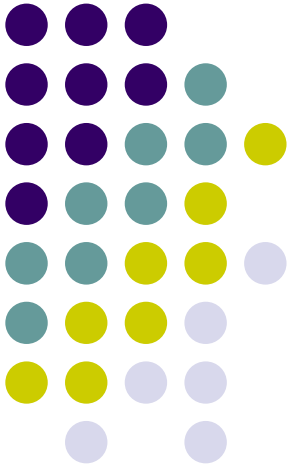
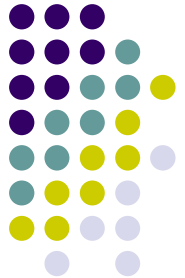
Automatic Test Factoring for Java



2006/08/23 高子騰

References



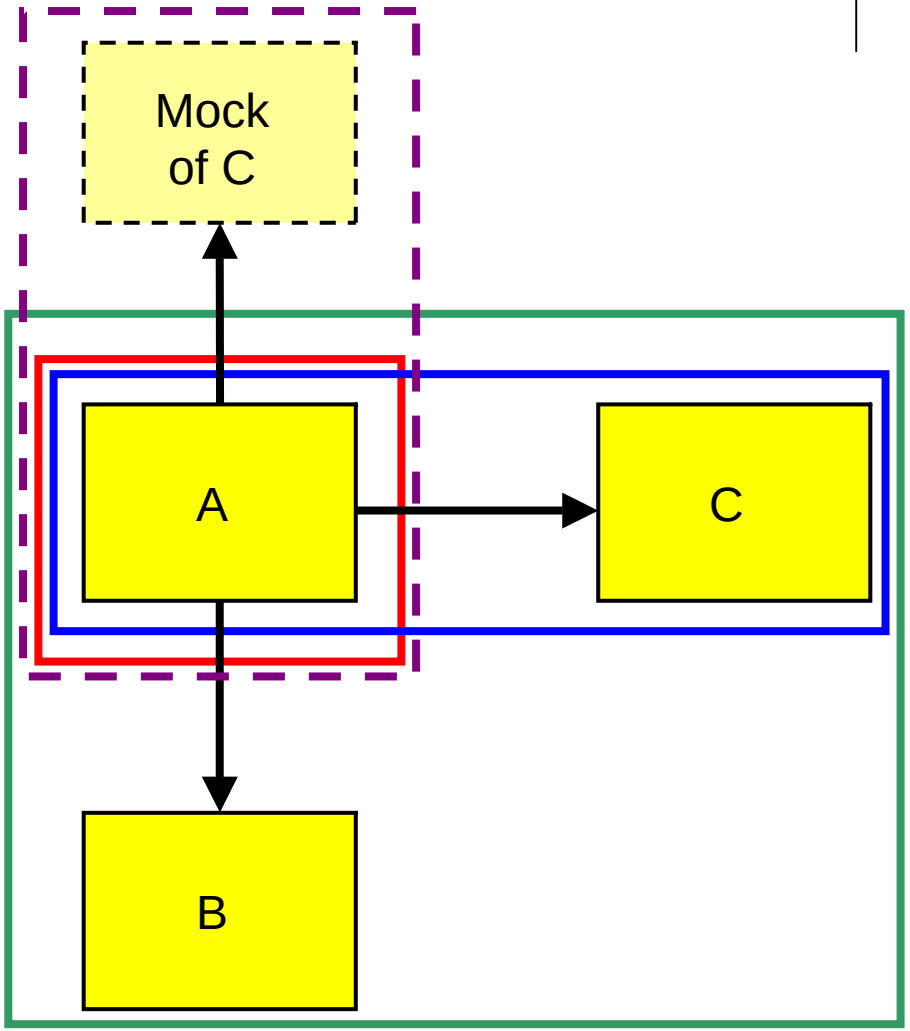
- D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, Automatic Test Factoring for Java, In ASE 2005, pages 114-123, November 2005.
- D. Saff and M. D. Ernst. Mock Object Creation for Test Factoring. In PASTE 2004, pages 49-51, June 2004.

