Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Кафедра Телематики (при ЦНИИ РТК)

Михаил Александрович Курочкин, Алексей Владимирович Востров

Тестирование ПО

Слайды видеолекций

Учебное пособие

Санкт-Петербург, 2015

Содержание курса

ЛЕКЦИЯ 1. Методы тестирования программного обеспечения
ЛЕКЦИЯ 2. Психологические аспекты тестирования ПО, стратегии тестирования, принципы тестирования
ЛЕКЦИЯ 3. Инспекции, сквозные просмотры и обзоры программ
ЛЕКЦИЯ 4. Проектирование тестов, метод белого ящика, метод черного ящика, причинно-следственные диаграммы
ЛЕКЦИЯ 5. Модульное тестирование, инкрементное тестирование, нисходящее, восходящее тестирование
ЛЕКЦИЯ 6. Высокоуровневое тестирование, функциональное, системное, приемочное тестирование
ЛЕКЦИЯ 7. Тестирование удобства пользователя
ЛЕКЦИЯ 8. Отладка, планирование и контроль тестирования, критерии завершения тестирования, приемочное тестирование
ЛЕКЦИЯ 9. Основные положения стандарта ISO/IEC 12207 (структура процессов жизненного цикла ПО) требования к тестированию верхнего и нижнего уровня

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

Лекция 1

Методы тестирования программного обеспечения

Санкт-Петербургский государственный политехнический университет
2015

Методы верификации программного обеспечения
Лекция 1

Введение, основные понятия. Проблемы верификации, ошибки в программах и их последствия. Тестирование и верификация

Тестирование программного обеспечения — процесс выявления ошибок в программном обеспечении (ПО). Методы тестирования ПО не позволяют однозначно и полностью устранить все дефекты и ошибки и установить корректность функционирования анализируемой программы особенно в закрытых частных программах. Поэтому все существующие методы тестирования действуют в рамках формального процесса проверки исследуемого или разрабатываемого ПО.

Существует множество подходов к решению задачи тестирования и верификации ПО, но эффективное тестирование сложных программных продуктов — это процесс в высшей степени творческий, не сводящийся к следованию строгим и чётким процедурам или созданию таковых.

Классификация по объекту тестирования:

Функциональное тестирование (functional testing)

Нагрузочное тестирование

- Тестирование производительности (perfomance/stress testing)
- Тестирование стабильности (stability/load testing)

Тестирование удобства использования (usability testing)

Тестирование интерфейса пользователя (UI testing)

Тестирование безопасности (security testing)

Тестирование локализации (localization testing)

Тестирование совместимости (compatibility testing)

Классификация по знанию системы:

- Тестирование чёрного ящика (black box)
- Тестирование белого ящика (white box)
- Тестирование серого ящика (gray box)
- **По степени автоматизированности:**
- Ручное тестирование (manual testing)
- Автоматизированное тестирование (automated testing)
- Полуавтоматизированное тестирование (semiautomated testing)

Методы верификации программного обеспечения
Лекция 1

Классификация

По степени изолированности компонентов:

- Компонентное (модульное) тестирование (component/unit testing)
- Интеграционное тестирование (integration testing)
- Системное тестирование (system/end-to-end testing)

По времени проведения тестирования:

- Альфа тестирование (alpha testing)
  - Тестирование при приёмке (smoke testing)
  - Тестирование новых функциональностей (new feature testing)
  - Регрессионное тестирование (regression testing)
  - Тестирование при сдаче (acceptance testing)
- Бета тестирование (beta testing)

Классификация
по признаку позитивности сценариев:
•Позитивное тестирование (positive testing)
•Негативное тестирование (negative testing)

по степени подготовленности к тестированию:
•Тестирование по документации (formal testing)
•Эд Хок (интуитивное) тестирование (ad hoc testing)

Верификация

Проверка на модели — это автоматический метод верификации параллельных систем с конечным числом состояний. Он обладает рядом преимуществ по сравнению с традиционными подходами к задаче верификации, основанными на моделировании, тестировании и дедуктивном анализе.

Верификация

Метод *верификации моделей* (или *проверки на модели)* применяется для верификации систем с конечным числом состояний. Одно из преимуществ, связанных с указанным ограничением, заключается в том, что проверка может быть осуществлена чисто автоматически. Обычно процедура верификации заключается в исчерпывающем обходе пространства состояний системы, для того чтобы выяснить, выполняется ли спецификация. При наличии достаточных ресурсов эта процедура всегда *завершается* с ответом <да» или <нет*.

В идеальном случае верификация проводится полностью автоматически. Однако на практике она часто требует содействия человека. Одной из сторон деятельности человека является анализ результатов верификации. Если результаты проверки отрицательные, то пользователю нередко предоставляют трассу, содержащую ошибку. Она строится в качестве контрпримера для проверяемого свойства и может помочь проектировщику проследить, где возникает ошибка. В этом случае анализ ошибочной трассы может повлечь за собой модификацию системы и повторное применение алгоритма проверки на модели.

Методы верификации программного обеспечения
Лекция 1

Контрольные вопросы

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 2

Психологические аспекты тестирования ПО, стратегии тестирования, принцип тестирования.

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

- **Стратегии тестирования**
- **Принципы тестирования программного обеспечения**

Тестирование

- “Тестирование — это процесс выполнения программы с целью обнаружения ошибок”.
- Тестирование программного продукта правильнее рассматривать как деструктивный процесс, нацеленный на обнаружение ошибок (причем их наличие заранее предполагаться)

Стратегии тестирования (1)

- Для того чтобы процесс тестирования имел оправданную с экономической точки зрения трудоемкость, необходимо заранее выработать ряд стратегий. Две наиболее распространенные из них — это тестирование методом “черного” и “белого” ящиков

Тестирование методом "черного ящика"

- Одной из важнейших стратегий тестирования является тестирование *методом черного ящика* (black-box testing), которое также называют *тести-рованием, управляемым данными* (data-driven testing), или *тестированием,управляемым входом и выходом* (input/output-driven testing). В соответствии с этим методом программа рассматривается как “черный ящик”, внутреннее поведение и структура которого не имеют никакого значения. Вместо этого все внимание фокусируется на выяснении обстоятельств, при которых поведение программы не соответствует спецификации.

