GenUTest: A Unit Test and Mock Aspect Generation Tool

Benny Pasternak
Shmuel Tyszberowicz
Amiram Yehudai

Tel Aviv University

HVC 2007, October 25, 2007

Agenda

- Motivation
- Example
- Implementation
- Experimentation
- Conclusion

Motivation

- Assumption #1 – Unit tests are good ☺

- Assumption #2 – Writing effective unit tests is a hard and tedious process
  - At maintenance phase, writing tests from scratch is not considered cost effective ☹
  - Corollary: Maintenance remains a difficult process

- Goal: Automatically generate unit tests for projects in maintenance phase

Example

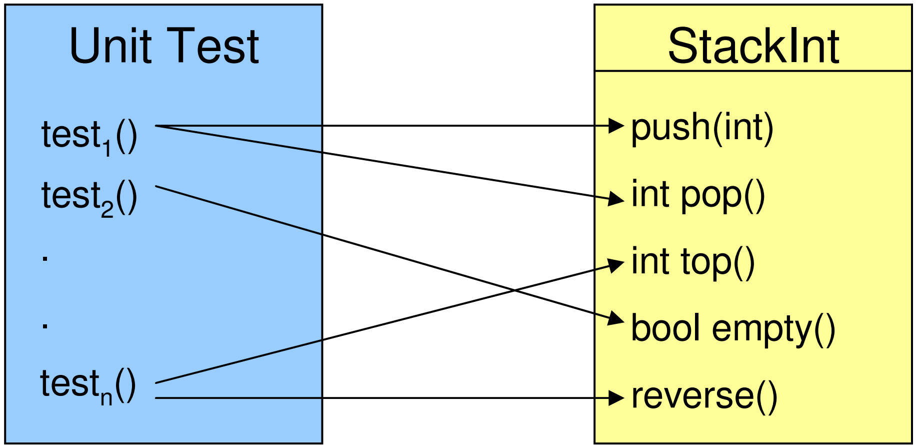
- Developers are asked to create unit tests for an existing software project

- **`StackInt`** is an implementation of integers' stack, with the operations:
  - Push
  - Pop
  - Top
  - Empty
  - Reverse

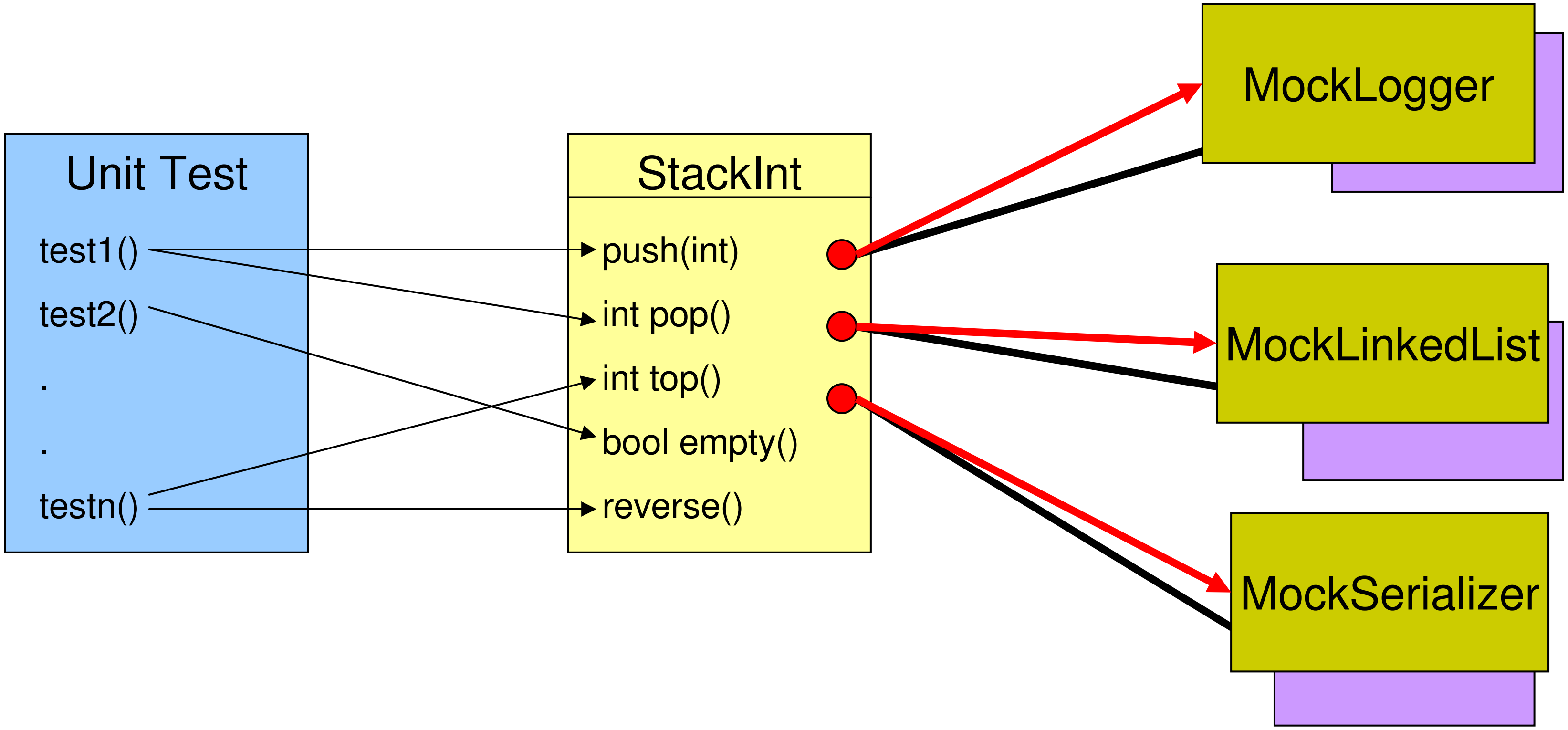
Example

- Goal is to test `StackInt` comprehensively and in isolation
- Comprehensiveness – unit test should exercise all class methods and achieve high code coverage rate
- Isolation – dependent objects (e.g., `Logger`, `Serializer`) should not be tested

