Технология программирования

Курс лекций для гр. 3057/2, 4057/2

Лекция № 9

Содержание

1. Введение
2. Модели жизненного цикла ПП
3. Модели команды разработчиков
4. Управление проектами
5. Словесная коммуникация
6. Языки, модели и методы проектирования
7. Тестирование
8. Надежность ПП
9. CASE-системы
10. Стандарты качества технологии программирования

Понятие надежности программ

Надежность программы- это комплексное свойство из двух составляющих:

- *безотказность,* или *безошибочность (correctness)* – долговременное соответствие требованиям (спецификации)
- *отказоустойчивость (fault-tolerance),* или *устойчивость (robustness)* – способность продолжать нормальное функционирование после отказов программ или аппаратуры
  - *Отказ (failure)* – по ГОСТу - нарушение работоспособности изделия и его соответствия требованиям технической документации
  - Применительно к программам отказ – это неспособность функциональной единицы системы, зависящей от программы, выполнять требуемую функцию в заданных пределах
  - Программный отказ – это проявление ошибки программирования
- Улучшение безотказности достигается хорошей технологией, предупреждающей программные ошибки. Однако 100% отсутствие ошибок недостижимо
- Отказоустойчивость достигается путем *резервирования*: временн**о**го, информационного, структурного или программного

Непосредственные причины программных отказов

Причина	Следствие
Ошибка в реализации алгоритма	⇒ Неверные результаты
Ошибки выполнения, выявляемые автоматически (см. слайд 8 лекции 8)	⇒ Аварийный останов
➢ Бесконечный цикл ➢ Бесконечное ожидание события	} Зависание или крах приложения/системы
Недопустимые входные данные	⇒ Все, что угодно
➢ Программа должна блокировать ввод недопустимых данных – “Idiot-proof UI”	

Отказы и сбои

Восстановление (recovery) - это возврат в исправное состояние путем:

а) ручного ремонта: замены, исправления кода программы

б) автоматически - задействованием *резервов*

в) самопроизвольно (обычно быстро)

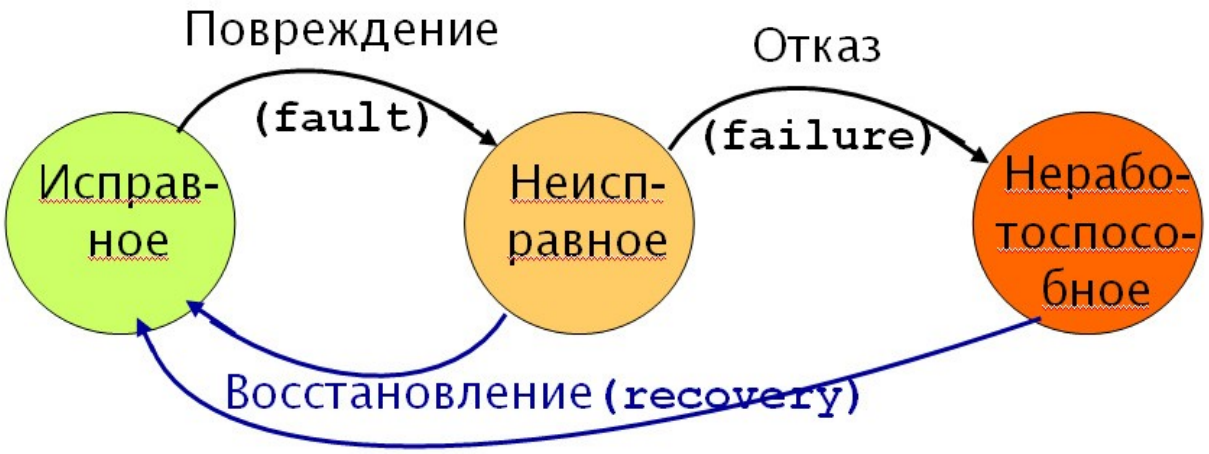
В случае в) отказ называется *сбоем (transient fault)*

Т.е., сбой - это кратковременный самоустраняющийся отказ

- Остальные отказы называются устойчивыми (по умолчанию отказ – устойчивый)
- В электронной аппаратуре сбои происходят на порядок чаще устойчивых отказов. Их причины - флюктуации питания, ситуации "гонок" сигналов, альфа-частицы и др.
- В программах аналогично сбоям ведут себя время-зависимые ошибки - "мерцающие" (blinking bugs)
  - «Самовосстановление» выглядит в этом случае как правильное поведение при повторном выполнении отказавшей функции

NB: не следует путать термины «сбой» и «отказ»

Состояния и переходы при отказе



- Неисправное состояние - промежуточное, когда в результате повреждения (fault) неисправность уже появилась, но еще не проявилась вовне в виде отказа (failure)

