

---

# COMP5111 – Fundamentals of Software Testing and Analysis

## Random Testing (Feedback Directed)

---



---

Shing-Chi Cheung

Computer Science & Engineering

HKUST

Some slides are adapted from [https://randoop.github.io/randoop/files/thesis\\_talk\\_post.pdf](https://randoop.github.io/randoop/files/thesis_talk_post.pdf)

Can someone write  
tests for me?



**Dream of a developer...**



# Automatic Software Testing

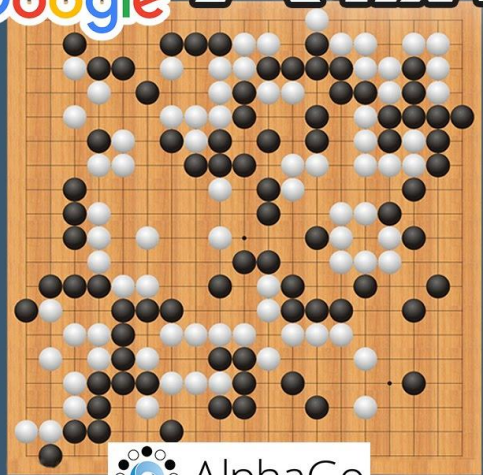
- Random testing
- Symbolic analysis
- Concolic testing
- Search-based testing

Can unit tests be automatically generated without program spec?

Can generated tests detect real faults?



# Google 人機世紀之戰



AlphaGo

VS



李世石

3/9 (三) 12:00

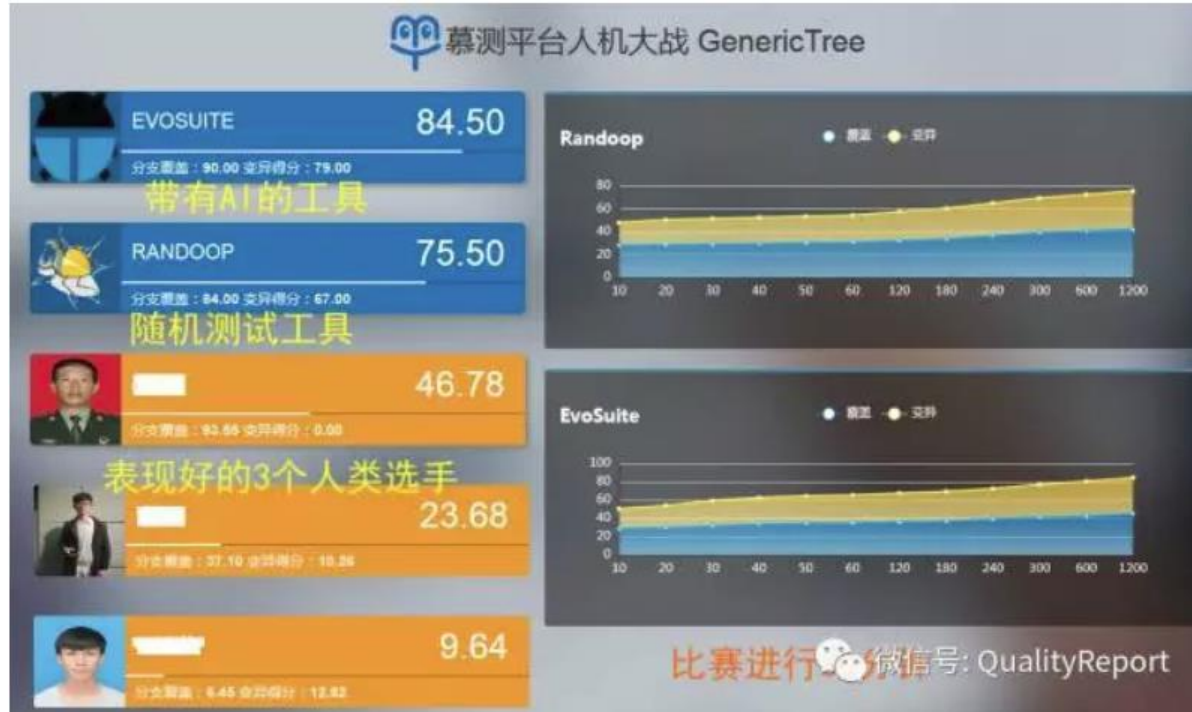
Round 1



LIVE

全程 中文 直播

# Three selected top testers among a few thousand contest participants vs two test generation algorithms in 2017

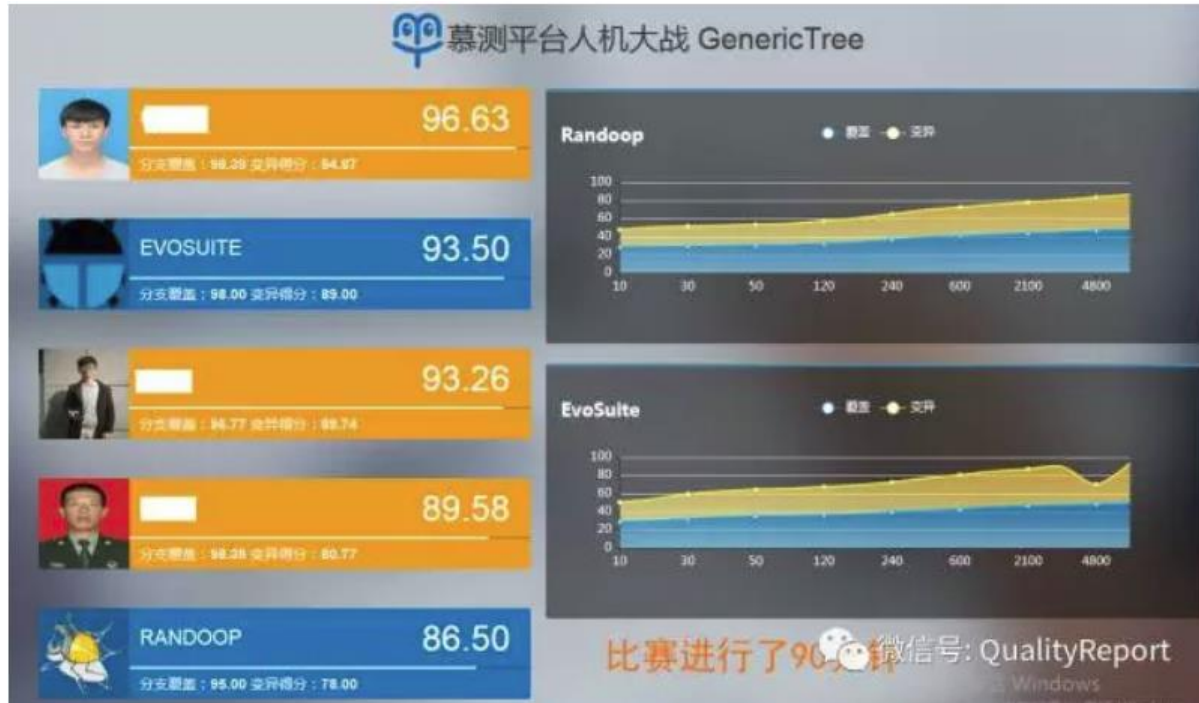




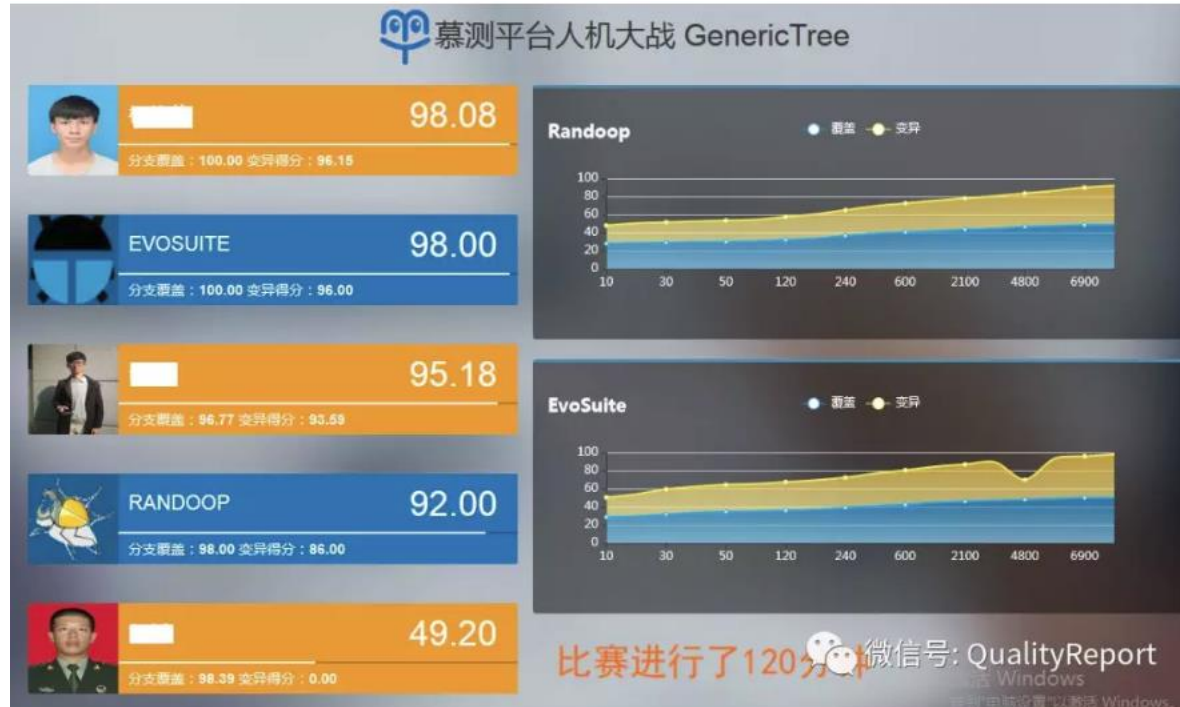
## After 60 mins



After 90 mins



After 120 mins





# Random Testing

```
foo (int &x, int &y) {  
    if (x>y) {  
        x = x + y;  
        y = x - y;  
        x = x - y;  
        if (x - y > 0) {  
            assert (false); // bug  
        }  
    }  
}
```

Options:

- Random input data generation
- Random user interaction sequence
- Random data selection from database
- Combinations of all above

*1<sup>st</sup> trial: x = 1321, y = 456;*

*2<sup>nd</sup> trial: x = -2908, y = 89;*

*...*

*n<sup>th</sup> trial: ...*

# Random Testing

- Mentioned first time by Glenford J. Myers in 1979.
- Popularly used by industry as fuzzing tests.

