McGill

COMP 303

Lecture 15

Inheritance II

Winter 2025

slides by Jonathan Campbell

Announcements

- Midterm on Wednesday, March 12
- Future project milestone guidelines posted.
- Project: event loop

Today

- Inheritance
    - Overloading methods
    - Abstract classes
- Review

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;
  }
}
```

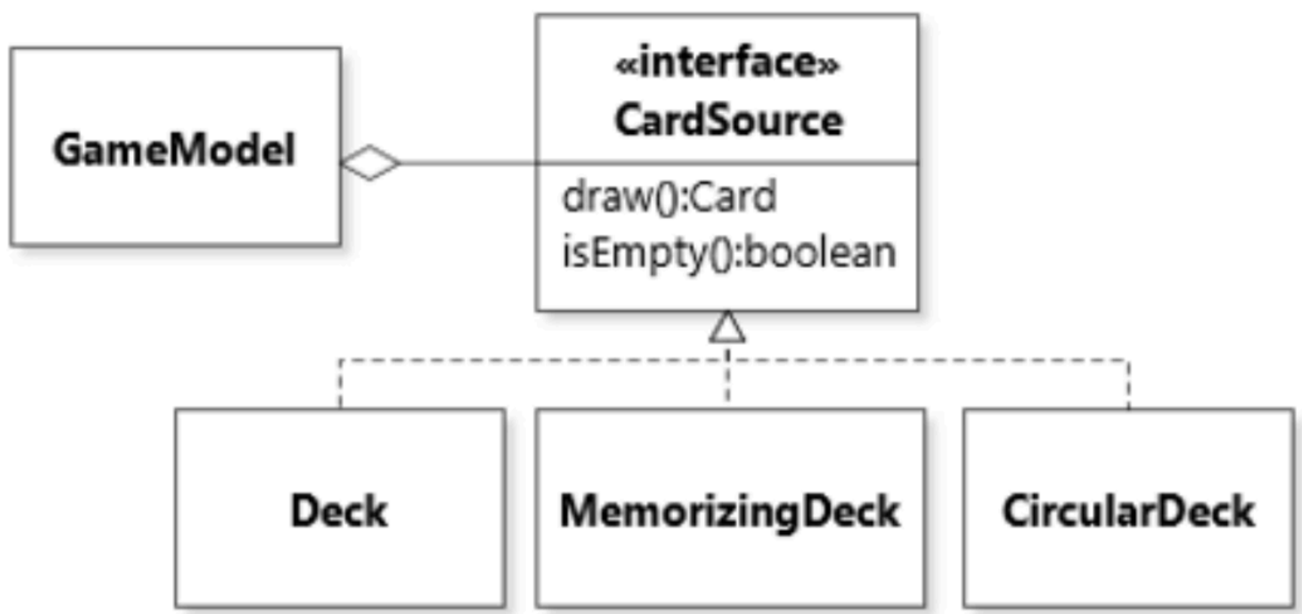
We can see that the code is split into an if/else.
It would be good to test both parts.

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

Recap

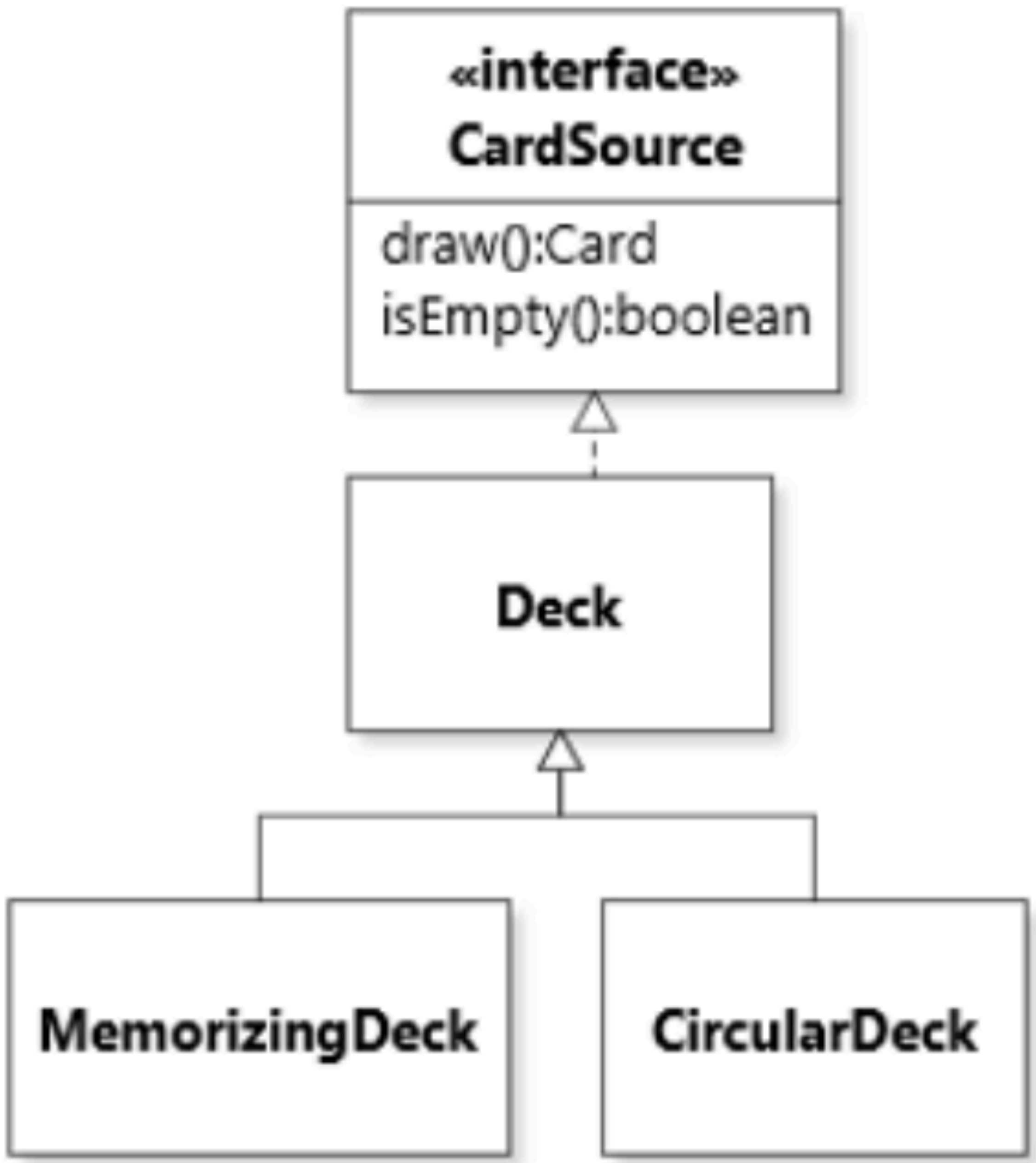
Polymorphism

- Recall: by using polymorphism to decouple client code from concrete types, we can make a design more extendable.



- Here, the GameModel only depends on the CardSource interface, so it can use any implementing type.

Inheritance



Run-time vs compile-time types

- When an object is created in Java, it has a **runtime type**. This type is the class name after `new`, and never changes.

- A compile-time (or static) type, by contrast, is associated with a variable, not an object. An object of a particular runtime type can be referred to by a variable of the same type, or any supertype.

    - Thus, the same object, with its never-changing runtime type, can be referred to by variables of different compile-time types.

Run-time vs compile-time types

