Optimizing unit test execution in large software programs using dependency analysis

Taesoo Kim,
Ramesh Chandra and Nickolai Zeldovich

MIT CSAIL

Running unit tests takes too long

> It's our policy to make sure **all tests pass at all times**.

django

- Large software programs often require **running full unit tests** for each commit
- But, unit tests take about **10 min** in Django
- With **our work**, it can be done within **2 sec**!

Current approaches for shortening testing time

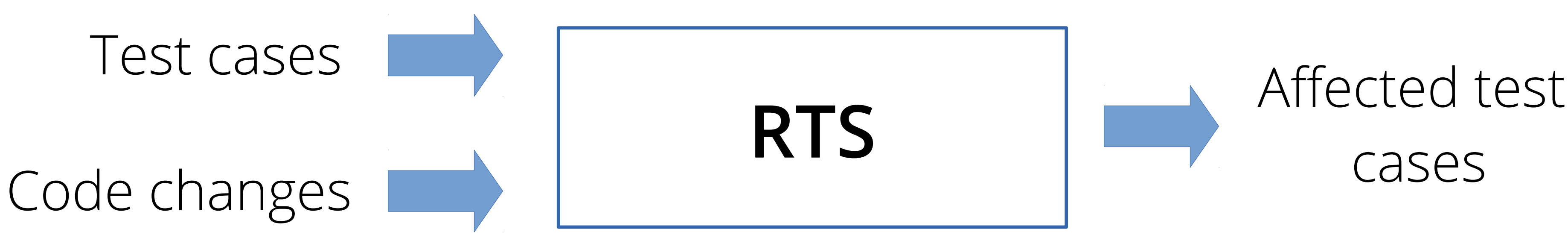
- **Modular unit tests (e.g., testsuite)**
  - Run a certain set of unit tests that might be affected
- **Test bot (e.g., gtest, autotest)**
  - Run unit tests remotely and get the results back

Problem: current approaches are very limited

- **Manual efforts involved**
  - Maintaining multiple test suites
- **Overall testing still takes too long**
  - Waiting for Test bot to complete full unit testing

Research: regression test selection (RTS)

- **Goal:** run **only necessary** tests instead of full tests
  - identify test cases whose results might change due to the current code modification
  - **Step 1**: analyze test cases (e.g., execution traces)
  - **Step 2**: syntactically analyze code changes
  - **Step 3**: output the affected test cases



Problem: RTS techniques are never adopted in practice

- **“Soundness” of RTS techniques kills adoption**
  - Soundness means **no false negatives**
  - Impose non-negligible perf. overheads (analysis/runtime)
  - Select lots of test cases (particularly in dynamic languages)

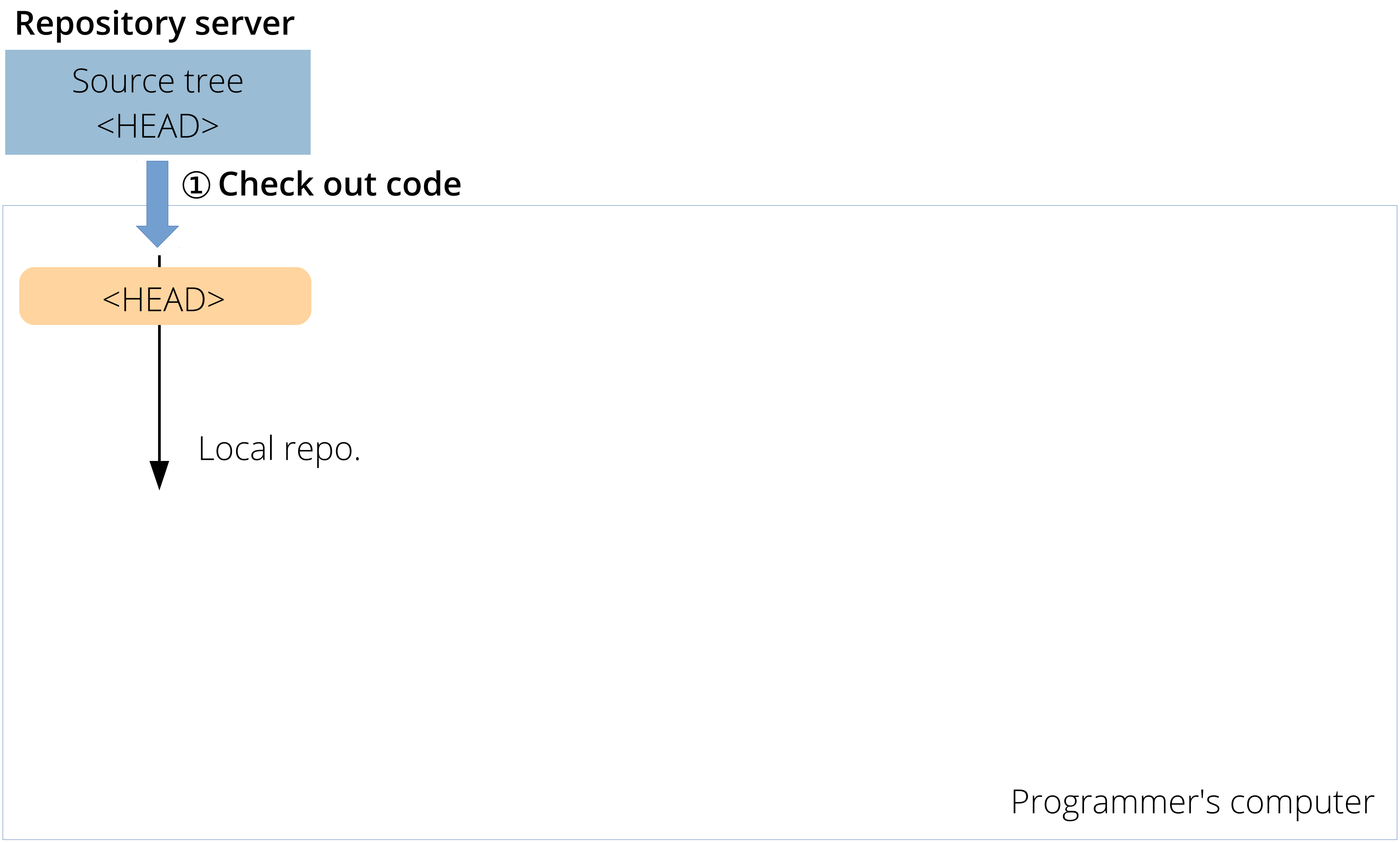
  - e.g., changes in **a global variable →** run **all** test cases

Goal: make RTS practical

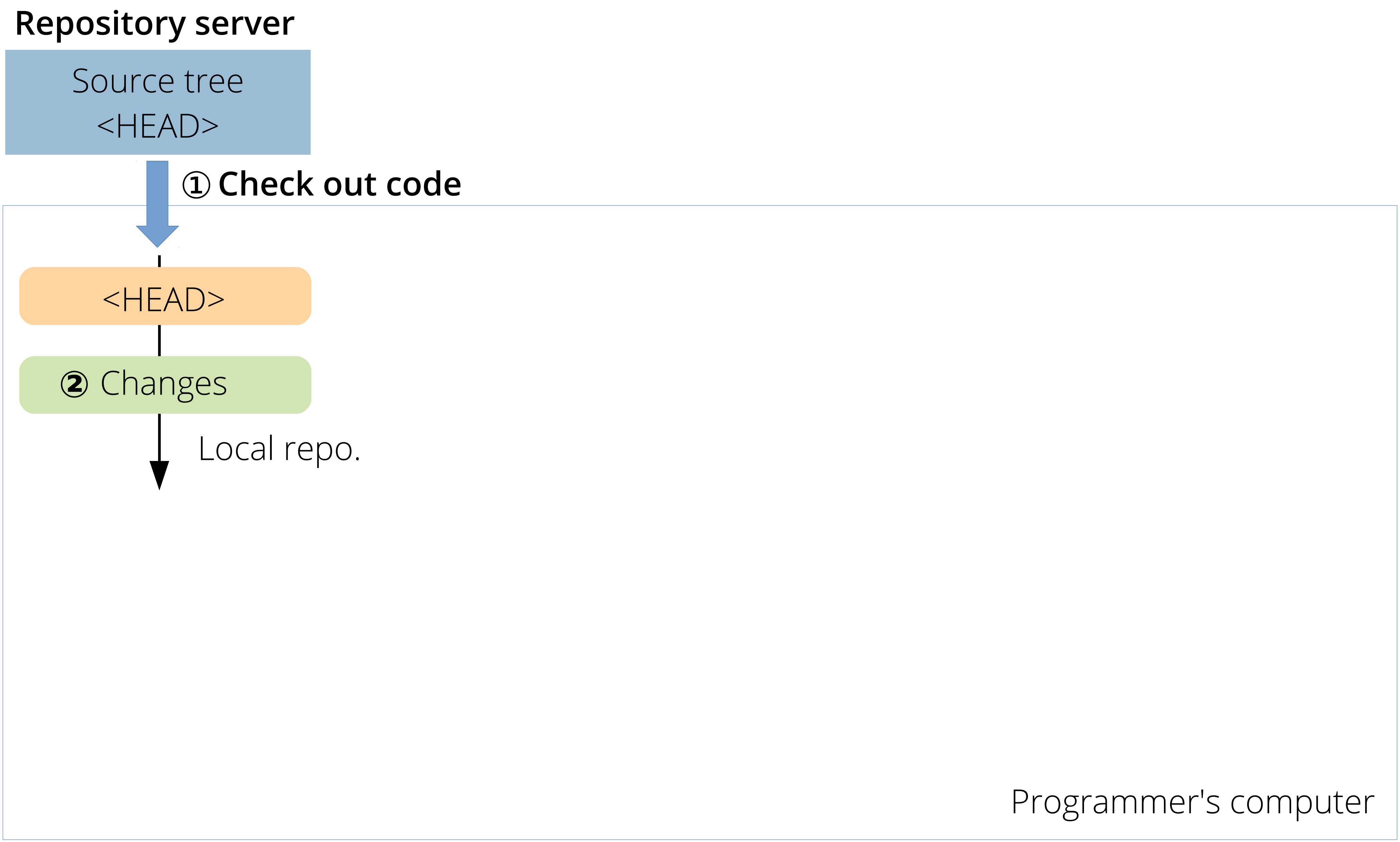
- **Idea 1: trade off soundness for performance**
  - Keep track of function-level dependency / changes
  - Fewer tests selected, may have false negatives

- **Idea 2: integrate test optimization into dev. cycle**
  - Maintain dependency information in code repository

