

# Верификация и валидация

Алексей Островский

Физико-технический учебно-научный центр НАН Украины

12 марта 2015 г.

# Верификация и валидация

## Определение

**Верификация** (англ. *verification*) — процессы проверки соответствия программного продукта требованиям и спецификациям, заданным при его проектировании.

## Определение

**Валидация** (англ. *validation*) — процессы проверки соответствия программного продукта потребностям заказчика; проверка корректности спецификаций.

	Верификация	Валидация
<b>Цель</b>	Правильно ли строится ПП?	Строится ли правильный ПП?
<b>Точка зрения</b>	разработчик (белый ящик)	конечный пользователь (черный ящик)
<b>Методы</b>	статические	динамические (тестирование)

# Методы верификации

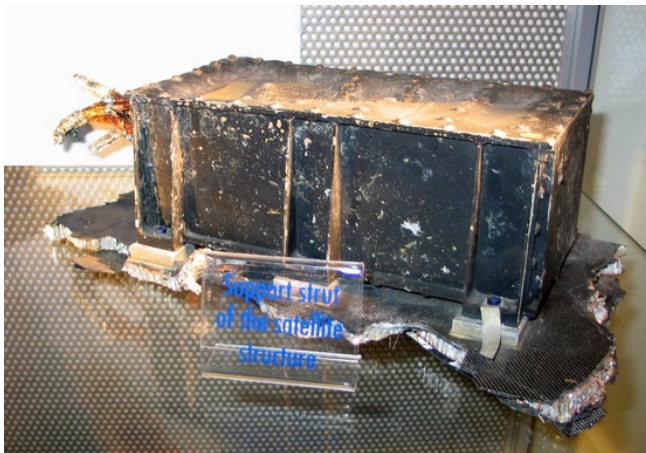
## Неформальные методы:

- ▶ инспекции и обзоры кода;
- ▶ инструменты для автоматического поиска ошибок в коде.

## Формальные методы верификации:

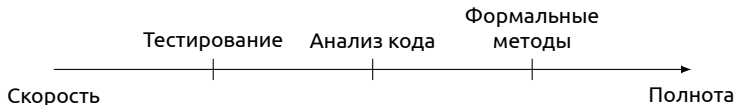
- ▶ проверка моделей ПО (англ. *model checking*);
- ▶ (автоматизированные) доказательства корректности программ;
- ▶ абстрактная интерпретация кода;
- ▶ системы типов данных.

# Необходимость верификации



Фрагмент спутника «[Ariane 5](#)», потерпевшего крушение в 1996 г. Причиной аварии стало преобразование из 64-битного числа с плавающей запятой в 16-битное целое без должной проверки, что привело к переполнению и аппаратному исключению. Цена ошибки составила ~350 млн. \$.

# Необходимость верификации



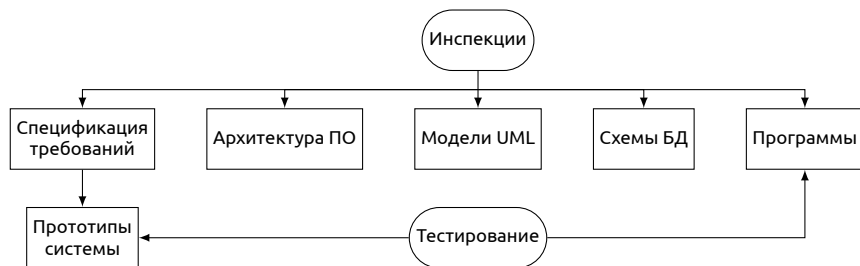
## Область применения верификации:

- ▶ критические системы, системы реального времени (напр., системы управления спутниками и ракетами);
- ▶ системы с высокими требованиями к надежности / отказоустойчивости (напр., системы шифрования);
- ▶ ПО, регулируемое нормативными документами (напр., ПО, использующееся в медицине).

# Инспекции

## Определение

**Инспекции кода** (англ. *code inspections*) — анализ кода программы или ее абстрактного представления (напр., UML диаграммы) человеком на предмет наличия ошибок.



Возможности инспекций шире, чем тестирования

## Преимущества инспекций

- ▶ Отсутствие взаимодействия между дефектами (в отличие от тестирования, где одна ошибка может скрывать другие).
- ▶ Возможность проверки неполных версий системы без дополнительных затрат.
- ▶ Проверка свойств программы (соответствие стандартам, портируемость, легкость поддержки).
- ▶ Выявление неэффективных / неподходящих алгоритмов.

