McGill

COMP 303

Lecture 13

Unit testing III

Winter 2025

slides by Jonathan Campbell

Announcements

- Proposal grades coming in next few days.

Today

- Unit testing
    - Stubs
    - Test coverage
    - PyTest & project example

Recap

Metaprogramming

- Typically, when we write code, we operate on data that represent real-world objects (like Cards, or bank records, or shapes, etc.).
- When we write code that operates on other code (e.g., writing code to test other classes, methods, fields, etc.), it is called **metaprogramming**.
  - In Java, metaprogramming is called **reflection**.

The Class<T> type

- Class<T> is the primary mechanism for introspection in Java.
- Every Java class is represented at runtime by an instance of Class<T>, where T is the class name.
- An object of type Class<T> holds metadata about the class that it represents; e.g., all the methods/fields inside.

Obtaining a Class<T> object

- Here's an example that passes a string to Class.forName, and uses Class<?>, where ? is a type wildcard, meaning that the class could be of any type (Card, Deck, etc.).

```
public static void main(String[] args) {
  try {
    Class<?> theClass = Class.forName(args[0]);
  } catch(ClassNotFoundException e) {
    e.printStackTrace();
  }
}
```

Creating a new instance

```
try {
  Card card1 = Card.get(Rank.ACE, Suit.CLUBS);

  Constructor<Card> cardConstructor =
       Card.class.getDeclaredConstructor(Rank.class, Suit.class);
  cardConstructor.setAccessible(true);

  Card card2 = cardConstructor.newInstance(Rank.ACE, Suit.CLUBS);

  System.out.println(card1 == card2);
} catch (ReflectiveOperationException e) {
  e.printStackTrace();
}
```

get the constructor

make it public

call the constructor

Modifying fields

```
try {
  Card card = Card.get(Rank.TWO, Suit.CLUBS);

  Field rankField = Card.class.getDeclaredField("aRank");
  rankField.setAccessible(true);

  rankField.set(card, Rank.ACE);

  System.out.println(card);
} catch (ReflectiveOperationException e) {
  e.printStackTrace();
}
```

get the field

make it public

set it to Ace for the given card

Basic principles for unit tests

- **speed**: unit tests should avoid things like intensive device I/O or network access, because they are run often.
- **independence**: each unit test should execute in isolation; it should not depend on another test running first, because we often like to execute a single test, and testing frameworks don't guarantee the order of execution of tests

Basic principles for unit tests

- **repeatable**: unit tests should produce the same result in different environments; they should not depend on environment-specific properties.
- **focus**: tests should exercise and verify a slice of code execution behaviour that is narrow as reasonably possible, to be able to pinpoint exactly what is wrong.
- **readability**: the UUT, input data and oracle (expected result), and the rationale, should all be easily identifiable.

Testing for exceptional conditions

```
class FoundationPile {
  boolean isEmpty() { ... }
  /*
  * @return The card on top of the pile.
  * @throws EmptyStackException if isEmpty()
  */
  Card peek() {
    if (isEmpty()) {
      throw new EmptyStackException();
    }
    ...
  }
}
```

This method throws an exception under some condition.
We should write a unit test to verify this case.
(If it doesn't throw the exception, then it is faulty.)

Testing for exceptional conditions

```
@Test
public void testPeek_Empty() {
  assertThrows(EmptyStackException.class, () -> aPile.peek());
}
```

Lambda expression

Unit testing III

Testing for exceptional conditions

```
class FoundationPile {
  boolean isEmpty() { ... }

  /*
  * @return The card on top of the pile.
  * @pre !isEmpty()
  */
  Card peek() { ... }
}
```

This method uses design by contract (preconditions specified), so we should not write a unit test for this condition, since the behaviour is unspecified.

Testing private methods

- We may argue that private methods shouldn't be tested because they are internal elements of other, accessible methods.
- We may also argue that we should test them and just ignore the private modifier.

Testing private methods

- Assume we want to test the following method of FoundationPile:

```
private Optional<Card> getPreviousCard(Card pCard) { ... }
```

Testing private methods

```
public class TestFoundationPile {
  private FoundationPile aPile = new FoundationPile();
  private Optional<Card> getPreviousCard(Card pCard) {
    try {
      Method method = FoundationPile.class.
               getDeclaredMethod("getPreviousCard", Card.class);
      method.setAccessible(true);
      return (Optional<Card>) method.invoke(aPile, pCard);
    } catch (ReflectiveOperationException exception) {
      fail();
      return Optional.empty();
    }
  }
  @Test
  public void testGetPreviousCard_empty() {
    assertFalse(getPreviousCard(Card.get(Rank.ACE, Suit.CLUBS)).
                                        isPresent());
  }
}
```

Uses metaprogramming to make the private field accessible, then calls it with invoke.

