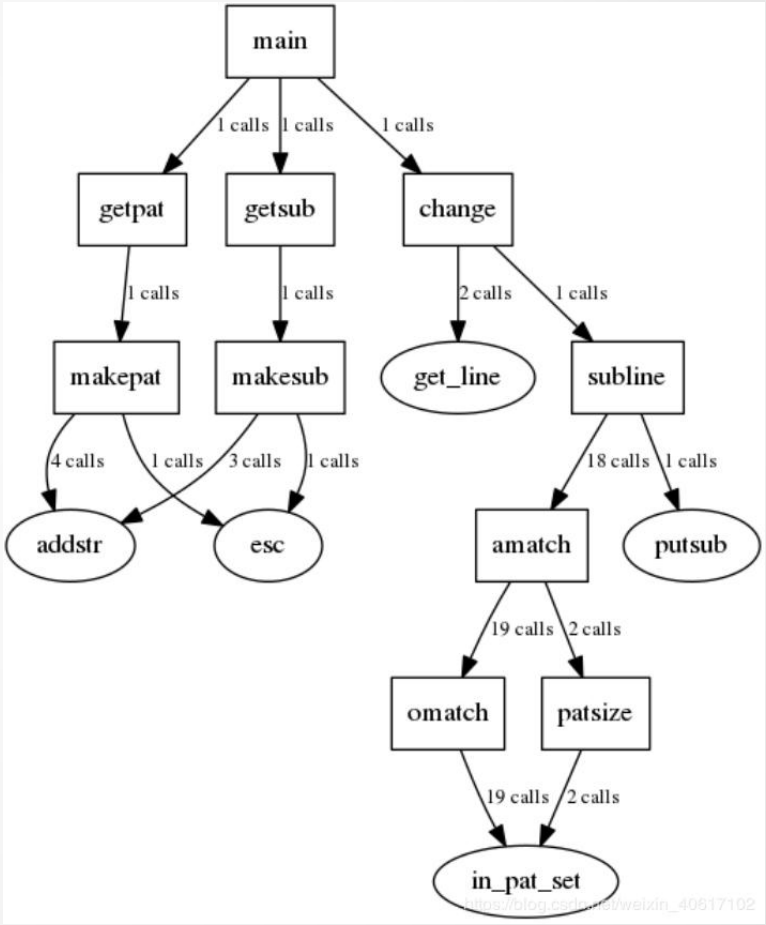
Тестирование ПО

Тестирование и процедурное программирование

Кулаков Кирилл Александрович

Процедурное программирование

- Парадигма программирования с использованием операторов последовательного исполнения, ветвления и безусловного перехода
- Корневая функция main
- Функции вызываются из функций — дерево вызовов
- Объект тестирования — функция / процедура



Функция как объект тестирования

- <result_type> function(param_1, param_2, …, param_n)

- Возвращаемое значение (результат) всегда одно
- Прямые входные данные — параметры
- Косвенные входные данные:
  - Глобальные переменные, в т.ч. структуры
  - Внешние объекты (файлы, потоки данных, …)
- Вложенные вызовы других функций/процедур

Подготовка кода к тестированию

- Цель: минимизировать затраты на тестирование

- Большой кусок кода — сложные тесты и проверки
  - делим код на функции и процедуры
- Много функций в одном файле
  - разносим функции по разным файлам (идеал — 1 функция — 1 файл)
- Вложенные вызовы
  - разделение объявления функции и ее использования

Тестирование внутренностей функции

- Тестирование белым ящиком — построение тестов на основе внутренней структуры объекта
- Техника Белого ящика включает в себя следующие методы тестирования:
  - покрытие решений
  - покрытие условий
  - покрытие решений и условий
  - комбинаторное покрытие условий

Белый ящик

- Покрытие операторов
  - выполнение каждого оператора программы по крайней мере один раз
  - Пример:

```
void func(int a, int b, float x) {
    if ((a > 1) && (b == 0)) x = x/a;
    if (a == 2 || x > 1) x++;
}
```

  - единственный тест со следующими значениями входных данных (a = 2, b = 0, x = 3)

Белый ящик

- Покрытие решений
  - каждое условие (if) в программе примет как истинное значение, так и ложное значение
  - Пример:

```
void func(int a, int b, float x) {
    if ((a > 1) && (b == 0)) x = x/a;
    if (a == 2 || x > 1) x++;
}
```

  - Тесты: (a=0, b=0, x=0), (a=2, b=0, x=2).

