Лекция 11. Цикл разработки программного обеспечения. Критерии качества

Цикл разработки программного обеспечения – это весь процесс его создания и применения от начала до конца. Этапы этого цикла и удельный вес общих затрат на каждый из этапов приведены на рис. 11.1.

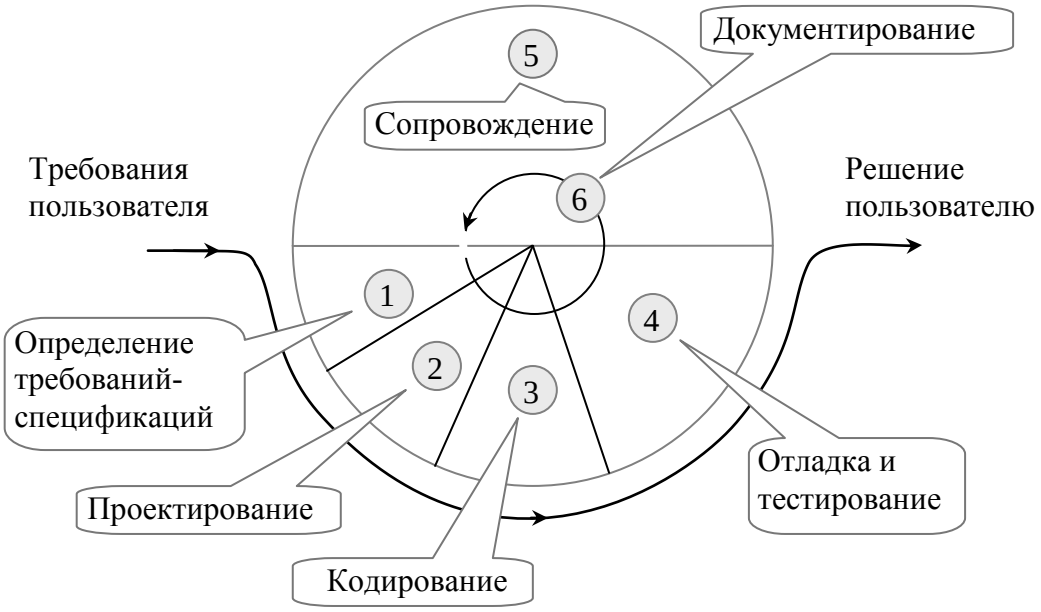


Рис. 11.1. Цикл разработки программного обеспечения

Как видно из рисунка, примерно половина затрат приходится на этап, называемый *сопровождением*. Суть сопровождения – внесение изменений в уже находящуюся в эксплуатации программу и поддержание ее в рабочем состоянии, а также накопление пожеланий пользователей для расширения функциональности в рамках создания новых версий программного продукта на новом витке цикла. Это чрезвычайно важная часть жизни программы, подробное рассмотрение которой выходит за рамки лекций. Сложность этапа сопровождения практически полностью определяется качеством выполнения предшествующих этапов разработки.

Этапы разработки программы отображены в нижней половине рисунка в той последовательности, как они выполняются. Этапы обладают свойством преемственности, но принципиально важно, чтобы эта преемственность реализовалась сверху вниз, т.е. чтобы предшествующие этапы захватывали последующие, а не наоборот. Возвратов назад должно быть как можно меньше.

Краткая характеристика этапов:

1. ***Составление внешней спецификации***. Цель – уточнение постановки задачи, а именно:

- *точная* формулировка задачи и всех имеющихся *ограничений*;
- определение *состава* и *структуры* входных и выходных данных и *форм* их ввода и вывода;
- описание *аномалий* – ситуаций, при которых решение не может быть получено (отсутствие необходимых файлов данных, выход исходных данных за границы диапазона, деление на ноль и т.п.);
- описание *тестов* – примеров, на которых будет проверяться работа программы;
- описание *метода решения* – связей входных и выходных величин, общей идеи и схемы решения задачи.