Example: Comprehensiveness



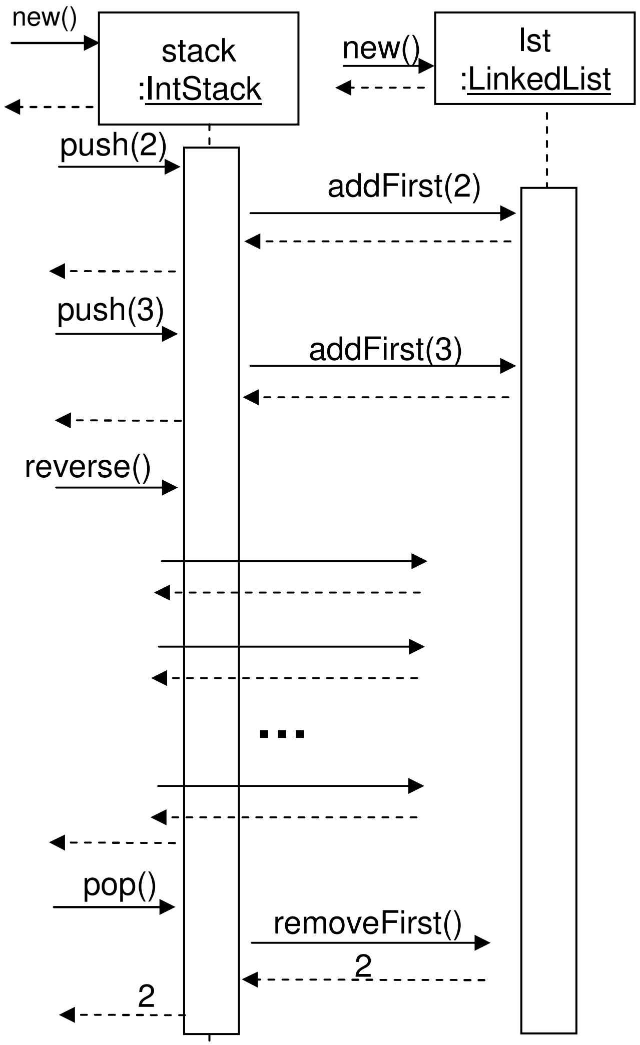
Example: Isolation



Obtaining Test Cases From Existing Tests

- **System/Module test** that exercises `IntStack` as follows:

- Test can be used to obtain test cases for unit tests



GenUTest

- Captures and records execution of **`IntStack`** during module/system tests in order to obtain test cases

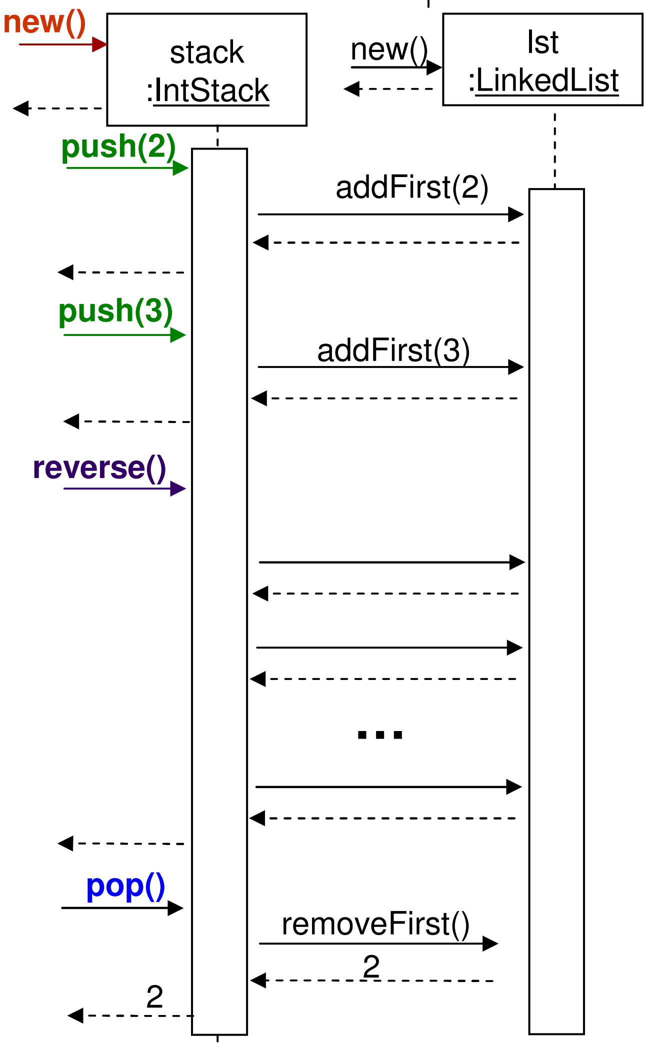
- Recorded events are used to generate unit tests for **`IntStack`**

