# Johns Hopkins Engineering

# Software Patterns

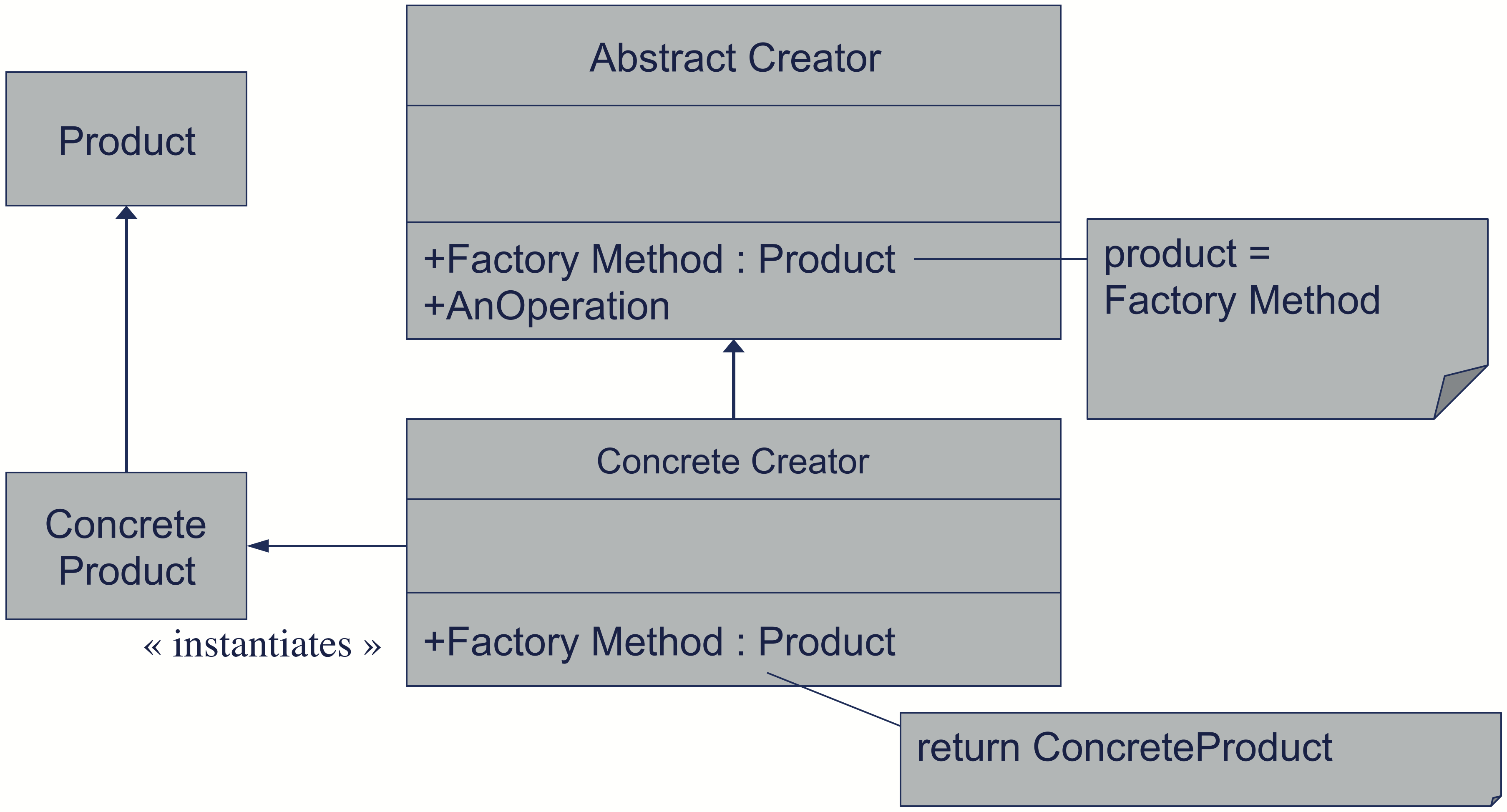
Module: Factory Method

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

## Factory Method

- Intent:
  - Define an interface for creating an object, but let subclasses decide which classes to instantiate. *Factory Method* lets a class defer instantiation.
- Opening Question:
  - How does *Factory Method* promote loosely coupled code?

