COMP 303

Lecture 11

Unit testing

Class will start at 2:35 p.m.

Winter 2025

slides by Jonathan Campbell

Announcements

- Review guidelines out
- TAs are grading proposals
- https://brutecat.com/articles/leaking-youtube-emails

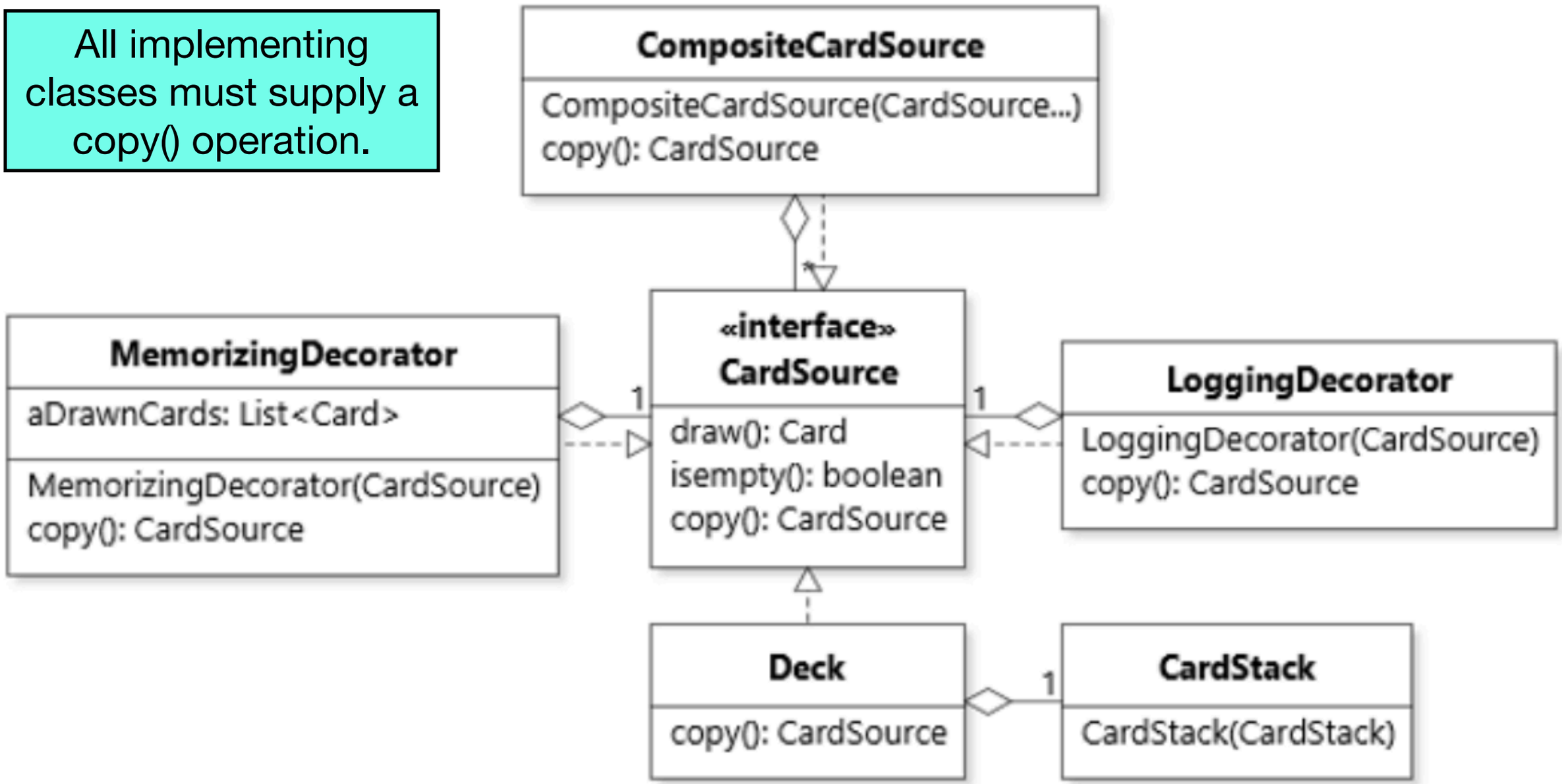
Today

- Law of Demeter
- Unit testing
  - Introduction to unit testing and JUnit
  - Metaprogramming (reflection): Class<T>
  - Stubs & test coverage
  - PyUnit

