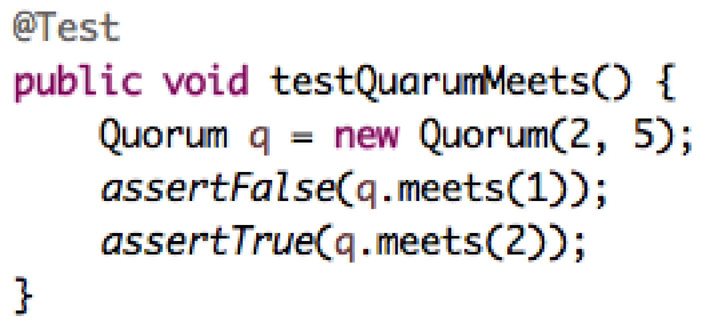
COM S 362
Object-Oriented Analysis & Design

Design and Testing

Good Design → Good Testing

- Open-Closed Principle: Good design makes it easy to add features without modifying existing code
- Open-Closed Principle applied to testing: Good design makes it easy to write tests without modifying code

What is automated testing?

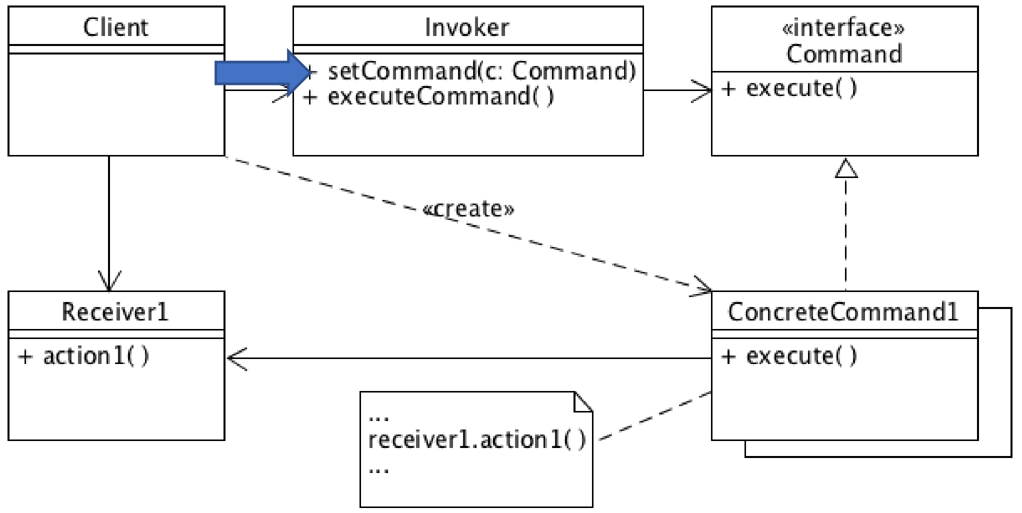


```
@Test
public void testQuarumMeets() {
    Quorum q = new Quorum(2, 5);
    assertFalse(q.meets(1));
    assertTrue(q.meets(2));
}
```

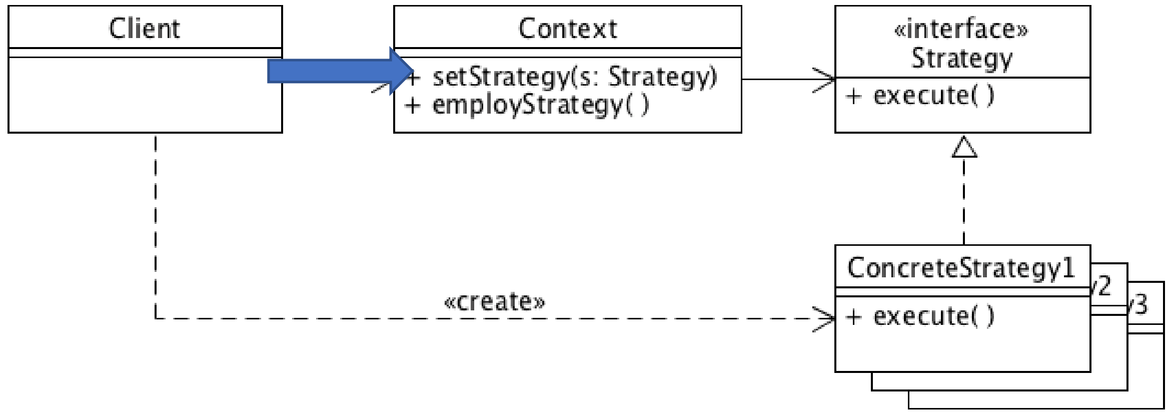
- Types of testing
  - Unit tests – test individual methods and classes
  - Integration tests – verify modules or services work together
  - End-to-end tests – test user interactions with complete application environment
- Code coverage
  - Function coverage – number of functions executed during tests
  - Statement coverage - number of statements executed during tests
  - Branch or decision coverage – number of decision control structures (e.g., if and while) executed
  - Condition coverage – testing both true and false branches of control structures
  - Path coverage – flows through sequences of conditions