## Microsoft uncovered more than 1,800 bugs in Office 2010 by running millions of 'fuzzing' tests using idling PCs.

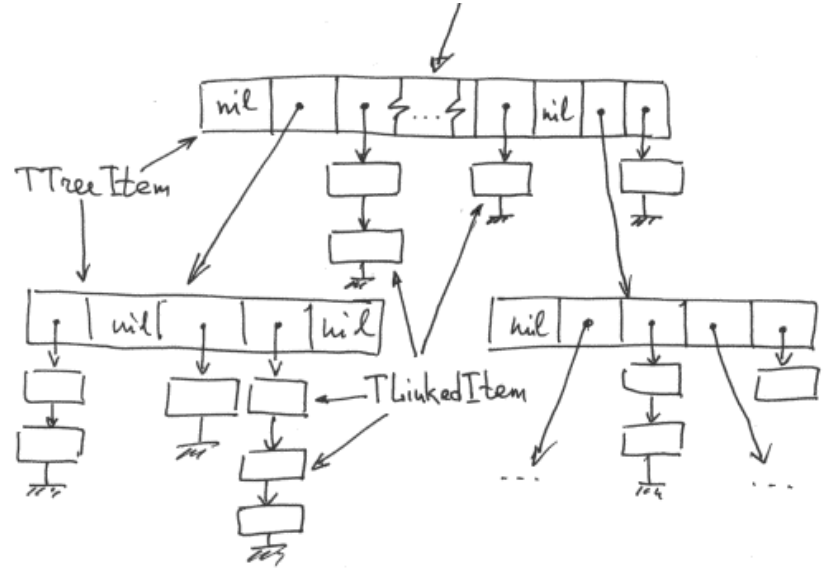
CWmike writes

"Microsoft uncovered more than 1,800 bugs in Office 2010 by tapping into the unused computing horsepower of idling PCs, a company security engineer said on Wednesday. Office developers found the bugs by running millions of 'fuzzing' tests, a practice employed by both software developers and security researchers, that searches for flaws by inserting data into file format parsers to see where programs fail by crashing. 'We found and fixed about 1,800 bugs in Office 2010's code,' said Tom Gallagher, senior security test lead with Microsoft's Trustworthy Computing group, who last week co-hosted a presentation on Microsoft's fuzzing efforts at the CanSecWest security conference. 'While a large number, it's important to note that that doesn't mean we found 1,800 security issues. We also want to fix things that are not security concerns.'"



# Random Unit Test Generators


- Challenge 1: Generate complex data structures
- Challenge 2: Avoid generating redundant tests
- Challenge 3: Generate test oracles (i.e., the assert statements)



# A Polynomial Library

Dynamic data structure

```
class Poly {  
    List <Mono> elements;  
    Poly() { ... }  
    Poly plus(Mono m) { ... }  
    Poly mult(Poly p) { ... }  
    Poly sum(Poly p) { ... }  
    Poly deriv() { ... }  
    ...  
}
```



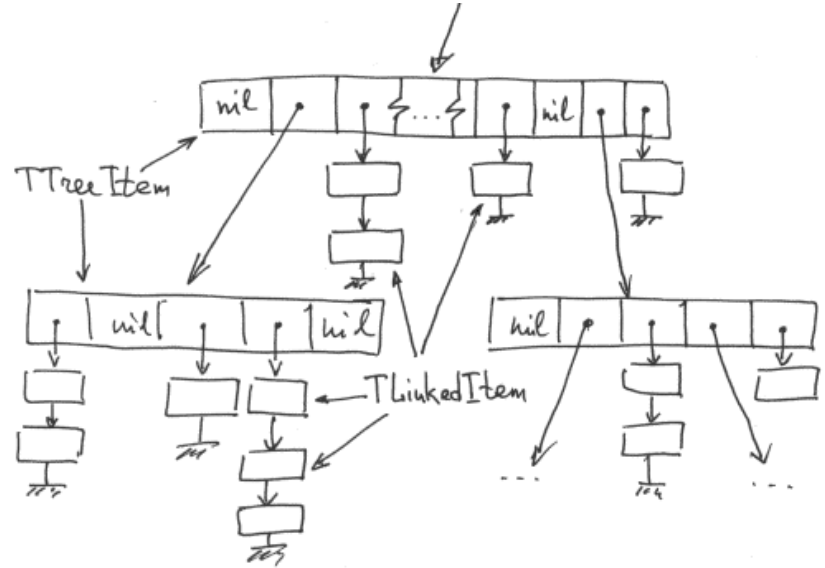
```
class Mono {  
    int num, den, exp;  
    Mono(int num, int den, int exp) {  
        ...  
    }  
}
```

$$\frac{num}{den} x^{\text{exp}}$$



# Random Unit Test Generators

- Challenge 1: Generate complex data structures
- Challenge 2: Avoid generating redundant tests
- Challenge 3: Generate test oracles (i.e., the assert statements)



# How can we generate tests with complex data structures?

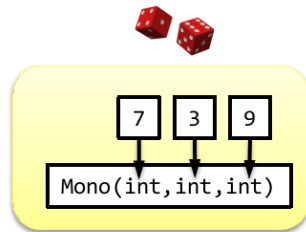
Insight: Complex data structures can be built incrementally from primitive values



# Common Generation Strategy

operation	input	output
Mono(int,int,int)	3 ints	a new Mono
Poly()	none	a new Poly
Poly plus(Mono)	a Poly, a Mono	a new Poly

random terms



```
public void test1() {  
    p = new Poly()  
        .mult(new Poly());  
  
    checkInvariant(p);  
}
```

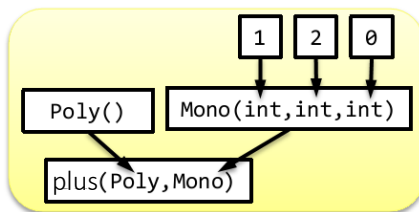
```
public void test2() {  
    p = new Poly()  
        .plus(new Mono(1,2,0));  
  
    checkInvariant(p);  
}
```

```
public void test3() {  
    m = new Mono(7,3,9);  
  
    checkInvariant(m);  
}
```

# Common Generation Strategy

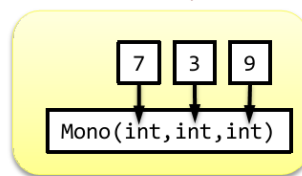
operation	input	output
Mono(int,int,int)	3 ints	a new Mono
Poly()	none	a new Poly
Poly plus(Mono)	a Poly, a Mono	a new Poly

random terms



```
public void test1() {  
    p = new Poly()  
        .mult(new Poly());  
  
    checkInvariant(p);  
}
```

```
public void test2() {  
    p = new Poly()  
        .plus(new Mono(1,2,0));  
  
    checkInvariant(p);  
}
```

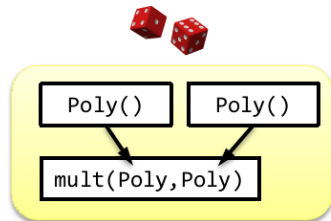


```
public void test3() {  
    m = new Mono(7,3,9);  
  
    checkInvariant(m);  
}
```

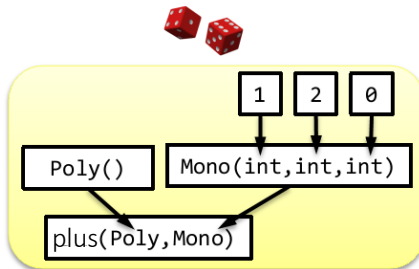
# Common Generation Strategy

operation	input	output
Mono(int,int,int)	3 ints	a new Mono
Poly()	none	a new Poly
Poly plus(Mono)	a Poly, a Mono	a new Poly

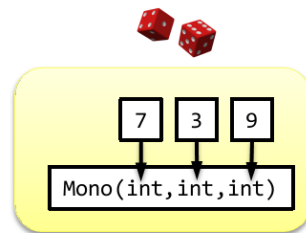
random terms



```
public void test1() {  
    p = new Poly()  
        .mult(new Poly());  
    checkInvariant(p);  
}
```



```
public void test2() {  
    p = new Poly()  
        .plus(new Mono(1,2,0));  
    checkInvariant(p);  
}
```



```
public void test3() {  
    m = new Mono(7,3,9);  
    checkInvariant(m);  
}
```

