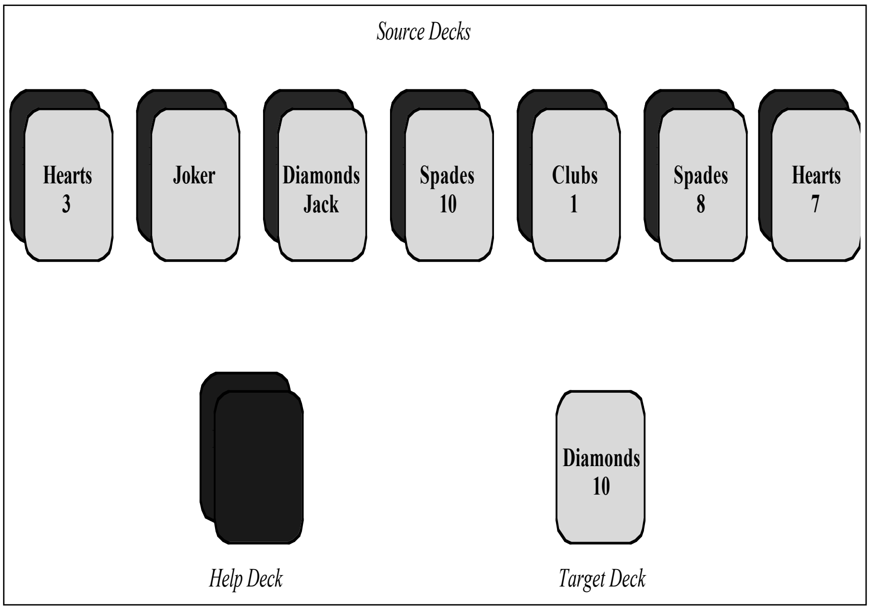
Department of Computer Science

The Liskov Substitution Principle

- Building hierarchies of classes
- Dynamic binding

Assignment



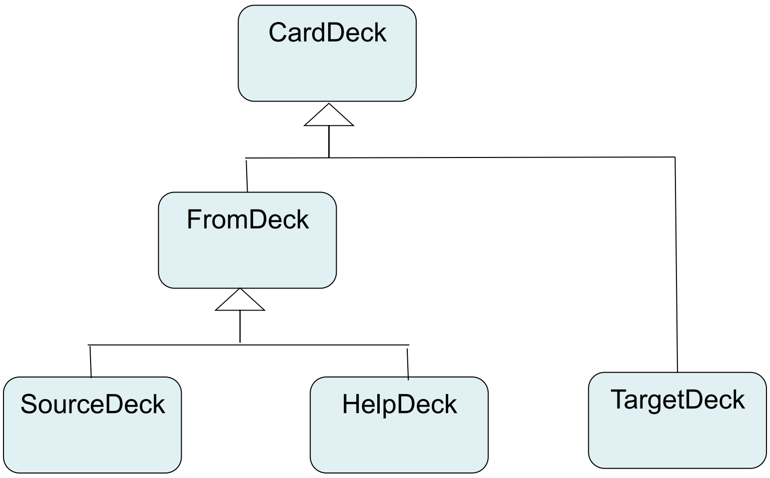
Assignment

KATHOLIEKE UNIVERSITEIT LEUVEN

- Hierarchy of Decks of Cards
  - Constructor involving minimum and maximum number of cards
  - Mutator “moveTop”
  - Mutators “pushCard” and “popCard”
  - Inspector “getNbCards”
  - Inspector “getCardAt”
  - Inspector “getMinimumNbCards” and “getMaximumNbCards”
  - Inspector “getCardAtTop”
  - Inspector “getAllCards”
  - ...
- Hierarchy of Cards
  - Inspector “matchesOn”
  - ...

