# Software Design

The construction of abstractions (variables, classes, objects…) of data and computation and the organization of these abstractions into a working software application.

Design attributes of code: understandability, readability, reusability, ease of implementation

# Refactoring

Improving the design of code without changing its functionality:

* original developer did not really get it right
* might want to add modules and features that do not integrate well with the existing design
* reduce accumulated design weaknesses

# General

Implicit Parameter: object upon which a method is called

Deck deck = new Deck();  
 deck.drawCard(); => deck is the implicit parameter

Explicit Parameter: object which is passed as an argument

shuffle(memorizingDeck) => memorizinDeck is the explicit parameter explicit

Run vs Compile time objects:

Deck deck = new MemorizingDeck();  
 compile-time: deck instance of Deck  
 run-time: deck instance of MemorizingDeck  
 run-time specification is the one with the “new” statement

Run-time type: never changes during an object’s lifecycle

Compile-time type: changes, is a reference to the type in which the object is stored

* static member can be accessed without creating an instance of the class, will be shared by all instances of the class and can be accessed without creating an instance of the class
  + class-wide utility methods
  + constant variables
  + singleton
  + counters
  + factory methods

# Encapsulation

Idea: enclose something as if it were in a capsule to limit the number of contact points between different parts of the code

Benefits: easier to understand, less error prone, easier to change

Object is a mechanism to group variables together and access their values through the process of dereferencing. Without encapsulation any variable can be accessed indiscriminately.

## Information hiding

Only reveal the minimum amount of information that is necessary to use them and hide the rest.

## Enums

Enumerated types are globally available constants. Avoid Primitive Obsession, cleaner code and less error prone.

## Abstraction

## Immutability

Goal: it is not possible to change the internal state from a reference.

Objects are immutable if their class provides no way to change the internal state of the object after initialization.

* Support sharing information without breaking encapsulation

Example: classes that are only setting private variables through the constructor and not mutating them.

### Copy Lists

Immutable List Element: When the element in the list is immutable, it is possible to return a copy of the list which makes the whole element immutable and there are no leaking references.

Mutable List Element: If the underlying list element is mutable, then every element from the list has to be copied as well, not only the list itself. This can be done through a new constructor within the copy method.

public List getCards() {   
ArrayList result = new ArrayList<>();   
for( Card card : aCards ) { result.add(new Card(card)); }   
return result; }

## Design by contract

Takes a set of preconditions and postconditions.

* lightweight version of design by contract where Javadoc @pre and @post tag specify them
* use assert statements at the beginning to check for the pre conditions

/\*\*   
\* @pre pRank != null && pSuit != null   
\*/   
public Card(Rank pRank, Suit pSuit) {

assert pRank != null && pSuit != null;   
aRank = pRank; aSuit = pSuit; }

## Escaping references

Following implementations are not good as the reference can escape the scope of the class and be altered in a different class.

**Returning the reference:**

private List aCards = new ArrayList<>();

public List getCards() { return aCards; }

**Reference to an internal object:**

main(String args[]){

deck = new Deck()

cards = new ArrayList<Cards>

}

**Overwriting the reference / Storing a reference internally:**

private List aCards = new ArrayList<>();

public void setCards(List<Card> cards) { aCards = cards; }

**Leaking reference through shared structure:**

private List aCards = new ArrayList<>();   
public void collect(List<List> pAllCards) { pAllCards.add(aCards); }

List<List> allCards = new ArrayList<>();   
Deck deck = new Deck();   
deck.collect(allCards);   
List cards = allCards.get(0);   
cards.add(new Card(Rank.ACE, Suit.HEARTS));

# Types and Interfaces

## Interface

Equivalent of a shell of the code.

* Interface Class Definition: An interface to a class consists of the methods of that class that are accessible (or visible) to another class.
  + All public methods from a class
* Interface Object Definition: The interface to an object is the set of methods that can be called on the object
  + Implements an interface and receives new methods
  + provide a specification of the methods that it should be possible to invoke on the objects of a class
  + abstract methods

“implements”

* guarantees that a class has the concrete implementations for a method
* creates a subtype relationship => e.g. Car is a type of Vehicle

## Polymorphism

Use polymorphism to make code more reusable. The implementation below shows the use of polymorphism so that a Deck is an instance of CardSource (subtype) and each CardSource needs the according method implementations.

public static List drawCards(Deck pDeck, int pNumber) {   
 List result = new ArrayList<>();   
 for( int i = 0; i < pNumber && !pDeck.isEmpty(); i++ ) {

result.add(pDeck.draw()); }   
return result; }

public class Deck implements CardSource  
CardSource source = new Deck();  
public static List drawCards(CardSource pSource, int pNumber) {   
 List result = new ArrayList<>();   
 for( int i = 0; i < pNumber && !pSource.isEmpty(); i++ ) {   
 result.add(pSource.draw());   
 }   
return result; }

* **Loose coupling**, because the code using a set of methods is not tied to a specific implementation of these methods
* **Extensibility**, because we can easily add new implementations of an interface (new “shapes” in the polymorphic relation).

## Function Objects

Objects whose interfaces map 1:1 that of an interface (e.g. Comparator). Functional interfaces.

Design trade-offs:

* Definition as standalone java class: compare methods will not have access to private attributes
* Definition as nested class: if access to private attributes of the class is required and the method has to be called several times

public class Card {   
 static class CompareBySuitFirst implements Comparator {   
 public int compare(Card pCard1, Card pCard2) { /\* Comparison code \*/ }   
 }   
}

* Definition as anonymous class: if the object is only called once => problem: encapsulation of the code, comparison is done outside of the class Card

public class Deck {   
 public void sort() {   
 Collections.sort(aCards, new Comparator() {   
 public int compare(Card pCard1, Card pCard2) { /\* Comparison code \*/ }   
 });   
 }  
}

* Definition as lambda expression: if the object is only called once => problem: encapsulation of the code, comparison is done outside of the class Card

public class Deck {   
 public void sort() {   
 Collections.sort(aCards, (card1, card2) ->   
 card1.getRank().compareTo(card2.getRank()));   
 }  
}

* Factory method to not break encapsulation

public class Card {   
 public static Comparator createByRankComparator() {   
 return new Comparator() {   
 public int compare(Card pCard1, Card pCard2) {   
 /\* Comparison code \*/   
 }   
 };   
 }   
}

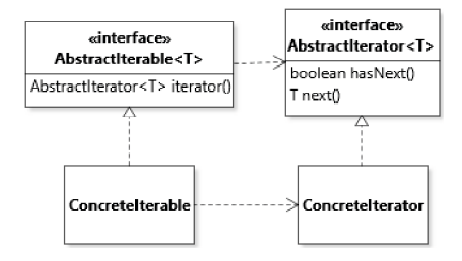
## Interface segregation ISP

* Client code should not be forced to depend on interfaces it does not need: break up large interfaces into smaller ones if you find that many methods of a type are not used in certain code locations

## Patterns

### Iterator

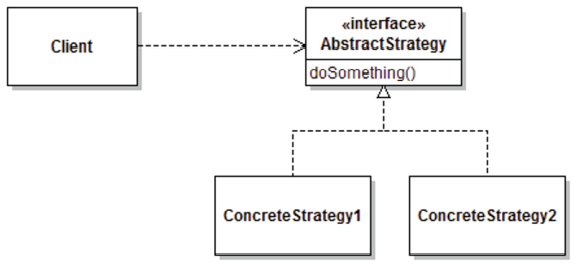
* Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
* access to a collection of objects encapsulated by another object without violating encapsulation and information hiding.
* Behavioral pattern



public class Deck implements Iterable<Card> {   
 private List<Card> aCards;   
 public Iterator<Card> iterator() { return aCards.iterator(); }   
}