## Области проверки кода

### ▶ Дефекты данных:

- ▶ инициализация переменных перед использованием;
- ▶ отсутствие «магических» констант;
- ▶ индексация элементов массивов / списков;
- ▶ потенциальные случаи переполнения буфера.

### ▶ Дефекты выполнения:

- ▶ проверка условий ветвления;
- ▶ конечность циклов;
- ▶ корректность блоков операций;
- ▶ полнота вариантов и наличие **break** в операторе **switch / case**.

### ▶ Дефекты ввода/вывода:

- ▶ использование всех входных переменных;
- ▶ присвоение выходных переменных.



## Области проверки кода

### ▶ Дефекты интерфейсов:

- ▶ корректное количество и порядок аргументов при вызове функций / методов;
- ▶ соответствие ожидаемых и фактических типов аргументов;
- ▶ идентичность структуры разделяемой памяти.

### ▶ Дефекты работы с памятью:

- ▶ корректность работы со связанными объектами (изменение всех требуемых ссылок);
- ▶ корректность выделения / освобождения памяти.

### ▶ Дефекты обработки исключений:

- ▶ полнота проверок возникновения исключительных ситуаций.

# Парное программирование

## Определение

**Парное программирование** (англ. *pair programming*) — способ разработки ПО, при котором код, написанный первым программистом, немедленно проверяется вторым программистом; альтернатива формальным инспекциям в гибкой методологии разработки.

**Достоинства:** более глубокое понимание кода — обнаружение большего числа дефектов.

**Недостатки:**

- ▶ повышенный шанс непонимания требований;
- ▶ пропуск ошибок из-за высокого темпа разработки;
- ▶ необъективность инспектора.

# Автоматизация инспекций

## Инструменты автоматизации инспекций:

- ▶ **среды разработки** — встроенные средства или плагины для проведения инспекций / неформальных обзоров кода;
- ▶ **системы контроля версий** (Subversion, git, ...) — упрощение доступа к коду, запросы на изменение / уточнение фрагментов программ;
- ▶ **Lint** — семейство программ статического анализа кода для:
  - ▶ поиска типичных ошибок,
  - ▶ устранения неपोर्टируемых / потенциально опасных конструкций;
  - ▶ соблюдения принятого стиля написания программного кода.

# Формальные методы

## Определение

**Формальные методы** в разработке ПО — методы спецификации, разработки и верификации ПО на основе строгих математических моделей.

### Уровни использования:

1. создание формальной модели;
2. разработка и верификация на основе мат. моделей;
3. формальное доказательство свойств ПО.

## Типы формальных методов

Формальный метод  $\simeq$  семантика ЯП.

- ▶ **Денотационная семантика** — подход к программе как функции, преобразующей входные данные в выходные.  
**Примеры:** используется в академической среде (функциональные ЯП).
- ▶ **Аксиоматическая семантика** — подход к программе как к набору пред- и постусловий для каждого выполненного действия.  
**Примеры:** логика Хоара, языки спецификации.
- ▶ **Операционная семантика** — подход к программе как последовательности действий в определенной модели.  
**Примеры:** проверка моделей, абстрактная интерпретация, символьное выполнение.

# Логика Хоара

## Определение

**Функциональная корректность алгоритма** — соответствие выходных данных ожидаемым для каждой комбинации входных данных.

## Определение

**Полная корректность** — алгоритм корректен и выполняется за конечное время для любого входа.

## Определение

**Логика Хоара** (англ. *Hoare logic*) — набор правил вывода для доказательства функциональной корректности компьютерных программ, записанных с помощью императивного языка программирования.

(Существует расширение логики Хоара для доказательства полной корректности.)

# Правила вывода в логике Хоара

## Утверждения — тройки Хоара:

$$\{P\} C \{Q\};$$

$C$  — команда языка программирования;

$P$  — предусловие (предикат, истинный до выполнения команды  $C$ );

$Q$  — постусловие (предикат, истинный после выполнения команды  $C$ ).

## Правила вывода (общий вид):

$$T_1 \vdash T_2;$$

- ▶ из утверждений  $T_1$  выводятся утверждения  $T_2$ ;
- ▶  $(\Leftrightarrow)$  для доказательства  $T_2$  достаточно доказать  $T_1$ .

