Краткий материал к устному зачету по курсу “Технологии Программирования”

Лекция №1

Компиляция

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера). Входной информацией для компилятора (исходный код) является описание алгоритма или программа на предметно-ориентированном языке (ЯП, специализированный для к,онкретной области применения), а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

Машинный код (платформенно-ориентированный код), машинный язык — система команд (набор кодов операций) конкретной вычислительной машины, которая интерпретируется непосредственно процессором или микропрограммами этой вычислительной машины.

Этапы компиляции:

- Лексический анализ («токенизация», от англ. *tokenizing*) — процесс аналитического разбора входной последовательности символов на распознанные группы — *лексемы*, с целью получения *токенов* (объект, создающийся из лексемы в процессе лексического анализа). (Лексема - это последовательность символов в исходной программе, которая соответствует шаблону для токена и идентифицируется лексическим анализатор как экземпляр этого токена.)
- Синтаксический анализ — последовательность лексем преобразуется в дерево разбора.
- Семантический анализ — дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
- Оптимизация — выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
- Генерация кода — из промежуточного представления порождается объектный код.

Если рассматривать конкретно C/C++, то компиляция исходных текстов на C/C++ в исполняемый файл происходит в три этапа:

1. препроцессинг (обработка директив препроцессора);
2. компиляция;
3. линковка (связывания воедино всех объектных файлов проекта).

Лекция №2

Этапы отладки

Отладка состоит из следующих этапов:

- воспроизведение дефекта (любым из доступных способов);
- анализ дефекта (поиск причины возникновения дефекта – root-cause);
- дизайн исправления дефекта (и возможно ревью, если есть альтернативы);
- исправление дефекта;
- валидация исправления;
- интеграция исправления в кодовую базу или целевую систему;
- дополнительные валидации после интеграции (если необходимо).

Воспроизведение дефекта — найти версию в которой появился дефект, попытаться воспроизвести дефект при соответствующем сценарии (настройки, входные данные, сценарий при котором дефект был найден и тд).

Поиск причины возникновения дефекта (root-cause) – найти “корень зла” в проекте, а не пытаться заткнуть все побочные дефекты порожденные основным; также определить условия его возникновения; узнать область повреждения; определить кто привнес и в какую версию.

Дизайн исправления дефекта — предложить каким методом исправить найденную ошибку, дизайн зависит от того какой именно дефект:

- технический – ”лажи в коде”;
- архитектурный – “не было предусмотрено, что при некотором сценарии может потребоваться сделать что-то”;
- технологический – “использование неподходящих технологий, пример: использовать http для постоянной связи клиент-сервер вместо websockets”

Исправление дефекта:

- не внести новые дефекты;
- исправление не должно менять логику и поведение;
- не стоит использовать “грязные” хаки;

- документирование — подробное описание исправление.

Валидация исправления — удостовериться, что ошибка была исправлена, а также само исправление не добавило другие ошибки, полноценное тестирование, проверить связанные сценарии (рассмотреть похожие сценарии), проверить сценарии root-cause..

Интеграция в целевую систему — проверить на целевой системы работоспособность исправленного кода, подробнее:

- влить в ствол с учетом возможного исправления конфликтов (т.к. проект не стоит на месте);
- проверить сборку и работоспособность;
- деплоинг в целевую систему;
- обновление пользовательских приложений;
- обновление документации.

Дополнительные валидации (проверяем систему как пользователи): проверяем еще раз сценарий ошибки, смотрим связанные сценарии.

Стандартные техники отладки

1. Запуск в отладчике:
    - софтверный — “фигачить в песочнице, как мы обычно делаем на ООП”(основной плюс - возможность делать шаги назад при исследовании ошибки);
    - “железный” — запуск на урезанной системе, типо ардуино и тд и тп, где нет возможности полноценного запуска среды”;
    - удалённый дебаггер — “дебаг на удалённом компьютере”.
2. Логирование:
    - логирование работы системы;
    - логирование работы программного кода.
3. Анализ кода без запуска — метод пристального взгляда, использование статических анализаторов;
4. Анализ поведения системы — предыдущий пункт, но в профиль:
    - упрощение сценария — упростить сценарий возникновения;
    - ограничение объема данных;
    - упрощение данных/запроса;

5. Unit-тестирование — проверка корректности одного или нескольких программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки;

   Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже протестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