- Unit tests assist developers in the testing process

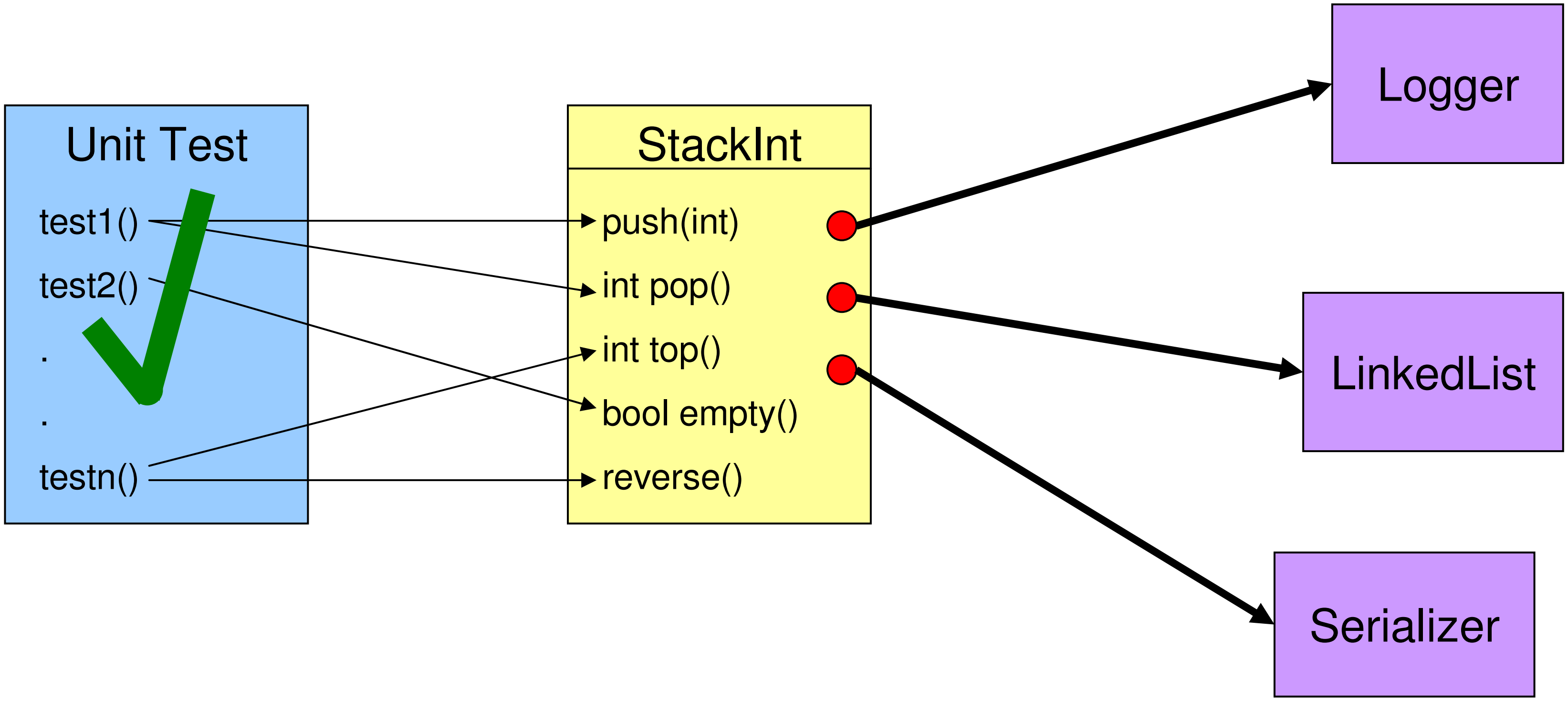
Example – Generated Unit Test

Unit Test Code

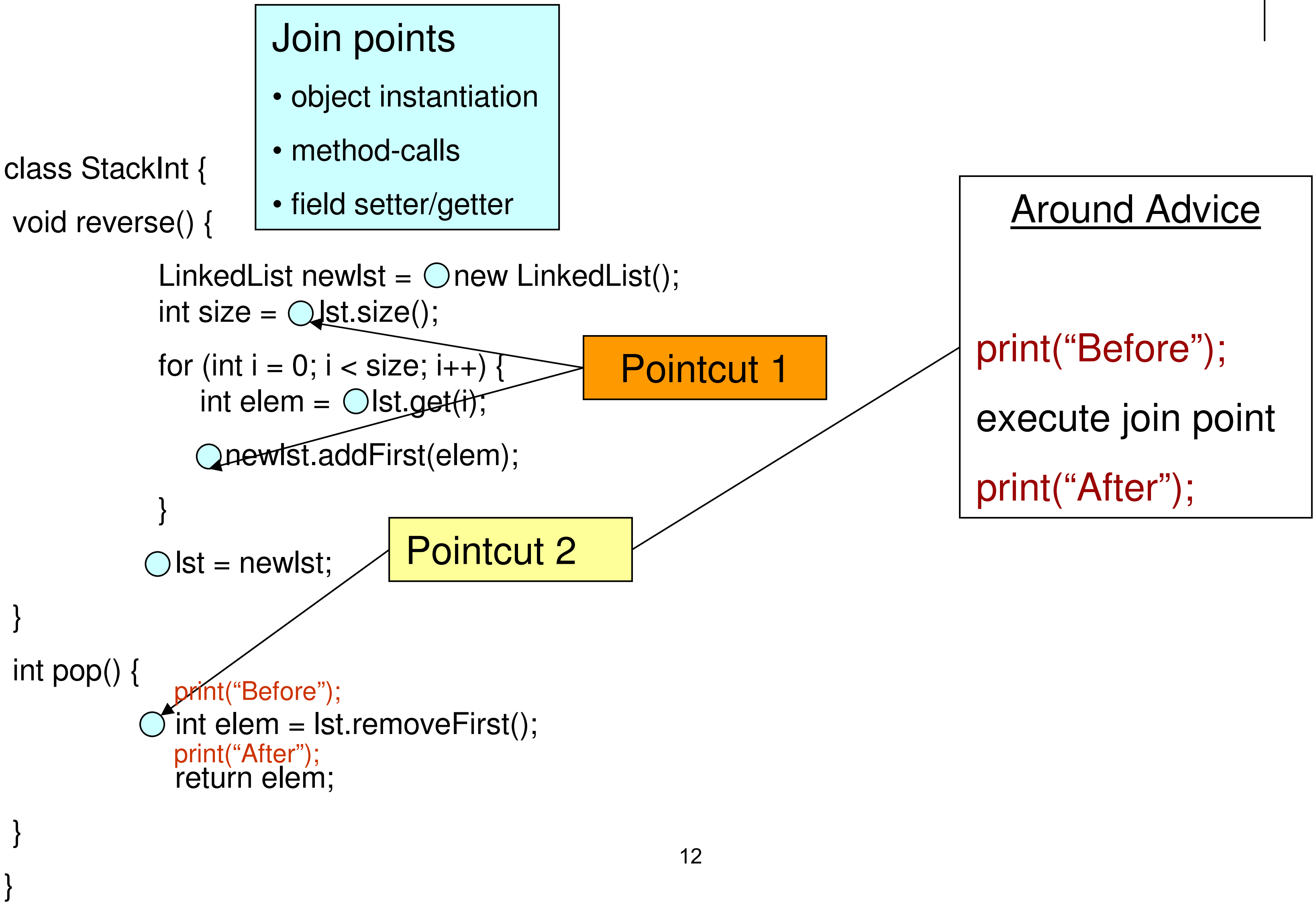
```
@Test public void testpop1()
{
    // test execution statements
    IntStack IntStack_2 = new IntStack();       // #1
    IntStack_2.push(2);                         // #2
    IntStack_2.push(3);                         // #3
    IntStack_2.reverse();                       // #4
    int intRetVal6 = IntStack_2.pop();          // #5

    // test assertion statements
   assertEquals(intRetVal6,2);
}
```