Recap

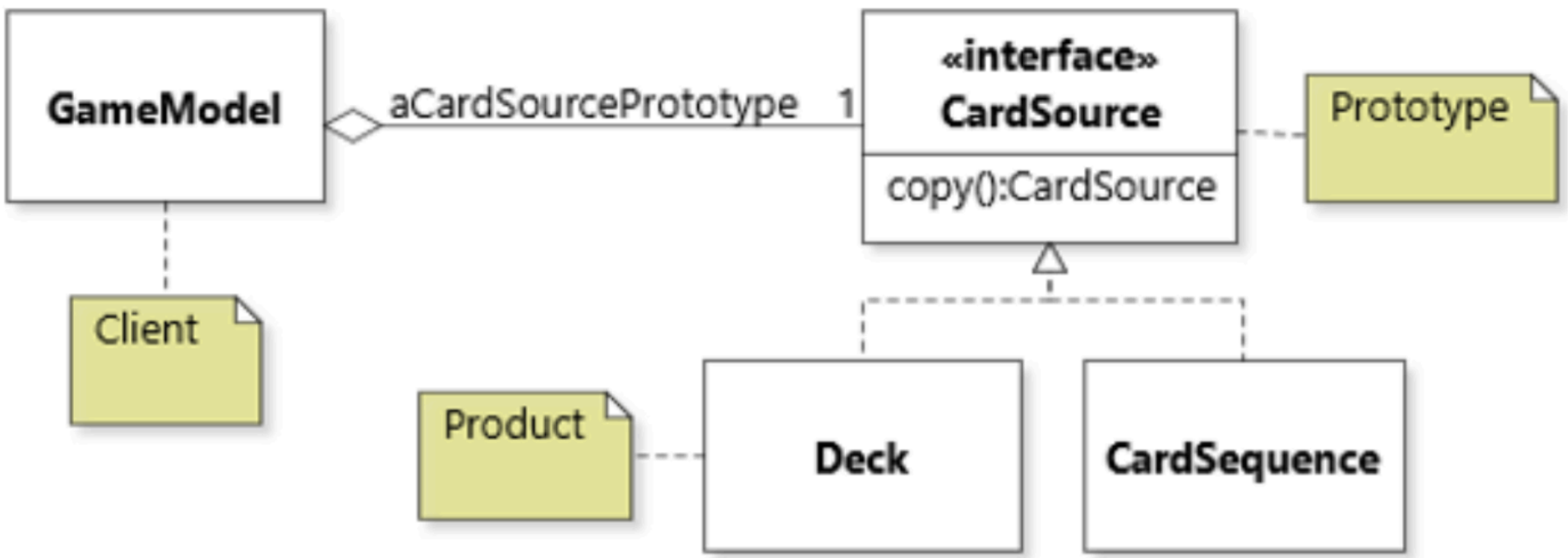
Polymorphic copying

All implementing classes must supply a copy() operation.