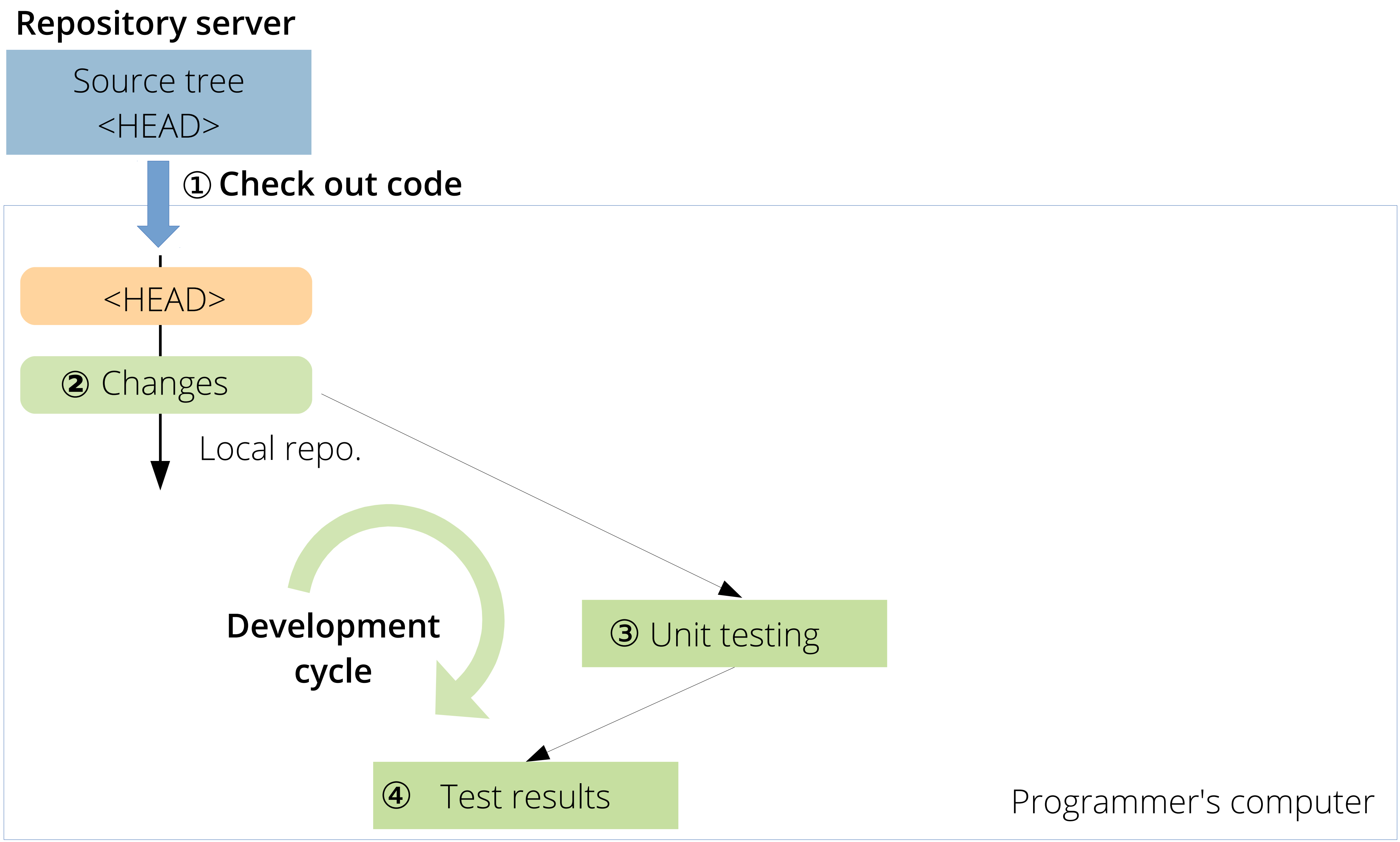
Current development cycle



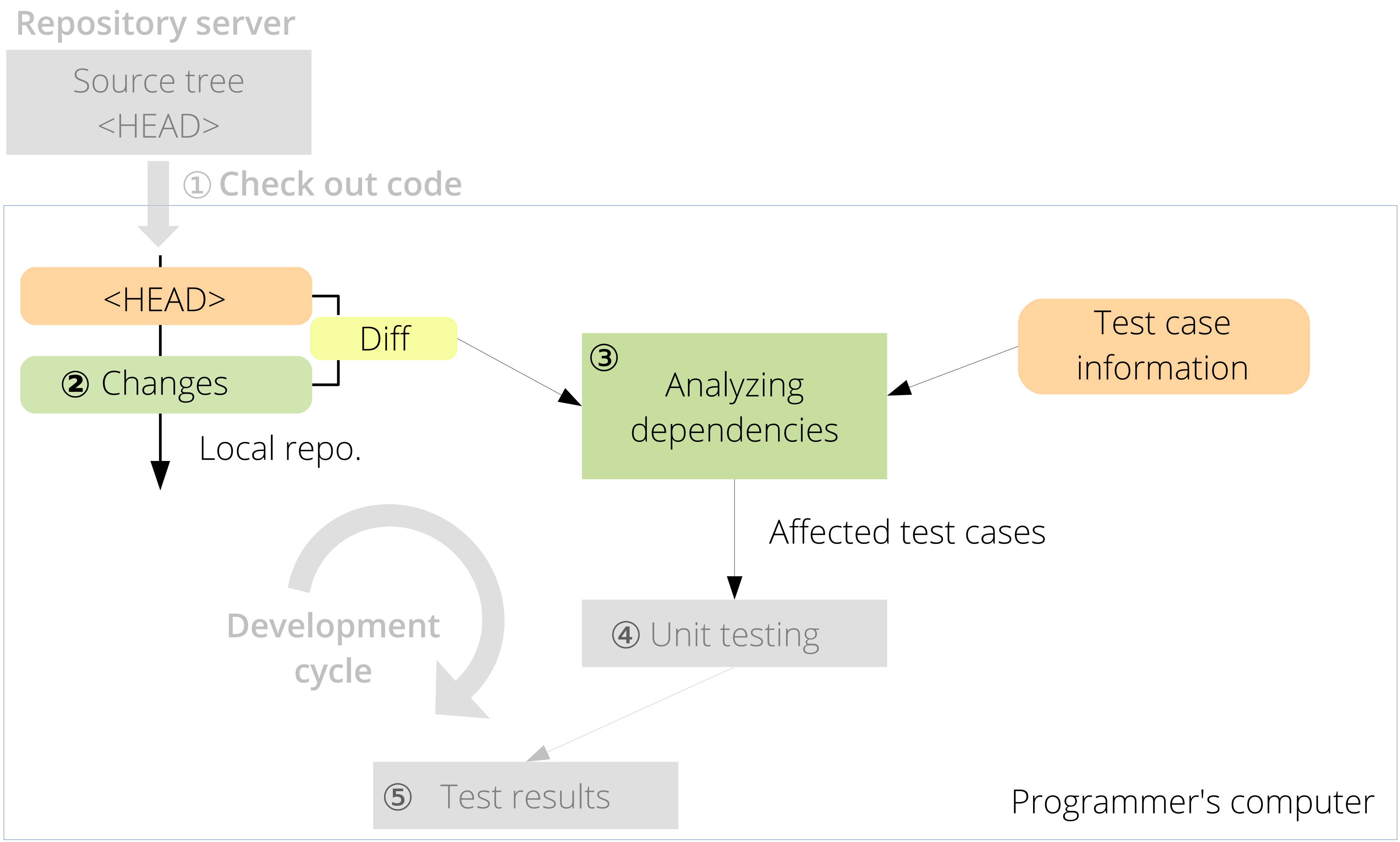
Current development cycle



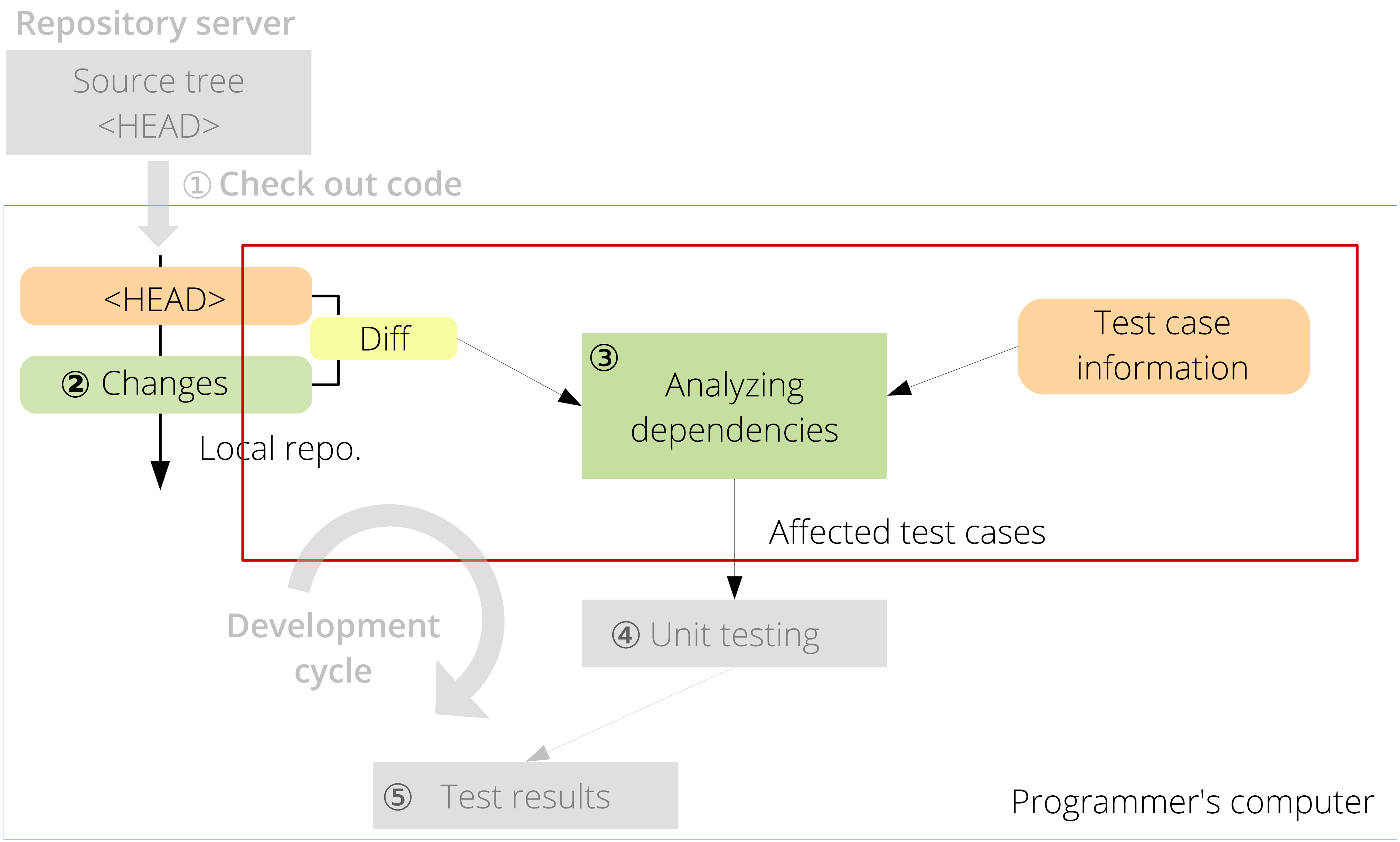
Current development cycle



New development cycle



New development cycle



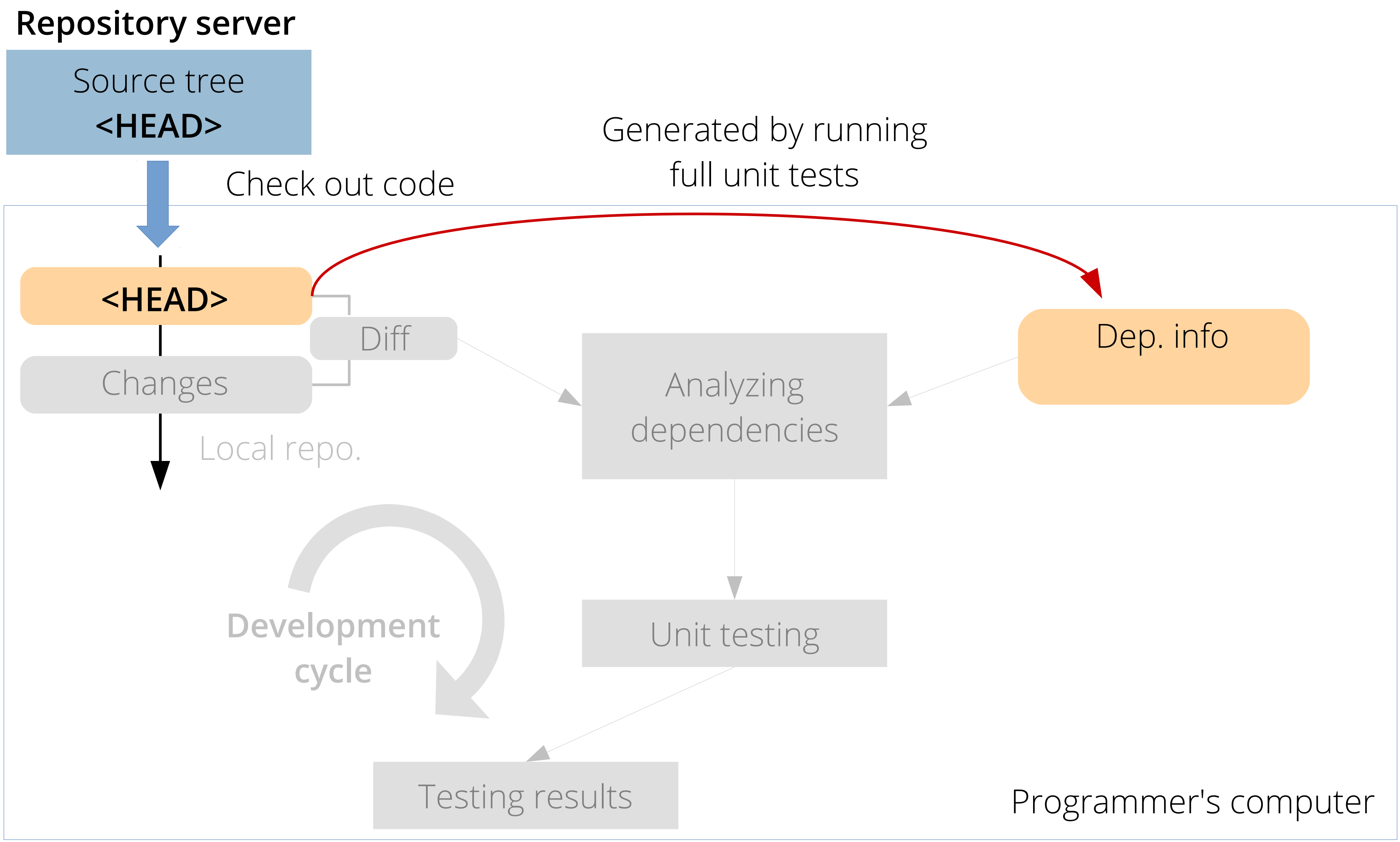
Identifying affected test cases by the code modification

- **Plan: track which tests execute which functions**
  - **Step 1**: generate function-level dependency info.
    - **Map**: invoked functions ↔ test case
    - Construct map by running all unit tests
