступ, однако, установлен sticky–бит, поэтому права записи в каталог для членов группы и для посторонних ограничены их собственными файлами, и только владелец имеет право изменять список файлов в каталоге, как ему вздумается. Такие каталоги называются разделяемыми, потому что предназначены они, как правило, для совместной работы всех пользователей в системе,

обмена информацией и т. п. При установке атрибута «t» доступ на выполнение для посторонних («t» в строчке атрибутов стоит на месте последнего «x») не отменяется. Просто они так редко используются друг без друга, что ls выводит их в одном и том же месте. Если кому-нибудь придёт в голову организовать разделяемый каталог без доступа посторонним на использование, ls выведет на месте девятого атрибута не «t», а «T».)

«-rws--x--x» (Обычный файл, установлен атрибут SetUID. Как и в случае с t-атрибутом, ls выводит букву «s» вместо буквы «x» в тройке «для владельца». Точно так же, если соответствующего x-атрибута нет (что бывает редко), ls выведет «S» вместо «s».)

«-rwx--s--x» (Обычный файл, установлен атрибут SetGID. Утилита ls выводит SetGID в виде «s» вместо «x» во второй тройке атрибутов («для группы»). Замечания касательно «s», «S» и «x» действительны для SetGID так же, как и для SetUID.)

права доступа представляются также в *двоичном* и *восьмиричном* виде.

Например:

Примеры записи прав доступа в двоичной форме	110 110 110	все могуть читать и изменять
	111 100 000	владелец имеет все права, группа - чтение, остальные не имееют никаких прав
	001 111 100 000	установлен t-атрибут, владелец имеет все права, группа - чтение, остальные не имееют никаких прав
	010 111 100 000	установлен атрибут SetGID, владелец имеет все права, группа - чтение, остальные не имееют никаких пра
	100 111 100 000	установлен атрибут SetUID, владелец имеет все права, группа - чтение, остальные не имееют никаких прав
Примеры записи прав доступа в восьмиричной форме	666	все могуть читать и изменять
	740	владелец имеет все права, группа - чтение, остальные не имееют никаких прав
	1740	установлен t-атрибут, владелец имеет все права, группа - чтение, остальные не имееют никаких прав
	2740	установлен атрибут SetGID, владелец имеет все права, группа - чтение, остальные не имееют никаких пра
	4740	установлен атрибут SetUID, владелец имеет все права, группа - чтение, остальные не имееют никаких прав

Особенности доступа к каталогам

Каталог — это особый тип файла. Его содержание — это список других файлов. Каталоги имеют те же «биты прав», что и остальные файлы. Однако то, что эти права означают может быть не таким простым для понимания.

Если каталог можно читать (r), то это означает, что разрешено только узнать список файлов, содержащихся в этом каталоге. Только список файлов, но не их свойства (размер, права доступа и др.).

Если каталог можно исполнять (x), то это означает, что в него можно заходить и просматривать содержимое файлов (доступ к которым разрешен для данной категории), узнавать свойства (атрибуты) файлов. Можно изменить содержимое файла (если его разрешено менять), но не имя файла.

Если каталог можно изменять (w), то это означает, что в нем можно изменять файлы, их имена, удалять их. **Опасность!** Это можно делать даже с файлами, доступ к которым запрещен для данной категории. **Лечение!** Вводят дополнительный t-бит. При его наличии пользователь может изменять только свои файлы.

Следует понимать, что:

- доступ к конкретному файлу также зависит от наличия доступа на исполнение к каталогам на протяжении всего пути;
- изменять существующие файлы можно, не имея доступа на запись в каталог, достаточно иметь доступ на запись самого файла.

Изменение прав доступа

Изменение прав доступа к указанному файлу (или каталогу) выполняется с помощью команды chmod.

При создании каталога также можно сразу указать права доступа к нему с помощью команды mkdir -m.

Изменение владельца и группы файлов выполняется с помощью команды chown. Группу также пожно назначить командой chgrp.