Dependency Injection

- In many design patterns we have seen a common theme regarding creation
- **Dependency injection** - an object receives the objects it depends on rather than creates them itself
- An object that uses services should not have to know how to construct the services



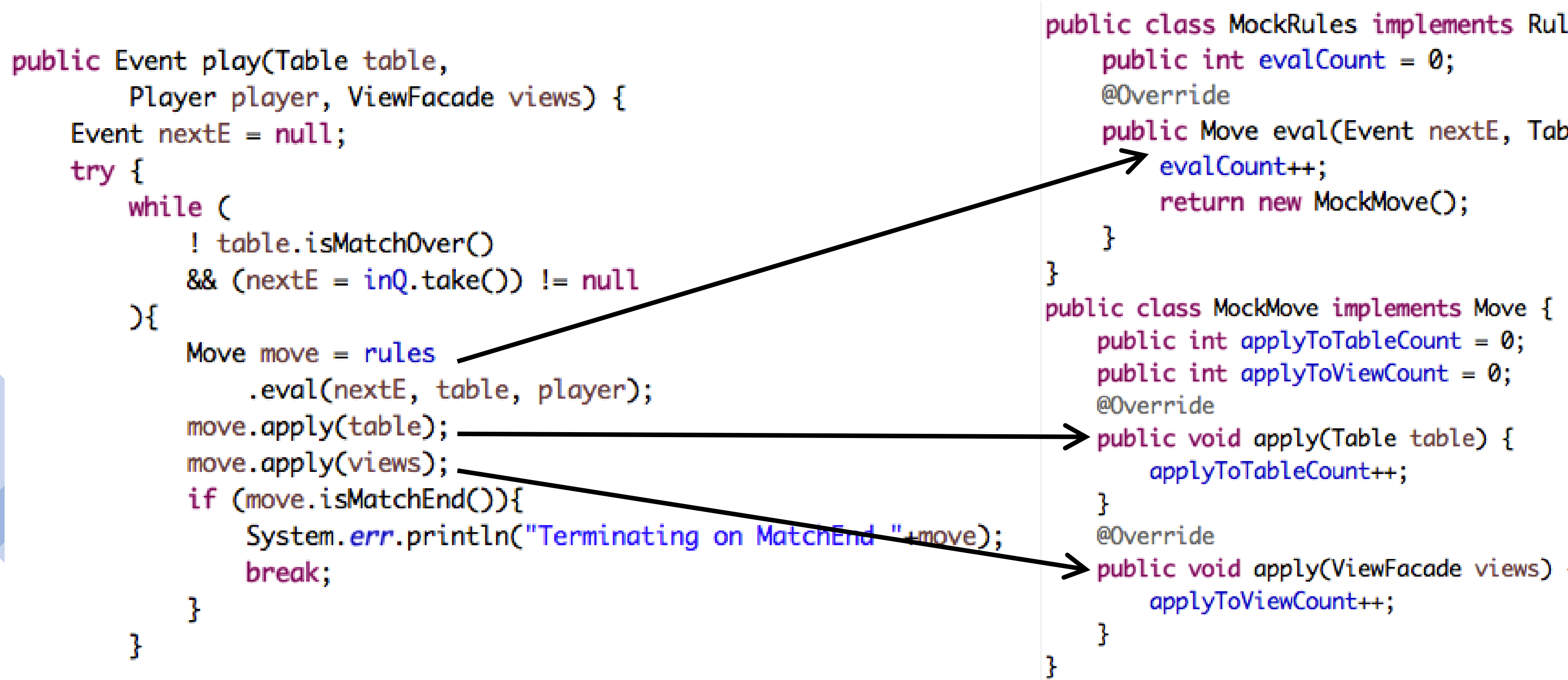
Command Pattern: Commands injected into Invoker.



