Declarative Mocking

Hesam Samimi

Rebecca Hicks

Ari Fogel

Todd Millstein

University of California, Los Angeles

ISSTA'13

Early development testing

```
class MySet {

  List elems;        ← unavailable dependency


  void add(Object o) {
    if (!elems.contains(o))
                              ← code under test
      elems.add(o);
  }
}
```

Early development testing

```
class MySet {
```

← unavailable dependency

```
  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }
}
```

← code under test

Early development testing

```
class MySet {
```

← unavailable dependency

```
  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }
}
```

← code under test

Early development testing

```
class MySet {
```

← unavailable dependency

```
  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }
}
```

← code under test

Early development testing

```
class MySet {

  List elems;

  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }
}
```

← unavailable dependency

← code under test

Mock objects

```
class MySet {

  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }

  void testAdd() {

    List mockList = mock(List.class);
    MySet s = new MySet(mockList);
    when(mockList.contains(0)).thenReturn(false);
    s.add(0);
    verify(mockList, times(1)).add(0);
}
```

mock

stub

behavior-checking

Mock objects

```
class MySet {

void testAdd() {

    List mockList = mock(List.class);
    MySet s = new MySet(mockList);
    when(mockList.contains(0)).thenReturn(false);
    s.add(0);
    verify(mockList, times(1)).add(0);
    when(mockList.contains(0)).thenReturn(true);
    s.add(0);

    //shouldn't add dups
    verify(mockList, times(1)).add(0);

}
```

Mock objects

```
class MySet {

void testAdd() {

    List mockList = mock(List.class);
    MySet s = new MySet(mockList);
    when(mockList.contains(0)).thenReturn(false);
    s.add(0);
    verify(mockList, times(1)).add(0);
    when(mockList.contains(0)).thenReturn(true);
    s.add(0);

    //shouldn't add dups
    verify(mockList, times(1)).add(0);
}
```

little effort

Mock objects

```
class MySet {

void testAdd() {

    List mockList = mock(List.class);
    MySet s = new MySet(mockList);
    when(mockList.contains(0)).thenReturn(false);
    s.add(0);
    verify(mockList, times(1)).add(0);
    when(mockList.contains(0)).thenReturn(true);
    s.add(0);

    //shouldn't add dups
    verify(mockList, times(1)).add(0);

}
```

same API

Mock objects

```
class MySet {

void testAdd() {

    List mockList = mock(List.class);
    MySet s = new MySet(mockList);
    when(mockList.contains(0)).thenReturn(false);
    s.add(0);
    verify(mockList, times(1)).add(0);
    when(mockList.contains(0)).thenReturn(true);
    s.add(0);

    //shouldn't add dups
    verify(mockList, times(1)).add(0);
}
```

✗ fragile, limited, inflexible, indirect

Mock objects

```
class MySet {

void testAdd() {

    List mockList = mock(List.class);
    MySet s = new MySet(mockList);
    when(mockList.contains(0)).thenReturn(false);
    s.add(0);
    verify(mockList, times(1)).add(0);
    when(mockList.contains(0)).thenReturn(true);
    s.add(0);

    //shouldn't add dups
    verify(mockList, times(1)).add(0);
}
```

✗ implementation dependent

Can we have the best of both worlds?

- little effort
- flexible mocks with all/part of real functionality

Can we have the best of both worlds?

- implementation missing but known APIs
- just "specify" intentions w/o code...

Can we have the best of both worlds?

- implementation missing but known APIs
- just "specify" intentions w/o code...

formal specifications
declarative execution

declarative execution?

```
class Counter {

    int count = 0;

    void reset() ensures count == 0;

    void add10() ensures count == old.count + 10;



}
```

only specs
no code!

declarative execution?

```
class Counter {

    int count = 0;

    void reset() ensures count == 0;

    void add10() ensures count == old.count + 10;

    static void main(String[] args) {

          new Counter().add10();

    }
}
```

only specs
no code!

declarative execution?

```
class Counter {

    int count = 0;

    void reset() ensures count == 0;

    void add10() ensures count == old.count + 10;

    static void main(String[] args) {

            new Counter().add10();

    }
}
```