Белый ящик

- Покрытие условий
  - все возможные результаты каждого условия в решении были выполнены по крайней мере один раз
  - выполнение каждого оператора по крайней мере один раз
  - Пример:

```
void func(int a, int b, float x) {
    if ((a > 1) && (b == 0)) x = x/a;
    if (a == 2 || x > 1) x++;
}
```

  - Тесты: (a=0, b=0, x=0), (a=2, b=1, x=2), (a=2, b=0, x=2).

Белый ящик

- Покрытие условий и решений
  - результаты каждого условия выполнялись хотя бы один раз
  - результаты каждого решения так же выполнялись хотя бы один раз
  - каждый оператор должен быть выполнен хотя бы один раз
- Недостатки:
  - не всегда можно проверить все условия
  - невозможно проверить условия, которые скрыты другими условиям
  - метод обладает недостаточной чувствительностью к ошибкам в логических выражениях

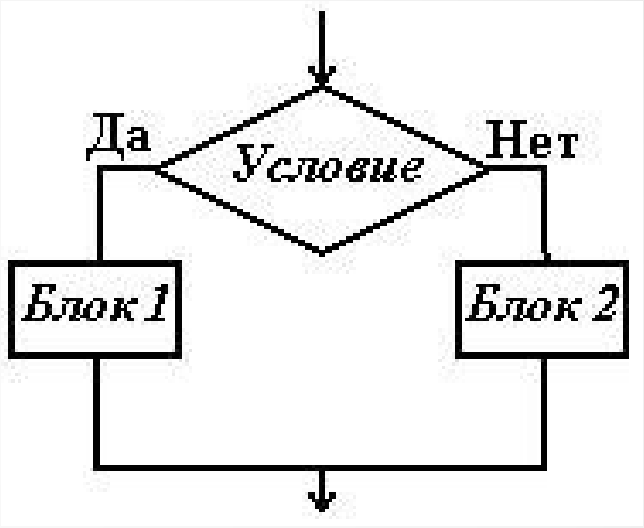
Белый ящик

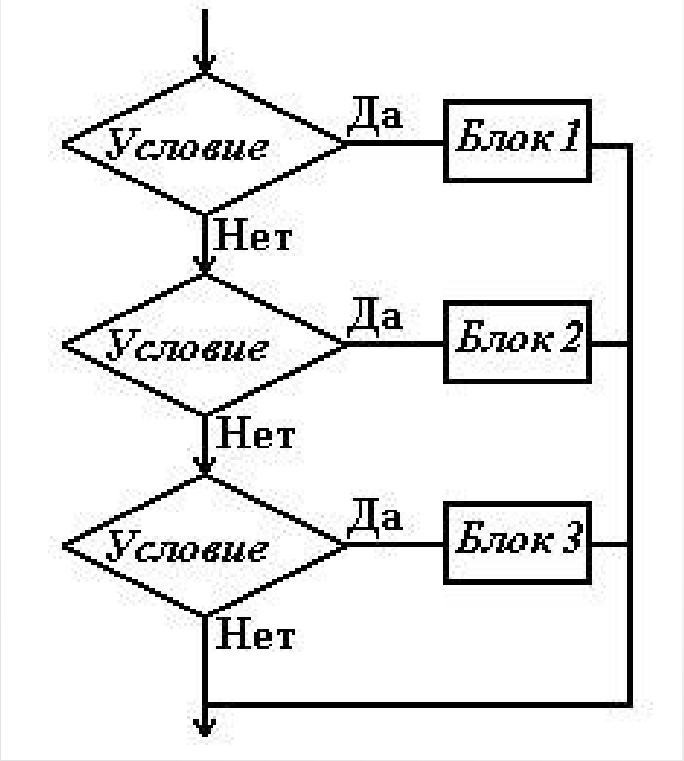
- Комбинаторное покрытие условий
  - все возможные комбинации результатов условий в каждом решении выполнялись хотя бы один раз
  - каждый оператор должен быть выполнен хотя бы один раз
  - Пример:

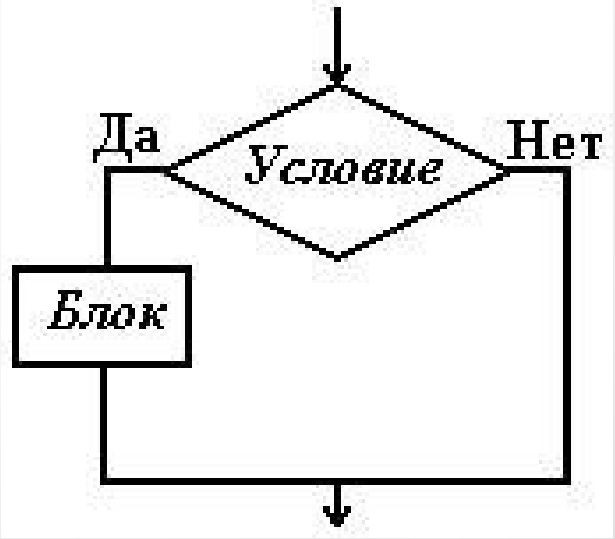
void func(int a, int b, float x) {
  if ((a > 1) && (b == 0)) x = x/a;
  if (a == 2 || x > 1) x++;
}
	a > 1, b = 0.
	a > 1, b ≠ 0.
	a ≤ 1, b = 0.
	a ≤ l, b ≠ 0.
	a = 2, x > 1.
	a = 2, x ≤ l.
	a ≠ 2, x > 1.
	a ≠ 2, x ≤ l.
		a = 2; b = 0; x = 4
		a = 0; b = 0; x = 0
		a = 2; b = 1; x = 0
		a = 0; b = 1; x = 2