Тем же побитовым представлением атрибутов регулируются и права доступа по умолчанию при создании файлов и каталогов. Делается это с помощью команды umask. Единственный параметр umask — восьмеричное число, задающее атрибуты, которые не надо устанавливать новому файлу или каталогу. Так, umask 0 приведёт к тому, что файлы будут создаваться с атрибутами «rw-rw-rw-», а каталоги — «rwxrwxrwx». Команда umask 022 убирает из атрибутов по умолчанию права доступа на запись для всех, кроме хозяина (получается «rw-r--r--» и «rwxr-xr-x» соответственно), а с umask 077 новые файлы и каталоги становятся для них полностью недоступны («rw-------» и «rwx------»).

Специальные атрибуты файлов

В файловой системе Linux присутствует поддержка дополнительных флагов или по другому "атрибутов" для файлов в ядрах начиная с серии 1.1. Файловая система в ядрах серий 2.2 и 2.4. позволяет работать со следующим набором атрибутов:

A	Atime система не апдейтит atime(access time) для данного файла.
S	Sync система фиксирует все изменения, происходящие в данном файле на физическом диске синхронно с приложением изменяющим данный файл.
a	Append система позволяет открывать данный файл с целью его дополнения и не позволяет никаким процессам перезаписывать или усекать его.Если данный атрибут применяется к директории процесс может создавать или модифицировать файлы в этой директории , но не удалять их.
i	Immutable система запрещает любые изменения данного файла. В случае директории, процессы могут модифицировать

	файлы уже содержащиеся в данной директории, но не могут удалять файлы или создавать новые.
d	No Dump программе создающей дампы дается указание игнорировать данный файл во время создания backup.
c	Compress система применяет прозрачную компрессию для данного файла. Т.е. чтение из него дает уже декомпрессованые данные и предварительно перед записью на диск производится их сжатие.
s	Secure Deletion удаление такого файла сопровождается перезаписью блоков диска,на которых он располагался нулями.
u	Undelete когда приложение запрашивает файл на удаление, система должна сохранить блоки на диске, на которых расположен данный файл, чтобы потом его можно было восстановить.

Несмотря на то, что файловая система поддерживает данный набор атрибутов, у ядра и различных приложений остается выбор, учитывать или не учитывать их. В зависимости от своих версий, приложения могут по разному работать с этими атрибутами.

В таблице ниже приведено соответствие, как различные версии ядра учитывают каждый атрибут:

* позволяет устанавливать флаг и учитывает его значение

i позволяет устанавливать флаг,но игнорирует его значение

- полностью игнорирует флаг

1.0	1.2	2.0	2.2	2.4
-	-	*	*	*
*	*	*	*	*
-	*	*	*	*
-	*	*	*	*
-	*	*	*	*
i	i	i	i	i
*	*	i	i	i
i	i	i	i	i

В некоторых версиях разработчики исключили атрибут 'secure deletion', поскольку, на их взгляд, его использование повышает общую безопасность лишь незначительно.

Флаг 'A' или 'atime' для определенных файлов может дать некоторую прибавку производительности, так как избавляет систему от необходимости апдейтить access time для этих файлов каждый раз, когда их открывают на чтение.

Атрибут 'S' или 'synchronous' предоставляет новый уровень поддержки целостности данных. Но поскольку все изменения в файлах немедленно сохраняются на диске несколько понижается производиельность.

Основное внимание мы уделим флагу 'a' ('append only') и 'i' ('immutable'), так как их использование дает наиболее ощутимые приемущества в плане обеспечения безопасности и целостности файловой системы.

Различные open source BSD системы, такие как FreeBSD и OpenBSD, с некоторых пор поддерживают аналогичные флаги в своих файловых системах (UFS и FFS соответственно) и частично решение о реализации аналогичных спецвозможностей в Linux основывается на этом положительном опыте.

Изменение специальных атрибутов файла выполняется с помощью команды chattr.

Просмотр специальных атрибутов файла выполняется с помощью команды lsattr.

Подробную информацию о файле или каталоге можно получить с помощью команды stat.

Изменить временные метки файлов или директорий можно с помощью команды touch.

Команды для работы с пользователями и группами пользователей

# id	# Показывает сводную информацию по текущему пользователю (логин, UID, GID, группы);
# finger User	# Показать информацию о пользователе User;
# last	# Показывает последних зарегистрированных пользователей;
# who	# Показывает имя текущего пользователя и время входа;
# useradd Denis	# Добавление нового пользователя Denis;
# groupadd ITShaman	# Добавление группы ITShaman;
# usermod -a -G ITShaman Denis	# Добавляет пользователя Denis в группу ITShaman (для Debian-подобных дистрибутивов);
# groupmod -A Denis ITShaman	# Добавляет пользователя Denis в группу ITShaman (SuSE);
# userdel Denis	# Удаление пользователя Denis;
# groupdel ITShaman	# Удаление группы ITShaman.