Example


Unit Test
test1()
test2()
.
.
testn()
StackInt
push(int)
int pop()
int top()
bool empty()
reverse()
Logger
LinkedList
Serializer

Aspect Oriented Programming



Join points

- object instantiation
- method-calls
- field setter/getter

```
class StackInt {
 void reverse() {
        LinkedList newlst = ○new LinkedList();
        int size = ○lst.size();
        for (int i = 0; i < size; i++) {
            int elem = ○lst.get(i);
            ○newlst.addFirst(elem);
        }
       ○lst = newlst;
 }
 int pop() {
        print("Before");
       ○int elem = lst.removeFirst();
        print("After");
        return elem;
 }
}
```

Pointcut 1

Pointcut 2

Around Advice

print("Before");

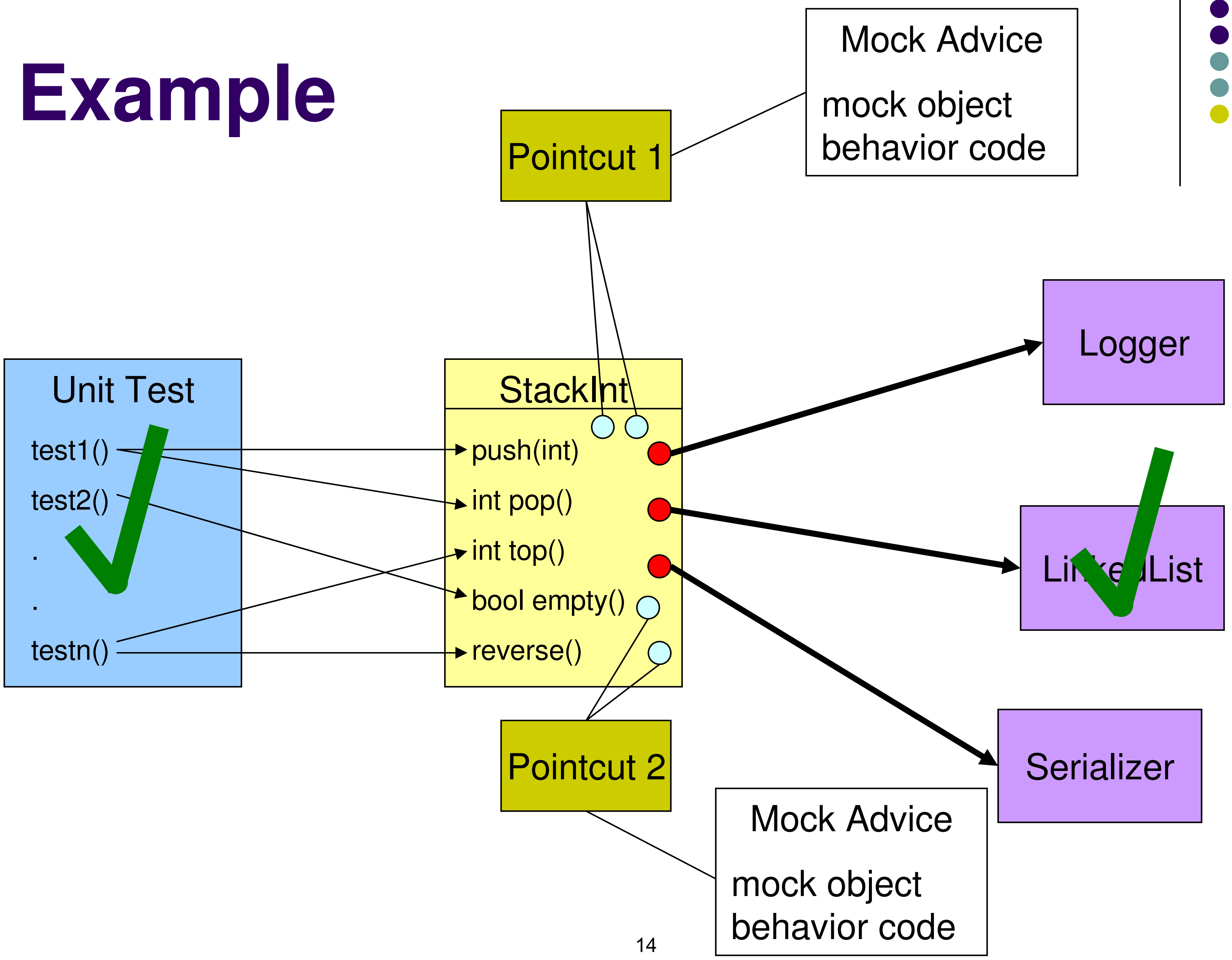
execute join point

print("After");

AOP – Quick Summary

- **Join points** – well defined execution points in the control flow of the program (object instantiation, method-calls, field member access)
- **Pointcut** – expression that specifies a set of join points
- **Advices** – code specified to execute *before, after, or around* pointcuts
- **Aspects** – The equivalent to class. Holds pointcut declarations and advices

Example


Mock Advice
mock object
behavior code
Pointcut 1
Logger
Unit Test
test1()
test2()
.
.
testn()
StackInt
push(int)
int pop()
int top()
bool empty()
reverse()
LinkedList
Serializer
Pointcut 2
Mock Advice
mock object
behavior code