Gets the result of the helper function.

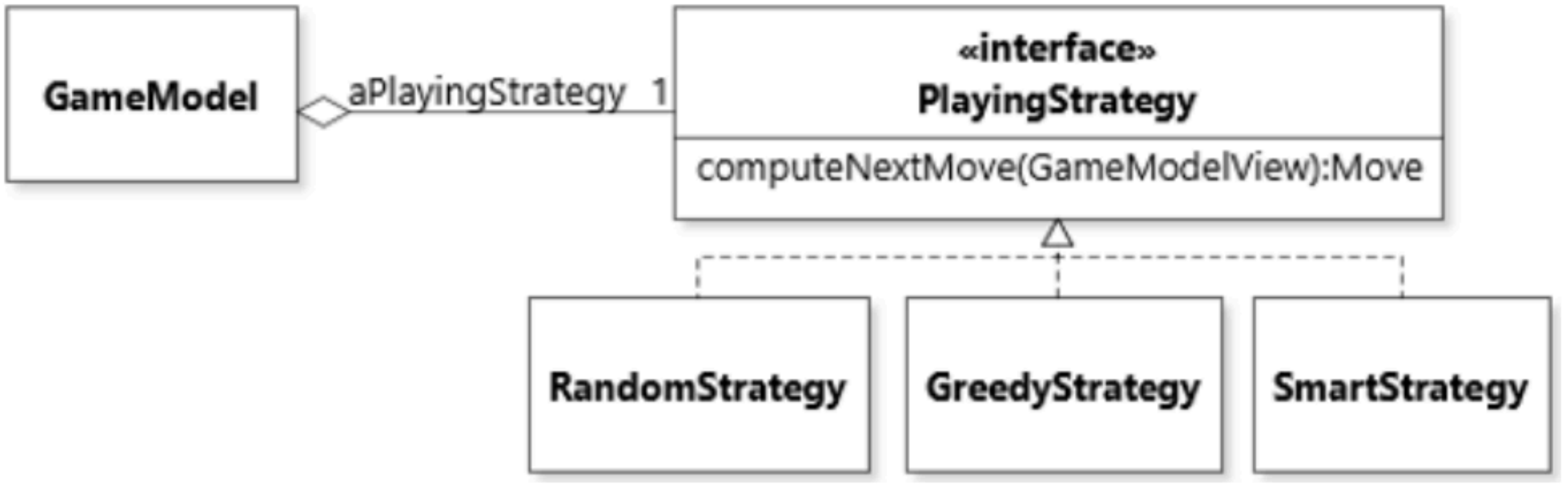
Testing with stubs

- Unit testing: test small parts of code **in isolation**. But sometimes, it can be difficult to do so, if the code we want to test:
    - triggers the execution of a large chunk of other code;
    - includes sections whose behaviour depends on the environment (e.g., system fonts);
    - involves non-deterministic behaviour (e.g., randomness).

