Design Patterns in Software Development

CSc3350

Dr. William Greg Johnson
Department of Computer Science
Georgia State University

Georgia State University | COLLEGE OF ARTS & SCIENCES

Design Patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Types of Patterns

- **Creational**: address problems of creating an object in a flexible way with separate creation, from operation/use
- **Structural**: address problems of using OOP constructs like inheritance to organize classes and objects into larger structures
- **Behavioral**: define the communication and interaction between objects

Georgia State University | COLLEGE OF ARTS & SCIENCES

Design Pattern Categories

Creational	Structural	Behavioral
Singleton	Adapter	Command
Factory Method	Bridge	Observer
Abstract Factory	Composite	Strategy
Builder	Façade	Template Method

Georgia State University | COLLEGE OF ARTS & SCIENCES

Design Problems Related to an Applied Pattern

1. Tell several objects that the state of some other object has changed (Observer pattern).
2. Clean up the interfaces to several related objects that have often been developed incrementally (Façade pattern).
3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
4. Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).
5. Dynamically choose the best "fit" algorithm at runtime (Strategy pattern.)

Georgia State University | COLLEGE OF ARTS & SCIENCES

Usage In Software Development

- **Singleton** pattern can be used to create a central logging object that can be accessed from anywhere in the application.
- **Factory Method** pattern can be used to create different types of database connections based on the configuration of the application.
- **Abstract Factory** pattern can be used to create families of related GUI widgets, such as buttons, text boxes, and labels, without having to specify the exact classes of the widgets that will be created.
- **Adapter** pattern can be used to allow two incompatible databases to work together.
- **Bridge** pattern can be used to decouple the implementation of a GUI widget from its abstraction, so that the widget can be easily plugged into different GUI frameworks.
- **Composite** pattern can be used to represent a tree structure of objects, such as a directory structure in a file system.

Usage In Software Development

- **Decorator** pattern can be used to add new functionality to existing GUI widgets without modifying their classes. For example, a decorator could be used to add a border to a button or a scrollbar to a text box.
- **Facade** pattern can be used to provide a unified interface to a complex set of database operations.
- **Command** pattern can be used to implement an undo/redo feature in an application.
- **Observer** pattern can be used to implement a publish/subscribe system, where objects can subscribe to notifications from other objects.
- **Strategy** pattern can be used to implement different sorting algorithms in an application.

Georgia State University | COLLEGE OF ARTS & SCIENCES

ALL Design Patterns Have 4 Elements

1. Name
   - A meaningful pattern identifier.
2. Problem description
3. Solution description
   - Not a concrete design but a template for a design solution that can be instantiated in different ways.
4. Consequences
   - The results and trade-offs of applying the pattern.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern

- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state are needed.
- Solution description
  - See slide with UML description.
- Consequences
  - Optimisations to enhance display performance are impractical.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern

Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

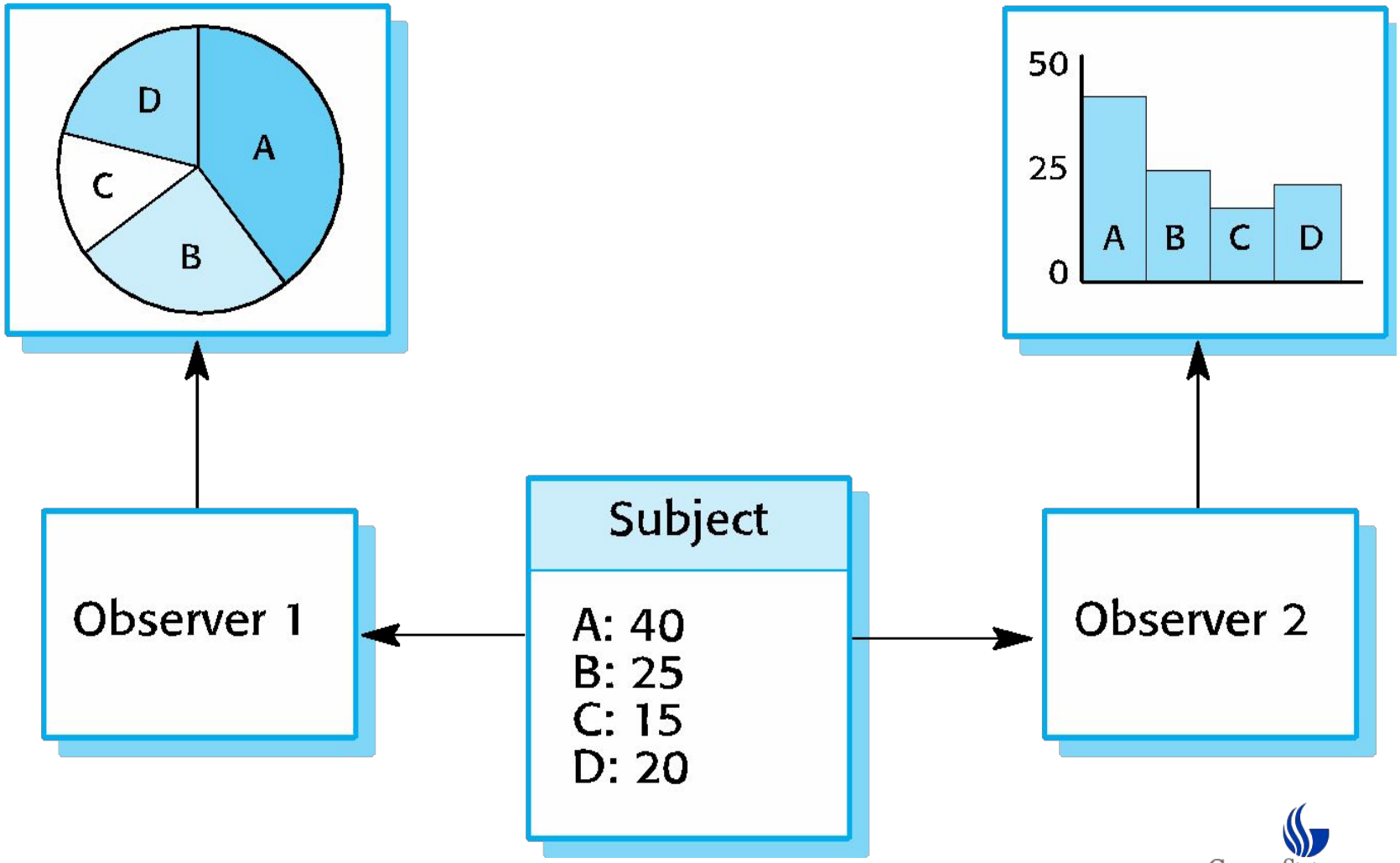
Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern (cont.)

Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

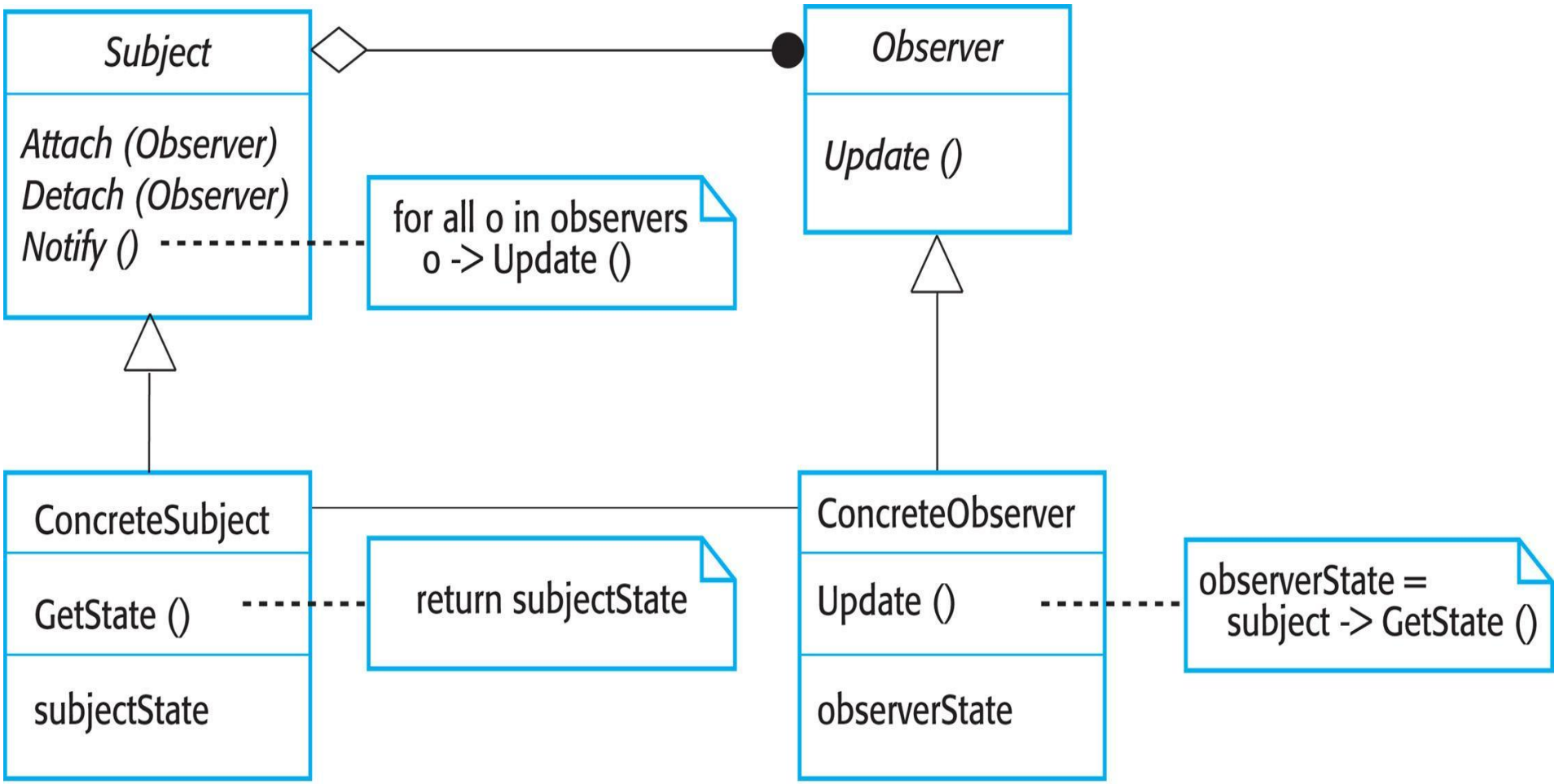
Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer: Multiple Displays of Same Data (Concrete Example)



Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern (UML)



Copyright ©2016 Pearson Education, All Rights Reserved

Observer Pattern

- **SubjectBase.** This is the abstract base class for concrete subjects. It contains a private collection of the observers that are subscribed to a subject and methods to allow new subscriptions to be added and existing ones to be removed. It also includes a method that can be called by concrete subjects to notify their observers of state changes. This Notify method loops through all of the registered observers, calling their Update methods.
- **ConcreteSubject.** Each concrete subject maintains its own state. When a change is made to that state, the object calls the base class's Notify method to indicate this to all of its observers. As the functionality of the observers is unknown, the concrete subjects also provide the means for the observers to read the updated state, in this case via a GetState method.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern

- **ObserverBase.** This is the abstract base class for all observers. It defines a method to be called when the subject's state changes. In many cases this Update method will be abstract, in which case you may decide to implement the base class as an interface instead.
- **ConcreteObserver.** The concrete observer objects are the subscribers that react to changes in the subject's state. When the Update method for an observer is called, it examines the subject to determine which information has changed. It can then take appropriate action.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern

- Intent is to establish a dependency (1..*) of objects so when one changes state, all dependents are notified and updated automatically.
- Use when:
  - Changing one object necessitates changing others, but it is unknown how many others exist.
  - You want to have loose coupling among the objects.
  - You want to have maximum flexibility of the system.
- Usage Examples:
  - Stock market
  - Conventional email delivery system
  - Central logging system
  - Ticket purchase system