Тестирование методом "белого ящика"

- В стратегии тестирования *методом белого ящика* (white-box testing), называемой также *тестированием, управляемым логикой программы* (logic- *driving testing*), разрешается исследовать внутреннюю структуру программы. Исходя из этой стратегии, тестировщик подбирает тестовые данные путем анализа логики программы (к сожалению, нередко забывая обращаться к ее спецификации).

Принципы тестирования программного обеспечения

- Необходимая часть любого теста - описание ожидаемых выходных данных или результатов
- Программист должен избегать тестирования собственных программ
- Компания-разработчик не должна тестировать собственное программное обеспечение
- Любой процесс тестирования должен включать в себя детальное изучение результатов каждого теста
- Готовьте тесты как для корректных и ожидаемых, так и для некорректных и непредсказуемых входных данных
- Проверка, делает ли программа то, что должна, составляет лишь ползадачи; вторая половина задачи — выяснить, не делает ли программа того, чего не должна делать

Принципы тестирования программного обеспечения (2)

- Не выбрасывайте тесты, если только программа, для которой они предназначены, не является фактически одноразовой
- Не приступайте к планированию тестов, заранее настраиваясь на то, что ошибки не будут обнаружены
- Вероятность того, что в некоторой части программы остались необнаруженные ошибки, прямо пропорциональна количеству уже обнаруженных там ошибок
- Тестирование — творческий процесс, требующий интеллектуальных усилий

Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 3

Инспекции, сквозные просмотры и обзоры программ

Инспекции и сквозные просмотры

.

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

- **Инспекция кода**
- **Группа инспектирования кода**
- **Сквозные просмотры**
- **Рецензирование**

Инспекции и сквозные просмотры

- Тремя основными методами ручного тестирования являются *инспекция кода* (code inspection), *сквозной просмотр* (walkthrough) и *тестирование удобства использования* (usability testing).
-
- Первые два метода, ориентированные на работу с кодом и могут применяться на любой стадии разработки программного обеспечения.
-
- Цель— нахождение ошибок, но не их устранение (т.е. тестирование, а не отладка).

Инспекция кода (1)

- Инспекция кода — это набор процедур и методик обнаружения ошибок путем анализа (чтения) кода группой специалистов. В большинстве случаев, основное внимание уделяют процедурным вопросам, формам и подлежащим заполнению.

Группа инспектирования кода

- В состав группы входят четыре человека, один из которых играет роль координатора. Координатор - квалифицированный программист, но не автором тестируемой программы, детальное знание которой от него не требуется. В его обязанности входит следующее:
- раздача материалов участникам заседания инспекционной группы и составление плана его проведения;
- проведение заседания;
- запись всех обнаруженных ошибок;
- последующая проверка того, что обнаруженные ошибки устранены

Группа инспектирования кода (2)

- Вторым участником группы является программист, а остальными — проектировщик программы и специалист по тестированию, знающий. методы тестирования ПО, и наиболее распространенные типы ошибок.

Сквозные просмотры

- Сквозной просмотр кода представляет собой совокупность процедур и методов обнаружения ошибок в коде путем его просмотра участниками группы тестирования. Сквозной просмотр проводится в форме непрерывного заседания, длящегося не более двух часов. Число участников группы, выполняющей сквозной просмотр, составляет 3-5 человек (координатор, секретарь, тестировщик, автор программы).

Сквозные просмотры

- Участники заседания “играют в компьютер”. Тестировщик, приходит на совещание с предварительно записанными на листах бумаги тестами — представительными наборами входных (и ожидаемых выходных) данных для программы или модуля. Во время заседания участники “прогоняют в уме” каждый тест, т.е. подвергают тестовые данные обработке вручную в соответствии с логикой программы. Состояние программы (значения переменных) отслеживается на бумаге или на доске.

Рецензирование

- *Рецензирование* (Peer rating) — это процедура анонимной оценки общих характеристик качества, обслуживаемости, расширяемости, удобства использования и ясности программного обеспечения. Цель данного метода — предоставить программисту возможность получить стороннюю оценку результатов своего труда.
- Руководство процессом осуществляет администратор, выбираемый из числа программистов. В свою очередь, администратор отбирает в группу рецензентов от 6 до 20 участников (6 - это необходимый минимум, обеспечивающий анонимность оценок). Предполагается, что все участники специализируются в одной области. Каждый из участников предоставляет для рецензирования две своих программы, одну из которых он считает наилучшей, а вторую — наихудшей по качеству.

Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 4

Проектирование тестов, метод белого ящика, метод черного ящика, причинно-следственные диаграммы

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

- **методы проектирования тестов**
- метод “белого ящика”
- **Тестирование путем покрытия логики про**
- покрытие условий
- покрытие решений и условий
- **Тестирование методом "черного ящика«**
- ***Определение классов эквивалентности***

методы проектирования тестов

Методы "черного ящика"	Методы "белого ящика"	
Эквивалентное разбиение	Покрытие операторов	
Анализ граничных значений	Покрытие решений	
Причинно-следственные диаграммы	Покрытие условий	
Прогнозирование ошибок	Покрытие решений и условий Комбинаторное покрытие условий	

метод “белого ящика”

- При тестировании методом “белого ящика” учитывается степень покрытия логики (кода) программы. Исчерпывающее тестирование методом “белого ящика” требует прохождения каждого пути в программе, однако в случае программ, содержащих циклы, удовлетворение этого условия — задача нереальная.

покрытие условий

- В некоторых ситуациях более эффективным по сравнению с критерием покрытия решений является критерий покрытия условий (condition coverage). В соответствии с этим критерием количество тестов должно быть таким, чтобы каждое из элементарных условий, образующих проверочное выражение в точке ветвления, принимало каждое из двух возможных логических значений, true и false, по крайней мере один раз.
- Поскольку, как и при покрытии решений, покрытие условий не всегда обеспечивает выполнение каждой инструкции, оно должно быть дополнено требованием, чтобы каждая точка входа в программу или подпрограмму, а также каждый блок **ON** были пройдены хотя бы один раз

покрытие решений и условий

- Очевидным способом разрешения этой дилеммы является введение критерия, называющегося *покрытие решений и условий* (decision/ condition coverage). Согласно этому критерию набор тестов является достаточно полным, если удовлетворяются следующие требования: каждое условие в решении принимает каждое возможное значение по крайней мере один раз, каждый возможный исход решения проверяется по крайней мере один раз и каждой точке входа управление передается по крайней мере один раз.

