

# ARTOO: Adaptive Random Testing for Object-Oriented Software

Authors: Ilinca Ciupa, Andreas Leitner, Manuel Oriol, Bertrand Meyer

Presentation by:

Daniela Costa (110882)

João Pedro Leite (110901)

Guilherme Branco (110850)

# What is random testing?

Create test cases from **randomly generated inputs**.

- Very fast with little overhead;
- Easy to automate;

Other strategies:

- Genetic Programming
- Symbolic Execution combined with Constraint Solving

**Problem** – truly random generation would require a **great number of test cases** to achieve coverage!

# How is adaptive RT different?

Defines some way to **guide the input generation** in order to **evenly distribute** the resulting test cases.

Can find the same number of faults in **half the tests** compared to purely random testing.

Must be **easy to automate!**

# ARTOO

Apply Adaptive Random Testing to find faults in **Object Oriented applications**.

**Challenge** - to evenly distribute the test cases it is necessary to define some notion of **distance between objects**:

- Complex data types;
- Different among themselves;

# Object Distance

# Distance Between Objects

How different those objects are in terms of:

- Values – calculate the **elementary distance** between direct values or values of references;

Specific formula for each data type.

Elementary  
distance for  
id = 19

```
class Student {  
    Int id;  
    String name;  
}
```



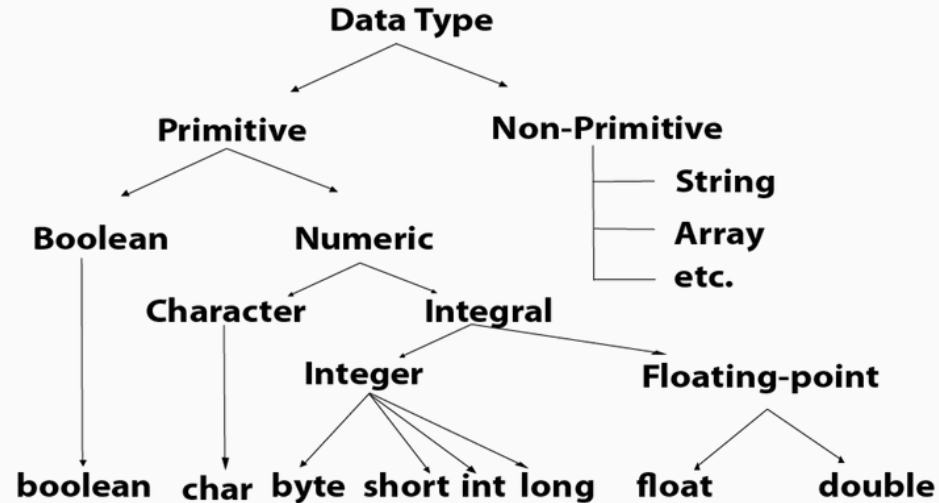
```
Student Dani = new Student( id=110882 )
```

```
Student Leite = new Student( id= 110901 )
```

# Distance Between Objects

How different those objects are in terms of:

- Dynamic types – calculate the **type distance**, independent of value, by finding the closest common ancestor between the types;



Student Dani = new Student( id=110882 )

Student Leite = new Student( name="Leite" )

Type  
distance = 7

# Distance Between Objects

How different those objects are in terms of:

- Attributes or references – calculate the **field distance** by traversing them recursively until their primitive types;

```
class Student {  
    Int id;  
    Double avg;  
}
```

```
class Course {  
    Student students[];  
    Int capacity;  
}
```

This is applied to pairs of fields:

- With matching types – calculate elementary or field distance;
- Different types – calculate type distance;



# Distance Between Objects

The final distance between two objects is obtained through a combination function, which is a weighted sum of the calculated distances.