Outline

- Introduction
- Test Factoring via Mock Object
- Instrumenting Java Classes
- Case Study
- Conclusion

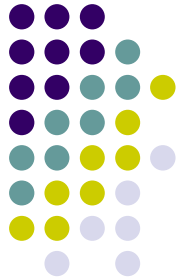
The evolution of software testing methodology

- Focused tests
- System tests
- Test factoring
  - Mock object

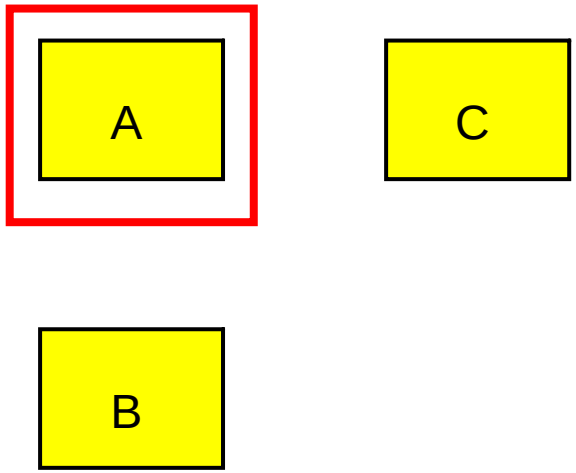


A software system

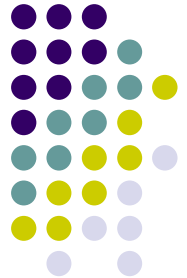
Focused tests



- A focused test exercises only part of a system.
- For example, a unit test exercises one component without relying on any other component.



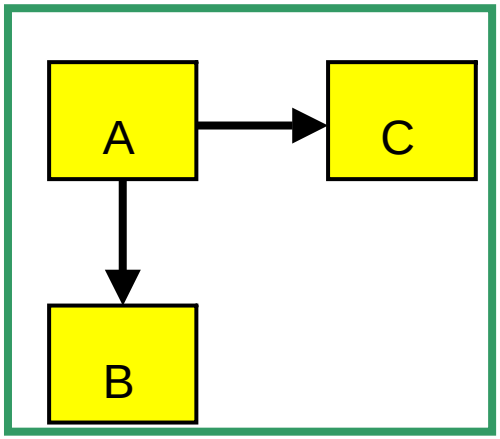
Focused tests have many benefits



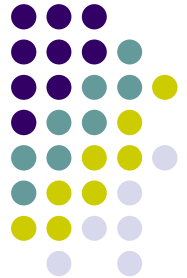
- Execute quickly, so they can provide fast feedback, and they can be run frequently.
- Isolate errors to a small amount of code, easing debugging on a smaller set of places.

Why is system test used?

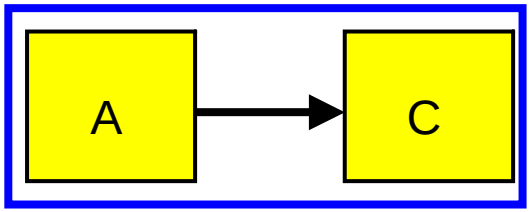
- But, focused tests are often not available.
- Instead, a software system may have system tests.
- System tests are long-running, end-to-end tests that exercise much of the functionality of the entire system.



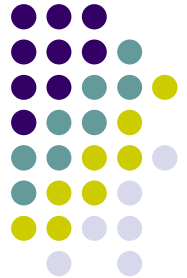
Test factoring provides benefits of focused tests



- Provide the benefits of focused tests to a developer who has only written system tests.
- Test factoring, for automatically creating fast, focused unit tests from slow system-wide test and each unit test exercises only a subset of the functionality exercised by system tests.

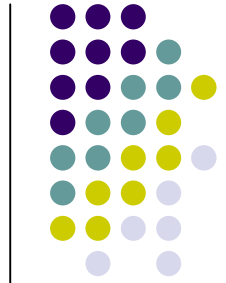


Inputs and The Output of Test Factoring

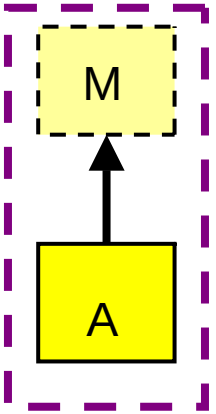


- Test factoring take three inputs:
  - A program
  - A system test
  - A partition of the program into the “code under test” and the “environment”
- The output of test factoring is a set of factored tests for the code under test.