  - **Step 2**: identify modified func., given code changes
  - **Step 3**: identify tests that ran the modified func.

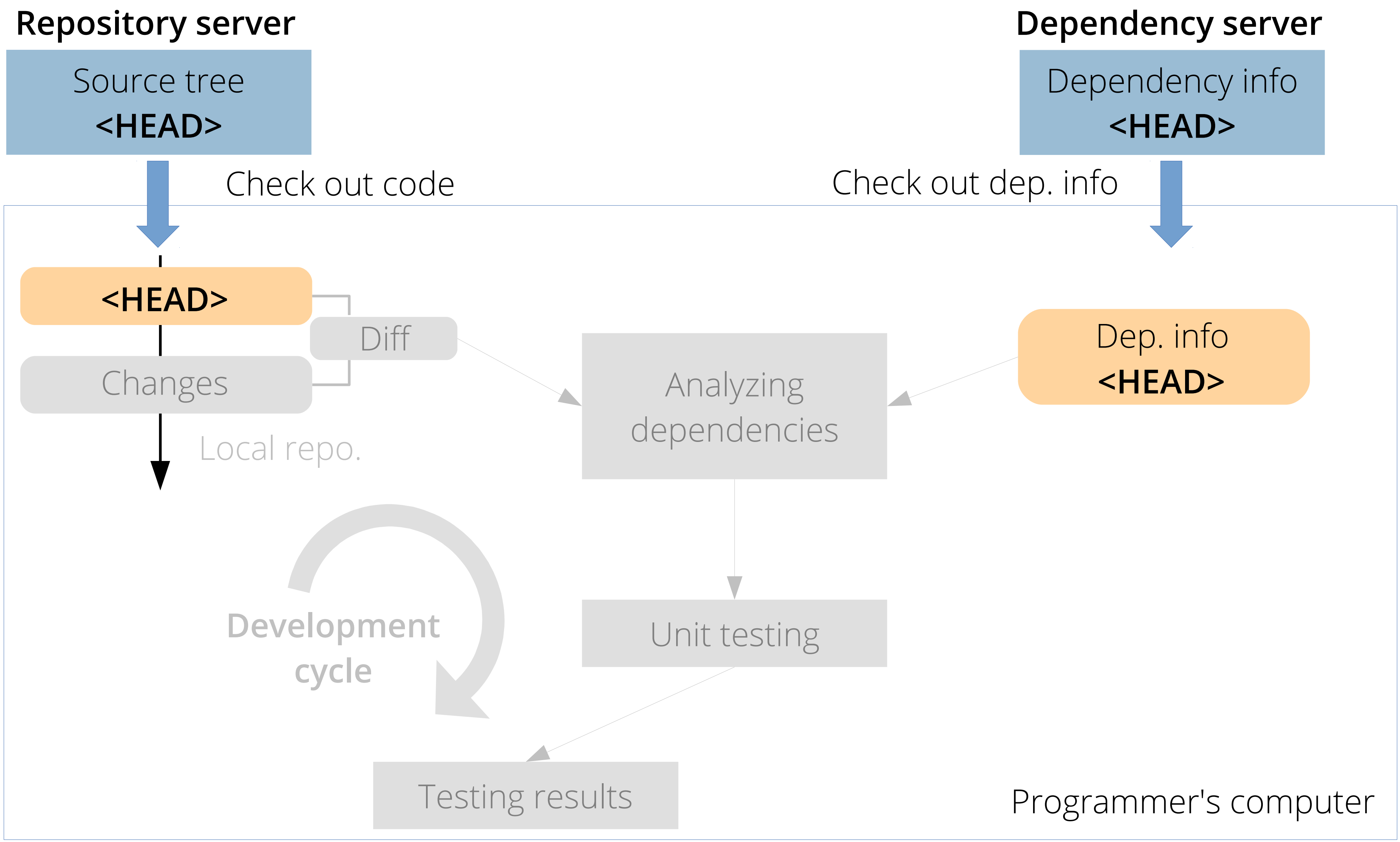
Identifying affected test cases by the code modification

- **Plan: track which tests execute which functions**
  - **Step 1**: generate function-level dependency info.
    - **Map**: invoked functions ↔ test case
    - Construct map by running all unit tests
  - **Step 2**: identify modified func., given code changes
  - **Step 3**: identify tests that ran the modified func.

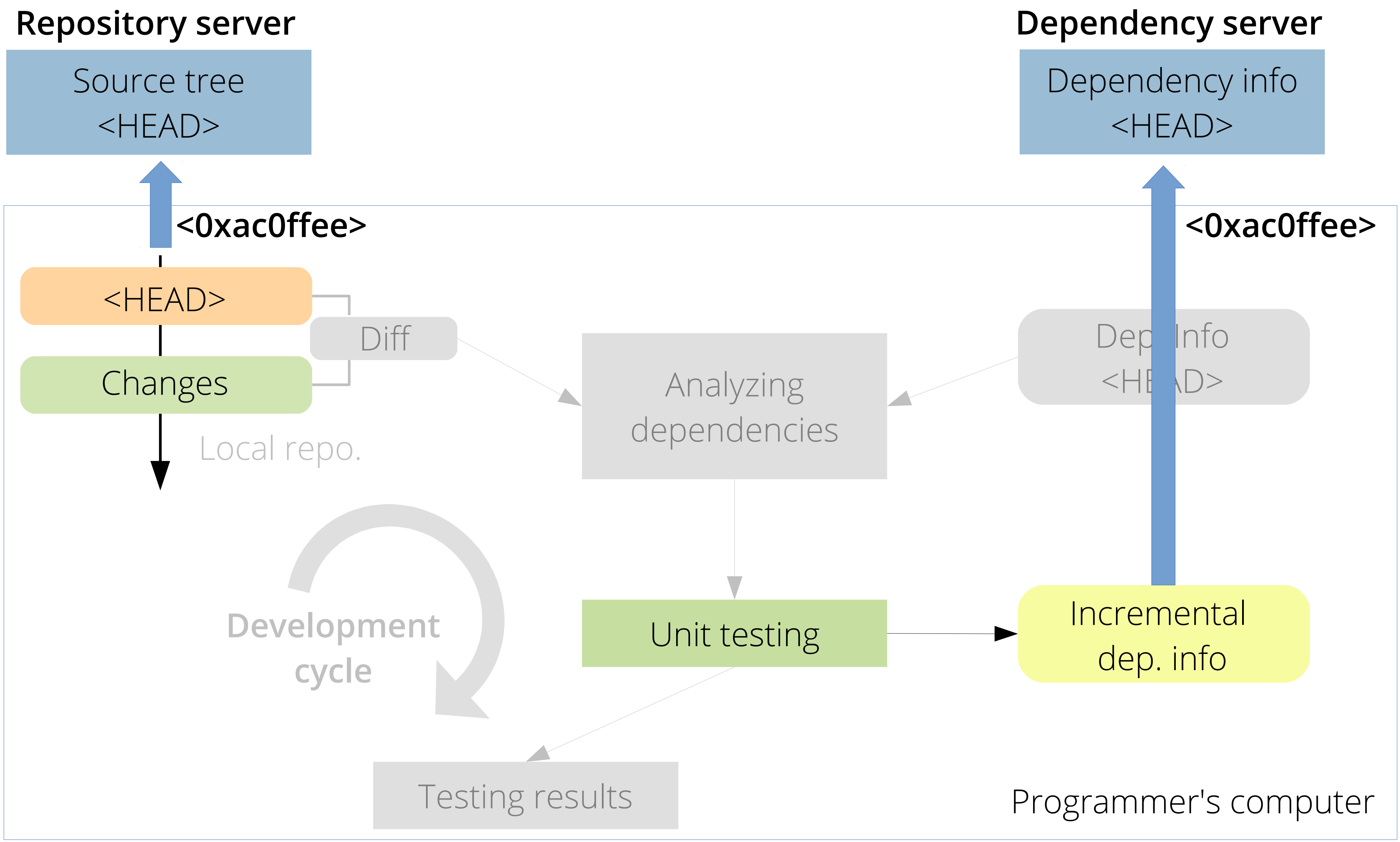
Bootstrapping dependency info.