Тестирование методом "черного ящика"

- Тестирование методом “черного ящика” основано на спецификациях программ. Целью тестирования является выяснение обстоятельств, при которых поведение программы не соответствует ее спецификации.
- Метод тестирования по принципу “черного ящика”, предусматривает процессы: *разбиение на классы эквивалентности* (equivalence partitioning); разработка набора “специфических” условий, подлежащих тестированию.

Определение классов эквивалентности

- В данном случае следует определить два типа классов: *допустимые классы эквивалентности* (valid equivalence classes), представляющие допустимые входные данные программы, и *недопустимые классы эквивалентности* (invalid equivalence classes), представляющие все остальные возможные состояния условий (т.е. недопустимые входные значения).

Анализ граничных значений

- Граничные условия — это ситуации, возникающие в области граничных значений входных и выходных классов эквивалентности. Анализ граничных значений отличается от методики разбиения на классы эквивалентности в двух отношениях.
- Вместо того чтобы выбирать любой элемент класса эквивалентности в качестве представителя всего класса, анализ граничных значений требует выбирать такой элемент или элементы, которые обеспечивают тестирование каждой границы класса.
- При создании тестов внимание фокусируется не только на входных условиях (пространство входных значений), но и на *пространстве результатов* (выходные классы эквивалентности).

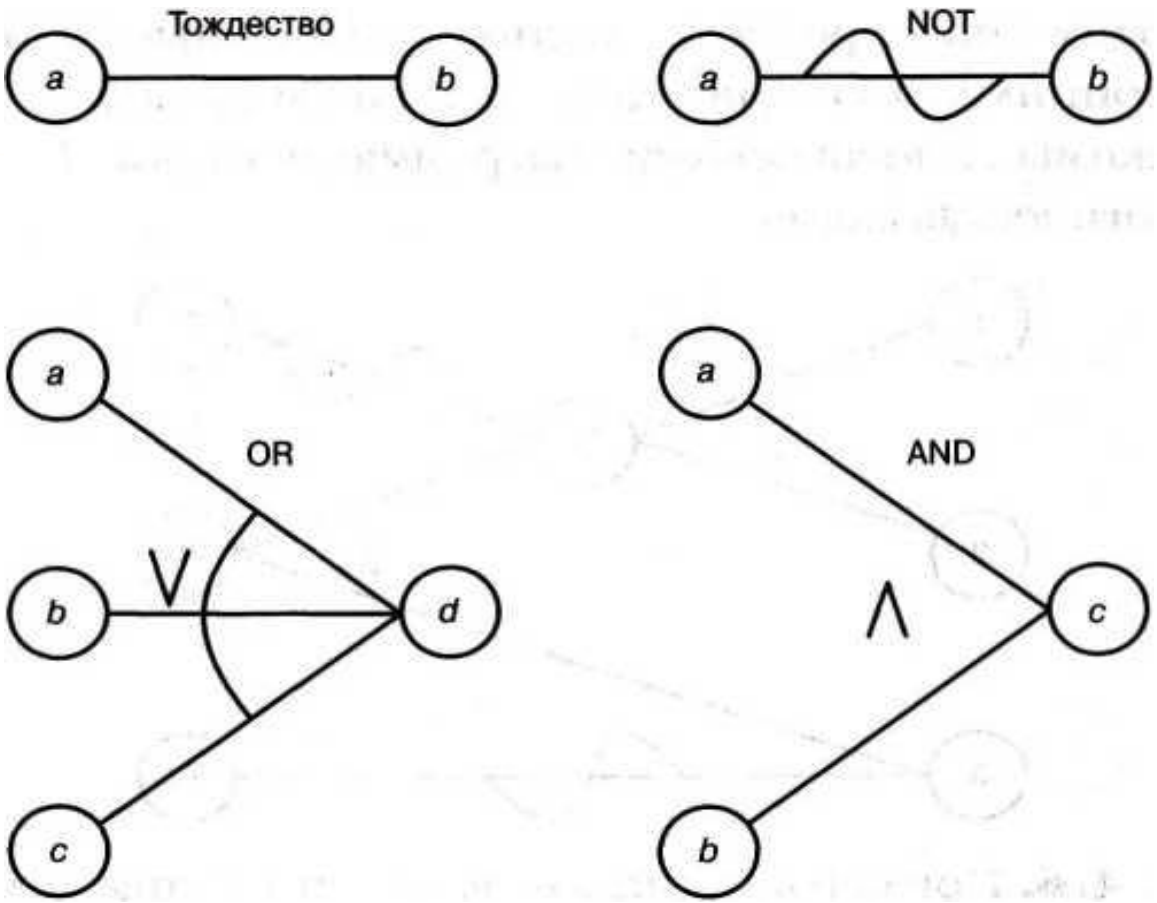
Применение причинно-следственных диаграмм

- Метод *причинно-следственных диаграмм* (cause-and-effect diagrams), или диаграмм Исикавы, помогает систематически выбирать высокорезультативные тесты. Его дополнительным преимуществом является то, что он позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.
- Причинно-следственная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Фактически такая диаграмма является аналогом цифровой логической схемы, но для ее описания используется более простая нотация по

Применение причинно-следственных диаграмм

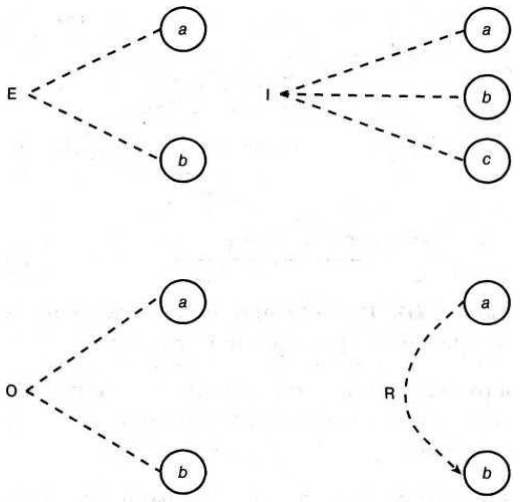
- Базовая нотация причинно-следственных диаграмм содержит узлы и связи. Каждый узел диаграммы может находиться в двух состояниях: 0 или 1, где 0 представляет состояние “отсутствует”, а 1 — “присутствует”.
- Функция *тождество* устанавливает, что если *a* равно 1, то и *b* равно 1; в противном случае *b* равно 0.
- Функция *not* устанавливает, что если *a* равно 1, то *b* равно 0; в противном случае *b* равно 1.
- Функция *or* устанавливает, что если *a,* или *b,* или *c* равно 1, то *d* равно 1; в противном случае *d* равно 0.
- Функция *and* устанавливает, что если *и a, и b* равно 1, то с равно 1; в противном случае с равно 0.