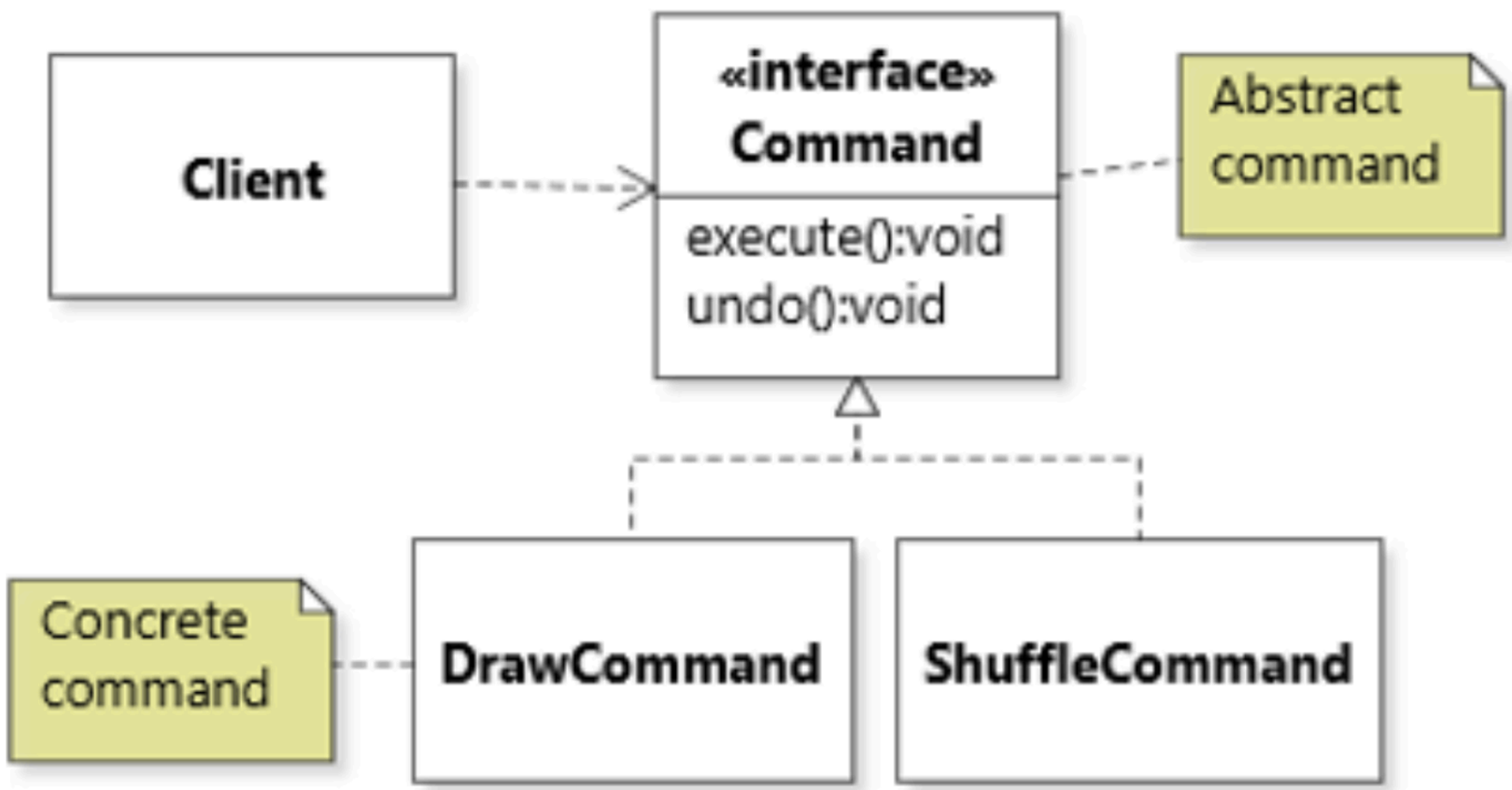
PROTOTYPE pattern

Prototype: the *abstract* element (typically an interface) whose *concrete* prototype must be instantiated at runtime.



Products: the objects that can be created by copying the prototype.

COMMAND pattern



Each piece of functionality is defined in its own class, which implements a Command interface.