Практическое занятие № 17

Тема 2.6: Типовые инструменты и методы анализа программных проектов

Цель: Изучить типовые инструменты и методы анализа программных проектов

Структурный анализ — один из формализованных методов анализа требований к ПО. Автор этого метода — Том Де Марко (1979). В этом методе программное изделие рассматривается как преобразователь информационного потока данных. Основной элемент структурного анализа — диаграмма потоков данных.

Диаграммы потоков данных

Диаграмма потоков данных ПДД — графическое средство для изображения информационного потока и преобразований, которым подвергаются данные при движении от

входа к выходу системы. Элементы диаграммы имеют вид, показанный на рисунке. Диаграмма может использоваться для представления программного изделия на любом уровне абстракции.

Диаграмма высшего (нулевого) уровня представляет систему как единый овал со стрелкой, ее называют основной или контекстной моделью. Контекстная модель используется для указания внешних связей программного изделия.

Для детализации (уточнения системы) вводится диаграмма 1-го уровня. Каждый из преобразователей этой диаграммы — подфункция общей системы. Таким образом, речь идет о замене преобразователя F на целую систему преобразователей.

Дальнейшее уточнение (например, преобразователя F3) приводит к диаграмме 2-го уровня.

Говорят, что ПДД 1 разбивается на диаграммы 2-го уровня.

Диаграмма потоков данных – это абстракция, граф. Для связи графа с проблемной областью (превращения в граф-модель) надо задать интерпретацию ее компонентов – дуг и вершин.

Описание потоков данных и процессов

Базовые средства диаграммы не обеспечивают полного описания требований к программному изделию. Очевидно, что должны быть описаны стрелки (потоки данных) и преобразователи (процессы). Для этих целей используются словарь требований (данных) и спецификации процессов.

Словарь требований (данных) содержит описания потоков данных и хранилищ данных. Словарь требований является неотъемлемым элементом любой CASE-утилиты автоматизации анализа. Структура словаря зависит от особенностей конкретной CASE-утилиты. Тем не менее можно выделить базисную информацию типового словаря требований.

Большинство словарей содержит следующую информацию.

1. *Имя* (основное имя элемента данных, хранилища или внешнего объекта).

2. *Прозвище* (Alias) — другие имена того же объекта.

3. *Где и как используется объект* — список процессов, которые используют данный элемент, с указанием способа использования (ввод в процесс, вывод из процесса, как внешний объект или как память).

4. *Описание содержания* — запись для представления содержания.

5. *Дополнительная информация* — дополнительные сведения о типах данных, допустимых значениях, ограничениях и т. д.

Спецификация процесса — это описание преобразователя. Спецификация поясняет: ввод данных в преобразователь, алгоритм обработки, характеристики производительности преобразователя, формируемые результаты. Количество спецификаций равно количеству преобразователей диаграммы.

Методы анализа, ориентированные на структуры данных

Элементами проблемной области для любой системы являются потоки процессы и структуры данных. При структурном анализе активно работают только с потоками данных и процессами.

Методы, ориентированные на структуры данных, обеспечивают:

1. определение ключевых информационных объектов и операций;

2. определение иерархической структуры данных;

3. компоновку структур данных из типовых конструкций – последовательности, выбора, повторения;

4. последовательность шагов для превращения иерархической структуры данных в структуру программы.

Наиболее известны два метода: метод Варнье-Орра и метод Джексона.

В методе Варнье-Орра для представления структур применяют диаграммы Варнье.

Для построения диаграмм Варнье используют 3 базовых элемента: последовательность, выбор, повторение.

Как показано на рисунке, с помощью этих элементов можно строить информационные структуры с любым количеством уровней иерархии. Как видим, для представления структуры газеты здесь используются три уровня иерархии.

Метод анализа Джексона

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

Методика Джексона

Метод Джексона (1975) включает 6 шагов. Три шага выполняются на этапе анализа, а остальные - на этапе проектирования.