### Strategy

* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it
* basic use of polymorphism
* relies on polymorphism to allow client code to dispatch the execution of an algorithm to a dynamically selected variant
* pulling out behaviour (e.g. flying and quacking) out of a (duck) class to add the behaviour as interface => behaviour get their own set of classes (flying classes, quacking classes)
* Behavioral pattern



public interface Strategy {public int doOperation(int num1, int num2);}

public class OperationAdd implements Strategy{  
 @Override   
 public int doOperation(int num1, int num2) { return num1 + num2; }   
}

public class OperationSubtract implements Strategy{  
 @Override   
 public int doOperation(int num1, int num2) { return num1 - num2; }   
}

public class Context {   
private Strategy strategy;   
   
 public Context(Strategy strategy){this.strategy = strategy; }   
  
 public int executeStrategy(int num1, int num2){   
 return strategy.doOperation(num1, num2);   
 }   
}

public class StrategyPatternDemo {   
 public static void main(String[] args) {   
 Context context = new Context(new OperationAdd());   
 System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());   
 System.out.println("10 - 5 = " + context.executeStrategy(10, 5));   
 }   
}

## Sorting

### Comparable

int compareTo(T): compares the implicit parameter (this)  
 return 0 if argument is the same, return -1 if object should come before, return 1 if the object should come after

public class Card implements Comparable {   
 public int compareTo(Card pCard) {   
 return this.aRank.compareTo(pCard.aRank);   
 }   
}

### Comparator

int compare(T pObject1, T pObject2): compares two explicit parameters with each other (object1, 2)

public class RankFirstComparator implements Comparator {   
 public int compare(Card pCard1, Card pCard2) { /\* Comparison code \*/ }   
}

public class SuitFirstComparator implements Comparator {   
 public int compare(Card pCard1, Card pCard2) { /\* Comparison code \*/ }   
}

Collections.sort(aCards, new RankFirstComparator());

# Concrete and abstract objects

## State objects

* State space: set of possible states for a variable or object

public class Player { private int score = 0; }  
> 232 possible states for this object  
> with every new field in a class the state space increases exponentially due to the permutations

* Concrete state: collection of values stored in the object’s field
* Abstract state: arbitrarily defined subset of the concrete state space

States like “EMPTY” or “THREE\_KINGS” in a deck

* Stateless objects: function objects often do not have any associated state
* Stateful objects: objects which have an associated state

The state space applies to mutable object. Immutable objects in practice only have a single state.

## Optional types

Problem: adding a Joker to a deck of cards (without a Rank and Suit)

Solutions:

* Null references: assign the jokers getRank and getCard methods to null => NullpointerException
* Bogus values: assign a value to the Joker, e.g. AceOfClubs => Makes context wise no sense
* Special enum values: add new enums e.g. INAPPLICABLE to both Rank and Suit
* Optional<T>: Generic type that acts as a wrapper for an instance of type T, which can be empty

public class Card {

private Optional aRank;

private Optional aSuit;

public Card() { aRank = Optional.empty(); aSuit = Optional.empty(); }

public boolean isJoker() { return !aRank.isPresent(); }

}

### Value retrieval

* Optional.of(value) : if the value is never null
* Optional.ofNullable(value) : value can be null
* Alter getter and setter to work with optionals or unwrap the values within the getter and setter
  + Interface change

public Optional<Suit> getSuit(){ return aSuit; }

* + Unwrap the values

public Suit getSuit(){ return aSuit.get() }

## Final keyword

* Limiting the state space of an object
* Immutable values
* Assignation only once, in the initializer or the constructor
* Attention: references to an object are final, if the object is mutable, it still can be altered

## Object Identity, Equality, Uniqueness

* Identity: refers to a particular object (reference / pointer to) within the memory space
* Equality: programmer-defined => override the default equals method (equality = identity)

@Override

public boolean equals(Object pObject) {

if( pObject == null ) { return false; } // As required by the specification

else if( pObject == this ) { return true; } // Standard optimization

else if( pObject.getClass() != getClass()) { return false; }

else { // Actual comparison code

return aRank == ((Card)pObject).aRank && ((Card)pObject).aSuit == aSuit;

}

}

@Override

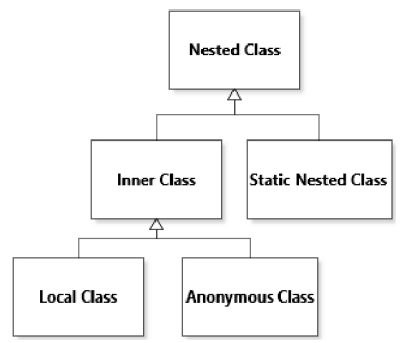
public int hashCode() {

return Objects.hash(aRank, aSuit);

}

* Uniqueness: If the objects of a class can be guaranteed to be unique, then we no longer need to define equality

## Nested classes



### Inner class

* declared within another class
* provide additional behavior that involves an instance of the enclosing class
* Instances of an inner class automatically get a reference to the corresponding instance of their enclosing class (called the outer instance).
* implicit access to additional state information through a reference to their outer instance
* Create a method in the outer class to access the inner class
* Calling the method from the outer class with additional behaviour from the inner class
* declared within a method

public class Deck {   
 public void shuffle() { ... }   
 public Shuffler newShuffler() { return new Shuffler(); }   
 public class Shuffler {   
 private int aNumberOfShuffles = 0;   
 private Shuffler() {}   
 public void shuffle() { aNumberOfShuffles++; Deck.this.shuffle(); }   
 public int getNumberOfShuffles() { return aNumberOfShuffles; }   
 }   
}

Deck deck = new Deck();   
Shuffler shuffler = deck.newShuffler();   
shuffler.shuffle();

#### Local class

* implicit access to additional state information through a reference to their outer instance
* rarely used
* work similar to anonymous classes

#### Anonymous class

* implicit access to additional state information through a reference to their outer instance
* useful when the code should only be executed once
* declared when an instance is created

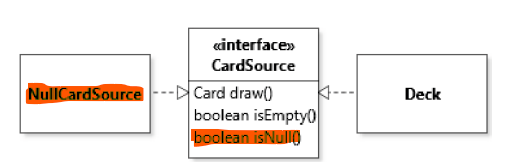
public class Deck(){  
 public static Comparator<Deck> createByRankComparator(Rank pRank){  
 return new Comparator<Deck>() {  
 public int compare(Deck pDeck1, Deck pDeck2) {  
 return countCards(pDeck1) - countCards(pDeck2);  
 }  
 private int countCards(Deck pDeck) {   
 int result = 0;  
 for( Card card : pDeck.aCards ) {  
 if( card.getRank() == pRank ){  
 result++;  
 }  
 }  
 return result;  
 }  
 };  
 }  
}

Comparator comp = Deck.createByRankComparator(Rank.KING);   
boolean result = comp.compare(deck1,deck2) < 1;

## Patterns

### Null Object

* avoiding the use of null references (using optionals or this pattern)
* Using a null object to represent a null value relies on polymorphism, so it is only applicable to situations where a type hierarchy is available
* create one special object to represent the absent value



Anonymous class of the null object pattern within the interface:

public interface CardSource {   
 public static CardSource NULL = new CardSource() { //basically a static class  
 public boolean isEmpty() { return true; }   
 public Card draw() { assert !isEmpty(); return null; }   
 public boolean isNull() { return true; }   
 };   
 Card draw();   
 boolean isEmpty();   
 default boolean isNull() { return false; }   
}

### Flyweight

* structural pattern
* unique instances of a class
* immutable
* ensure uniqueness of objects of a class
* used when instances of a class are heavily shared
* control the creation through an access method
  + private constructor
  + static flyweight store to keep the instances
  + static access method (check if object already exists & creates or returns the objects)

public class Card {

private Card(Rank pRank, Suit pSuit ) { aRank = pRank; aSuit = pSuit; }  
 // flyweight store  
 private static final Card[][] CARDS = new Card[Suit.values().length][Rank.values().length];