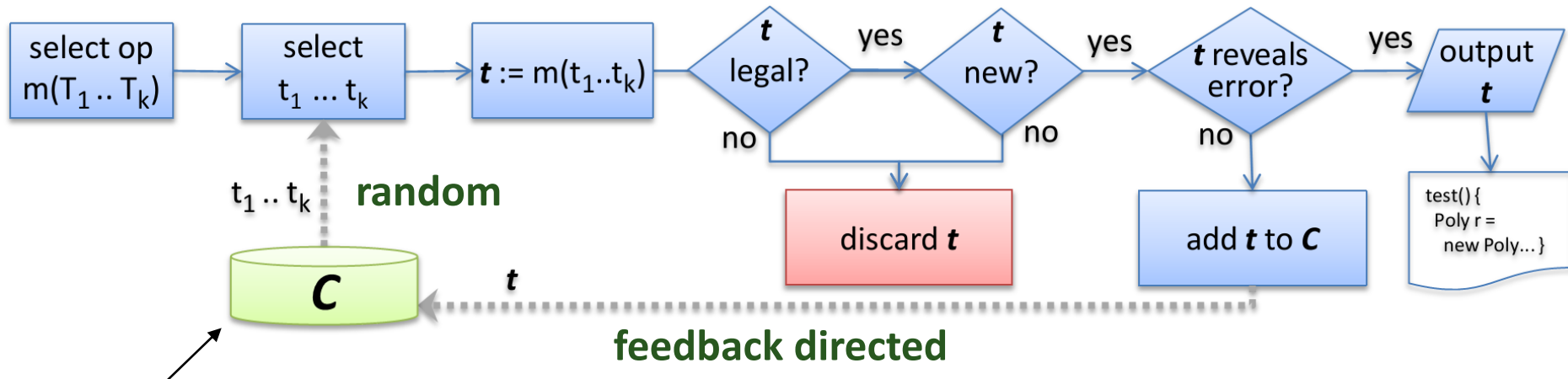


# How can we build complex data structures mechanically?

Solution: Feed generated data structures back to a repository



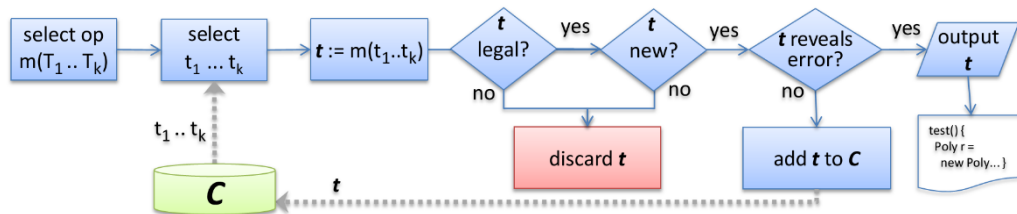
# Feedback Directed Random Test Generator



- $C$  is a repository containing possible terms used by the Test Generator.

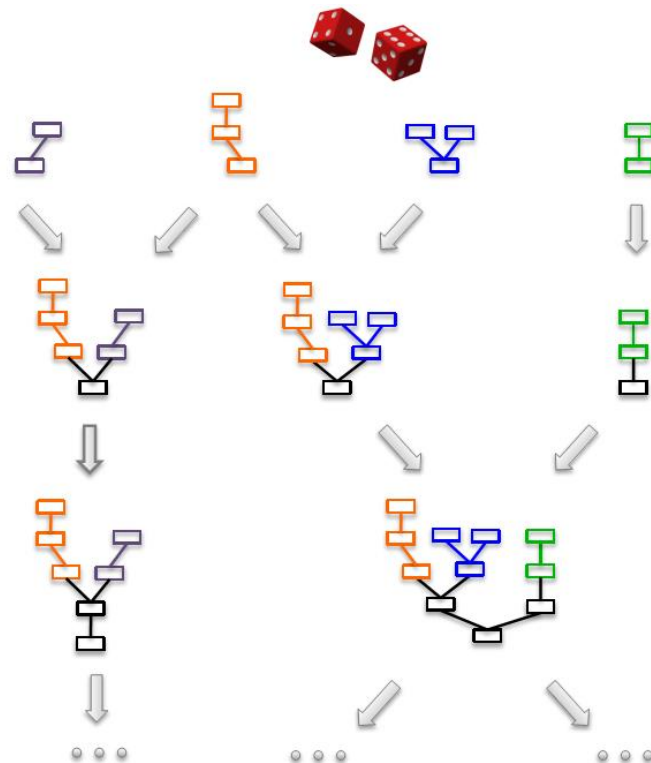
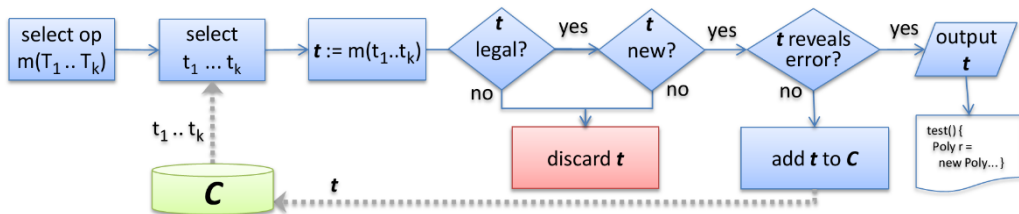
# Feedback Directed Random Test Generator

- Generate tests with simple inputs randomly from repository  $C$ .



# Feedback Directed Random Test Generator

- Generate tests with simple inputs randomly from repository  $C$ .
- Build new test inputs incrementally from previous ones.



# Feedback Directed Random Test Generator

- Carlos Pacheco [ICSE 2007, ISSTA 2008]
- Randoop (<http://randoop.github.io/randoop/>)
- Able to reveal unknown faults in widely used libraries

distinct errors revealed (Java)

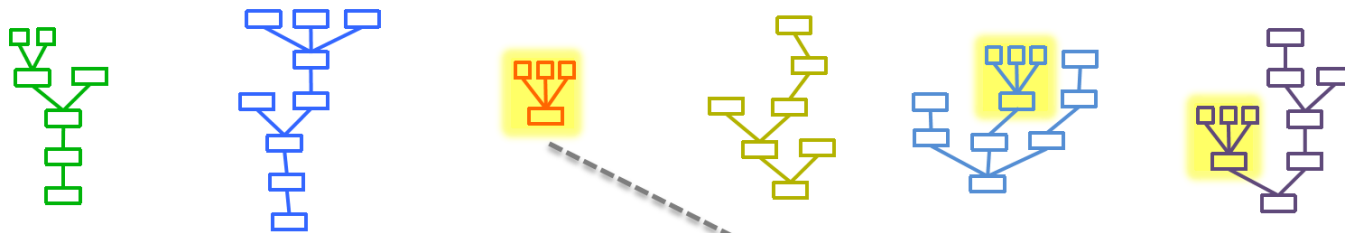
code base	Randoop	JPF (model checker)	JCrasher (random tester)
<b>Sun JDK</b> (272 classes,43KLOC)	<b>8</b>	0	1
<b>Apache libraries</b> (974 classes, 114KLOC)	<b>6</b>	1	0

distinct errors revealed (.NET)

code base	Randoop	symbolic execution unit test generator
<b>.NET library</b> (1439 classes, 185KLOC)	<b>30</b>	<b>0</b>



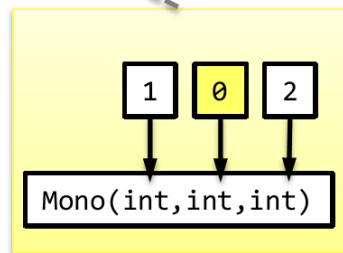
# But ...



**generates useless inputs**

› illegal, repetitive

$$\frac{num}{den} x^{exp}$$



illegal input to Mono

throws IllegalArgumentException

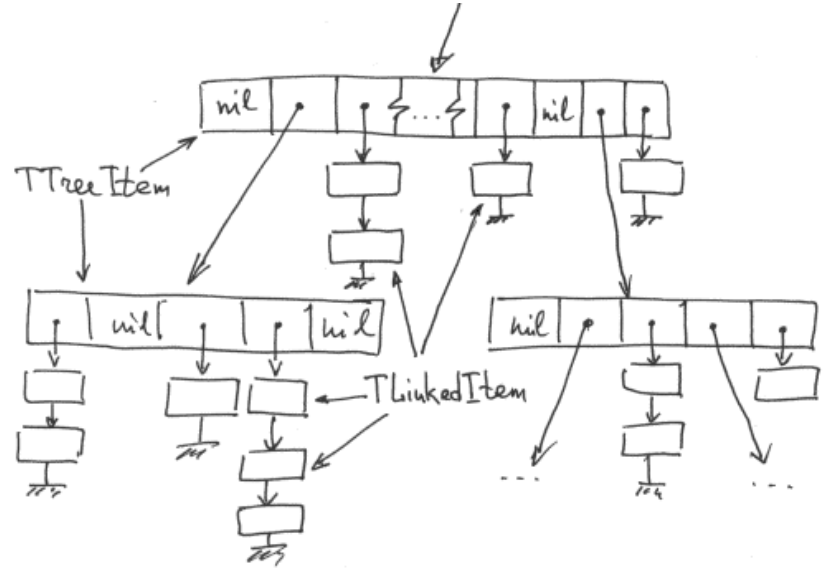
**Mono(int num, int den, int exp) { ... }**  
**Expects den  $\neq$  0 and exp  $\geq$  0**



**Input Space Pruning**

# Random Unit Test Generators

- Challenge 1: Generate complex data structures
- Challenge 2: Avoid generating redundant tests
- Challenge 3: Generate test oracle



