Task 1

Resources:

https://www.geeksforgeeks.org/prototype-design-pattern/

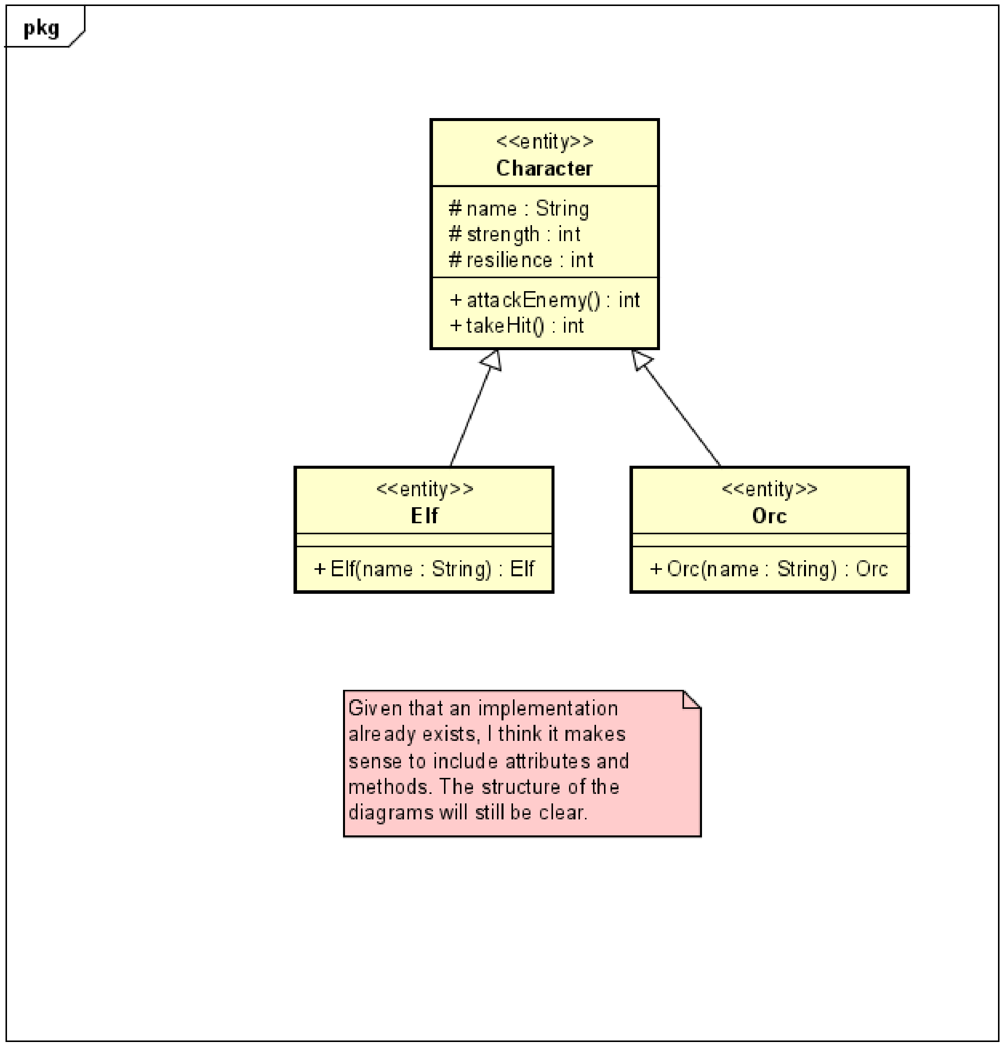
https://en.wikipedia.org/wiki/Memento_pattern

