Property-based testing in Java

Ahmad Abdelghany

Supervisors:
Horatiu Cirstea Pierre-Etienne Moreau

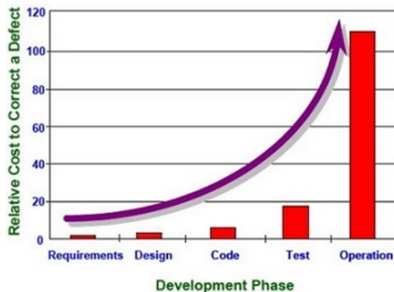
June, 2015

Software Quality

- Ensure correctness, security, reliability, . . .
- Most importantly, bug-free software.

Software Quality

- Ensure correctness, security, reliability, ...
- Most importantly, bug-free software.
- Cost of fixing software bugs



Property-based testing

General Idea

- Instead of one-input test scenario, a function is tested with many auto-generated test inputs.
- Properties that a function should fulfill are verified.
- No external specification language/tools needed.
- Higher confidence in quality.

Outline

1. Introduction
2. PropCheck: A testing framework
3. Factory Generator
4. Conclusion

Outline

1. Introduction
   - What is property-based testing?
2. PropCheck: A testing framework
   - Implementation
   - PropCheck in action
3. Factory Generator
   - Implementation
4. Conclusion

Property-based testing tools

- In Haskell:
  - **QuickCheck [ICFP00]** - Random testing
  - **SmallCheck [ACM08]** - Exhaustive testing
- In Java:
  - **QuickCheck for Java, JCheck, junit-quickcheck**

    $\Rightarrow$ Only primitives and some built-in types (e.g. arrays, Lists)

    $\Rightarrow$ No support for user-defined data types.

    $\Rightarrow$ Eager by nature.

Verifying Properties

$$\forall s : Stack, n \in \mathbb{N} \bullet top(push(s, n)) = n$$

```
@DataPoints int[] n = {1, -3, 5}
@DataPoints
Stack[] stack = {getListOfStacks()}
@Theory
public void testTop(Stack s, int n) {
  assertThat(s.push(n).top(), is(n));
}
```

Verifying Properties

$$\forall s : Stack, n \in \mathbb{N} \bullet top(push(s, n)) = n$$

```
@Theory
public void testTop(
  @ForSome Stack s, @ForSome int n
) {
  assertThat(s.push(n).top(), is(n));
}
```

Verifying Properties

$$\forall s : Stack, n \in \mathbb{N} \bullet top(push(s, n)) = n$$

```
@Theory
public void testTop(
   @ForSome Stack s, @ForSome int n
) {
   assertThat(s.push(n).top(), is(n));
}
```

```
Testing testTop().
100 tests were generated.
(tested/assumption violation/bad input):
    100/0/0
```

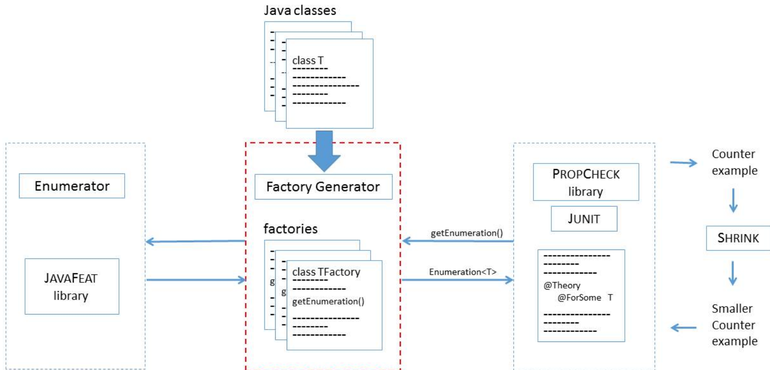
Contribution

- Design & Implementation of Factory generator.
- The factory is part of PropCheck testing framework.
- Enumerate user-defined Java types.
- Successfully tested for different use cases.

Outline

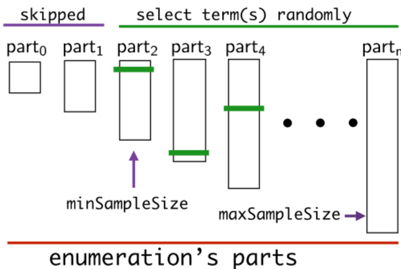
1. Introduction
   - What is property-based testing?
2. PropCheck: A testing framework
   - Implementation
   - PropCheck in action
3. Factory Generator
   - Implementation
4. Conclusion