//pre initialized store  
 static {   
 for( Suit suit : Suit.values() ) {   
 for( Rank rank : Rank.values() ) {   
 CARDS[suit.ordinal()][rank.ordinal()] = new Card(rank, suit);   
 }   
 }   
 }

public static Card get(Rank pRank, Suit pSuit) {   
 if( CARDS[pSuit.ordinal()][pRank.ordinal()] == null ) {   
 CARDS[pSuit.ordinal()][pRank.ordinal()] = new Card(pRank, pSuit);   
 }   
 return CARDS[pSuit.ordinal()][pRank.ordinal()];   
 }  
}

* getter Method cannot take itself as an object to get an object (else we would be in an endless loop) => it needs some identification for the object (e.g. suit and rank; name of a person…)
* lazy loading | pre initialize set of variables

### Singleton

* creational pattern
* single instance of a class
  + Single deck of cards within the whole game => singleton
  + Unique card instances within the deck => flyweight
* Mutable
* stateful
* ensure that there is only one instance of a given class at any point in the execution of the code
* global access
* reduced overhead by limiting the number of objects created
* instance that represents the aggregated state of the game
  + private constructor
  + global variable
  + accessor method (usually called instance())

public class GameModel {  
 private static final GameModel INSTANCE = new GameModel();  
 private GameModel() {}  
 public static GameModel instance() { return INSTANCE; }  
}

# Testing

* Functional (or black-box) testing: cover as much specified behavior as possible
* Structural (or white-box) testing: cover as much of the implemented behavior as possible

## Unit Tests

* test a small part of the code in isolation
* UUT: unit under test
* Input => comparion => expected behavior
* exhaustive testing: testing all possible combinations
* uses metaprogramming to execute tests
* coverage: what is for sure missing in your tests
* does not tell you how good the code is

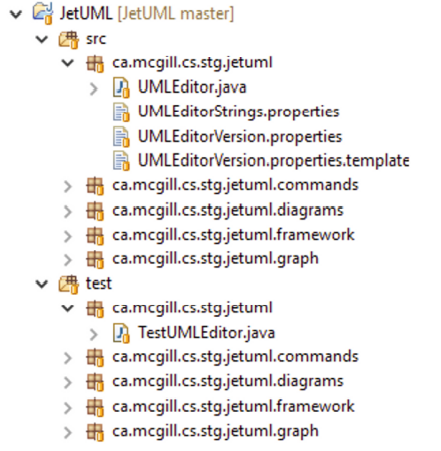
Statement coverage: percentage of statements in the code that are executed at least once.

Branch coverage: percentage of branches in the control flow of the code (e.g. if and switch statements) that are taken at least once.

Function coverage: percentage of functions or methods in the code that are executed at least once.

Path coverage: percentage of unique paths through the code that are executed at least once.

Condition coverage: percentage of conditions within the control flow of the code that are evaluated to both true and false



## Regression Tests

* Running tests to ensure what was tested as correct still is
* Continuous tests over a period of time

## Metaprogramming / reflection

* write code that operates on a representation of a program’s code
* change the accessibility of class members
  + efficient when flyweight or singleton patterns are used

### Introspection

* obtain a reference to an object that represents a piece of code to learn about it

try {   
 String fullyQualifiedName = "cards.Card";   
 Class cardClass = (Class) Class.forName(fullyQualifiedName);   
} catch(ClassNotFoundException e) { e.printStackTrace(); }

Class cardClass1 = Card.class;  
Card card = Card.get(Rank.ACE, Suit.CLUBS);   
Class cardClass2 = card.getClass();   
System.out.println(cardClass1 == cardClass2);

### Manipulation

Make a private constructor public

try {   
 Card card1 = Card.get(Rank.ACE, Suit.CLUBS);   
 Constructor 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( ReflectiveOperationExceptione){ e.printStackTrace(); }

Manipulate private 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( ReflectiveOperationExceptione){ e.printStackTrace(); }

Manipulate a method and test it (private method in the actual class). Test the private method from the test class and within this method the actual method under test gets invoked:

private Optional getPreviousCard(Card pCard) {   
 try {   
 Method method = FoundationPile.class.getDeclaredMethod("getPreviousCard", Card.class);   
 method.setAccessible(true);   
 return (Optional) 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());   
}

## Annotations

* typed and checked by the compiler

## Fixtures

* Baseline objects used for testing
* Declared as fields in a test class
* Support the execution of the test

## Stubs

* simplified version of an object that mimics its behavior

## Structures

* Fasts
  + run often with focus on a fast completion
  + Examples: acceptance tests & integration tests
* Independent
  + Execute in isolation and each test has a fresh init state
* Repeatable
  + Produce the same result in different environments
* Focused
  + Test as small parts of code as possible
* Readable
  + Naming convention

## Exceptions

Card peek() { if( isEmpty() ) { throw new EmptyStackException(); } }

@Test(expected = EmptyStackException.class)   
public void testPeek\_Empty() { new FoundationPile().peek(); }

@Test public void testPeek\_Empty() {   
 FoundationPile pile = new FoundationPile();   
 try {   
 pile.peek();   
 fail();   
 } catch(EmptyStackException e ) {}   
 assertTrue(pile.isEmpty()); }

# Composition

* For an object to be composed of other objects means that one object stores a reference to one or more other objects

Composition means that one object holds a reference to another object and delegates some functionality to it. Although this sounds simple, unprincipled composition can lead to a mess of spaghetti code.

* For an object to be composed of other objects means that one object stores a reference to one or more other objects.

Examples:

* One deck consists of several cards
* One String consists of several chars

Property:

* Transitive: an object that is composed of other objects can, itself, be one component or delegate of another parent object

## Concepts and Principles

### Divide and conquer

* define larger abstractions in terms of smaller ones
* problem solving strategy

## Law of Demeter

The Law of Demeter is a design guideline intended to help avoid the consequences of MESSAGE CHAIN†. The code of a method should only access:

* The instance variables of its implicit parameter;
* The arguments passed to the method;
* Any new object created within the method;
* (If need be) globally available objects.

## Programming Mechanisms

### Aggregation

Parent object that is being referred to

* A String is an aggregation of chars
* A Deck is an aggregation of Cards

### Delegation

aggregate object delegates some services to the objects that serve a role of specialized service to the aggregate

* Avoid god classes

### Cloning

* designs that make heavy use of polymorphism, copying can turn out to be a problem
* mechanism that provides us with polymorphic object copying. Specifically, we want to be able to make copies of objects without knowing the exact concrete type of the object:

if( source.getClass() == Deck.class ) {   
 return new Deck((Deck) source); }   
else if( source.getClass() == CardSequence.class ) {   
 return new CardSequence((CardSequence) source); })

…

* One specialized use of polymorphic copying is to support polymorphic instantiation

1. Declaring to implement the Cloneable interface;
2. Overriding the Object.clone() method; Calling super.clone() in the clone() method;
3. Catching CloneNotSupportedException in the clone() method;
4. Optionally, declaring the clone() method in the root supertype of a cloneable hierarchy.
   1. public interface CardSource extends Cloneable {   
       Card draw();   
       boolean isEmpty();   
       CardSource clone();   
      }

public class Deck implements Cloneable {  
 public Deck clone() {   
 try {   
 Deck clone = (Deck) super.clone();   
 clone.aCards = new CardStack(clone.aCards);   
 return clone;   
 } catch( CloneNotSupportedException e ) {   
 assert false;   
 return null; }  
 }

}

## Design Techniques

### Combining design patterns

Composite code

public void draw()

{

for( Figure figure : aFigures ) { figure.draw(); }

}

Decorator code

public void draw()