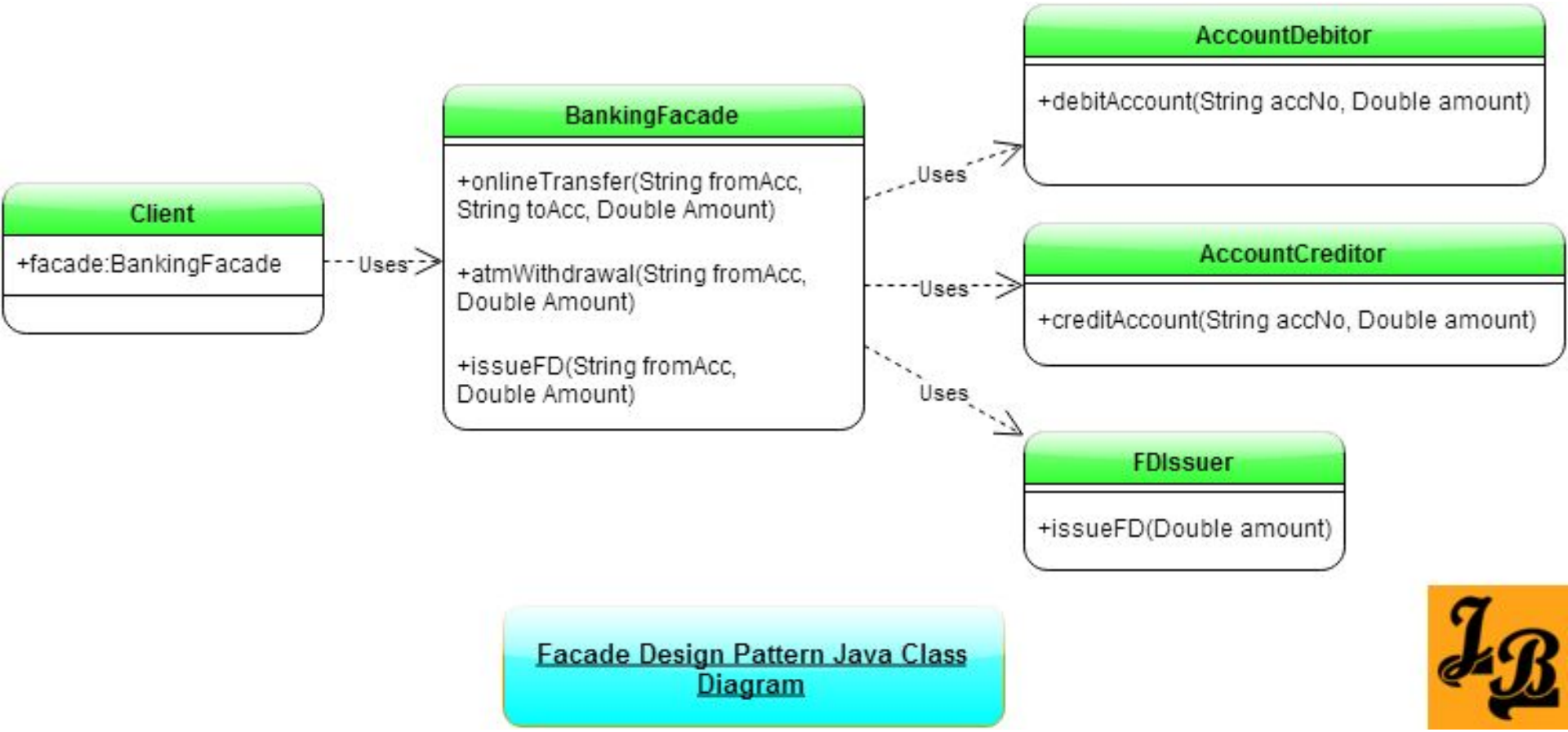
Georgia State University | COLLEGE OF ARTS & SCIENCES

Observer Pattern

- Pros
  - Supports loose coupling (low is good!)
  - Communicates to many objects very efficiently
  - No modifications required when new observers are added
  - Add and remove observers dynamically
- Cons
  - Can introduce unnecessary complexity
  - The order of updating to objects is unpredictiable