Strategy Pattern: Strategies injected into Context.

Why dependency <u>injection</u> matters for testing?

- A common method for instrumenting tests is to create dummy or **mock objects**
  - Implement the same interface as the real objects but only provide default behaviors
  - Purpose is to monitor the object under test
  - For example, confirms that a method was called with the correct parameters



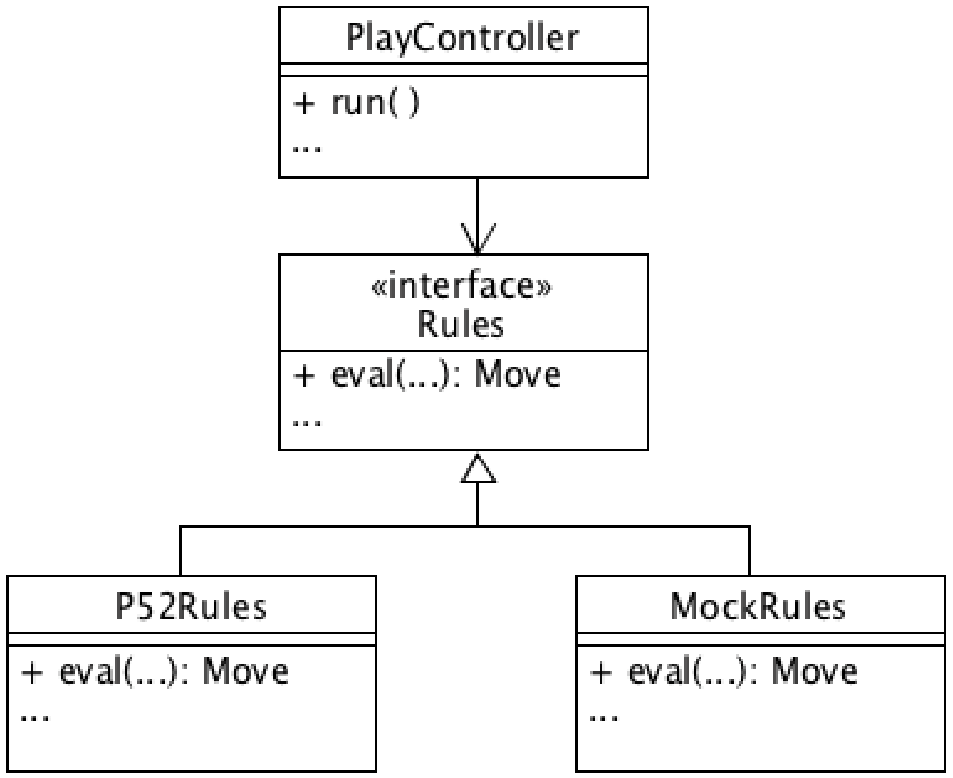
```
public Event play(Table table,
        Player player, ViewFacade views) {
    Event nextE = null;
    try {
        while (
            ! table.isMatchOver()
            && (nextE = inQ.take()) != null
        ){
            Move move = rules
                .eval(nextE, table, player);
            move.apply(table);
            move.apply(views);
            if (move.isMatchEnd()){
                System.err.println("Terminating on MatchEnd "+move);
                break;
            }
        }
```

```
public class MockRules implements Rules {
    public int evalCount = 0;
    @Override
    public Move eval(Event nextE, Table table, Player player) {
        evalCount++;
        return new MockMove();
    }
}
public class MockMove implements Move {
    public int applyToTableCount = 0;
    public int applyToViewCount = 0;
    @Override
    public void apply(Table table) {
        applyToTableCount++;
    }
    @Override
    public void apply(ViewFacade views) {
        applyToViewCount++;
    }
}
```

Mocks to help write test confirming that PlayController applies events to rules and applies the resulting moves to the table and views.