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

Шаг объект-действие начинается с определения проблемы на естественном языке.

Шаг объект-структура – структура объектов описывает последовательность действий над объектами (в условном времени). Для представления структуры объектов Джексон предложил 3 типа структурных диаграмм.

Шаг начального моделирования.

Начальное моделирование - это шаг к созданию описания системы как модели реального мира. Описание создается с помощью диаграммы системной спецификации. Элементами диаграммы системной спецификации являются физические процессы (имеют суффикс 0) и их модели (имеют суффикс 1). Как показано на рисунке, предусматриваются 2 вида соединений между физическими процессами и моделями.

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

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

Метод структурного проектирования

Исходными данными для метода структурного проектирования являются компоненты модели анализа ПС, которая представляется иерархией диаграмм потоков данных. Результат структурного проектирования — иерархическая структура ПС. Действия структурного проектирования зависят от типа информационного потока в модели анализа.

Типы информационных потоков

Различают 2 типа информационных потоков:

1) поток преобразований;

2) поток запросов.

Как показано на рисунке, в потоке преобразований выделяют 3 элемента: Входящий поток, Преобразуемый поток и Выходящий поток.

Потоки запросов имеют в своем составе особые элементы — запросы.

Назначение элемента-запроса состоит в том, чтобы запустить поток данных по одному из нескольких путей. Анализ запроса и переключение потока данных на один из путей действий происходит в центре запросов.

Структуру потока запроса иллюстрирует рисунок.

Метод проектирования Джексона

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

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

Доопределение функций

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

1. *Встроенные функции* (задаются командами, вставляемыми в структурный текст процесса-модели).

2. *Функции впечатления* (наблюдают вектор состояния процесса-модели и вырабатывают выходные результаты).

3. *Функции диалога*. Они решают следующие задачи:

· наблюдают вектор состояния процесса-модели;

· формируют и выводят поток данных, влияющий на действия в процессе-модели;

· выполняют операции для выработки некоторых результатов.

Учет системного времени

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

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

· временной интервал на выработку команды STOP; он должен выбираться путем анализа скорости транспорта и ограничения мощности;

время реакции на включение и выключение ламп панели.

Практическое занятие № 18

Тема 2.7:Инструментарий различных сред разработки

Цель: Изучить инструментарий различных сред разработки

Инструменты разработки программных средств.

При разработке программных средств используется в той или иной мере компьютерная поддержка процессов разработки и сопровождения ПС. Это достигается

путем представления хотя бы некоторых программных документов ПС (прежде всего, программ) на компьютерных носителях данных (например, на дискетах) и предоставлению в распоряжение разработчика ПС специальных ПС или включенных в состав компьютера специальных устройств, созданных для какой-либо обработки таких документов. В качестве такого специального ПС можно указать компилятор с какого-либо языка программирования. Компилятор избавляет разработчика ПС от необходимости писать программы на языке компьютера, который для разработчика ПС был бы крайне неудобен, - вместо этого он составляет программы на удобном ему языке программирования, которые соответствующий компилятор автоматически переводит на язык компьютера. В качестве специального устройства, поддерживающего процесс разработки ПС, можно указать, например, эмулятор какого-либо языка. Эмулятор позволяет выполнять (интерпретировать) программы на языке, отличном от языка компьютера, поддерживающего разработку ПС, например, на языке компьютера, для которого эта программа предназначена.

ПС, предназначенное для поддержки разработки других ПС, будем называть *программным инструментом* разработки ПС, а устройство компьютера, специально предназначенное для поддержки разработки ПС, будем называть *аппаратным инструментом* разработки ПС.

Инструменты разработки ПС могут использоваться в течение всего жизненного цикла ПС для работы с разными программными документами. Так текстовый редактор может использоваться для разработки практически любого программного документа. С точки зрения функций, которые инструменты выполняют при разработке ПС, их можно разбить на следующие четыре группы:

- редакторы,
- анализаторы,
- преобразователи,
- инструменты, поддерживающие процесс выполнения программ.