```
public static boolean isMemorizing(Deck pDeck) {
  return pDeck instanceof MemorizingDeck;
}
public static void main(String[] args) {
  Deck deck = new MemorizingDeck();
  MemorizingDeck memorizingDeck = (MemorizingDeck) deck;
  boolean isMemorizing1 = isMemorizing(deck); // true
  boolean isMemorizing2 = isMemorizing(memorizingDeck); // true
}
```

The MemorizingDeck created at the start of main has one runtime type, but several compile-time types.

Downcasting

- We can only call methods on an object that are applicable for its static type. The following will thus result in an error:

```
Deck deck = new MemorizingDeck();
deck.getDrawnCards();
```

- But it works out if we downcast:

```
MemorizingDeck memorizingDeck = (MemorizingDeck) deck;
Iterator<Card> drawnCards = memorizingDeck.getDrawnCards();
```

Inheriting fields

```
public class Deck implements CardSource {
  private final CardStack aCards = new CardStack();
  ...
}

public class MemorizingDeck extends Deck {
  private final CardStack aDrawnCards = new CardStack();
  ...
}
```

Initializing fields

```
public class MemorizingDeck extends Deck {
  private final CardStack aDrawnCards = new CardStack();
  public MemorizingDeck(Card[] pCards) {
    /* Automatically calls super() */
    ...
  }
}
```

This is done by having each constructor call its superclass constructor before doing anything else. (Because constructors are called bottom-up.)

But, in this case, we must call super() ourselves, to pass the pCards.

Initializing fields

```
public class Deck {
  private final CardStack aCards = new CardStack();
  public Deck() {} // Relies on the field initialization
  public Deck(Card[] pCards) {
    for (Card card : pCards) {
      aCards.push(card);
    }
  }
}
public class MemorizingDeck extends Deck {
  private final CardStack aDrawnCards = new CardStack();
  public MemorizingDeck(Card[] pCards) {
    super(pCards);
  }
}
```

Overriding methods

```
public class MemorizingDeck extends Deck {
  public Card draw() {
    Card card = super.draw();
    aDrawCards.push(card);
    return card;
  }
}
```

super() will go up the class hierarchy, and find the first implementation of the same method.
(Which could itself override something in a further parent class.)

Inheritance II

Overriding methods

- Overriding a method lets the subclass specify a different implementation, and Java will choose to run the implementation most specific for the run-time type of the implicit parameter.
    - So if draw() is called on an object of runtime type MemorizingDeck, and MemorizingDeck extends Deck, and both implement draw(), the MemorizingDeck draw() will be called.
    - But if draw() is called on an object of runtime type Deck, or on runtime type CircularDeck, and CircularDeck has no draw(), then the regular Deck draw() will be called.

Overloading methods

- Overloading is when multiple versions of a method are specified, **each with a different signature** (i.e., different explicit parameters).
    - Note: A method's signature only comprises its name and parameter types, not return type.

Overloading methods

```
public class MemorizingDeck extends Deck {
  private CardStack aDrawnCards = new CardStack();
  public MemorizingDeck() {
    /* c1: Does nothing besides initialization */
  }
  public MemorizingDeck(CardSource pSource) {
    /* c2: Copies all cards of pSource into this object */
  }
  public MemorizingDeck(MemorizingDeck pSource) {
    /* c3: Copies all cards and drawn cards of pSource
     * into this object */
  }
}
```

Overloading methods

- Java will select which method to run based on the **number and <u>static (i.e., compile-time)</u> types of the parameters** given to it.
    - It will choose the **most specific** out of the **compatible** options.

Overloading methods

```
public class MemorizingDeck extends Deck {
  private CardStack aDrawnCards = new CardStack();
  public MemorizingDeck() {
    /* c1: Does nothing besides initialization */
  }
  public MemorizingDeck(CardSource pSource) {
    /* c2: Copies all cards of pSource into this object */
  }
  public MemorizingDeck(MemorizingDeck pSource) {
    /* c3: Copies all cards and drawn cards of pSource
     * into this object */
  }
}
```

Overloading methods

```
// calls empty constructor
MemorizingDeck memorizingDeck = new MemorizingDeck();

// Both c2 and c3 are applicable,
// but c3 is more specific.
Deck newDeck1 = new MemorizingDeck(memorizingDeck);

