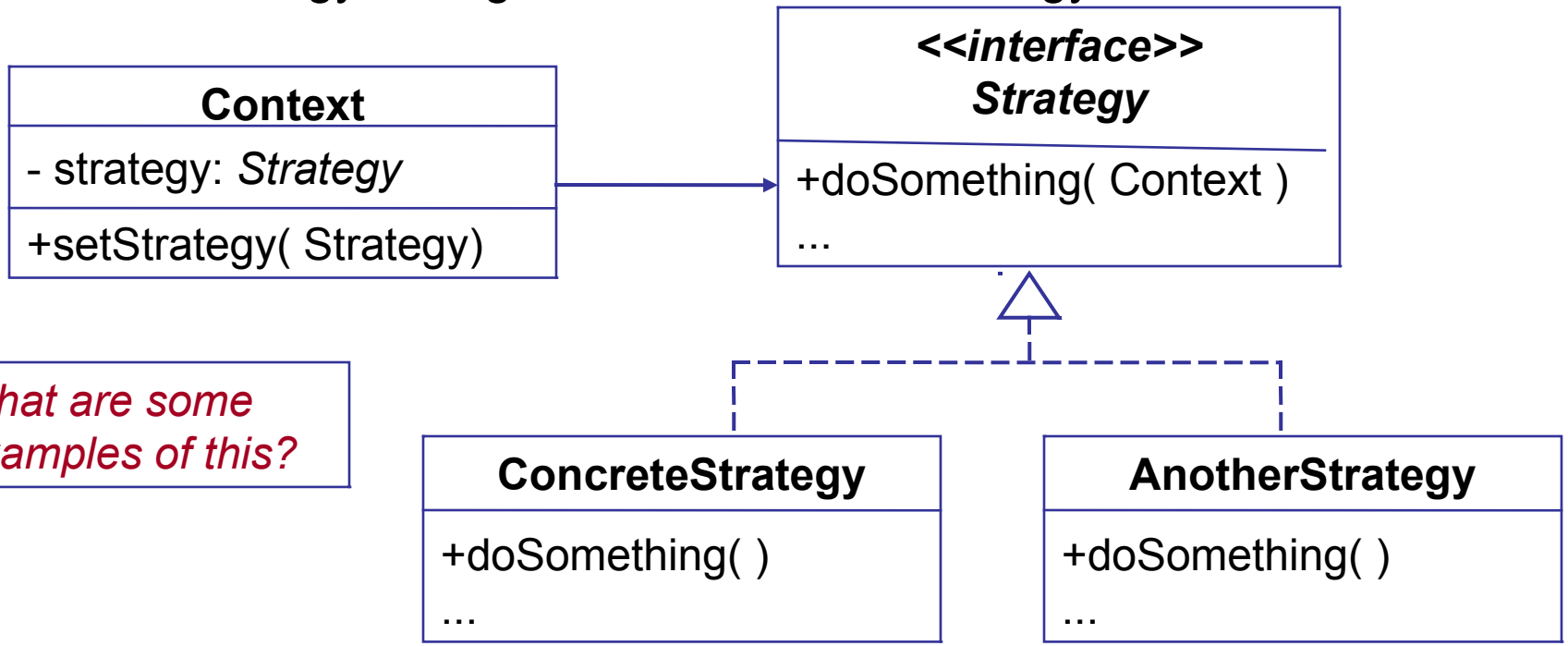
Strategy Pattern

Context: A class requires some behavior, but there are many ways that this behavior can be implemented.

Solution: implement the behavior in a separate class, called the *Strategy*.

Create a Strategy interface to de-couple the context class from the Strategy. Delegate the task to the strategy.