  - Напр., из-за ошибки в программе вычислен неверный промежуточный результат, но он еще не использован для вычисления конечного результата, неверное значение которого приведет к отказу системы

Отказ элемента – это повреждение системы: он переводит систему, его содержащую, в неисправное состояние

Время нахождения системы в неисправном, но еще работоспособном состоянии называется латентным (скрытым) периодом отказа

Способы восстановления

Отказоустойчивость обеспечивается постоянным контролем исправности и автоматическим восстановлением путем задействования резервов

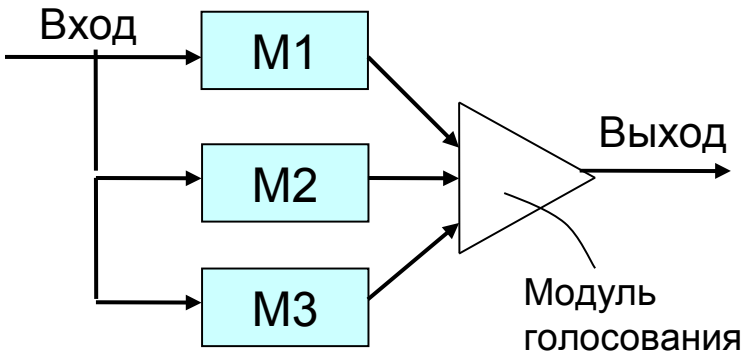
- контроль в программе – с помощью assert и exception
- восстановление – с помощью exception

- Подготовительный шаг: сохранение состояния программы в определенный момент – в точке восстановления (recovery point)
  - Частный случай – периодическое запоминание резервной (backup) копии данных (информационный резерв)
- В случае обнаружения отказа элемента (т.е., неисправности системы) - откат процесса к точке восстановления и затем:
  - повторное выполнение программы (временной резерв)
    - это эффективно при сбоях
  - запуск альтернативной версии программы (временной + программный резерв)
    - эта версия может быть упрощенной (graceful degradation – плавная потеря качества)

Восстановление во время латентного периода отказа делает неисправность невидимой на уровне системы

Восстановление путем маскирования отказа

1. Структурное резервирование



Троирование модулей
(Triple Modular Redundancy, TMR)
Отказ одного из модулей не виден извне

- Аппаратное резервирование: идентичные модули
- Программное резервирование: модули, разработанные разными командами (program diversity)

2. Корректирующие коды: избыточные контрольные символы, позволяющие обнаружить и исправить искаженные символы (информационное резервирование)

- Коды Рида-Соломона: 2N контрольных символов обеспечивают исправление N-кратных ошибок
- Обмен с CD-ROM, передача данных в WiMax и многие другие процессы защищены корректирующими кодами

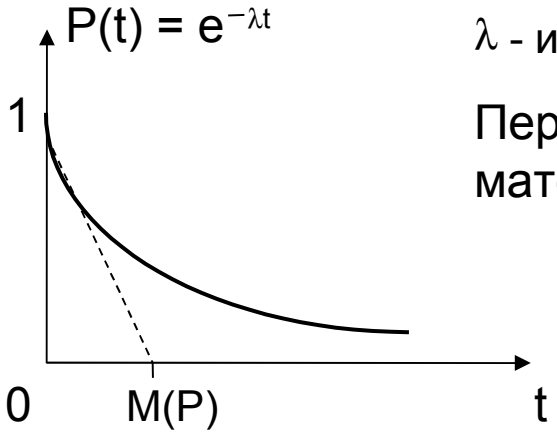
Количественные характеристики надежности программ

- Надежность нужно оценивать, измерять, предсказывать – обеспечивать заданные требования к надежности во время проектирования и проверять их выполнение в продукте
- *Внутренняя* характеристика надежности – количество оставшихся ошибок в программе – интересна больше разработчикам, чем потребителям
  - потребителям важно знать не число ошибок, а частоту их проявления
- Для потребителей важны *внешние* характеристики поведения, традиционные для теории надежности, основанные на предположении о стохастическом (случайном во времени) процессе возникновения отказов:
  - *интенсивность отказов* – среднее их количество в единицу времени
  - *среднее время безотказной работы* (MTBF – Mean Time Between Failures)
  - *коэффициент готовности* (availability): отношение полезного времени работы к полному (включая простои во время восстановления) – интегральная характеристика безотказности и скорости восстановления одновременно

NB: термин availability часто переводят неточно, как «доступность»

Функция распределения времени между отказами

В предположении простейшего потока отказов (отказы независимы, редки и их вероятность неизменна во времени) функция P(t) – вероятность безотказной работы в течение периода времени t – хорошо описывается законом Пуассона (экспоненциальным распределением вероятности)



λ - интенсивность отказов (обычно в 1/час)

Первый момент распределения – математическое ожидание M(P) = MTBF = 1/ λ

В теории надежности технических систем это основная модель отказов, хорошо соответствующая реальности; ее же применяют к описанию надежности программ и следовательно, аппаратно-программных комплексов