Testing with stubs

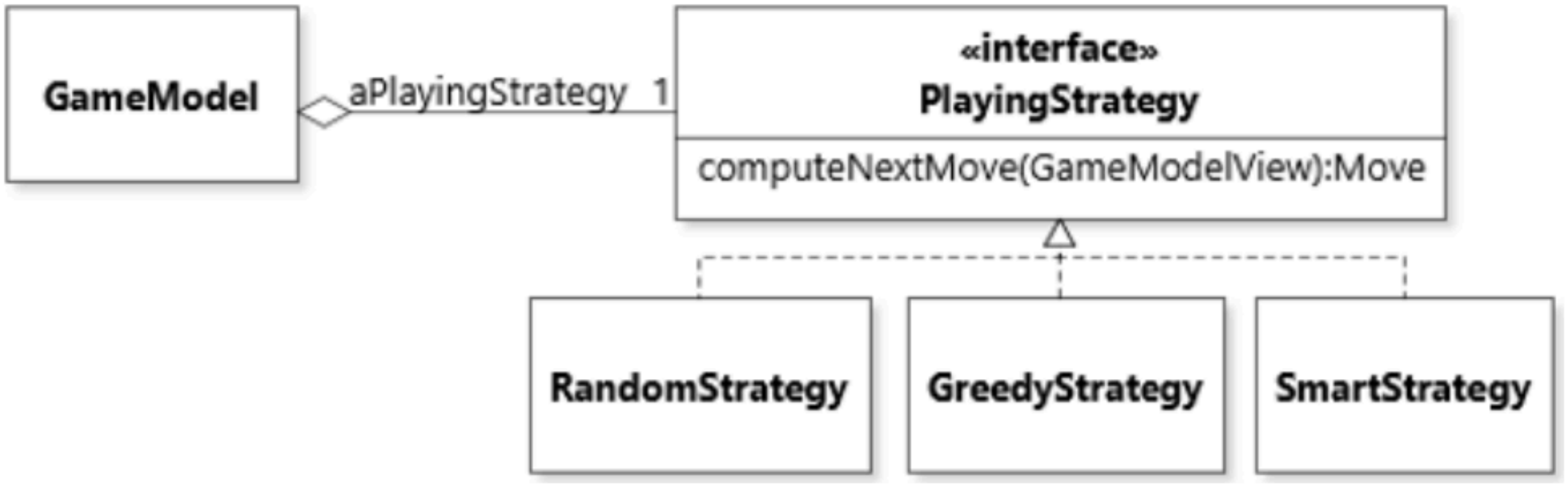
Assume there is a method GameModel.tryToAutoPlay() which plays a game of Solitaire according to a PlayingStrategy.

We want to write a unit test for such method.



Testing with stubs

Assume there is a method GameModel.tryToAutoPlay() which plays a game of Solitaire according to a PlayingStrategy.

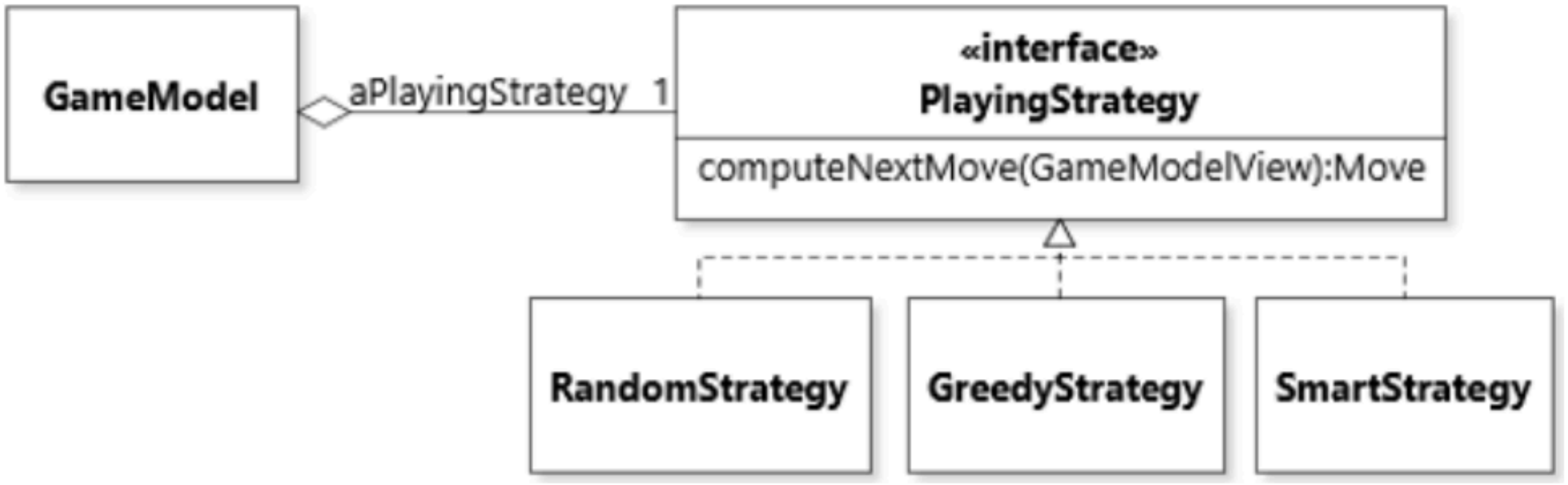


Problems:

1) The method will call computeNextMove of a PlayingStrategy, probably a complex method, so we are no longer testing a small part.