https://www.geeksforgeeks.org/template-method-design-pattern/

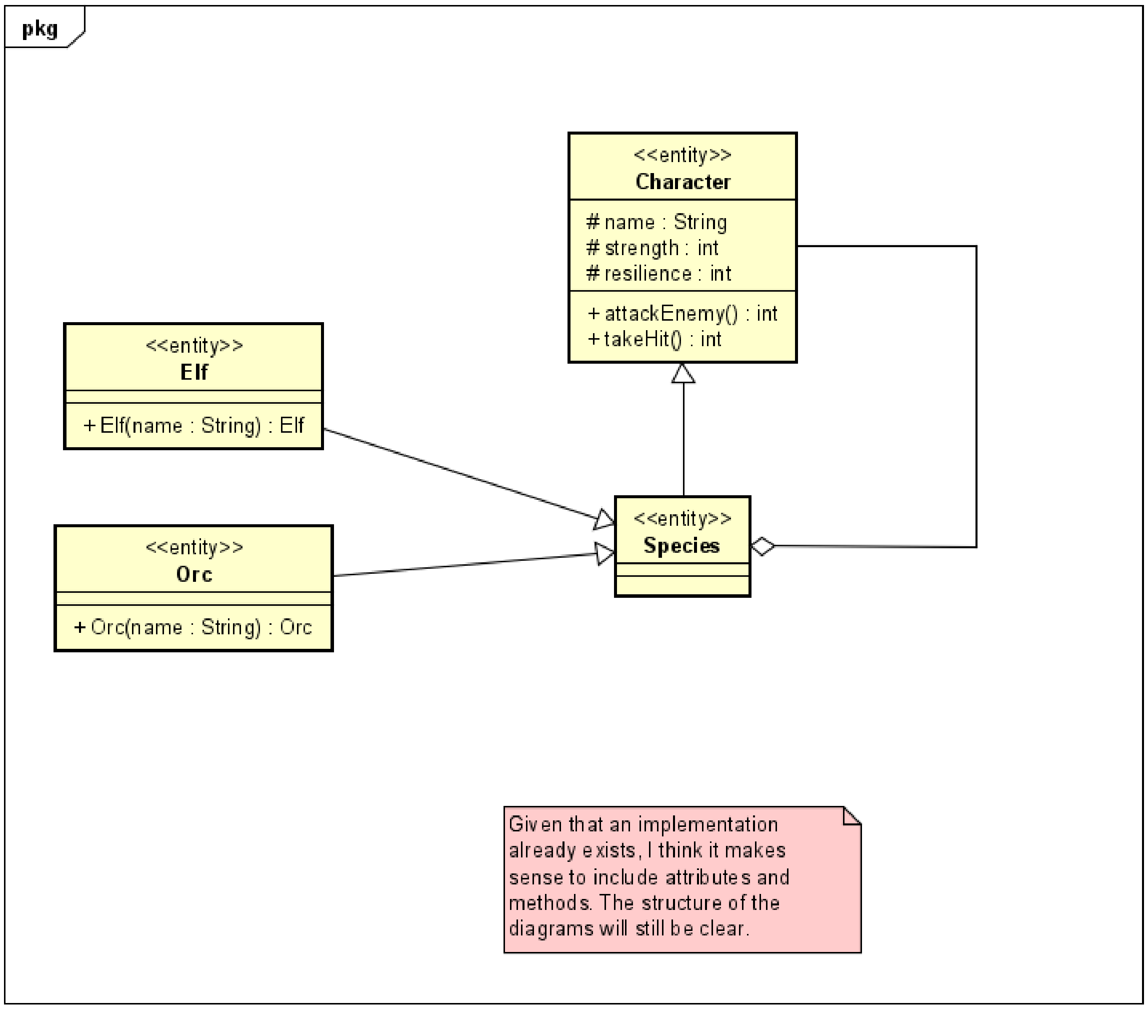
Task 2

For project pattern 2, I decided that a Decorator design pattern would be appropriate. Because some simple generalization is being used to have different types of characters, I thought that the Decorator design pattern might help open the door for more modularity to allow for a character to become an Orc or an Elf. It also helps make the creation of different species of characters easier and helps prevent the creation of a monolithic Character class that houses every possible character attribute.

