Unit Testing: Mock Objects

219343: Software Testing

Some materials are from Alberto Savoia's slides on unit testing,
George Necula's software engineering course,
and Hunt and Thomas, "Pragmatic Unit Testing," 2003.

Testing dependent classes

- Each unit usually interacts with other units
- Techniques
  - using stub
  - using mock objects

Using stub

- If your code calls `System.currentTimeMillis()`, and this return value is crucial to your testing:
  - Encapsulate this call.

```
public long getTime() {
    return System.currentTimeMillis();
}
```

  - Add stub

```
public long getTime() {
   if(debug)
      return debug_cur_time;
   else
      return System.currentTimeMillis();
}
```

  - Quite messy!

Using mock objects

- Use an interface to describe the object
- Implement the interface for production code
- Implement the interface in a mock object for testing

- With mock objects, you can do interaction-based testing

Mock objects: example

- Interface

```
public interface Environmental {
    public long getTime();
}
```

- Real implementation

```
public class SystemEnv implements Environmental {
    public long getTime() {
        return System.getTimeMillis();
    }
}
```

- Mock implementation

```
public class MockSystemEnv implements Environmental {
    public long getTime() {
        return current_time;
    }
    public void setTime(long t) {
        current_time = t;
    }
    private long current_time;
}
```

State-Based Testing & Interaction-Based Testing

- What we have done so far could be called "**state-based testing**."
  - We inject inputs into the objects, and see if their states change accordingly.
  - If there is no state change in the objects, it is difficult to use state-based testing.

- **Interaction-based testing** looks at how the objects interact.

Further reading: Martin Fowler's article "*Mocks Aren't Stubs*," and Nat Pryce's article "*State vs. Interaction Based Testing*". Google it.

Mock Libraries

- EasyMock
  - Create mock objects by “record-and-playback”
  - Easy to use
- jMock
  - Create mock by specifying how it interacts

Easy Mock

- **Easy Mock** is a tool that let you create a mock object and specify how it interacts using a record-and-replay approach.

- Eliminate the need to write a concrete class.