- Distances may be normalized to the **same bound interval**, with minimum 0;
- Definition of weights depends on the application being tested;

# ART00 – Generating Objects for Input

Uses object distance to generate inputs for test cases that are **evenly distributed**:

- Keep track of previously used objects;
- From an **object pool**, choose object for next input with **highest ~~average~~ total distance** to those already used;

Similar approach to the original ART.

# Implementation

# AutoTest

- Eiffel
  - Object Oriented
  - Design by Contract
- AutoTest
  - Automatic unit testing
    - Randomly generated inputs
  - Uses contracts as test oracles
  - Two-process model:
    - Instructor
    - Interpreter



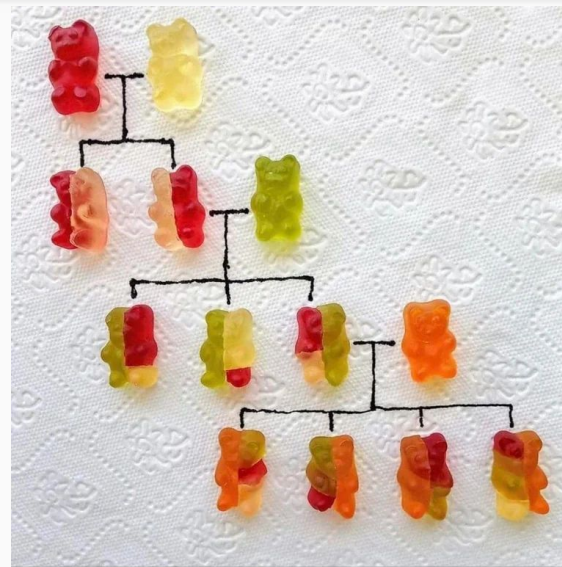
# AutoTest - Random Input Generation

- Object pool
- Routines require target objects and methods
  - Objects are either fetched from the pool or created
    - The decision is probabilistic
      - (25% creation chance is recommended)
    - Constructors are randomly selected
  - Primitive objects are special
    - Can select from entire value spectrum
    - or specific set of troublesome inputs



# AutoTest - Random Input Generation

- Diversification
  - Randomly select object from the pool,
  - Randomly apply state mutating methods,
  - 50/50 chance of mutation on method call.
- Methods and constructors must obey contracts
  - Any mutation must also obey the contracts!
  - Diversification is a safe way to acquire new objects.



# AutoTest - ARTOO

- ARTOO is a plugin for AutoTest
  - Only changes the selection and creation algorithms.
  - Same probabilities for diversification and creation.
- Adapts original ARTOO algorithm
  - New objects can enter the pool.
  - Any recursion is capped at 2 steps.



# ART00 - Example

```
class BANK_ACCOUNT
create
  make

feature -- Bank account data
  owner: STRING
  balance: INTEGER

feature -- Initialization
  make (s: STRING; init_bal: INTEGER) is
    -- Create a new bank account.
  require
    positive_initial_balance : init_bal >= 0
    owner_not_void: s /= Void
  do
    owner := s
    balance := init_bal
  ensure
    owner_set: owner = s
    balance_set: balance = init_bal
end
```

```
transfer (other_account: BANK_ACCOUNT; sum: INTEGER
) is
  -- Transfer 'sum' to 'other_account'.
require
  can_withdraw: sum <= balance
do
  balance := balance - sum
  other_account.deposit (sum)
ensure
  balance_decreased: balance < old balance
  sum_deposited_to_other_account : other_account.balance
    > old other_account.balance
end
```



# ART00 - Transfer Example

ba1: *BANK\_ACCOUNT*, *ba1.owner*="A", *ba1.balance*=675234  
*ba2*: *BANK\_ACCOUNT*, *ba2.owner*="B", *ba2.balance*=10  
ba3: *BANK\_ACCOUNT*, *ba3.owner*="O", *ba3.balance*=99  
*ba4* = Void  
*i1*: *INTEGER*, *i1* = 100  
*i2*: *INTEGER*, *i2* = 284749  
*i3*: *INTEGER*, *i3* = 0  
*i4*: *INTEGER*, *i4* = -36452  
*i5*: *INTEGER*, *i5* = 1