Спецификация представляет собой интерфейс между разработчиком программы и ее пользователем. Большая часть желательных качеств программы должна быть сконцентрирована в ее спецификации, так, чтобы программа после первой ее реализации требовала улучшений в минимально возможной мере. Исключением является быстродействие, которое можно повышать уже после реализации программы.

2. ***Проектирование алгоритма***. Цель – создание алгоритма, выделение подзадач. Центральный и наиболее трудоемкий этап. Требует творчества и не может быть полностью формализован. Созданный на этом этапе текст алгоритма – носитель всей информации о решении, поэтому для записи алгоритма целесообразно использовать наиболее универсальные и одновременно точные средства – «отказ от красивого и умного в пользу незамысловатого и четкого». Такими средствами являются блок-схемы и полуформальные языки, используемые для проектирования, – ***псевдокоды***.

3. ***Кодирование программы*** состоит в переводе алгоритма с одного языка, достаточно жесткого (псевдокода или блок-схем), на другой, полностью формализованный (язык программирования). Если первые два этапа выполнены тщательно, то кодирование выполняется без особых трудностей.

4. ***Отладка и тестирование***. Цель – получение правильно и надежно работающей программы. Очень важный этап, требующий особого рассмотрения и здесь представленный только самыми основными положениями.

Отладка – процесс изменения программы с целью исправления ошибок.

Тест – совокупность специально подобранных входных и соответствующих им выходных данных, используемая для контроля правильности программы.

Ошибки в коде

- **Синтаксические**

Может найти синтаксический и лексический анализатор кода.

- **Семантические** (смысловые), логические

Ищете ВЫ

ОТЛАДКА – процесс поиска (локализации) и исправления ошибок

ТЕСТИРОВАНИЕ – один и способов проверки НАЛИЧИЯ ошибок

Проверить ОТСУТСТВИЕ ошибок – нельзя!

Тестирование – исполнение программы на наборе тестов на компьютере.

Отладка включает тестирование.

Принципиальная трудность проектирования тестов состоит в практической невозможности составления всех тестовых наборов данных для всех возможных режимов работы алгоритма (для исчерпывающего тестирования). Поэтому задача проектирования тестов сводится к проектированию ограниченного набора, гарантирующего с достаточной достоверностью правильную работу программы во всех практически значимых режимах.

Содержание тестов определяется спецификацией задачи и логикой ее решения.

Функциональные тесты составляются на уровне внешней спецификации, до решения задачи. Будущий алгоритм рассматривается как **«черный ящик»** – функция с неизвестной структурой, преобразующая входы в выходы. Суть функциональных тестов: каким бы способом ни решалась задача, при заданных входных значениях должны получиться соответствующие выходные значения.

Структурные тесты составляются для проверки логики работы уже готового конкретного алгоритма. Логика определяется последовательностью операций, их условным выполнением (ветвление) или повторением (циклы). Совокупность структурных тестов должна обеспечить проверку каждой из таких конструкций. Также к этим требованиям обычно добавляется необходимость выполнения каждого из операторов программы хотя бы один раз для хотя бы одного тестового набора.

Чаще всего совокупность тщательно составленных функциональных тестов покрывает большую часть структурных тестов.

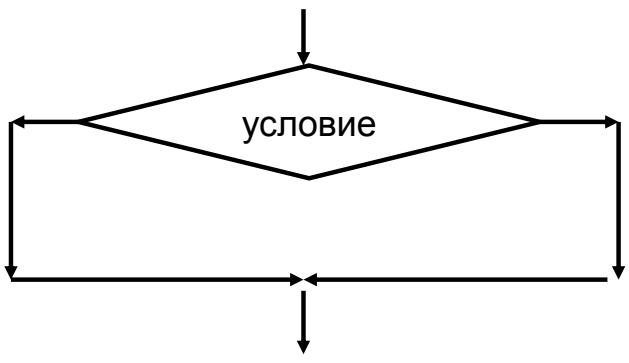
Тесты

Функциональные

- Исходные данные (диапазоны)
- Результаты в том числе альтернативные решения
- Программа как «черный ящик» - что внутри не известно, но есть входы и выходы, и известна функциональность (что должна делать)

Структурные

- Есть алгоритм и код
- Тестируются управляющие структуры
- По каждому пути на графе (блок-схеме) от входа до выхода требуется пройти не менее 1 раза.