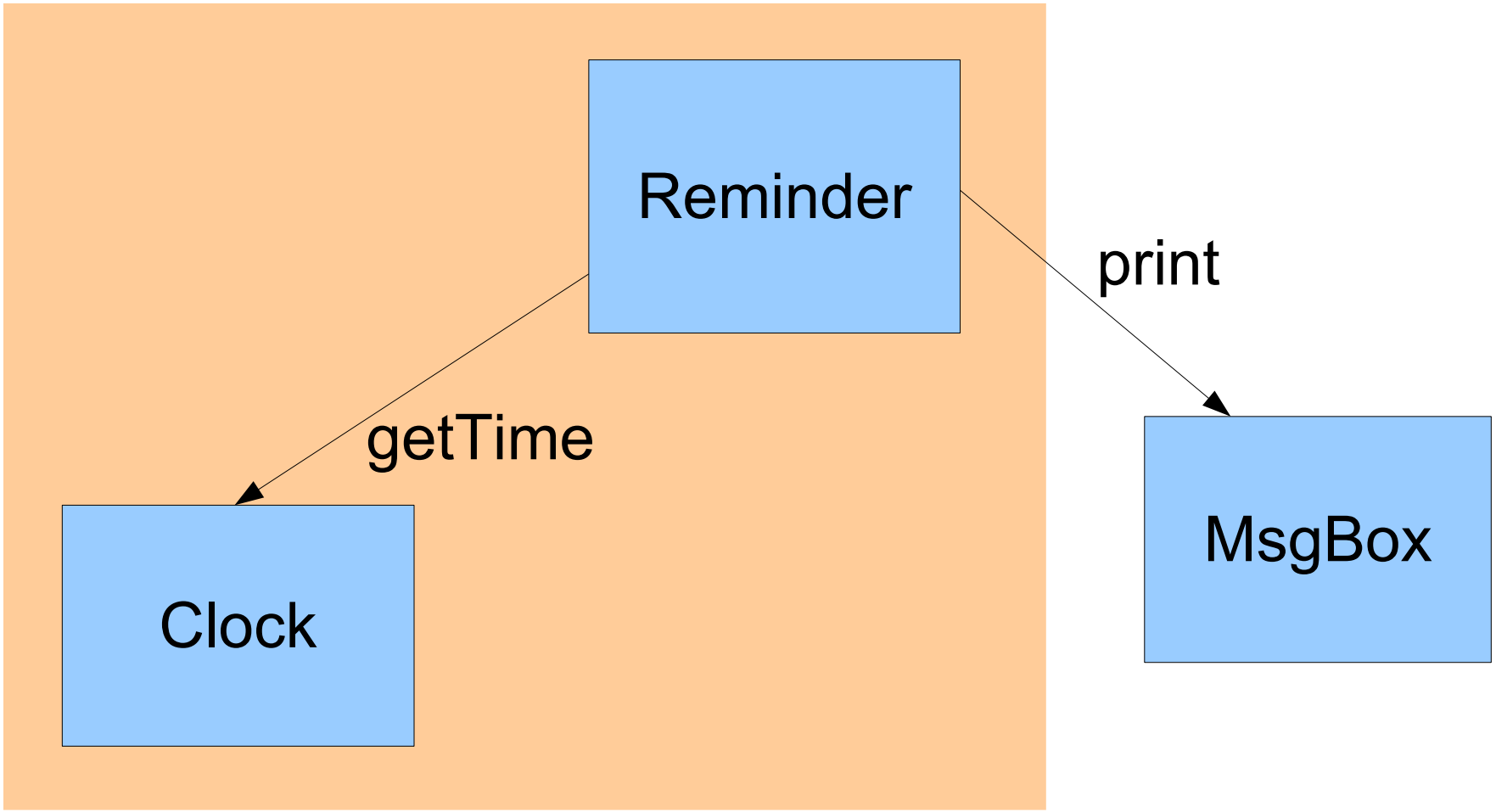
Easy Mock: Steps

- Record:
  - Create a mock object
  - Record the interaction, specify the return values
  - Press “replay”.
- Replay:
  - Run the test
  - The mock object would act as recorded.
  - In every step, it would verify all the interactions, i.e., all the calls.

Example: Reminder

```
public class Reminder { ...
  class Item { public int time; public String msg; }
  protected Vector<Item> items;
  public void refresh() {
    int currTime = getTime();
    for(Enumeration<Item> e=items.elements();
        e.hasMoreElements();) {
      Item i = e.nextElement();
      if((i.time> prevTime)&&(i.time <= currTime))
        System.out.println(i.msg);
    }
    prevTime = currTime;
}}
```

Refactor



Interface Clock

```
public interface Clock {
    public int getTime();
}
```

Constructor of Reminder

```
public Reminder(Clock clk) {
     clock = clk;
     prevTime = -1;
     items = new Vector<Item>();
}
```

Method Refresh

```
public void refresh() {

  int currTime = clock.getTime();

  for(Enumeration<Item> e=items.elements();
      e.hasMoreElements();) {...}
  prevTime = currTime;
}
```

Mocks: creating

```
@Before public void setUp() {

   cMock = createMock(Clock.class);

   rem = new Reminder(cMock);
}
```

Mocks: setting up

```
@Test public void testReminder() {

   expect(cMock.getTime()).andReturn(1);
   expect(cMock.getTime()).andReturn(2);
   expect(cMock.getTime()).andReturn(3);

   replay(cMock);

   ..
}
```