Georgia State University | COLLEGE OF ARTS & SCIENCES

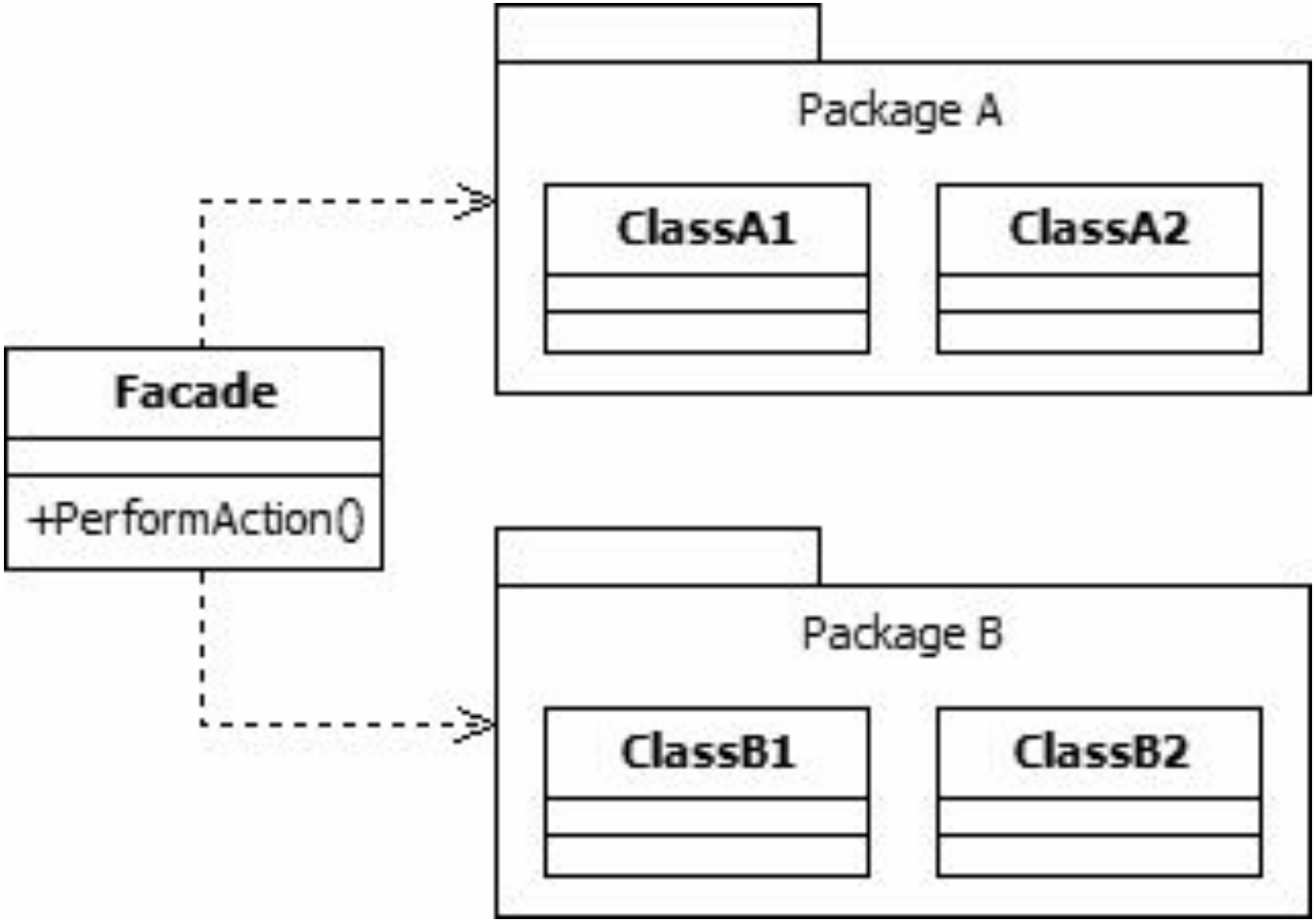
Façade Pattern: Banking System



https://www.javabrahman.com/design-patterns/facade-design-pattern-in-java/

Georgia State University | COLLEGE OF ARTS & SCIENCES

Façade Pattern (UML)



Georgia State University | COLLEGE OF ARTS & SCIENCES

Façade Pattern

- **Facade**. This class contains the set of simple functions that are made available to its users and that hide the complexities of the difficult-to-use subsystems.
- **Package (A/B)**. The complex functionality that is accessed via the facade class does not necessary reside in a single assembly. The packages in the diagram illustrate this, as each can be an assembly containing classes.
- **Class (A/B,1/2)**. These classes contain the functionality that is being presented via the facade.

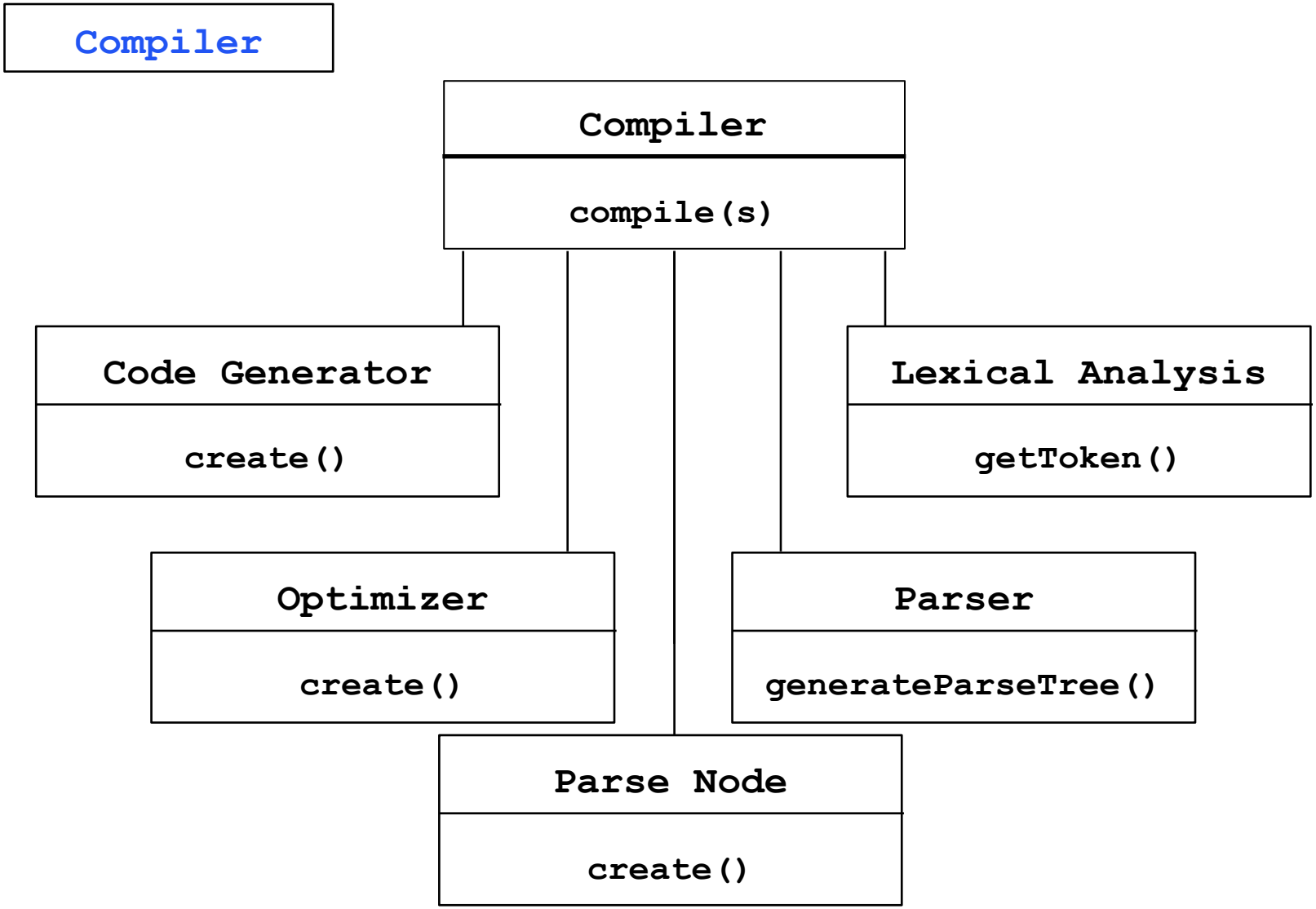
Georgia State University | COLLEGE OF ARTS & SCIENCES