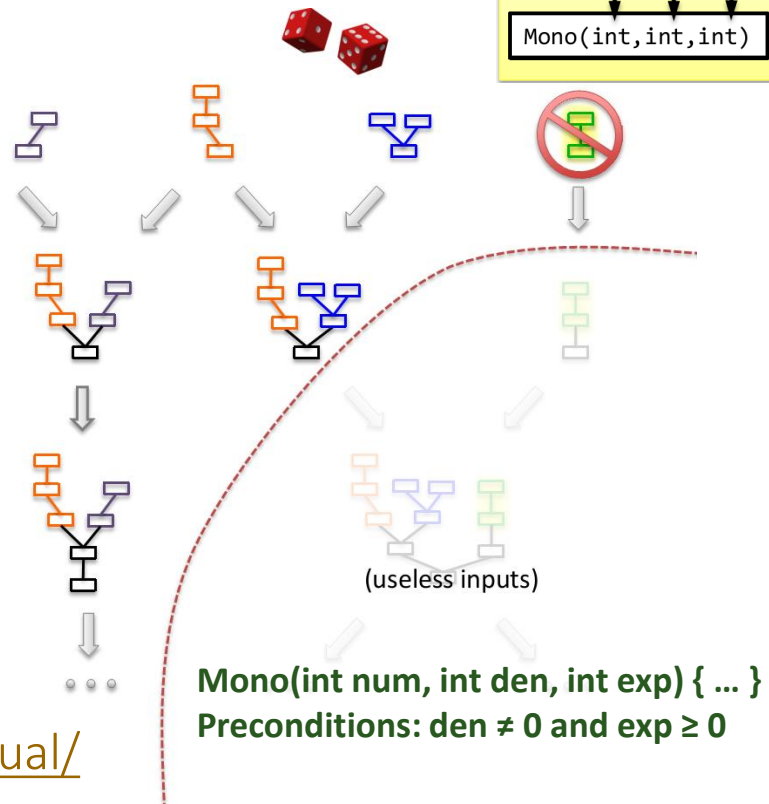
# How can we avoid redundant tests?

Insight: Prune the input space with pre-conditions and equivalence



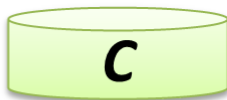
# Pruning Input Space

- Executes inputs
- Discards the ones useless for extension
  - ❑ illegal, redundant
- Prune input space
  - ❑ Specify pre-conditions on method parameters
  - ❑ See method pre-conditions at <https://randoop.github.io/randoop/manual/>



# Example: Mono(int, int, int)

select op  
 $m(T_1..T_k)$

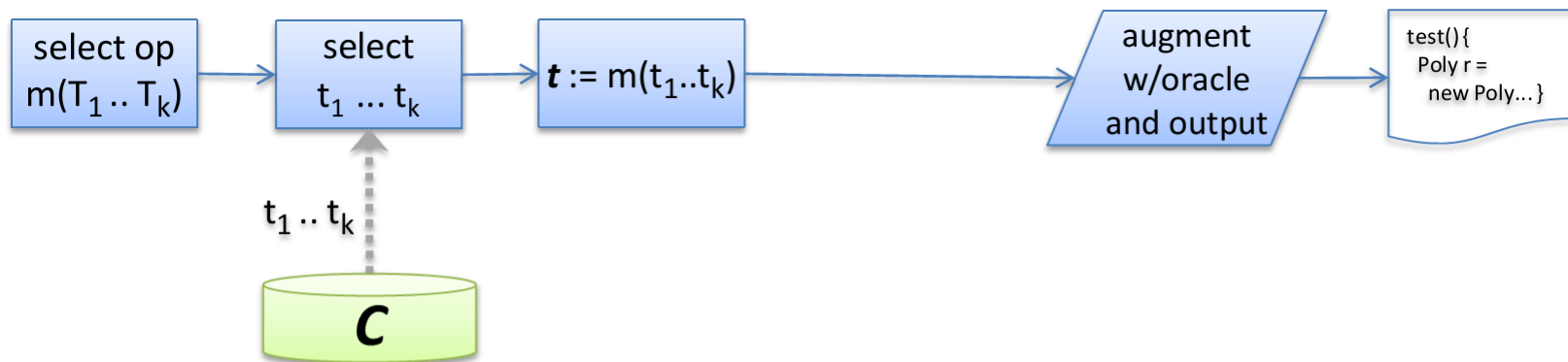


**component set of terms**

$C = \{ \emptyset, 1, 2, \text{null}, \text{false}, \text{etc.} \}$



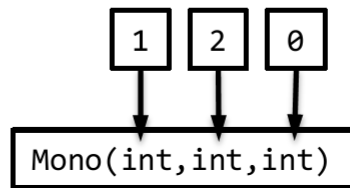
# Example: Mono(int, int, int)



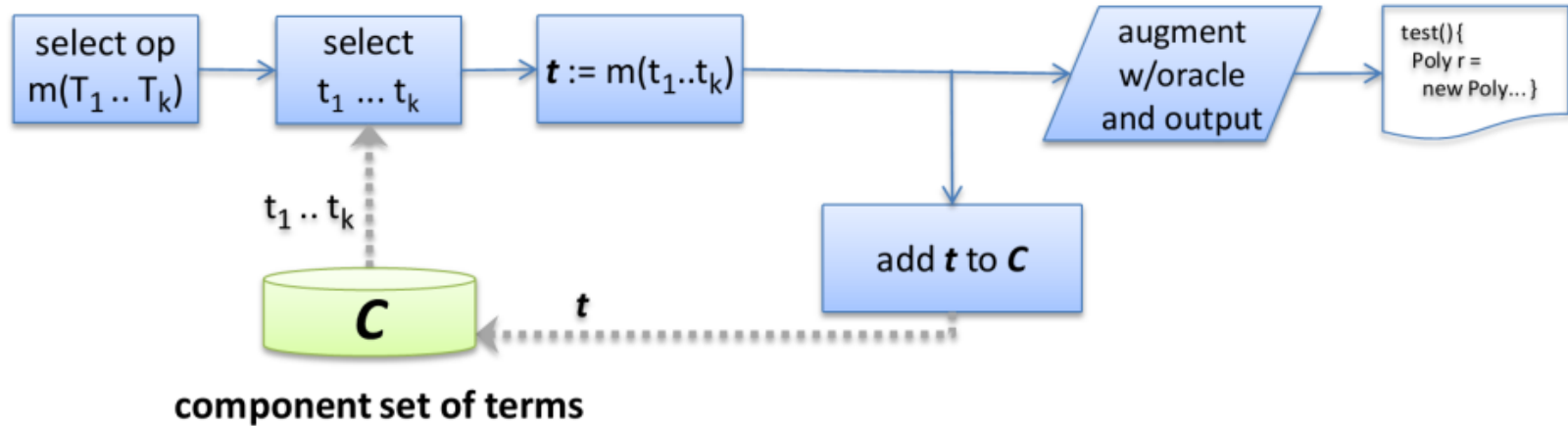
**component set of terms**

$C = \{ \emptyset, 1, 2, \text{null}, \text{false}, \text{etc.} \}$

Example:

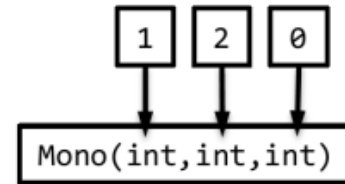


# Example: Mono(int, int, int)

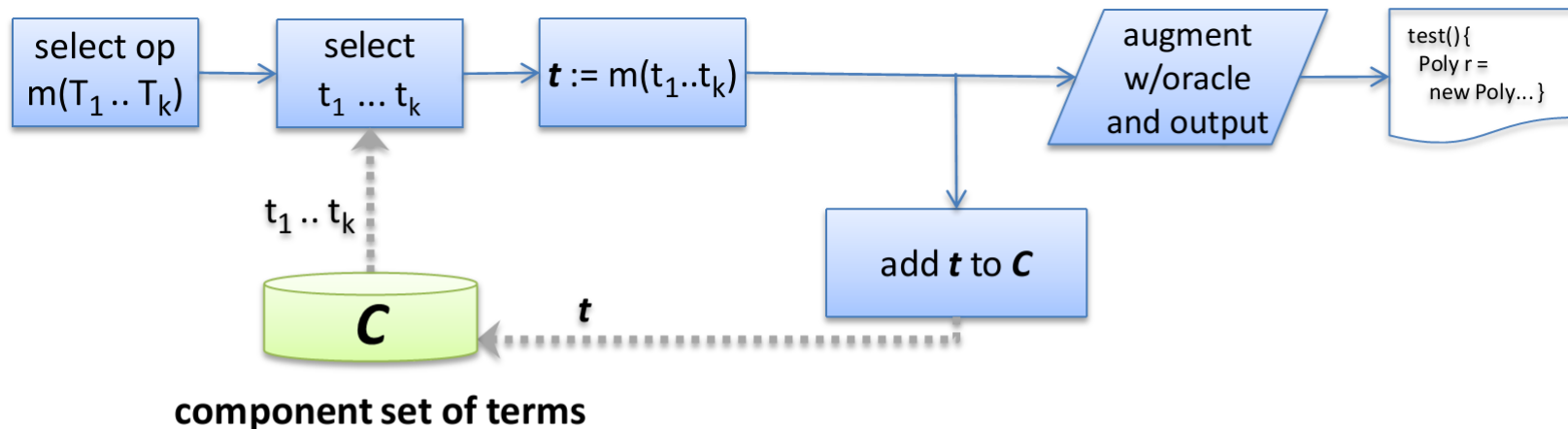


$C = \{ \emptyset, 1, 2, \text{null}, \text{false}, \text{Mono}(1, 2, \emptyset) \}$