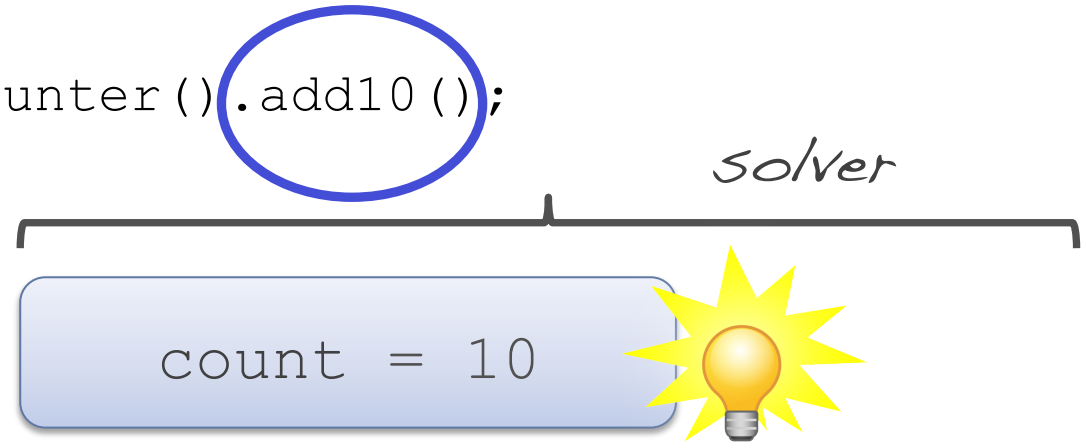
domain of solver

current state: count_old = 0

postcondition: count = count_old + 10

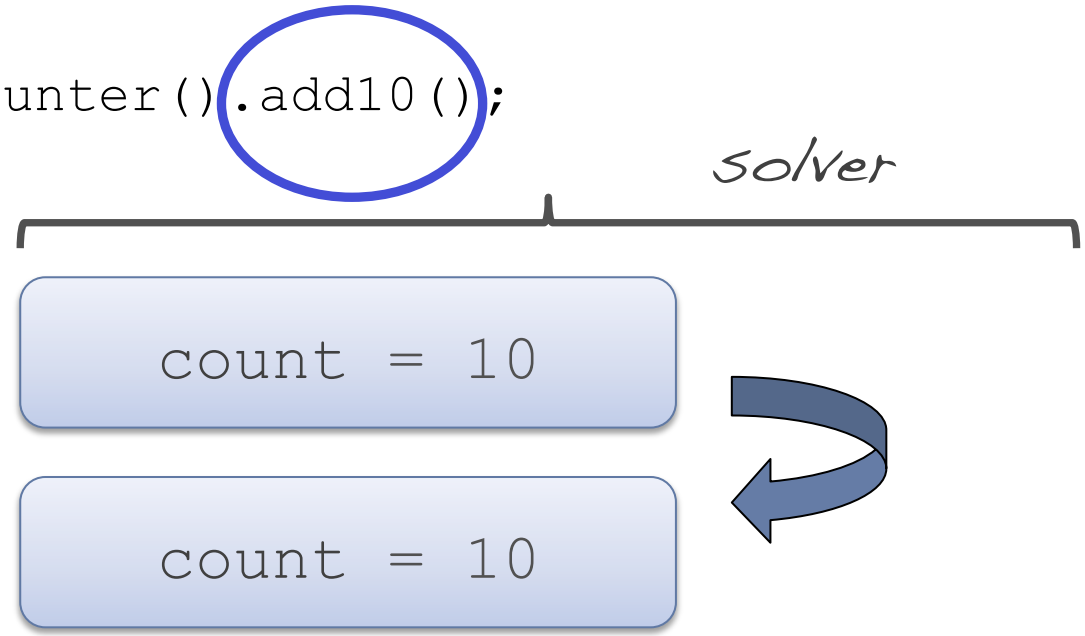
declarative execution?

```
class Counter {

    int count = 0;

    void reset() ensures count == 0;

    void add10() ensures count == old.count + 10;

    static void main(String[] args) {

            new Counter().add10();

    }
}
```