Façade Pattern

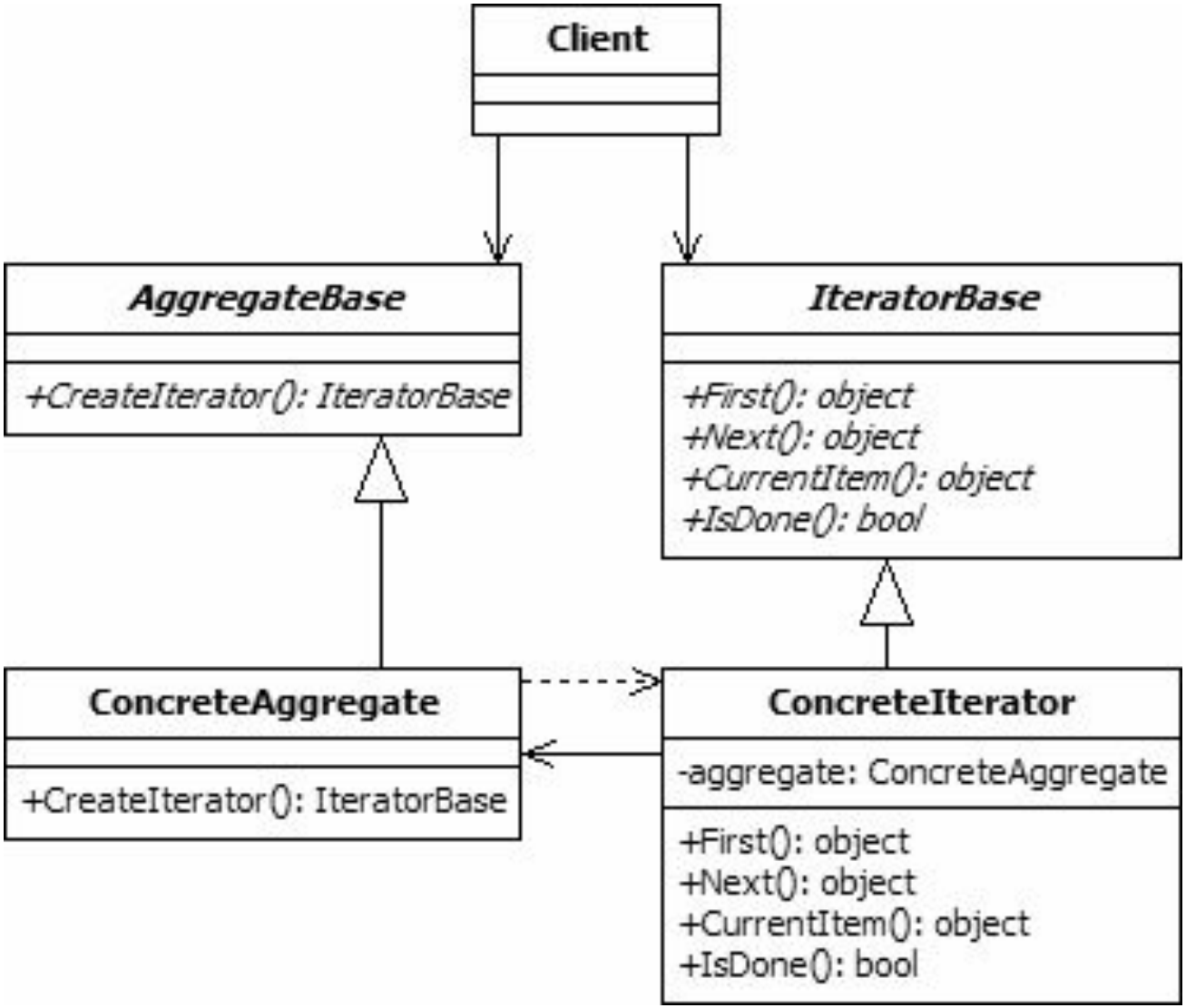
- Usage Examples:
  - Open Database Connectivity (ODBC)
  - Java Database Connectivity (JDBC)
  - Web services (RESTful API)
  - Programmatic access to complexity of legacy systems (mainframe, transaction processing)

Georgia State University | COLLEGE OF ARTS & SCIENCES

Realizing a compiler with a Façade Pattern



Iterator Pattern (UML)



Georgia State University | COLLEGE OF ARTS & SCIENCES

Iterator Pattern

- **Client**. Objects of this type are the consumers of the iterator design pattern. They request an iterator from an aggregate object when they wish to loop through the items that it holds. The methods of the iterator are then used to retrieve items from the aggregate in an appropriate sequence.
- **AggregateBase**. This abstract class is the base class for aggregate objects. It includes a method that generates an iterator, which can be used to obtain references to the objects that subclasses contain. This class is often implemented as an interface.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Iterator Pattern

- **ConcreteAggregate**. The concrete aggregate classes provide the real functionality for aggregate objects that contain collections of items that can be traversed using an iterator.
- **IteratorBase**. This abstract class is the base class for iterators. It defines a standard interface that includes methods to allow the elements of the aggregate that generated it to be looped through in sequence. This class is often implemented as a simple interface.
- **ConcreteIterator**. Concrete iterators implement the interface defined by the IteratorBase class. They provide functionality specific to the ConcreteAggregate class used to generate them, hiding the implementation of the aggregate from the client.