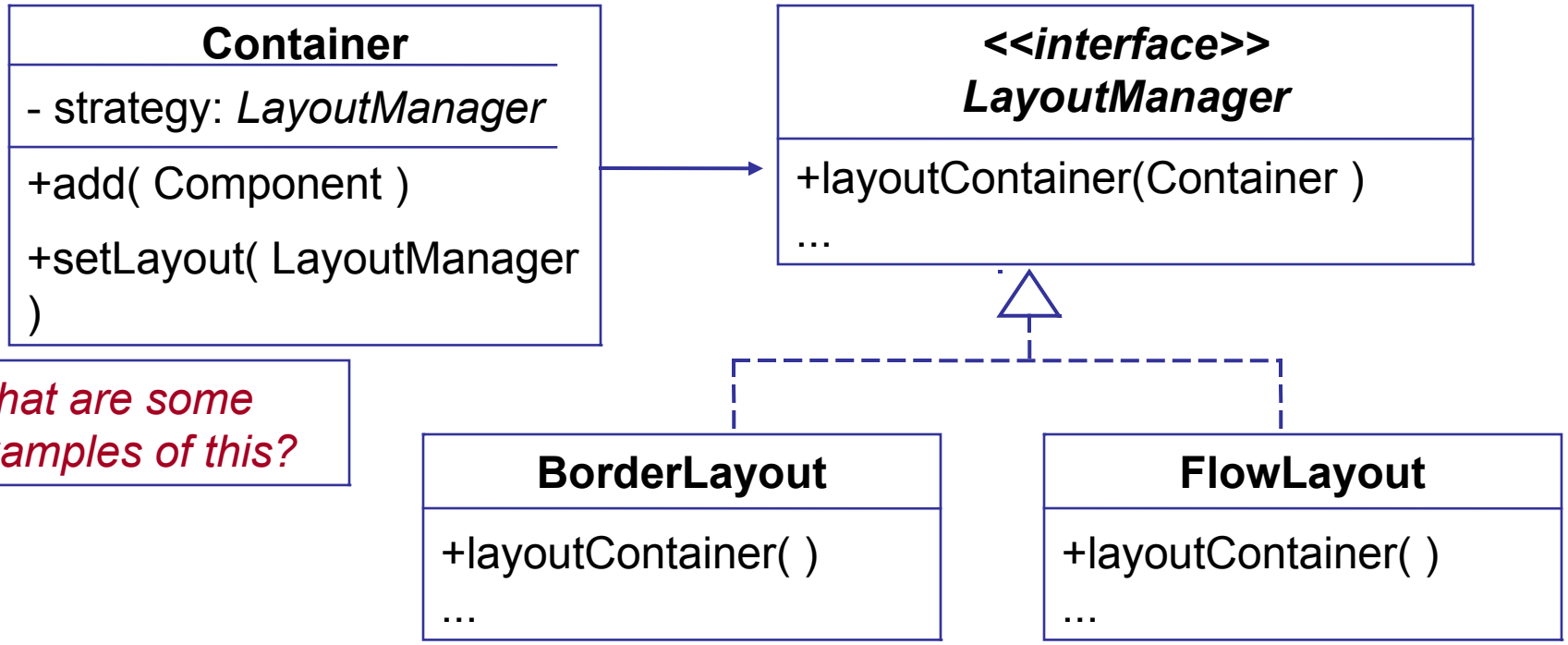


Container uses Strategy Pattern

Context: Swing container.

Strategy: LayoutManager.

Create a Strategy interface to de-couple the context class from the Strategy.