Применение причинно-следственных диаграмм (2)



Применение причинно-следственных диаграмм (3)

- Ограничение E требует, чтобы всегда выполнялось условие, в соответствии с которым только *a* или только *b* может быть равно 1 *(a* и *b* не могут быть равны 1 одновременно, но обе величины могут быть равны 0). Ограничение I требует, что бы по крайней мере одна из величин, *a*, *b* или *c,* была равна *1 (a, b и c не могут* быть равны 0 одновременно). Ограничение O требует, чтобы одна и только одна из величин, *a* или *b,* была равна 1. Ограничение R требует, чтобы *a* было равно 1, только если *b* равно 1 (т.е. *a* не может быть равно 1, если *b* равно 0).



Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 5
Модульное тестирование, инкрементное тестирование, нисходящее, восходящее тестирование

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

Модульное тестирование

Инкрементное тестирование

Нисходящее и восходящее тестирование

Модульное тестирование, инкрементное тестирование, нисходящее, восходящее тестирование

- Модульное тестирование — это процесс тестирования отдельных блоков, подпрограмм, классов или процедур, образующих крупную программу. Цель модульного тестирования — сравнение функций, реализуемых модулем, со спецификациями, описывающими его функциональные или интерфейсные характеристики. Модульное тестирование ориентировано на использование метода “белого ящика”.

Модульное тестирование

- Процесс модульного тестирования зависит от двух важных аспектов: проектирование эффективного набора тестов и технология объединения модулей в работающую программу. Второй аспект очень важен, поскольку от него зависят:
- форма записи модульных тестов;
- типы инструментов тестирования, которые могут быть использованы;
- очередность кодирования и тестирования модулей;
- стоимость генерации тестов;
- стоимость отладки (т.е. локализации и устранения ошибок).
- Известно два подхода к тестированию — *инкрементный* и *неинкрементный.*
- Для инкрементного подхода разработаны две стратегии, используемые как при разработке, так и при тестировании программного обеспечения: стратегию тестирования *сверху вниз,* или *нисходящую стратегию,* и стратегию тестирования *снизу вверх,* или *восходящую стратегию.*

Инкрементное тестирование

Инкрементный процесс добавления очередного модуля к множеству или подмножеству ранее протестированных модулей продолжается до тех пор, пока не будет протестирован последний модуль.

Неинкрементное тестирование более трудоемко.

Программные ошибки, связанные с несоответствием межмодульных интерфейсов или с некорректными предположениями относительно взаимодействия модулей, при использовании инкрементного подхода будут обнаруживаться раньше, поскольку тестирование комбинаций модулей в этом случае начинается на более раннем этапе.

.При инкрементном тестировании отладка программ упрощается..

Результаты инкрементного тестирования могут оказаться более полными.

Использование неинкрементного подхода предоставляет больше возможностей для параллельной организации работы на начальном этапе

Нисходящее и восходящее тестирование

- две стратегии: тестирование *сверху вниз,* или *нисходящее тестирование,* и тестирование *снизу вверх,* или *восходящее тестирование.*

Нисходящее тестирование

- Нисходящее тестирование начинается с верхнего в иерархической структуре, или головного, модуля программы. Какой-либо единственно “правильной” процедуры, регламентирующей последующий выбор каждого очередного модуля, подлежащего инкрементному тестированию, не существует. Руководствоваться следует лишь тем, что подходящим для этих целей будет такой модуль, для которого найдется по крайней мере один вызывающий его модуль, уже прошедший тестирование.

Восходящее тестирование

- Во многих отношениях восходящее тестирование противоположно нисходящему; таким образом, преимущества первого становятся недостатками второго, а недостатки — преимуществами. С учетом этого обсуждение восходящего тестирования будет более кратким.

Восходящее тестирование (2)

- В соответствии с данной стратегией тестирование начинают с терминальных модулей программы (т.е. тех, которые не вызывают другие модули). Аналогично предыдущему случаю, наилучшего универсального рецепта выбора следующего модуля для инкрементного тестирования просто не существует. Единственное, что можно сказать по этому поводу, — выбираемый модуль должен быть таким, чтобы все его подчиненные (вызываемые им) модули предварительно были протестированы.

Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 6
Высокоуровневое тестирование, функциональное, системное, приемочное тестирование

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

- Высокоуровневое тестирование, функциональное, системное, приемочное тестирование
- Модель цикла тестирования
- Высокоуровневые методы тестирования
- Функциональное тестирование
- Системное тестирование
- Приемочное тестирование