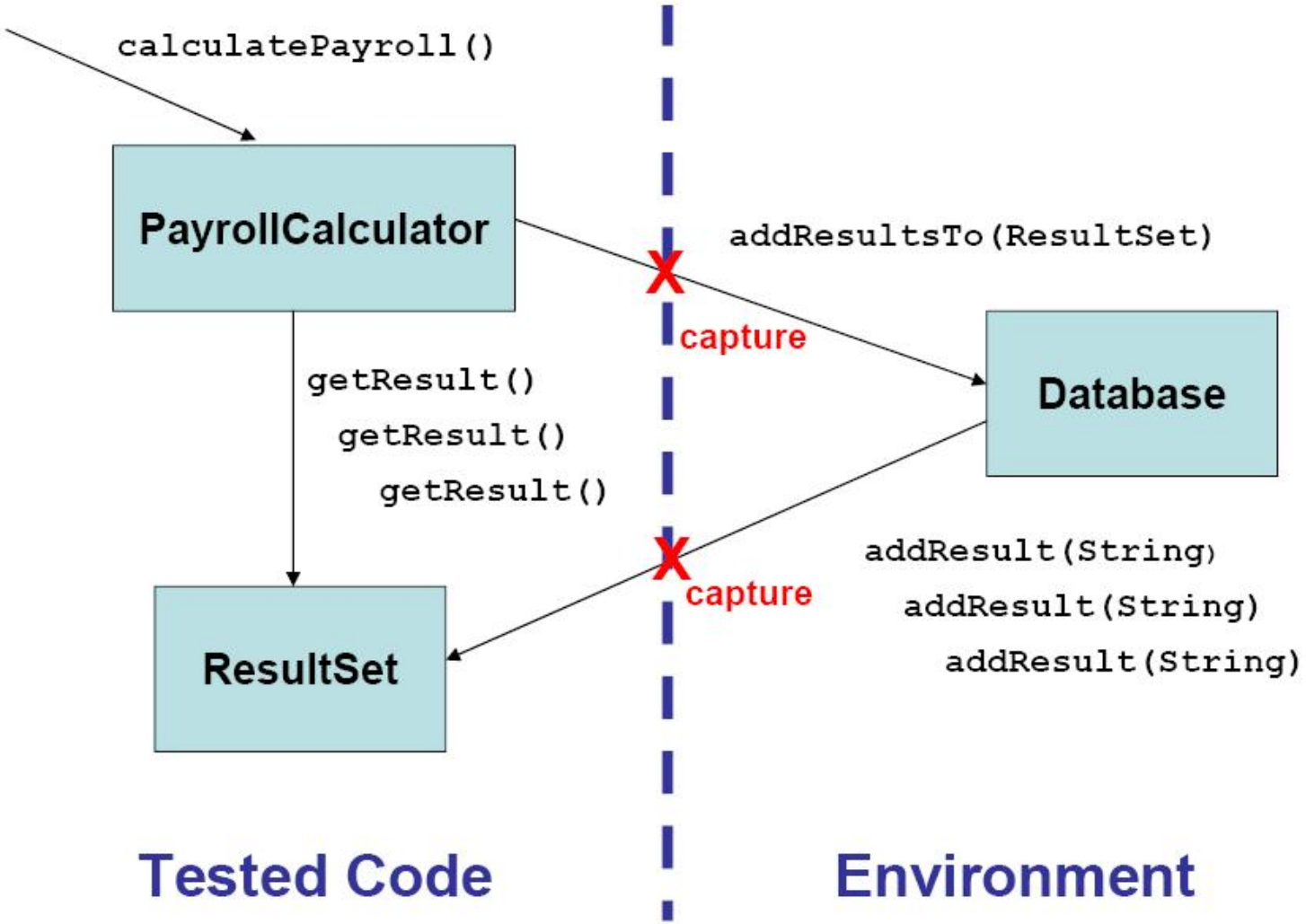
Test factoring by mock objects



- Test factoring replaces the environment part of the program by mock objects.
- Mock object, which is a stub that requires that it is used in particular ways, has a subset of the functionality of real object.
- A test that utilizes the mock object rather than the expensive resource can be cheaper (for example, faster).

