Asserting Expectations

Andreas Zeller

1

Your Submissions

- Program must behave exactly as specified (i.e., input, output, flags, etc.)
- Program must use recent Python 2 version (i.e., Python 2.6 – installed in CIP pools)
- If *we* have to fix your programs, *you* will lose points

Oral Exams

- Some time after the course (dates set up individually, in two clusters)

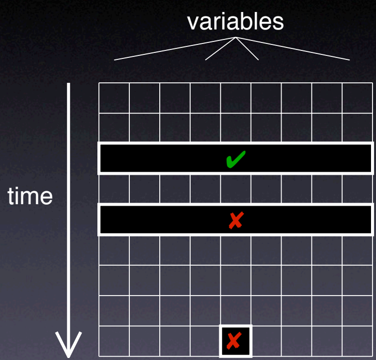
Asserting Expectations

Andreas Zeller

4

Search in Time

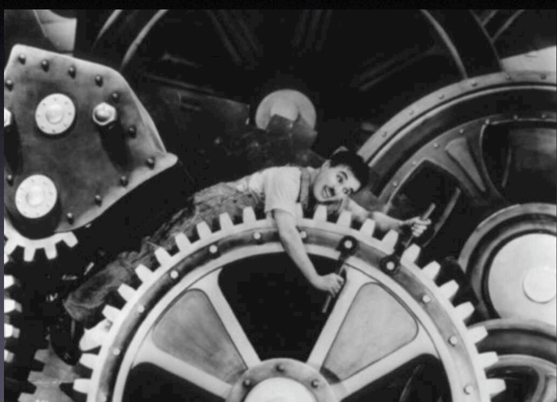
- During execution, the state becomes infected.
- Basic idea: Observe a *transition* from sane to infected.



5

5

Manual Observation



6

6

Automated Observation

Automated Observation

what to observe	when to observe	what to expect

Basic Assertions

```
if (divisor == 0) {
    printf("Division by zero!");
    abort();
}
```

Specific Assertions

```
assert (divisor != 0);
```


Implementation

```
void assert (int x)
{
  if (!x)
  {
    printf("Assertion failed!\n");
    abort();
  }
}
```


Execution

```
$ my-program
Assertion failed!
Abort (core dumped)
$
```

Better Diagnostics

```
$ my-program
divide.c:37:
   assertion 'divisor != 0' failed
Abort (core dumped)
$ _
```

13

13

Assertions as Macros

```
#ifndef NDEBUG
#define assert(ex) \
((ex) ? 1 : (cerr << __FILE__ << ":" << __LINE__ \
            << ": assertion '" #ex "' failed\n", \
               abort(), 0))
#else
#define assert(x) ((void) 0)
#endif
```

14

14

Automated Observation

what to observe	when to observe	what to expect
state checked in assertion	location of assertion	checked property of program state

15

15

When to observe

- Data invariants
- Pre- and postconditions

16

16

Asserting Invariants

17

17

A Time Class

```
class Time {
public:
    int hour();     // 0..23
    int minutes();  // 0..59
    int seconds();  // 0..60 (incl. leap seconds)

    void set_hour(int h);
    …
}
```

Any time from 00:00:00 to 23:59:60 is valid

18

18

Ensuring Sanity

```
void Time::set_hour(int h)
{
    // precondition
    assert (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
    …
    // postcondition
    assert (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```


Ensuring Sanity

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane());   // precondition
    …
    assert (sane());   // postcondition
}
```


Ensuring Sanity

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

sane() is the *invariant* of a Time object:

- holds *before* every public method
- holds *after* every public method

Ensuring Sanity

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane());
    …
    assert (sane());
}
```

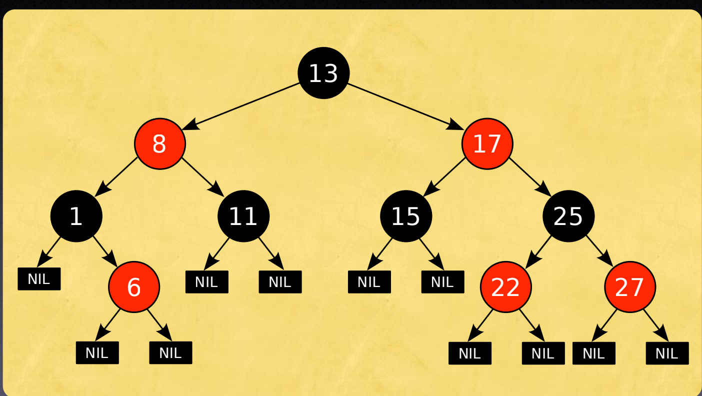
same for set_minute(), set_seconds(), etc.

Locating Infections

- Precondition failure = infection *before* method
- Postcondition failure = infection *within* method
- All assertions pass = no infection