Высокоуровневое тестирование, функциональное, системное, приемочное тестирование

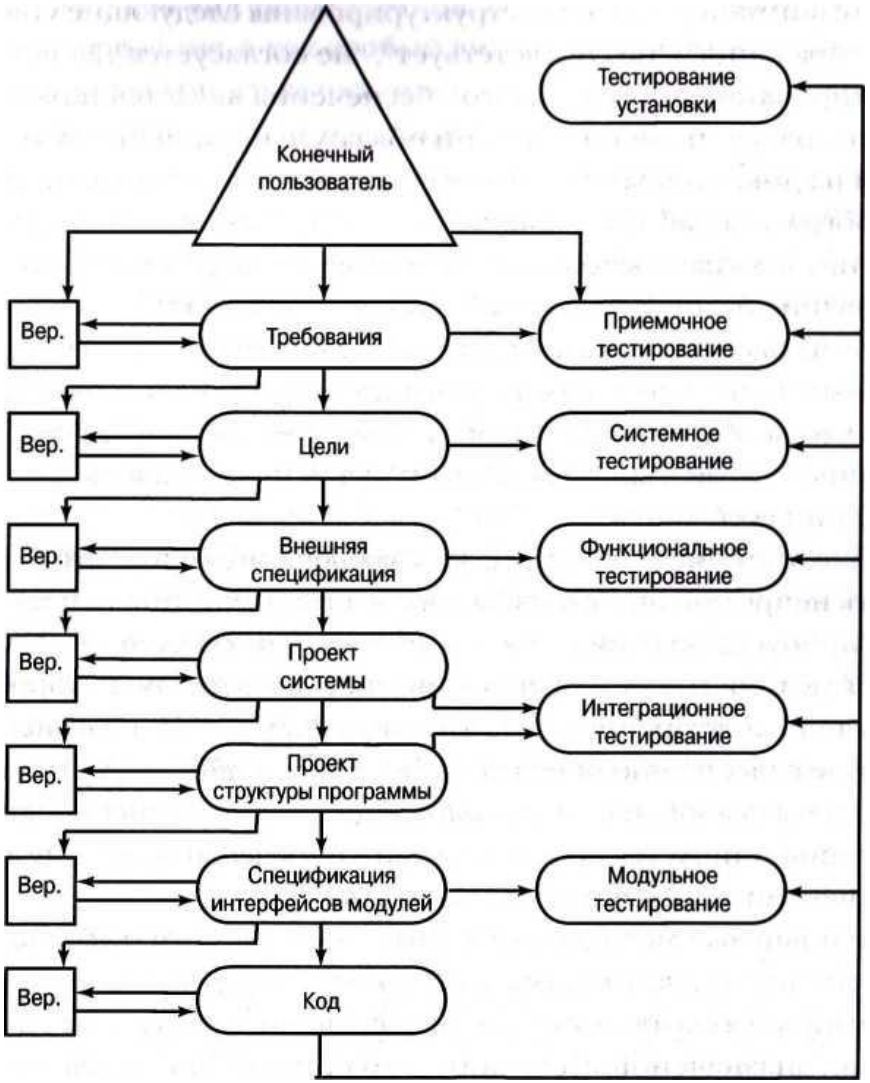
- Разработка программного обеспечения в значительной степени представляет собой процесс передачи информации о том, что собой должна представлять конечная программа, и трансляции этой информации из одной формы в другую, от общих понятий — к конкретике. Поэтому подавляющее большинство программных дефектов обусловлено сбоями, помехами и все-возможными искажениями в процессе передачи и трансляции соответствующей информации.

Модель цикла тестирования

Цикл тестирования структурно встраивается в модель цикла разработки. Другими словами, вы должны быть в состоянии установить взаимно однозначное соответствие между процессами разработки и тестирования, например, так, как показано ниже.

- Цель *модульного теста* — найти расхождения между характеристиками модулей программы и спецификациями их интерфейсов.
- Цель *функционального теста* — продемонстрировать, что программа не соответствует внешним спецификациям.
- Цель *системного теста* — продемонстрировать, что свойства продукта не согласуются с первоначально сформулированными целями.

Модель цикла тестирования (2)



Высокоуровневые методы тестирования

- Высокоуровневые методы тестирования, в наибольшей степени применимы к *программным продуктам* (т.е. программам, написанным в соответствии с контрактными обязательствами, или программам, предназначенным для широкого применения, в отличие от экспериментальных программ или программ, используемых лишь их автором). В случае программ, которые создавались не как продукты, формальные требования и цели зачастую отсутствуют. Для таких программ единственным высокоуровневым тестом может быть лишь функциональный тест. К этому также можно добавить, что с увеличением размера программ необходимость в высокоуровневом тестировании возрастает. Объясняется это тем, что для больших программ отношение числа ошибок проектирования (ошибок, допущенных на ранних стадиях процесса разработки) к числу ошибок кодирования значительно выше, чем для программ небольшого размера.

Функциональное тестирование

- *Функциональное тестирование* (function testing) — это процесс, нацеленный на выявление расхождений между поведением программы и внешней спецификацией. Внешняя спецификация — это точное описание поведения программы с точки зрения конечного пользователя.

Системное тестирование

Системное тестирование решает отдельную задачу: сопоставить результат реализации системы или программы с первоначально сформулированными для нее целями. Отсюда можно сделать два вывода.

1. Системное тестирование не ограничивается только системами. Если продукт является программой, то системное тестирование в данном случае представляет собой попытку продемонстрировать, какие из стоящих перед программой целей не реализуются.
2. Системное тестирование по определению невозможно, если отсутствует документ, отражающий набор измеримых целей, достижение которых возлагается на продукт.

Основное назначение системного теста - сравнение программы с исходным документом, описывающим цели ее создании

Нагрузочное тестирование

- При нагрузочном тестировании (stress testing) программу заставляют работать в режиме интенсивной нагрузки. Этот вид тестирования не следует пугать с тестированием на больших объемах данных. Интенсивная нагрузка возникает в условиях обработки пикового объема данных или выполнения пикового количества операций в течение короткого промежутка времени. Здесь можно провести аналогию с проверкой квалификации секретаря-машинистки. Тестирование на больших объемах позволило бы определить, сможет ли машинистка напечатать многостраничный отчет, тогда как нагрузочное тестирование позволило бы выяснить, способна ли она печатать текст со скоростью 50 слов в минуту.

Приемочное тестирование

- В соответствии с общей моделью разработки, представленной в графиче¬ском виде на слайде 3, приемочное тестирование (acceptance testing) — это процесс сравнения возможностей разработанной программы с исходными требованиями и потребностями конечных пользователей.

Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 7
Тестирование удобства пользователя

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

- Основы тестирования удобства использования
- Процесс тестирования удобства использования
- Выбор типичных пользователей для тестирования

Основы тестирования удобства использования

- Тестирование удобства использования, или *тестирование, ориентированное на пользователя* (user-based testing), является одной из разновидностей тестирования по принципу “черного ящика”
- Анализ человеческого фактора всегда остается предметом в высшей степени субъективным. Перечень контрольных вопросов, с которыми необходимо сверяться при тестировании удобства использования.
- Учитывались ли при разработке пользовательского интерфейса возможные различия в интеллектуальном и образовательном уровне конечных пользователей и предпринимались ли попытки нейтрализовать воздействие возможных неблагоприятных внешних факторов на пользователей в процессе работы?

Основы тестирования удобства использования (2)

- Будут ли выходные данные программы понятны пользователю, достаточно ли внимательно они отобраны для учета его интересов и исключена ли из них ненужная техническая информация?
- Выводится ли диагностическая информация, в том числе сообщения об ошибках, в такой форме, чтобы для ее понимания пользователю не надо было иметь специальное образование?