Commands in the project

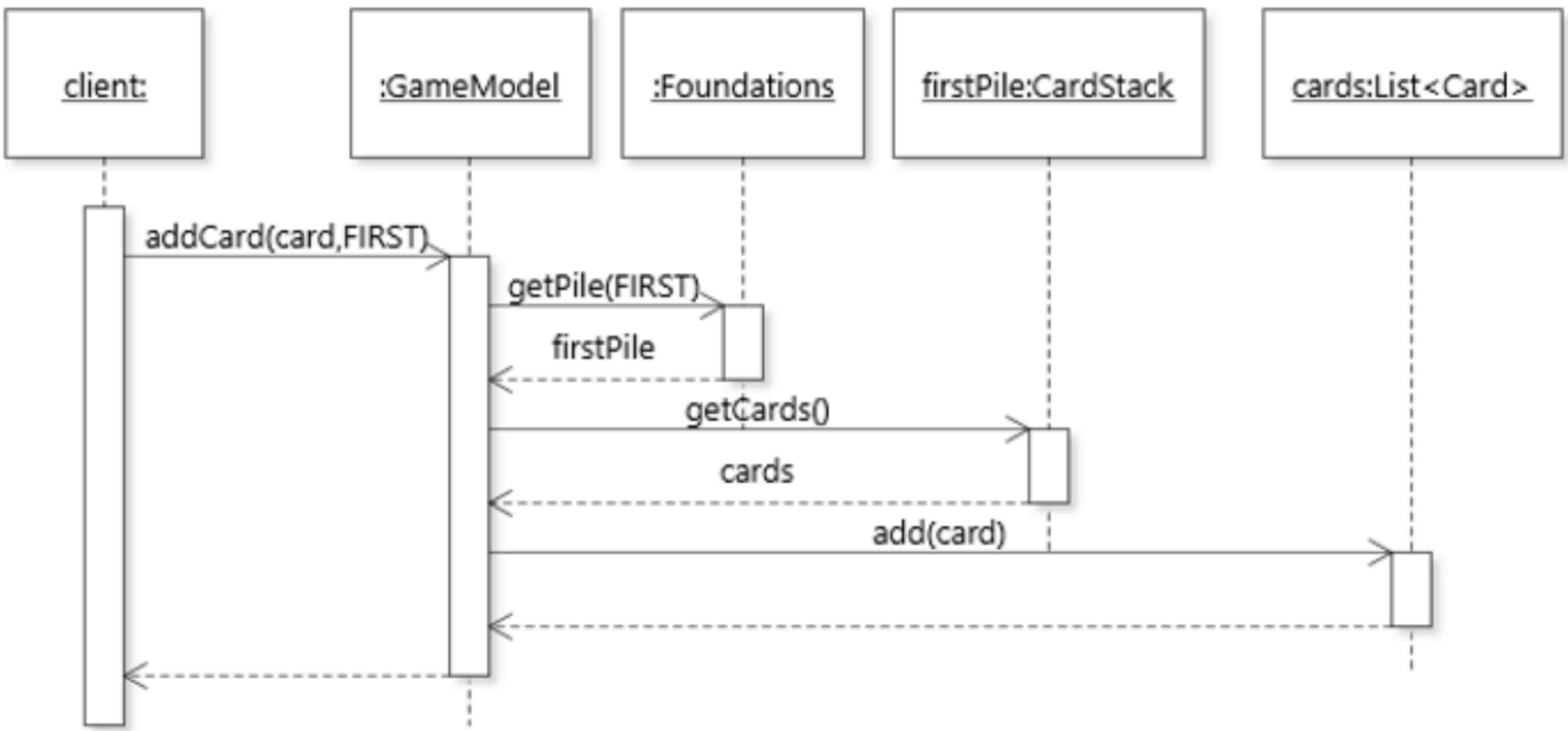
- Command, ChatCommand and MenuCommand.
- Design choices

The Law of Demeter

Delegation chains

Delegation chains

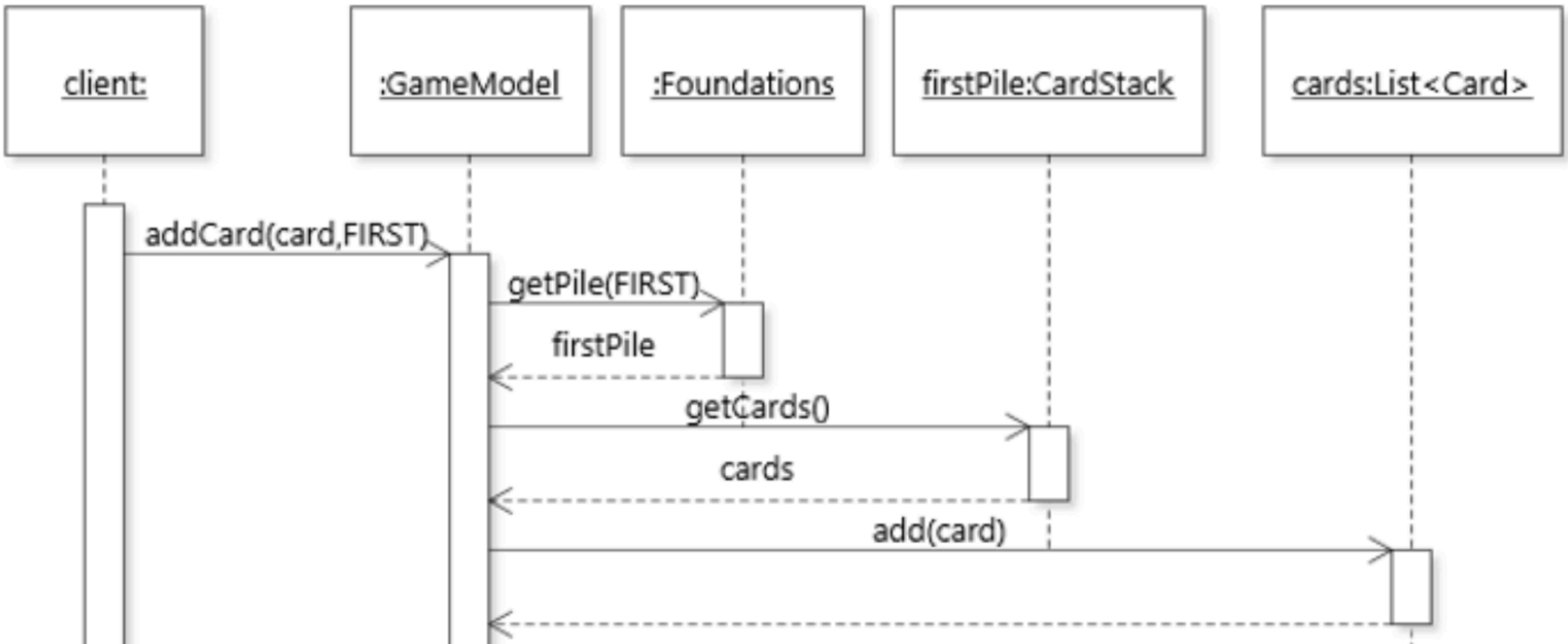
```
aFoundations.getPile(FIRST).getCards().add(pCard);
```