Типичные значения характеристик надежности

Вид компонента	MTBF	
	час	лет
Обычная электромеханическая аппаратура	$10^2 – 10^3$	10^{-1}
Обычная электронная аппаратура	$10^3 – 10^4$	1
Большие интегральные схемы	$10^6 – 10^8$	$10^2 – 10^4$
Высоконадежная электронная аппаратура	$10^3 – 10^4$	$10^1 – 10^2$
Программы общего назначения	$10^1 – 10^3$	$10^{-3} – 10^{-2}$
Высоконадежные программы	$10^4 – 10^5$	$10^{-1} – 10^1$

Основной источник ненадежности вычислительных систем – программы

В критически важных приложениях требуется очень малая вероятность отказов. Напр., для бортовой системы управления космическим зондом требуется $\lambda = 10^{-9}$ 1/час, чтобы вероятность отказа в первые 10 лет работы была не более 10^{-4} (или вероятность безотказной работы 0,9999), что означает MTBF = 100 тысяч лет !

Методы оценки и измерения характеристик надежности программ

А. Внутренняя характеристика – число оставшихся ошибок – абсолютное или относительное

N - число строк исх. кода на 1 ошибку

Вид программы	N
Модуль до начала независимого тестирования	100
Рядовой коммерческий ПП	500
Системный ПП	5000
ПП критического назначения	50000

- За четыре года эксплуатации ядра Linux 2.6 зарегистрировано менее 1000 ошибок на 5,7 млн. строк кода, т.е. одна ошибка на 6000 строк
- Типичный уровень для Windows – 1 ошибка на 1000 строк исходного кода
- Windows 7: 2000 ошибок исправлено по результатам бета-тестирования

Как проверить соответствие ПП требованиям заданного уровня плотности ошибок?

Метод Миллса (IBM, 1972)

- метод «меченых рыб»

- Группа тестирования искусственно засоряет программу ошибками и, продолжая тестирование, анализирует долю внесенных ошибок среди обнаруживаемых
- Пусть внесено S ошибок, обнаружено L собственных и V внесенных ошибок
  - Предположение: вероятность обнаружения внесенных и собственных ошибок одинакова
  - Тогда первоначальное количество оставшихся ошибок N = S * L / V
- Проверка гипотезы: осталось k собственных ошибок:

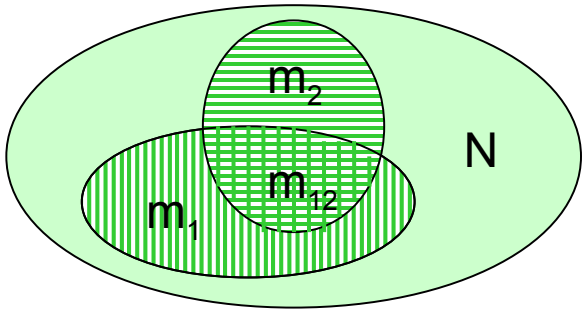
вносится еще S ошибок и тестирование продолжается, пока все они не обнаружены

  - Пусть к этому моменту найдено n собственных ошибок
  - Тогда уровень значимости гипотезы (т.е., вероятность того, что она истинна) = 0 при n > k и S / (S + k +1) при n ≤ k
    - (это формула мат. статистики – не для запоминания)

Метод Руднера (1977)

- тестирование осуществляется двумя независимыми группами тестеров параллельно

- Группы обнаруживают m_1 и m_2 ошибок соответственно, из которых m_{12} ошибок – общие
- Предполагаем равную вероятность обнаружения для всех ошибок
- Обозначим N - число первоначально содержавшихся в программе ошибок
- Эффективность каждой из групп: E1 = m_1/N и E2= m_2 /N; эту же эффективность группы показали для любого случайным образом выбранного подмножества ошибок
- В частности, первая группа – для множества ошибок, найденных второй группой: E1 = m_{12}/m2. Следовательно, $m_1/N = m_{12}/m_2$ и $N = m_1 * m_2 / m_{12}$



Оставшееся число ошибок =

$m_1 * m_2 / m_{12} - m_1 - m_2 + m_{12}$

Метод Моторолы

- **Моторола** использует в качестве меры надежности своих ПП – управляющих программ реального времени – среднее число ошибок на 1000 строк исходного кода
- Фирма разработала метод оценки времени тестирования без отказов (zero-failure method), подтверждающего заданную надежность в смысле количества оставшихся ошибок
- Метод основан на некоторой, принятой фирмой Моторола, но не публикуемой зависимости $T = f(N, n, t)$, где:

T – время окончательного тестирования – испытаний без отказов

N – допустимое число ошибок в коде

n – общее число ошибок, обнаруженных в ходе тестирования

t – время, потраченное на выявление всех n ошибок

- Если за время T ни одного отказа не произошло, считается, что в программе не более N ошибок и тестирование можно закончить
- Если же отказ произошел через промежуток времени $t_r < T$, то испытание продолжается, причем функция T пересчитывается заново для n+1 и $t + t_r$
- Очевидно, существо метода скрыто в виде функции $T = f(N, n, t)$, которая была найдена экспериментально

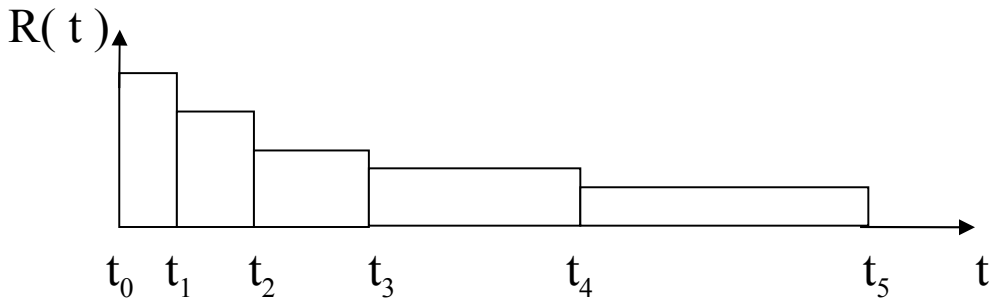
Модель Джелинского – Моранды

Б. Оценка одновременно и числа оставшихся ошибок, и внешней характеристики – интенсивности отказов

Допущения:

1. Интенсивность обнаружения ошибок $R(t)$ пропорциональна текущему числу ошибок в программе, т.е. числу исходных ошибок минус обнаруженные.
2. Все ошибки равновероятны и их проявление не зависит друг от друга
3. Ошибки постоянно исправляются без внесения новых.
4. $R(t)$ постоянна в промежутке между двумя смежными моментами обнаружения t_i и t_{i+1}.
5. Предполагается, что интенсивность проявления ошибок во время эксплуатации остается такой же, как при тестировании, т.е. значение λ принимается равным $R(t_n)$ в момент окончания тестирования t_n ($\lambda(t)$ "замораживается", т.к. прекращается исправление ошибок)

Тогда $R(t)$ – ступенчатая монотонно убывающая функция вида:



Модель Джелинского – Моранды (2)

Если N – число ошибок, обнаруженных к моменту времени t_i, то до следующего обнаружения в момент t_{i+1} справедливо $R(t) = K * (B - N)$

- где B – неизвестное число исходных ошибок,
- K – неизвестный коэффициент пропорциональности

Сняв допущение 4, перейдем к упрощенной модели, где функция $R(t)$ непрерывная: $R(t) = K(B - N(t))$. Дифференцируя это равенство, получаем:

$$\frac{dR(t)}{dt} = -K\frac{dN(t)}{dt};$$

и поскольку $dN(t)/dt = R(t)$ - числу ошибок, обнаруженных в единицу времени, получаем следующее дифференциальное уравнение с начальными условиями:

$$\frac{dR(t)}{dt} + KR(t) = 0; R(0) = KB$$

Его решение: $R(t) = KB\, e^{-Kt}$. Обозначая $a = \ln(KB)$, $b = -K$, получаем запись решения в виде: $R(t) = \exp(a + bt)$. Логарифмируя обе части этого равенства и переходя к дискретному времени t_i, получим систему n уравнений:

$$\ln R(t_i) = a + b\,t_i; \quad i = 1, \ldots, n$$

Модель Джелинского – Моранды (3)