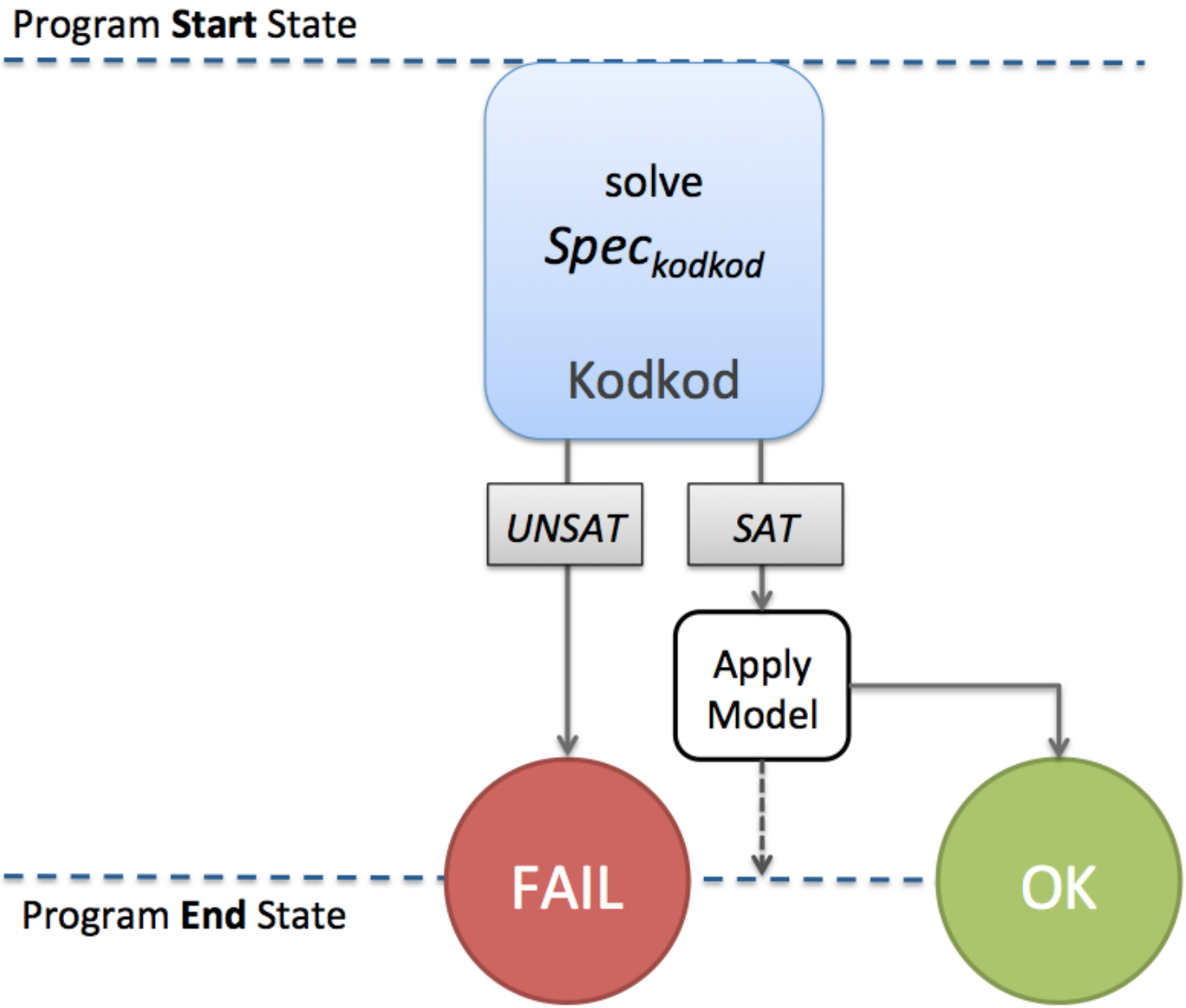


declarative execution?

```
class Counter {

    int count = 0;

    void reset() ensures count == 0;

    void add10() ensures count == old.count + 10;

    static void main(String[] args) {

            new Counter().add10();

    }
}
```



declarative execution for Java (PBnJ)



Mocking with PBnJ

```
class MockList implements List {

  Object[] elems; int size;

  pure void contains(Object o)
    ensures result <==>
      some int i : 0 .. size - 1 |
        elems[i].equals(o);


  void add(Object o)
    modifies fields MockList:elems, MockList:size
    ensures size == old.size + 1
      && elems[this.old.size] == o
      && all int i : 0 .. old.size - 1 |
        elems[i] == old.elems[i];
}
```

declarative mock: no code!