{

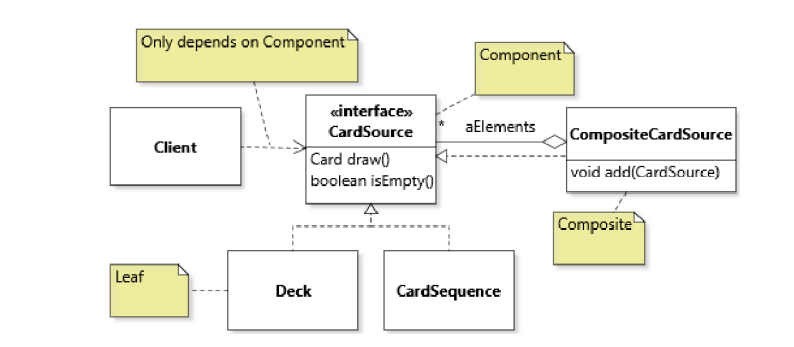
aFigure.draw(); // Additional code to draw the border   
}

## Patterns

### Composite

In this pattern, the three main roles are component, composite, and leaf:

* structural pattern
* aggregates a number of different objects of the component type
* implements the component interface -> composite objects to be treated by the rest of the code in exactly the same way as leaf elements
* define a class that represents multiple CardSources while still behaving like a single one



public CompositeCardSource {

private final List<CardSource> aElements;

public CompositeCardSource(List<CardSource> pCardSources)

{

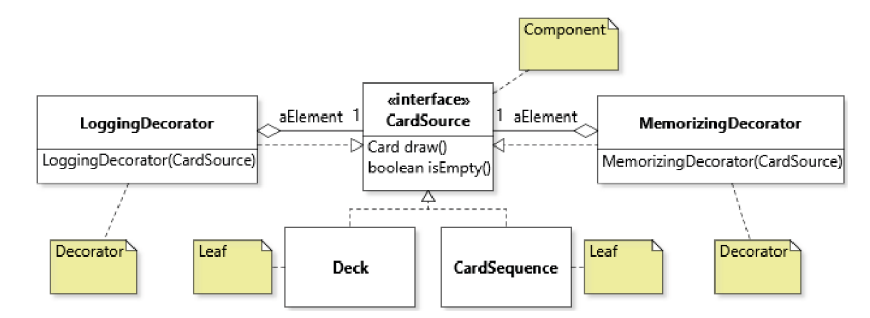
aElements = new ArrayList<>(pCardSources);

}

}

### Decorator

* structural pattern
* use case: java io libraries (e.g. filter input stream => bufferedinputstream => linenumberinputstream)
* accommodate run-time adjustments in the features of an object
* “decorate” some objects with additional features, while being able to treat the decorated objects just like any other object of the undecorated type
* Adapter vs decorator:
  + Adapter uses interfaces, delegation is done to the object, changing the interface
  + Decorator inheritance, adding functionalitiy



public class MemorizingDecorator implements CardSource

{

private final CardSource aElement;

private final List<Card> aDrawnCards = new ArrayList<>();

public MemorizingDecorator(CardSource pCardSource)

{ aElement = pCardSource; }

public boolean isEmpty()

{ return aElement.isEmpty(); }

public Card draw()

{

// 1. Delegate the original request to the decorated object   
 Card card = aElement.draw();

// 2. Implement the decoration

aDrawnCards.add(card);

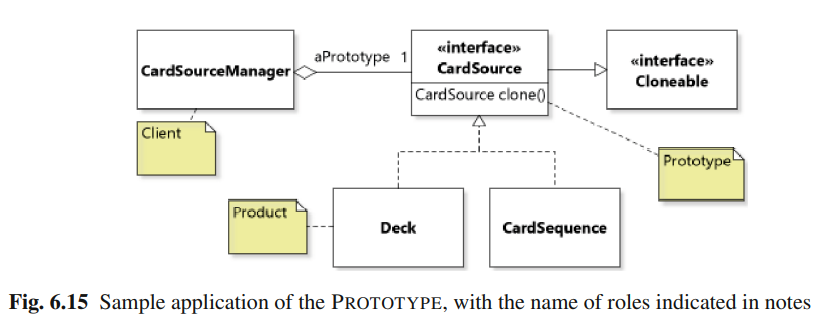
return card;

}

}

### Prototype

* option to create objects of different types does not increase the amount of control flow
* creational pattern



public class CardSourceManager

{

private CardSource aPrototype = new Deck(); // Default

public void setPrototype( CardSource pPrototype )

{ aPrototype = pPrototype; }

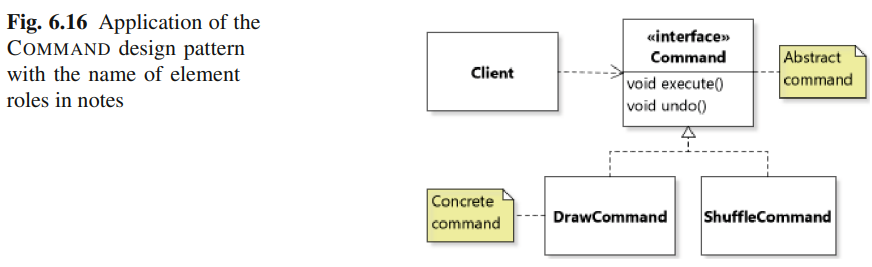
public CardSource createCardSource()

{ return aPrototype.clone(); }

}

### Command

* for objects to serve as manageable units of functionality
* way to manage abstractions that represent commands
* trigger the execution of a command whose exact nature is also determined at run-time



public class Deck

{

private CardStack aCards = new CardStack();

public createDrawCommand()

{

return new Command()

{

Card aDrawn = null;

public Optional<Card> execute()

{

aDrawn = draw();

return Optional.of(aDrawn);

}

public void undo()

{

aCards.push(aDrawn);

aDrawn = null;

}

}

}

}  
  
Deck deck = new Deck();

Command command = deck.createDrawCommand();

Card card = command.execute().get();

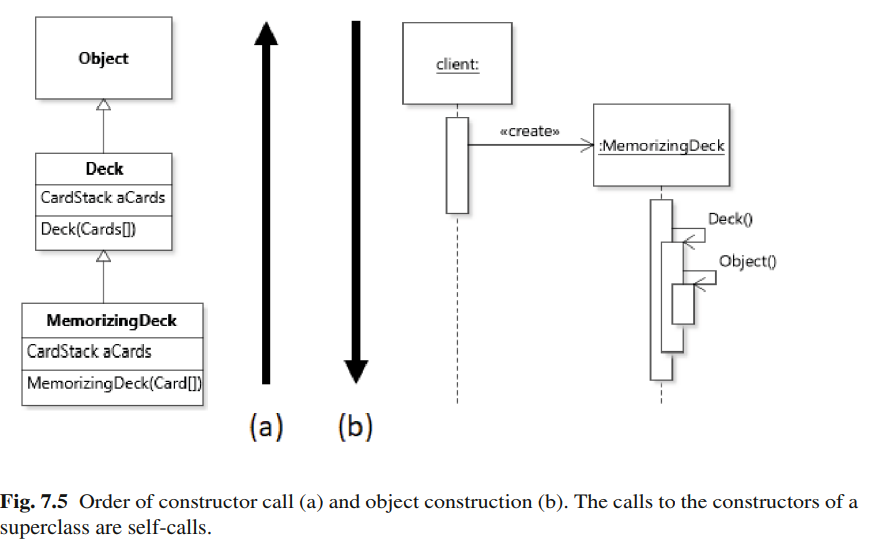
command.undo();