▶ *ba1*: *BANK\_ACCOUNT*, *ba1.owner*="A", *ba1.balance*=675235  
▶ *ba2*: *BANK\_ACCOUNT*, *ba2.owner*="B", *ba2.balance*=10  
▶ *ba3*: *BANK\_ACCOUNT*, *ba3.owner*="O", *ba3.balance*=98  
*ba4* = Void  
*i1*: *INTEGER*, *i1* = 100  
*i2*: *INTEGER*, *i2* = 284749  
*i3*: *INTEGER*, *i3* = 0  
*i4*: *INTEGER*, *i4* = -36452  
▶ *i5*: *INTEGER*, *i5* = 1

ba3.transfer(ba1, i5)



# ART00 - Example

*ba1*: *BANK\_ACCOUNT*, *ba1.owner*="A", *ba1.balance*=675234  
*ba2*: *BANK\_ACCOUNT*, *ba2.owner*="B", *ba2.balance*=10  
*ba3*: *BANK\_ACCOUNT*, *ba3.owner*="O", *ba3.balance*=99  
*ba4* = *Void*  
*i1*: *INTEGER*, *i1* = 100  
*i2*: *INTEGER*, *i2* = 284749  
*i3*: *INTEGER*, *i3* = 0  
*i4*: *INTEGER*, *i4* = -36452  
*i5*: *INTEGER*, *i5* = 1

Greatest distance to *i5*? ***i2!***  
Greatest distance to *ba1* is ***ba4***,  
because *void* has max distance.

▶ *ba1*: *BANK\_ACCOUNT*, *ba1.owner*="A", *ba1.balance*=675235  
▶ *ba2*: *BANK\_ACCOUNT*, *ba2.owner*="B", *ba2.balance*=10  
▶ *ba3*: *BANK\_ACCOUNT*, *ba3.owner*="O", *ba3.balance*=98  
*ba4* = *Void*  
*i1*: *INTEGER*, *i1* = 100  
*i2*: *INTEGER*, *i2* = 284749  
*i3*: *INTEGER*, *i3* = 0  
*i4*: *INTEGER*, *i4* = -36452  
▶ *i5*: *INTEGER*, *i5* = 1

Greatest distance to *ba3*? (**target non-void**)  
distance *ba2*: owner: 1, balance: 88  
distance *ba1*: owner: 1, balance: 675137

next target is  
***ba1!***

# ART00 - Example

```

ba1: BANK_ACCOUNT, ba1.owner="A", ba1.balance=675235
ba2: BANK_ACCOUNT, ba2.owner="B", ba2.balance=10
ba3: BANK_ACCOUNT, ba3.owner="O", ba3.balance=98
ba4 = Void
i1: INTEGER, i1 = 100
i2: INTEGER, i2 = 284749
i3: INTEGER, i3 = 0
i4: INTEGER, i4 = -36452
i5: INTEGER, i5 = 1

```

▶ *ba1*: *BANK\_ACCOUNT*, *ba1.owner*="A", *ba1.balance*=675235  
 ▶ *ba2*: *BANK\_ACCOUNT*, *ba2.owner*="B", *ba2.balance*=10  
 ▶ *ba3*: *BANK\_ACCOUNT*, *ba3.owner*="O", *ba3.balance*=98  
 ▶ *ba4* = Void  
 ▶ *i1*: *INTEGER*, *i1* = 100  
 ▶ *i2*: *INTEGER*, *i2* = 284749  
 ▶ *i3*: *INTEGER*, *i3* = 0  
 ▶ *i4*: *INTEGER*, *i4* = -36452  
 ▶ *i5*: *INTEGER*, *i5* = 1

$$ba1.transfer(ba4, i2)$$


## Void receiver: Found a fault!

**No change!**

# ART00 - Example

```
ba1: BANK_ACCOUNT, ba1.owner="A", ba1.balance=675235
ba2: BANK_ACCOUNT, ba2.owner="B", ba2.balance=10
ba3: BANK_ACCOUNT, ba3.owner="O", ba3.balance=98
ba4 = Void
i1: INTEGER, i1 = 100
i2: INTEGER, i2 = 284749
i3: INTEGER, i3 = 0
i4: INTEGER, i4 = -36452
i5: INTEGER, i5 = 1
```