```
void Time::set_hour(int h)
{
    assert (sane());   // precondition
    …
    assert (sane());   // postcondition
}
```


Complex Invariants



http://en.wikipedia.org/wiki/Red-black_tree

Complex Invariants

```
class RedBlackTree {
  …
  boolean sane() {
      assert (rootHasNoParent());
      assert (rootIsBlack());
      assert (redNodesHaveOnlyBlackChildren());
      assert (equalNumberOfBlackNodesOnSubtrees());
      assert (treeIsAcyclic());
      assert (parentsAreConsistent());

      return true;
  }
}
```


Deletion

```
void
delete_one_child(struct node *p)
{
    /*
     * Precondition: p has at most one non-null child.
     */
    struct node *child = is_leaf(p->right) ? p->left : p->right;
    replace_node(p, child);
    if (p->color == BLACK) {
        if (child->color == RED)
            child->color = BLACK;
        else {
            /* delete_case1(child): */
            struct node *n = child;

            /* this loop performs tail recursion on delete_case1(n) */
            for (;;) {
                /* delete_case1(n): */
                if (n->parent != NULL) {
                    /* delete_case2(n): */

                    struct node *s;
                    s = sibling(n);
                    if (s->color == RED) {
                        n->parent->color = RED;
```

http://en.wikipedia.org/wiki/Red-black_tree

Invariants as Aspects

```
public aspect RedBlackTreeSanity {
    pointcut modify():
        call(void RedBlackTree.add*(..)) ||
        call(void RedBlackTree.del*(..));

    before(): modify() {
        assert (sane());
    }

    after(): modify() {
        assert (sane());
    }
}
```

Invariants in GDB

```
(gdb) break 'Time::set_hour(int)' if !sane()
Breakpoint 3 at 0x2dcf: file Time.C, line 45.
(gdb) _
```

28

28

Asserting Correctness

29

29

Postconditions

```
def divide(dividend, divisor):
    # Actual computation goes here
    …
    assert quotient * divisor + remainder == dividend
    return (quotient, remainder)
```

30

30

Postconditions

```
void Time::set_hour(int h)
{
   // Actual code goes here

   assert (hour() == h);    // postcondition
}
```

31

31

Postconditions

```
void Sequence::sort()
{
    // Actual code goes here

    assert (is_sorted());
}
```

helper function

32

32

Postconditions

```
void Container::insert(Item x)
{
    // Actual code goes here

    assert (has(x));
}
```

a helper function that is also a useful public method

33

33

Postconditions

```
void Heap::merge(Heap another_heap)
{
    assert (sane());
    assert (another_heap.sane());

    // Actual code goes here

    assert (sane());
}
```

Invariants are always part of pre- and postconditions

34

Checking Earlier State

```
void Time::set_hour(int h)
{
   int old_minutes = minutes();
   int old_seconds = seconds();
   assert (sane());

   // Actual code goes here

   assert (sane());
   assert (hour() == h);
   assert (minutes() == old_minutes &&
           seconds() == old_seconds);
}
```


35

Contracts

```
set_hour (h: INTEGER) is
        -- Set the hour from `h'
    require
        sane_h: 0 <= h and h <= 23
    ensure
        hour_set: hour = h
        minute_unchanged: minutes = old minutes
        second_unchanged: seconds = old seconds
```

This *contract* specifies interface properties

36

Z Invariant

$$
\begin{array}{|l}
\textit{Date} \\
\hline
\textit{hours}, \textit{minutes}, \textit{seconds} : \mathbb{N} \\
\hline
0 \leq \textit{hours} \leq 23 \\
0 \leq \textit{minutes} \leq 59 \\
0 \leq \textit{seconds} \leq 59 \\
\hline
\end{array}
$$

seconds can be 60!

Make a point about errors in specs

37

37

Z Conditions

$$
\begin{array}{|l}
\textit{set\_hour} \\
\hline
\Delta \textit{Date} \\
h? : \mathbb{N} \\
\hline
0 \leq h? \leq 23 \\
\textit{hours}' = h? \\
\textit{minutes}' = \textit{minutes} \\
\textit{seconds}' = \textit{seconds} \\
\hline
\end{array}
$$

38

38

Spec vs Code

Contracts