pattern 2 Before Design Pattern Integration

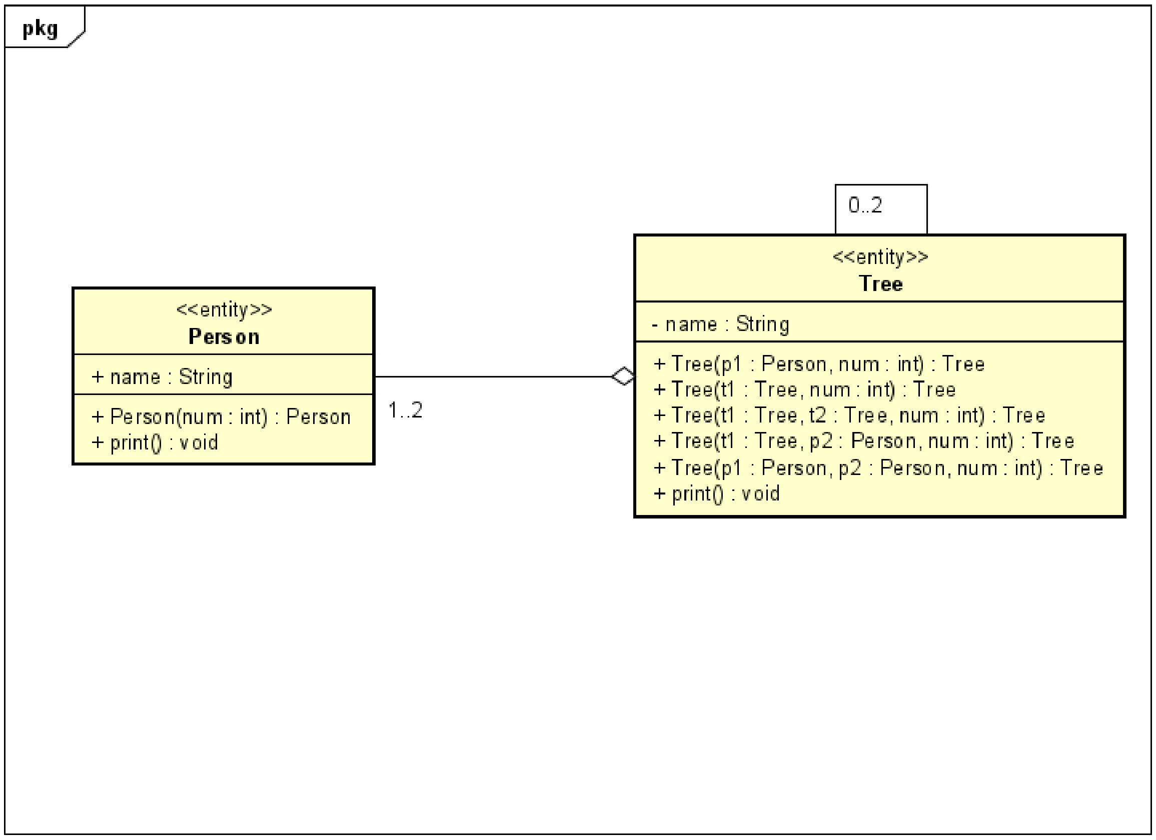

pkg
<<entity>>
Character
# name : String
# strength : int
# resilience : int
+ attackEnemy() : int
+ takeHit() : int
<<entity>>
Elf
+ Elf(name : String) : Elf
<<entity>>
Orc
+ Orc(name : String) : Orc
Given that an implementation already exists, I think it makes sense to include attributes and methods. The structure of the diagrams will still be clear.