# Правила вывода в логике Хоара

1. **Аксиома пропуска:**  $\vdash \{P\} \text{ skip } \{P\}.$

2. **Аксиома присвоения:**  $\vdash \{P[E/x]\} x := E \{P\},$

$P[E/x]$  — замена всех вхождений выражения  $x$  в предикате  $P$  на  $E$ .

**Пример:**  $\vdash \{x + 1 \leq 5\} x := x + 1 \{x \leq 5\}.$

3. **Правило композиции:**  $\{P\} S \{Q\}, \{Q\} T \{R\} \vdash \{P\} S; T \{R\}.$

**Пример:**

$$\{x \leq 4\} x := x + 1 \{x \leq 5\}, \{x \leq 5\} y := x \{y \leq 5\} \vdash$$

$$\vdash \{x \leq 4\} x := x + 1; y := x \{y \leq 5\}.$$



## Правила вывода в логике Хоара

### 4. Правило ветвления:

$$\{B \wedge P\} S \{Q\}, \{\neg B \wedge P\} T \{Q\} \vdash \{P\} \text{ if } B \text{ then } S \text{ else } T \{Q\}.$$

### 5. Правило следствия: $P_1 \Rightarrow P_2, \{P_2\} S \{Q_2\}, Q_2 \Rightarrow Q_1 \vdash \{P_1\} S \{Q_1\}.$

(Можно усиливать предусловие и/или ослаблять постусловие.)

**Пример:** Доказать  $\{x \leq 10\} \text{ if } x < 10 \text{ then } x := x + 1 \{x \leq 10\}$ , если  $x \in \mathbb{Z}$ .

1. По правилу ветвления нужно доказать: (1)  $\{x < 10\} x := x + 1 \{x \leq 10\}$ ;  
(2)  $\{x = 10\} \text{ skip } \{x \leq 10\}$ .
2. По аксиоме присвоения  $\{x + 1 \leq 10\} x := x + 1 \{x \leq 10\}$ , что эквивалентно (1).
3. По правилу следствия  $\{x \leq 10\} \text{ skip } \{x \leq 10\}, (x = 10) \Rightarrow (x \leq 10) \vdash$  (2).

## Правила вывода в логике Хоара

6. **Правило цикла:**  $\{P \wedge B\} S \{P\} \vdash \{P\} \textbf{while } B \textbf{ do } S \{\neg B \wedge P\}.$

**Пример:** Доказать  $\{x \leq 10\} \textbf{while } x < 10 \textbf{ do } x := x + 1 \{x = 10\}$ , если  $x \in \mathbb{Z}$ .

1. Преобразуем оказуемое утверждение:

$\{x \leq 10\} \textbf{while } x < 10 \textbf{ do } x := x + 1 \{\neg(x < 10) \wedge (x \leq 10)\}.$

2. По правилу цикла надо доказать  $\{(x \leq 10) \wedge (x < 10)\} x := x + 1 \{x \leq 10\}.$

3.  $\{x < 10\} x := x + 1 \{x \leq 10\}$  — по правилу присвоения.

Конструкции **for**, **switch / case**, **do ... while** сводятся к правилу ветвления или цикла.

# Языки спецификации

## Определение

**Язык спецификации** (англ. *specification language*) — формальный язык для описания программной системы на более высоком уровне, чем язык ее реализации.

**Инструменты:** логика первого порядка, теория множеств.

**Примеры языков спецификации:**

- ▶ VDM (Vienna development method);
- ▶ Z;
- ▶ Java Modeling Language (расширение Java);
- ▶ Спеc# (расширение C#).

## Применение языков спецификации

- ▶ Упрощение анализа требований; строгая запись требований к программной системе.
- ▶ Проектирование — определение модели отдельных компонентов ПО и взаимодействий между ними.
- ▶ Автоматизированная генерация кода.
- ▶ Детализация поведения программы, невозможная при помощи ЯП.
- ▶ Доказательство корректности с помощью программных инструментов.
- ▶ Автоматизированное создание тестовых вариантов.