Testing with stubs

Assume there is a method GameModel.tryToAutoPlay() which plays a game of Solitaire according to a PlayingStrategy.

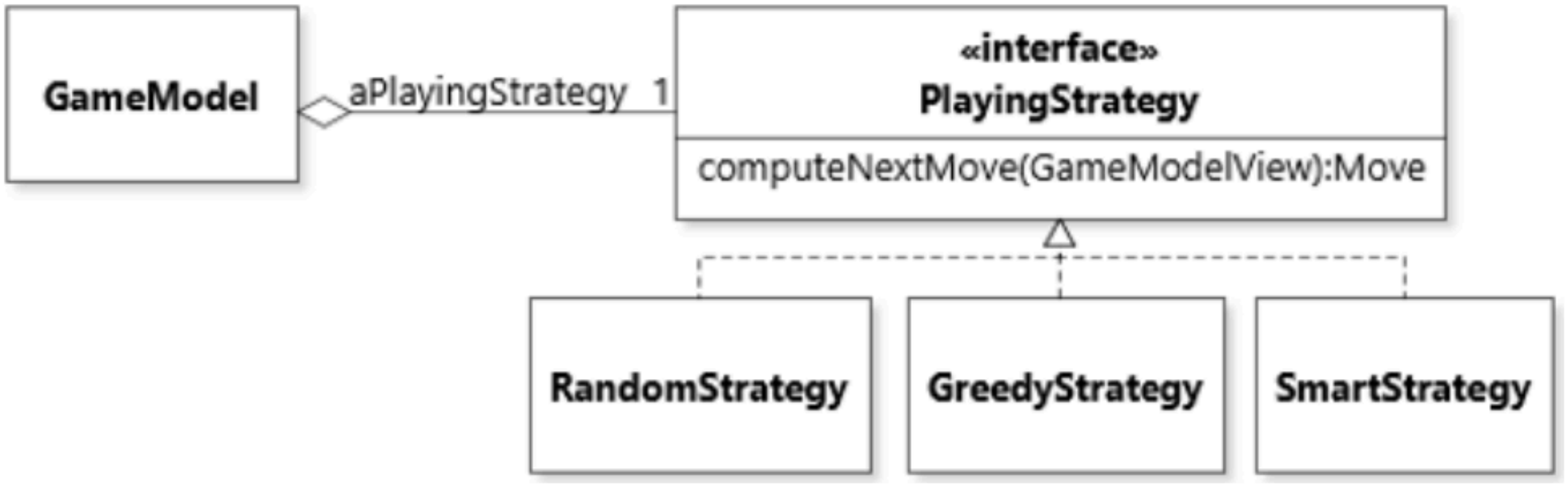


Problems:

2) The particular strategy may involve some randomness (e.g., RandomStrategy), breaking the repeatability principle.

Testing with stubs

Assume there is a method GameModel.tryToAutoPlay() which plays a game of Solitaire according to a PlayingStrategy.