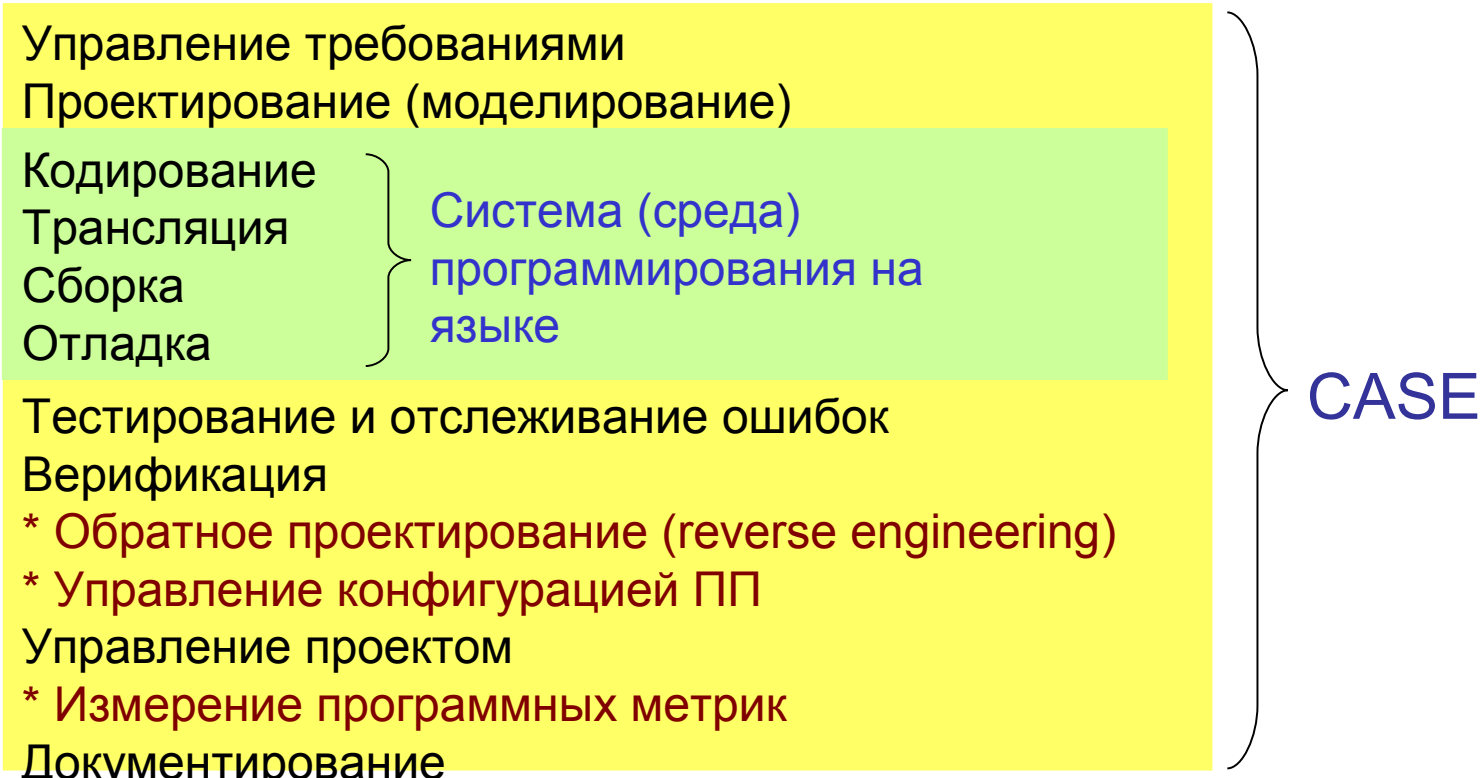
Решая эту систему относительно двух неизвестных a и b методом наименьших квадратов (при этом ищутся оценки мат. ожиданий случайных величин a и b), находим оценки величин K и B в соответствии с критерием максимального правдоподобия и, значит:

- оценка числа оставшихся ошибок: $B - N$
- искомое значение интенсивности отказов: $\lambda = R(t_n)$

- Этот метод был применен для ряда конкретных проектов IBM, в том числе программного обеспечения Apollo (пилотируемый полет на Луну)
- Метод наиболее хорошо подходит для оценки надежности крупномасштабных программных разработок с продолжительным периодом тестирования

CASE-средства

- средства автоматизации всех видов работ по разработке ПП:



В идеале, CASE-средства образуют единый конвейер

Обратное проектирование

- Это сравнительно распространенная задача восстановления проекта (т.е. высокоуровневого описания) программы, не имеющей такого описания
- Задача возникает, если нужно переписать имеющийся ПП на другом языке или перенести его в новую среду:
  - при переносе программы на новую платформу (например, игры с одной приставки на другую)
  - при модернизации так назыв. «унаследованных» (legacy) ПП (например, переписывание программы с Кобола на C++)
  - при подключении ПП с простым файловым интерфейсом к базе данных
- Существуют CASE-средства обратного проектирования, напр. SEEC Reengineering Workbench

Управление конфигурацией ПП

Software Configuration Management (SCM)

- управление *изменяющимся* в процессе разработки наборами файлов ПП:
  - исполняемые модули (.exe и .dll)
  - исходные модули
  - документация (требования, проект, пользовательская докум., …)
  - тестовые наборы, сценарии и отчеты
  - обрабатываемые данные (напр., графика)