Основы тестирования удобства использования (3)

- Обладает ли полный набор пользовательских интерфейсов достаточной концептуальной целостностью, внутренней согласованностью, а также однородностью синтаксиса, соглашений, семантики, форматов, стилей и сокращений?
- Предусмотрена ли достаточная избыточность входных данных? Например, для проверки того, что доступ к банковскому счету осуществляется авторизованным клиентом, а не посторонним лицом, такая система должна запрашивать номер счета, а также имя и персональный идентификационный номер (PIN) владельца счета.

Основы тестирования удобства использования (4)

- Проста ли программа в использовании? Если, например, ввод чувствителен к регистру, то навещается ли пользователь об этом? Если выполнение некоторых операций требует навигации по многим пунктам меню или опциям, то достаточно ли понятно в любой ситуации, как вернуться к главному меню? Может ли пользователь легко переместиться на одну ступень вверх или вниз в системе меню?
- Продуман ли интерфейс программы таким образом, чтобы вероятность возникновения ошибок в результате неправильных действий пользователя была сведена к минимуму? Доставляли ли эти ошибки лишь некоторое неудобство (пользователь был в состоянии сам их исправить), или же некорректные действия либо выбор пользователя приводили к прекращению работы приложения?

Основы тестирования удобства использования (5)

- Легко ли пользователю повторить ранее выполненные действия в более поздних сеансах работы с приложением? Способствует ли дизайн приложения самостоятельному обучению пользователя эффективным методам работы с системой?
- Чувствует ли пользователь себя уверенно при навигации по различным пунктам меню?
- Достигает ли приложение тех целей, которые предусматривались проектом? Соответствуют ли фактические функциональные возможности программы исходным спецификациям. Работает ли приложение так, как того требуют спецификации?

Процесс тестирования удобства использования

- Любое тестирование удобства использования должно начинаться с планирования. Следует подготовить набор практических, связанных с реальной деятельностью, повторяющихся тестовых заданий, которые должен будет выполнить каждый пользователь. Проектируйте эти тестовые сценарии так, чтобы в процессе их выполнения пользователь столкнулся со всеми аспектами эксплуатации программного обеспечения, знакомясь с ними в каком-то определенном или же случайном порядке.

Выбор типичных пользователей для тестирования

- Полный протокол тестирования удобства использования предполагает многократное повторное выполнение тестов одними и теми же пользователями, а также выполнение набора тестов группой пользователей. Зачем нужны тесты первой из указанных категорий? Одна из характеристик, которые мы хотим протестировать, — поддержка факторов, способствующих *приобретению пользователем навыков работы с программой.* Если говорить конкретно, то нас интересует, как много из того, что пользователь узнал о работе программного обеспечения на протяжении одного рабочего сеанса, он сможет вспомнить во время следующего сеанса.

Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 8

Отладка, планирование и контроль тестирования, критерии завершения тестирования, приемочное тестирование

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

- Отладка
- Методы «грубой силы»
- Индуктивная отладка
- Обратная трассировка
- Дедуктивная отладка
- Принципы устранения ошибок
- Анализ ошибок

Отладка

- отладка — это тот вид работ, который выполняется после проведения успешного теста.
- Отладка является необходимой и неотъемлемой частью процесса тестирования.

Методы "грубой силы"

- Наиболее распространенной схемой отладки программ является метод “грубой силы”. Причина его популярности заключается в том, что он почти не требует размышлений и умственного напряжения. К сожалению, он неэффективен и редко приводит к успеху.
- Методы “грубой силы” можно разделить на следующие категории:
- отладка с использованием дампа памяти;
- отладка в соответствии с рекомендацией “расставить инструкции вывода на печать по всей программе”;
- отладка с использованием инструментальных средств

Методы "грубой силы« (2)

Отладка посредством вывода дампа памяти наименее эффективна, так как:

- Трудно устанавливать соответствие между ячейками памяти и переменными в исходной программе.
- Дамп памяти любой достаточно сложной программы содержит огромное количество данных, многие из которых совершенно несущественны для отладки.
- Дамп памяти дает статическую картину программы, отражая ее состояние в какой-то один момент времени, тогда как для нахождения ошибок требуется изучать динамику программы (изменение ее состояния во времени).

Методы "грубой силы« (3)

Отладка посредством вывода дампа памяти наименее эффективна, так как:

- Время получения дампа памяти редко совпадает с моментом наступления ошибки, и потому он не отражает состояние программы, в котором она находилась в интересующий вас момент. Действия, осуществляемые программой между моментом наступления ошибки и моментом получения дампа, могут маскировать важные подсказки, облегчающие поиск ошибки.
- Подходящих методологий нахождения ошибок путем анализа дампа памяти не существует (множество программистов просто тоскливо вглядываются в дамп, надеясь на то, что ошибка сама проявится в нем каким-то чудесным образом).

Методы "грубой силы« (4)

- Расстановка инструкций печати по всем участкам сбойной программы. Этот метод лучше предыдущего, поскольку он отображает динамику программы и дает возможность исследовать информацию, которую легче соотносить с исходной программой, но и него имеются недостатки.
- Вместо того чтобы размышлять о возможных причинах проблемы программист действует методом проб и ошибок.
- Данный метод вынуждает анализировать огромные объемы данных.
- Он требует внесения изменений в программу, а эти изменения могут скрыть ошибку, нарушить критические временные соотношения или внести в программу новые ошибки.

Методы "грубой силы« (5)

- У всех описанных методов “грубой силы” имеется один общий недостаток: они не заставляют *думать*

Индуктивная отладка

- Одним из методов отладки, требующих в первую очередь рассуждений, является *индуктивная отладка* (inductive debugging), от частных наблюдений к общим выводам. Такая отладка начинается с рассмотрения ряда ключевых факторов (симптомов ошибки и, возможно, результатов одного или нескольких тестов) с последующим переходом к установлению взаимосвязей между этими факторами.

Обратная трассировка

- Эффективным методом локализации ошибок в небольших программах является метод *обратной трассировки* (backtracking) — прослеживание появления некорректных результатов в обратном направлении до той точки, в которой происходит сбой в работе логики программы. Отладка начинается в точке программы, в которой программа выдает неправильный результат (например, печатает некорректные данные). При этом, исходя из наблюдаемого вывода, устанавливают, какими должны быть значения переменных в данной точке. Мысленно выполнив программу в обратном порядке и повторно применив логику “если, то”, т.е. используя рассуждения типа “если в этой точке состояние программы было таким-то, то в предыдущей точке оно должно быть таким-то”, можно достаточно быстро определить точное местонахождение ошибки. Это место будет находиться где-то между точкой, в которой состояние программы совпадает с ожидаемым, и первой точкой, в которой это соответствие нарушается.