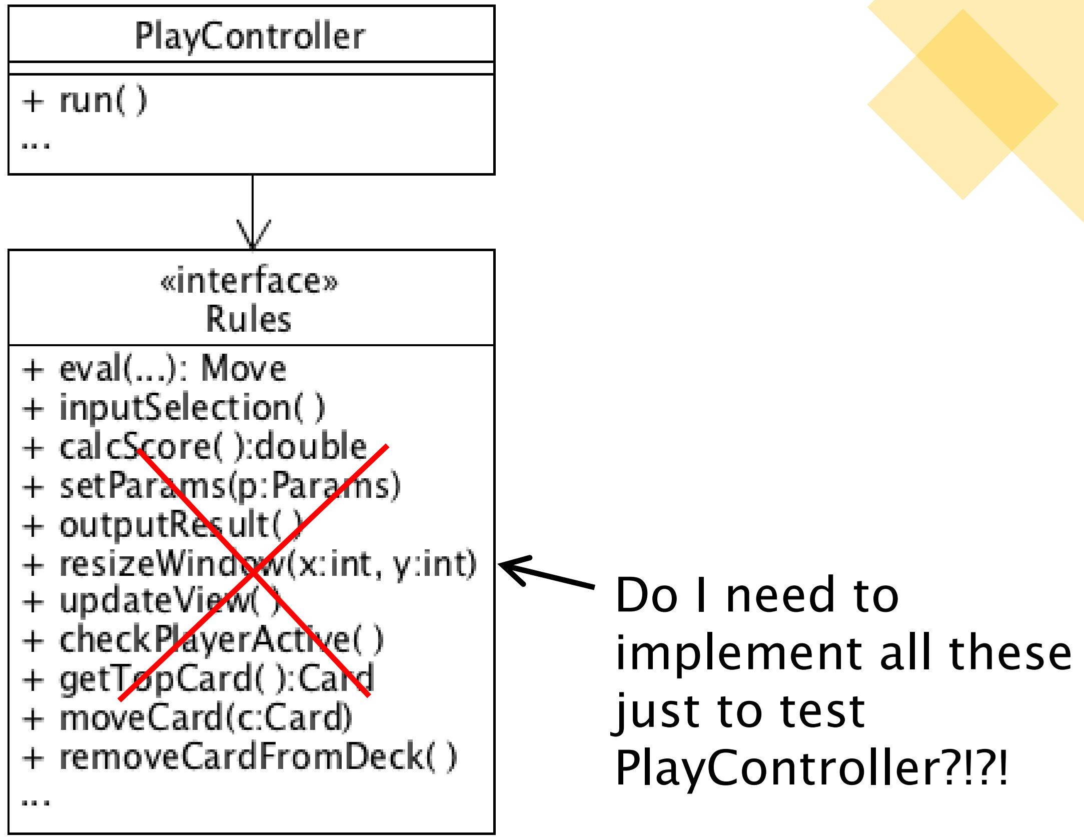
Dependency Inversion and Testing

- **Decency Inversion** – high-level classes should depend on interfaces, not classes
- Impact on testing: mock classes are less likely to require change when they are only implementing an interface
- If changes to a class require changes to mock/test code then the **test framework is adding complexity and time cost!**



Interface Segregation on Testing

- Often setting up a test can take as much time as writing the code that is being tested
- Big interfaces don't make the requirements of a class clear
  - Waste time creating mock methods that are not required
- Narrowly defined interfaces make requirements clear



Separation of Concerns on Testing

- **“Functions should do one thing. They should to it well. They should to it only.”**
  — Robert C. Martin, Clean Code
- Classes that have a cohesive set of responsibilities are easier to test
- Methods that “do one thing” are easier to test

Encapsulation on Testing

- **Encapsulation** – don’t expose implementation details
  - May seem to make testing more difficult
  - Temptation is to make all class member variables public/protected or add getter and setters everywhere just for the purposes of testing
- If we have a good open-closed design, well encapsulated code should not be more difficult to test
- Use mocks to test the expected behavior