Greatest distance to i2? **i4!**  
Greatest distance to ba1 is **ba2**,  
because it has less balance.



```
ba1: BANK_ACCOUNT, ba1.owner="A", ba1.balance=675235
ba2: BANK_ACCOUNT, ba2.owner="B", ba2.balance=10
ba3: BANK_ACCOUNT, ba3.owner="O", ba3.balance=98
ba4 = Void
i1: INTEGER, i1 = 100
i2: INTEGER, i2 = 284749
i3: INTEGER, i3 = 0
i4: INTEGER, i4 = -36452
i5: INTEGER, i5 = 1
```

Greatest distance to ba1? (**target non-void**)  
Can only choose ba2.

next target is  
**ba2!**

# ART00 - Example

*ba1: BANK\_ACCOUNT, ba1.owner="A", ba1.balance=675235*  
*ba2: BANK\_ACCOUNT, ba2.owner="B", ba2.balance=10*  
*ba3: BANK\_ACCOUNT, ba3.owner="O", ba3.balance=98*  
*ba4 = Void*  
*i1: INTEGER, i1 = 100*  
*i2: INTEGER, i2 = 284749*  
*i3: INTEGER, i3 = 0*  
*i4: INTEGER, i4 = -36452*  
*i5: INTEGER, i5 = 1*

*ba1: BANK\_ACCOUNT, ba1.owner="A", ba1.balance=675235*  
*ba2: BANK\_ACCOUNT, ba2.owner="B", ba2.balance=10*  
*ba3: BANK\_ACCOUNT, ba3.owner="O", ba3.balance=98*  
*ba4 = Void*  
*i1: INTEGER, i1 = 100*  
*i2: INTEGER, i2 = 284749*  
*i3: INTEGER, i3 = 0*  
*i4: INTEGER, i4 = -36452*  
*i5: INTEGER, i5 = 1*

*ba2.transfer(ba2, i4)*

**Negative amount: Found a fault!**

**No change!**

Also transferring to itself

# ART00 - Example

- Testing would still continue. Only showed a few iterations.
- No object creation or diversification in this example, but you get the idea!

# Experimental Results

# Experimental setup

- **Evaluated on real-world code** – EiffelBase
  - Proves **practical applicability**
- Tested against **directed random strategy** (RAND), more efficient than purely random





# Evaluation

Criteria:

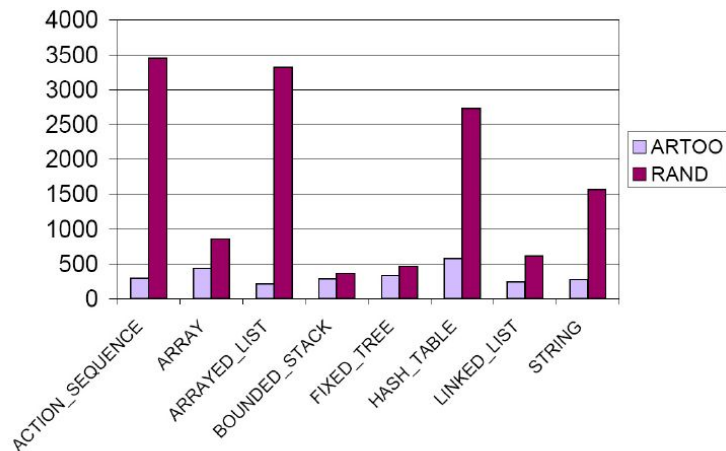
- **Number of tests** until first fault is found
- **Time elapsed** until first fault is found