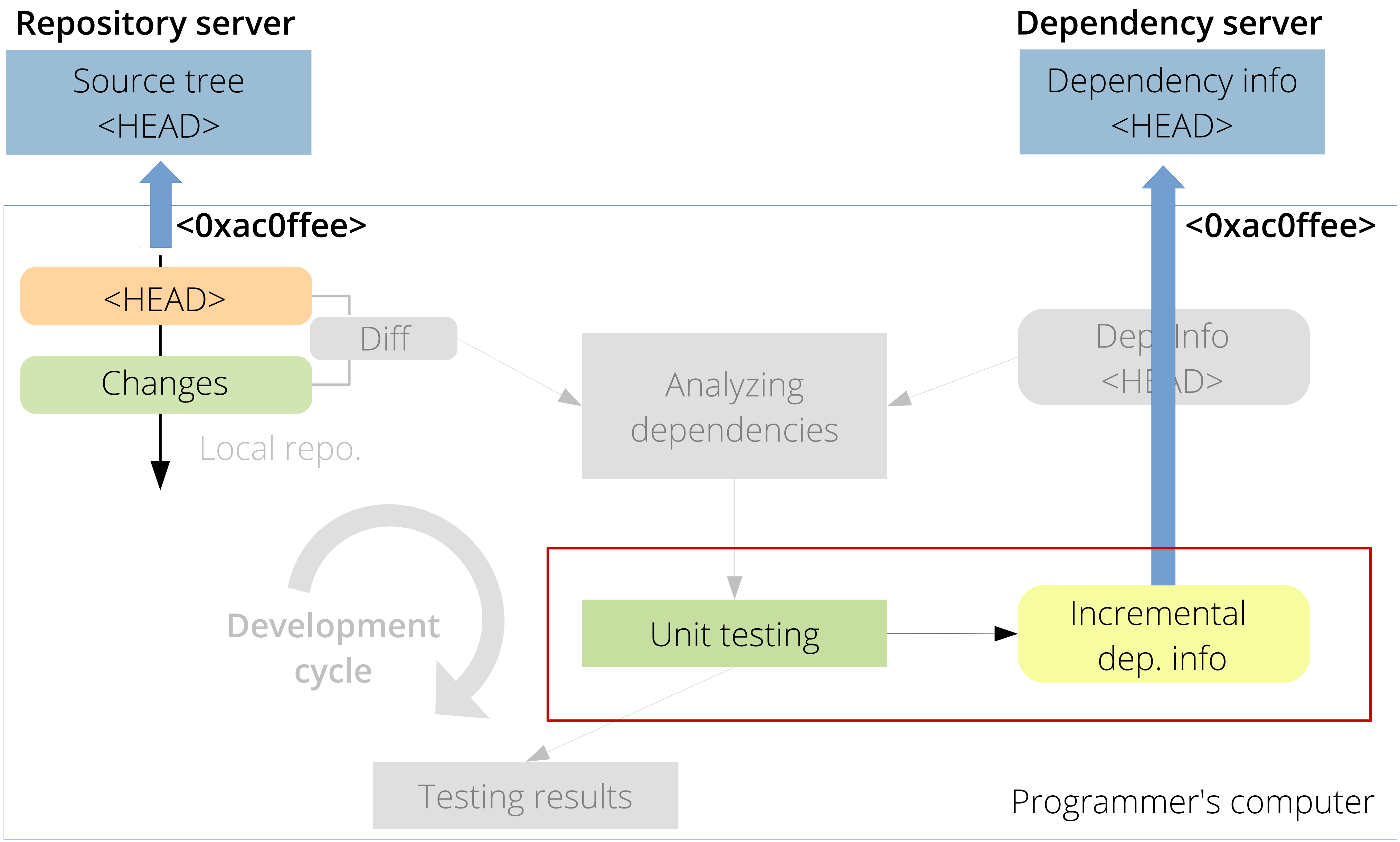
Bootstrapping dependency info.



Update dependency information



Update dependency information



Problem: false negatives

- Function-level tracking can **miss some dependencies** and cause **false negatives**
  - Failed to identify some test cases that are actually affected

- Identified **five types** of missing dependencies
  - Inter-class dependency
  - Non-determinism
  - Class variable
  - Global-scope
  - Lexical dependency

Problem: false negatives

- Function-level tracking can **miss some dependencies** and cause **false negatives**
  - Failed to identify some test cases that are actually affected

- Identified **five types** of missing dependencies
  - Inter-class dependency
  - Non-determinism
  - Class variable
  - Global-scope
  - Lexical dependency

Example: inter-class dep. in Python

```
class A:
  def foo():
    return 1
class B(A):
  pass



def testcase():
  assertEqual(
    B().foo(), 1)
```

Example: inter-class dep. in Python

```python
class A:
  def foo():
    return 1
class B(A):
  pass




def testcase():
  assertEqual(
    B().foo(), 1)
```

Dependency info:

```
testcase() →
    B.__init__()
    A.foo()
```

Example: inter-class dep. in Python

```
  class A:
    def foo():
      return 1
  class B(A):
-   pass
+   def foo():
+     return 2

  def testcase():
    assertEqual(
      B().foo(), 1)
```

Dependency info:

```
testcase() →
    B.__init__()
    A.foo()
```

Modified functions:

```
B.foo()
```

Example: missing dep. because of non-determinism in Python

```diff
 def foo():
-    return 1
+    return 2

 def testcase():
     if rand()%2:
         assertEqual(
             foo(), 1)
```

Dependency info:

```
testcase() →
    rand()
    foo()
```

or

```
testcase() →
    rand()
```

Modified functions:

```
foo()
```

Example: missing dep. because of non-determinism in Python

```
  def foo():
-   return 1
+   return 2

  def testcase():
    if rand()%2:
      assertEqual(
        foo(), 1)
```

Dependency info:

```
testcase() →
    rand()
    foo()
```

or

```
testcase() →
    rand()
```

Modified functions:

```
foo()
```

Example: class-var. dep. in Python

```diff
  class C:
-   a = 1
+   a = 2
  def foo():
    return C.a

  def testcase():
    assertEqual(
      foo(), 1)
```

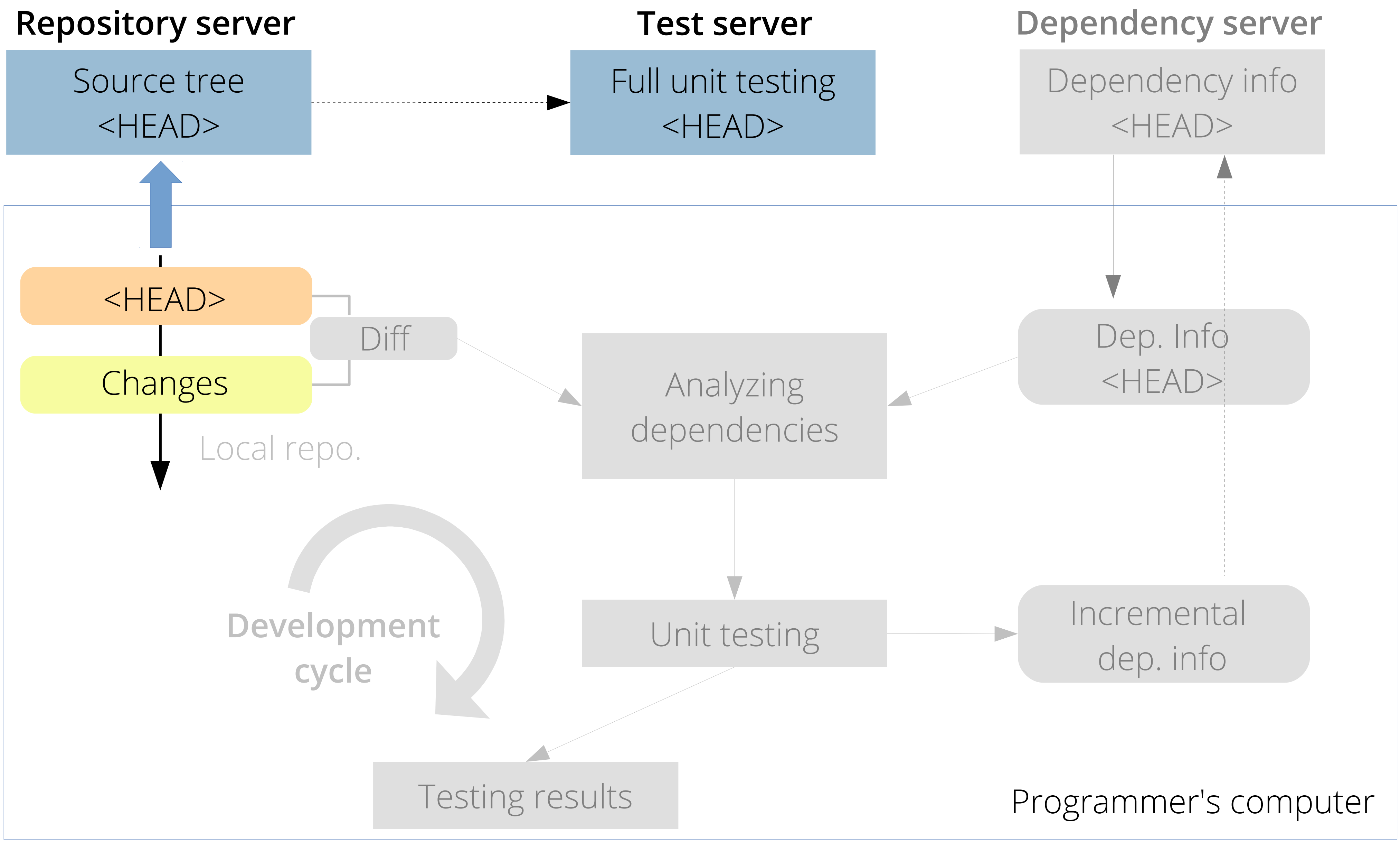
Dependency info:

`testcase() → foo()`

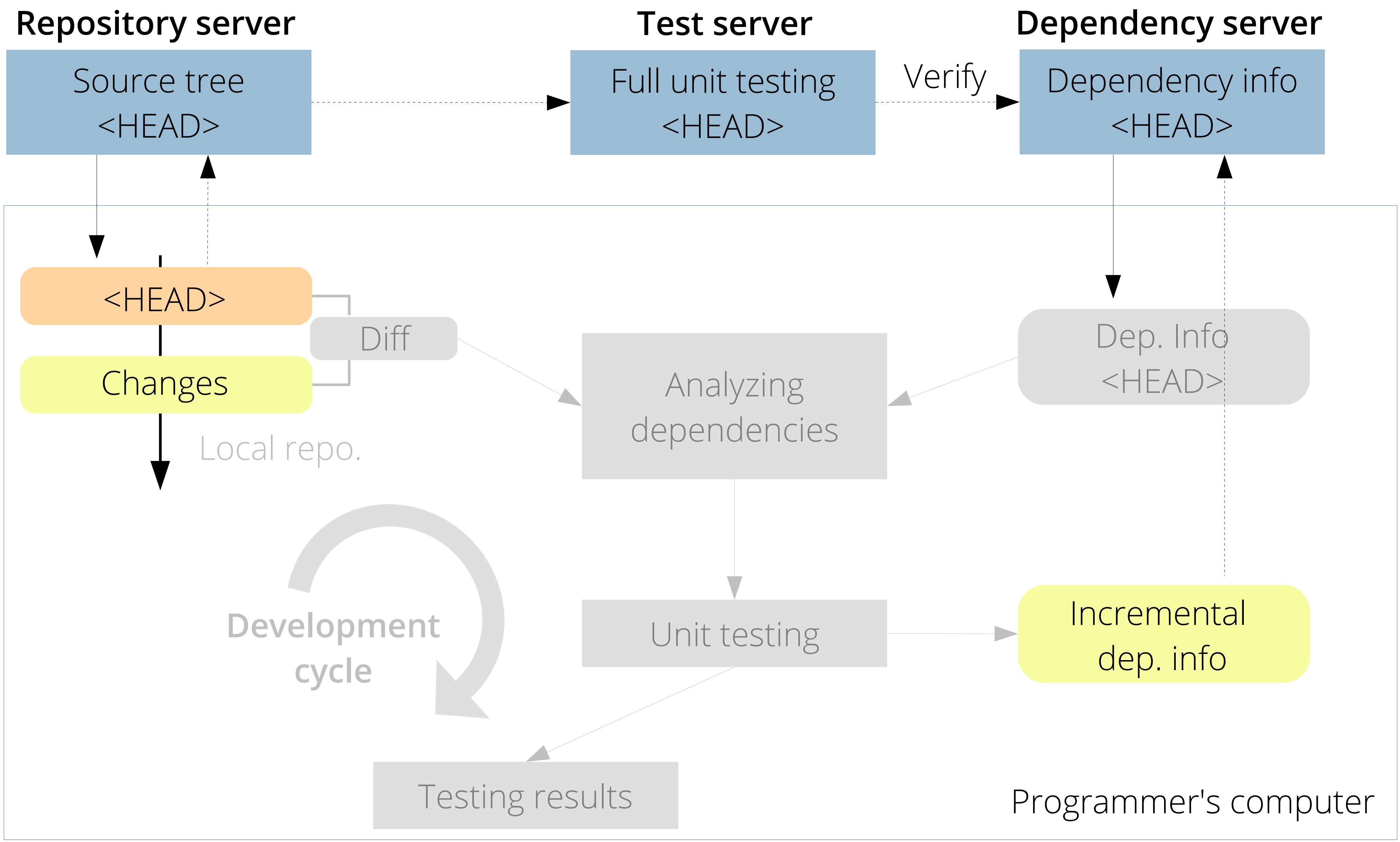
Modified functions:

`N/A`

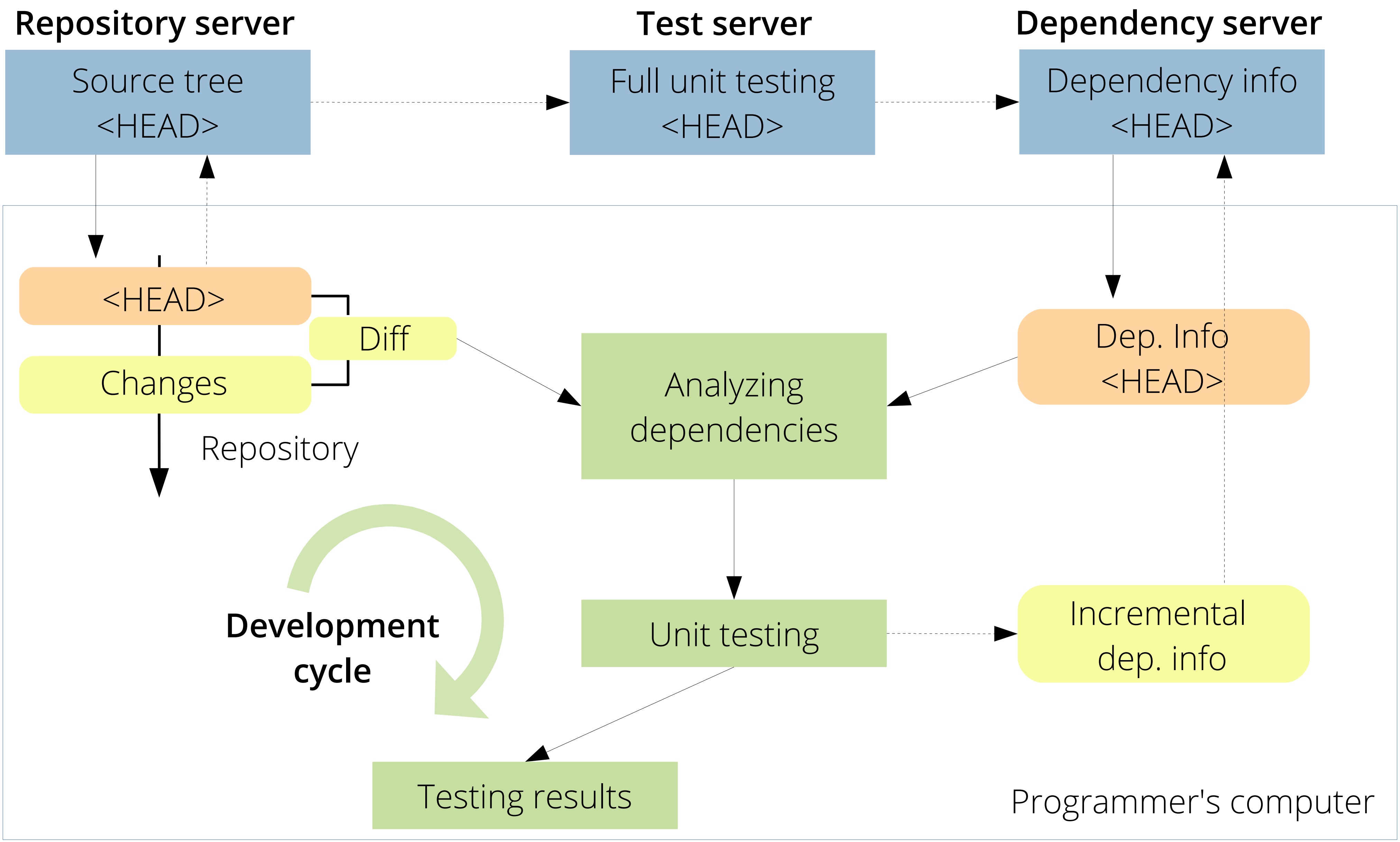
Solution: test server runs all tests async.



Test server also verifies dep. info



TAO: a prototype for PyUnit



Implementation

- TAO: a prototype for PyUnit
  - Extending standard **python-unittest** library
  - Patch analysis: using **ast/diff** python module
  - Dependency tracking: using **settrace()** interface
  - 800 Lines of code in Python

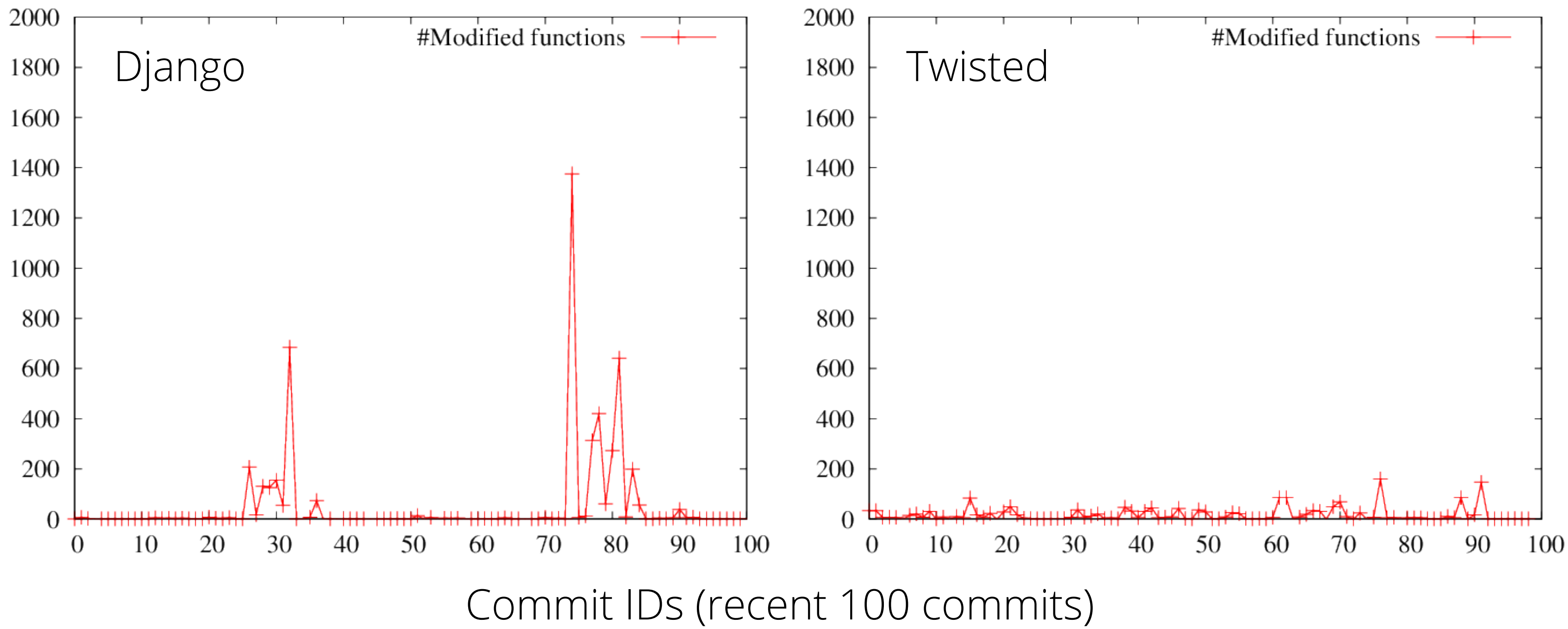
Evaluation

- How many functions are modified in each commit in large software programs?
- How much testing time can be saved as result?
- How many false negatives does TAO incur?
- What is the overall runtime overhead of TAO?

Experiment setup

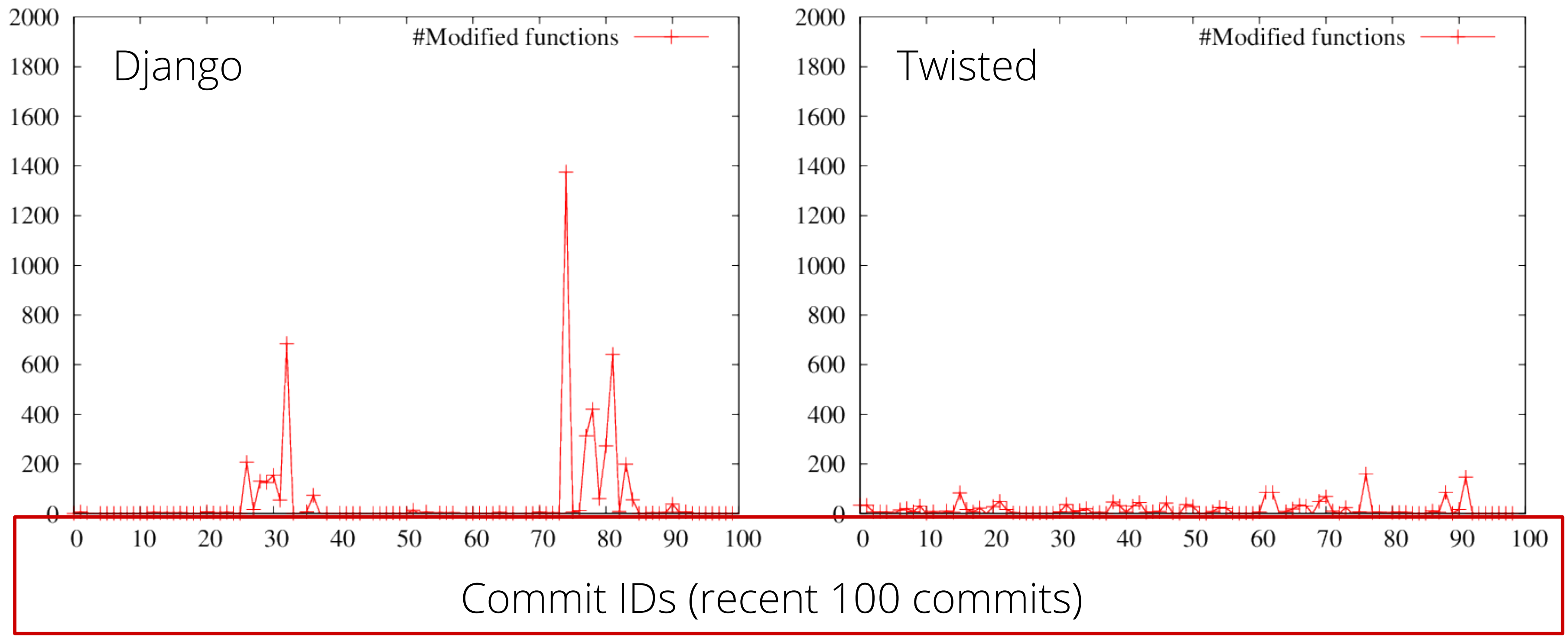
- Two popular projects: Django and Twisted
  - **Django**: a web application framework
  - **Twisted**: a network protocol engine
  - Use existing unit tests of both projects
  - Integrate TAO into both projects
  - Analyze the latest **100 commits** of each project