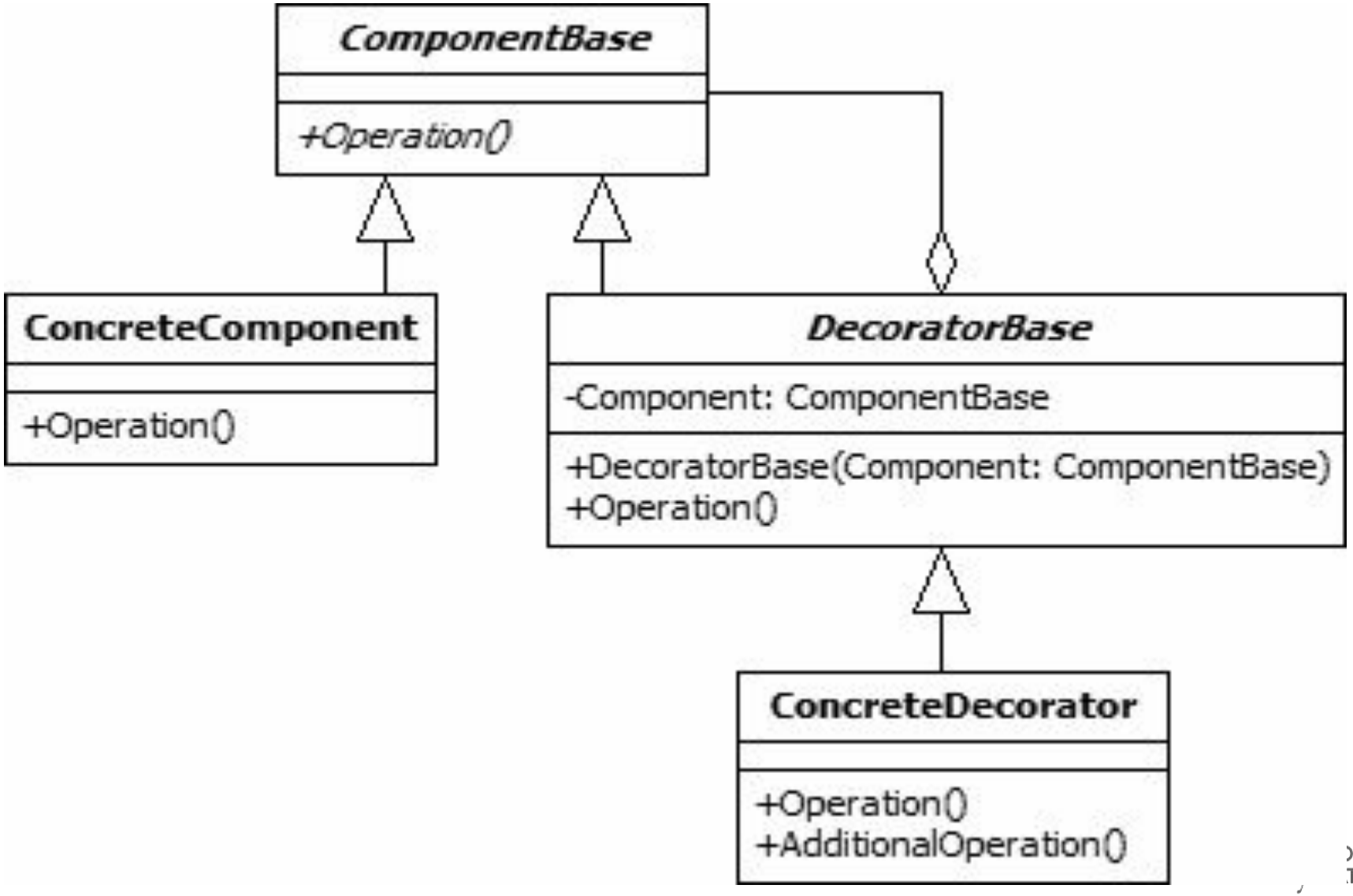
Georgia State University | COLLEGE OF ARTS & SCIENCES

Iterator Pattern

- Usage Examples:
  - Online banking (multiple account types, one source)
  - Online real estate (multiple properties, one seller)
  - File / directory in operating system
  - Classic java: "Scanner"

Georgia State University | COLLEGE OF ARTS & SCIENCES

Decorator Pattern (UML)



Decorator Pattern

- **ComponentBase**. This abstract class is the base class for both the concrete components and all decorator classes. The base class defines any standard members that will be implemented by these classes. If you do not wish to create any actual functionality in this class you may decide to create it as an interface instead.
- **ConcreteComponent**. The ConcreteComponent class inherits form the ComponentBase class. There may be multiple concrete component classes, each defining a type of object that may be wrapped by the decorators.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Decorator Pattern

- **ConcreteComponent**. The ConcreteComponent class inherits form the ComponentBase class. There may be multiple concrete component classes, each defining a type of object that may be wrapped by the decorators.
- **DecoratorBase**. This abstract class is the base class for all decorators for components. The class inherits its public interface from ComponentBase so that decorators can be used in place of concrete objects. It adds a constructor that accepts a ComponentBase object as its parameter. The passed object is the component that will be wrapped. As the wrapped object must inherit from ComponentBase, it may be a concrete component or another decorator. This allows for multiple decorators to be applied to a single object. When calls to the DecoratorBase's members are made, these are passed unmodified to the matching member of the wrapped object.