6. Прототипирование — “отдельно вынести модуль из системы и протестировать его“;
7. Отладка с помощью дампов — раскрутить дамп памяти (сделать снимок памяти и рассмотреть абсолютно все значения переменных, что было когда вызванным и тд и тп);
8. Отладка с помощью перехватов — рассматривать какие функции вызываются в какой момент времени какими параметрами используя hooking или spiders;🏆
9. Профилирование кода — какую долю из суммарного времени выполнения занял определённый блок кода, необходимость: найти блоки, где рационально улучшать время выполнения (особенно в распаралеленых программах);
10. Выполнение кода в другой среде — если нет удобных средств в песочнице или наоборот залить в песочницу, где удобно отлаживать и развертывание будет куда быстрее;
11. Отладка метода RPC (remote procedure ;call) — возможность вызвать любую функцию/процедуру с внешней системы, передав параметры, и в ответ получить результат выполнения этой процедуры
12. Отладка путем анализа документации, проектных документов;
13. Отладка трансляцией кода:
    - “трансляция вниз” — написать на высокоуровневом языке и транслировать в ассемблерный (к примеру) и взглянуть что под капотом;
    - “трансляция вверх” — наоборот к вышеуказанному (дизассемблировать, к примеру);
14. Отладка разработкой интерпретатора — “разработать скриптовый язык, который будет интерпретировать пользовательские сценарии в системные запросы”;
15. Метод индукции — “пройтись от момента возникновения ошибки до момента когда что-то пошло не так” (от частного к общему);

16. Метод дедукции — исправить общую ошибку и проверить, что при частных исходах всё исправлено и корректно работает (от общего к частному);
17. Обратное движение по алгоритму — фактически метод индукции (мы так в основном делаем).

Лекция №3

Статический анализ кода умеет:

1. Выявление ошибок в коде
2. Рекомендации по оформлению кода
3. Подсчет различных метрик исходного кода

Преимущества:

1. Полное покрытие кода(проверит все, в отличие от человека)
2. Не зависит от используемого компилятора и среды разработки
3. Легко и быстро обнаруживает опечатки

Слабости:

1. Трудности в выявлении ошибок в параллельном программировании и утечек памяти
2. Ложно-положительные срабатывания(рассчитан на стандартный подход, в сложных (define) не всегда корректен)

Принципы хорошей архитектуры:

1. Эффективность (насколько работоспособна при разных сценариях):
    - Безопасность (защищенность от атак, утечки данных (пароли сохраняет в открытом доступе))
    - Надежность
    - Производительность (время, количество параллельно обрабатываемых запросов)
    - Масштабируемость (Насколько доступно ее укрупнение в плане количества пользователей, запросов, использования дополнительной техники и др)
2. Гибкость системы
    - Изменения текущего функционала
    - Исправление ошибок (должны быть отделимыми, ошибка в одной строке не должна ломать систему)
    - Настройка системы(под пользователя, под разные задачи
3. Расширяемость
    - Возможность добавлять новые сущности и функции(под сущностями стоит понимать поддержку новых объектов, алгоритмов обработки и тд. К примеру, вы хотите добавить функцию печати на испанском)
    - “Внесение наиболее вероятных изменений должно требовать наименьших усилий” - совет
4. Масштабируемость процесса разработки (Можно ли распараллелить между разными людьми)
5. Тестируемость (Отдельно тестировать разные модули)
6. Возможность повторного использования кода (Например кусок кода, где файл открывается) - планируется в процессе разработки каждой части

7. Сопровождаемость

Критерии плохой архитектуры:

1. Жесткость (Тяжело изменить, тяжело добавить новый функционал)
2. Хрупкость (Изменения нарушают другие модули) Система из костылей падает!
3. Неподвижность (Тяжело извлечь модуль наружу) - анти 6 пункт из хорошей системы

Руководство:

1. High Cohesion - Высокая сопряженность внутри модулю, модуль сфокусирован на 1 задаче
2. Low Coupling - Слабая связь между модулями. Модулю должны быть независимы друг от друга, либо слабо связаны. Не должно быть френдовых классов в разных модулях

Закон Деметры:

- Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B, у которого есть доступ к объекту C. - Принцип минимального знания

Принципы SOLID:

- The single Responsibility Principle - Класс должен иметь ответственность и эта ответственность должна быть инкапсулирована в класс. Изменяется тот файл, на части работы которого мы хотим изменить систему
- The open Closed Principle - Изменения должны быть допустимы путем внесения новых сущностей, но внесение их не должно вести к изменению кода, который эти сущности используют
- The Liskov Substitution Principle - Вместо объектов класс A должно быть возможно использование объектов класса B, если B наследник A. Пример - lxml и lxml2, если взять lxml2, имеющий новые теги, система не должна сломаться. Система, работающая с xml не должна знать о тегах
- The interface segregation Principle - Много интерфейсов, специально предназначенных для клиентов лучше, чем один интерфейс общего назначения. Пример: картинки, видео и фото - первое не заблочить
- The dependency inversion Principle(Модули верхних не должны зависеть от модулей нижних. Абстракции не должны зависеть от деталей - детали должны зависеть от абстракций. Пример: есть 2 типа файлов (A и B) и 2 интерфейса, нельзя в интерфейсах прописывать условия на конкретный тип файла (if A {...} else if B {...}).

Лекция №4

Этапы проектирования

Формирование требований

Разработка концепции

Техническое задание

Эскизный проект

Технический проект

Рабочая документация

Постановка/ввод в действие

Сопровождение

1. Формирование требований

Обследование объекта (объект - предметная область, должны понимать как это должно работать, на какую целевую аудиторию рассчитано)

Обоснование необходимости создания(какую проблему пользователей вы хотите решить? Еще и когда! Вдруг потом это будет не актуально)

Формирование требований пользователей. (анализируя целевую аудиторию, сформировать полные, общие требования, которые нужны им)

Подготовка отчетности по этапу (точно зафиксировать требования пользователя, так как потом делиться этим с разработчикам, основные функции)

2. Разработка концепции

Изучение объекта автоматизации (осознать инфраструктуру, с которой вы работаете. Чтобы пользователям было удобно.(пример о том, что мобильная сеть не выдержит 4к видео, а вы сделали “ютуб” только на 4к)

Проведение необходимых НИР(провести сначала научно-исследовательскую работу (пример про отрисовку солнечного света при дожде))

Разработка вариантов концепции системы (приложение или вебсайт)

Подготовка отчетности по этапу (зафиксировать результат)

Выбор формата поставки(клиент-серверное ПО или еще как-то. Предусмотреть механизм защиты, разные компоненты могут иметь разные компоненты защиты)

Целевое оборудование (смотря под что вы пишете (например, под ардуино)*

Построение высокоуровневой архитектуры системы (понимать организационный подход. Если несколько компонент, можно разделить на разных людей и тп)

Выбор/разработка новых технологий/алгоритмов

3. Техническое задание

(Нужно, например, если вы передаете задание какой-то внешней команде. Описать логику взаимодействия)

Описание системы

Описание функциональности

Описание требований (На уровне разработчика. Такая-то скорость, такое-то количество оперативки, такое-то время и тп.)

Описание сценариев использования (

Условия сдачи (считать, что не сделали работу, пока не выполнены эти условия)

4. Эскизный проект

Разработка прототипов частей системы (по какому-то критерию упрощают, чтобы посмотреть, как оно получается(обычно по качеству кода))

Тестирование

Оценка качества и производительности (про все альфа, бета релизы)

Изменение прототипов(если что-то не так, улучшать этот момент)

Подготовка документации(история развития системы, может понадобится, чтобы переиспользовать систему (как база)

Часто это просто MVP минимальный жизнеспособный продукт

(Иногда это система с базовым функционалом

Иногда – система с урезанными контентом

Иногда – менее производительная. Супер заинтересованные пользователи помогут тестировать, а потом еще и друзьям расскажут)

5. Технический проект

Разработка частей системы

Разработка документации

Разработка заданий на проектирование смежных частей (чтобы к системе можно было делать дополнительный функционал. То есть как пристроится к вашей системе.)

Тестирование

Оценка качества и производительности

6. *Рабочая документация*

Сценарии использования (что сделать, чтобы что-то произошло)

Описание логики работы

Описание производительности (значимая информация, для разработчика полезно)

Примеры использования(“Как убрать красные глаза ” - открыть цветокоррекцию..)