Small number of functions are modified in each commit



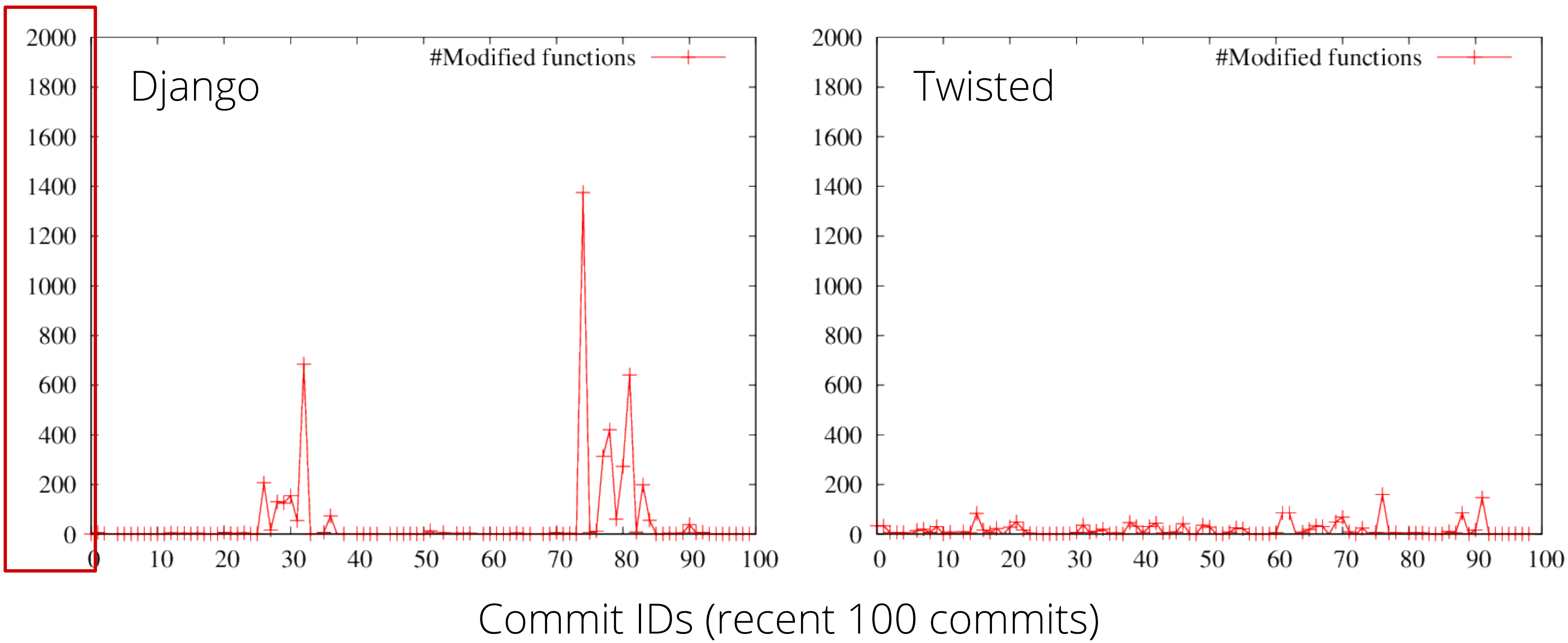
- Django: 50.8 / 13k functions (0.3%)
- Twisted: 18.2 / 23k functions (0.07%)

Small number of functions are modified in each commit



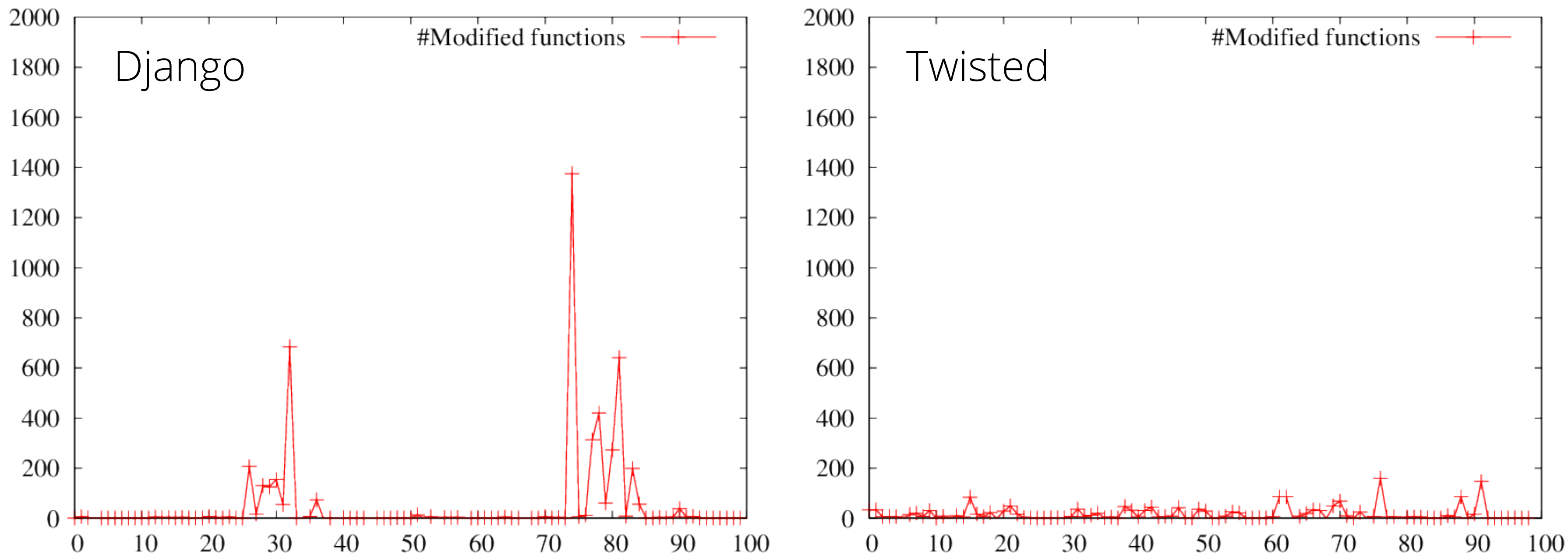
- Django: 50.8 / 13k functions (0.3%)
- Twisted: 18.2 / 23k functions (0.07%)

Small number of functions are modified in each commit



- Django: 50.8 / 13k functions (0.3%)
- Twisted: 18.2 / 23k functions (0.07%)

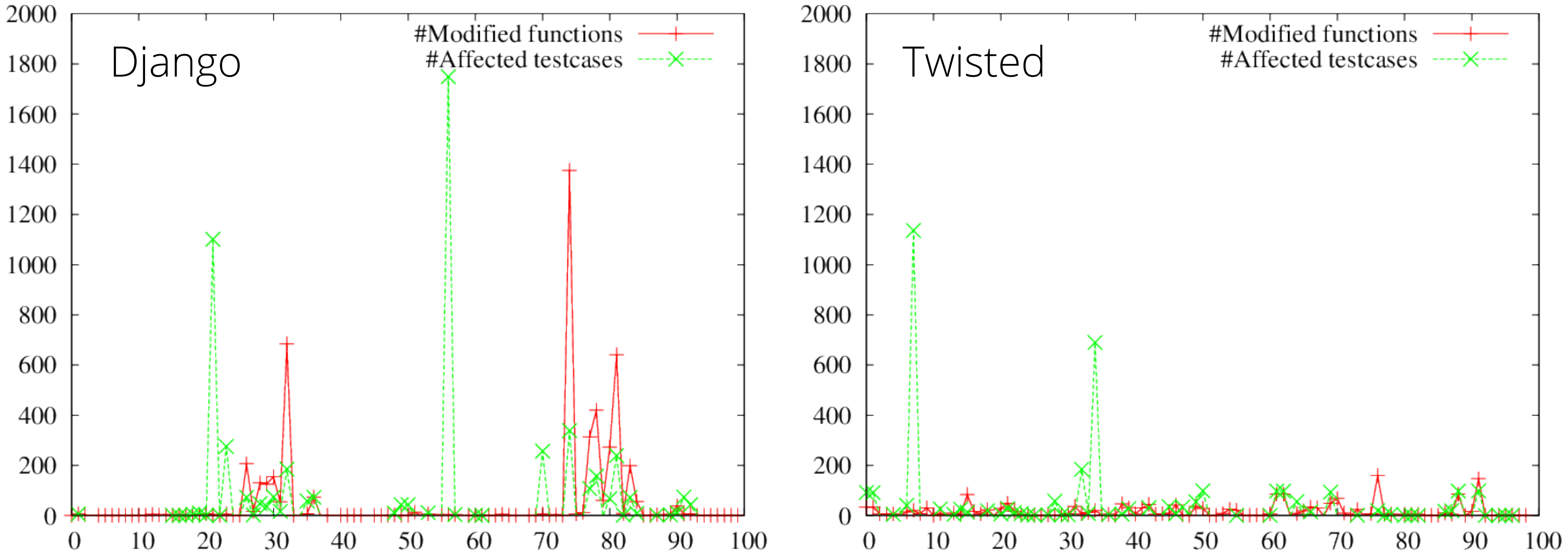
Small number of functions are modified in each commit



Commit IDs (recent 100 commits)

- Django: 50.8 / 13k functions (0.3%)
- Twisted: 18.2 / 23k functions (0.07%)

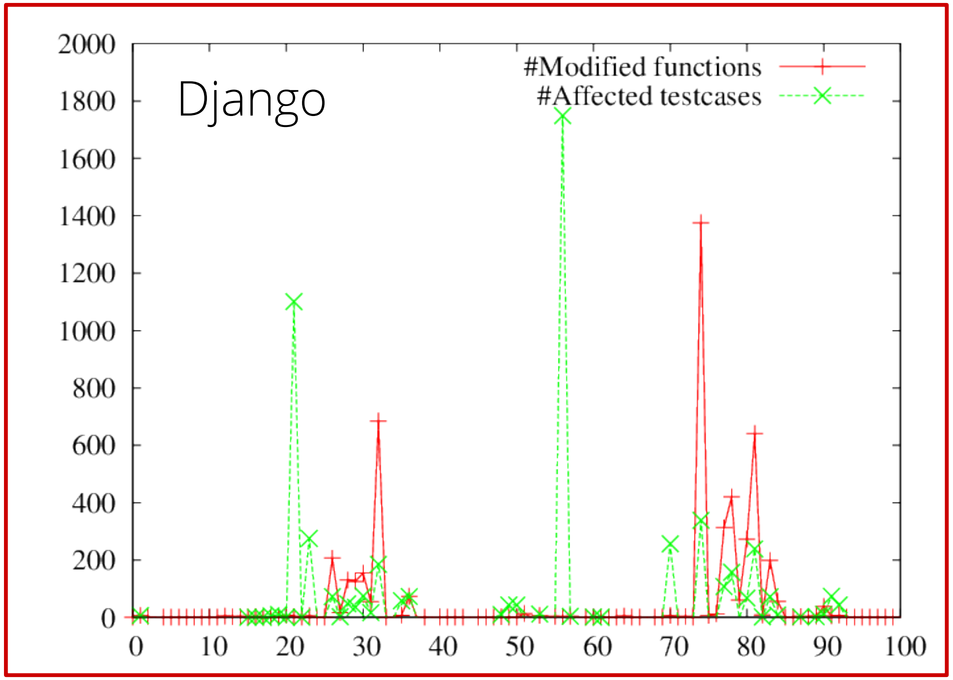
Small number of test cases need to be rerun