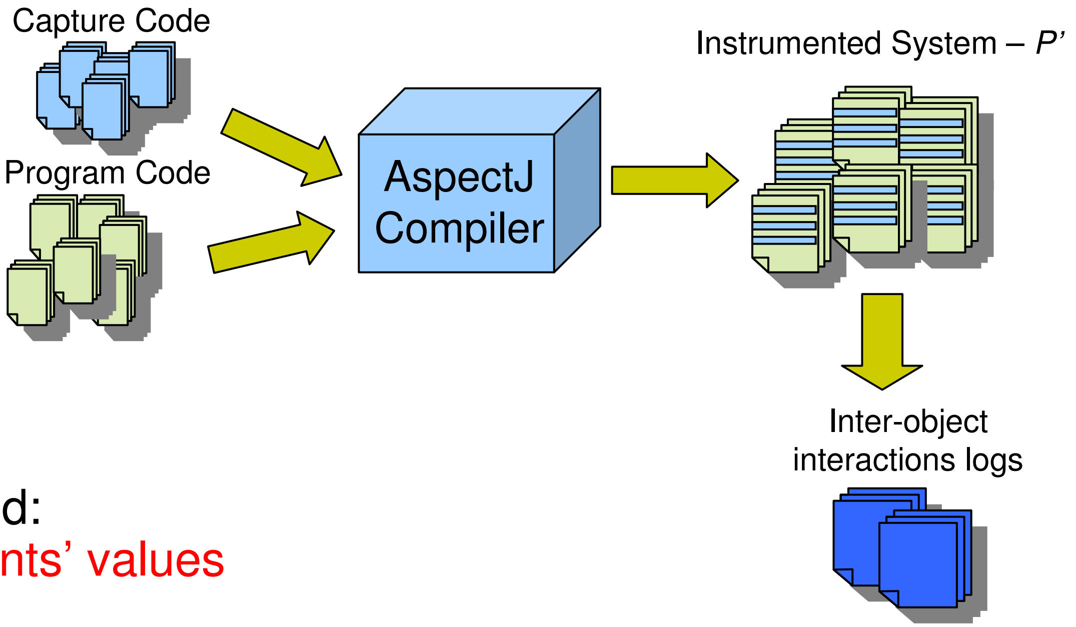
Implementation

Capture Phase

- Software is instrumented with capture functionality at constructor-calls, method-calls, field getter/setters

- Inter-object interactions are captured and logged during runtime

- Attributes of interactions captured:
  signature, target object, arguments' values
  return value/thrown exception



$$\underbrace{\underbrace{\text{returnVal}}_{\text{return value}} = \underbrace{\text{targetObject}}_{\text{target object}}\text{.someMethod}(\underbrace{\text{arg}_1 \text{ , } \ldots \text{ arg}_n}_{\text{arguments' values}}\text{);}}_{\text{signature}}$$

Capture Phase

- Instrumentation is performed using AspectJ
- More elegant and simpler mechanism
- However, it is a weaker mechanism than conventional instrumentation techniques that directly access a program’s Java bytecode
  - → Requires the use of elegant workarounds to handle special cases:
    - → non primitive arrays: `obj1.peform(myArray[6]);`
    - → string syntactic: `String me = “Benny”;`

Generation Phase – Step I

- Given a testable event, a backtracking algorithm recursively generates the statements needed for executing the test

```
@Test public void testpop1() {
    // test execution statements
    IntStack IntStack_2 = new IntStack();        // #1
    IntStack_2.push(2);                          // #2
    IntStack_2.push(3);                          // #3
    IntStack_2.reverse();                        // #4
    int intRetVal6 = IntStack_2.pop();           // #5



}
```

Backtracking Algorithm

- Generally, in order to execute a test, GenUTest needs to generate statements that replay the relevant sequence of recorded events in a correct manner
  - **Execution of:**

    `intRetVal1 = obj1.process(obj2)`
  - **Requires:** `obj1` and `obj2` must be in the correct state

Backtracking Algorithm

- Object states are represented by method-calls sequences:

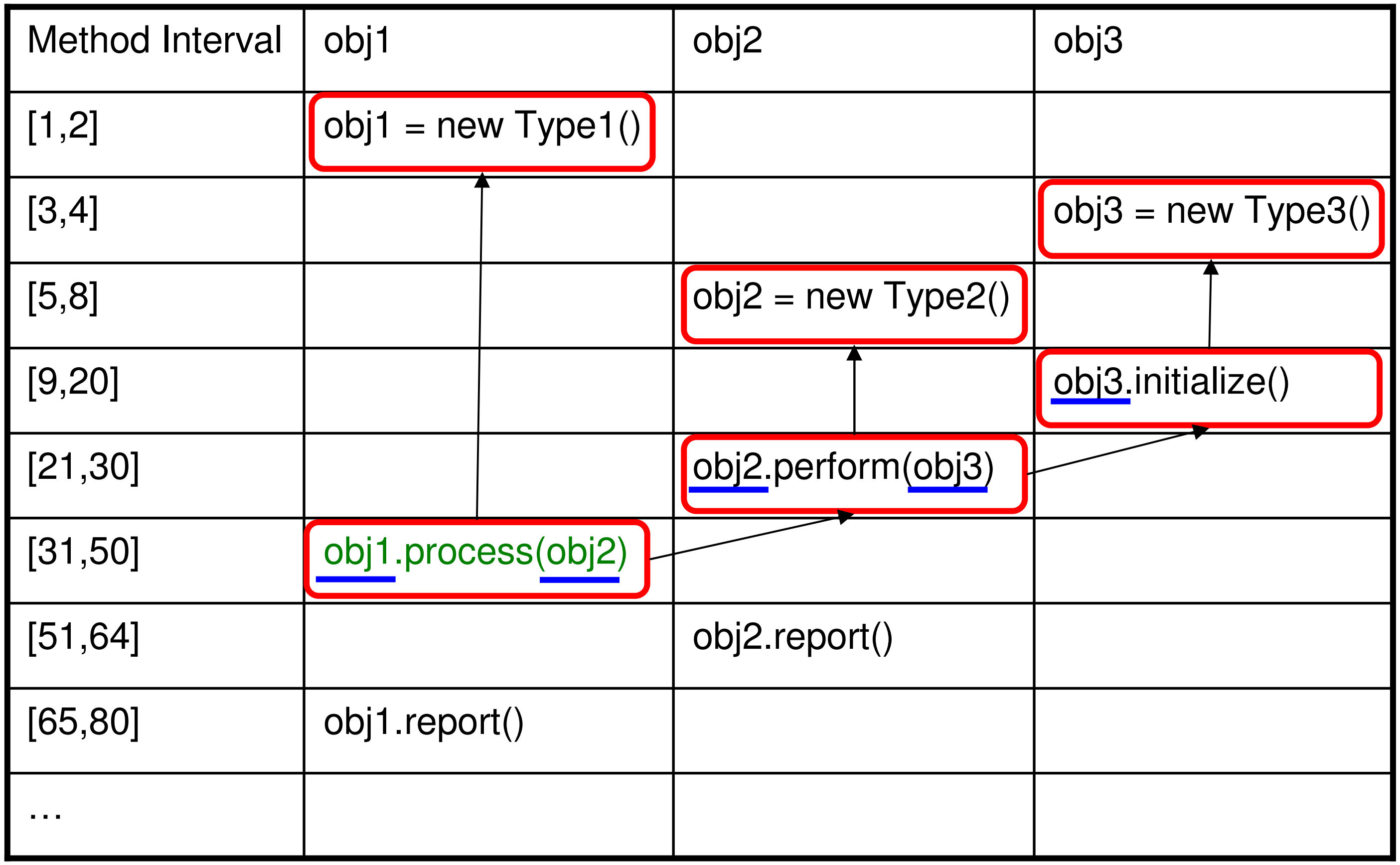
$$\text{state}_T(o) = (\text{method}_{t_1} , \text{method}_{t_2} , \ldots \text{method}_{t_n})$$