# Factory Method



# Johns Hopkins Engineering

# Software Patterns

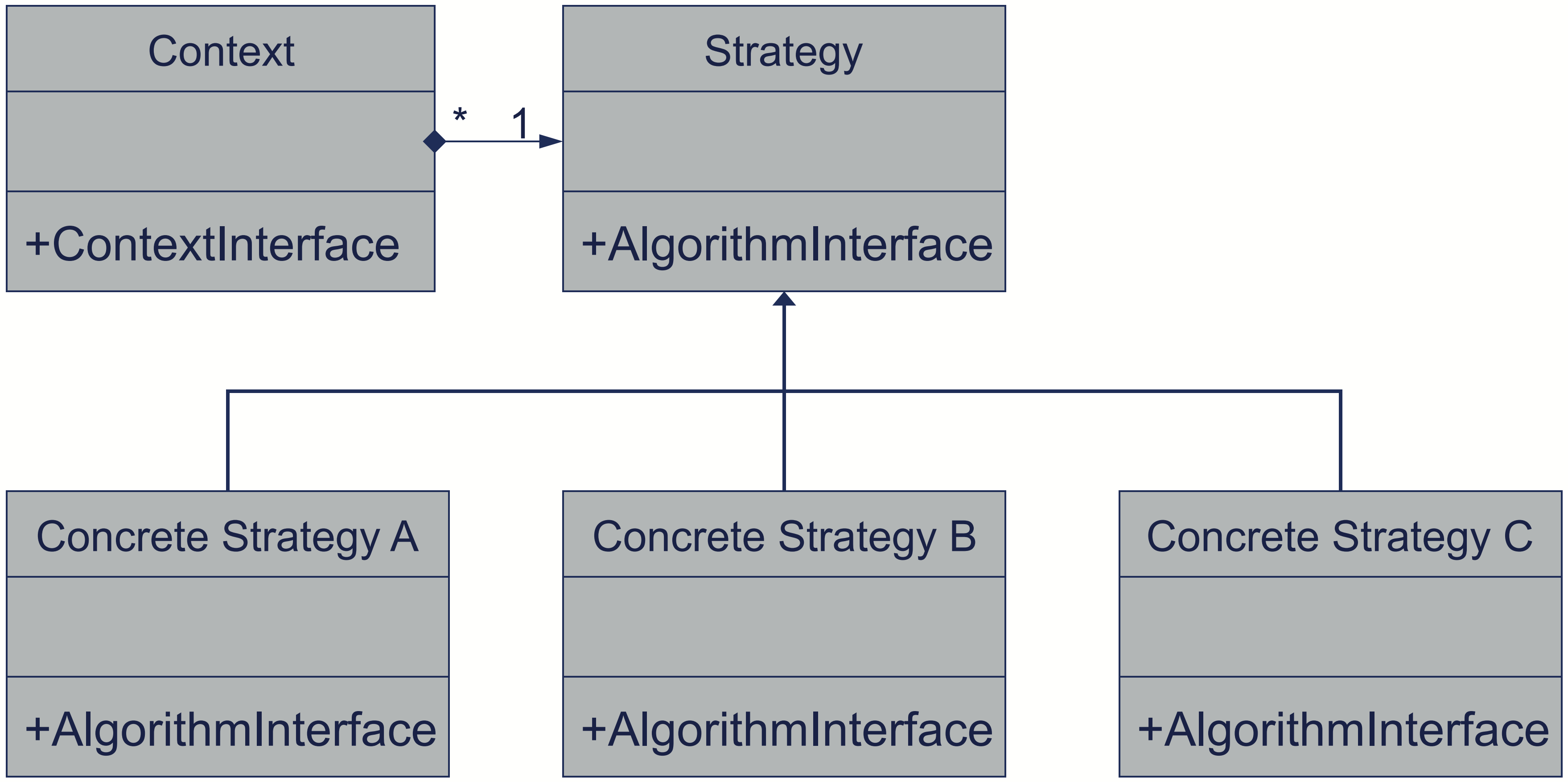
Module:  Strategy

JOHNS HOPKINS
WHITING SCHOOL
*of* ENGINEERING

# Strategy

- Intent:
  - Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Opening Question:
  - What happens when a system has an explosion of Strategy objects? Is there some better way to manage these strategies?
  - In the implementation section of this pattern, the authors describe two ways in which a strategy can get the information it needs to do its job. One way describes how the strategy could get passed a reference to the context object, thereby giving it access to the context data. But is it possible the data will not be available from the context's interface? How could you remedy this potential problem.