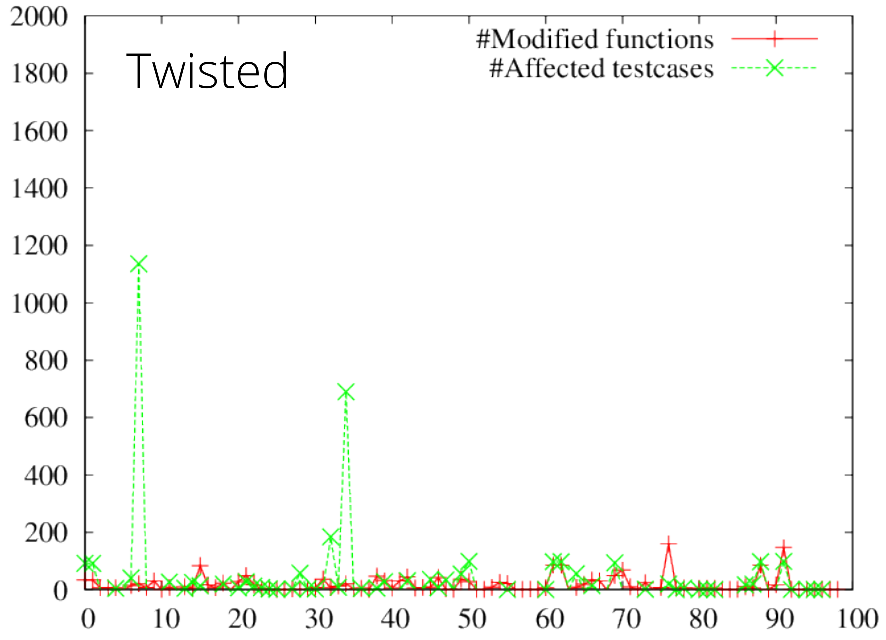


Commit IDs (recent 100 commits)

- Django: 50.4 / 5k test cases (1.0%)
- Twisted: 28.7 / 7k test cases (0.4%)

Small number of test cases need to be rerun

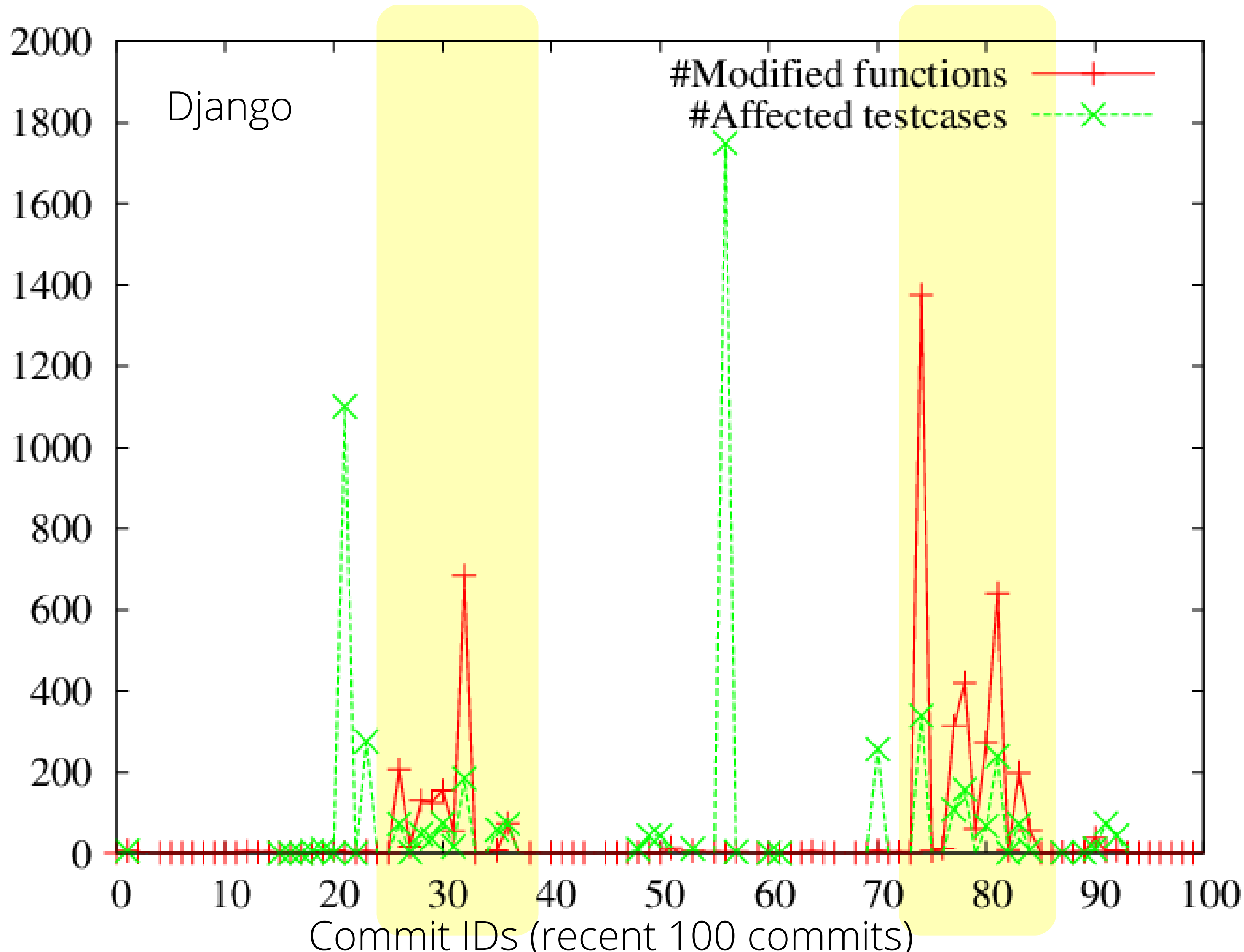




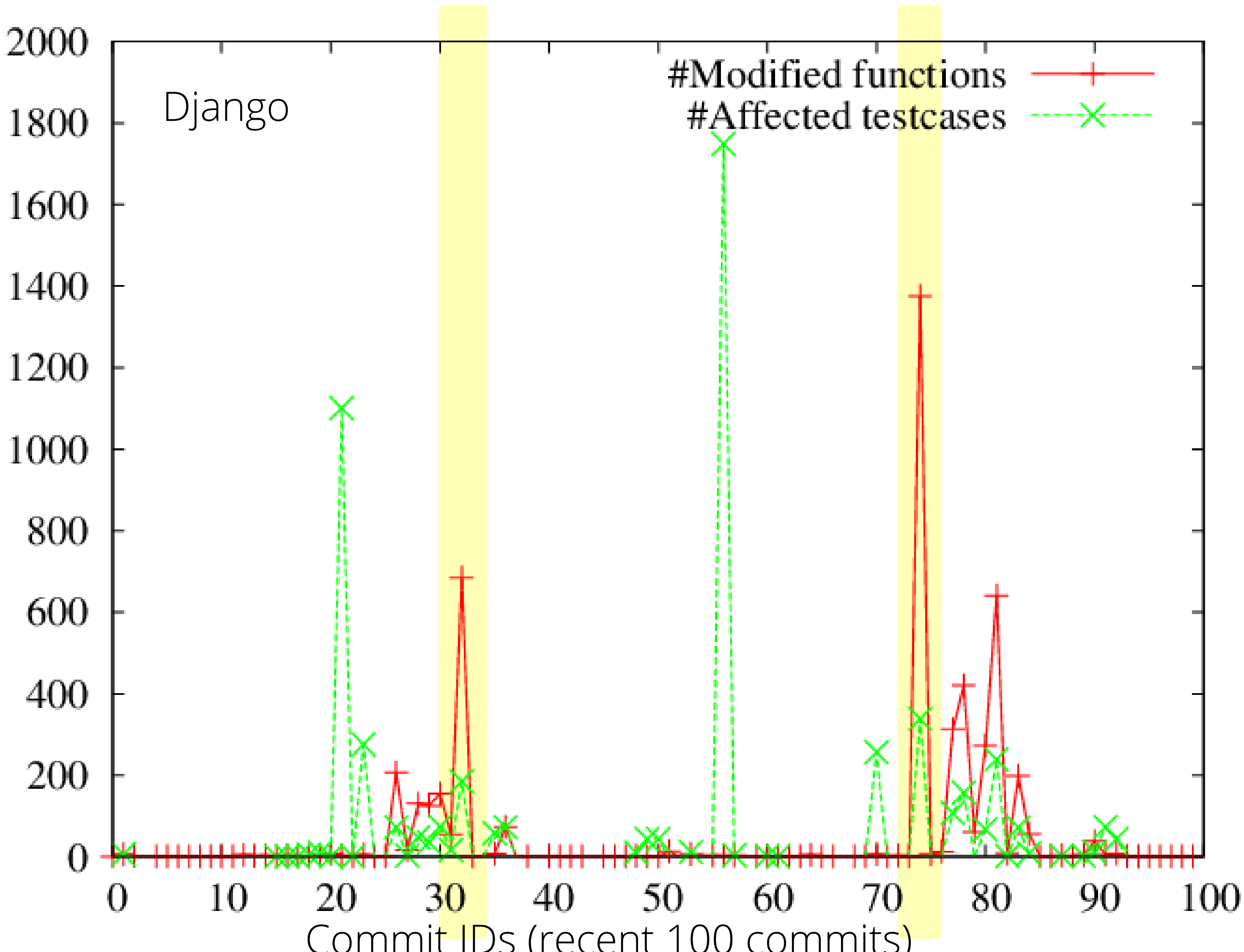
Commit IDs (recent 100 commits)

- Django: 50.4 / 5k test cases (1.0%)
- Twisted: 28.7 / 7k test cases (0.4%)