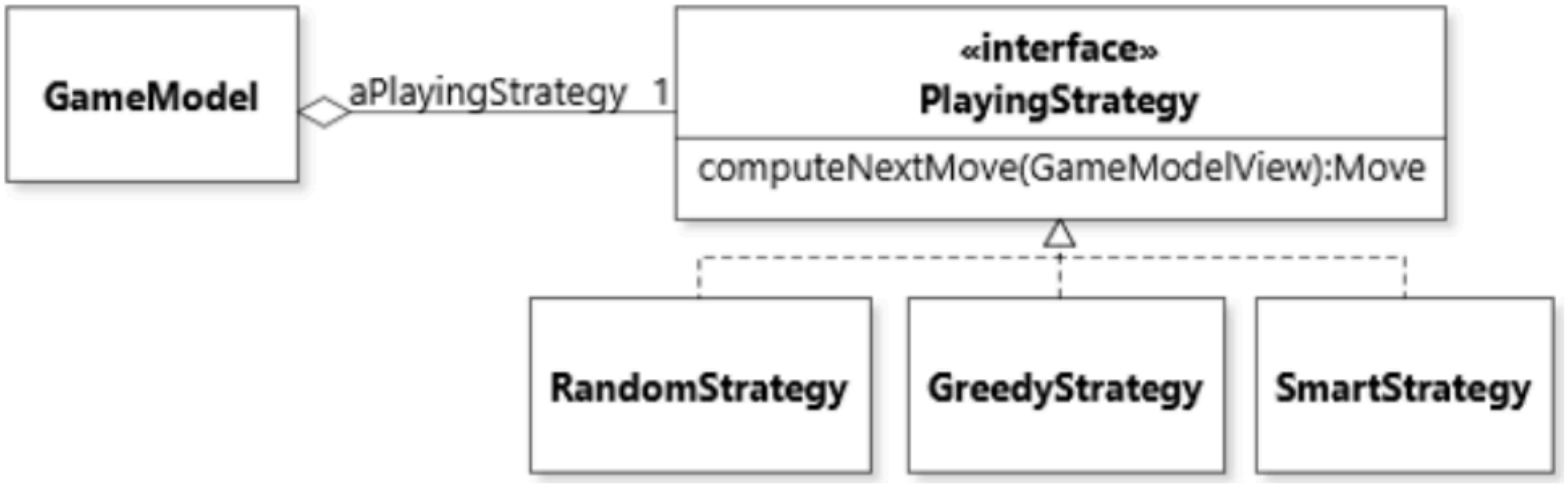


Problems:

3) We don't know which strategy will be used, so how can we test the expected result?

Testing with stubs

Assume there is a method GameModel.tryToAutoPlay() which plays a game of Solitaire according to a PlayingStrategy.



Problems:

4) How is testing tryToAutoPlay different than testing the individual strategies on their own?

Testing with stubs

- Solution: Test only the part of tryToAutoPlay that you can't test by other means.
    - Since we can test the computeNextMove of individual strategies on their own, we don't want to test that part in tryToAutoPlay.
    - We simply want to test its main purpose, which is to delegate the computation of next move to the particular strategy object.
    - To test only this small part, we write a **stub**.

Testing with stubs

- A stub is a greatly simplified version of an object that mimics its behaviour only enough to support the testing of a UUT that uses this object.

- For testing of tryToAutoPlay, a stub would be a new PlayingStrategy that does (almost) nothing.

Testing with stubs

```
public class TestGameModel {
  static class StubStrategy implements PlayingStrategy {
    private boolean aExecuted = false;
    public boolean hasExecuted() {
      return aExecuted;
    }
    public Move computeNextMove(GameModelView pModelView) {
      aExecuted = true;
      return new NullMove();
    }
  }
}
```

Just keep track of whether the method has been called.

NullMove: application of Null Object pattern.

Testing with stubs

```
@Test
public void testTryToAutoPlay() {
  Field strategyField =
     GameModel.class.getDeclaredField("aPlayingStrategy");
  strategyField.setAccessible(true);
  StubStrategy strategy = new StubStrategy();
  GameModel model = GameModel.instance();
  strategyField.set(model, strategy);

  model.tryToAutoPlay();
  assertTrue(strategy.hasExecuted());
}
```

set the field's value to the strategy

check that the strategy's computeNextMove was called.

Test coverage

- As we've discussed, it's very unlikely that we can exhaustively test methods.
    - For example, to test methods of the Deck class, there are 2.2×10^{68} number of possible decks of 52 playing cards; we could never run through them all.
- We want to select some subset of these possibilities to test. There are two ways to do so.

Test coverage

- **Functional (black-box) testing**.
  - Cover as much of the specified behaviour of a UUT as possible, based on some external specification.
    - E.g., for Deck.draw(), the specification is that the method should result in the top card of the deck being removed and returned.
  - In other words, we don't care about how it's implemented, and don't want to test that it is implemented in any particular way, but just want to verify that it does what it should.
  - Advantages: don't need to access the code of the UUT, can reveal problems with the specification and missing logic.

Test coverage

- **Structural (white-box) testing**.
  - Cover as much of the implemented behaviour of the UUT as possible, based on an analysis of the source code of the UUT.
  - Advantage: can reveal problems caused by low-level implementation details.
  - We will focus on this kind of testing.

canMoveTo: structural testing

For structural testing, we check the source code of the UUT.

```
boolean canMoveTo(Card pCard) {
  if (isEmpty()) {
    return pCard.getRank() == Rank.ACE;
  }
  else {
    return pCard.getSuit() == peek().getSuit() &&
      pCard.getRank().ordinal() == peek().getRank().ordinal()+1;
  }
}
```

We can see that the code is split into an if/else.
It would be good to test both parts.