Тестирование условий

- Варианты тестов:
  - условие истинно
  - условие ложно

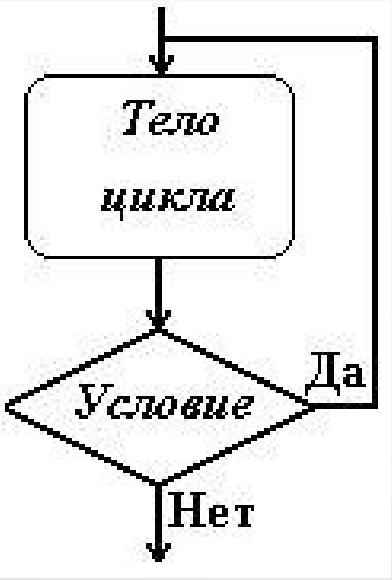


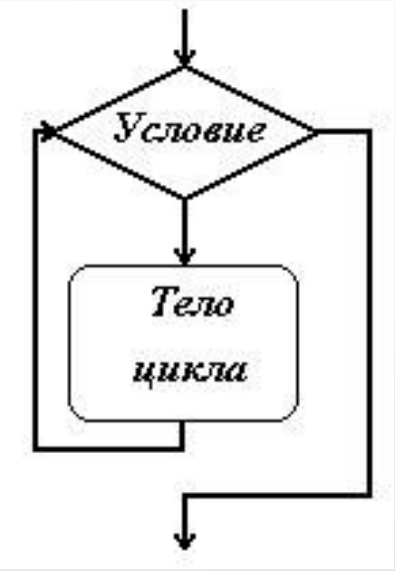




Тестирование циклов

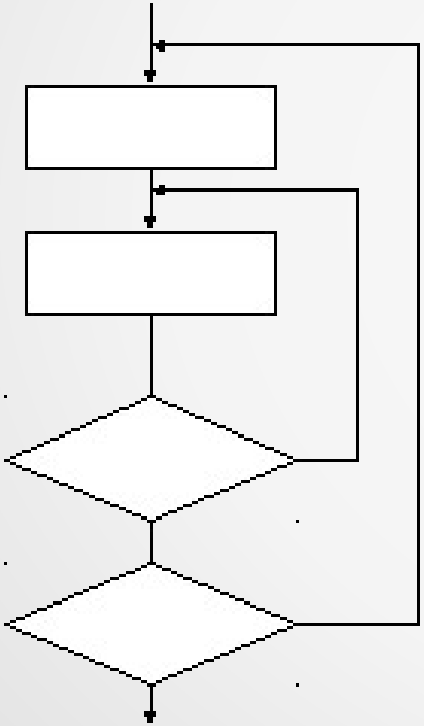
- Варианты тестов
  - тело не выполняется
  - тело выполняется 1 раз
  - тело выполняется m раз (m<n)
  - тело выполняется n раз