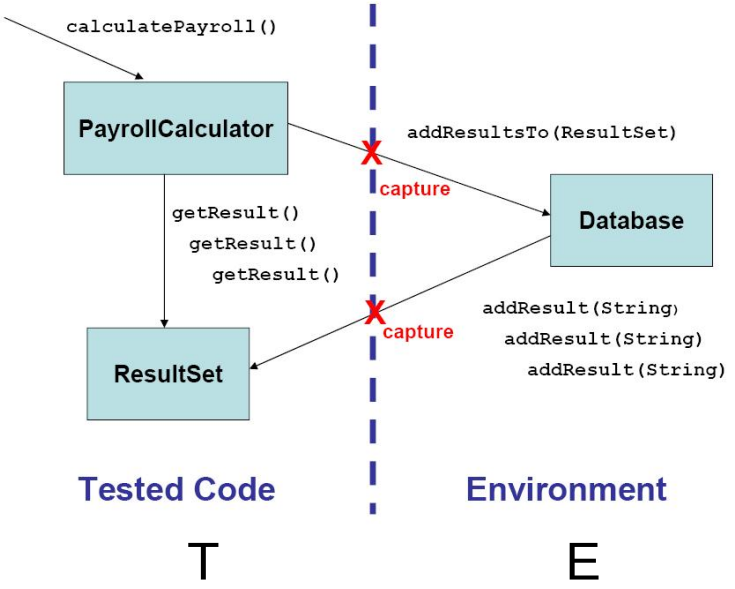


An example: The payroll system

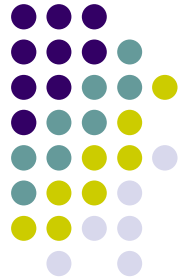


Usage of symbol of this paper

- Suppose that software system is composed of two parts, T and E.
  - T is the code under test
  - E is the environment
  - T|E is the software system
  - E_m is the mock object
  - T' is a part that T is changed to