Declarative fn mocking

```
class MySet {

  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }

  void testAdd() {

    List mockList = new MockList();
    MySet s = new MySet(mockList);
    s.add(0);
    s.add(0);
    //shouldn't add dups
    assert s.size() == 1;
}
```

Declarative fn mocking

```
class MySet {

  void add(Object o) {
    if (!elems.contains(o))
      elems.add(o);
  }

  void testAdd() {

    List mockList = new MockList();
    MySet s = new MySet(mockList);
    s.add(0);
    s.add(0);
    //shouldn't add dups
    assert s.size() == 1;
}
```

dynamic

reusable

direct testing

no coupling w/ code

Declarative data/env mocking

```
class MySet {

  List elems;

  void test1() {
    assume elems.size() == 0;
    // now run test…
  }

  void test2() {
    assume elems.contains(null);
    // now run test…
  }
}
```

data of mock or real objects…

Declarative mocking

```
class MySet {

  List elems;

  void test() {
    unique assume elems.contains(null)
       && elems.size < 5;

    // now run test…
  }
}
```

try all possible scenarios…

exploratory study

find cool applications of mocking

- potentials
- limitations

evaluation

port existing

unit tests

in practice…

Advantages

declarative expression

underspecification

nondeterminism

compositionality

extensibility

reusability

experimentation

data integrity

JDBC: mocking a DB

"Does `button1` *work properly assuming that we are connected to a database named* `shop`*, which has a table named* `inventory`*, for which when I run the query*

```
select * from inventory where price = 0
```

I get no results? What about when I get two rows?"

declarative expression

underspecification

TFTP: mocking the server

```
class MockServer {
  Message sendMessage()
  ensures isErrorInducingDATA(result)
  || isWellFormedDATAMessage(result)
  || isNonErrInducingDATAMessage(result);

  spec boolean isCorrectMessage(Message m) { … }
  spec boolean isErrorInducingDATA(Message m){ … }
  spec boolean isNonErrInducingDATA(Message m){ … }
  spec boolean isWellFormedDATA(Message m){ … }

}
```

nondeterminism

TFTP: mocking the server

```
class MockServer {
  unique Message sendMessage()
  ensures isErrorInducingDATA(result)
  || isWellFormedDATAMessage(result)
  || isNonErrInducingDATAMessage(result);

  spec boolean isCorrectMessage(Message m) { … }
  spec boolean isErrorInducingDATA(Message m){ … }
  spec boolean isNonErrInducingDATA(Message m){ … }
  spec boolean isWellFormedDATA(Message m){ … }

}
```

nondeterminism

Limitations

effort

- vs. stubs
- vs. imperative code
    - JDBC
        - mocking fn not worth it
        - mocking data is

efficiency

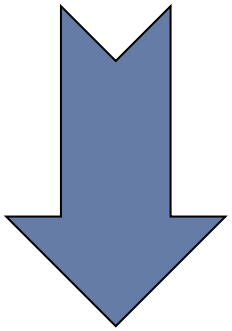
scalablity

evaluation

6 apps, 114 tests from code.google.com

```
when(mockList.contains(0)).thenReturn(false)
```

stub



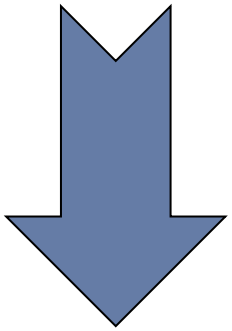
PBnJ spec

evaluation


```
when(mockList.contains(0)).thenReturn(false)
```

stub

phase I



exact fn

```
contains(int x) ensures x == 0 ==> !result;
```

spec

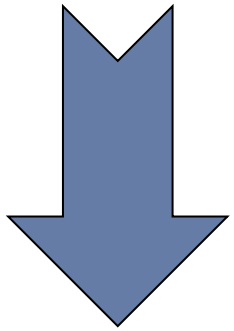
overhead?