In this design, the GameModel is in charge of adding a Card to a specific List<Card>, even though that field is stored several layers deep.

Delegation chains

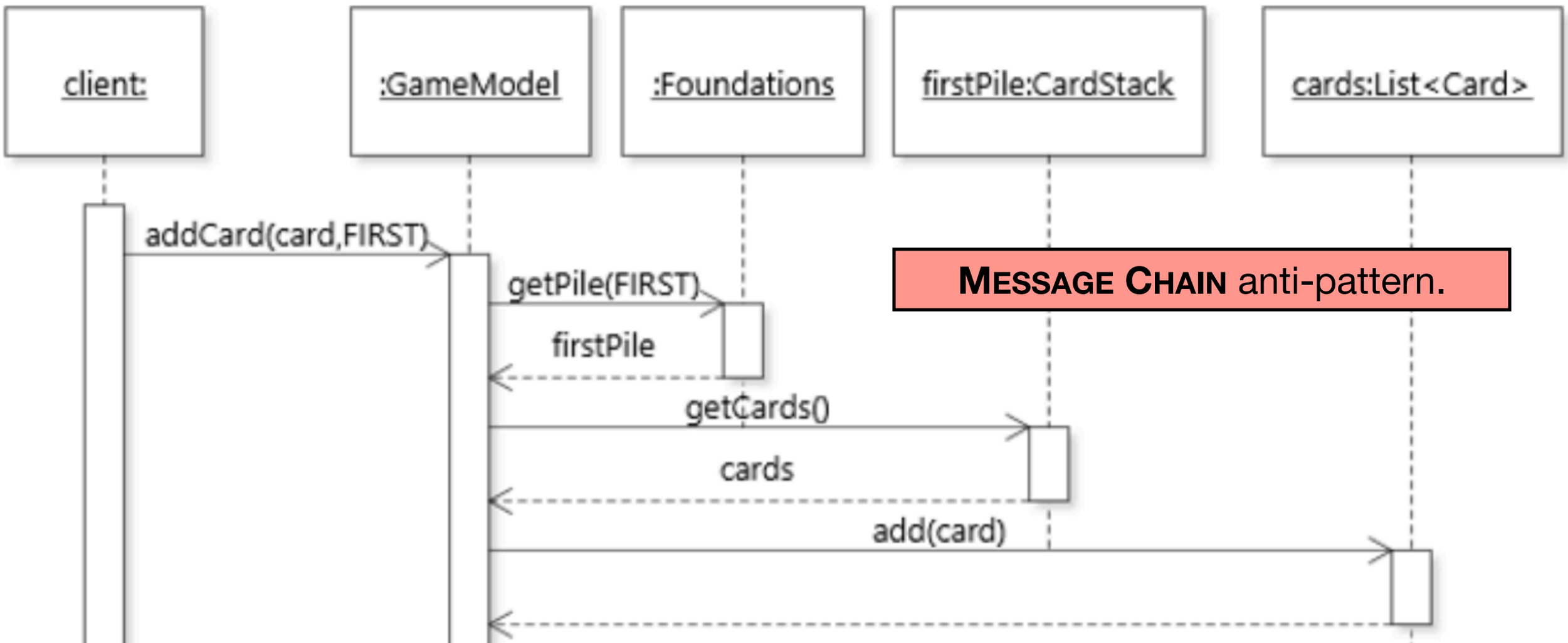
```
aFoundations.getPile(FIRST).getCards().add(pCard);
```