Редакторы поддерживают конструирование (формирование) тех или иных программных документов на различных этапах жизненного цикла. Как уже упоминалось, для этого можно использовать один какой-нибудь универсальный текстовый редактор. Однако, более сильную поддержку могут обеспечить специализированные редакторы: для каждого вида документов - свой редактор. В частности, на ранних этапах разработки в документах могут широко использоваться графические средства описания (диаграммы, схемы и т.п.). В таких случаях весьма полезными могут быть графические редакторы. На этапе программирования (кодирования) вместо текстового редактора может оказаться более удобным синтаксически управляемый редактор, ориентированный на используемый язык программирования.

Анализаторы производят либо *статическую* обработку документов, осуществляя различные виды их контроля, выявление определенных их свойств и накопление статистических данных (например, проверку соответствия документов указанным стандартам), либо *динамический* анализ программ (например, с целью выявление распределения времени работы программы по программным модулям).

Преобразователи позволяют автоматически приводить документы к другой форме представления (например, форматеры) или переводить документ одного вида к документу другого вида (например, конверторы или компиляторы), синтезировать какой-либо документ из отдельных частей и т.п.

Инструменты, поддерживающие процесс выполнения программ, позволяют выполнять на компьютере описания процессов или отдельных их частей, представленных в виде, отличном от машинного кода, или машинный код с дополнительными возможностями его интерпретации. Примером такого инструмента является эмулятор кода другого компьютера. К этой группе инструментов следует отнести и различные отладчики. По существу, каждая система программирования содержит программную

подсистему периода выполнения, которая выполняет программные фрагменты, наиболее типичные для языка программирования, и обеспечивает стандартную реакцию на возникающие при выполнении программ исключительные ситуации (такую подсистему мы будем называть *исполнительной поддержкой*). Такую подсистему также можно рассматривать как инструмент данной группы.

Инструментальные среды разработки и сопровождения программных средств и принципы их классификации

Компьютерная поддержка процессов разработки и сопровождения ПС может производиться не только за счет использования отдельных инструментов (например, компилятора), но и за счет использования некоторой логически связанной совокупности программных и аппаратных инструментов. Такую совокупность будем называть *инструментальной средой разработки и сопровождения ПС*.

Часто разработка ПС производится на том же компьютере, на котором оно будет применяться. Это достаточно удобно. Во-первых, в этом случае разработчик имеет дело только с компьютерами одного типа. А, во-вторых, в разрабатываемое ПС могут включаться компоненты самой инструментальной среды. Однако, это не всегда возможно. Например, компьютер, на котором должно применяться ПС, может быть неудобен для поддержки разработки ПС или его мощность недостаточна для обеспечения функционирования требуемой инструментальной среды. Кроме того, такой компьютер может быть недоступен для разработчиков этого ПС (например, он постоянно занят другой работой, которую нельзя прерывать, или он находится еще в стадии разработки). В таких случаях применяется так называемый *инструментально-объектный подход*. Сущность его заключается в том, что ПС разрабатывается на одном компьютере, называемым *инструментальным*, а применяться будет на другом компьютере, называемым *целевым* (или *объектным*).

Инструментальная среда не обязательно должна функционировать на том компьютере, на котором должно будет применяться разрабатываемое с помощью ее ПС.

Совокупность инструментальных сред можно разбивать на разные классы, которые различаются значением следующих признаков:

- ориентированность на конкретный язык программирования,
- специализированность,
- комплексность,
- ориентированность на конкретную технологию программирования,
- ориентированность на коллективную разработку,
- интегрированность.

Ориентированность на конкретный язык программирования (*языковая ориентированность*) показывает: ориентирована ли среда на какой-либо конкретный язык программирования (и на какой именно) или может поддерживать программирование на разных языках программирования. В первом случае информационная среда и инструменты существенно используют знание о фиксированном языке (*глобальная ориентированность*), в силу чего они оказываются более удобным для использования или предоставляют дополнительные возможности при разработке ПС. Но в этом случае такая среда оказывается не пригодной для разработки программ на другом языке. Во втором случае инструментальная среда поддерживает лишь самые общие операции и, тем самым, обеспечивает не очень сильную поддержку разработки программ, но обладает свойством *расширения* (*открытости*). Последнее означает, что в эту среду могут быть добавлены отдельные инструменты, ориентированные на тот или иной конкретный язык программирования, но эта ориентированность будет лишь *локальной* (в рамках лишь отдельного инструмента).