# Inheritance

* single inheritance, only inherit from one class
  + if there are no extensions, by default Class “Object” is extended
    - Every class extends Object
* avoid duplicated code (this is also the problem with interfaces)
* supports code reuse and extensibility
* idea: define a new class (the subclass) which extends the existing code base (superclass)

public Car {}; => superclass   
public Porsche extends Car {}; subclass

public static boolean isMemorizing(Deck pDeck) { return pDeck instanceof MemorizingDeck; }  
  
Deck deck = new MemorizingDeck();   
MemorizingDeck memorizingDeck = (MemorizingDeck) deck;   
boolean isMemorizing1 = isMemorizing(deck); // true   
boolean isMemorizing2 = isMemorizing(memorizingDeck); // true



## Composition vs Inheritance

* Composition
  + More flexibility
  + Contexts with many possible configurations (run-time configs)
  + Fewer options for detailed access
  + Identity problem there as we are dealing with multiple objects
* Inheritance
  + Context that require lot of compile-time configuration
  + Identity problem not there

## Casting

* enable unsafe type conversion operations

Reason for casting, this code would give a compilation error, since getDrawnCards is not in Deck but an extension to MemorizingDeck:

public class MemorizingDeck extends Deck { public Iterator getDrawnCards() { ... } }

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

Solution: downcasting (from the superclass down to a subclass)

MemorizingDeck memorizingDeck = (MemorizingDeck) deck;   
Iterator drawnCards = memorizingDeck.getDrawnCards();

## Fields

pulic class Deck implements CardSource { private final CardStack aCards = new CardStack();}   
public class MemorizingDeck extends Deck { private final CardStack aDrawnCards = new CardStack();}

MemorizingDeck Objects will have 2 fields: aCards and aDrawnCards. MemorizingDeck cannot access aCards but does have the field. To make the field accessible the field needs to be changed from private to protected.

* allow subclasses to manipulate some structure of the superclass when overriding methods
* fields are initialized top down => form object to superclass to class

Initialize fields from the superclass without changing the accessibility level:

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

* @Overriding is not a must if the method has the exact same definition (parameter and name) => with @Overriding it is another safety mechanism
* In case of multiple methods with the same name (overridden methods) from superclasses the JVM selects the most specific method based on the run-time type of the implicit parameter
  + This is called dynamic dispatch / dynamic binding

Deck deck = new MemorizingDeck();   
Card card = deck.draw();

* The selection of the code above would be the draw method from MemorizingDeck not Deck

Overriding a method without having write access to a field (private) from the superclass could be done with a super call:

@Override  
public class MemorizingDeck extends Deck {   
 public Card draw() {   
 // Problematic as aCards is private  
 // Card card = aCards.pop();  
 // Problematic as this method would be called not super  
 // Card card = draw();  
 Card card = super.draw();   
 aDrawCards.push(card);   
 return card;   
 }   
}

## Overloading

* selecting the method based on the types of the explicit parameters
* constructor overloading
* primitives overloading

public class MemorizingDeck extends Deck {   
 private CardStack aDrawnCards = new CardStack();   
 public MemorizingDeck() { /\* Version 1: Does nothing besides the initialization \*/ }   
 public MemorizingDeck(CardSource pSource) { /\* Version 2: Copies all cards of pSource into \* this object \*/ }   
 public MemorizingDeck(MemorizingDeck pSource) { /\* Version 3: Copies all cards and drawn cards of pSource \* into this object \*/ }   
}

MemorizingDeck memorizingDeck = new MemorizingDeck(); //constructor 1  
Deck deck = memorizingDeck;   
Deck newDeck1 = new MemorizingDeck(memorizingDeck); // constructor 3  
Deck newDeck2 = new MemorizingDeck(deck); //constructor 2 => CardSource but not a subtype of MemorizingDeck

## Abstracts

* represents a correct but incomplete set of class member declarations
* cannot be instantiated
* no longer needs to supply an implementation for all the methods in the interface
  + concrete classes need to have all interface classes

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

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

## Finals

* Class: class that cannot be subclassed
  + help to ensure that the class is not extended or modified in unintended ways: more robust code
  + limits its flexibility and extensibility
* Method: method cannot be overridden by subclasses
  + Benefits: Same as class + improve performance

## Patterns

### Template Method

* common algorithm applies to objects of a certain base type, but varies from subclasses
* re-implementing common behavior
* no duplicated code
* behavioral pattern

Template example

public abstract class AbstractMove implements Move {   
 protected final GameModel aModel;   
 protected AbstractMove(GameModel pModel) { aModel = pModel; }   
 public final void perform() { aModel.pushMove(this); execute(); log(); }   
 protected abstract void execute();   
 private void log() { System.out.println(getClass().getName()); }   
}

#### Final methods

* method cannot be overridden by subclasses
* classes declared to be final cannot be inherited

#### Abstract methods

* this method must be overridden in the subclass with a meaningful function

## Liskov substitution principle (LSP)

* The intuition that inheritance should only be used for extension
* Methods of subclasses:
  + Cannot have stricter preconditions;
  + Cannot have less strict postconditions;
  + Cannot take more specific types as parameters
  + Cannot make a method less accessible (public to protected)
  + Cannot throw more checked exceptions
  + Cannot have a less specific return type
* Circle–Ellipse problem: class to represent a circle is defined by inheriting from an Ellipse class and preventing clients from creating any ellipse instance that does not have equal proportions

# Inversion of control

* reversing the usual flow of control from caller code to called code to achieve a better separation of concerns and looser coupling
* Applicable when: stateful objects need to be kept consistent
* Synchronization problem needs to be solved

## Model View Control

* decomposition of concerns
* separate abstractions responsible for storing data from abstractions responsible for viewing data, from abstractions responsible for changing data
* Model: Data storage
* View: Viewing data
* Control: Logic data

## Event Handling

* objects in the component graph act as models in the OBSERVER
  + Define a handler for the even
  + Instantiate a handler
  + Register the handler

GUI component with event handling:

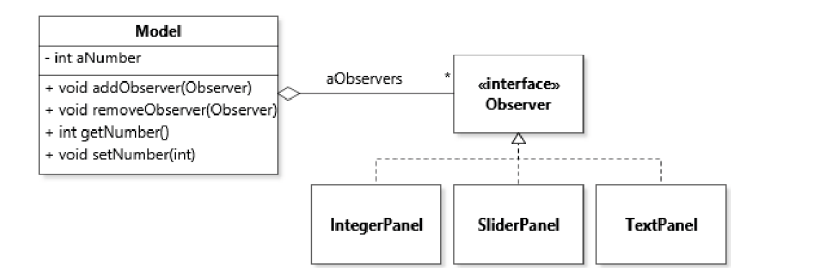
* Observing the model
* Observing changes in the element and send to the model
* Adding the element to the root

public class IntegerPanel extends Parent implements Observer {   
 private TextField aText = new TextField();   
 private Model aModel;   
 public IntegerPanel(Model pModel) {   
 aModel = pModel;   
 aModel.addObserver(this);   
 aText.setText(new Integer(aModel.getNumber()).toString());   
 getChildren().add(aText);   
 aText.setOnAction(new EventHandler() {   
 public void handle(ActionEvent pEvent) {   
 int number = 1;   
 try{ number = Integer.parseInt(aText.getText()); }  
 catch(NumberFormatException pException ) { /\* Just ignore. \*/ }   
 aModel.setNumber(number); }   
 });   
 }   
}