Create a mock object with a transcript

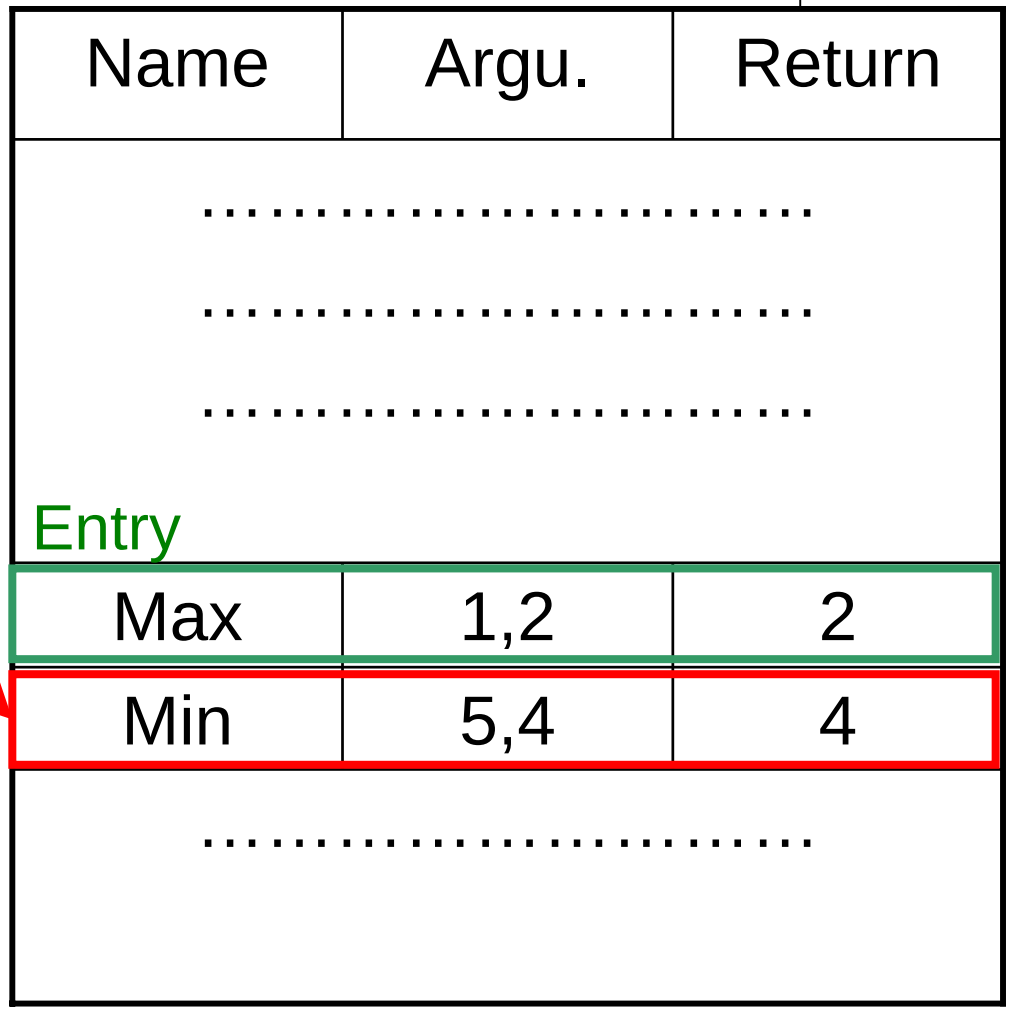


- One common implementation of a mock object incorporates a lookup table, which they call a “transcript”.
- Each entry in the transcript has:
  - Method name
  - Arguments
  - Return value

Name	Argu.	Return
……		
……		
……		
Max	1,2	2
Min	5,4	4
……		

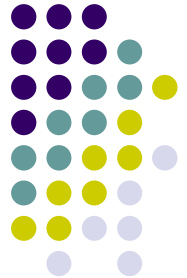
An example of a transcript

- For example
  - Int Max(a,b)
    - Max(1,2)=2
  - Int Min(a,b)
    - Min(5,4)=4