evaluation


```
when(mockList.contains(0)).thenReturn(false)
```

stub

phase II



generalize/
reuse

```
contains(int x) ensures
   some Object e : elems | e == x;
```

spec

benefit from advantages of specs:

coverage

effort

phase I

name	# tests with stubs	spec / code LoC	avg time (sec.)
j2bugzilla	13	1.4	4
jscep	4	2.6	0
tjays1-project	18	1.8	1
gcm-server	23	0.9	1
shivaminesweeper	15	1.8	1
birthdefectstracker	41	2.8	0

measure of "effort" → 2:1

1 sec.

		phase I		phase II		
name	# tests with stubs	spec / code LoC	avg time (sec.)	% test enhanced	spec / code LoC	avg time (sec.)
j2bugzilla	13	1.4	4	85	0.4	12
jscep	4	2.6	0	0	-	-
tjays1-project	18	1.8	1	44	0.8	1
gcm-server	23	0.9	1	30	1.0	2
shivaminesweeper	15	1.8	1	93	0.7	52
birthdefectstracker	41	2.8	0	73	1.9	104
		2:1	1 sec.	54%	1:1	34 sec.

Specs not useful...

```
void testHandleForCertificate() {
  verifier = mock(CertificateVerifier.class);
  // check our code invokes the verifier…
}
```

stub is irrelevant to the test...

Specs not useful...

```
class JDBC {

  Map<String,Table> tables;

  void createTable(String id, String[] fields)

     ensures createTa✗Spec() {

     tables.put(id, new Table(id, fields));

  }

}
```

mocking functionality with imperative code is straightforward

Related work

mocking frameworks

[Mockito, JMock, EasyMock, …]

- easier mocking with stubs

declarative mocking

- more expressive mocking

Related work

test/mock generation

[PEX, ASE '06]

- symbolic execution
- analyze test code
- produce inputs/stubs for high path coverage

declarative mocking

- not useful for code coverage
- dynamic
- fill in for missing data/functionality at run time
- independent of tests

Related work

others

- [Qi, WCRE'12]
  - env models based on execution traces
- [Henkel, TOSEM'108]
  - term rewriting on algebraic specs
- [Galler, AST'10]
  - statically producing stubs based on preconditions
- [Wilmore, ICSE'06]
  - db preparation. intensional specs as constrained queries

Conclusions
declarative mocking of:

functionality

- logic of test dependent on mock
- underspec
- nondeterminism

data

- complex integrity properties
- underspec
- nondeterminism

Conclusions

declarative mocking of:

functionality

- logic of test dependent on mock
- underspec
- nondeterminism

data

- complex integrity properties
- underspec
- nondeterminism

Kodkod + mockito

Thank you!

Backup Slides

exploratory study

- JStock – mock webserver's data
- JDBC – mock database fn & data
- TFTP – mock server nondeterminism
- Hadoop Map Reduce – mock cloud fn & env

TFTP: mocking the server

```
spec boolean isNonErrorInducingMsg(Msg m) { … }

spec boolean isWellFormedMsg(Msg m) { … }

spec boolean isWellFormedErrorInducingMsg(Msg m) {
  return !isNonErrorInducingMsg(m)
    && isWellFormedMsg(m);
}
```

compositionality

Hadoop: mocking the scheduler

```
class MockScheduler {
  spec boolean assignTasksSpec(){
    // no reduce jobs until all map jobs done…
    // etc.
  }
}

class MockScheduler_FIFO extends MockScheduler {
  spec boolean assignTasksSpec() {
    super.assignTasksSpec()
      && assignTasksSpec_FIFO();
  }
}
```

extensibility, reusability, experimentation

JDBC: mocking a DB

```
class Table ensures uniqueRows() {

  String primaryKey;
  List <String> columns;
  List <Tuple> rows;

  spec boolean uniqueRows () {
    int primaryIdx = columns.indexOf(primaryKey);
    return all int i : 0 .. rows.size() - 1 |
      all int j : 0 .. rows.size() - 1 |
        (i != j ==>
           rows.get(i).get(primaryIdx) !=
           rows.get(j).get(primaryIdx));
  }

}
```

data integrity