Обучающие материалы

7. Поставка / ввод в действие

Подготовка объекта автоматизации (настроить его, а не просто отдать код)

Подготовка персонала(рассказать все сотрудникам как пользоваться)

Комплектация системы поставляемыми изделиями(зависит от требований пользователя и производительности ПО)

Проведение предварительных испытаний(провести тестирование производительности, функциональное тестирование

Опытная эксплуатация (сначала ввод туда, где небольшая ответсвенность, если вдруг программа сломается. То есть с минимальными риски проверить полный функционал)

Приемочные испытания (независимый человек должен посмотреть на работоспособность, попробовать)

8. Сопровождение

- Гарантийные обязательства (срок на поиск и обнаружения ошибок, которые вы обязаны исправить. В архитектуру сразу нужно заложить, как отследить ошибки. (логирование!))
- Послегарантийное обслуживание (предусмотреть механизм перевода на новую версию, чтобы ничего не потерялось. Исправление ошибок, "платностью" отличается от гарантийного)

Методологии разработки

Выбор зависит от многих факторов: Специфика проекта, Система финансирования, Вариант поставки(как лицензию, как софт, как сайт и тп)

Модель водопада (waterfall *каскадная модель*)

- Полное прохождение стадий друг за другом
- Следующая стадия не начинается, пока не закончена предыдущая

Четкие сроки, стоимость, результат

Минусы: не можете, когда непонятно, что будет в конце. То есть когда требования меняются в процессе разработки.

Тестирование в конце.

V - модель

сразу две ветки: прохождения каждого этапа по одной ветке проверено по другой. На каждом этапе есть контроль.

Верификация

1. Требования бизнеса / 1. концепция
2. Функциональные требования / 2. высокоуровневая архитектура
3. Архитектура / 3. детальное проектирование
4. Реализация

Валидация

1. Приемо-сдаточное тестирование / 1. поставка и поддержка
2. Функциональное тестирование / 2. тестирование и проверка
3. Интеграционное тестирование / 2. тестирование и проверка
4. Модульное тестирование / 3.

Инкрементная модель

- Первая версия: базовый функционал
- Далее добавляются дополнительные возможности

На каждом этапе производится:

- Определение требований(что хотим)
- Проектирование(как сделаем)
- Кодирование
- Внедрение
- Тестирование

Итерационная модель

Каждый этап – база для определения дальнейших требований (то следующее зависит от предыдущих)

Ключевой момент – каждая новая версия полностью работоспособна

Требования к следующей версии составляются на основе анализа использования текущей

// Пример для понимания различий:
- В инкрементной модели: сначала сажаем дерево, а потом вокруг него постепенно начинаем сажать разные цветы.
- В итерационной: строим башню из кубиков постепенно, после добавления нового кубика и увеличения высоты башни смотрим, не шатается ли она и что нужно сделать для ее устойчивости; если все нормуль, ставим следующий кубик и т.д.

Спиральная модель

может существовать отдельно, а может в рамках других как подэтап

Этапы:

- Планирование
- *Анализ рисков(сколько потратим денег, ресурсов, времени, какова вероятность, что все это провалится)*
- Конструирование
- Оценка результатов

RAD Model

Rapid Application Development Model

цель - быстро разработать ПО

- Различные модули разрабатываются различными командами
- Жестко ограниченное время
- Интеграция отдельных модулей в один
- *(!)Использование инструментов автоматической сборки и генерации кода(можно из нарисованной схемы архитектуры получать готовый код)

Этапы:

- Бизнес-моделирование(понимание задач)
- Анализ и создание модели данных(как они взаимодействуют)
- Анализ и создание модели процесса(как обрабатывать, передавать)
- Автоматическая сборка приложения
- Тестирование

Agile

Гибкая методология разработки

Процесс разработки разделен на итерации

Результат можно оценивать после каждой итерации

Scrum

Краткие ежедневные встречи

- Оценить текущие результаты
- Обсудить проблемы
- Принять тактические решения(о мелких проблемах в функциях и тп.)

и Sprint

Периодические встречи

- Направление развития / стратегические решения
- Определение и распределение задач на следующий спринт
- Обсуждение итогов предыдущего спринта(помогает грамотно оценить сроки, что показывать заказчику)

Лекция №5

Антипаттерны

1. В объектно-ориентированном программировании
2. В кодировании
3. Методологические
4. Управление конфигурацией
5. Прочее