Специализированность инструментальной среды показывает: ориентирована ли среда на какую-либо предметную область или нет. В первом случае информационная среда и инструменты существенно используют знание о фиксированной предметной

области, в силу чего они оказываются более удобными для использования или предоставляют дополнительные возможности при разработке ПС для этой предметной области. Но в этом случае такая инструментальная среда оказывается не пригодной или мало пригодной для разработки ПС для других предметных областей. Во втором случае среда поддерживает лишь самые общие операции для разных предметных областей. Но в этом случае такая среда будет менее удобной для конкретной предметной области, чем специализированная на эту предметную область.

Комплексность инструментальной среды показывает: поддерживает ли она все процессы разработки и сопровождения ПС или нет. В первом случае продукция этих процессов должна быть согласована. Поддержка инструментальной средой фазы сопровождения ПС, означает, что она должна поддерживать работу сразу с несколькими вариантами ПС, ориентированными на разные условия применения ПС и на разную связанную с ним аппаратуру, т.е. должна обеспечивать *управление конфигурацией ПС*.

Ориентированность на конкретную технологию программирования показывает: ориентирована ли инструментальная среда на фиксированную технологию программирования либо нет. В первом случае структура и содержание информационной среды, а также набор инструментов существенно зависит от выбранной технологии (*технологическая определенность*). Во втором случае инструментальная среда поддерживает самые общие операции разработки ПС, не зависящие от выбранной технологии программирования.

Ориентированность на коллективную разработку показывает: поддерживает ли среда управление (management) работой коллектива или нет. В первом случае она обеспечивает для разных членов этого коллектива разные права доступа к различным фрагментам продукции технологических процессов и поддерживает работу *менеджеров* [16.1] по управлению коллективом разработчиков. Во втором случае она ориентирована на поддержку работы лишь отдельных пользователей.

Интегрированность инструментальной среды показывает: является ли она интегрированной (и в каком смысле) или нет. Инструментальная среда считается *интегрированной*, если взаимодействие пользователя с инструментами подчиняется единообразным правилам, а сами инструменты действуют по заранее заданной информационной схеме, связаны по управлению или имеют общие части. В соответствие с этим различают три *вида интегрированности*:

- интегрированность по пользовательскому интерфейсу,
- интегрированность по данным,
- интегрированность по действиям (функциям),

Интегрированность по пользовательскому интерфейсу означает, что все инструменты объединены единым пользовательским интерфейсом. *Интегрированность по данным* означает, что инструменты действуют в соответствии с фиксированной информационной схемой (моделью) системы, определяющей зависимость друг от друга различных используемых в системе фрагментов данных (информационных объектов). В этом случае может быть обеспечен контроль полноты и актуальности программных документов и порядка их разработки. *Интегрированность по действиям* означает, что, во-первых, в системе имеются общие части всех инструментов и, во-вторых, одни инструменты при выполнении своих функций могут обращаться к другим инструментам.

Инструментальную среду, интегрированную хотя бы по данным или по действиям, будем называть *инструментальной системой*. При этом интегрированность по данным предполагает наличие в системе специализированной базы данных, называемой *репозиторием*. Под *репозиторием* будем понимать центральное компьютерное хранилище информации, связанной с проектом (разработкой) ПС в течение всего его жизненного цикла.

Основные классы инструментальных сред разработки и сопровождения программных средств

В настоящее время выделяют три основных класса инструментальных сред разработки и сопровождения ПС (рис. 1):

- инструментальные среды программирования,
- рабочие места компьютерной технологии,
- инструментальные системы технологии программирования.

Рис. 1. Основные классы инструментальных сред разработки и сопровождения ПС.

Инструментальная среда программирования предназначена в основном для поддержки процессов программирования (кодирования), тестирования и отладки ПС. Она не обладает рассмотренными выше свойствами комплексности, ориентированности на конкретную технологию программирования, ориентированности на коллективную разработку и, как правило, свойством интегрированности, хотя имеется некоторая тенденция к созданию интегрированных сред программирования (в этом случае их следовало бы называть *системами программирования*). Иногда среда программирования может обладать свойством специализированности. Признак же ориентированности на конкретный язык программирования может иметь разные значения, что существенно используется для дальнейшей классификации сред программирования.