Each if/else also has multiple parts;
we're starting to get a lot of tests!

Test coverage

- Coverage: how much of a program's source code is tested by a set of tests, e.g., how much of the code executes.
- There are several metrics that can evaluate test coverage in different ways:
    - statement coverage
    - branch coverage

Statement coverage

- The number of statements executed by test(s).

```java
@Test
public void testCanMoveTo_Empty() {
  assertTrue(aPile.canMoveTo(ACE_CLUBS));
  assertFalse(aPile.canMoveTo(THREE_CLUBS));
}
```

- This test achieves 67% coverage for canMoveTo, because the if condition and statement in the if branch is executed, but the statement in the false branch is not.

Statement coverage

- Statement coverage is a poor metric, because it heavily depends on the detailed structure of the code.
- By rewriting canMoveTo in a different way, we can achieve 100% test coverage with the same test.
    - Also, not all statements are created equally. Some are a lot more complex than others.

Statement coverage

```java
boolean canMoveTo(Card pCard) {
  boolean result = pCard.getSuit() == peek().getSuit() &&
    pCard.getRank().ordinal() == peek().getRank().ordinal()+1;
  if (isEmpty()) {
    result = pCard.getRank() == Rank.ACE;
  }
  return result;
}
```

Branch coverage

- Number of program branches executed by test(s) divided by the total number of branches in the code of interest.
- We consider a branch to be:
    - an outcome of a condition (if / else if / else)
    - an outcome of a boolean expression within a statement

Branches in canMoveTo

```
boolean canMoveTo(Card pCard) {
  if (isEmpty()) {
    return pCard.getRank() == Rank.ACE;
  }
  else {
    return pCard.getSuit() == peek().getSuit() &&
      pCard.getRank().ordinal() == peek().getRank().ordinal()+1;
  }
}
```

if

boolean expression

boolean expression

boolean expression

Each condition/boolean expression has two outcomes: true or false. Thus, there are 8 branches in this code.

Branch coverage

```java
@Test
public void testCanMoveTo_Empty() {
  assertTrue(aPile.canMoveTo(ACE_CLUBS));
  assertFalse(aPile.canMoveTo(THREE_CLUBS));
}
```

This test thus tests 3 of the 8 branches (37.5% branch coverage), the particular ones being "true" for the if condition, and "true" and "false" for the boolean expression inside the if body.

Branch coverage

```java
@Test
public void testCanMoveTo_NotEmptyAndSameSuit() {
  aPile.push(ACE_CLUBS);
  assertTrue(aPile.canMoveTo(TWO_CLUBS));
  assertFalse(aPile.canMoveTo(THREE_CLUBS));
}
```

By adding this test, we now get 7/8 = 87.5% branch coverage for our two tests.

We are missing the branch where the pile is not empty, and the card at the top is of a different suit than the card passed in.

Branch coverage

- One of the most useful test coverage metrics.
- Well supported by testing tools.
- Subsumes statement coverage.

Unit testing in Python

- There are various unit testing frameworks in Python.
- We will go over PyTest.
- We must first install it with `pip3 install pytest`.

Unit tests in PyTest

```python
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

Running tests with PyTest

- We can run our tests by running **`pytest test_class.py`** in the command line.
    - (The command will be slightly different for the project, since we will structure our test files differently.)

Unit tests in PyTest

```
class TestClassDemoInstance:
    value = 0

    def test_one(self):
        self.value = 1
        assert self.value == 1

    def test_two(self):
        assert self.value == 1
```

Each test has a unique instance of the test class.

So test_two won't see the value changed in test_one.

Checking for exceptions

```
class TestClass:
  def test_one(self):
    with pytest.raises(ZeroDivisionError):
      some_method()
```

Fixtures

```
@pytest.fixture
def fruit_bowl():
  return [Fruit("apple"), Fruit("banana")]
```

By using the name of a fixture in the parameter list,
it will automatically be called and passed in.

```
def test_fruit_salad(fruit_bowl):
  fruit_salad = FruitSalad(*fruit_bowl)

  for fruit in fruit_salad.fruit:
     assert fruit.cubed
```

Unit tests for the project

- You will be required to write unit tests for your project code.
- We will go over an example of this.

References

- Robillard ch. 5.8-5.9 (p. 117-124)
  - Exercises #8-11: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter5.md

Coming up

- Next lecture:
    - Inheritance