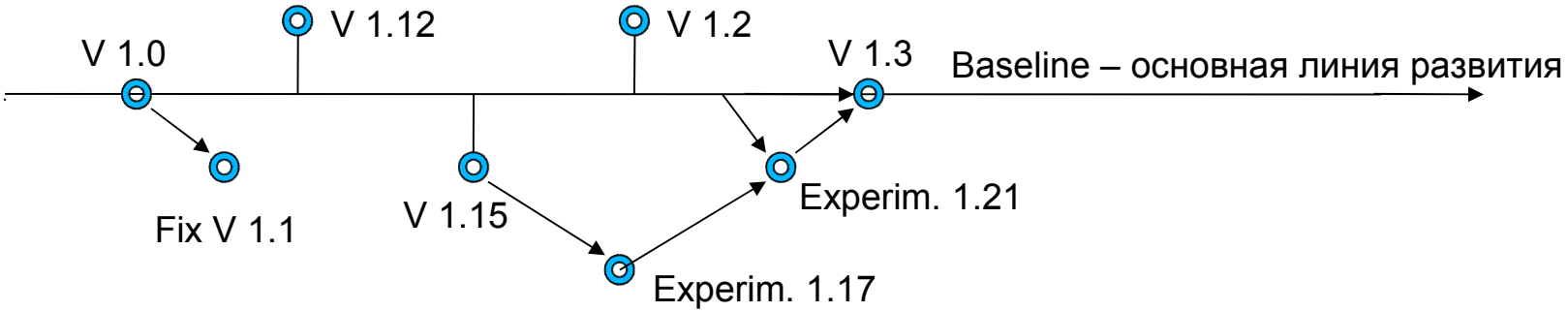
Главная задача – поддержание целостности (consistency) наборов файлов (взаимного соответствия файлов) и целостности всего ПП (соответствие наборов друг другу)

Эту задачу решают системы управления версиями (Version Control Systems):

- CVS (Concurrent Version System) — под лицензией GNU
- VSS (Visual Source Safe) – от Microsoft, интегрирована с Visual Studio
- PVCS (Program Version Control System) — содержит Tracker – систему отслеживания ошибок
- Perforce – для файлов большого объема (напр., мультимедиа)
- ClearCase – от IBM Rational

SCM: многоверсионность

- Сборка (Build) — полный комплект модулей для трансляции и сборки программы
- Версия (Version) — «замороженная» сборка (изменения временно прекращены)
- Выпуск (Release) — отчуждаемая версия



- Все сборки и версии хранятся в общем *репозитории* в течение всего проекта
- Последовательные версии кода *мало* отличаются друг от друга
  - Значит, можно хранить только различия между файлами — экономия памяти
- Старые версии должны храниться долго, чтобы к ним можно было вернуться при обнаружении ошибки в более поздних версиях
- Обычно работа ведется параллельно над несколькими версиями
  - Напр., одна тестируется заказчиком, другая -тестерами, третья в стадии развития
- Ошибка, обнаруженная в одной из версий, должна быть как можно быстрее исправлена во всех других актуальных версиях

SCM: коллективная разработка

Одновременная модификация: когда два или несколько программистов работают по отдельности над одним и тем же модулем, то тот из них, кто реально вносит изменения, может уничтожить работу остальных.
Решения:

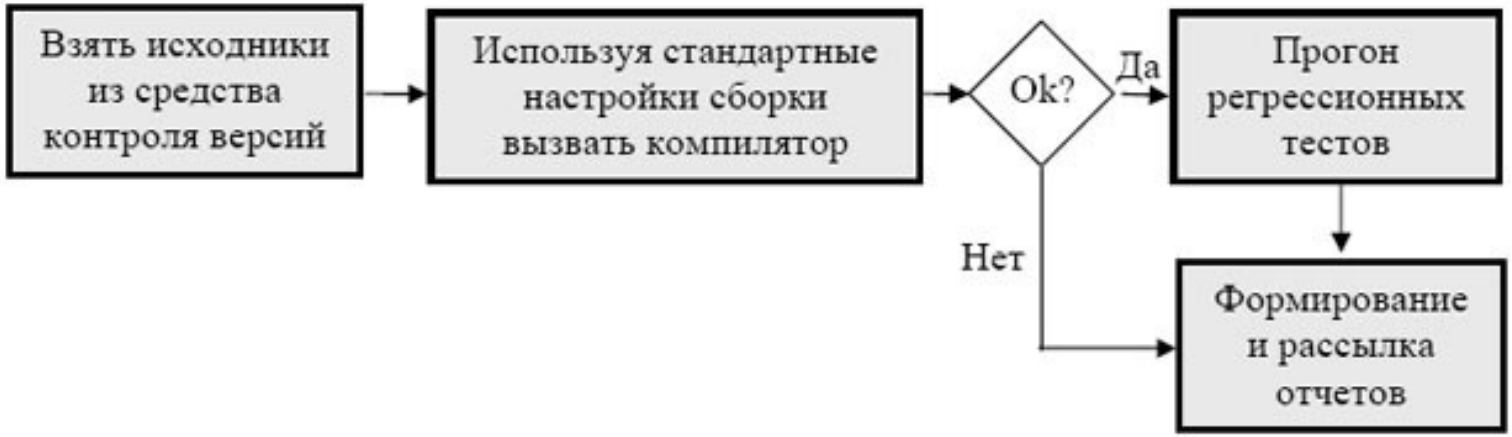
- Запрет доступа к файлу для записи нескольким программистам одновременно: функция lock/unlock
- «Интеллектуальное» слияние изменений