Card Decks

KATHOLIEKE UNIVERSITEIT LEUVEN



Overview

- The substitution principle by Liskov supports the view of generalization/specialization inheritance
  - The principle states that objects of a superclass must be substitutable by objects of its subclasses
    - The principle by Liskov has consequences for all possible items involved in the redefinition of a method
- Dynamic binding in Java only applies to the implicit argument of instance methods
  - Double dispatch has been proposed as a technique to simulate dynamic binding on explicit arguments in an elegant way
    - The pattern does not completely solve the adaptability problem, raised by a lack of support for dynamic binding
- Epilogue

Substitution Principle: Basics

- Objects of superclasses must be substitutable by objects of any of its subclasses
  - Whenever a method is invoked against a variable of a supertype, the behavior of that method must be preserved
    - This must be true even if the variable is actually referencing an object of a subclass
  - The principle is also referred to as “Behavioral Subtyping”

```
BankAccount myAccount;

myAccount = new SavingsAccount(…);
myAccount.withdraw(100);

myAccount = new UniversalAccount(…);
myAccount.withdraw(100);
```

Class Invariants

- Class invariants can only be strengthened at the level of subclasses
  - Subclass-objects must at least satisfy all restrictions imposed on superclass-objects
    - Technically, class invariants at the level of the subclass or and-ed with inherited class invariants

```
/**
 * @invar …
 *      | getBalance() > 0
 */
public class SuperClass {
  …
}
```

```
/**
 * @invar …
 *      | getBalance() > 100
 */
public class SubClass
    extends Superclass{
  …
}
```

Preconditions

- Preconditions can only be weakened at the level of subclasses
  - The redefinition of a method may promise its effects under more conditions than initially stated at the level of the superclass
    - Technically, preconditions of methods at the level of the subclass or or-ed with preconditions inherited from their superclass version

```
/**
 * @pre …
 *    | partner != null
 * @pre …
 *    | partner.getAge() >= 18
 */
public void marry
    (Person partner) {
  …
}
```

```
/**
 * @pre …
 *    | partner != null
 * @pre …
 *    | partner.getAge() >= 16
 */
@Override
public void marry
    (Person partner) {
  …
}
```

Postconditions

KATHOLIEKE UNIVERSITEIT LEUVEN

- Postconditions can only be strengthened at the level of subclasses
  - The redefinition of a method may promise more or better effects than initially stated at the level of the superclass
    - Technically, postconditions of methods at the level of the subclass or and-ed with postconditions inherited from their superclass version
  - Strengthening postconditions is only meaningful if the superclass-version is non-deterministic

```
/**
 * @return …
 *    | (result >= 0.0) &&
 *    | (result <= 1.0)
 */
public double random() {
  …
}
```

```
/**
 * @return …
 *    | (result >= 0.25) &&
 *    | (result <= 0.75)
 */
@Override
public double random() {
  …
}
```

Postconditions

KATHOLIEKE UNIVERSITEIT LEUVEN

- Strengthening postconditions of mutators also requires some non-determinism in the superclass-version
  - Specifications of mutators are complemented with clauses derived from the frame-axioms
    - A mutator “withdraw” implicitly leaving the credit limit of an account untouched cannot be redefined such that it changes that credit limit

```
/**
 * @post …
 *    | (new.getCredits() >= this.getCredits()/3) &&
 *    | (new.getCredits() <= this.getCredits()*3)
 */
public void gamble() { … }
```

```
/**
 * @post …
 *    | (new.getCredits() >= this.getCredits()/2) &&
 *    | (new.getCredits() <= this.getCredits()*2)
 */
@Override public void gamble() { … }
```

Exceptions

- The set of exceptions that can be thrown by methods can only be narrowed at the level of subclasses
  - Java supports the narrowing of (checked) exceptions in redefining methods at the level of subclasses

```
public void withdraw(MoneyAmount amount)
   throws IllegalArgumentException, IllegalStateException {
  …
}
```

```
@Override
void withdraw(MoneyAmount amount)
   throws IllegalArgumentException {
  …
}
```

Argument Types

- Types of arguments for methods can only be weakened at the level of subclasses
  - Types of formal arguments are just special kinds of preconditions, supported by the programming language
    - In the literature, the widening of types is referred to as contravariance
  - Java does not support the weakening (widening) of argument types in method redefinitions

```
public void marry
     (Woman partner) {
  …
}
```

```
@Override
public void marry
     (Person partner) {
  …
}
```

Return Types

- Return types for methods can only be strengthened at the level of subclasses
  - Return types are just special kinds of postconditions, supported by the programming language
    - In the literature, the narrowing of types is referred to as covariance
  - Since Java 1.5, the language supports the strengthening (narrowing) of return types in method redefinitions

```
public Person getSpouse() {
  …
}
```

```
@Override
public Woman getSpouse() {
  …
}
```