Example:



# Example: Poly( )

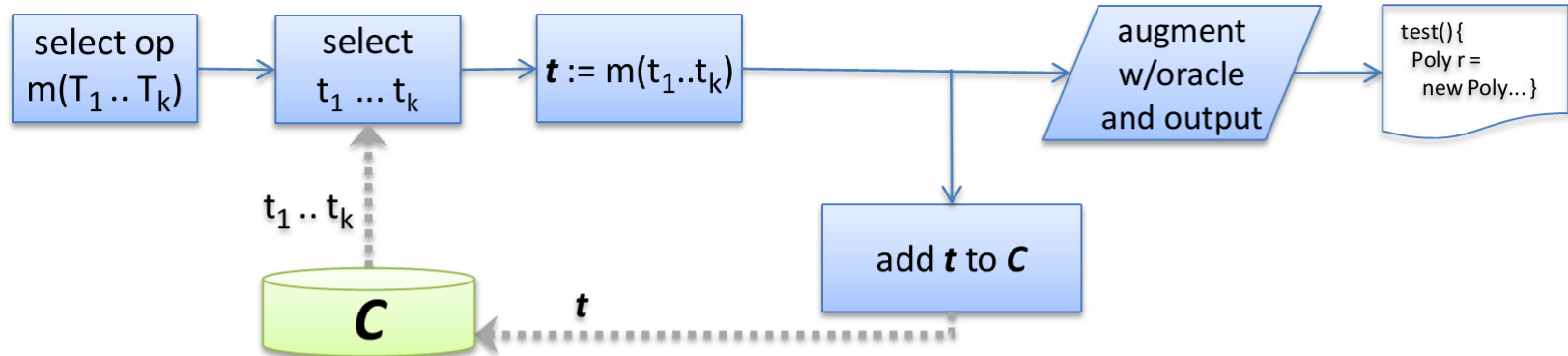


$C = \{\emptyset, 1, 2, \text{null}, \text{false}, \text{Mono}(1,2,\emptyset), \text{Poly}() \}$

Example:

`Poly()`

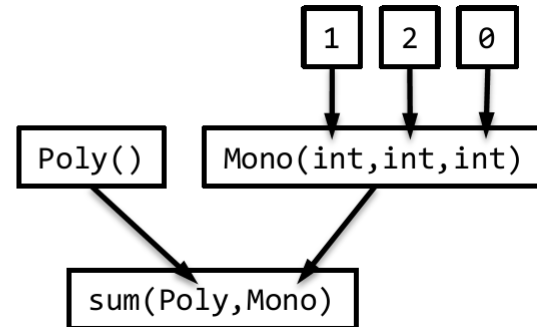
# Example: plus(Poly, Mono)



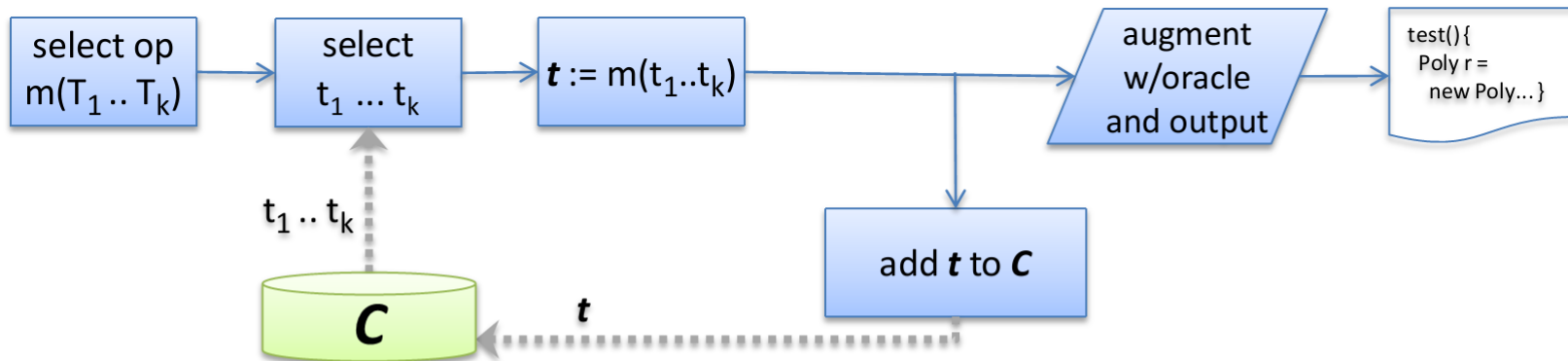
component set of terms

$C = \{ \emptyset, 1, 2, \text{null}, \text{false}, \text{Mono}(1,2,\emptyset), \text{Poly}() \}$

Example:



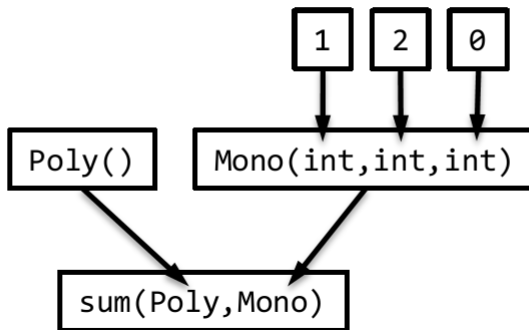
# Example: plus(Poly, Mono)



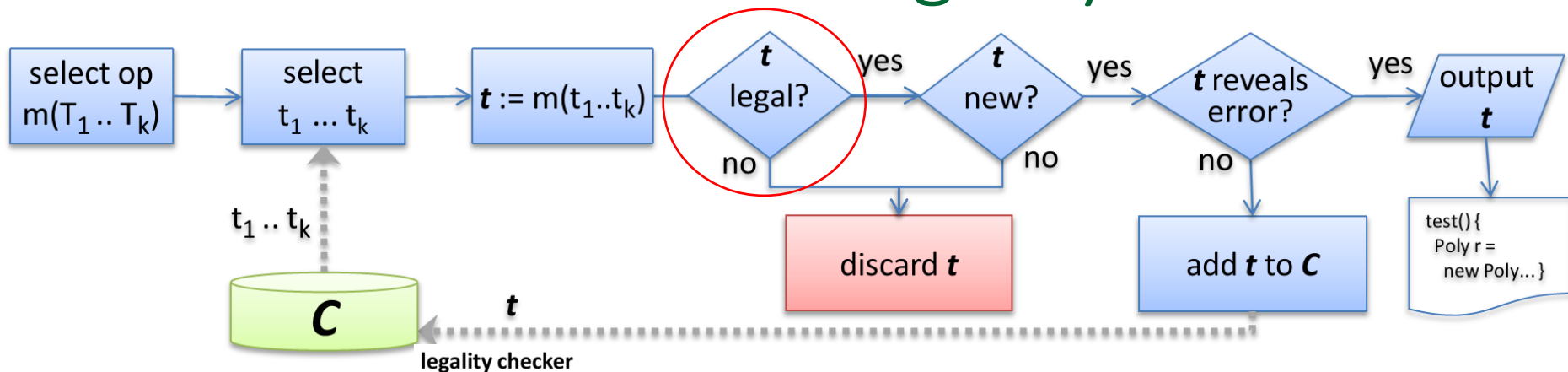
**component set of terms**

$C = \{ \emptyset, 1, 2, \text{null}, \text{false}, \text{Mono}(1,2,\emptyset), \text{Poly}(), \text{sum}(\text{Poly}(), \text{Mono}(1,2,\emptyset)) \}$

Example:



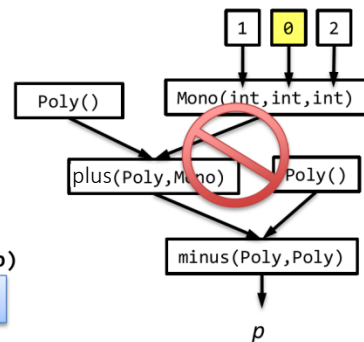
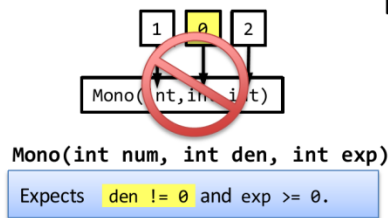
# Guided Generator - Legality



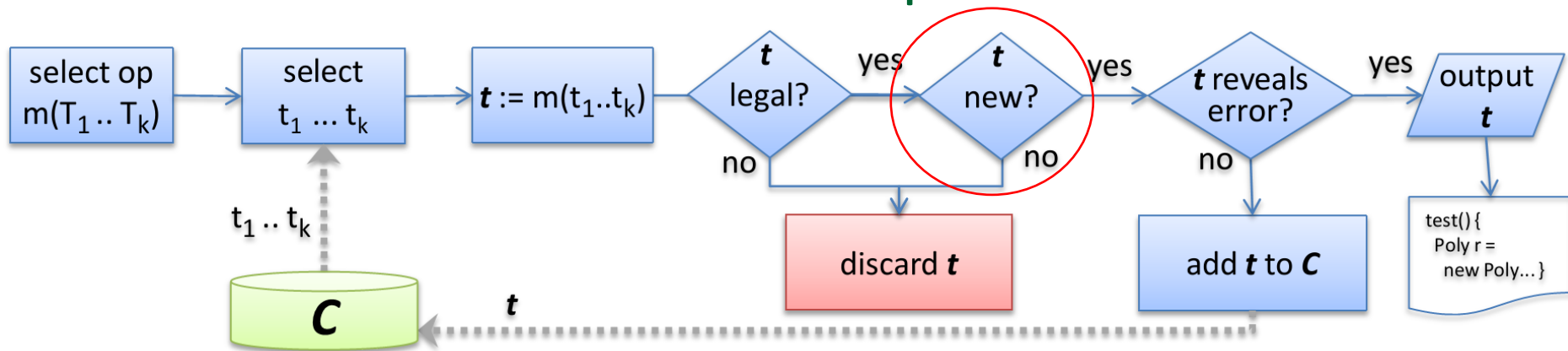
legality checker

- › determines if a term is legal/illegal
- › discard illegal terms

Violate pre-conditions of  $m()$



# Guided Generator - Equivalence



- Determines if two terms are equivalent
  - e.g.,  $\text{Mono}(1,2,1)$  and  $\text{Mono}(2,4,1)$
- Discard a term if equivalent to one in  $C$
- Implement equivalence as  $t.\text{equals}(t')$