В объектно-ориентированном программировании

- **Базовый класс-утилита** - наследование функциональности из класса-утилиты вместо делегирования(передачи) ему. Классы-утилиты - это своеобразные помощники, выполняющие некоторую вспомогательную функциональность. Наследовать от него один из основных (не вспомогательных) классов - не самая лучшая идея.
- **Anemic Domain Problem (Анемическая модель домена)** - боязнь размещать логику в объектах предметной области. Смысл ООП, чтобы разделить логику между классам, эта модель что-то типа, процедурного программирования. Таким образом не может быть гарантирована корректность такой модели в любое время, так как вся логика находится где-то снаружи.
- **Вызов предка** - для реализации функциональности методу потомка приходится вызывать те же методы у родителя. Если в потомке мы хотим расширить метод предка, то все хорошо, но если мы хотим заменить метод предка и для этого приходится вызывать его, то все плохо. Это нарушает логику программы так, как для работы этого блока кода необходимо знать что делает методы у родителя.
- **Ошибка пустого подкласса** - когда класс обладает различным поведением по сравнению с классом, который наследуется от него без изменений.
- **Божественный объект** - концентрация функциональности в одном классе/модуле/системе. Так же нарушает идею ООП - разделить одну проблему на несколько маленьких и создание для решений каждой из них. Божественный объект - объект, который хранит большую часть информации/функционала о всей программе, а также предоставляет большинство методов для управления этими данными. В результате вместо того, чтобы общаться друг с другом непосредственно, другие объекты полагаются на божественный объект. Так как на божественный объект ссылается так много кода, его обслуживание (внесение изменений) становится сложным: велик риск сломать существующую функциональность.

- **Объектная клоака** - переиспользование объектов, находящихся в непригодном для переиспользования состоянии. Были созданы объекты, возможно использованы, а потом мы решили использовать там, где они никак не подходят. Например, есть dll на 32 бита, делаем новый проект на 64 бита, но подключаем dll на 32, она не пригодна для использования, работать будет, но не всегда верно.
- **Полтергейст** - объекты, чье единственное предназначение – передавать данные другим объектам. К примеру из одного класса в другой. То есть только для передачи данных мы создали отдельный класс, единственное предназначение которого, к примеру считать файл и передать бинарный блок с определенным размером. Однако, если бы мы делали проверки на корректность данных, нулевой ли буфер и тд, то есть выполняет некую логику, то все хорошо.
- **Проблема йо-йо** - чрезмерная размытость сильно связанного кода по иерархии классов. Сильно связаный код - код который, к примеру, выполняется по порядку. Пример: делаем обработку данных, есть иерархия классов *A, B* и *C.* И программа работает следующим образом: *A - вызов функции 1; B - вызов функции 2; C - вызов функции 3; B - вызов функции 2; A - вызов функции 4*. То есть в коде, который идет друг за другом мы скачем по разным точкам иерархии, рассчитывая на то, что реализации методов отличаются. Читаемость кода практически отсутствует.
- **Одиночество** - неуместное использование паттерна синглтон. Если система подразумевает, что объект этого типа будет создаваться только одни раз. К примеру, file-reader, push и тд. Для таких объектов не нужно делать синглтон. Это уменьшает возможность расширения, то есть в программе однопоточное чтение => один file-reader. Если для него сделать синглтон, то мы не сможем расширить систему до многопоточного ввода, так как file-reader гарантированно один.
- **Приватизация** - сокрытие функциональности в приватной части, что затрудняет расширение в наследниках. Если класс будет расширяться, и от него будут наследоваться, то все приватная логика будет недоступна. Поэтому лучше делать protected.
- **Френд-зона -** Неуместное использование дружественных классов и функций. Есть 2 класса *A* и *B.* Хотим, что бы *B* мог вызывать класс *A*, при этом единственными пользователями класса *A* будет класс *B,* то стоит объявить его френдом, спрятав все возможности класса *A* в private часть, чтобы только *B* мог вызывать их. Но не стоит все классы внутри подсистемы делать френдами друг другу.
- **Каша из интерфейсов** - Объединение нескольких интерфейсов, предварительно разделенных, в один. К примеру, класс реализует несколько интерфейсов и мы весь общий интерфейс отдаем

пользователю. Важным моментом в этом антипаттерне является то, что происходит объединение интерфейсов, которые были предварительно разделены. Но объединять интерфейсы, которые никак не были связаны - можно.
- **Висящие концы** - Интерфейс, большинство методов которого бессмысленные и являются «пустышками». Методы с пустым телом, без какой-либо логики.
- **Заглушка** - Попытка «натянуть» мало подходящий по смыслу интерфейс на класс. К примеру, есть интерфейс для воспроизведения музыки, а мы берем его и используем для текстового редактора. И для метода перемотки аудиофайла мы реализуем прокрутку текста на экране. Это сильно ухудшает поддержку кода, нарушается логика построения программы.