## Пример спецификации

### Спецификация извлечения квадратного корня на языке JML:

```
1  public class SquareRoot {
2
3      /**
4       * Точность вычисления корня.
5       */
6      private static final double EPS = 1e-4;
7
8      // Предусловие
9      /*@ requires x >= 0.0; @*/
10     // Постусловие
11     /*@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, EPS);
12         @*/
13     public static double sqrt(double x) {
14         // ...
15     }
16 }
```

# Проверка моделей

## Определение

**Проверка моделей** (англ. *model checking*) — алгоритмическая проверка модели программной системы на соответствие заданной спецификации.

### Области проверки:

- ▶ параллельное выполнение — отсутствие тупиков (англ. *deadlock*) и состояний гонки (англ. *race condition*);
- ▶ проверка ошибок времени выполнения;
- ▶ проверка пользовательских интерфейсов;
- ▶ низкоуровневый анализ кода программы.

## Математический аппарат проверки моделей

- ▶ **Конечные автоматы** (англ. *finite state machine*) — строятся для представления взаимоотношений между состояниями, в которых может находиться приложение ( $\simeq$  направленный граф).  
**Использование:** анализ хода выполнения программы; перебор всех способов выполнения.
- ▶ **Символьное выполнение** (англ. *symbolic execution*) — аппроксимация семантики ЯП с помощью символьных вычислений.  
**Использование:** определение входов, приводящих к заданному порядку выполнения операций.
- ▶ **Абстрактная интерпретация** (англ. *abstract interpretation*) — аппроксимация семантики ЯП с помощью монотонных функций на упорядоченных множествах.  
**Использование:** определение возможных значений численных переменных и корректности операций над ними.

# Java Pathfinder

**Java Pathfinder (JPF)** — система для верификации скомпилированных Java-приложений, разработанная в NASA.

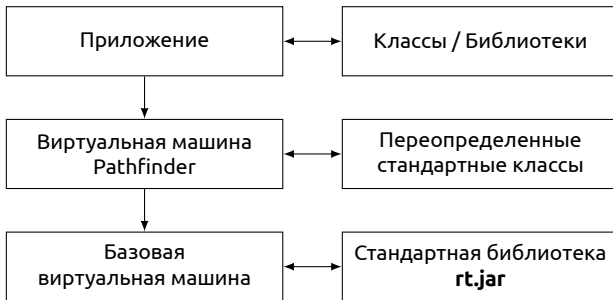


Схема выполнения приложения в Java Pathfinder.



## Пример: анализ состояния гонки

### Гонка, возникающая из-за возможности выполнения (1) перед (2)

```
1  public class Racer implements Runnable {
2      private int d = 42;
3
4      public void run() {
5          try { Thread.sleep(1001); } catch (InterruptedException e) {}
6          d = 0;                                // (1)
7      }
8
9      public static void main(String[] args) {
10         Racer racer = new Racer();
11         Thread t = new Thread(racer);
12         t.start();
13
14         try { Thread.sleep(1000); } catch (InterruptedException e) {}
15         int c = 420 / racer.d;                // (2)
16         System.out.println(c);
17     }
18 }
```

## Пример: анализ состояния гонки

1. JPF создает конечный автомат для всевозможных состояний программы.
2. Выполняются все варианты взаимодействия между двумя потоками выполнения.
3. Обнаруживается ошибка:

```
1  ===== error #1
2  gov.nasa.jpf.listener.PreciseRaceDetector
3  race for field Racer@13d.d
4    main at Racer.main(Racer.java:15)
5      "int c = 420 / racer.d;" : getfield
6    Thread-0 at Racer.run(Racer.java:6)
7      "d = 0;" : putfield
```

## Выводы

1. Верификация и валидация — дополняющие друг друга процессы проверки корректности ПО на соответствие спецификации и ожиданиям заказчика соответственно. Тестирование — частный случай валидации.
2. Методы верификации ПО делятся на неформальные (инспекции кода) и формальные (логика Хоара, проверка моделей).
3. Цель инспектирования кода — устранение часто допускаемых ошибок и повышение легкости сопровождения ПО.
4. Формальные методы применяются для строгого доказательства корректности программ и обнаружения «редких» ошибок. Т. к. затраты на формальные методы верификации высоки, областью их применения является критическое ПО.

# Материалы



**Sommerville, Ian**

Software Engineering.

Pearson, 2011. — 790 p.



**Лавріщева К. М.**

Програмна інженерія (підручник).

К., 2008. — 319 с.

Спасибо за внимание!