Access Rights

- Access rights for methods can only be widened at the level of subclasses
  - The redefinition of a method may turn the method more accessible than its superclass-version
- Java supports the widening of access rights for all the members of a class

```
protected void setSpouse
    (Person spouse) {
  …
}
```

```
@Override
public void setSpouse
    (Person spouse) {
  …
}
```

Task 1

Exceptions

- Throws clauses are extended with conditions under which methods can throw exceptions
  - A method must terminate in a normal way if none of the conditions are satisfied under which it must or can throw exceptions
  - A method must terminate with an exception if at least one of the conditions is satisfied under which it must throw an exception
  - A method may terminate in a normal way or with an exception if only conditions under which it can throw exceptions are satisfied
    - A question mark separates conditions under which a method must throw an exception from conditions under which it can throw it

```
/**
 * @throws IllegalArgumentException
 *       | amount <= 0 ? Amount >= 10000
 */
public void withdraw(long amount)
   throws IllegalArgumentException { … }
```

Exceptions

- Conditions under which a method can throw an exception can both be strengthened and weakened
  - The redefinition of a method may promise its effects under more conditions than initially stated at the level of the superclass
  - The redefinition of a method may always throw exceptions under conditions that could be thrown at the level of the superclass

```
/**
 * @throws IllegalArgumentException
 *       | (amount <= 0) ? (amount > 10000)
 */
public void withdraw(long amount)
   throws IllegalArgumentException { … }
```

Task 2

```
/**
 * @throws IllegalArgumentException
 *   | (amount <= 0) || (amount > 20000) ? (amount > 15000)
 */
public void withdraw(long amount)
   throws IllegalArgumentException { … }
```

Overview

- The substitution principle by Liskov supports the view of generalization/specialization inheritance
  - The principle states that objects of a superclass must be substitutable by objects of its subclasses
    - The principle by Liskov has consequences for possible items involved in the redefinition of a method
- Dynamic binding in Java only applies to the implicit argument of instance methods
  - Double dispatch has been proposed as a technique to simulate dynamic binding on explicit arguments in an elegant way
    - The pattern does not completely solve the adaptability problem, raised by a lack of support for dynamic binding
- Epilogue

Dynamic Binding

- As for most object-oriented programming, dynamic binding in Java only applies to the implicit argument
  - If a class introduces methods “f(SuperClass)” and “f(SubClass)”, the selection is based on the static type of the actual argument
    - If method “f” is invoked with an argument of static type Superclass, the method “f(SuperClass)” is selected
      - This is even so if the dynamic type of the argument is “SubClass”

Double Dispatch

- If different versions of a method exist with different types of arguments, the selection must be programmed manually
  - One alternative is to use conditional statements (switch statements) for selecting the most appropriate version to be executed
    - Such controlling logic is extremely unhandy at the time new subclasses must be added
- Double dispatch has been proposed as a pattern (or a technique) to solve this problem
  - The idea of double dispatch is to change the role of the objects involved, turning an explicit argument into the implicit argument
    - A method f(Superclass x) may invoke x.f(this), resulting in dynamic binding on the object x
  - A class must introduce similar methods with different types of arguments
    - This would also be the case if dynamic binding on explicit arguments were supported

Task 3

Overview

- The substitution principle by Liskov supports the view of generalization/specialization inheritance
  - The principle states that objects of a superclass must be substitutable by objects of its subclasses
    - The principle by Liskov has consequences for possible items involved in the redefinition of a method
- Dynamic binding in Java only applies to the implicit argument of instance methods
  - Double dispatch has been proposed as a technique to simulate dynamic binding on explicit arguments in an elegant way
    - The pattern does not completely solve the adaptability problem, raised by a lack of support for dynamic binding
- Epilogue

Epilogue

KATHOLIEKE UNIVERSITEIT LEUVEN

- To be worked out

Homework

KATHOLIEKE UNIVERSITEIT LEUVEN

- Extend the game of thieves with the following cards
  - Queens: A queen matches a jack and a joker, and vice versa.
  - Kings: A king matches a queen and a joker, and vice versa.
  - Odd-matcher: an odd matcher matches a joker and a numbered card with value 1, 3, 5, 7 or 9
  - Even-matcher: an even matcher matches a joker and a numbered card with value 2, 4, 6 or 8
  - Prime-matcher: a prime matcher matches a joker, a numbered card with value 1, 2, 3, 5 or 7, a jack and a king