В кодировании
- **Ненужная сложность**
- **Действие на расстоянии** - взаимодействие между широко разнесенными частями системы.
- **Накопить и запустить** - установка параметров подпрограмм в глобальных переменных.
- **Слепая вера** - недостаточная проверка корректности и полноты исправления ошибки или результата работы.
- **Лодочный якорь** - сохранение неиспользуемой части программы.
- **Активное ожидание** - потребление ресурсов в процессе ожидания запроса, путем выполнения проверок, чтений файлов и т.д., вместо асинхронного программирования.
- **Кэширование ошибки** - не сбрасывание флага ошибки после ее обработки.
- **Воняющий подгузник** - сброс флага ошибки без ее обработки или передачи на уровень выше.
- **Проверка типа вместо интерфейса** - проверка на специфический тип, вместо требуемого определенного интерфейса.
- **Инерация кода** - избыточное ограничение системы из-за подразумевания постоянной ее работы в других частях системы.
- **Кодирование путем исключения** - добавление нового кода для каждого нового особого случая. К примеру, через if'ы обрабатываем все ошибки, если появится новая ошибка, то просто добавим новый if. Такой код постоянно нуждается в поддержке.
- **Таинственный код** - использование аббревиатур/сокращений вместо логичных имен.
- **Жесткое кодирование** - внедрение предположение в слишком большое количество точек в системе.

- **Мягкое кодирование** - настраивается вообще все, что усложняет конфигурирование.
- **Поток лавы** - сохранение нежелательного кода из-за боязни последствий его удаления/исправления.
- **Волшебные числа** - использование числовых констант без объяснения их смысла.
- **Процедурный код** - когда Ценастоило отказаться от ООП…
- **Спагетти-код** - код с чрезмерно запутанным порядком выполнения. К примеру, если содержится много операторов goto, исключений и других конструкций, ухудшающих структурированность.
- **Лазанья-код** - использование неоправданно большого числа уровней абстракции. Блоки кода настолько сложны и взаимосвязаны, что изменение одного потребует изменения всех остальных.
- **Равиоли-код** - объекты настолько склеены между собой, что невозможно провести рефакторинг.
- **Мыльный пузырь** - объект, инициализированный мусором (не инициализированный) слишком долго ведет себя как корректный.
- **Мьютексный ад** - внедрение слишком большого количества примитивов синхронизации в код.
- **(Мета-) шаблонный рак** - неадекватное использование шаблонов везде, где только получилось, а не где нужно. Это уменьшает понимание и сопровождение кода и замедляет компиляцию.

Методологические

- **Использование паттернов** - значит, имеется недостаточный уровень абстракции.
- **Копирование-вставка** - нужно было делать более общий код. Копирование (и лёгкая модификация) существующего кода вместо создания общих решений
- **Дефакторинг** - процесс уничтожения функциональности и замены ее документацией.
- **Золотой молоток** - использование любимого решения везде, где только получилось.
- **Фактор невероятности** - гипотеза о том, что известная ошибка не проявится.
- **Преждевременная оптимизация** - оптимизация при недостаточной информации. Программа состоит из 2-х функций, и работает 50 секунд. Мы потратили много времени и улучшили первую функцию. Теперь программа работает 49 секонд. Пришли отчеты по производительности и выяснилось, что первая функция работает 2 секунды, а вторая 47 секунд. Мораль: в начале проверяем, что не оптимизировано и только потом оптимизируем.

- **Метод подбора** - софт разрабатывается путем небольших изменений.
- **Изобретение велосипеда** - создание с нуля того, для чего есть готовое решение.
- **Изобретение квадратного колеса** - создание плохого решения, когда уже есть хорошее готовое.
- **Самоуничтожение** - мелкая ошибка приводит к фатальной. Фатальная ошибка либо нестандартное поведение программы, приводящая к отказу в обслуживании, возникшая вследствие другой менее серьезной ошибки. Например, при возникновении ошибки, приложение начинает очень быстро и много писать в лог, вследствие чего заканчивается место на жёстком диске быстрее, чем это обнаружит мониторинг.
- **Два тоннеля** - вынесение новой функциональности в отдельное приложение вместо расширения уже имеющегося. Чаще всего применяется, когда по каким-либо причинам (в основном, при нехватке времени либо нежелании менеджмента) внесение изменений в уже имеющийся код требует больших затрат, чем создание нового. При этом у клиента в конечном итоге работают два приложения, запускаясь одновременно либо попеременно друг из друга.
- **Коммит-убийца** - внесение отдельных изменений в систему контроля версий без проверки влияния их на другие части программы. Как правило, после подобных коммитов работа коллектива парализуется на время исправления проблем в местах, которые ранее работали безошибочно.