Рабочее место компьютерной технологии ориентировано на поддержку ранних этапов разработки ПС (системного анализа и спецификаций) и автоматической генерации программ по спецификациям. Оно существенно использует свойства специализированности, ориентированности на конкретную технологию программирования и, как правило, интегрированности. Более поздние рабочие места компьютерной технологии обладают также свойством комплексности. Что же касается языковойориентированности, то вместо языков программирования они ориентированы на те или иные формальные языки спецификаций. Свойством ориентированности на коллективную разработку указанные рабочие места в настоящее время, как правило, не обладают.

Инструментальная система технологии программирования предназначена для поддержки всех процессов разработки и сопровождения в течение всего жизненного цикла ПС и ориентирована на коллективную разработку больших программных систем с продолжительным жизненным циклом. Обязательными свойствами ее являются комплексность, ориентированность на коллективную разработку и интегрированность. Кроме того, она или обладает технологической определенностью или получает это свойство в процессе расширения (настройки). Значение признака языковой ориентированности может быть различным, что используется для дальнейшей классификации этих систем.

Инструментальные среды программирования

Инструментальная среда программирования включает, прежде всего, текстовый редактор, позволяющий конструировать программы на заданном языке программирования, а также инструменты, позволяющие компилировать или интерпретировать программы на этом языке, тестировать и отлаживать полученные программы. Кроме того, могут быть и другие инструменты, например, для статического или динамического анализа программ. Взаимодействуют эти инструменты между собой через обычные файлы с помощью стандартных возможностей файловой системы.

Различают следующие классы инструментальных сред программирования (см. рис. 2):

- среды общего назначения,
- языково-ориентированные среды.

Инструментальные среды программирования *общего назначения* содержат набор программных инструментов, поддерживающих разработку программ на разных языках программирования (например, текстовый редактор, редактор связей или интерпретатор языка целевого компьютера) и обычно представляют собой некоторое расширение возможностей используемой операционной системы. Для программирования в такой среде на каком-либо языке программирования потребуются дополнительные инструменты, ориентированные на этот язык (например, компилятор).

Рис.2. Классификация инструментальных сред программирования.

Языково-ориентированная инструментальная среда программирования предназначена для поддержки разработки ПС на каком-либо одном языке программирования и знания об этом языке существенно использовались при построении такой среды. Вследствие этого в такой среде могут быть доступны достаточно мощные возможности, учитывающие специфику данного языка. Такие среды разделяются на два подкласса:

- интерпретирующие среды,
- синтаксически-управляемые среды.

Интерпретирующая инструментальная среда программирования обеспечивает интерпретацию программ на данном языке программирования, т.е. содержит, прежде всего, интерпретатор языка программирования, на который эта среда ориентирована. Такая среда необходима для языков программирования интерпретирующего типа (таких, как Лисп), но может использоваться и для других языков (например, на инструментальном компьютере). *Синтаксически-управляемая* инструментальная среда программирования базируется на знании синтаксиса языка программирования, на который она ориентирована. В такой среде вместо текстового используется синтаксически-управляемый редактор, позволяющий пользователю использовать различные шаблоны синтаксических конструкций (в результате этого разрабатываемая программа всегда будет

синтаксически правильной). Одновременно с программой такой редактор формирует (в памяти компьютера) ее синтаксическое дерево, которое может использоваться другими инструментами.

Основная литература:

1. Основы проектирования компонентов автоматизированных систем: учебное пособие/ Т. В. Волкова Оренбург: ОГУ, 2016, 226 с. То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=471129

2. Сетевые средства Linux. 2-е изд., исправ./ А. И. Бражук - М.: Национальный Открытый Университет «ИНТУИТ», 2016, 148с То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=428794

3. Модели оптимизации. Математическое программирование, исследование операций : учебно-методическое пособие / составители Т. А. Бенгина, В. Г. Саркисов, Л. Н. Смирнова. — 2-е изд. — Самара : Самарский государственный технический университет, ЭБС АСВ, 2018. — 156 с. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/90633.html. — Режим доступа: для авторизир. Пользователей

Дополнительная литература

1. Рудаков А. Технология разработки программных продуктов: учебник. / Рудаков А. - Изд.Academia. Среднее профессиональное образование. 2013 г. 208 стр.