pattern 2 After Design Pattern Integration

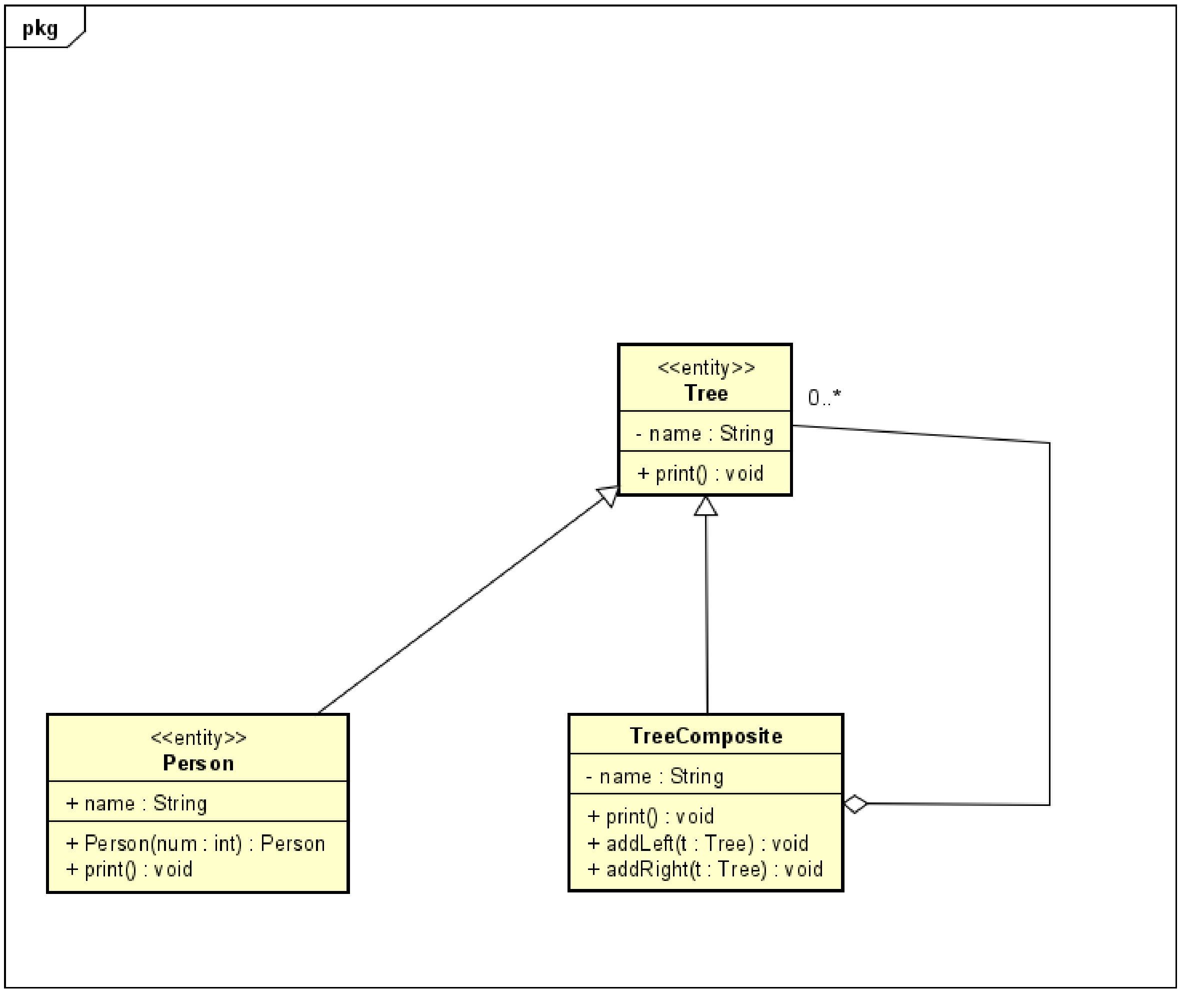

pkg
<<entity>>
Character
# name : String
# strength : int
# resilience : int
+ attackEnemy() : int
+ takeHit() : int
<<entity>>
Elf
+ Elf(name : String) : Elf
<<entity>>
Orc
+ Orc(name : String) : Orc
<<entity>>
Species
Given that an implementation already exists, I think it makes sense to include attributes and methods. The structure of the diagrams will still be clear.

For project pattern 1, I decided that the Composite design pattern would be most appropriate. Because the program is trying to generate family trees using a Tree class, which itself can contain additional Trees, a Composite would be the best suited design pattern for handling Tree classes and their Tree substructures. By treating child trees as a separate Component class, this would also help eliminate the need for several constructors for handling different Tree configurations.

pattern 1 Before Design Pattern Integration


pkg
0..2
<<entity>>
Tree
- name : String
+ Tree(p1 : Person, num : int) : Tree
+ Tree(t1 : Tree, num : int) : Tree
+ Tree(t1 : Tree, t2 : Tree, num : int) : Tree
+ Tree(t1 : Tree, p2 : Person, num : int) : Tree
+ Tree(p1 : Person, p2 : Person, num : int) : Tree
+ print() : void
<<entity>>
Person
+ name : String
+ Person(num : int) : Person
+ print() : void
1..2