Общий код: когда в нем исправляется ошибка или вносятся изменения, все участники немедленно должны узнать об этом

- Решение: автоматическая рассылка сообщений об изменениях при выполнении функции check in – при записи программистом новых версий измененных им файлов в репозиторий
- (Обратная функция - check out – выдает любую из хранимых версий одного или нескольких файлов)

SCM: управление сборками в VSTS

VSTS – Virtual Studio Team Service



Регулярная (напр., еженощная) автоматическая трансляция и сборка кода

CASE-инструменты IBM Rational

Наиболее важные из более чем 20 инструментов:

1. Rational Unified Process (RUP) - методологическая основа для всего, что выпускает Rational:
   - Руководства для всех членов команды и для каждого этапа жизненного цикла ПП
   - Инструкции по пользованию инструментальными средствам Rational, которые автоматизируют все этапы процесса создания ПП
   - Примеры и шаблоны для Rational Rose
   - Шаблоны планов-графиков в формате Microsoft Project - отражают итерационную разработку
   - Development Kit – инструменты и шаблоны для конфигурирования и расширения RUP для специфических нужд проекта
   - Доступ к Resource Center в интернете, который содержит последние публикации, обновления, подсказки, методики, ссылки на сервисы

Инструменты IBM Rational (2)

2. RequisitePro – инструмент для ввода и управления требованиями
   - Задание иерархий требований, установка их приоритетов
   - Назначение требований конкретным исполнителям
   - Трассировка связей между требованиями
   - Хранение истории изменения каждого требования

3. Rational Rose – инструмент проектирования программы на UML
   - Создание и синтаксический контроль графической модели
   - Кодогенерация: преобразование нарисованной модели в текст на конкретном языке программирования: C++, Ada, Java, VB и т.д.
   - Обратное проектирование: готовую программу на C++ или базу данных (на Oracle) вводят в Rose и получают наглядную визуальную структурную модель

4. SoDa - автоматизация документирования ПП на базе шаблонов: более 70 шаблонов документов, соответствующих RUP

5. ClearCase - SCM

Программные метрики

- количественные показатели, характеризующие ПП («метрики кода») и производственный процесс. Они используются для:

- Предварительной, текущей и итоговой оценки экономических параметров проекта: трудоемкости, длительности, производительности труда, стоимости
- Оценки рисков по проекту: риска нарушения сроков и невыполнения проекта, риска увеличения трудоемкости на поздних этапах проекта и пр.
- Принятия оперативных управленческих решений: на основе отслеживания метрик проекта можно своевременно предупредить возникновение нежелательных ситуаций

Метрики кода: размер, сложность, надежность (см. след. слайд)

Метрики процесса разработки: значения метрик кода для разных фрагментов программы/ разных исполнителей, что показывают динамику их изменения, напр.:

- Динамика изменений кода: количество добавленных, удаленных и измененных строк в очередной версии по отношению к предыдущей
- Динамика тестирования / отладки

Метрики кода

Метрики размера

- Количество строк исходного кода (Lines of Code – LOC) - наиболее простой способ оценки объема работ
- Длина программы по Холстеду: число операторов + число вхождений операндов (т.е., упоминаний переменных)
- Мера документированности: процент строк комментариев (20% считается нормой)
- Метрика для прогнозирования размеров кода: среднее число строк для функций (классов, файлов)

Метрики надежности

- Плотность ошибок (выявленных и оставшихся)
- Оценка среднего времени наработки на отказ

Метрика сложности

- Цикломатическая сложность управляющего графа программы:

  $C = e - n + 2$, где e – число ребер, а n – число узлов

Стандарты качества технологии

Популярны два международных стандарта: ISO 9000 и CMM

- ISO 9000 – группа универсальных (для любых отраслей) стандартов International Organization for Standardization (ISO)
- Их преимущества: известность, распространенность, признание на мировом уровне, большое количество экспертов и аудиторов и невысокая стоимость услуг по сертификации
- Но их универсальность приводит к недостаткам:
  - они - высокоуровневые, задают абстрактные модели
  - в них лишь упоминаются требования, которые должны быть реализованы, но не говорится о том, как это можно сделать

Стандарт CMMI

CMM (Capability Maturity Model) - "модель зрелости процесса разработки" (или "модель совершенствования возможностей")

- Стандарт разработан в Software Engineering Institute (SEI) - подразделении Университета Карнеги-Меллона в Питтсбурге
- Начальной целью разработки стандарта в 1991 году было создание методики оценки процессов разработки ПП , позволяющей правительственным организациям США выбирать наилучших поставщиков ПП
- Стандарт оказался пригодным и для обычных компаний-разработчиков, желающих улучшить существующие процессы
- В 2002 г. опубликована новая, продвинутая версия CMM – CMM Integration (CMMI)