Управление конфигурацией

- **Ад зависимостей (DLL-hell в Windows)** - Разрастание зависимостей до уровня, что раздельная установка/удаление программ становится если не невозможным, то крайне сложным. В сложных случаях различные установленные программные продукты требуют наличия разных версий одной и той же библиотеки. В наиболее сложных случаях один продукт может косвенно потребовать сразу две версии одной и той же библиотеки.

Прочее

- **Дым и зеркала** - демонстрация того, как будут работать ненаписанные функции
- **Раздувание ПО** - разрешение последующим версиям использовать все больше и больше ресурсов
- **Функции для галочки** - превращение программы в «сборную солянку» плохо работающих и не связанных между собой функций (как правило, для того, чтобы заявить в рекламе, что функция есть).

Лекция №6

1. Непрерывная интеграция (CI)- это практика разработки программного обеспечения, которая заключается в слиянии рабочих копий в общую основную ветвь разработки несколько раз в день и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

Автоматические тесты проверяют конкретные модули кода, работу UI, производительность приложения, надежность API и пр. Все эти этапы в совокупности обычно называют «сборкой».

Задача CI — обезопаситься от разрушительных изменений вследствие рефакторинга.

Процесс Build-Deploy-Test в целом выглядит следующим образом: разработчики делают изменения, перед заливкой изменений Build Server собирает всё в Release конфигурации и если сборка проходит, изменения заливаются на сервер.

CI – это своеобразная страховочная сетка, позволяющая разработчикам избежать массы проблем перед сдачей проекта. Поэтому программисты чувствуют себя увереннее, сдавая такой код, но сама работа при этом вряд ли ускорится – возможно, развертывание так и будет выполняться вручную, подолгу, и на этом этапе также могут возникать ошибки.

Максимум, что разработчики могут сделать на первом этапе – добиться, чтобы комплект автоматизированных тестов получился исчерпывающим и достаточно стабильным, чтобы можно было спокойно пускать любую сборку, прошедшую CI, сначала в обкатку, а затем и в продакшен. Так можно обойтись без длительного ручного тестирования (QA).

2.Непрерывная доставка (CD)C.Delivery – это практика автоматизации всего процесса релиза ПО. Идея заключается в том, чтобы выполнять CI, плюс автоматически готовить и вести релиз к продакшену. При этом желательно добиться следующего: любой, кто обладает достаточными привилегиями для развертывания нового релиза может выполнить развертывание в любой момент, и это можно сделать в несколько кликов. Программист, избавившись практически от всей ручной работы, трудится продуктивнее.

Как правило, в процессе непрерывной доставки требуется выполнять вручную как минимум один этап: одобрить развертывание в продакшен и запустить его. В сложных системах с множеством зависимостей конвейер непрерывной доставки может включать дополнительные этапы, выполняемые вручную либо автоматически.

3. Непрерывное развертывание располагается «на уровень выше» непрерывной доставки. В данном случае все изменения, вносимые в исходный код, автоматически развертываются в продакшен, без явной отмашки от разработчика. Как правило, задача разработчика сводится к

проверке запроса на включение (pull request) от коллеги и к информированию команды о результатах всех важных событий.

Непрерывное развертывание требует, чтобы в команде существовала отлаженная культура мониторинга, все умели держать руку на пульсе и быстро восстанавливать систему.

Ответы на прошедшую к/р

Группа 1

1.1 Почему семантический анализ кода проводится после синтаксического анализа кода?

1.2 Зачем в компиляторах используются некоторые внутренние представления программного кода? (например, IR в LLVM)+

Ответы:

1.1 Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка.

1.2 Для того чтобы генерировать оптимизированный, не зависящий от языка код. Ну тот же LLVM IR можно скомпилировать почти под любой язык.

Группа 2

2.1 Зачем нужны дополнительные валидации программной системы после интеграции?

2.2 Что такое “технологический дизайн исправления дефекта”?

Ответы:

2.1 Это нужно чтобы точно убедиться,в работоспособности вашего проекта на других устройствах.Если у вас на компьютере все работает, то это не значит, что у другого пользователя на компьютере все работает. То есть это больше такой этап проверки - от имени пользователя.