pattern 1 After Design Pattern Integration



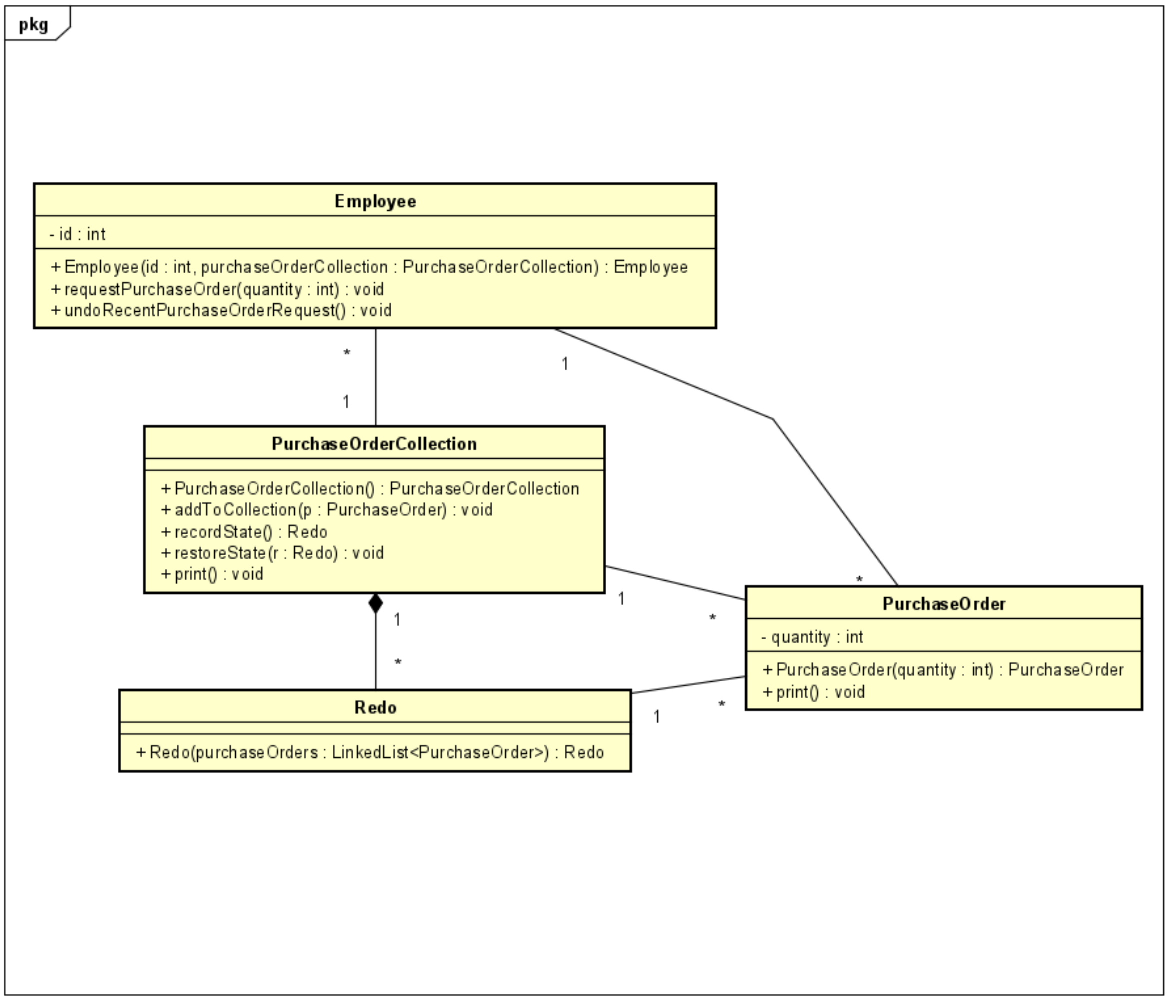
Task 3

Below is a UML class diagram of an implementation of the Memento design pattern based on some of the requirements for the Shop System. The Memento design pattern is a design that handles situations where the previous state of an object must be restored. This is done through an additional Memento class, which saves the state of the object being updated in its creation.

I've modified the requirements so that the system specifies only the need for an Employee to request any number of Purchase Orders. The additional requirement is that the system should be allowed to handle undoing the request. In order to handle this, the Memento design pattern is used to store the previous collection of Purchase Orders in the Purchase Order Collection before a new Purchase Order is added to the collection.