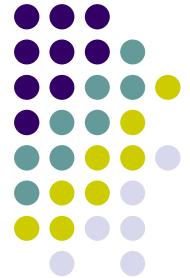
A transcript

Creates mock objects via a dynamic capture-replay approach



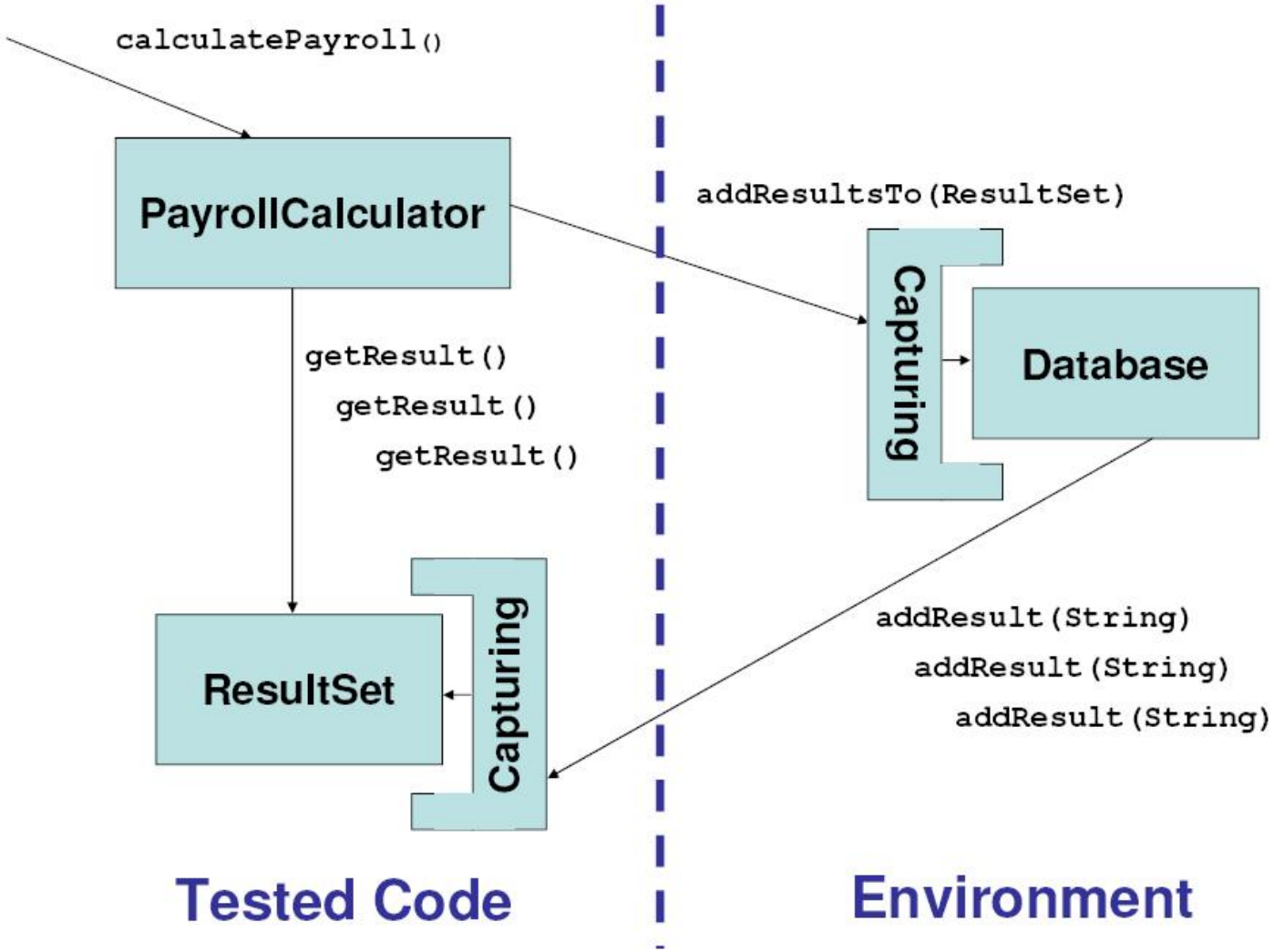
- The capture step
  - It occurs ahead of time, not at test time.
  - It executes in the context of the original system T|E, and records all interactions between T and E.
  - It can be thought of as encoding a transition function for object in the environment.

Creates mock objects via a dynamic capture-replay approach



- The replay phase
  - It occurs during execution of the factored tests, that is T'|E_m.
  - Real object E is replaced by mock object E_m.
  - Mock objcet checks that it was called with the same arguments as the next entry in the transcript.
  - If not, it throws a ReplayException.