## Patterns

### Observer

* store data of interest in a specialized object
* allow other objects to observe this data
* model can be used without any observer
* possible to register and deregister observes at runtime
* behavioral pattern
* one to many relationship => loose coupling => interact with each other without knowing anything about it => good thing



public class Model {   
 private int aNumber = 5;   
 private ArrayList<Observer> aObservers = new ArrayList<>();   
 public void addObserver(Observer pObserver) { aObservers.add(pObserver); }   
 public void removeObserver(Observer pObserver) { aObservers.remove(pObserver); }   
 private void notifyObservers() { // use the call back method to inform the observers  
 for(Observer observer : aObservers) { observer.newNumber(aNumber); }   
 }  
 // set new number in the model and automatically update the observers  
 public void setNumber(int pNumber) {   
 if( pNumber <=0){ aNumber = 1; }   
 else if( pNumber > 10 ) { aNumber = 10; }   
 else { aNumber = pNumber; }   
 notifyObservers();   
 }  
 /\* getters and setters for all elements \*/  
}

public interface Observer {   
 void newNumber(int pNumber); //callback method push data flow  
 void newNumber(Model pModel); // callback method for a pull data flow   
}

class IntegerPanel implements Observer {   
 // User interface element that represents a text field   
 private TextField aText = new TextField();  
 // push strategy model  
 public void newNumber(int pNumber) {   
 aText.setText(Integer.toString(pNumber));   
 }  
 }

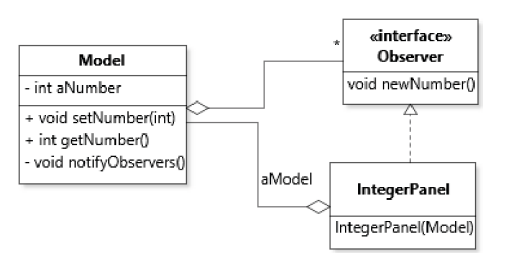
class IntegerPanel implements Observer {   
 // User interface element that represents a text field   
 private TextField aText = new TextField();  
 // pull strategy model  
 public void newNumber(Model pModel) {  
 aText.setText(Integer.toString(pModel.getNumber()));   
 }  
 }

**Callback method**: used to inform observers about the changes, do not tell them what to do

**Push data flow strategy**: assumption that we know what data the model sends the observers

**Pull data flow strategy**: any data available through the methods of class Model becomes available to the observers, increase the coupling between model and its observers

Class diagram of a pull data flow strategy:

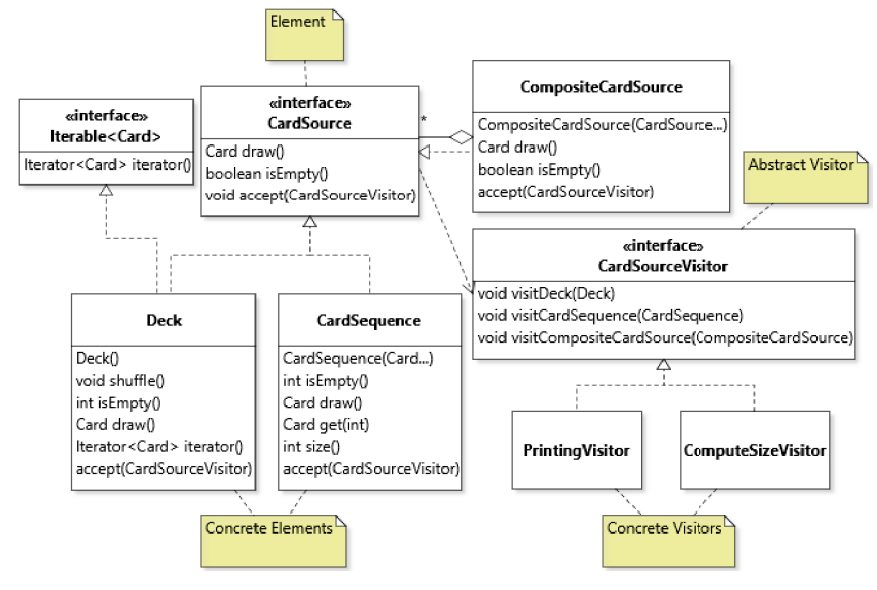


#### Event based programming

* Model: Event source: generates a series of events that correspond to different state changes of interest
* Observers: Event handlers: in charge of reacting to these events

### Visitor

* make it possible to support an open-ended number of operations that can be applied to an object graph
* interface that describes objects that can “visit” objects of all classes of interest in an object graph: abstract visitor
* enable the integration of an open-ended set of operations
* can be applied by traversing an object graph, often a recursive one
* stateful
* behavioral pattern



public interface CardSourceVisitor {   
 void visitCompositeCardSource( CompositeCardSource pSource );   
 void visitDeck( Deck pDeck );   
 void visitCardSequence( CardSequence pCardSequence );   
}

Concrete Visitor of the interface above:

* organize code in terms of functionality as opposed to data => one class which handles the printing of different classes: functionality-centric vs. data-centric

public class PrintingVisitor implements CardSourceVisitor {   
 public void visitCompositeCardSource(CompositeCardSource pSource) {}   
 public void visitDeck(Deck pDeck) { for( Card card : pDeck) { System.out.println(card); } }  
 public void visitCardSequence(CardSequence pCardSequence) {   
 for( int i = 0; i < pCardSequence.size(); i++ ) {  
 System.out.println(pCardSequence.get(i));   
 }   
 }   
}

Initial interface adaptions are required: an “accept” method is required:

public interface CardSource {   
 Card draw(); boolean isEmpty();   
 void accept(CardSourceVisitor pVisitor);   
}

Actual calls to the visitor pattern from the classes:

public void accept(CardSourceVisitor pVisitor) { pVisitor.visitDeck(this); }   
public void accept(CardSourceVisitor pVisitor) { pVisitor.visitCardSequence(this); }

PrintingVisitor visitor = new PrintingVisitor();   
Deck deck = new Deck();   
deck.accept(visitor);

Traversal code for an object graph:

* within the accept method: encapsulation is more important
* in the visit method: traversal order is important => might lead to duplicated code if there is more than one concrete class => prevent this with inheritance, use an abstract class for a one time implementation and reuse it

// example of a traversal method in the accept method  
public class CompositeCardSource implements CardSource {   
 private final List aElements;   
 public void accept(CardSourceVisitor pVisitor) {   
 pVisitor.visitCompositeCardSource(this);   
 for( CardSource source : aElements ) { source.accept(pVisitor); }   
 }   
}

# Functional Design

* function as its primary building block
* high-order functions: function that takes other functions as argument
* design goal: parameterize the behavior of a function
* mechanism: pass a reference to an object in the parameter
* computation is organized by transforming data, ideally without mutating state

## Functional interfaces

* interface type that declares a single abstract method
* Function<List<Card>, Card> => return a Card given a List of Cards

public interface Filter { boolean accept(Card pCard); }

## Lambda expressions

* anonymous functions
* supply custom behavior not defined anywhere else
* a single expression (e.g., a == 1)
* block of one or more statements (e.g., {return a == 1;})

Predicate blackCardFilter = (Card card) -> card.getSuit().getColor() == Suit.Color.BLACK;

Predicate blackCards = (Card card) -> { return card.getSuit().getColor() == Suit.Color.BLACK; };

## Method references

* reuse our method as first-class function
* pass a reference to the classes’ method to the executed function
* support using both static and instance methods as first-class functions
* indicated with a double colon expression
  + defined\_Class::method\_Of\_Interest

public final class Card {   
 public boolean hasBlackSuit() { return aSuit.getColor() == Color.BLACK; }   
 public boolean hasRedSuit() { return aSuit.getColor() == Color.RED; }   
}

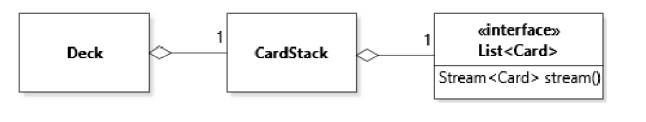
// lambda: ends in duplicated code  
ArrayList cards = ... cards.removeIf( card -> card.getSuit().getColor() == Suit.Color.BLACK);