## Applied Randoop to 13 libraries

- › built-in oracles
- › heuristic guidance
- › default time limit  
(2 minutes/library)

Outputs one test per violating method

.NET libraries specification:

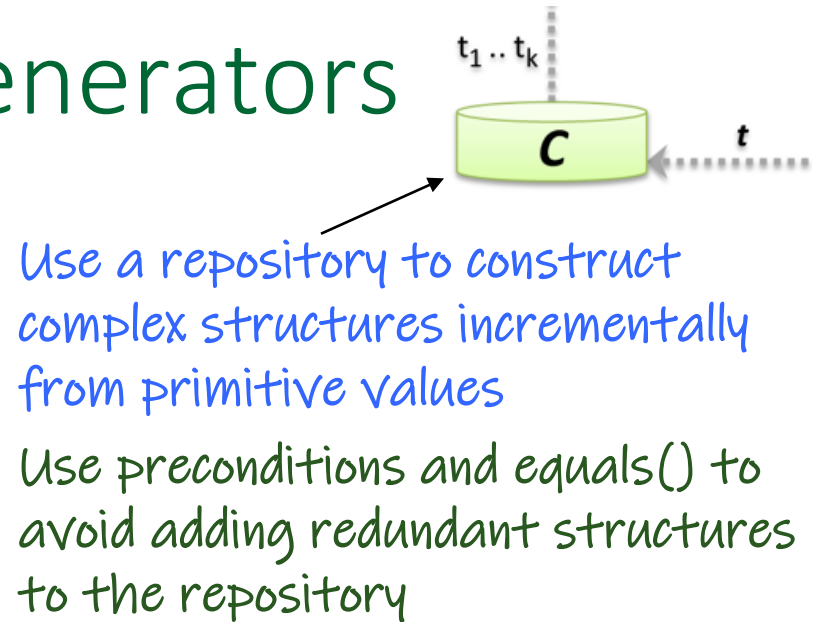
*"no method should throw NPEs, assertion violations, or IllegalMemAccess exception"*

	library	LOC	classes	tests output	errors revealed
JDK	java.util	39K	204	20	6
	java.xml	14K	68	12	2
Apache commons project	chain	8K	59	20	0
	collections	61K	402	67	4
	jelly	14K	99	78	0
	logging	4K	9	0	0
	math	21K	111	9	2
	primitives	6K	294	13	0
.NET	mscorlib	185K	1439	19	19
	system.data	196K	648	92	92
	system.security	9K	128	25	25
	system.xml	150K	686	15	15
	web.services	42K	304	41	41
TOTAL		750K	4451	411	206



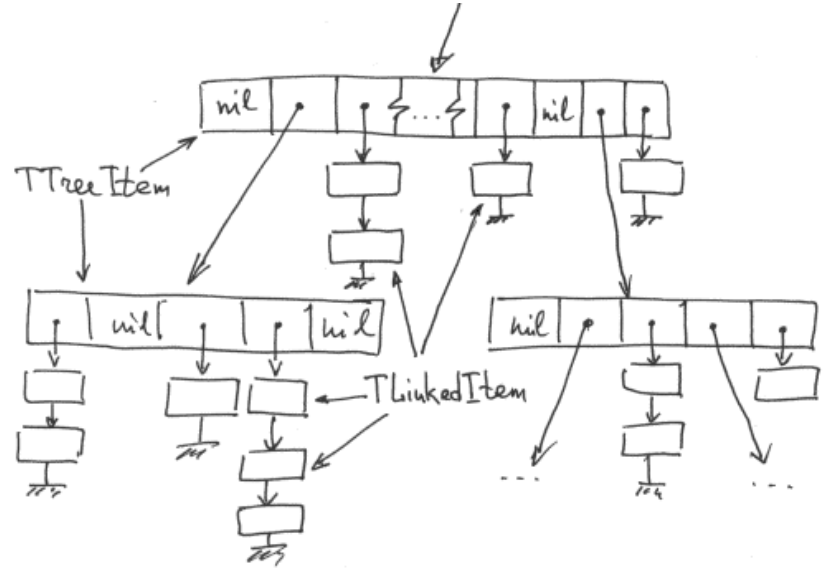
# Random Unit Test Generators

- Challenge 1: Generate complex data structures
- Challenge 2: Avoid generating redundant tests
- Challenge 3: Generate test oracles (i.e., the assert statements)



# Random Unit Test Generators

- Challenge 1: Generate complex data structures
- Challenge 2: Avoid generating redundant tests
- Challenge 3: Generate test oracle



# How can we define test oracles?

Insight: Define generic oracles  
(underfitting oracles) and regression  
oracles (overfitting oracles)



# Five Built-in Oracles

- Contracts over `Object.equals()`
  - Reflexivity: `o.equals(o) == true`
  - Symmetry: `o1.equals(o2) == o2.equals(o1)`
  - Transitivity: `o1.equals(o2) && o2.equals(o3)` implies `o1.equals(o3)`
  - Equals to null: `o.equals(null) == false`
  - It does not throw an exception

# Five Built-in Oracles

- Contracts over `Object.hashCode()`
  - `o1.equals(o2) == true` implies `o1.hashCode() == o2.hashCode()`
  - It does not throw an exception
- Contracts over `Object.toString()`
  - It does not return null
  - It does not throw an exception

# Five Built-in Oracles

- Contracts over `Object.clone()`
  - ❑ It does not throw an exception, including `CloneNotSupportedException`
  - ❑ It does not throw an exception

# Five Built-in Oracles

- Contracts over `Comparable.compareTo()` and `Comparator.compare()`
  - Reflexivity: `o.compareTo(o) == 0`
  - Anti-symmetry: `sgn(o1.compareTo(o2)) == -sgn(o2.compareTo(o1))`
  - Transitivity: `o1.compareTo(o2) > 0 && o2.compareTo(o3) > 0` implies `o1.compareTo(o3) > 0`
  - Substitutability of equals: `x.compareTo(y) == 0` implies `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`
  - Consistency with equals(): `x.compareTo(y) == 0` implies `x.equals(y)`
  - It does not throw exception

# Regression Test Oracle

- Generate assertion using the current test output

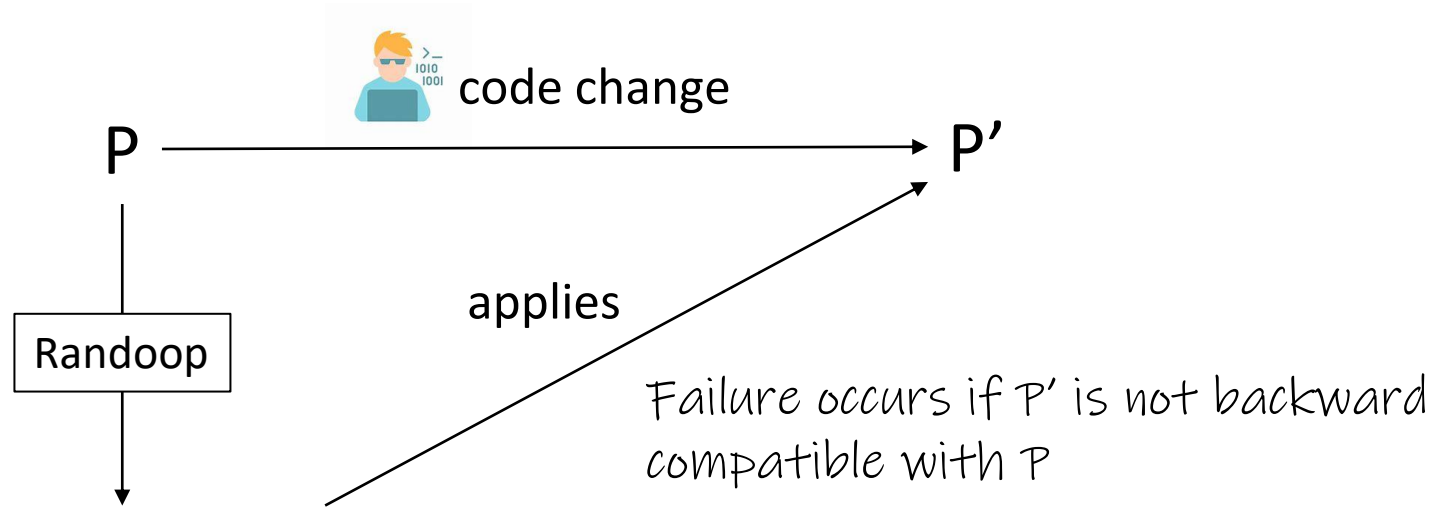
```
public String toString() {  
    String s = elements[0].toString();  
    for (int i=1; i<elements.length; ++i) {  
        s = s + " + " + elements[i].toString();  
    }  
    return s;  
}
```