Trend 1: #affected test cases is correlated with #modified functions

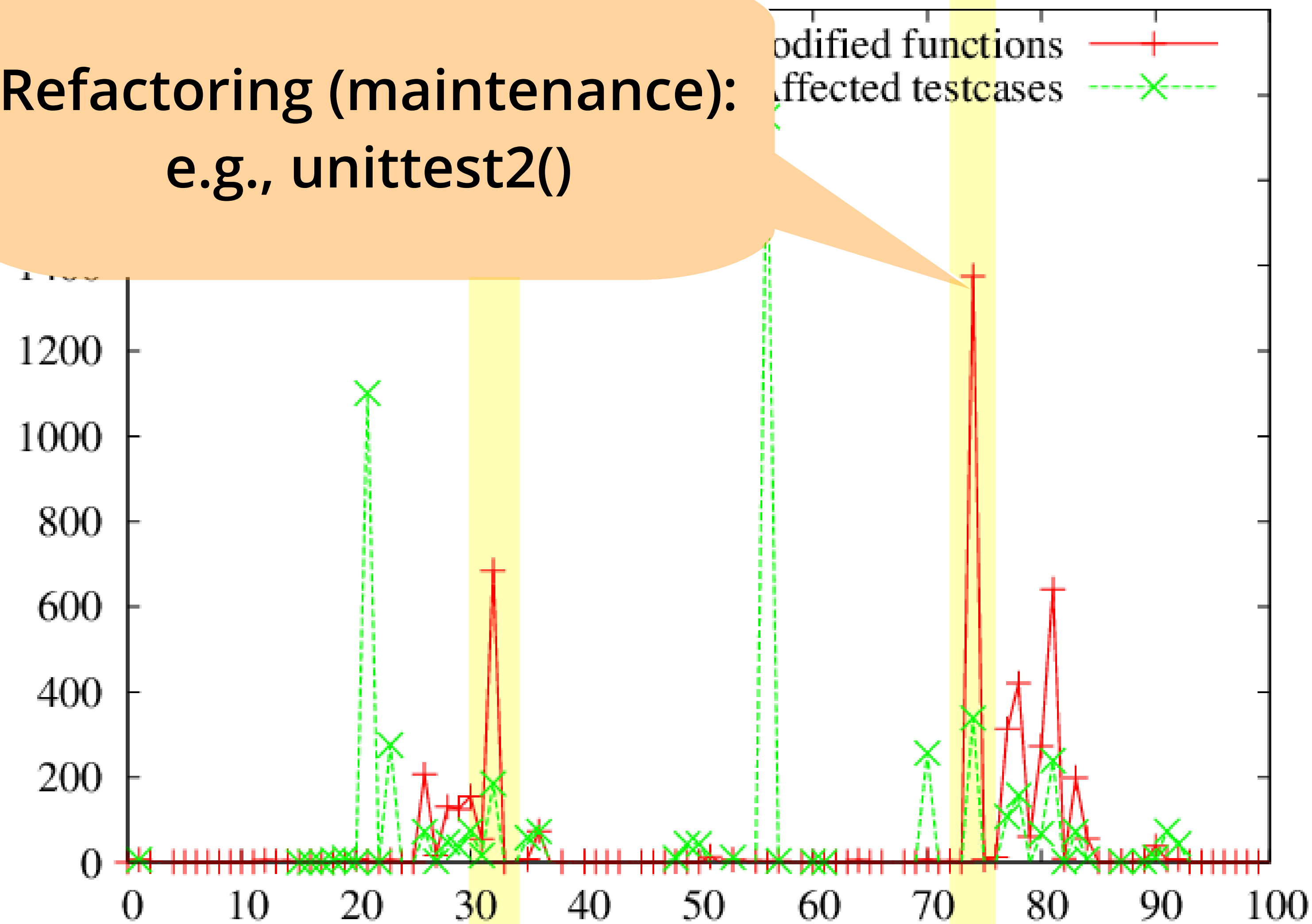


Trend 2: many modified functions, few affected test cases



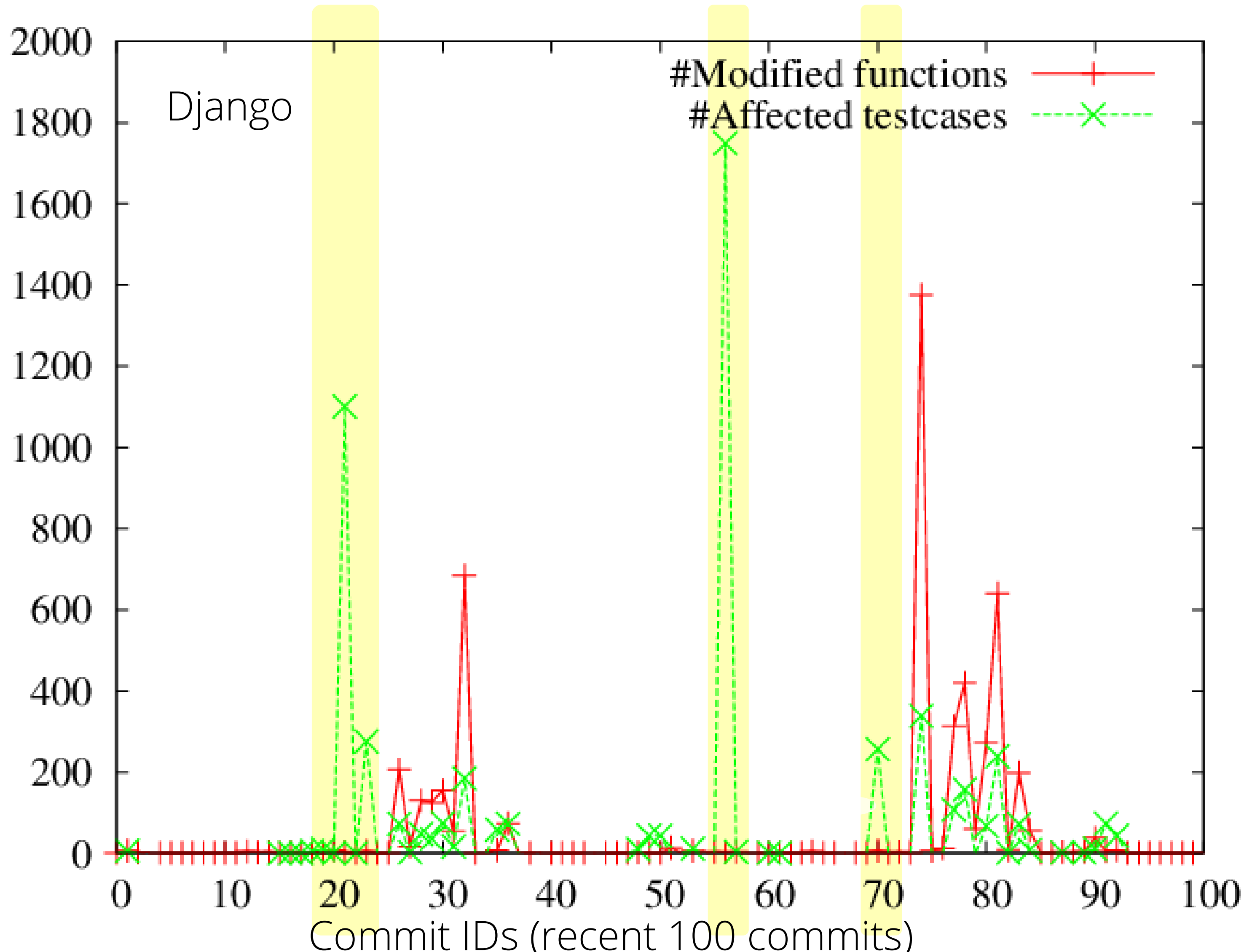
Trend 2: many modified functions, few affected test cases

Refactoring (maintenance): e.g., unittest2()

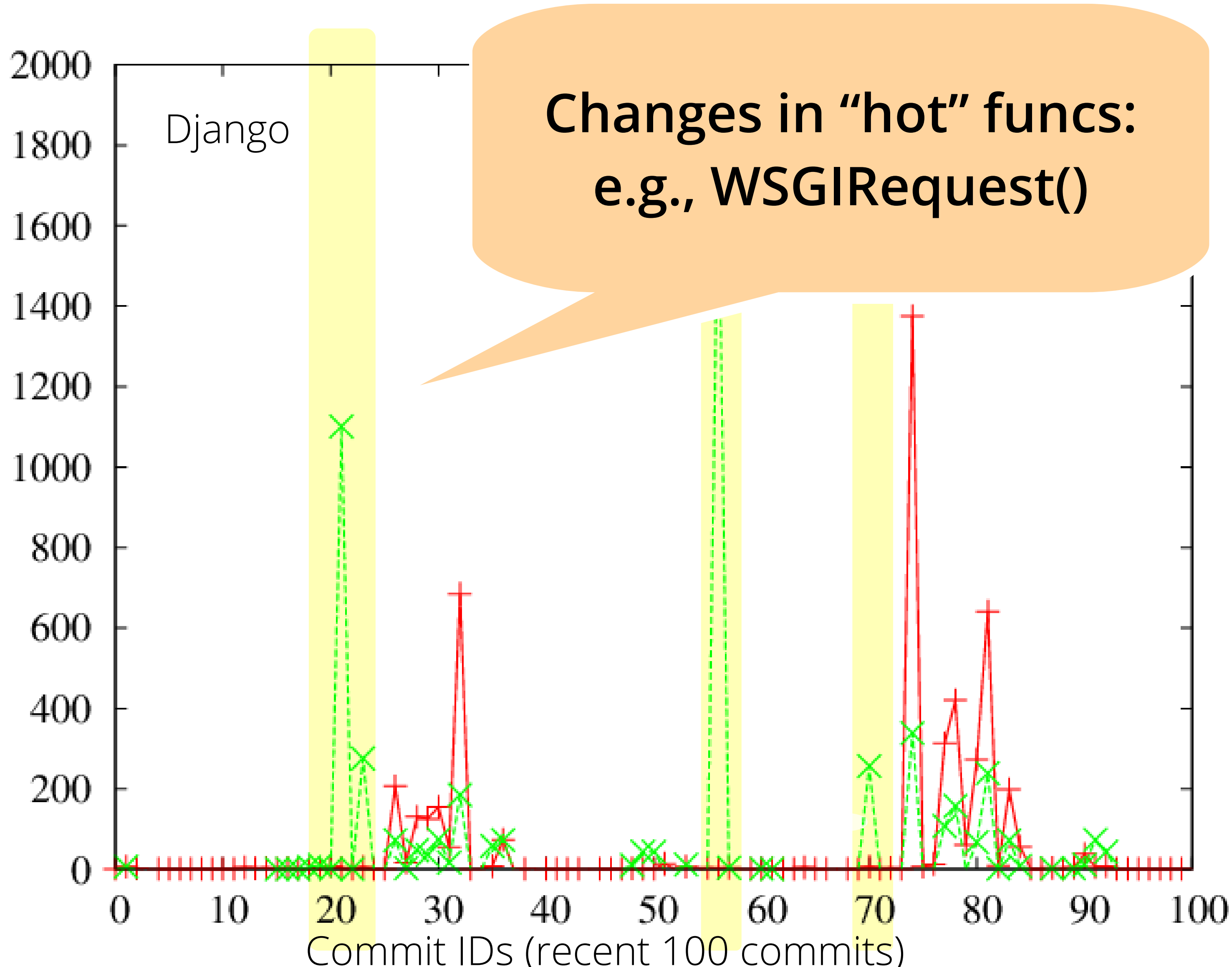


Commit IDs (recent 100 commits)

Trend 3: few modified functions, many affected test cases



Trend 3: few modified functions, many affected test cases



TAO can improve the overall execution time for unit testing

Project	#Test cases		Execution time (s)	
	All	TAO	All	TAO
Django	5,166	50.8	520.3s	1.7s
Twisted	7,150	28.7	72.1s	2.2s

- Django: 520.3s → 1.7s (5k → 50.8 test cases)
- Twisted: 72.1s → 2.2s (7k → 29.7 test cases)

TAO has few false negatives (FN)

Project	FN/I (inter-class)	FN/N (non-det.)	FN/G (global scope)	FN/C (class var.)	FN/L (lexical dep.)
Django	0/0	0/0	2/8	1/3	1/23
Twisted	1/2	0/0	1/20	1/17	0/11

- We **manually identified** types of missing dependencies and false negatives on each commit
- Django: 3 false negatives (one commit is counted in both G/L)
- Twisted: 3 false negatives

TAO has few false negatives (FN)

Among class variable deps we identified,
how many false negatives end up getting at?

Project	FN/I (inter-class)	FN/N (non-det.)	FN/G (global scope)	FN/C (class var.)	FN/L (lexical dep.)
Django	0/0	0/0	2/8	1/3	1/23
Twisted	1/2	0/0	1/20	1/17	0/11

- We **manually identified** types of missing dependencies and false negatives on each commit
- Django: 3 false negatives (one commit is counted in both G/L)
- Twisted: 3 false negatives

Example: not all missing deps cause false negatives

Missing dep.: class var.

```
  class DecimalField(IntegerField):
      default_error_messages = {
          ...
-         'max_digits': _(msg)
+         'max_digits': ungettext_lazy(msg)
      ...

      def __init__(...):
          ...
-             raise ValidationError(oldmsg)
+             raise ValidationError(newmsg)
```

Function-level dependency

Dependency tracking imposes performance overheads

Project	Runtime		Storage	
	no TAO	TAO	Full	Incremental
Django	520.3s	1,129.1s	9.9MB	270KB
Twisted	72.1s	115.6s	1.3MB	280KB

- Django: 10 min (117%) to generate dep. info (9.9MB)
- Twisted: <1 min (60%) to generate dep. info (1.3MB)
- Performance can be improved if we implement function-level tracing natively, instead of using settrace() library.

Incremental dependency information is small

Project	Runtime		Storage	
	no TAO	**TAO**	**Full**	**Incremental**
Django	520.3s	1,129.1s	9.9MB	270KB
Twisted	72.1s	115.6s	1.3MB	280KB

- Django: 270KB incremental dep. info (per commit)
- Twisted: 280KB incremental dep. info (per commit)

Related work

- **Regression test selection:**
  - **RTS [Biswas '11]:** survey of available RTS techniques
    - → Simple function-level dependency is effective in practice
    - → TAO can be integrated into the programmer's workflow

- **Dependency tracking:**
  - **Poirot [Kim '12]**: intrusion recovery
  - **TaintDroid [Enck '12]**: privacy monitoring
    - → Dependency tracking can optimize unit test execution

Summary

TAO: a system that optimizes unit test execution using dependency analysis

- Tracks function-level dependency of each unit test
- Analyzes code changes to find the affected test cases
- Runs only affected test cases (but few false negative)
- Integrated into programmer's development cycle