// method refrerence: reuse own written function hasBlackSuit()  
cards.removeIf(Card::hasBlackSuit);

## Streams

* a sequence of data elements (like a collection)
  + sequence represents a data flow
  + collection represents a data store
* pipelining: output stream of a function
  + combine operations on streams
* terminal operation: stream functions that do not produce a stream as result
* essential higher-order function for streams is forEach

|  |  |
| --- | --- |
| **Collection** | **Stream** |
| * Elements exist before being added to the collection | * Can be computed on demand |
| * Store a finite number of elements | * Store an infinite number of elements |
| * Can be traversed multiple times => traversal code is located outside the collection: e.g. for loop or iterator | * Traversed once: elements are consumed * Traversal code hidden within the higher-order function provided |
| * Almost no support for higher order functions | * Parallelized due to traversal code * Support functional-style programming |



public class CardStack implements Iterable {   
 private final List aCards;   
 public Stream stream() { return aCards.stream(); }   
}

public class Deck implements CardSource , Iterable {   
 private CardStack aCards;   
 public Stream stream() { return aCards.stream(); }   
}

Stream cards = new Deck().stream();   
long total = cards.count(); // terminal function  
Stream sortedCards = cards.stream().sorted(); // return values are coming from the “pipeline”  
Stream sortedCards = cards.stream().sorted().limit(10); // combining pipelines is easy  
Stream cards = Stream.concat(new Deck().stream(), new Deck().stream());  
long result = cards.stream() // filtering cards => apply filter first and then the terminal methods  
 .filter(Card::isFaceCard) // use method from class card  
 .filter(card -> card.getSuit() == Suit.CLUBS) // create new filter method  
 .count();   
cards.stream().map(card -> card.getSuit().getColor() ); //return a stream of colors   
long result = cards.stream()   
 .map(card -> card.getSuit().getColor() ) // returns a color stream  
 .filter( color -> color == Color.BLACK ) //filters the black ones  
 .count(); //returns the counts of the black ones

## Patterns

### Strategy

* Extension of Strategy-Pattern above
* define families of functions whose type is compatible and invoke them interchangeably
* interface: single method => abstract strategy
* prevent switch statements
* concrete strategy: use lambda expressions to access the methods
* behavioral pattern

public class AutoPlayer {   
 private Function<List, Card> aSelectionStrategy;   
 public AutoPlayer(Function<List, Card> pSelectionStrategy) {   
 aSelectionStrategy = pSelectionStrategy;   
 }   
 public void play() { Card selected = aSelectionStrategy.apply(getCards()); ... }   
 // Gets the cards to supply to the strategy   
 private List getCards() {...}   
}

public final class CardSelection {   
 private CardSelection() {}   
 public static Card lowestBlackCard(List pCards) { ... }   
 public static Card highestFaceCard(List pCards) { ... }   
 ...   
}

AutoPlayer player = new AutoPlayer(CardSelection::lowestBlackCard);

### Command

Limitations (same in strategy):

* rely on external variable names to keep track of the objects -> not self-documenting
* cannot arbitrarily add constraints to the declaration of its method -> no additional pre-condition adding
* if a data outflow is required, mechanisms need to be implemented to support that

public class DeckCommand {   
 private final Runnable aCommand; // Runnable: functional interface with a .run() method  
 private final String aName; // keeps track of the object (documentation)  
 public DeckCommand(String pName, Runnable pCommand) {   
 aName = pName; aCommand = pCommand;   
 }   
 public void execute() { aCommand.run(); }   
}

Deck deck = new Deck();   
List drawnCards = new ArrayList<>();   
DeckCommand draw = new DeckCommand("draw", ()-> drawnCards.add(deck.draw()));

# Antipatterns

### PAIRWISE DEPENDENCIES†

Changes in an interface, the interface then directly contacts all other interfaces and updates their view in regards of the changes:

* Low extensibility
* High coupling
* No general entity managing the data
* No single source of truth
* Use an MVC pattern

### Long Method†

### PRIMITIVE OBSESSION†

Tendency to use primitive types to represent other abstractions. E.g. numbers to represent a card.

* Value object pattern: It's a simple and lightweight object that only contains a small amount of data and has no behavior

### INAPPROPRIATE INTIMACY†

class that is mostly accessed through getters and setters points to a design weakness, because the abstraction the object represents is not effective

### GOD CLASS†

Choosing the path of the least resistance (put all the logic into one class because all the objects know each other) -> no thoughts of a design pattern is needed

### MESSAGE CHAIN†

Overly long commands:  
aFoundations.getPile(FIRST).getCards().add(pCard);

* Law of dementor

### SWITCH STATEMENT†

* Having too many switch case statements and if else’s
* Having flags to execute methods differently
* Use strategy pattern

### SPECULATIVE GENERALITY†

* What if we need this one day?
* Example: Useless states which are not necessary and relevant in an application
* Visitor design pattern

### TEMPORARY FIELD†

* information should not be stored in an object unless it uniquely contributes to the intrinsic value represented by the object
* Example: manipulating a value within a function instead of letting the caller use a immutable function  
  private int aSize = 0;   
  public int size() { return aSize; }   
  public Card draw() { aSize--; return aCards.remove(aCards.size()); }  
  instead of  
  public int size() { return aCards.size(); }

### LONG METHOD†

* Well-designed methods are short and simple

### NULL REFERENCES†

* Variable is correct but temporarily un-initialized
* Variable is incorrectly initialized
* Flag that correctly represents the absence of a value
* Flag that must be interpreted differently
* Prevent with input validation or design by contract

public Card(Rank pRank, Suit pSuit) {

if( pRank == null || pSuit == null) { throw new IllegalArgumentException(); }

aRank = pRank; aSuit = pSuit;

}

/\*\*

\* @pre pRank != null && pSuit != null;

\*/

public Card(Rank pRank, Suit pSuit) {

assert pRank != null && pSuit != null;

aRank = pRank; aSuit = pSuit;

}

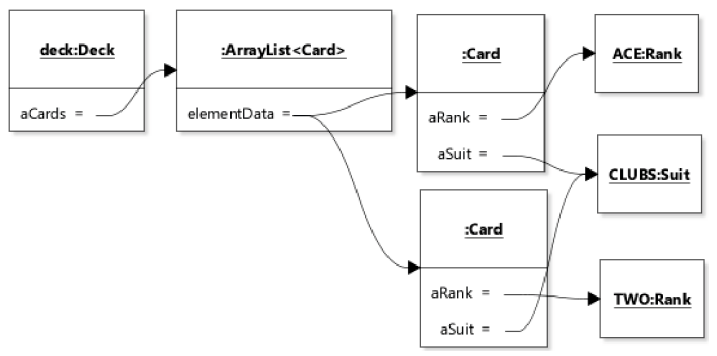
### DUPLICATED CODE†

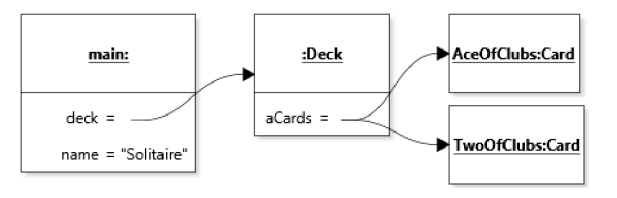
# UML

* a stereotype is a variation on an element type <<anonymous>>, <<interface>>, <<abstract>>