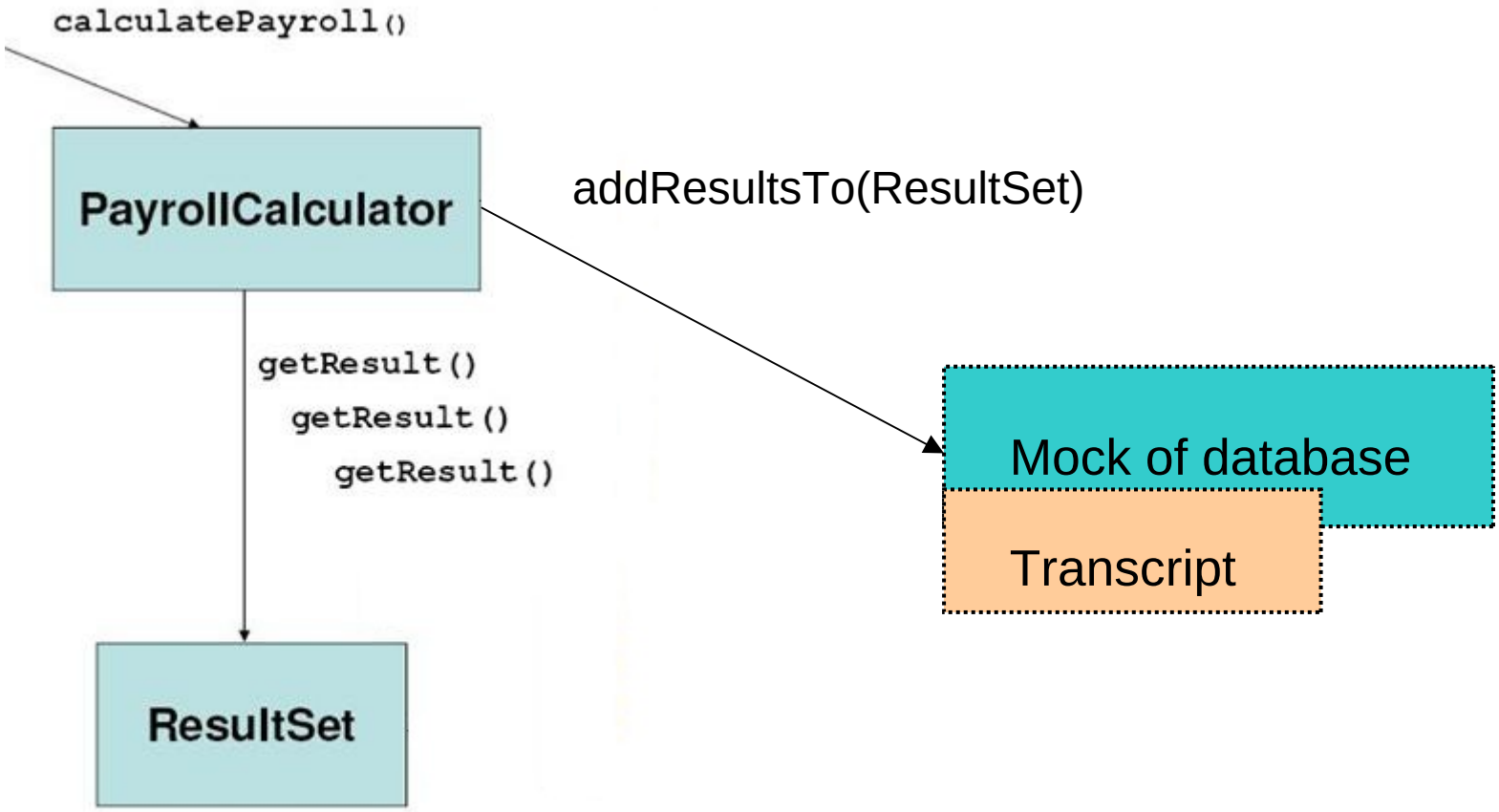
An example: The payroll system for behavior capture



T

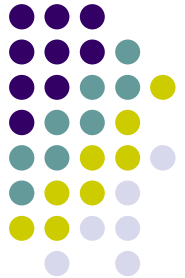
E

An example: The payroll system for behavior replay



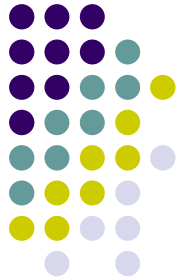
T’ E_m

Introducing interface to support testing factoring



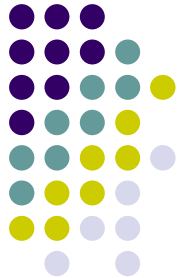
- Our approach proceeds in two steps:
  - The first, behavior-preserving step introduces a new interface for every class in the program.
  - The second step introduces new classes that implement the interface, therefore can be used in place of the original retrofitted ones.
- It must be possible for an instrumented class to co-exit with the uninstrumented version.

Capture and Replay Classes



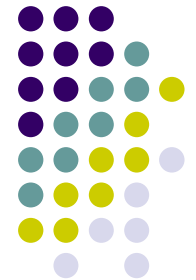
- Both the capture and replay steps replace some objects from the environment by different objects that satisfy the same specification.
- Only those objects that interact with the code under test need to be replaced.

Capture and Replay Classes



- While capturing, the replacement objects are wrappers around the real ones.
  - The wrappers delegate the work to the real objects and record arguments and return value.
- While replaying, the replacement objects are mock objects that read from the transcript.
  - A mock object verifies that the arguments are as expected and returns whatever the transcript indicates.