Mocks: using & verifying

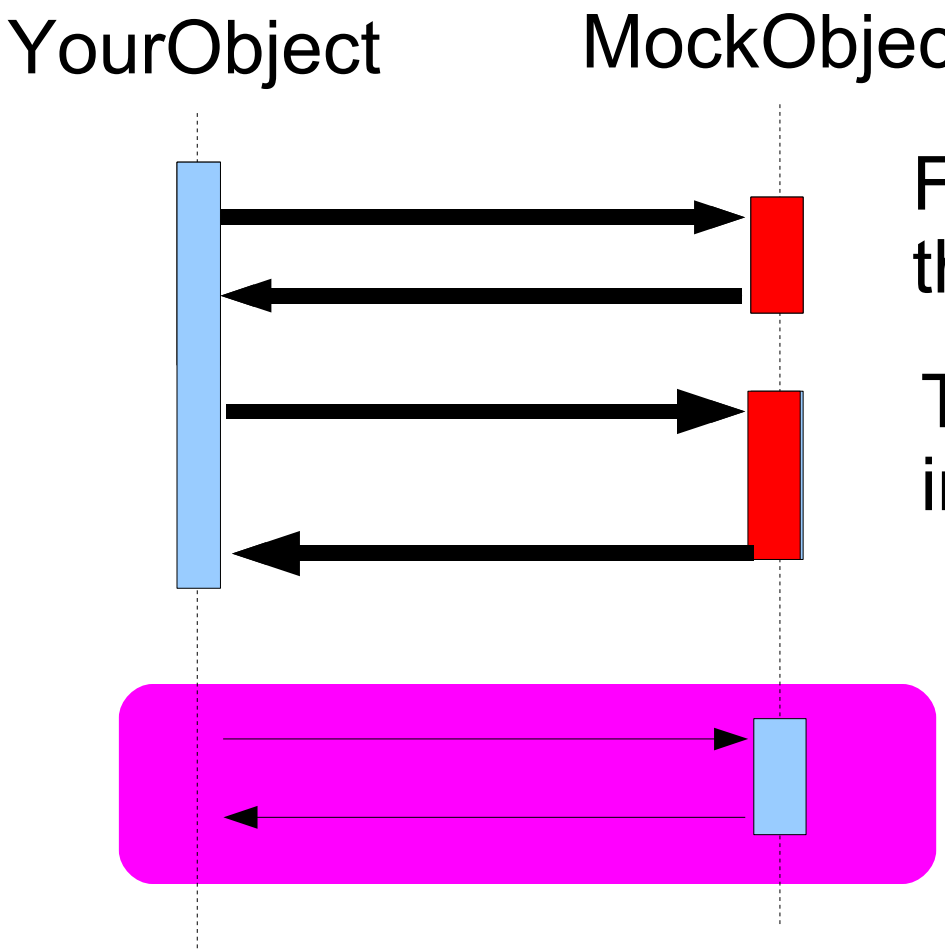
```
@Test public void testReminder() {
  expect(cMock.getTime()).andReturn(1);   ...
  replay(cMock);

   rem.add(1,”hello1”);
   rem.add(2,”hello2”);
   rem.refresh();
   rem.refresh();
   rem.refresh();

   verify(cMock);
}
```

What does EasyMock do?

- Checks the interaction.

- At the end, a call to `verify` makes sure that every specified interaction is called.

Interactions



First, you program the interaction.

Then, check the interactions.

Verify makes sure that you every interaction is made.

More on Easy Mock

- See the demo.

- http://www.easymock.org/
- Document: http://www.easymock.org/EasyMock2_3_Documentation.html

Note for JUnit

- Testing in a project
  - Declare members as *protected* so that testcases in the same package can see it.
  - If we want to place the testcases in another directory, we can duplicates the program package directory structure so that the testcases are still in the same package.

Design for test

- Testing force you to reorganize your design
- (More on this later)

Conclusion

- Unit testing is important
  - Mainly a partial correctness assertion
    - Weak assertion / strong assertion
- Good test:
  - RIGHT-BICEP
- Unit testing dependent systems
  - Use stub and mock object