Технологии программирования

Старичков Никита

Лекция №1

Компиляторы

Компиляция - трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду. Входной информацией для компилятора является описание алгоритма или программа, а на выходе - эквивалентное описание алгоритма на машинно-ориентированном языке

1) Лексический анализ

- Разбор последовательности символов на распознанные сущности - лексемы, с последующим анализом и выдачей токенов
- Оно же "токенизация", токен - название множества лексем
- Одному токену может соотвествовать целое множество лексем

2) Синтаксический анализ

- Генерация дерева синтаксического разбора
- Внутренние вершины - операторы, Листья - операнды
- Обход дерева Post-order алгоритмом

3) Семантический анализ

- Проверка корректности

`return return`

- Статическая проверка типов

`(int)a / (string)b`

- Вывод типов (выражения наподобие *auto*)
- Раскрытие "синтаксического сахара"

4) Оптимизация

- Перестроение дерева для генерации более эффективного машинного кода

5) Генерация кода

- Генерируется машинный код
- На выходе - объектный файл
- Машинный код - уже машиннозависимый, т.е. для каждой архитектуры/поколения процессоров/моделей может быть разным

Методы отладки ПО

1) Воспроизведение дефекта

- Необходимо хранить все сборки, уходившие клиентам, и исходный код, из которого они были получены
- Необходимо иметь инструменты логирования
  - Поддержание баланса между размером логов и их подробностью
  - Разные уровни логирования

2) Анализ дефекта

- Рассмотреть не только сценарий от пользователя, а еще и другие вероятные сценарии
- Проанализировать не только часть кода, но и код вокруг

3) Дизайн исправления дефекта

- В отличие от исправления ошибки в коде исправление логической ошибки может быть в разы сложнее

4) Исправление дефекта

- Исправление не должно привнести новых дефектов

Лекция №2

5) Валидация исправления

- Ревью кода членами команды
- Ревью кода сторонними разработчиками (проверка качества кода, содержание уязвимостей)
- Воспроизведение сценария дефекта
- Запуск юнит-тестов для этой подсистемы
- Запуск всех остальных тестов

6) Интеграция исправления в код или целевую систему

- Разрешение конфликтов при мердже

7) Дополнительные валидации после интеграции

- Запуск тестов в основной ветке

1. Воспроизведение дефекта

1) Точное соответствие версии

2) Точное соответствие настроек

- Ведение логов настроек

3) Получение данных, на которых наблюдается ошибка

- Логирование операций с данными (пользователь не всегда готов прислать данные)

4) Точное воспроизведение действий пользователя/сценария

- Ведение логов сценариев

2. Анализ дефекта

1) Root-cause (источник проблемы)

- Ошибки в коде, логические/технологические проблемы

2) Условия возникновения

3) Область "повреждения"

4) Кто привнес?

- Возможно необходимо лучше покрыть подсистему тестами

5) В какую версию?

3. Дизайн исправления дефекта

1) Технический

- Что поменять в коде?

2) Архитектурный

- Что изменится в архитектуре системы и ее логике/поведении?

3) Технологический

- Добавление новых технологических решений в систему или изменение старых

4) Ревью и согласование

4. Изменение дефекта

1) Не привнести новые дефекты

2) "Ожидаемое" поведение и логика

- Повторять "ожидаемое" поведение системы в целом, которое не совсем корректно, или реализовать новую правильную логику, которая может конфликтовать с другими частями системы

3) "Костыли" и "грязные хаки"

4) Документирование

5. Валидация исправления

1) Проверка исходного сценария

2) Проверка сценарием возникновения ошибки

3) Проверка связанных сценариев

4) Полноценное тестирование системы

6. Интеграция исправления в код или целевую систему

1) Совмещение со "стволом"

2) Проверка сборки и работоспособности

3) Деплоинг новой версии в целевую систему

4) Обновление серверных/пользовательских приложений

5) Обновление документации!