Помимо тестирования часто применяются и **другие способы локализации ошибок**:

- Вывод сообщений и промежуточных значений на всех этапах выполнения программы;
- Точки останова (в главном меню *Run* → *Add-Breakpoints*);
- Трассировка (пошаговое выполнение + отслеживание значений переменных): пошаговое выполнение: *F7* (*Run* → *Trace-Into*), *F8* (*Run* → *Step-Over*), *F4* (*Run* → *Run-to-Cursor*) и отслеживание значений переменных *Run* →*Add-Watch;*
- Мысленный анализ кода (дедукция, индукция, обратный ход)

Длительность и сложность данного этапа полностью определяется качеством выполнения предыдущих этапов.

5. ***Документирование***. Для удобства описания представлено как некоторый этап, но по сути таковым не является.

Элементы документации – все элементы проекта, полученные при его разработке, т.е. описание результатов отдельных этапов, Если процесс разработки программы организован правильно, то документация появляется постепенно и процесс ее создания сопровождает все этапы. Поэтому стрелка-окружность на рис. 11.1 проходит через все этапы разработки программы. Кроме того, очень полезно сохранять все версии программы, для возможности их анализа и даже возврата к более ранним версиям.

Критерии качества программного обеспечения

В разных источниках могут приводиться различные наборы критериев. Но, несмотря на это, как правило, эти наборы сводимы один к другому.

1. **Работоспособность** (при выполнении определенных программных и аппаратных требований).

2. **Правильность** (программа должна решать именно поставленную, а не более широкую, более узкую или измененную задачу).

3. **Надежность** (программа должна работать при любых значениях исходных данных – анализировать их правильность и выдавать результаты или как можно более подробные сообщения об обнаруженных ошибках).

4. **Легкость отладки и тестирования** (определяется по большей части следующими тремя критериями)

5. **Читабельность** (текст программы – итоговый носитель всей информации о решении, поэтому он должен иметь четкую и ясную организацию, отображающую решение: используйте отступы при вложении управляющих структур языка, «говорящие» имена констант, переменных, типов и процедур, комментарии).

6. **Модифицируемость** (возможность внесения изменений в программу без глобальной ее переделки. Не совмещайте алгоритмы в один запутанный «клубок змей»: вы потратите свое время сначала создание и отладку более сложного алгоритма вместе нескольких простых, а затем, возможно, – на переделку).

7. **Документированность** (наличие документации по всему процессу разработки, начиная от постановки задачи до отзывов пользователей при сопровождении); документированность обеспечивает возможность передачи программы другим лицам для продолжения ее разработки и доработки.