$$t_1 < t_2 < .. < t_n < T$$

- Time is represented by a sequence number incremented *before* a method begins execution and *after* it finishes execution
- The interval [*before, after*] is called the *method-interval*

Backtracking Algorithm

- Logged interactions:



Method Interval	obj1	obj2	obj3
[1,2]	obj1 = new Type1()		
[3,4]			obj3 = new Type3()
[5,8]		obj2 = new Type2()	
[9,20]			obj3.initialize()
[21,30]		obj2.perform(obj3)	
[31,50]	obj1.process(obj2)		
[51,64]		obj2.report()	
[65,80]	obj1.report()		
…			

Backtracking Algorithm (cont)

- Generated statements:

```
Type1 obj1 = new Type1();
Type3 obj3 = new Type3();
Type2 obj2 = new Type2();
obj3.initialize();
obj2.perform(obj3);
int intRetVal1 = obj1.process(obj2);
```

- Algorithm may need to remove redundant statements
- Static and dynamic types of objects are stored for:
  - casting – `myObject = (MyObject)List.get(2);`
  - null values – `obj1.process(null);`
  - static methods – `System.out.println(“Hello World”);`
  - changes in modifier access policy – inner private class inheriting from a public outer one

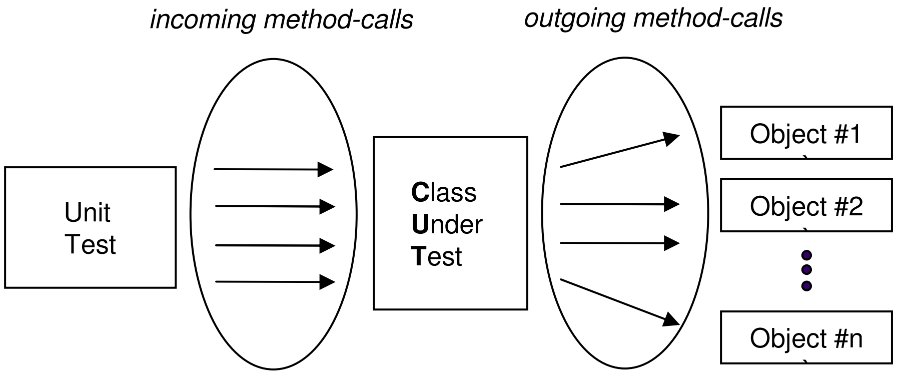
Generation Phase – Step II

- Case I – Value is returned from the call
  - Generate statements that compare value$_{\text{test}}$ with value$_{\text{captured}}$.
- Case II – An exception is thrown
  - Generate statements that expect a particular exception

```
@Test public void testpop1() {
    // test execution statements
    IntStack IntStack_2 = new IntStack();       // #1
    IntStack_2.push(2);                         // #2
    IntStack_2.push(3);                         // #3
    IntStack_2.reverse();                       // #4
    int intRetVal6 = IntStack_2.pop();          // #5

    // test assertion statements
    assertEquals(intRetVal6,2);
}
```

Mock Aspect Generation

- <u>Definitions:</u>
  - *Incoming method-calls* – method-calls invoked by the unit test on the Class Under Test (CUT)
  - *Outgoing method-calls* – method-calls invoked by the CUT on dependent objects



Mock Aspect Generation

- Definitions:
  - *mi(A())* – method interval of A() [Before_A, After_A]
  - method A() **contains** method B() if mi(A()) contains *mi(B())*

    [Before_A, After_A] $\supset$ [Before_B, After_B]

- Observations:
  - method B() resides in the control flow of method A() iff method A() contains method B()
  - An outgoing method-call of the CUT is contained in **exactly one** incoming method-call