## Object Diagram

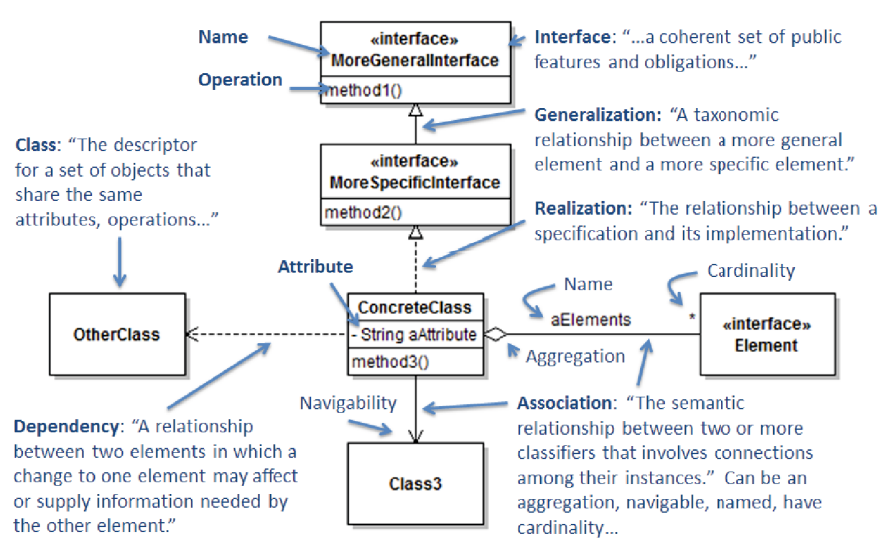
* name:type => name of objects are underlined.
* Objects can contain fields (primitive values, reference types…)
* Direct arrow representation
* represent a snapshot in the execution of a program
* represents a dynamic (run-time) perspective: corresponds to the set of all values and references held by the variables in a program at different points in time





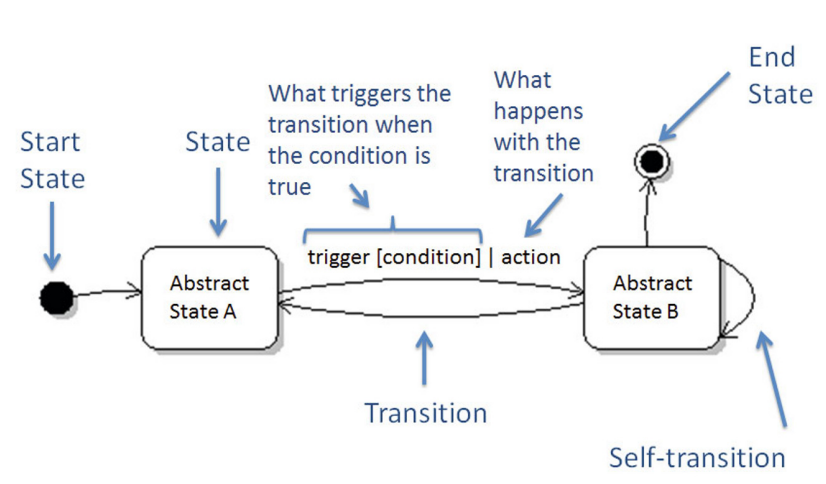
## Class Diagram

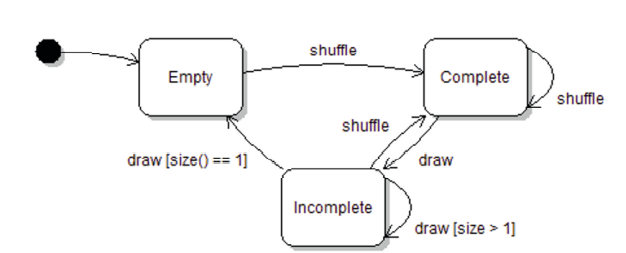
* represent a static, or compile-time, view of a software system
* closest to the code
* poor vehicle for capturing any kind of run-time property of the code
* aggregation (white diamond) and composition (black diamond).
* inheritance is denoted by a solid line with a white triangle
* interface is denoted by a dashed line with a white triangle
* static structure of the system



## State diagrams

* represents a dynamic (run-time) perspective corresponds to the set of all values and references held by the variables in a program at different points in time
* *absence of a transition* means that the absent transition is not possible => drawing card from empty deck is not possible
* *self-transition* events that do not result in a change of abstract state
* *action* could be added to the transition: “draw | remove card from stack”
* *end state* is a certain state at the end of the object’s lifetime. When all references to an object are eliminated and the object is ready for garbage collection
* States have to be named with adjectives

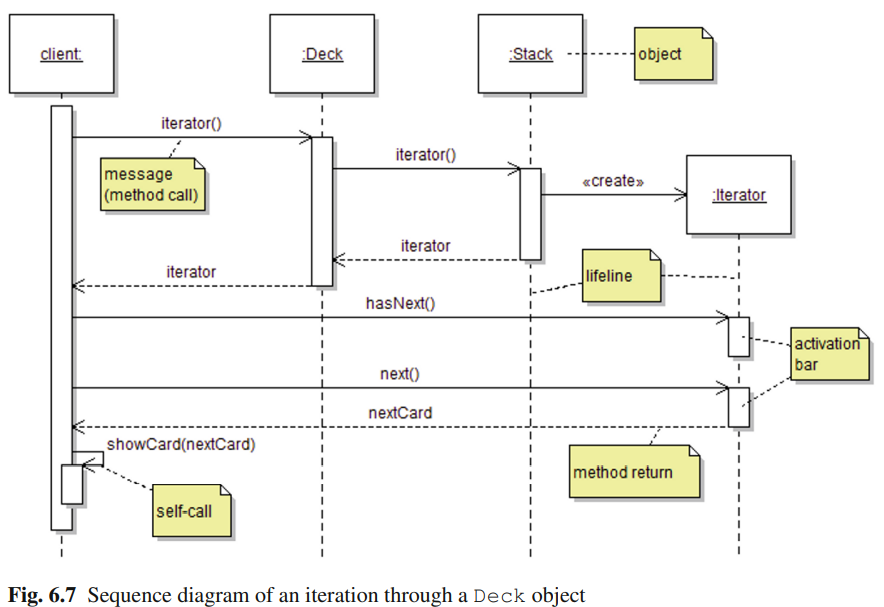




## Sequence diagrams

* represents a dynamic (run-time) perspective: corresponds to the set of all values and references held by the variables in a program at different points in time
* represent a specific execution of the code
* closest representation to what one would see when stepping through the execution of the code in a debugger
* complements the class diagram by showing a dynamic aspect of the design that is invisible on the class diagram
* dynamic behavior of the system

implicit parameter: object in a sequence diagram (rectangle on top)



# Patternbook

## Abstract Factory

* makes it easy to create and manipulate objects without knowing exactly what they are
* benefit: anonymity of the Concrete Factory and Product promotes reuse
* cons: product doesn’t do what you want, you may have to change the Abstract Product interface, which is difficult

## Factory Method

* Let subclasses decide which objects to instantiate
* Most can be done in superclass => decision which object you’ll be working on during runtime
* create specialized worker objects in a specialized subclass
* pro: small version of abstract factory
* con: forces inheritance + not the best way to achieve reuse

## Adapter Pattern

* Make a class appear to support a familiar interface that it doesn’t actually support.
* easy to add classes without changing code
* Difficult to implement when the library is designed poorly

## Bridge

* decouple subsystems so that either subsystem can change radically without impacting any code in the other one, put a set of interfaces between two subsystems and code to these interfaces
* independence
* pro: customized behavior to specific platforms
* con: interface creation is difficult

# Off topic

* tetris candidate classes:
  + GameOperator
  + Player
  + Shape
    - Shape1
    - Shape2
    - Shape3
  + Board
  + Cell
  + State
    - Occupied
    - Empty

|  |  |
| --- | --- |
| **Game** | |
| Superclass: None | |
| Subclass: None | |
| Set up the initial state of the game | Board, Cell |
| Get players’ inputs and create player objects | Player |
| Display the state of the game | None |
| Give response when user perform some actions | Cell |
| Let users take turns | None |
| Evaluate and declare winner | Player |