Deck deck = memorizingDeck;
Deck newDeck2 = new MemorizingDeck(deck);
// static type of deck is Deck.
// so the only compatible option is c2.
// c3 cannot be used because we are looking at static
// (compile-time) type of the parameter, not runtime type.
```

Overloading methods

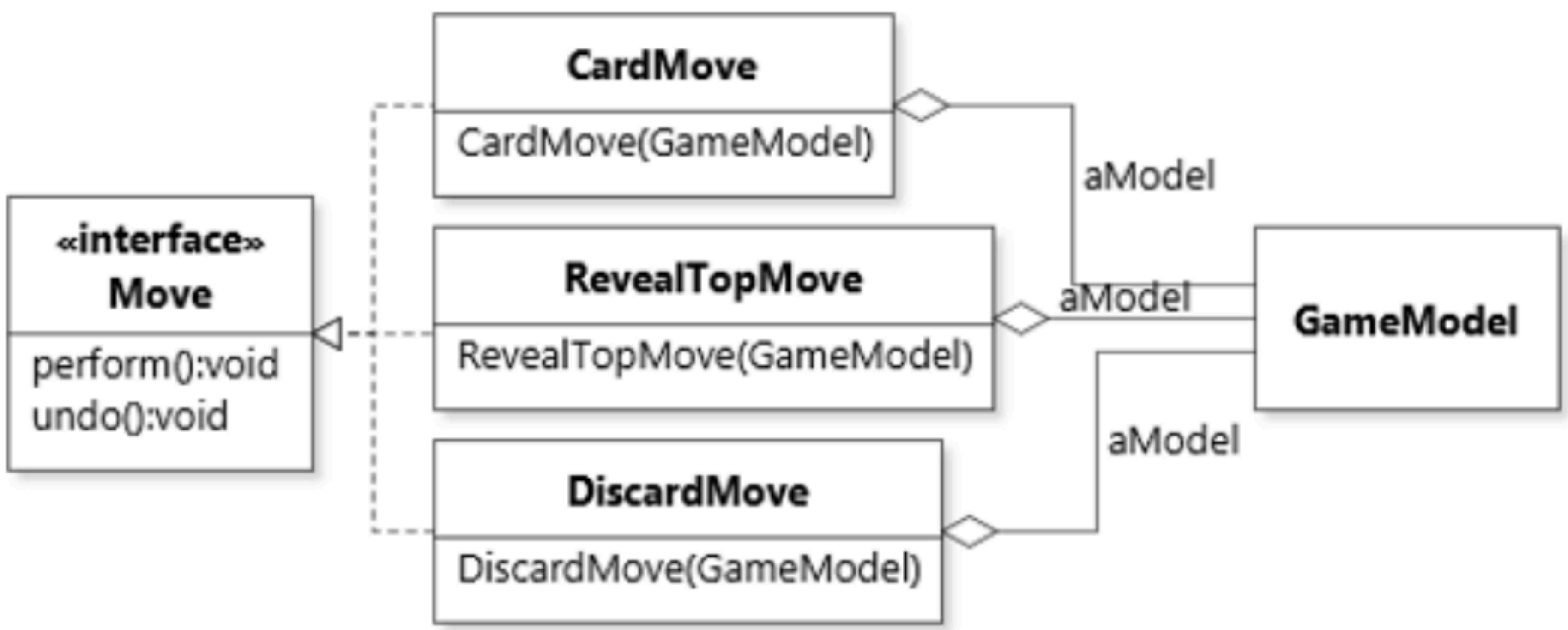
- Overloading methods can quickly result in hard-to-understand code.
- We try to avoid overloading methods except for popular use cases (e.g., overloading constructors, or library methods that are meant to support multiple primitive types, like math.abs).

Abstract classes

- Suppose we apply the Command pattern to game moves, creating an interface as follows:

```
public interface Move {
  void perform();
  void undo();
}
```

Abstract classes

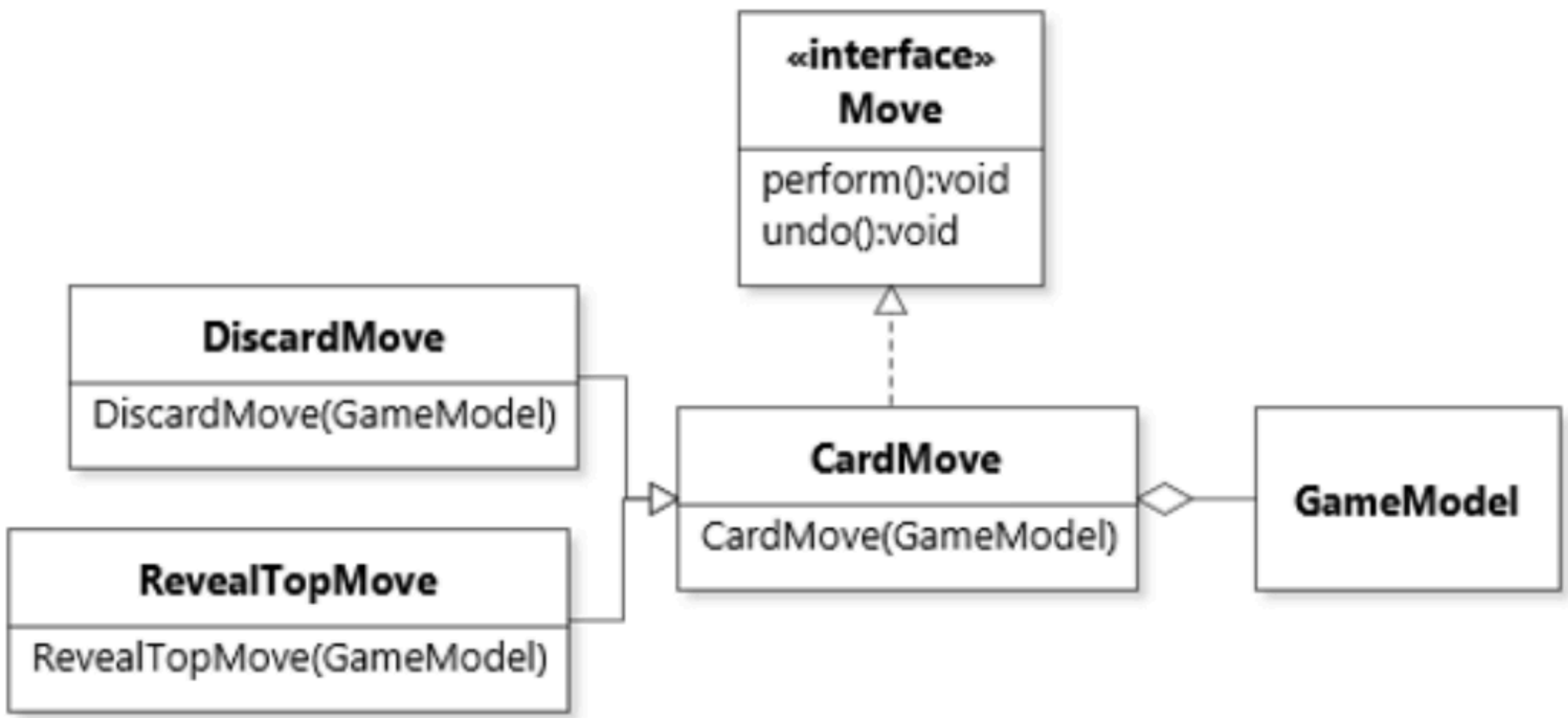


Abstract classes

- In this situation, each concrete Move class stores an instance of GameModel, so that perform/undo can take effect on it.

- To avoid defining a field for the GameModel in each Move class (duplicated code), we'd like to use inheritance.
  - We thus need to decide what will be the parent and what will be the child classes.

Abstract classes



Here we've arbitrarily selected one of the commands (CardMove) and made it the parent class.

Not a good design.

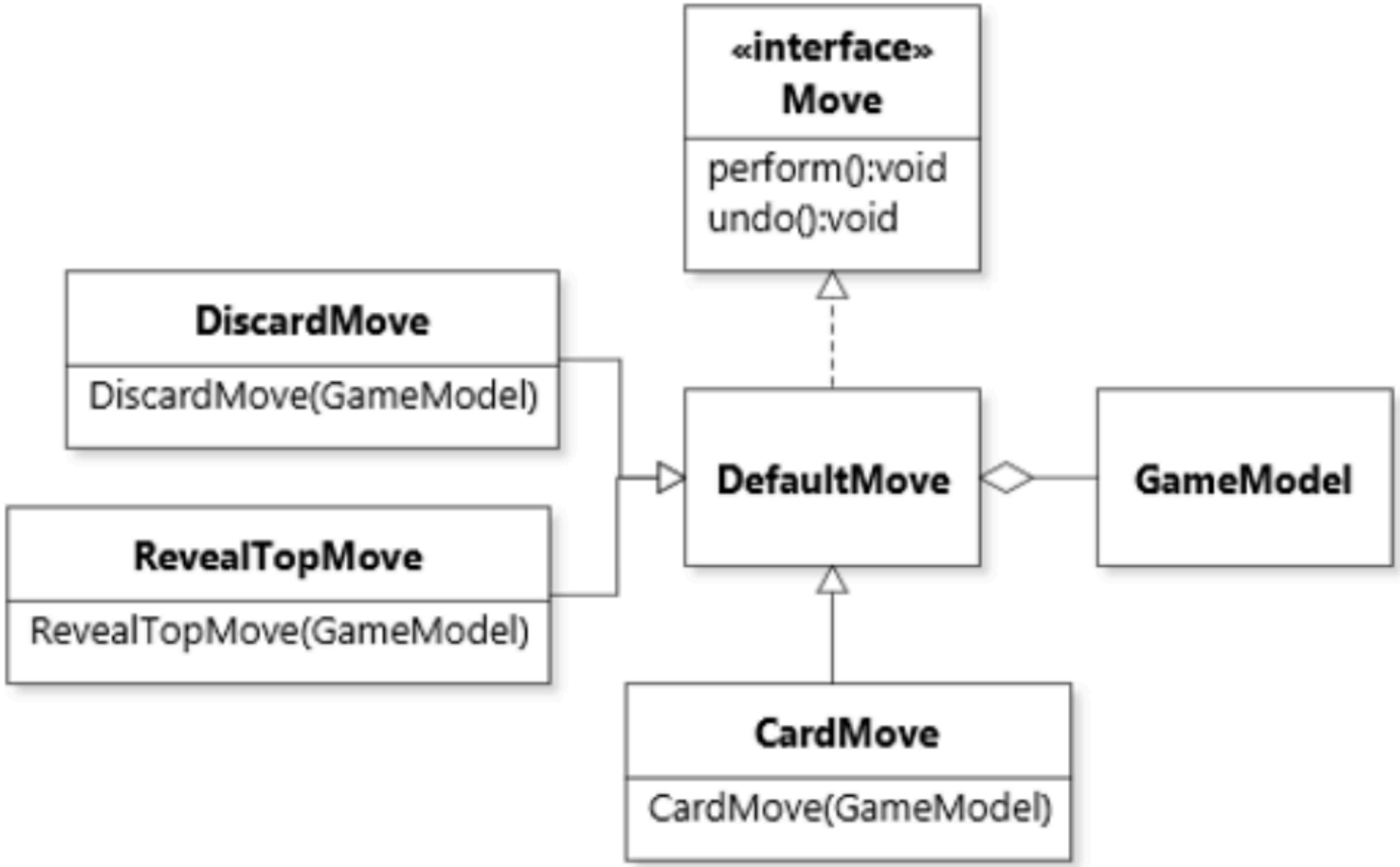
Abstract classes

- A subclass should be a **natural subtype** of the base class, extending its behaviour in some way.
- But a DiscardMove doesn't extend the behaviour of a CardMove. They are two completely different moves.
  - DiscardMove would inherit any non-interface methods of CardMove, even though it wouldn't need to use them.
  - Also, if we forget to implement perform or undo in DiscardMove, it will default to the CardMove behaviour, which would be totally wrong yet not throw a compiler error.

Abstract classes

- Here, we want to create an entirely new base class, which all Moves will inherit from.
- We don't want to specify any behaviour for perform/undo in this base class. In fact, we don't want to be able to instantiate this class at all on its own.
- Instead, we will make it **abstract**.

Abstract classes



Abstract classes