Дедуктивная отладка

- *Дедуктивная отладка* (deductive debugging) начинается с анализа общих теоретических положений или допущений и через цепочку логических умозаключений путем постепенного исключения и уточнения фактов приводит к конкретному решению (о местонахождении ошибки)

Принципы устранения ошибок

- Где есть одна ошибка, там может быть и вторая
- Устранив ошибку, обязательно проверьте, нет ли чего-либо подозрительного поблизости.
- Устраняйте ошибки, а не их признаки
- Если предполагаемое исправление не согласуется со всеми признаками ошибки, то, возможно, вы устраняете ошибку лишь частично
- **Никогда нельзя быть полностью уверенным в правильности вносимых исправлений**
- Никогда не исходите из того, что код, добавляемый в программу для устранения ошибки, корректен. Код исправлений намного более подвержен ошибкам, чем первоначальный код самой программы. Как следствие этого, код исправлений подлежит обязательному тестированию, причем, вероятно, даже более строгому, чем код самой программы.

Принципы устранения ошибок (2)

- ***Вероятность*** корректности вносимых исправлений ***снижается с*** увеличением размера программы

- Отношение количества ошибок, вызванных некорректными исправлениями, к количеству первоначально содержащихся в программе ошибок, в крупных программах возрастает. Вы всегда должны быть уверены в том, что исправляете саму ошибку, а не ее симптомы.

- ***Остерегайтесь*** внесения новых ***ошибок в ходе*** исправлений
- Необходимо следить не только за правильностью кода вносимых исправлений, но и за тем, чтобы внесение внешне уместного исправления не вызвало нежелательных побочных эффектов и не способствовало появлению новых ошибок.

Принципы устранения ошибок (3)

- Процесс устранения ошибок должен временно возвращать вас на этап проектирования

- Вдвойне важно инспектировать код и после устранения содержа-щихся в нем ошибок.
- Изменять следует исходный код, а не объектный

- При отладке крупных систем, особенно тех, которые написаны на языке ассемблера, иногда пытаются исправить ошибку, изменяя непосредственно объектный код в расчете на то, что изменения в исходный код будут внесены позднее. Такой подход порождает две проблемы: 1) обычно это указывает на то, что используется “отладка экспериментами”, и 2) это приводит к нару¬шению соответствия между объектным и исходным кодом, а значит, ошибка может вновь легко всплыть на поверхность в результате повторной компиля¬ции или ассемблирования кода.

Анализ ошибок

- *Где была допущена ошибка?*.
- *Кто допустил ошибку?*
- *Какова причина ошибки?*
- *Как можно было предотвратить ошибку?* т.
- *Почему ошибка не была обнаружена раньше?*
- *Каким образом ошибку можно было обнаружить раньше?*
- дополнительных успешных тестов для этой или будущих программ?
- Проведение такого анализа требует дополнительных затрат времени и денег, но получение ответов на перечисленные выше вопросы предоставит вам ценную информацию, которую можно будет с пользой применить в будущих проектах.

Вопросы?

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт прикладной математики и механики

Телематика

М.А.Курочкин

Лекция 9
Основные положения стандарта *ISO/IEC 12207*

Тестирование ПО

Слайды видеолекций
для магистров технических направлений

Санкт-Петербургский государственный политехнический университет
2014

План занятий

Стандарты тестирования
Тестирование верхнего уровня
Тестирование нижнего уровня
Стандарт ГОСТ Р ИСО/МЭК 12207-2010

Стандарты тестирования

- Стандарты тестирования зависят от того в какой области применяется разрабатываемое ПО.

  Стандарт ISO 9001
  ISO 9001 - стандарт, основанный на принципах контроля качества. В нём, по существу, задаются ключевые функциональные требования, для каждого из которых нужно сказать, что делается, как сделать то, что сказано, и иметь возможность показать, что было сделано. Реализация данного стандарта в среде ПО - ISO 9000-3.

  Стандарт ISO/IEC 12207 и IEEE/EIA 12207
  ISO/IEC 12207 - это международный стандарт, описывающий структуру процессов жизненного цикла ПО от концепции до изъятия из обращения. Стандарт IEEE/EIA 12207 - адаптация ISO/IEC 12207 для США.

Стандарты тестирования(2)

- В соответствии с этими стандартами в конкретной отрасли производства выдвигаются требования к тестированию ПО. Например в авиации США на основе ISO/IEC 12207 был выработан стандарт RTCA(Requirements and Technical Concepts for Aviation). В нём перечислены требования к тестированию верхнего и нижнего уровня:

Тестирование верхнего уровня

- - Требования высокого уровня должны включать в себя системные требования к ПО,
  - Требования высокого уровня должны формулироваться с учётом архитектуры ПО,
  - Программный код должен удовлетворять архитектуре ПО и требованиям низкого уровня ,
  - Откомпилированный и готовый к использованию код должен удовлетворять требованиям к ПО,
  - Используемые значения должны технически соответствовать поставленным целям и выполнять их для всех уровней ПО

Тестирование нижнего уровня

- Проверку (Verification) требований нижнего уровня,
- Проверку архитектуры программного обеспечения (ПО),
- Проверку логического покрытия для всех функций написанных в ПО,
- Контроль процедур тестирования,
- Независимость ПО от тестирования. Т.е. ПО не должно перестраиваться особым образом под тесты,
- Тестирование должно несколько раз покрывать исходный код, для обнаружения определённого класса ошибок,
- Робастное тестирование,
- Тестирование на предмет косвенного обнаружения ошибок. Например: соответствие стандартам разработки ПО.