2.2

Группа 3

3.1 Что такое ложно-положительные срабатывания при статическом анализе кода? Приведите

пример

3.2 Какие метрики программного кода вы знаете? (приведите минимум 2)

Ответы:

3.1

3.2 Глубина вложенности при наследовании, средняя длинна чего-либо (функции, класса), насколько часто используются рекурсивные методы, количество комментариев, количество пустых строк.

Группа 4

4.1 Что такое “масштабируемость процесса разработки”?

4.2 Приведите пример, когда архитектуру можно охарактеризовать как “хрупкую”
4.3 Приведите пример использования принципа обращения зависимостей
4.4 Приведите пример, иллюстрирующий закон Деметры

Ответы:

4.1 Насколько просто можно распределить разные модули кода между разными людьми. То есть масштабируемость - когда несколько людей могут делать свои отдельные куски общей программы, а потом в конце их уже объединят. Там вводятся какие-то общие требования на взаимосвязь между этими модулями и все знают, что объединить точно получится. А не так, когда один человек долго долго пишет свой код.
4.2 Когда несколько независимых по функционалу модуля зависят от одного общего модуля, и всё летит к чертям если он ломается. Как будто ваш сервер упал из за рассинхрона с сервисом точного времени.
4.3 Пример - использование порождающего паттерна AbstractFactory. Мы от него наследуем фабрики, и реализация функционала фабрики ложится на дочерний класс.
4.4 Если есть класс Main, Adapter, Screen, и Main имеет доступ к Adapter, а Adapter к Screen, Main не должен иметь доступа и взаимодействовать с Screen.

Группа 5

5.1 Допустимо ли изменение порядка выполнения этапов проектирования?
5.2 В чем обычно заключается послегарантийное обслуживание ПО?

Ответы:

5.1 Нет, где ж это видано, чтобы сначала делать большой проект, а потом думать зачем он нужен и искать всякие критические уязвимости, которые могут появится “в корне” программы.
5.2 Когда гарантийное обслуживание закончилось, скорее всего будет предлагаться перейти на новое обновление. И вот тут важно, чтобы сохранить клиента, уметь все его данные без потерь перенести на это новое обновление. Чтобы и программа функционировала лучше, так как это обновление, и он от этого ничего не потерял, а только приобрел новые возможности. И он счастлив, и у вас есть клиент.
Обслуживание может также заключаться в исправлении ошибок + что оно "платностью" отличается от гарантийного

Группа 6

6.1 В чем отличия инкрементной и итерационной моделей разработки?
6.2 В чем схожесть Agile и итерационной моделей разработки?

Ответы:

6.1 Инкрементальная модель подразумевает введение нового функционала as-is, и если новый функционал не отвечает каким то требованиям пользователя, его можно забыть и выкатить другой. В итерационной модели же так сделать нельзя. Нельзя добавлять новый функционал, пока старый не доведен до ума. Как сказал один умный человек: “Это как башня из кубиков”
6.2 Не требует для начала полной спецификации требований. Определяется основная задача, но реализация может меняться с течение времени.

Группа 7

7.1 Приведите пример антипаттерна “проблема йо-йо”
7.2 Приведите пример антипаттерна “висящие концы”
7.3 Как исправить антипаттерн “активное ожидание”? Приведите пример
7.4 Как исправить антипаттерн “проверка типа вместо интерфейса”? Приведите пример

Ответы:

7.1 Любая программа, где есть очень длинное наследование с частым обращениям к суперклассам.
7.2 Интерфейс с методами удалить кнопку, добавить кнопку, и всеми возможными манипуляциями с кнопкой и одной функцией, которая тебе создаёт кнопку по параметрам. В итоге предыдущие методы не нужны, так как есть 1 функция, которая делает всё за них.
7.3 Использовать асинхронное программирование. Например при запросе к серверу не ждать ответа в висящем потоке, а вынести в отдельный, работать дальше, а когда придёт ответ - обработать его
7.4

Группа 8

8.1 В чем отличие Continuous Integration и Continuous Delivery?
8.2 В чем отличие Continuous Delivery и Continuous Deployment?

Ответы:

8.1 Continious delivery отвечает за быструю развертку ПО на какой-то машине, а Continious Integration - это просто постоянный прогон тестов и пуш в центральный репо готового кода, и направлено это на как можно более раннее обнаружение ошибок.
8.2 Continious deployment это просто быстрый и безболезненный деплой нового, рабочего функционала на сервера (ну или туда, где это надо), направленный на то, чтобы последние всегда работали на новых версиях (делается автоматически)
Continuous delivery обеспечивает постоянный выпуск обновлений пользователям.