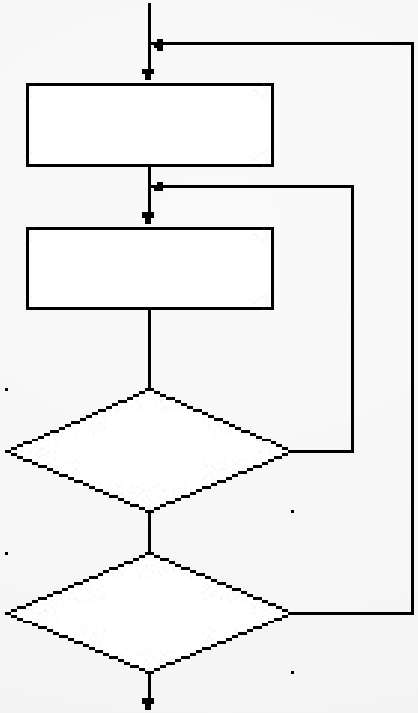


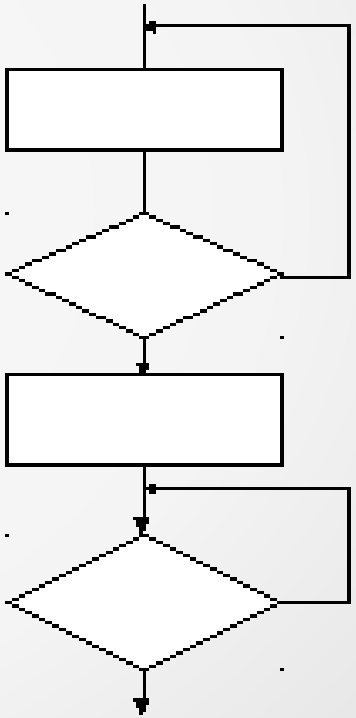


Тестирование циклов

- Циклы могут быть разные







Тестирование процедур

- Процедура = функция без возвращаемого значения
- Проблема проверки результата
  - Проверяем изменение косвенных объектов
  - Проверяем изменение внешних объектов
  - Проверяем вызовы вложенных процедур/функций

Подмена зависимостей

- Реальный объект не всегда подходит для использования в тестировании
  - Сложная работа
  - Отсутствие реализации
  - Объект сторонний (например, библиотека для firebase)
  - Невозможность изменения результата (имитация редких ошибок)
  - Невозможность отслеживания вызова
- Решение: использование подмены объекта

Заглушка

- Заглушка — управляемая замена существующей зависимости (или компаньона) в системе
- Функция-заглушка — функция, не выполняющая никакого осмысленного действия, возвращающая пустой результат или входные данные в неизменном виде.

- Преимущества
  - наглядность при проектировании структуры приложения
  - простота реализации
  - ограничение доступа к окружению

Использование заглушки

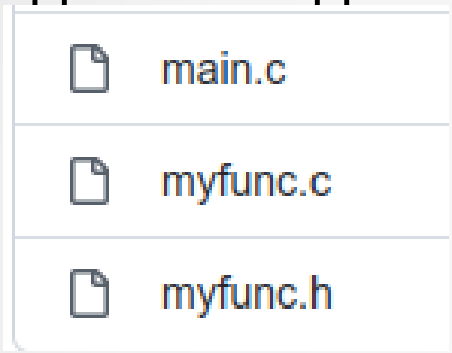
- Определение интерфейса вызова (прототипа)
  - модификация кода для выделения интерфейса
- Определение необходимого результата
  - возвращаемое значение
  - воздействие на систему
- Реализация заглушки
- Запуск теста для объекта с заглушкой

Как запустить заглушку?

- Необходима возможность подмены куска кода (зазор)
  - Выделение прототипа функции/процедуры
  - Разделение функций по отдельным файлам
  - Управляемая компиляция тестов (замена объектных файлов)
- Контроль работы заглушки
  - Использование самодельного кода (запись вызовов)
  - Использование mock объекта внутри функции

Пример встраивания заглушек на языке С

- Делаем разделение кода на функции и прототипы, функции выделяем в динамические библиотеки (shared library myfunc)



- Пишем заглушку для теста «mockmyfunc.c»:

```
#include «myfunc.h»
Int myfunc(int val) {
    Return global_myfunc_ret;
}
```

Пример встраивания заглушек на языке C

- Линковка драйвера с объектным файлом mockmyfunc и библиотекой myfunc
  - Компилятор присоединяет первую найденную реализацию

- Запуск теста:

```
TEST(…) {
    global_myfunc_ret = 5;
    EXPECT_EQ(tested_function(), 3);
}
```

Симуляция внешних объектов

- Имитация деятельности объекта/модуля
- Использование для сторонних объектов или связей
  - сторонние библиотеки
  - системные библиотеки ОС
  - Legacy-код
- Необходимо наличие интерфейса взаимодействия (API)

Примеры

- Реализация системных утилит
- Реализация драйверов виртуальных объектов
- Реализация эмуляторов
- Реализация веб сервисов

Недостатки заглушек/симуляций

- Написание дополнительного кода
- Заглушки могут быть дорогостоящими
- Необходимость рефакторинга для использования заглушек
- Проблема соответствия заглушки и реального объекта
- Проблема "интеллектуализации" заглушки
- Проблема определения результата работы заглушки
- Заглушки могут скрывать ошибки
- Сложно использовать заглушку в других тестах

Интеграционное тестирование

- Проверка взаимодействия функций/процедур
  - Вложенный вызов
  - Последовательная работа с данными
- Заглушки заменяются на реальные методы