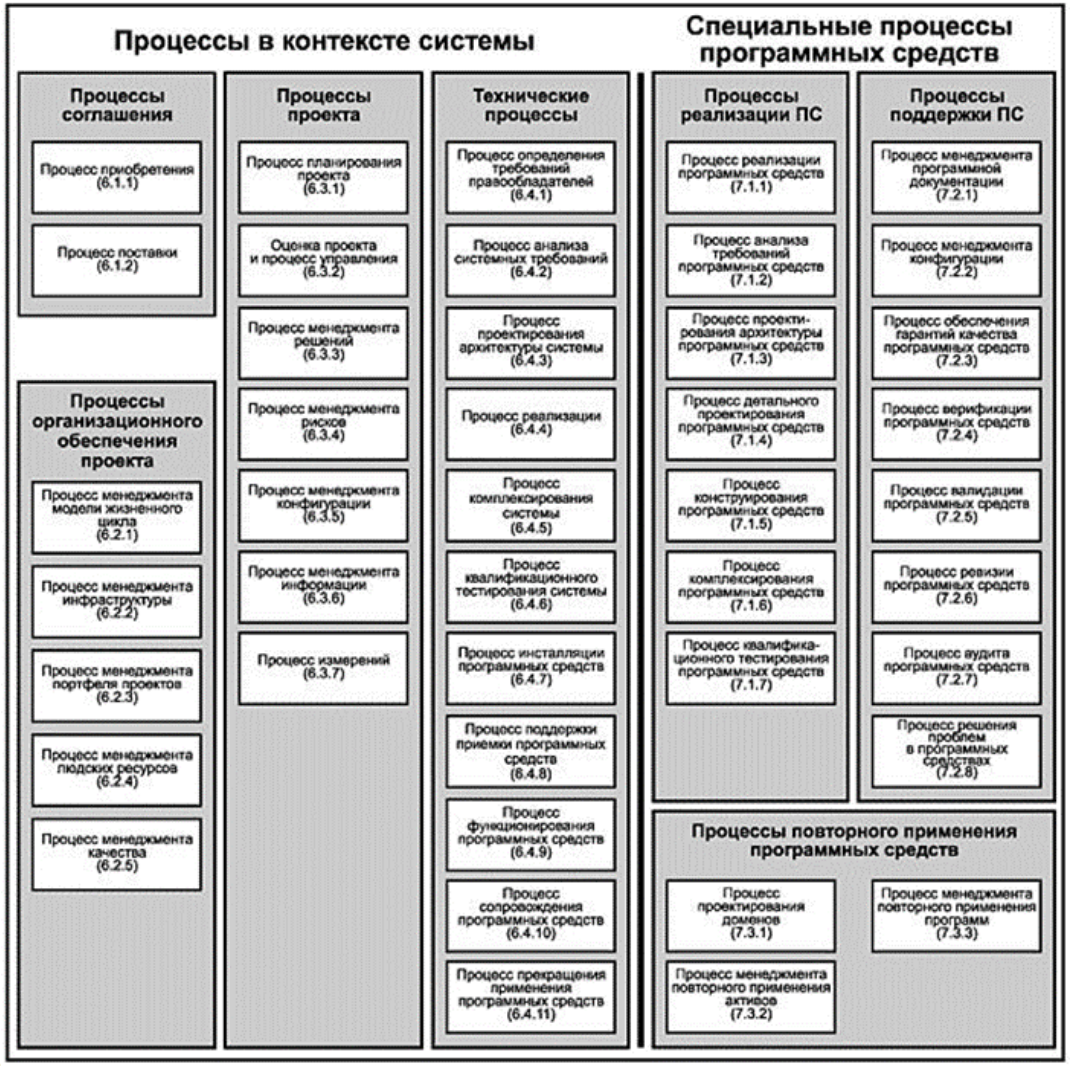
Стандарт ГОСТ Р ИСО/МЭК 12207-2010

- НАЦИОНАЛЬНЫЙ СТАНДАРТ РОССИЙСКОЙ ФЕДЕРАЦИИ Информационная технология Системная и программная инженерия
- ПРОЦЕССЫ ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНЫХ СРЕДСТВ
- Information technology. System and software engineering. Software life cycle processes

Стандарт ГОСТ Р ИСО/МЭК 12207-2010 (2)

- Настоящий стандарт, используя устоявшуюся терминологию, устанавливает общую структуру процессов жизненного цикла программных средств, на которую можно ориентироваться в программной индустрии. Настоящий стандарт определяет процессы, виды деятельности и задачи, которые используются при приобретении программного продукта или услуги, а также при поставке, разработке, применении по назначению, сопровождении и прекращении применения программных продуктов. Понятие программного средства включает в себя встроенный фирменный программный компонент.

Стандарт ГОСТ Р ИСО/МЭК 12207-2010 (3)



Стандарт ГОСТ Р ИСО/МЭК 12207-2010 (4)

- **Группы процессов жизненного цикла**
- Группы процессов жизненного цикла включают в себя:
- процессы соглашения — 2;
- процессы организационного обеспечения проекта — 5;
- процессы проекта — 7;
- технические процессы — 11;
- процессы реализации программных средств — 7;
- процессы поддержки программных средств — 8;
- процессы повторного применения программных средств — 3.
- Процессы соглашения
- Поставка
- Приобретение

Стандарт ГОСТ Р ИСО/МЭК 12207-2010 (5)

- **Группы процессов жизненного цикла**
-
- Процессы организационного обеспечения проекта
- Процесс менеджмента модели жизненного цикла;
- Процесс менеджмента инфраструктуры;
- Процесс менеджмента портфеля проектов;
- Процесс менеджмента людских ресурсов;
- Процесс менеджмента качества.
- Процессы проекта
- Процессы менеджмента проекта
  - процесс планирования проекта;
  - процесс управления и оценки проекта.
- Процессы поддержки проекта
  - процесс менеджмента решений;
  - процесс менеджмента рисков;
  - процесс менеджмента конфигурации;
  - процесс менеджмента информации;
  - процесс измерений.

Стандарт ГОСТ Р ИСО/МЭК 12207-2010 (6)

Цель процесса комплексирования системы заключается в объединении системных элементов (включая составные части технических и программных средств, ручные операции и другие системы, при необходимости) для производства полной системы, которая будет удовлетворять системному проекту и ожиданиям заказчика, выраженным в системных требованиях.

Выходы

В результате успешного осуществления процесса комплексирования системы:
a) определяется стратегия комплексирования системы в соответствии с приоритетами системных требований;
b) разрабатываются критерии для верификации соответствия с системными требованиями, распределенными по элементам системы, включая интерфейсы между ними;
c) верифицируется комплексированная система с применением определенных критериев;
d) разрабатывается и применяется стратегия регрессии для повторного тестирования системы в случае, если выполняются изменения;
e) устанавливается согласованность и прослеживаемость между системным проектом и интегрированными элементами системы;
f) конструируется комплексированная система, демонстрирующая соответствие с системным проектом;
g) конструируется комплексированная система, демонстрирующая существование полной совокупности пригодных для применения поставляемых системных элементов.

Вопросы?