Violates the principle of information hiding:
the GameModel must know about how the Foundations object manages its pile,
and that that pile is a CardStack,
and that that CardStack contains a List<Card>.

Delegation chains

```
aFoundations.add(pCard);
```



MESSAGE CHAIN anti-pattern.

Violates the principle of information hiding:
the GameModel must know about how the Foundations object manages its pile,
and that that pile is a CardStack,
and that that CardStack contains a List<Card>.

Law of Demeter

- The following line:

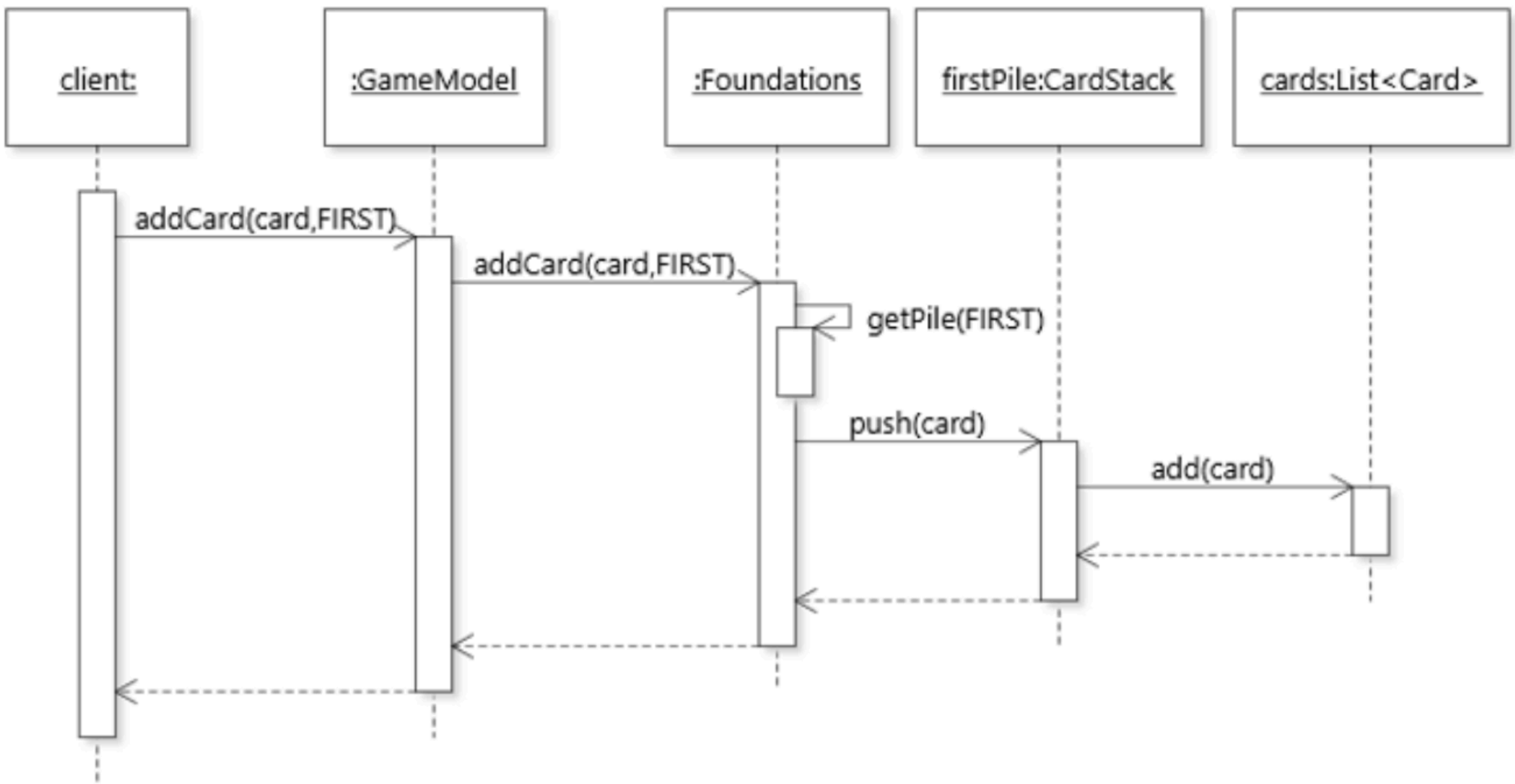
```
aFoundations.getPile(FIRST).getCards().add(pCard);
```

