Testing debugging & Verification - Property based testing (Stateful)

Atze van der Ploeg

Unit tests & Property based testing

Great for stateless code

What about for stateful code?

A trivial example

```
class BoolState {

  private boolean flag;

  public BoolState(){ flag = false; }

  public boolean getFlag(){ return flag; }

  public void setFlag(boolean flag){ flag = flag; }

}
```

Very lame indeed, but what to test?

Instead of testing single function, test **interaction** between functions

Properties:

Unit tests:

```
new BoolState().getFlag() == false;
```

```
  new BoolState().getFlag() == false;
  {
  BoolState b = new BoolState();
  b.setFlag(true);
  b. getFlag() == true;
  }
```

```
forall (b : BoolState) (f : boolean).
  b.setFlag(f);
  b.getFlag() == f;
```

Algebraic: Test equality of sequences of statements

```
void action1(BoolState b){
  b.setFlag(b.getFlag());
}
```

==

```
void action2(BoolState b){
}
```

How to test this?

Instead of generating input data, generate programs from the actions we can do!

Actions: get, set

Example program:

Shared random prefix

```
boolean test1(){
 BoolState b = new BoolState ();
 b.set(true);
 b.set(false);
 b.set(false);
 b.set(b.get());
 return b.get();
}
```

```
boolean test2(){
 BoolState b = new BoolState ();
 b.set(true);
 b.set(false);
 b.set(false);
 return b.get();
}
```

Equivalent code

Observe state

More complicated example: Mutable Sets

```
void action1(Set<int> x,int elem){
  x.add(elem);
  x.add(elem);
}
```

==

```
void action2(Set<int> x,int elem){
  x.add(elem);
}
```

```
void action3(Set<int> x,int elem){
  if(!x.contains(elem)) {
    x.add(elem);
    x.remove(elem);
  }
}
```

==

```
void action4(Set<int> x,int elem){
}
```

```
void action1(Set<int> x,int elem){
  x.add(elem);
  x.add(elem);
}
```

==

```
void action2(Set<int> x,int elem){
  x.add(elem);
}
```

Example program:

Shared random prefix

Equivalent code

Shared random postfix

Observe state

```
int[] test1(){
 Set<int> s = new Set();
 b.add(2);
 b.add(3);
 b.add(1);
 b.add(5);
 b.add(5);
 b.remove(6);
 b.add(10);
 b.add(3);
 return b.allElements();
}
```

```
int[] test2(){
 Set<int> s = new Set();
 b.add(2);
 b.add(3);
 b.add(1);
 b.add(5);
 b.remove(6);
 b.add(10);
 b.add(3);
 return b.allElements();
}
```

```
void action3(Set<int> x,int elem){
  if(!x.contains(elem)) {
    x.add(elem);
    x.remove(elem);
  }
}
```

==

```
void action4(Set<int> x,int elem){
}
```

Example program:

```
int[] test1(){
 Set<int> s = new Set();
 b.add(2);
 b.add(3);
 b.add(1);
 if(!b.contains(4)) {
    b.add(4);
    b.remove(4);
  }
 b.remove(6);
 b.add(10);
 b.add(3);
 return b.allElements();
}
```

```
int[] test2(){
 Set<int> s = new Set();
 b.add(2);
 b.add(3);
 b.add(1);
 b.remove(6);
 b.add(10);
 b.add(3);
 return b.allElements();
}
```

Shared random prefix

Equivalent code

Shared random postfix

Observe state

Shrinking on stateful test cases


```
int[] test1(){
 Set<int> s = new Set();
 b.add(2);
 b.add(3);
 b.add(1);
 if(!b.contains(4)) {
    b.add(4);
    b.remove(4);
  }
 b.remove(6);
 b.add(10);
 b.add(3);
 return b.allElements();
}
```

Shrink

Shrink

```
int[] test2(){
 Set<int> s = new Set();
 b.add(2);
 b.add(3);
 b.add(1);
 b.remove(6);
 b.add(10);
 b.add(3);
 return b.allElements();
}
```

Shared random prefix

Equivalent code

Shared random postfix

Observe state

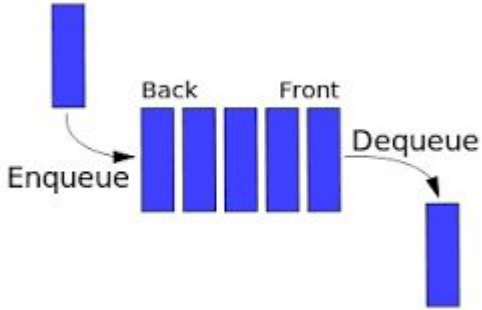
Yet another example: queues

First - in - first - out (FIFO)

Operations:

```
empty : Queue
enqueue : (int,Queue) -> ()
isEmpty : Queue -> bool
front   : Queue -> int
dequeue : Queue -> ()
```

Let’s think of equivalent statements



Sollutions

Sollutions

```
bool action1(){
  return isEmpty(empty);
}
```

==

```
bool action2(){
   return true
}
```

```
int action2(int x){
  q1 = empty;
  enqueue(x,q1);
  return front(q1);
}
```

==

```
int action3(){
  q2 = empty();
  enqueue(x,q2);
  return x;
}
```

```
int action3(int n ,int m){
  enqueue(n,q1)
  enqueue(m,q1);
  return front(q1);
}
```

==

```
int action4(int n, int m){
  enqueue(n,q2);
  int res = front(q2);
  enqueue(m,q2);
  return res
}
```

Sollutions

```
Queue action1(int m){
 Queue q = empty();
  enqueue(m,q);
  dequeue(q);
  return q;
}
```

==

```
Queue action2(int m){
 return empty();
}
```

```
Queue action2(Queue q, int x, int y){
   enqueue(x,q);
   enqueue(y,q);
   dequeue(q);
   return q;
}
```

==

```
Queue action2(Queue q, int x, int y){
   enqueue(x,q);
   dequeue(q);
   enqueue(y,q);
   return q;
}
```

Property: equivalence

```
class SuperSmartSet { ... }
```

Generate random *programs*,
Check that they give the same output
No algebraic rules needed!

```
class SlowAndSimpleSet { ... }
```

A true Story - Google LevelDB

Superfast simple persistent key/value store

Based on Log Structed Merge Trees , Bloom Filters etc

Operations:

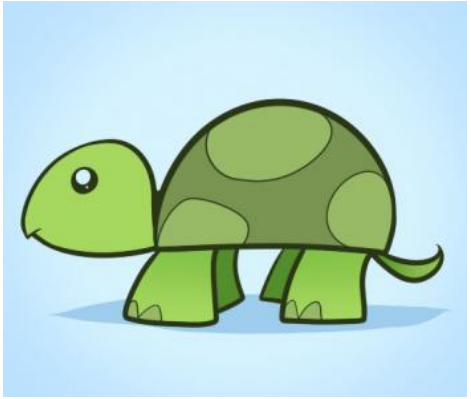
```
void insert(String key, String val);

String lookup(String key);
```

A true Story - Google LevelDB

Joseph Wayne Norton implemented a [Talk link]

slow and simple key value store, based on Google’s spec



More cool stuff by John