```
set_hour (h: INTEGER) is
        -- Set the hour from `h'
    require
        sane_h: 0 <= h and h <= 23
    ensure
        hour_set: hour = h
        minute_unchanged: minutes = old minutes
        second_unchanged: seconds = old seconds
```

This *contract* specifies interface properties

31

Z Conditions

$$
\begin{array}{|l}
\textit{set\_hour} \\
\hline
\Delta \textit{Date} \\
h? : \mathbb{N} \\
\hline
0 \leq h? \leq 23 \\
\textit{hours}' = h? \\
\textit{minutes}' = \textit{minutes} \\
\textit{seconds}' = \textit{seconds} \\
\hline
\end{array}
$$

33

Integrated spec
limited to language

Separate spec
can express anything

39

39

JML

```
/*@ requires 0 <= h && h <= 23
  @ ensures  hours() == h &&
  @          minutes() == \old(minutes()) &&
  @          seconds() == \old(seconds())
  @*/
void Time::set_hour(int h) …
```

Translated into run-time assertions

40

Developed by Gary Leavens et al, now a cooperative effort of dozens of researchers

40

JML as Spec

```
/*@ requires x >= 0.0;
  @ ensures JMLDouble
  @         .approximatelyEqualTo
  @         (x, \result * \result, eps);
  @*/
```

What does this specify?

41

41

```
public class Purse {
  final int MAX_BALANCE;
  int balance;
  //@ invariant  0 <= balance && balance <= MAX_BALANCE;

  byte[] pin;
  /*@ invariant pin != null && pin.length == 4 &&
    @          (\forall int i; 0 <= i && i < 4;
    @                          0 <= byte[i] && byte[i] <= 9)
    @*/

  /*@ requires   amount >= 0;
    @ assignable balance;
    @ ensures    balance == \old(balance) - amount &&
    @            \result == balance;
    @ signals    (PurseException) balance == \old(balance);
    @*/
  int debit(int amount) throws PurseException { … }
```

42

42

More use of JML

- Documentation
- Unit testing with JMLUnit
- Invariant generation with DAIKON
- Static checking with ESC/Java
- Verification with theorem provers

43

43

Relative Debugging

Rather than checking a spec, we can also compare against a *reference run:*

- The environment has changed–e.g. ports or new interpreters
- The code has changed
- The program has been reimplemented

44

44

Relative Assertions

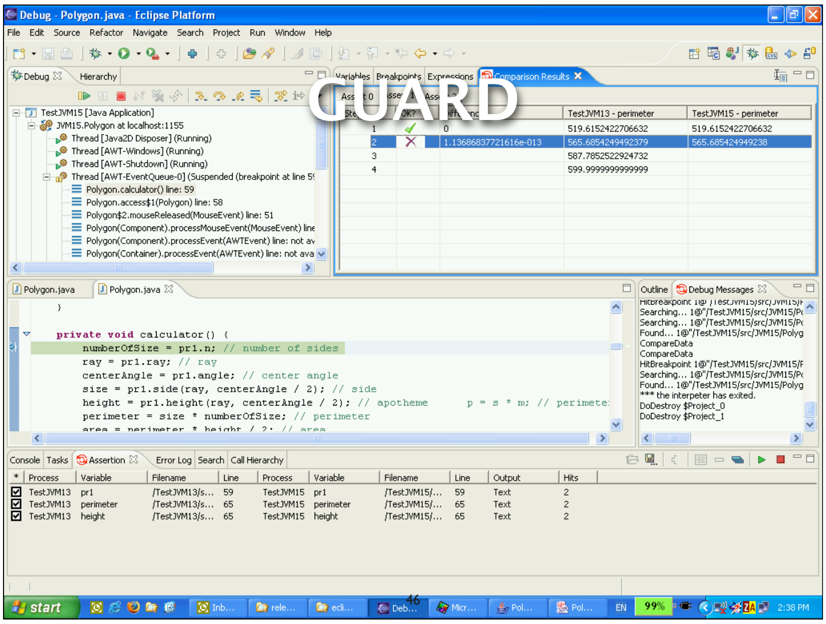
- We compare two program runs
- A *relative assertion* compares variable values across the two runs:

```
assert \
p1::perimeter@polygon.java:65 == \
p0::perimeter@polygon.java:65
```

- Specifies when and what to compare

45

45



46

Concepts

- Assertions catch infections before they propagate too far
- Assertions check preconditions, postconditions and invariants
- Assertions can serve as specifications
- A program can serve as reference to be compared against

47

48