```
public abstract class AbstractMove implements Move {
  private final GameModel aModel;
  protected AbstractMove(GameModel pModel) {
    aModel = pModel;
  }
}
```

Abstract classes

```
public class CardMove extends AbstractMove {
  public CardMove(GameModel pModel) {
    super(pModel);
  }

  void perform() { ... }
  void undo() { ... }
}
```

Abstract classes

- An abstract class cannot be instantiated directly. Its constructor can only be called from subclass constructors.

- It does not need to specify an implementation for all methods. But any concrete (i.e., non-abstract) class inheriting from it will need to do so.

- We can also declare new abstract methods in an stract class (more on this later).

Abstract classes vs. interfaces

- Interfaces cannot have instance variables. They can only have constants. (Abstract classes can.)
- In Java, a class can extend only one class (abstract or otherwise), but can implement many interfaces.
- Abstract classes are best used when there is a clear hierarchical relationship. Interfaces are for defining a contract for a narrow slice of behaviour across different classes.

References

- Robillard ch. 7.5, 7.8 (p. 170-171, 177-180)
    - Exercise #3: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter7.md

Review

General overview of topics

- Chapters 1-6 of textbook

Design principles

- Encapsulation (2.1, 2.5, 2.7)
  - Information hiding (2.1)
  - Abstractions (2.2)
  - Immutability (2.6)
  - Input validation (2.8)
  - Design by contract (2.9)

Design principles

- Types and interfaces (3.1, 3.2)
  - Coupling/decoupling behaviour (3.1)
  - Polymorphism (3.1)
  - Subtyping (3.1)
  - Separation of concerns (3.2)
  - Interface segregation principle (3.9)

Design principles

- Object state (4.2)
  - Abstract vs concrete state and state space (4.2)
  - Life cycle (4.4)
  - Nullability (4.5)
  - Equality, identity, uniqueness (4.7)

Design principles

- Composition (6.1)
  - Aggregation / delegation (6.1)
  - Polymorphic copying (6.6)
  - Law of Demeter (6.9)

Design principles

- Unit testing (5.1, 5.5, 5.6, 5.7)
  - JUnit (5.2, 5.5, 5.6)
  - Metaprogramming (5.4)
  - Stubs (5.8)
  - Test coverage metrics (5.9)

Design patterns

- Iterator (3.5, 3.6)
- Strategy (3.7)
- Null object (4.5)
- Flyweight (4.8)
- Singleton (4.9)
- Composite (6.2)
- Decorator (6.4, 6.5)
- Prototype (6.7)
- Command (6.8)

Anti-patterns

- Primitive obsession (2.2)
- Inappropriate intimacy (2.5)
- Switch statement (3.4)
- Speculative generality (4.4)
- Temporary field (4.4)
- Long method (4.6)
- God class (6.1)
- Message chain (6.9)

Diagrams

- Object diagram (2.4)
- Class diagram (3.3)
- State diagram (4.3)
- Sequence diagram (6.3)

Language features

- Enums (2.2)
- Scope & access modifiers (2.3)
- Assertions (2.9)
- Function objects (3.4)
- Optional<T> type (4.5)
- Final fields (4.6)
- Class<T> type (5.4)

Java-specific questions

- Optional<T>, Null object pattern
- Final fields
- JUnit, Class<T>

Studying

- Prepare crib sheet (one double-sided 8.5x11 page)
- Textbook exercises
- Practice midterm
    - https://github.com/prmr/COMP303/blob/2019F/Sample-Midterm.pdf

Coming up

- Next lecture:
    - More about inheritance