Стандарт CMMI (2)

Главное понятие стандарта - зрелость (maturity) организации

Незрелой считается организация, в которой:

- ✓ отсутствует долговременное и проектное планирование
- ✓ процесс разработки программного обеспечения и его ключевые составляющие не идентифицированы, реализация процесса зависит от текущих условий, конкретных менеджеров и исполнителей
- ✓ методы и процедуры не стандартизированы и не документированы
- ✓ результат не предопределен
- ✓ процесс выработки решения происходит стихийно, на грани искусства

Стандарт CMMI (3)

Основные признаки зрелой организации:

- ✓ в компании имеются четко определенные и документированные процедуры управления требованиями, планирования проектной деятельности, создания и тестирования программных продуктов, отработанные механизмы управления проектами
- ✓ эти процедуры постоянно уточняются и совершенствуются
- ✓ оценки времени, сложности и стоимости работ основываются на накопленном опыте, разработанных метриках и количественных показателях, что делает их достаточно точными
- ✓ существуют обязательные для всех правила оформления методической, программной и пользовательской документации
- ✓ технологии незначительно меняются от проекта к проекту на основании стабильных и проверенных подходов и методик
- ✓ максимально используются наработанные в предыдущих проектах организационный и производственный опыт, программные модули, библиотеки программных средств
- ✓ активно апробируются и внедряются новые технологии, производится оценка их эффективности

Уровни зрелости по CMMI

Аттестация по стандарту CMMI

- При сертификации эксперты проводят оценку соответствия всех ключевых областей по 10-балльной шкале
- Для успешной квалификации данной ключевой области необходимо набрать не менее 6 баллов и ежегодно подтверждать эту аттестацию
- Это следует делать всем компаниям, выпускающим заказные продукты, особенно по модели оффшорного программирования
- В 2005 году в мире существовало всего 8 компаний, аттестованных по пятому уровню CMMI (шесть индийских и по одной из России и США)
- С другой стороны, насчитывалось несколько тысяч компаний, сертифицированных по 3 или 4 уровню CMM и 140 – то же для CMMI
  - То есть, колоссальный разрыв между оптимизированным уровнем зрелости и предыдущими уровнями
- Свыше 70% всех компаний-разработчиков ПП находятся на 1 уровне

Заключение

- Надежность ПП характеризуют два свойства: безотказность и отказоустойчивость
- Количественные показатели надежности ПП – те же, что и для аппаратуры: интенсивность отказов, среднее время между отказами и коэфф. готовности
- Еще один показатель - число оставшихся ошибок – используется производителем для нужд оценки качества процесса
- Интенсивность программных отказов существенно выше интенсивности аппаратных отказов
- Отказы аппаратных и программных элементов системы делаются невидимыми на уровне системы с помощью средств отказоустойчивости: контроля и резервирования
- Измерение программных метрик – основа количественного управления качеством ПП
- Сертификат качества производителям ПП выдается по результатам проверки соответствия стандартам ISO или CMMI

Дополнительные вопросы

1. Почему надежность вычислительной аппаратуры удалось повысить в последние десятилетия на несколько порядков, а сделать это для программ – нет?
2. Какова величина MTBF, по вашим наблюдениям, у Windows XP ? У Word'а ? У других распространенных ПП ?
3. Какие предположения метода Миллса представляются далекими от реальности?
4. Какова по методу Миллса вероятность того, что в программе больше не осталось ошибок, если мы внесли искусственно 4 ошибки, нашли именно их при последующем тестировании, причем не обнаружили собственных ошибок ?
5. Насколько модель метода Руднера отвечает реальности?
6. Какие из допущений 1 – 5 на слайде 16 представляются вам наиболее проблематичными ?
7. Почему использование стандартных возможностей файловой системы для хранения версий (например, в отдельных каталогах с именами – датами версий) плохо решают проблему множественности версий ? (слайд 22)

Общий вопрос по курсу:

- Какие темы лекционного курса содержали новую для вас информацию, а какие – уже известную? Какие темы следовало бы расширить?

Дополнительные вопросы

Эта проблема не возникает в модели разработки, при которой рабочие множества файлов большинства программистов не пересекаются, а руководитель проекта занимается планированием их работы и объединением результатов труда. Для проектов небольшого размера такой подход является вполне допустимым. Почему он не проходит для больших проектов со сжатыми сроками разработки?

В каких ситуациях и почему такое запирание файла для записи нежелательно?

В чем недостатки такого способа оценки трудоемкости?

К какому уровню из пяти вы отнесли бы технологию в организации, где вы работаете?

Какие темы лекционного курса содержали новую для вас информацию, а какие – уже известную?

Какие темы следовало бы расширить?