7. Дополнительные валидации после интеграции

1) Проверка сценария возникновения ошибки

2) Проверка связанных сценариев

3) Полноценная проверка системы

4) Проверка устойчивости и работоспособности всех версий

5) Проверка корректности обновления

Лекция №3

Стандартные техники отладки ПО

1) Запуск программ в отладчики (трассировка)

- Софтверный
- "Железный"
- Удаленный дебагер

2) Логирование

- Работы подсистемы
- Программного кода

3) Анализ кода без исполнения программы

- "Метод пристального взгляда"

4) Анализ поведения системы

- Упрощения сценария
- Ограничение объема данных
- Упрощение данных/запроса

5) Unit-тестирование

6) Прототипирование

- Создание упрощенного прототипа для отладки конкретного модуля

7) Отладка с помощью дампов

- Анализирование снимка памяти

8) Отладка с помощью перехватов

- Можем отследить, когда вызывается определенная функция, и выполнить дополнительные действия, например, посмотреть аргументы, с которыми была вызвана функция

9) Профилирование кода

- Можем, посмотреть как часто вызываются отдельные функции и сколько работают

10) Выполнение кода в другой среде

11) Отладка методом RPC (remote procedure call)

- Удаленный вызов процедур

12) Отладка путем анализа документации, проектных документов и т.д.

- Можно найти логические ошибки в архитектуре

13) Отладка трансляцией кода

- "Трансляция вниз" (из высокоуровневого языка в низкоуровневый)
- "Трансляция вверх"

14) Отладка разработкой интерпретатора

15) Метод индукции (от частного к общему)

16) Метод дедукции (от общегок частному)

17) Обратное движение по алгоритму

Статический анализ кода

Статические анализаторы

1) Что умеют?

- Выявление ошибок в коде
- Рекомендации по оформлению кода
- Подсчет различных метрик исходного кода

2) Преимущества

- Полное покрытие кода
- Не зависит от используемого компилятора и среды разработки
- Можно легко и быстро обнаруживать опечатки и прочее

2) Слабости

- Трудности в выявлении ошибок в параллельном программировании и утечек памяти
- Ложно-положительные срабатывания

Критерии оценки архитектуры

Критерии хорошей архитектуры

1) Эффективность системы

- Надежность
- Безопасность
- Производительность
- Масштабируемость

Лекция №4

2) Гибкость системы

- Изменение текущего функционала
- Исправление ошибок

  Если даже простая ошибка требует больших изменений архитектуры, то такая система недостаточно гибкая
- Настройка системы
  - Под пользователей
  - Под разные задачи

3) Расширяемость системы

- Возможность добавлять новые сущности о функции
- *Внесение наиболее вероятных изменений должно требовать наименьших усилий*

  Обоснованный выбор того, что будет реализовано с возможностью расширения, а что будет "прибито гвоздями"

4) Масштабируемость процесса разработки

- Чем больше людей, тем меньше уходит времени на разработку

$$\frac{t_{old} * p_{old}}{t_{new} * p_{new}}$$

  Чем коэффициент ближе к **1**, тем более масштабируем процесс разработки

5) Тестируемость

- Система должна быть тестируема

6) Возможность повторного использования

- Широкие возможности повторного использования в других проектах
- Части проекта друг от друга отделимы

7) Сопровождаемость

- Способность обеспечить хорошее сопровождение

Критерии плохой архитектуры

1) Жескость

- Тяжело изменить, настроить

2) Хрупкость

- Изменения нарушают другие модули
- Мало того, что тяжело настроить, так после этого что-нибудь еще и развалится

3) Неподвижность

- Тяжело извлечь модуль наружу (Идеальный модуль - черный ящик)

High Cohesion + Low Coupling

- **Hign Cohesion**
  - Высокая сопряженность внутри модуля
  - Модуль сфокусирован на одной задачи
- **Low Coupling**
  - Слабая связь между модулями
  - Модули независимы друг от друга, либо слабо связаны

Закон Деметры (Law of Demeter)

Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B и у объекта B есть доступ к объекту C (принцип минимального знания)

Принципы **SOLID**

- **Принцип единственной ответственности (The Single Responsibility Principle - SRP)**
  - Каждый должен иметь лишь одну ответственность и эта ответственность должна быть инкапсулирована в класс
  - Существует лишь одна причина (изменение функции), проводящая к изменению класса
  - Меняем класс только тогда, когда хотим изменить функцию, и меняем только его
- **Принцип открытости/закрытости (The Open Closed Principle - OCP)**
  - Программные сущности должны быть открыты для расширения, но закрыты для модификации
  - Изменения должны быть допустимы путем внесения новых сущностей, но внесение новых сущностей не должно вести к изменению кода, который эти сущности использует
- **Принцип подстановки Барбары Лисков (The Liskov Substitution Principle - LSP)**
  - Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы
  - Наследуемый класс должен дополнять, а не изменять базовый