Chosen because:

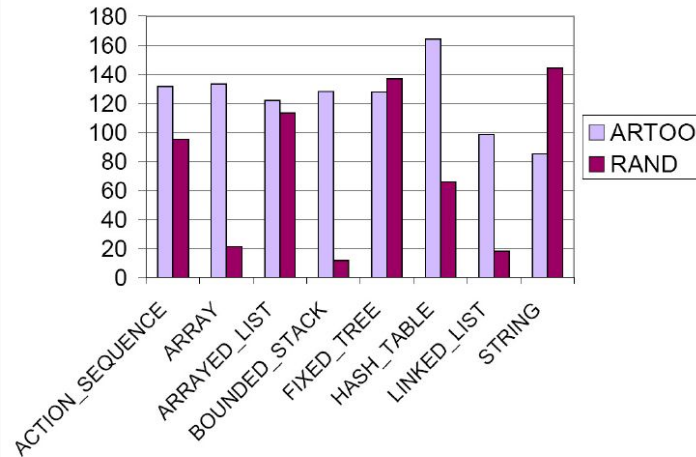
- **Fault-detecting ability** is most important
- Industrial software projects have **time constraints**

Times reported include **time spent generating and selecting test cases**, to resemble **performance in practice**

# Results



**Figure 2:** Comparison of the average number of tests cases to first fault required by the two strategies for every class. ARTOO constantly outperforms RAND.



**Figure 3:** Comparison of the average time to first fault required by the two strategies for every class. RAND is generally better than ARTOO.

# Results

Class	Routine	Tests to first fault	Time to first fault (seconds)	#faults
ARRAYED_LIST	remove	167	46	1
FIXED_TREE	child_is_last	717	283	1
FIXED_TREE	duplicate	422	134	1
STRING	grow	492	163	2
STRING	multiply	76	17	2

**Table 4: Faults which only ARTOO finds**

- The inverse is also true and there are faults which only RAND finds in the same span of time

Class	Timeout (minutes)	StDev(NumberFoundFaults)	
		ARTOO	RAND

**Table 5: Standard deviations of numbers of found faults for each strategy due to the influence of the seed for the pseudo-random number generator, and the average and standard deviation of the standard deviations for each strategy. ARTOO is generally less sensitive to the choice of seed.**

# Discussion

- ARTOO and RAND have different strengths
- Better results by **combining them**:
  - Use RAND until it is unlikely to discover new faults
  - Use ARTOO to find remaining faults
- **ARTOO** should be preferred over RAND for **computation-intensive** contexts
  - **Lower number of tests runnable per time-unit**
  - **Finding faults with less tests** is more important

# Future Work

# Optimizations

- Obtain more generalizable results:
  - Evaluate with **more classes**
  - Use **more than 5 random seeds**
- Use **clustering algorithms** to cluster objects by distance
  - Computation of distances between clusters **reduces overhead**

A preliminary implementation of this testing strategy shows an average improvement of the time to first fault over ARTOO of 25% at no cost in terms of faults found.

# Optimizations

- Use information from **manual tests** to guide further research
  - e.g. consider **distances** to the **manual inputs** and to **previous automatic inputs**

Preliminary experiments using a first prototype implementation of this testing strategy show that it can reduce the number of tests to first fault by an average factor of 2 compared to the basic implementation of ARTOO described above.

- Defining object distance in a **less computationally intensive way**
  - May require fewer tests and less time to first fault

# Optimizations

- Integrate **semantics** into the computation of **object distance**
  - Requires human intervention
  - Enriches the model and its **accuracy**
  - Allows **finer-grained control** over the testing process

Some support for this is already available through the constants used in the object distance calculation, whose values can easily be changed for fine tuning the distance.



# Questions