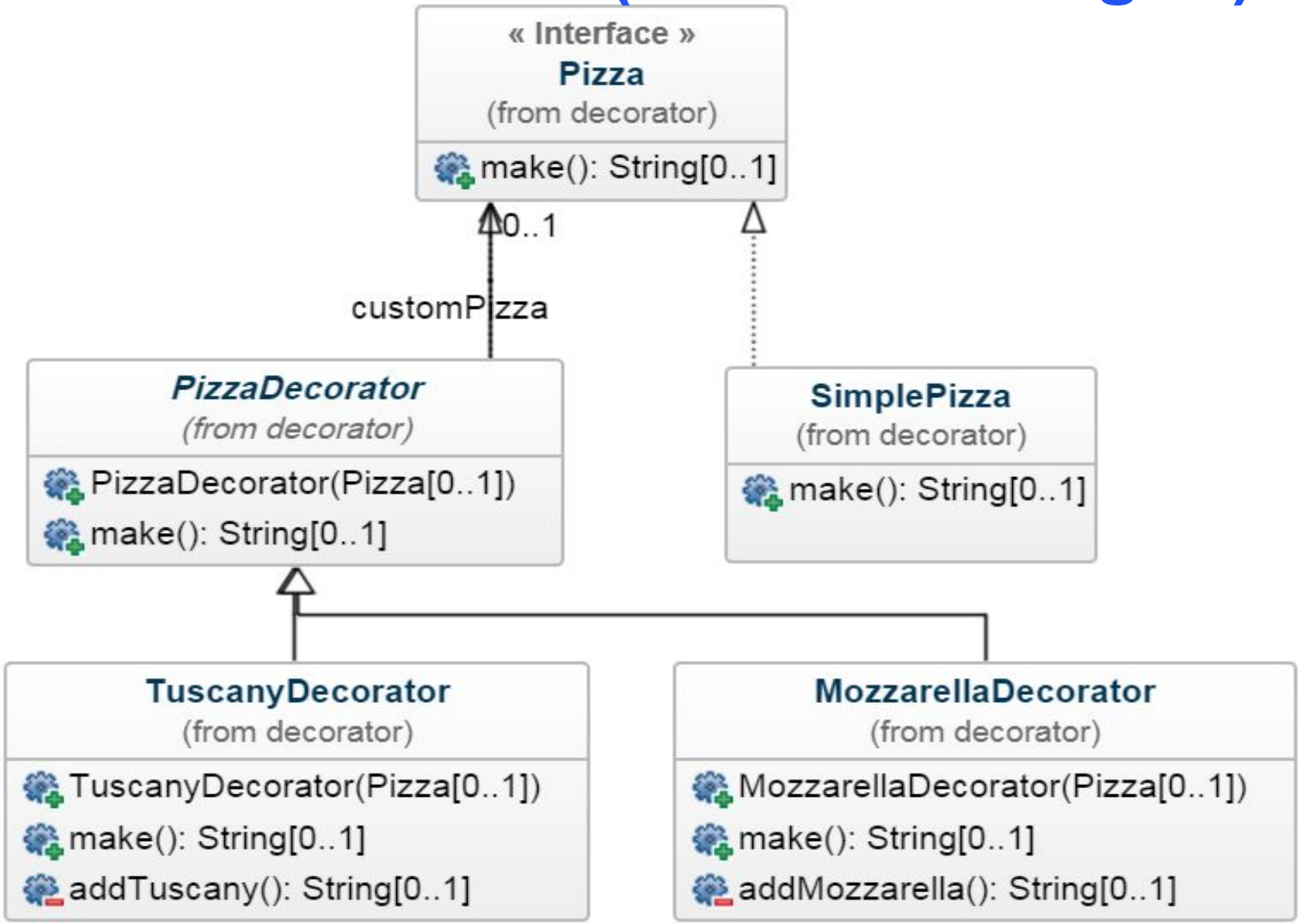
Georgia State University | COLLEGE OF ARTS & SCIENCES

Decorator Pattern

- **ConcreteDecorator**. The ConcreteDecorator class provides a decorator for components. In the diagram, an additional method has been included in the decorator. The Operation member can be implemented in two manners. Firstly, the operation may be unchanged. In this case, the implementation simply calls the base method. Secondly, the underlying operation may be modified or replaced completely. In this case, new functionality will be added, which may or may not call the base method.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Decorator Pattern (more meaningful)



https://nobugsproject.com/2017/09/24/design-patterns-saga-17-real-project-situations-with-decorator/

Decorator Pattern

- Usage Examples:
  - Online car customization app
  - An application that allows the user to dynamically build and /or change something (runtime vs compile time)
  - Online shopping cart

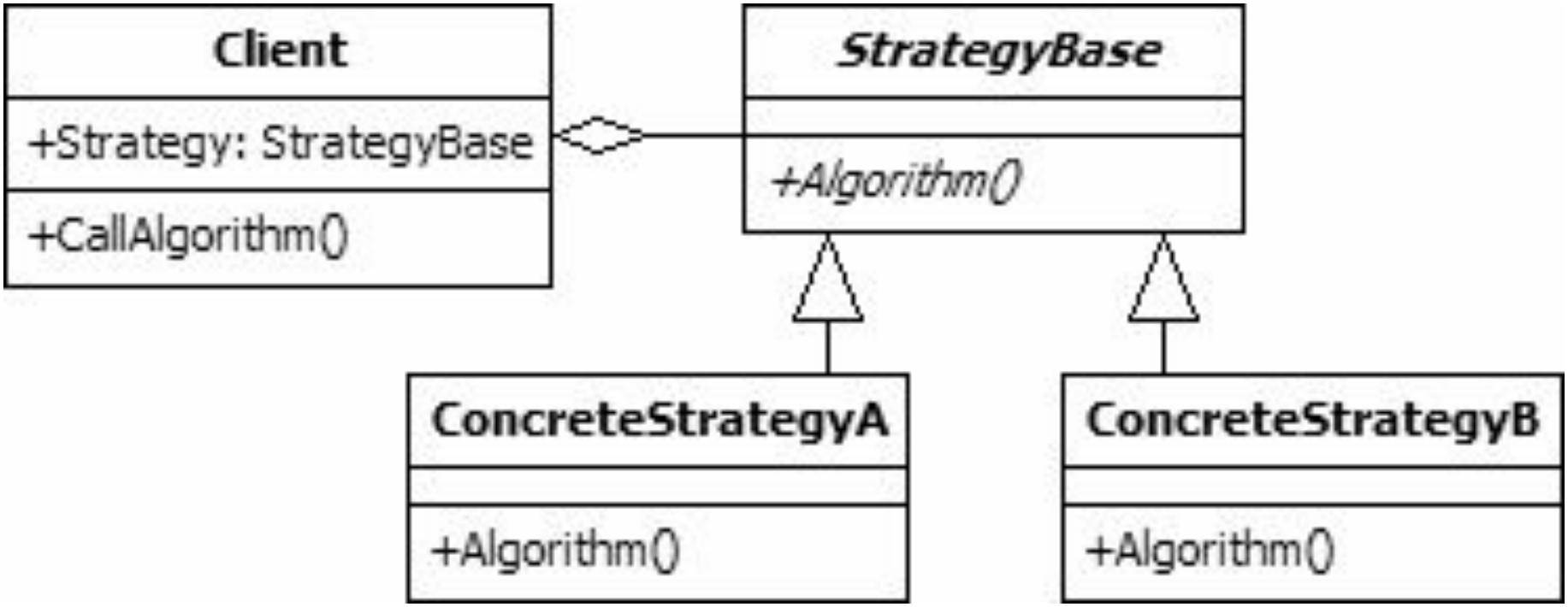
Georgia State University | COLLEGE OF ARTS & SCIENCES