- **Принцип разделения интерфейса (The Interface Segregation Principle - ISP)**
  - Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения
  - Например, лучше создать два отдельных интерфейса на работу с фото и видео, чем один общий
- **Принцип инверсии зависимостей (The Dependency Inversion Principle - DIP)**
  - Зависимость на Абстракциях, нет зависимости на что-то конкретное
  - Модули верхних уровней не должны зависеть от модулей нижних уровней
  - Абстракции не должны зависеть от деталей, детали должны зависеть от абстракций
  - Например, при наследовании зависимости могут возникать на уровне базовых классов, но классы-потомки не должны зависеть друг от друга или от базовых классов

Лекция №5

Этапы проектирования

1. Формирование требований

1) Обследование объекта

2) Обоснование необходимости создания

3) Формирование требований пользователей

4)Подготовка отчетности по этапу

5) Общение с клиентом (заказчиком)

6) Общение с пользователем

7) Анализ прикладной области

8) Формирование оценок требуемой производительности/качества/...

2. Разработка концепции

1) Изучение объекта автоматизации

- Изучить объект очень детально
- Формализовать все данные с предыдущих этапов

2) Проведение необходимых НИР (научно-исследовательских работ)

- Понять можно ли вообще реализовать концепцию
- Если можно то понять как это реализовать

3) Разработка вариантов концепции системы

4) Подготовка отчетности по этапу

5) Выбор формата поставки

- Например, веб-страница, мобильное приложение, десктопное приложение

6) Целевое оборудование

7) Построение высокоуровневой архитектуры системы

- Описание архитектуры системы на высоком уровне абстракции - описание модулей, из которых состоит система

8) Выбор/разработка новых технологий/алгоритмов/...

- Обоснование выбора

3. Техническое задание

Описание финальной, четкой формализации задания

1) Описание системы

- Описание концепции в целом

2) Описание функциональности

- Более конкретное, детальное описание

3) Описание требований

4) Описание сценариев использования

5) Условия сдачи

<Темы еще не были разобраны>

4. Эскизный проект

5. Технический проект

6. Рабочая документация

7. Ввод в действие

8. Сопровождение системы

</Темы еще не были разобраны>

Антипаттерны

1) В объектно-ориентированном программировании

- Связанны с особенностью проектирования

2) В кодировании

- Как не нужно писать код

3) Методологические

- Как не нужно организовывать работу

4) Управления конфигурацией

5) Прочее

Антипаттерны в ОПП

1) Базовый класс-утилита

- Наследование функциональности из класса-утилиты вместо делегирования ему
- Т.е. вместо использования функциональности другого класса происходит наследование от него с последующим получением его функциональности

2) Anemic Domain Problem

- Боязнь размещать логику в объектах предметной области
- Например, если мы создаем класс, соответствующий реальной сущности, и выносим поля/методы соответствующие этому классу в другое место (например, потому что в реальной жизни этот реальный объект ими не обладает (файл сам себя не сохраняет, дата производства колеса на колесе не пишется))
- Т.е. когда при разработке программной модели, соответствующей реальной предметной области, появляются методы/поля характерные только программной реализации, но которых нет в реальном мире, происходит разделение на разные классы

3) Вызов предка

- Для реализации функциональности методу потомка приходится вызывать те же методы родителя
- Возможным решением является добавление новых виртуальных методов

4) Ошибка пустого подкласса

- Когда класс обладает различным поведением по сравнению с классом, который наследуется от него без изменений
- Т.е. при наследовании без изменений новый объект перестает работать так же, как и исходный класс
- В **SOLID** говорится только о корректности работы, а здесь говорится о том, что должно работать точно так же

5) Божественный объект

- Концентрация функциональности в одном классе/модуле/системе

6) Объектная клоака

- Переиспользование объектов, находящихся в непригодном для переиспользования состоянии
- Перед переиспользованием объекта должен произойти процесс подготовки к повторному использованию

7) Полтергейст

- Объекты, чье единственное предназначение – передавать данные другим объектам
- Т.е. создание объекта *C* для связи между **конкретными** объектами *A*, *B*, который сам по себе больше ничего не делает
- Важно, что объекты обеспечивающие связь в общем случае, под этот антипаттерн не попадают

8) Проблема йо-йо

- Чрезмерная размытость сильно связанного кода по иерархии классов
- Например, в одном месте в коде происходят вызовы функций из разных уровней иерархии

9) Одиночество

- Неуместное использование паттерна синглтон

10) Приватизация

- Сокрытие функциональности в приватной части, что затрудняет расширение в наследниках