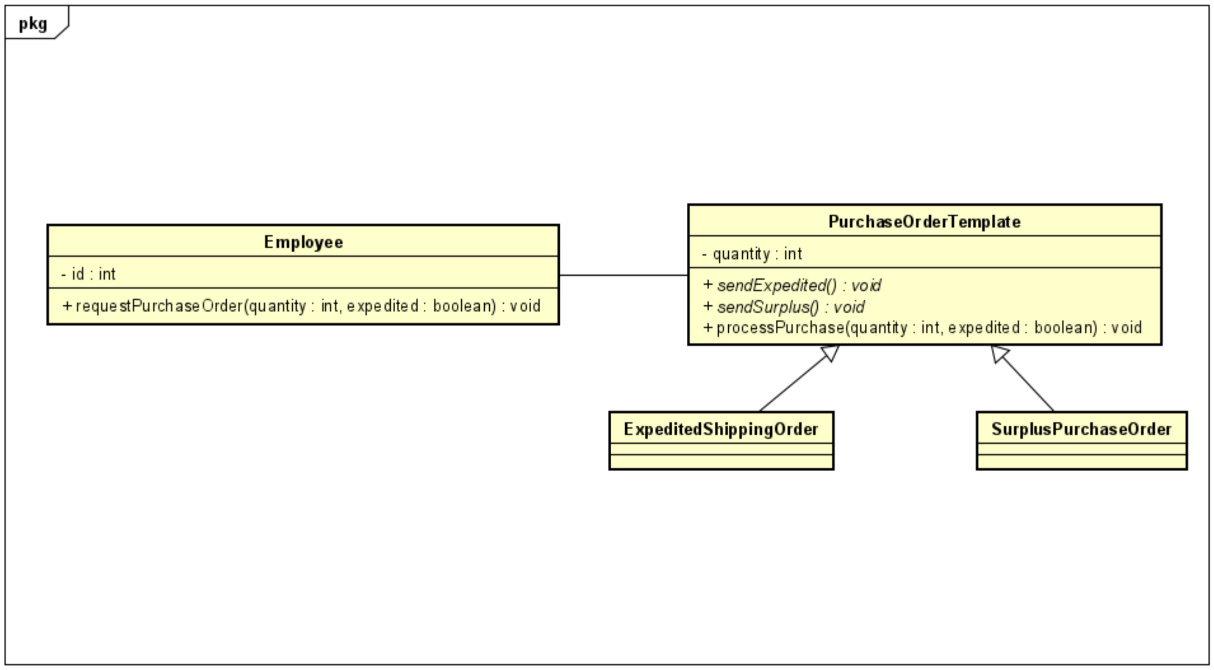
This design pattern acts as an effective and efficient means of restoring system state and is good in this context since it allows users of the system more flexibility in adjusting to mistakes made when using the system, like when accidentally requesting a Purchase Order.

Memento Design Pattern


pkg
Employee
- id : int
+ Employee(id : int, purchaseOrderCollection : PurchaseOrderCollection) : Employee
+ requestPurchaseOrder(quantity : int) : void
+ undoRecentPurchaseOrderRequest() : void
PurchaseOrderCollection
+ PurchaseOrderCollection() : PurchaseOrderCollection
+ addToCollection(p : PurchaseOrder) : void
+ recordState() : Redo
+ restoreState(r : Redo) : void
+ print() : void
PurchaseOrder
- quantity : int
+ PurchaseOrder(quantity : int) : PurchaseOrder
+ print() : void
Redo
+ Redo(purchaseOrders : LinkedList<PurchaseOrder>) : Redo
*
1
1
*
1
*
1
*
1
*

Below is a UML diagram of an implementation of the Task Method design pattern. Its function is to create an abstract class that helps implement subclasses that behave differently to each other while also providing an “invariant” or non-changing implementation of a function that is the same regardless of which subclass is being used. This helps reduce duplicate code when implementing objects with similar functions that behave differently while also providing a framework for additional objects to fall under an existing abstract class.

The implementation of the Template Method is again based on the Shop System. I modified the requirements so that they are concerned only with the Employee's ability to send purchase order requests. These purchase order requests can behave differently depending on whether the purchase order is expedited or if the quantity of the order is greater than 5. Because both types of orders are purchase orders, they both need to be processed. However, because one is expedited and the other is surplus, additional specifications must be made. Because there is an invariant circumstance in the circumstance of having both orders processed and a variant circumstance on what happens if the order is expedited or surplus, the Template Method design pattern is the perfect pattern to handle this situation. In this case, the pattern helps eliminate the need for the same invariant behavior to be defined multiple times across similar objects.


pkg
Employee
- id : int
+ requestPurchaseOrder(quantity : int, expedited : boolean) : void
PurchaseOrderTemplate
- quantity : int
+ sendExpedited() : void
+ sendSurplus() : void
+ processPurchase(quantity : int, expedited : boolean) : void
ExpeditedShippingOrder
SurplusPurchaseOrder