- calls the getCards method on an object returned by a method. Not good! Instead, we would prefer to write:

```
aFoundations.add(pCard, FIRST);
```

- after defining the appropriate method in the Foundations class. In general, to respect the Law and only use our own fields, we have to define additional methods.

Law of Demeter



Law of Demeter

- Code in a method should only access:
  - the instance variables of its implicit parameter (this);
  - the arguments passed to the method;
  - any new object(s) created within the method;
  - (if need be) globally available objects.

Unit testing

Testing & unit testing

- Testing: Check that code works properly.
- Unit testing: Write **little** tests, one per every behaviour / edge case ("unit") of a method.
- When we change the method later on, we can re-run all the tests that we've written, to make sure it still works.

Components of a unit test

- UUT: The **unit under test**. E.g., the method.
- Input data: The arguments to the method. Also, the implicit argument (this/self).
- Expected result ("oracle"): what the method should return.

Example: testing Math.abs

- UUT: Math.abs(int)
- Input data: 5
- Expected result ("oracle"): 5

Example: testing sameColorAs

```
public enum Suit {
  CLUBS, DIAMONDS, SPADES, HEARTS;

  public boolean sameColorAs(Suit pSuit) {
    assert pSuit != null;

    // if even, black suit; if odd, red suit.
    return (this.ordinal() - pSuit.ordinal()) % 2
== 0;
  }
}
```

Example: testing sameColorAs

```
public static void main(String[] args) {
  boolean oracle = false;

  // UUT: Suit.sameColorAs
  // Input: CLUBS (implicit arg.); HEARTS (explicit arg.)
  // Expected result: false

  System.out.println(oracle == CLUBS.sameColorAs(HEARTS));
}
```

Regression testing

- Re-running tests to make sure that was correct is still correct after some change was made to a method.
  - E.g., if we re-order the suits in the Suit enum, our test will fail. (Because sameColorAs relies on an undocumented and unchecked assumption about the order!)

Exhaustive testing

- Testing all possible combinations of arguments.
- For sameColorAs, we would try each combination of implicit and explicit Suit (4*4 = 16 combinations).
- Almost never possible. Unit testing cannot **verify** code to be completely correct. (For that, we need formal verification methods.)

JUnit

Unit testing frameworks

- Automatic software testing is typically done using a unit testing framework.
- These frameworks automate running the tests, reporting the results of tests, and have other nice things.
- In Java, the most popular such framework is **JUnit**. We will cover the basics of it.

Unit tests in JUnit

@Test: indicates that the annotated method should be run as a unit test

```
public class AbsTest {
  @Test
  public void testAbs_Positive() {
    assertEquals(5, Math.abs(5));
  }
  @Test
  public void testAbs_Negative() {
    assertEquals(5, Math.abs(-5));
  }
  @Test
  public void testAbs_Max() {
    assertEquals(Integer.MAX_VALUE,
                 Math.abs(Integer.MIN_VALUE));
  }
}
```

assert method: Will check the given argument(s), and report a failure if appropriate.

Unit tests in Unit

Finished after 0.094 seconds

Runs: 3/3 Errors: 0 Failures: 1

AbsTest [Runner: JUnit 5] (0.010 s)

- testAbs_Max (0.007 s)
- testAbs_Positive (0.000 s)
- testAbs_Negative (0.001 s)

Failure Trace

```
java.lang.AssertionError: expected:<2147483647> but was:<-2147483648>
at designbook/chapter5.AbsTest.testAbs_Max(AbsTest.java:36)
at java.base/java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:183)
```

References

- Robillard ch. 6.9 (p.153-156)
  - Exercises #17, 19: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter6.md
- Robillard ch. 5-5.2 (p. 99-104)

Coming up

- Next lecture:
  - More about unit testing