@Test

```
public void test008() throws Throwable {  
    Poly poly0 = new Poly();  
    Poly poly1 = poly0.clone();  
    Poly poly2 = poly1.clone();  
    java.lang.String str3 = poly1.toString();  
    ...  
    assertNotNull(poly1);  
    assertNotNull(poly2);  
    assertEquals(str3, "(0/1)x^0 + (0/1)x^1 + ...");  
    ...  
}
```

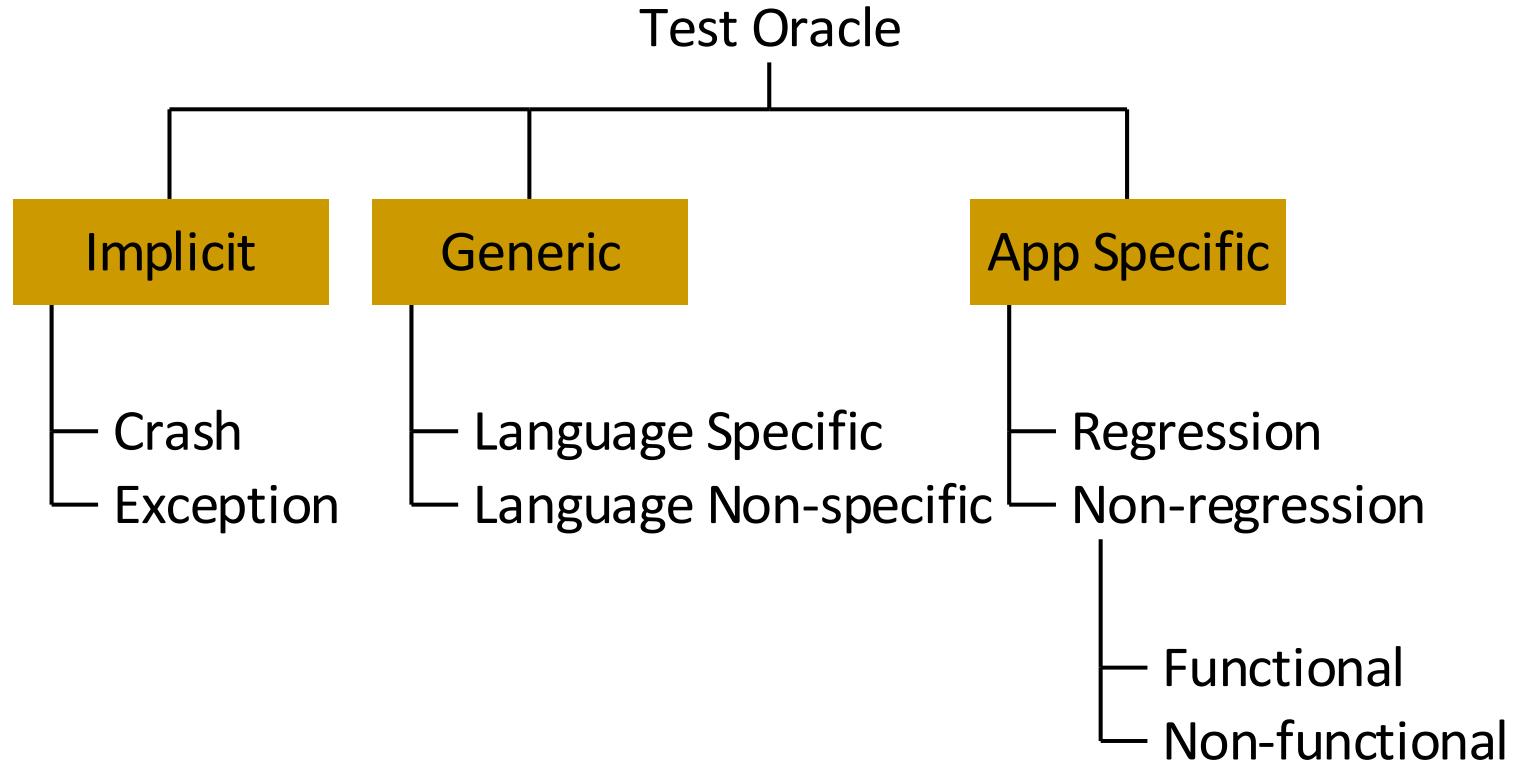


# What is Regression Test Oracle used for?



Generated tests with regression  
test oracles: {t1, t2, ...}

# Taxonomy of Test Oracles



# Evaluation and Industry Adoption

# Comparison

## JDK

- › 6 methods that create objects violating reflexivity of equality
- › 2 well-formed XML objects cause `hashCode/toString` NPEs

## Apache

- › 6 constructors leave fields unset, leading to NPEs

## .NET

- › 175 methods throw forbidden exceptions
- › 7 methods that violate reflexivity of `equals`

## .NET

- › library hangs given legal sequence of calls

*without guidance*

none revealed

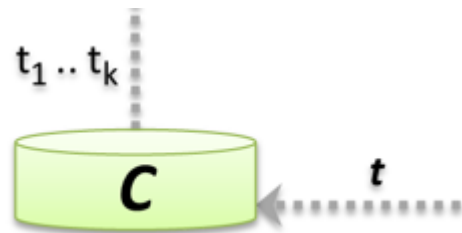
66% fewer revealed

70% fewer revealed

not revealed

# Comparison

- Why is Randoop more effective?
  - Prune useless inputs
  - Generates longer tests
  - Regression oracles



**Feedback Directed**

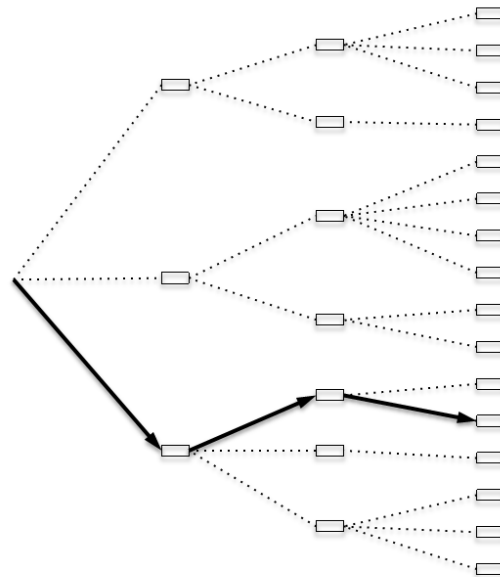
# Test Length vs. Test Effectiveness

random testing is more effective when generating **long chains** of operations

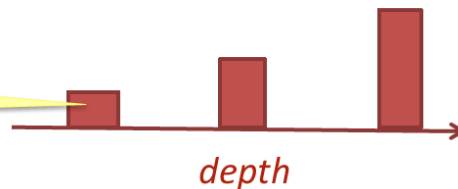
```
m=Mono(1,1,1)
```

```
p=Poly()
```

```
p2=p.add(m)
```



chances of an operation revealing an error



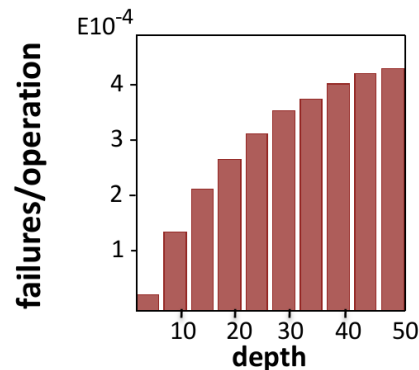
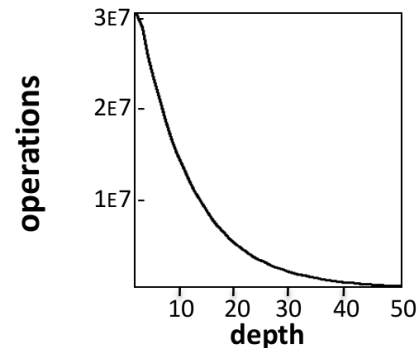
# An Experiment

- Pure random testing
  - Start from empty sequence
  - Take random steps
  - Restart if error or exception
- Exercise 10M operations per library
  - Take several days

library	classes	LOC
java.util	204	39K
collections	402	61K
primitives	294	6K
trove	336	87K
jace	164	51K

# Results

- Fail to create long chains (due to exceptions)
- Failure rate is higher at greater depths
- Ineffective → performs most operations where failure rate is the lowest

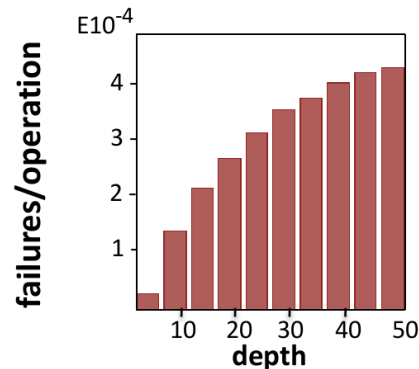
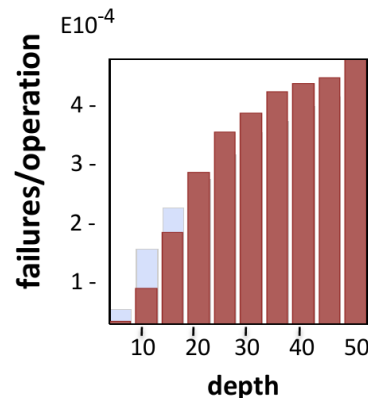
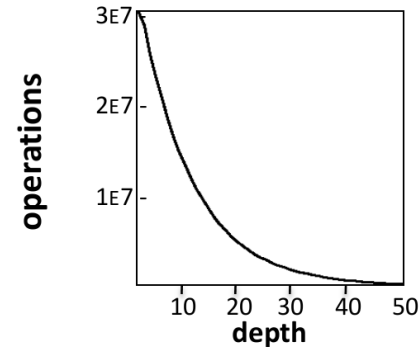
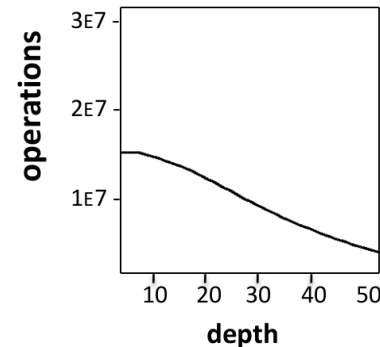


pure random



# Using Randoop (i.e., Feedback Directed Random)

- More tests with longer chains
- Able to reveal more failures (under the same budget)