An example of introducing interface to a class C



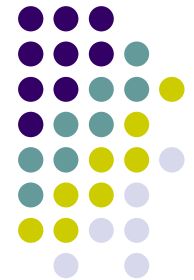
Before:

```
class C {
  Integer foo(int x) { ... }
  void bar(Date d) { ... }
}
```

After:

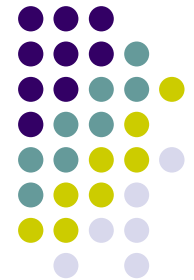
```
interface C__iface {
  Integer__iface foo__iface(int x);
  void bar__iface(Date__iface d);

class C implements C__iface {
  Integer__iface foo__iface(int x) { ... }
  void bar__iface(Date__iface d) { ... }
}
```

An example of capturing version of class



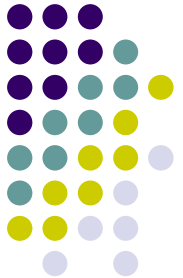
```
class C__capturing implements C__iface {
  C _delegate;
  Integer__iface foo__iface(int x) {
    ... // record arguments
    Integer result = _delegate.foo(x);
    ... // record results
    return getReferenceTo(result);
  }
  void bar__iface(Date__iface d) {
    ... // record arguments (no results to record)
    _delegate.bar(getReferenceTo(d));
  }
```

An example of capturing version of class



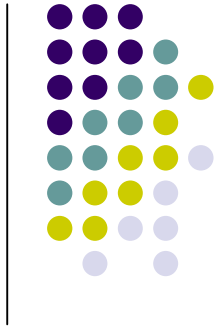
```
WeakHasccap<T, T__capturing> cc;
T__iface getReferenceTo(T__iface in) {
  if (in instanceof T__capturing) {
    return ((T__capturing) in)._delegate;
  } else if (in instanceof T) {
    T real = (T)in;
    if (!cc.contains(real)) {
      cc.put(in, new T__capturing(real));
    }
    return cc.get(real);
  } else {
    throw new Error("this can't happen");
  }
}
```

Case Study



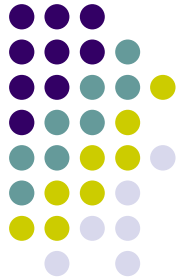
- Background
  - Two developer, one professional and one undergraduate, work independently on Daikon.
  - Daikon is a tool for detecting potential program invariants through dynamic analysis.

Measurements



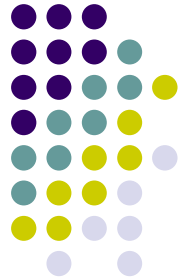
- Baseline
  - The unit tests take less time than compilation itself, test factoring is unlikely to help significantly.
  - The regression tests are 24 end-to-end system test that much more of Daikon, and running them with the **make** command completes in about 15 minutes.
  - They simulated running the tests after each CVS check-in, with and without test factoring.
  - Expect to test more frequently, and to build intuition about effective times to test.
  - Simulated continuous testing.

Measurements



- Quantities
  - Test time
  - Time of failure
  - Time to success
- Applying test factoring
  - If any factored test results in a ReplayException, the corresponding system test is then rerun.

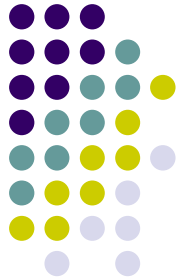
Experimental results



- Test factoring generally reduces the running time, but it is influenced by the developer's working style.
- For the CVS data, the tests always succeed, so the time to success is always the same as the test time.

	Test time	Time to failure	Time to success
Dev. 1	.79 (7.4/9.4 min)	1.56 (14/9 sec)	.59 (5.5/9.4 min)
Dev. 2	.99 (14.1/14.3 min)	1.28 (64/50 sec)	.77 (11.0/14.3 min)
CVS	.09 (0.8/8.8 min)	n/a	.09 (0.8/8.8 min)

Conclusion



- Test factoring mines fast, focused unit tests from slow system-wide tests; each new unit test exercises only a subset of the functionality exercised by the system test.
- Case study shows that test factoring can significantly reduce the running time of a system test suite.

Q&A