# Strategy


Context
+ContextInterface
*
1
Strategy
+AlgorithmInterface
Concrete Strategy A
+AlgorithmInterface
Concrete Strategy B
+AlgorithmInterface
Concrete Strategy C
+AlgorithmInterface

# Johns Hopkins Engineering

# Software Patterns

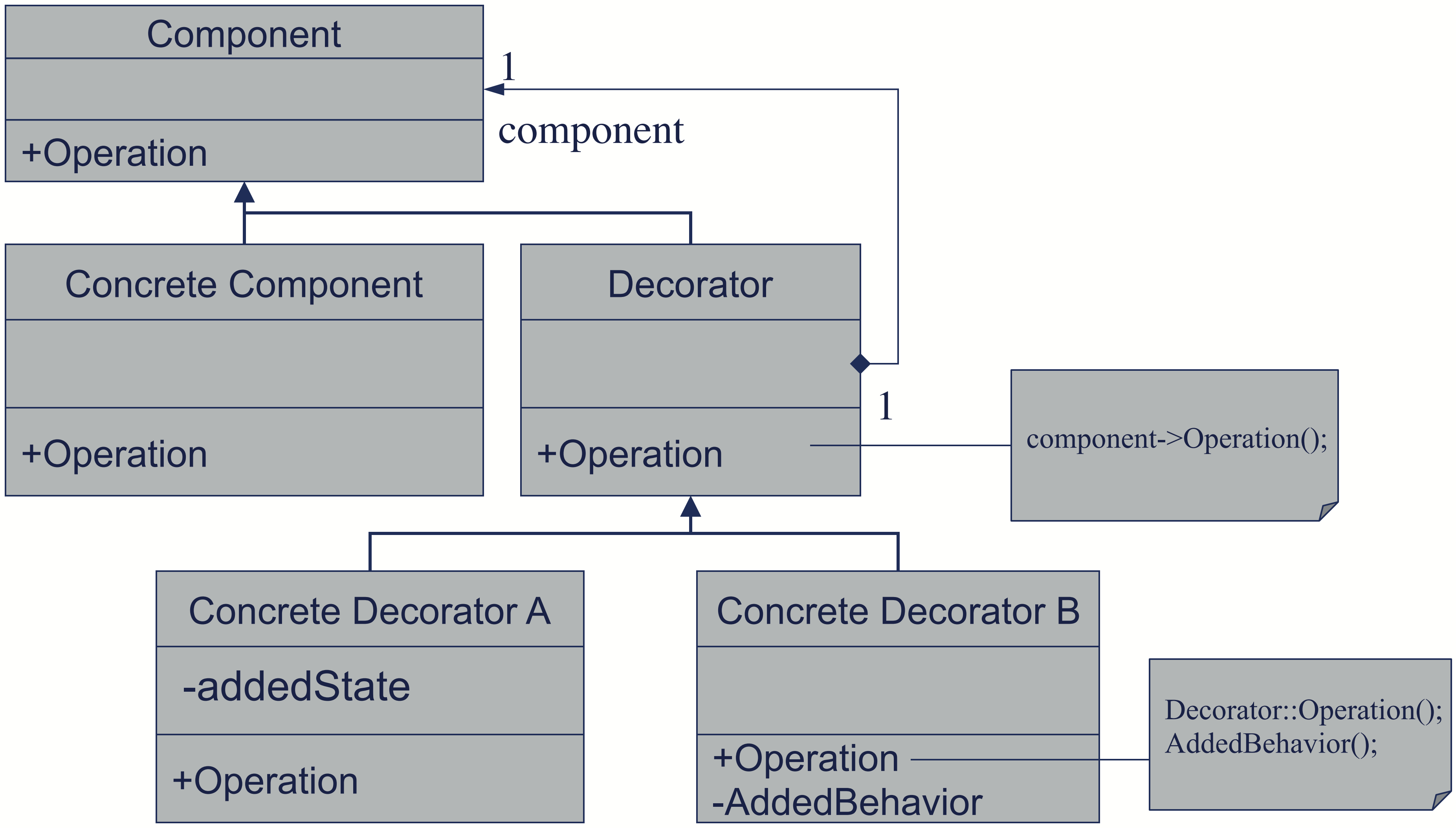
Module: Decorator

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

## Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Opening Questions
  - *A decorator object's interface must conform to the object that it decorates.* Now consider an object A that is decorated with an object B. Since object B "decorates" object A, object B shares an interface with object A. If some client is then passed an instance of this decorated object, and that method attempts to call a method on B that is not part of A's interface, does this mean that the object is no longer a Decorator in the strict sense of the pattern? Furthermore why is it important that a decorator object's interface conforms to the interface of the component that it decorates?

# Decorator



Component
+Operation
1
component
Concrete Component
+Operation
Decorator
+Operation
1
component->Operation();
Concrete Decorator A
-addedState
+Operation
Concrete Decorator B
+Operation
-AddedBehavior
Decorator::Operation();
AddedBehavior();