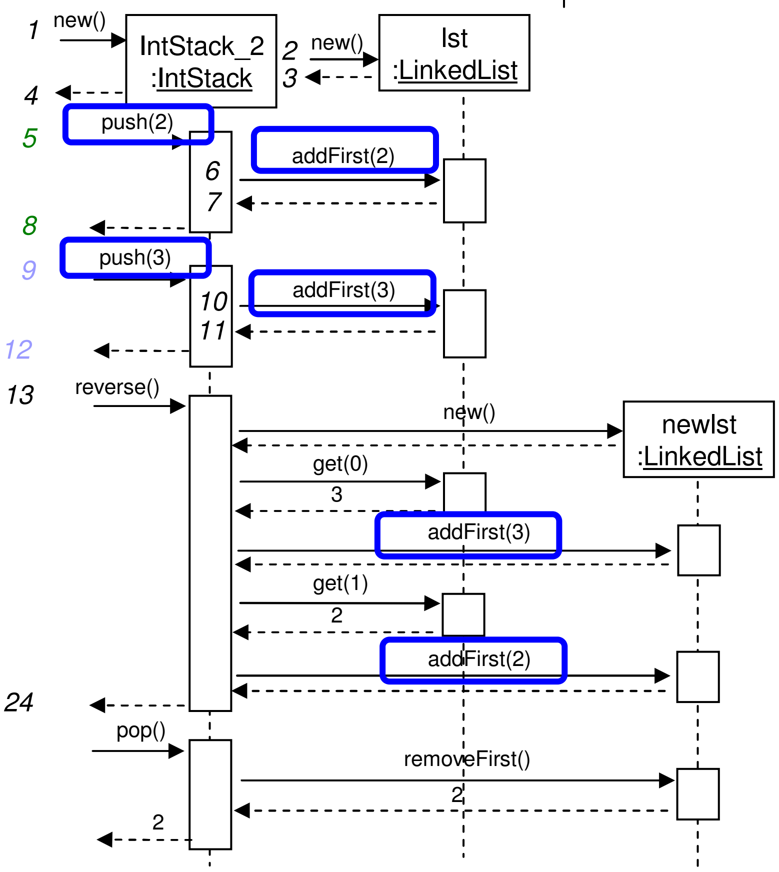
Mock Aspect Generation

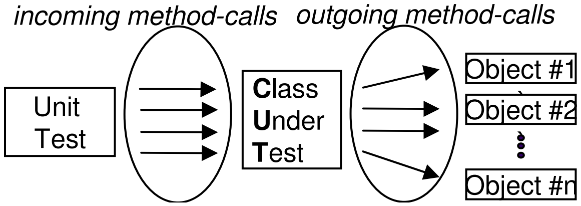
- Last definition ☺
  - *Outgoing(I())* is the sequence <Io_1(), Io_2(), …, Io_n()>
    - I() is an incoming method call
    - Io_1(), Io_2(), …, Io_n() are **all** the outgoing method-calls **contained** in I()
- If method o() is contained in method I() and method o() is the j^{th} element in Outgoing(I()) then method o() is uniquely identified by the pair (mi(I()), j)

Mock Aspect Generation – Needed Example

- Four outgoing method-calls to addFirst()
- mi(push(2)) is [5,8]
- *Outgoing(push(2))* = <addFirst(2)>
- addFirst(2) is uniquely identified by <[5,8],1>

- *mi(push(3))* is [9,12]
- Outgoing(push(3)) = <addFirst(3)>
- addFirst(3) is uniquely identified by <[9,12],1>

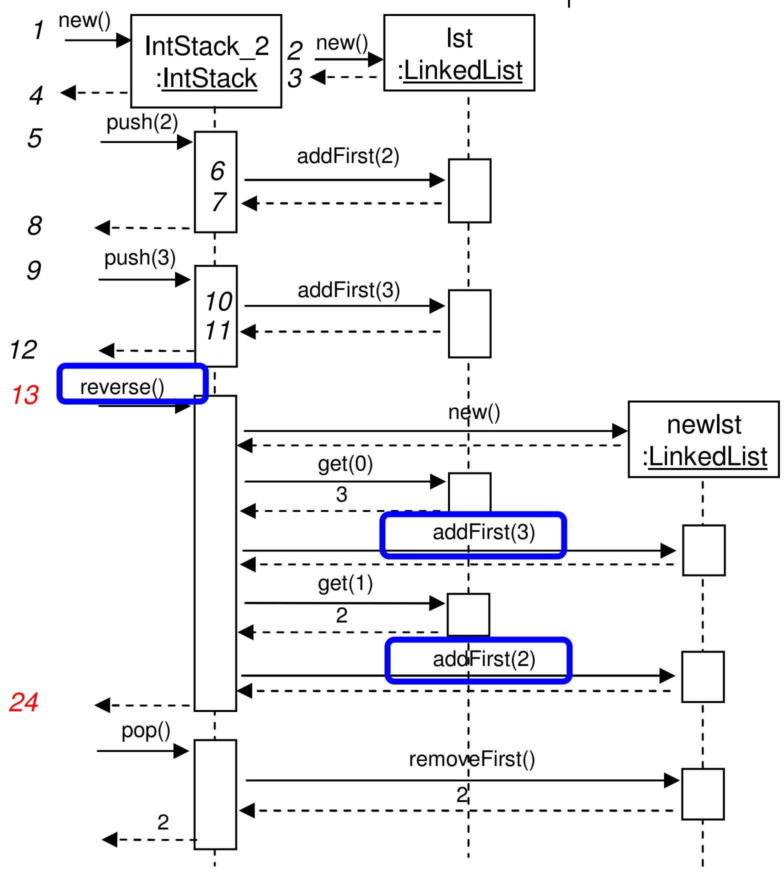


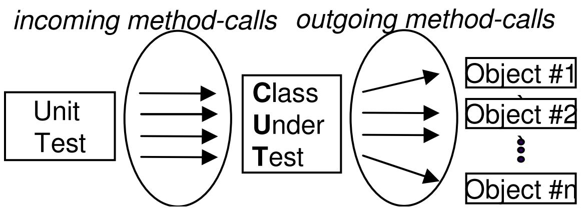


Mock Aspect Generation – Needed Example

- Mi(reverse()) is [13,24]
- Outgoing(reverse()) = <get(0), addFirst(3), get(1), addFirst(2)>

- get(0) is uniquely identified by <[13,24],1>
- addFirst(3) is uniquely identified by <[13,24],2>
- get(1) is uniquely identified by <[13,24],3>
- addFirst(2) is uniquely identified by <[13,24],4>





Mock Aspect Generation

- Algorithm works as follows:
  1. For each incoming method-call I() of the CUT, outgoing(I()) is calculated
  2. Each outgoing method-call is uniquely identified
  3. For each incoming method-call I() different pointcut and advice are generated
  4. A statement that sets method interval and clears the element counter is added before the incoming method call is invoked in the unit test
  5. Bookkeeping code is added in advice
  6. Backtracking algorithm is applied to mimic the behavior of the dependent object in the advice