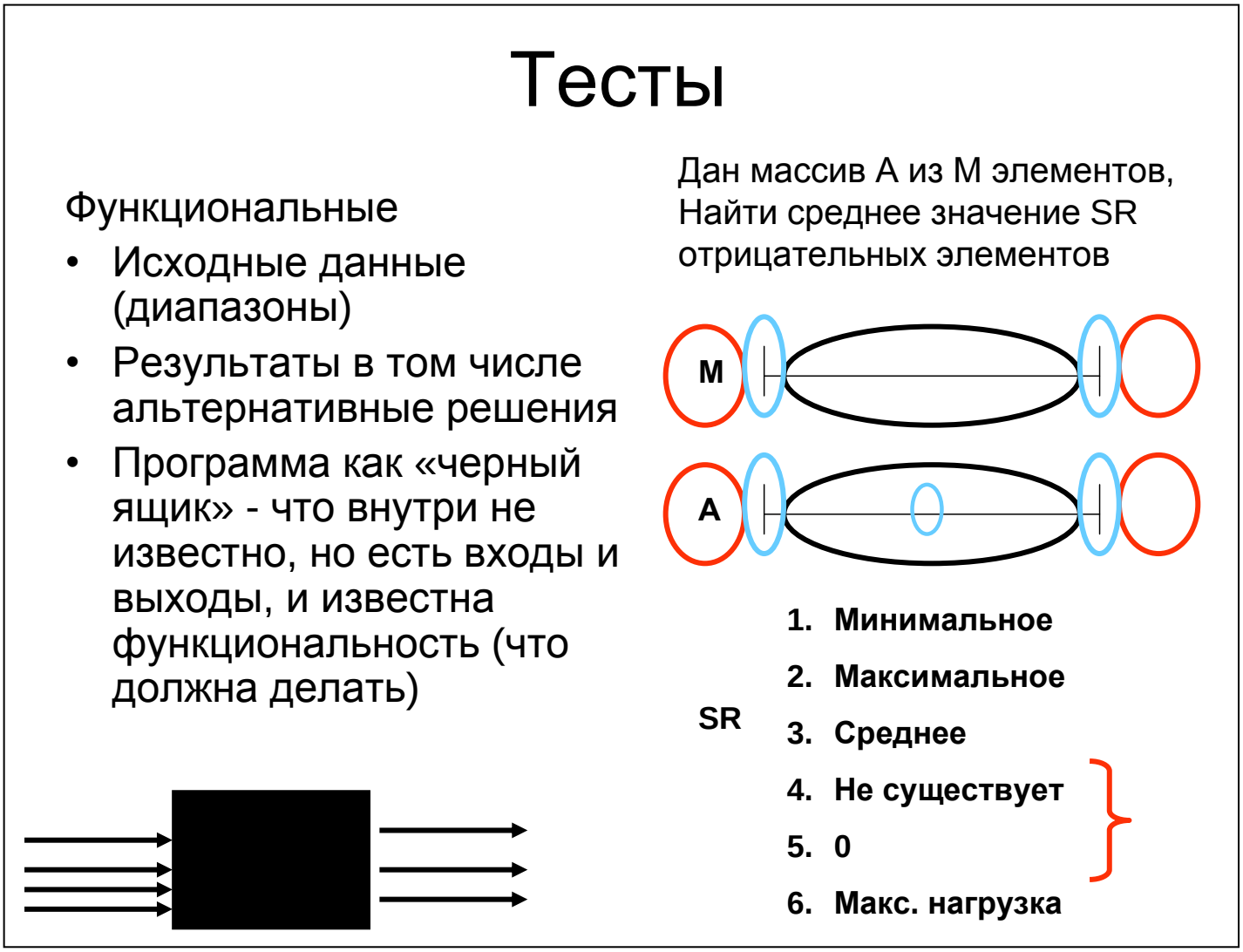
8. **Простота пользования**, наличие сервиса (дружелюбный интерфейс, справочный файл, инструкции для установки и использования, телефон/адрес службы сопровождения)

9. **Эффективность** применительно к компьютеру (использование минимума машинных ресурсов – памяти и времени выполнения).

Критерии 5, 6, 7 взаимосвязаны, критерий 4 ими определяется; критерии 3 – 8 противоречат 9-му, и сам он внутренне противоречив и не актуален без 1 и 2.

Пример составления функциональных тестов

Для составления функциональных тестов достаточно уточнить и проанализировать условие задачи. Например, рассмотрим следующую задачу:



Доопределим типы и диапазоны исходных данных, и диапазоны значений результата, и учтём получение отрицательного альтернативного решения при отсутствии отрицательных значений в массиве, и невозможность $SR=0$, что дает возможность использовать это значение как флаг для альтернативного решения.

Затем составим тестовые набора таким образом, чтобы попасть в каждую из указанных областей (области эквивалентности) для исходных данных и в перечень пунктов для результата, как минимум по одному разу, стараясь при этом минимизировать общее количество тестов:

Функциональные

1. M=0 нет ввода A, нет решения
2. M=21 нет ввода A, нет решения
3. M=1; A=(10) ➔ нет среднего
4. M=20; A=(-100,..,-100) ➔ SR=-100 (минимальное)
5. M=3; A=(100, 0, 100) ➔ нет среднего
6. M=4; A=(-50, -20, 3, -3) ➔ SR=(-50-20-3)/3=-24,3(3)
7. M=3; A=(20,-101, -7) стоп
8. M=3; A=(101, 100, -8) стоп
9. M=3; A=(0,-1,-1) ➔ SR=-1 (максимальное)

Дан целочисленный массив A из M (1..20) элементов (-100..+100), Найти среднее значение SR отрицательных элементов.