Randoop

pure random

# Comparison

library	random walk	Randoop only leg.	Randoop	Include also equivalence checking Randoop
java.util	20	21	27	<b>Failures detected</b>
collections	28	37	48	
primitives	16	13	19	
trove	20	27	27	
jace	15	15	26	
<b>TOTAL</b>	<b>99</b>	<b>113</b>	<b>147</b>	

# EVALUATION AT INDUSTRY SETTING

# Other findings ...

Case study [ISSTA 2008]

- Microsoft test team
- Randoop (.NET version)
- Applied to highly-tested library
  - Tested over 5 years by 40 engineers

- *Can generated tests detect real bugs?*
- *Is automated test generation cost effective?*



Findings: revealed more errors in 15 hours than the team typically discovers in 1 person-year of effort

# Case Study Statistics

## ■ Facts

- ❑ Human time interacting with Randoop: 15 hours
- ❑ CPU time running Randoop: 150 hours
- ❑ Total distinct method sequences: 4 million
- ❑ New errors revealed: 30

**Interacting with Randoop  
Inspecting the resulting tests  
Discarding redundant failures**

## ■ Randoop

- ❑ 30 new errors in 15 hours of human effort
- ❑ 1 new error for ½ hour effort

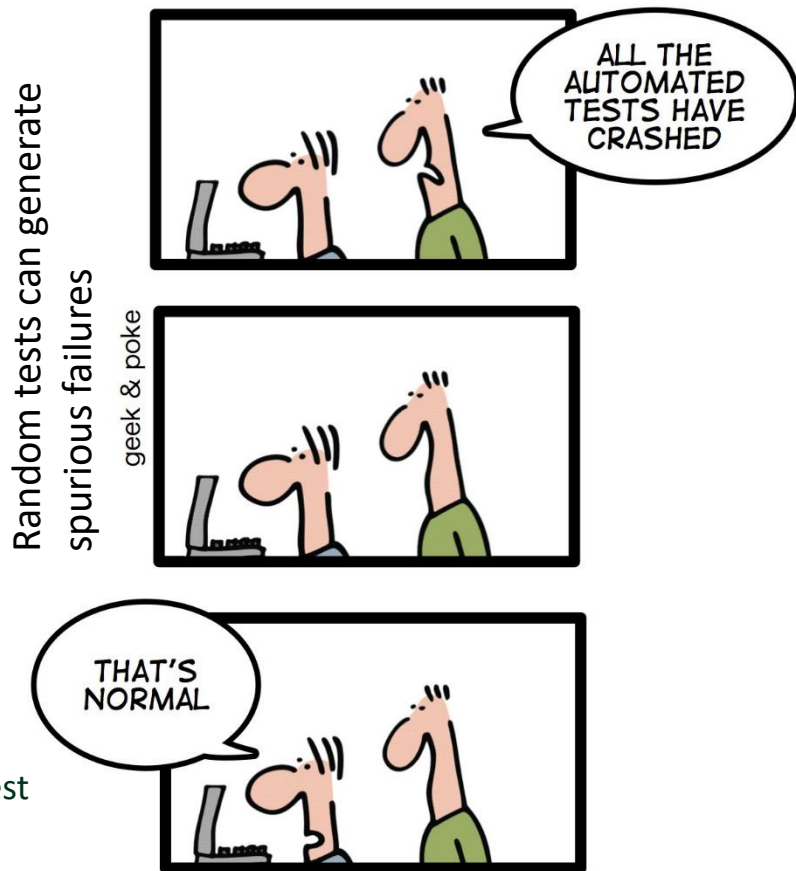
## ■ Existing team methods

- ❑ 20 new errors per year
- ❑ 1 new error for 100 hours human effort

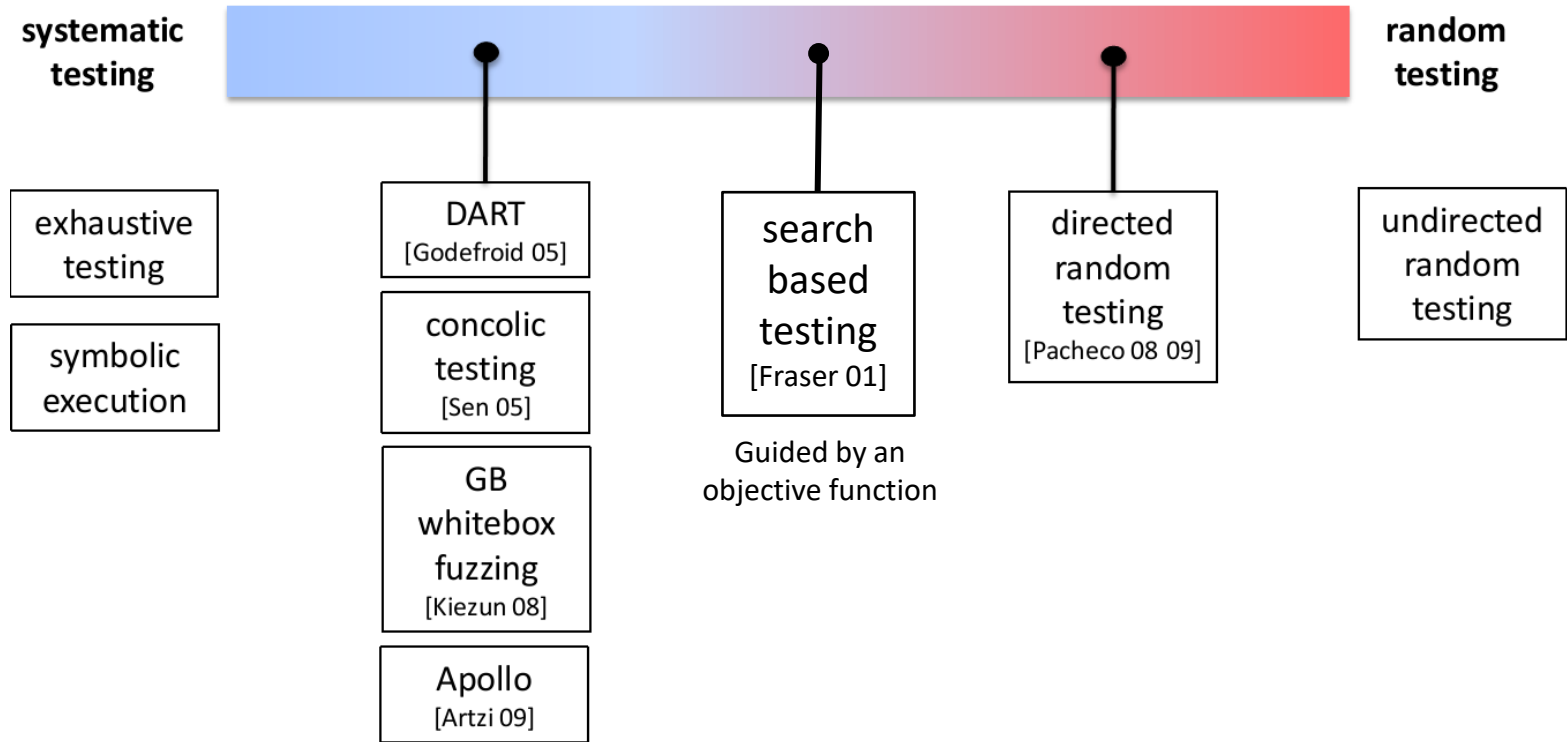
# Limitations

- Can generate spurious failures
  - Unaware of implicit pre-conditions\*
- Weak assert statements
- Low coverage ( $< 50\%$ ) for reactive programs
  - Android apps, GUI applications
  - Not driven by coverage

\*Mijung Kim, Shing-Chi Cheung, Sunghun Kim. Which Generated Test Failures Are Fault Revealing? Prioritizing Failures Based on Inferred Precondition Violations using PAF. In ESEC/FSE 2018.



# Spectrum of Testing Techniques



# Using Randoop

<https://randoop.github.io/randoop/manual/index.html>

Video: <https://www.youtube.com/watch?v=nPdb-72-EJY>



# Further Readings

- Other popular random testing tools
  - Jubula for GUI testing (<http://www.eclipse.org/jubula/>)
  - Monkey & Stocat for Android software testing  
(<http://developer.android.com/tools/help/monkey.html>)  
(<https://github.com/tingsu/Stocat>)
  - Sapienz for Android random testing (<https://github.com/Rhapsod/sapiens>)
    - Facebook prototype to be replaced by an official release in 2019
- API invariance inference
  - Robillard, M.P.; Bodden, E.; Kawrykow, D.; Mezini, M.; Ratchford, T.,  
"Automated API Property Inference Techniques," *IEEE Transactions on Software Engineering*, vol.39, no.5, pp.613-637, May 2013.