Mock Aspect Generation – Sample Code

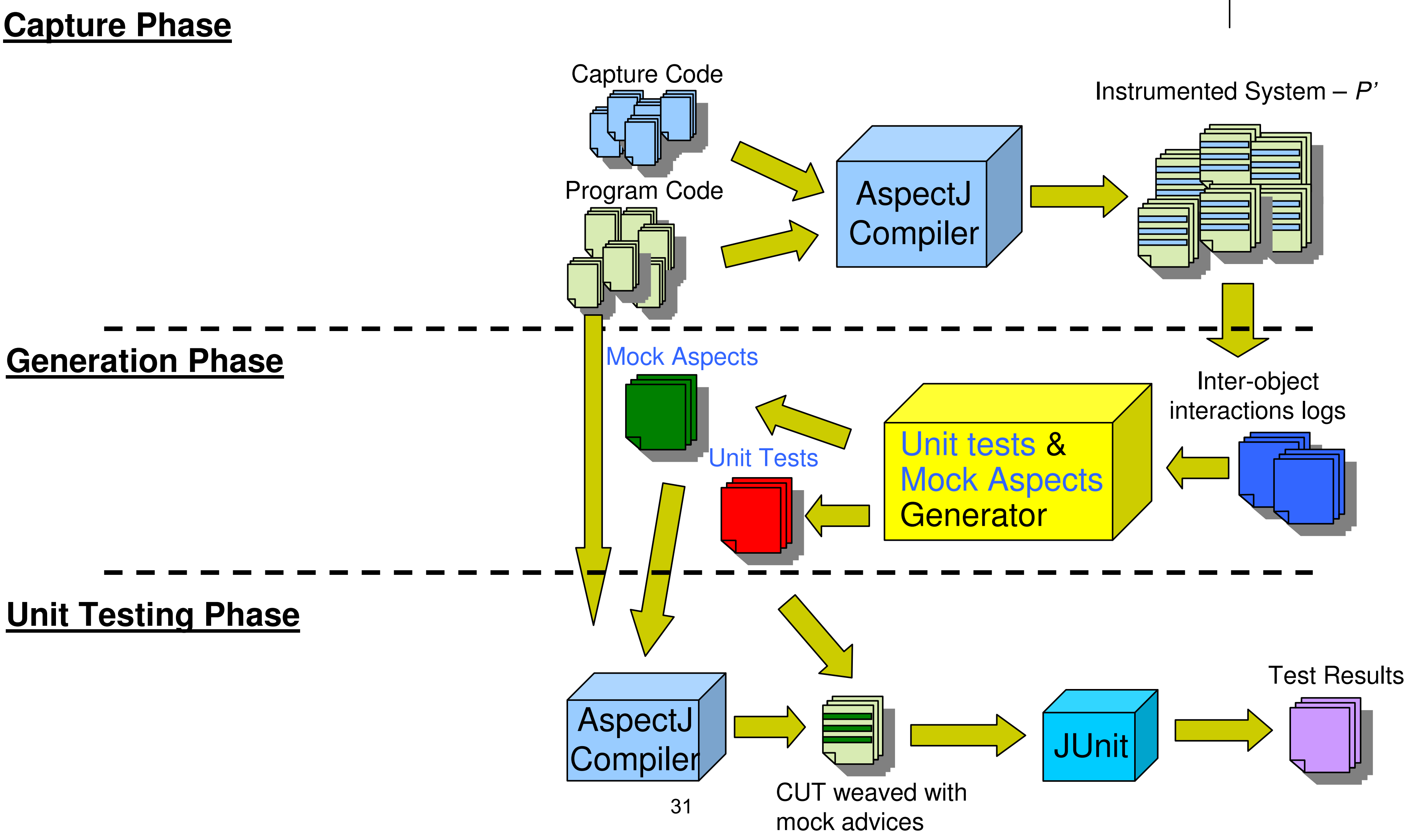
StackIntTest.java

```
@Test public void testpop1()
{
    // test execution statements
    IntStack IntStack_2 = new IntStack();
    IntStack_2.push(2);
    IntStack_2.push(3);

 4  StackIntMockAspect.setMI(13,24);
    IntStack_2.reverse();


    int intRetVal6 = IntStack_2.pop();

    // test assertion statements
    assertEquals(intRetVal6,2);
}
```

StackIntMockAspect.aj

```
Integer around(): call (Object
 3      java.util.LinkedList.get(int)) &&
        restriction()
{
   5 if (before == 13 && after == 24) {
          if (elementCounter == 1) {
             elementCounter++;
            6return 3;
          }

   5      if (elementCounter == 3) {
             elementCounter++;
            6return 2;
          }
     }
     thrown new RuntimeException(“Invalid
     method interval”);
}

void setMI(int b, int a)
{
     before = b;
     after = a;
     elementCounter == 1;
}
```

Implementation Overview



Capture Phase

Capture Code

Program Code

AspectJ Compiler

Instrumented System – *P’*

Generation Phase

Mock Aspects

Unit Tests

Unit tests & Mock Aspects Generator

Inter-object interactions logs

Unit Testing Phase

AspectJ Compiler

CUT weaved with mock advices

JUnit

Test Results

Experimentation

- Used on open source project JODE (Java Optimize and Decompile Environment) http://jode.sourceforge.net/
- JODE is a medium sized project ~35K loc
- Executed JODE combined with GenUTest on a chosen input
- GenUTest generated 592 unit tests from recorded data captured during runtime

Experimentation

- Measured code coverage with EclEmma (www.eclemma.org/):

1. Execution of JODE on chosen input
   Coverage is 25% of JODE’s lines of code

2. Execution of generated unit tests with JUnit
   Coverage is 5.2% of JODE’s lines of code

- Current limitations and bugs may cause generation of invalid tests
  - Primary reason for differences in loc coverage rate

Limitations

- Partial support for inner classes and anonymous classes
- Does not support multi-thread applications
- Support of arrays need to be improved
- Scalability and performance issues

Related work

- Automatic Test Factoring for Java [Saff, Artzi, Perkins, Ernst]
- Selective Capture and Replay of Program Executions [Orso, Kennedy]

  - Capture interactions between a subsystem ***s*** and the system **S**.
  - Recorded interactions can later be used as a mock environment

    - Caveat: requires instrumentation of program

- Carving Differential Unit Test Cases from System Test Cases [Elbaum, Chin, Dwyer, Dokulil]

  - Make use of concrete object states -> incurs heavy price on performance and storage requirements
  - More sensitive to change than method sequence representation

Related work

- Substra: A Framework For Automatic generation of Integration Tests [Yuan, Xie]

  Generates method-call sequences with random values.

  Sequences are subject to constraints inferred using dynamic analysis

- Eclipse Test & Performance Tools Platform Project

  - only supports simple parameters and return value types

Future Work

- Handle limitations and extend support:

  Inner/Anonymous classes, multi-threaded support,

  Optimize array handling, optimize performance
- Scalability – selective capturing, detect redundant tests, discard non mutating events, make use of concrete object states
- Research effectiveness in detecting regression bugs

Thank you for listening

Questions?