Создание структурных тестов

При решении задачи из выше приведенного примера наверняка будет использована структура выбора для проверки диапазона M, например, такая:

```
If (M>0) and (M<=20) Then  {1}
```

или такая

```
If (M>=1) and (M<=20) Then {2}
```

или такая

```
If (M>0) and (M<21) Then   {3}
```

или такая

```
If (M>=1) and (M<21) Then  {4}
```

В зависимости от конкретного кода программы структурные тесты будут разными. Например (см. рисунок далее), для первого из приведенных случаев надо создать как минимум пять тестов (два с положительным исходом и три с отрицательным, еще четыре случая невозможны), и поскольку в коде было использовано сравнение с 0, то помимо вышеперечисленных функциональных тестов нужен тест с $M<0$. Для второго варианта строки новых тестов не понадобится, а для третьего и четвертого потребуются тесты с $M>21$, и опять же $M<0$ для третьего.

Аналогично проверяются итерационные циклы ПОКА и ДО.

Тесты – условный переход -

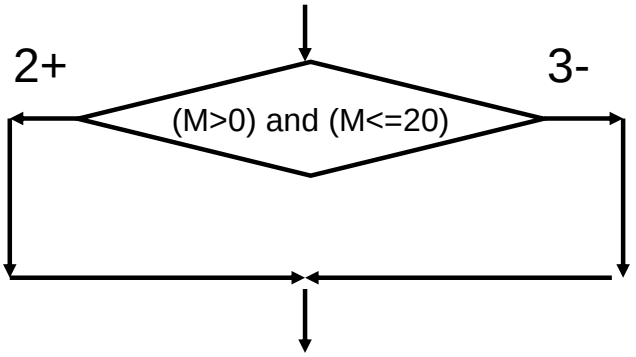
ветвление и итерационные циклы

(M>0) and (M<=20)

1. **True and True**
   1. **(M>0) and (M<20)**
   2. **(M>0) and (M=20)**
2. **True and False**
   1. **(M>0) and (M>20)**
3. **False and True**
   1. **(M<0) and (M<20)**
   2. **(M<0) and (M=20)** ——
   3. **(M=0) and (M<20)**
   4. **(M=0) and (M=20)** ——
4. **False and False**
   1. **(M<0) and (M>20)** ——
   2. **(M=0) and (M>20)** ——

Структурные

- Есть алгоритм и код
- Тестируются управляющие структуры (ветвление, циклы)
- По каждому пути на графе (блок-схеме) от входа до выхода требуется пройти не менее 1 раза.



Для проверки работы каждого цикла ДЛЯ надо провести как минимум три теста, если они возможны: 1) цикл не выполняется вовсе, то есть условие выполнения цикла сразу ложно; 2) цикл выполняется только один раз; 3) есть повторение выполнения тела цикла. Например, для цикла из алгоритма поиска максимального значения в одномерном массиве:

Поиск максимального значения Max в одномерном массиве A из N элементов	Смысл	Тесты

`Max:=A[1];`
`For i:=2 to N do`
`  If A[i]>Max then`
`  Max:=A[i];`
	цикл не выполняется вовсе	N=1; A=(-100) → Max=-100

	один раз (N=2)	N=2; A=(-100,-3) → Max=-3
	несколько раз (N>2)	N=4; A=(2,2,3,1) → Max=3

В этом алгоритме также присутствует оператор выбора, который также должен быть проверен структурными тестами, поэтому в третий из приведенных тестов включены все три возможных случая: A[2]=Max, A[3]>Max и A[4]<Max.

Хорошим и удачным называют тест, который *выявляет* ошибку!

Чтобы составить хорошие тесты надо пытаться ее найти, и верить, что она есть. Не стоит составлять тесты с убеждением, что ваша программа – идеальна.