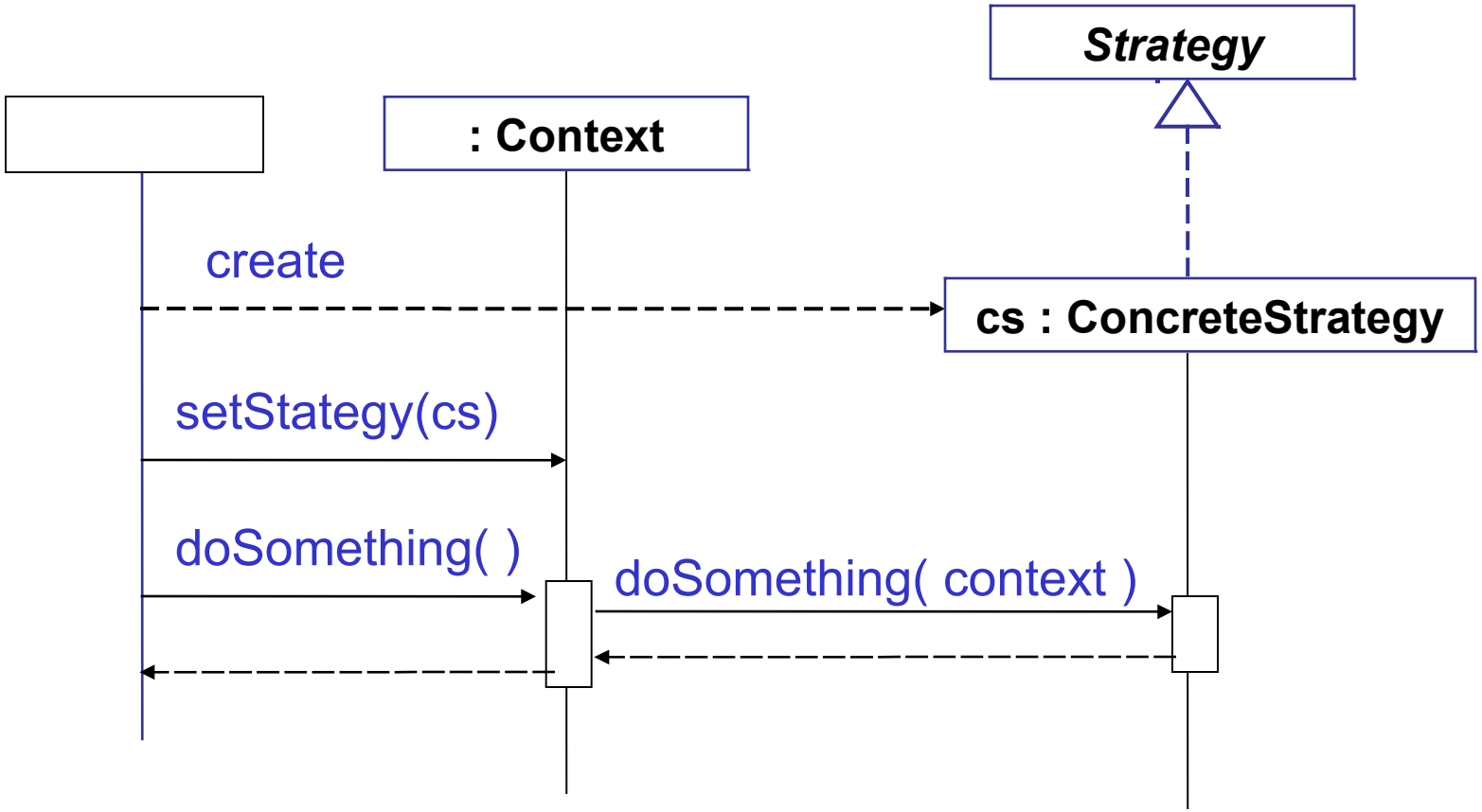


What are some examples of this?

Using the Strategy Pattern

(1) The application creates a concrete strategy and assigns it to the context.

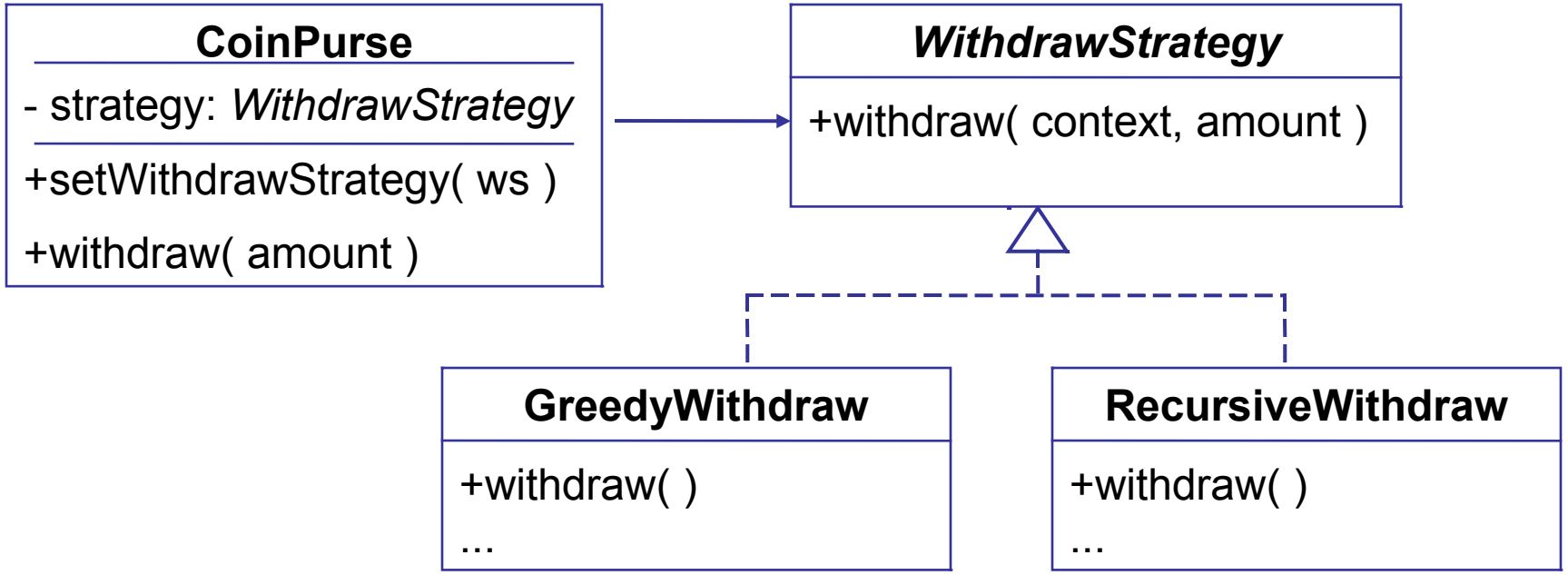
(2) The context delegates some work to the Strategy.



Strategy Pattern for Coin Purse

Context: A coin purse must decide what coins to withdraw; there are many ways to do this and we may want to change strategies.

Solution: Separate the withdraw( ) method from the Purse. Define a *WithdrawStrategy* interface for the withdraw operation, and modify the purse to *delegate* the withdraw operation to a concrete instance of *WithdrawStrategy*.



Strategy needs access to Context

To do its job, the Strategy usually needs a reference to the Context or some data of the Context.

Context: AWT/Swing Container (JPanel ...) contains components.

Strategy: A LayoutManager arranges and resizes components.

LayoutManger needs a reference to Container to get size and list of Components.