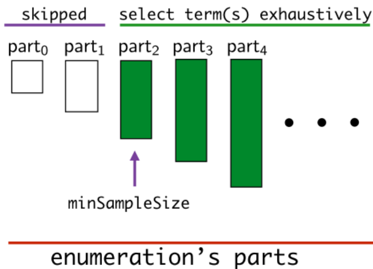
Architecture



JavaFeat enumerator

FEAT	Functional Enumeration of Algebraic Types.[ACM12]
Enumeration	Lazy (possibly infinite) set of Terms of type T
Parts	Finite subsets containing terms of given size
Size	Term size is computed as number of constructors





Algebraic data types

- Bool = False | True

```
Enumeration<Boolean> boolEnum =
  Enumeration.singleton(true).plus(
    Enumeration.singleton(false)
  )
)
```

Algebraic data types

- Bool = False | True

```
Enumeration<Boolean> boolEnum =
  Enumeration.singleton(true).plus(
    Enumeration.singleton(false)
  )
)
```

- Character = a | b | c ... x | y | z

```
Enumeration<Character> charEnum =
  Enumeration.singleton(a).plus(
    Enumeration.singleton(b) ...
  )
)
```

- Nat = Zero | Succ Nat

Enumerating Java types as algebraic types

```
public class Student {
  ...
  public Student(int code,String name) {
    this.code = code;
    this.name = string;
  }
}
```

Enumerating Java types as algebraic types

```
public class Student {
   ...
   public Student(int code,String name) {
      this.code = code;
      this.name = string;
   }
}
```

```
Student = student(i:Int, s:String) | ...
```

Our options are:

- Use Tom/Gom extensions.
- Manual definition.

Enumerating Java types as algebraic types

```
public class Student {
  ...
  public Student(int code,String name) {
    this.code = code;
    this.name = string;
  }
}
```

```
Student = student(i:Int, s:String) | ...
```

Our options are:

- Use Tom/Gom extensions. automate?!
- Manual definition.

Outline

1. Introduction
   - What is property-based testing?
2. PropCheck: A testing framework
   - Implementation
   - PropCheck in action
3. Factory Generator
   - Implementation
4. Conclusion

Factory generator overview

- Uses Java Reflection API.
- Captures type definition via annotations.
- Uses Apache Velocity template engine.
- Generates custom factories.
- Works for many structurally-complex types.

Simple types

```
@Enumerate
public Student(
  @Enumerate(maxSize = 8) int code,
  @Enumerate(maxSize = 4) String name
) {
  this.code = code;
  this.name = name;
}

public class StudentFactory {
  public static final Enumeration<Student>
      getEnumeration() {
     ...
  }
}
```

Generated Factory

```
public static final Enumeration<Student>
    getEnumeration() {
   ...
   F<Integer, F<String, Student>> _student =
   ...
   Enumeration<Integer> noEnum =
      Combinators.makeInteger().parts().take(8);
   ...
   Enumeration<String> stringEnum =
      Combinators.makeString().parts().take(4));
   ...
}
```

Handling Dependencies

- One-To-One relations:
  - Factory for referenced type generated.
  - Generated factory used in referencing type.
- One-To-Many relations:
  - Factory for referenced type generated.
  - Collections factories used.
- Many-To-Many relations:
  - Decomposed into One-To-Many relations.

Abstract types

- No constructors!

```
@Enumerate
public Person(
   @Enumerate(
      concreteClasses = {Dog.class, Cat.class}
   ) Pet pet
) {
   this.pet = pet;
}
```

```
Size --> Enumeration
  0  --> []
  1  --> [Cat [name=, age=0, nick=], Dog [name=, age=0]]
  2  --> [Cat[name=, age=0, nick=a],...,Dog[name=a, age=0],...]
```

Mutually-recursive types

- Challenges:
  1. Detecting cyclic types.
  2. Reference To Enumeration.
- Solution:
  1. Build dependency tree.
  2. Define a fix point.

Construction from methods

```java
@Enumerate
public ListStack() {
   stack = new ArrayList<Integer>();
}
@Enumerate
public IStack push(@Enumerate Integer elem) {
   stack.add(elem);
   return this;
}
```

- We need a reference to "this" instance to call non-static methods
- Handled as recursive types.
- Generated enumeration is used to refer to "this" instance.

Outline

1. Introduction
   - What is property-based testing?
2. PropCheck: A testing framework
   - Implementation
   - PropCheck in action
3. Factory Generator
   - Implementation
4. Conclusion

Conclusion and future work

- Conclusion:
  - Powerful property-based testing framework for Java.
  - Functional enumeration of built-in and user-defined types (lazily).
  - Easy to use and available in all Java environments.
- Future work:
  - Consider more complex data types.
  - Specify distribution of test data.

Thank You!