11) Френд-зона

- Неуместное использование дружественных классов и функций

12) Каша из интерфейсов

- Объединение нескольких интерфейсов, предварительно разделенных, в один

13) Висящие концы

- Интерфейс, большинство методов которого бессмысленные и являются "пустышками"
- Например, в базовом классе объявлены методы, которые в некоторых классах-наследниках так никогда и не будут реализованы

14) Заглушка

- Попытка "натянуть" на объект уже имеющийся малоподходящий по смыслу интерфейс, вместо создания нового

Антипаттерны в кодировании

1) Ненужная сложность

- Внесение ненужной сложности в решение

2) Действие на расстоянии

- Взаимодействие между широко разнесенными частями системы

3) Накопить и запустить

- Установка параметров подпрограмм в глобальных переменных
- Более правильным решением является последовательный запуск подпрограмм, так код получается более понятный, изменяемый

- Кроме того в процессе исполнения подпрограмм данные могли измениться, или что-то могло сломаться раньше, чем потребовались какие-то полученные данные

4) Слепая вера

- Недостаточная проверка корректности и полноты исправления ошибки или результата работы

5) Лодочный якорь

- Сохранение неиспользуемой части программы

6) Активное ожидание

- Потребление ресурсов в процессе ожидания запроса, путем выполнения проверок, чтений файлов и т.д., вместо асинхронного программирования

7) Кэширование ошибки

- Несбрасывание флага ошибки после ее обработки

8) Воняющий подгузник

- Сброс флага ошибки без ее обработки или передачи на уровень выше

9) Проверка типа вместо интерфейса

- Проверка на специфический тип, вместо требуемого определенного интерфейса

10) Инерация кода

- Избыточное ограничение системы из-за подрузамевания постоянной ее работы в других частях системы

11) Кодирование путем исключения

- Добавление нового кода для каждого нового особого случая

12) Таинственный код

- Использование аббревиатур/сокращений вместо логичных имен

13) Жесткое кодирование

- Внедрение предположений в слишком большое количество точек в системе

14) Мягкое кодирование

- Настраивается вообще все, что усложняет конфигурирование

15) Поток лавы

- Сохранение нежелательного (например, содержащего уязвимости) кода из-за боязни последствий его удаления/исправления

17) Волшебные числа

- Использование числовых констант без объяснения их смысла

18) Процедурный код

- Когда стоило отказаться от ООП...
- Например, весь код состоит из вложенных вызовов статических функций

19) Спагетти-код

- Код с чрезмерно запутанным порядком выполнения

20) Лазанья-код

- Использование неоправданно большого числа уровней абстракции

21) Равиоли-код

- Объекты настолько склеены между собой, что невозможно провести рефакторинг

22) Мыльный пузырь

- Объект, инициализированный мусором (не инициализированный), слишком долго ведет себя как корректный

23) Мьютексный ад

- Внедрение слишком большого количества примитивов синхронизации в код

24) (Мета-)шаблонный рак

- Неадекватное использование шаблонов везде, где только получилось, а не где нужно

Методологические антипаттерны

1) Использование паттернов

- Значит, имеется недостаточный уровень абстракции

2) Копирование-вставка

- Нужно было делать более общий код

3) Дефакторинг

- Процесс уничтожения функциональности и замены ее документацией

4) Золотой молоток

- Использование любимого решения везде, где только получилось
- Например, писать быструю сортировку, чатик, сайт, мобильное приложение на *C++*

5) Фактор невероятности

- Гипотеза о том, что известная ошибка не проявится

6) Преждевременная оптимизация

- Оптимизация при недостаточной информации

8) Метод подбора

- Софт разрабатывается путем небольших изменений

9) Изобретение велосипеда

- Создание с нуля того, для чего есть готовое решение

10) Изобретение квадратного колеса

- Создание плохого решения, когда уже есть хорошее готовое

11) Самоуничтожение

- Мелкая ошибка проводит к фатальной

12) Два тоннеля

- Вынесение нового функционала в отдельное приложение

13) Коммит-убийца

- Внесение изменений без проверки влияния на другие части программы

Антипаттерны управления конфигурацией

1) Ад зависимостей *(DLL-hell в Windows)*

- Разрастание зависимостей до уровня, что раздельная установка/удаление программ становится если не невозможным, то крайне сложные

Прочее

1) Дым и зеркала

- Демонстрация того, как будут работать ненаписанные функции

2) Раздувание ПО

- Разрешение последующим версиям использовать все больше и больше ресурсов

3) Функции для галочки

- Превращение программы в "сборную солянку" плохо работающих и не связанных между собой функций (функциональностей)