Strategy Pattern

- Intent is to establish a dynamic variation of an algorithm
- Use when:
  - Several classes that are related behave differently.
  - A class design has multiple conditional statements for operations that produce a singular outcome. Move the condititon branches into their own Strategy classes.
  - You want to have runtime flexibility of a component in a larger system.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Strategy Pattern (UML)



Georgia State University | COLLEGE OF ARTS & SCIENCES

Strategy Pattern

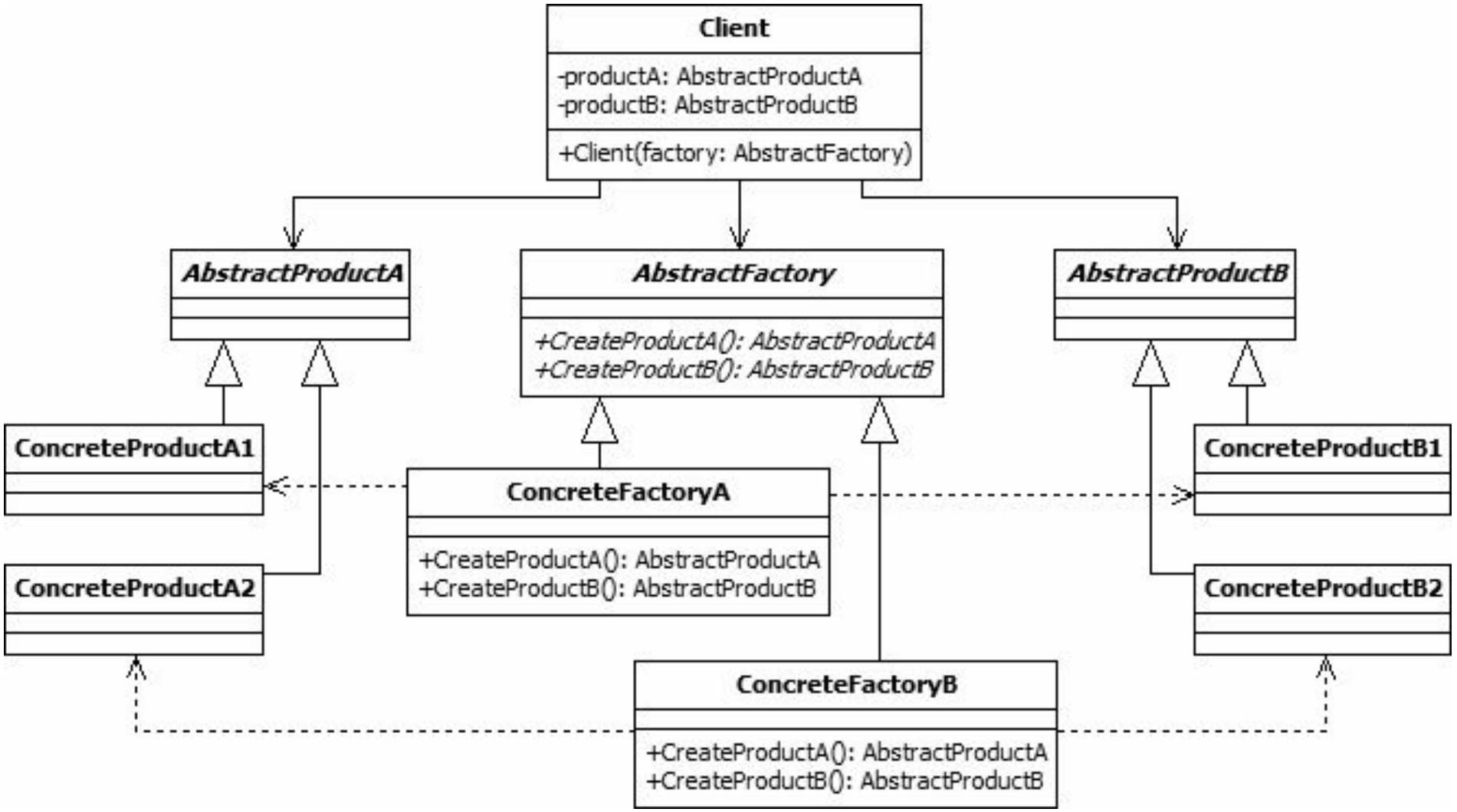
- **Client**. This class is the user of an interchangeable algorithm. The class includes a property to hold one of the strategy classes. This property will be set at run-time according to the algorithm that is required.
- **StrategyBase**. This abstract class is the base class for all classes that provide algorithms. In the diagram the class includes a single method. However, there is no reason why a number of properties and methods may not be included. This class may be implemented as an interface if it provides no real functionality for its subclasses.
- **ConcreteStrategy A/B**. The concrete strategy classes inherit from the StrategyBase class. Each provides a different algorithm that may be used by the client.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Strategy Pattern

- Usage Examples:
  - Online payment system
  - File compression application
  - IDE application for multiple file types

Abstract Factory (UML)



Georgia State University | COLLEGE OF ARTS & SCIENCES

Abstract Factory Pattern

- **Client**. This class uses the factories to generate a family of related objects. In the UML diagram, the client has two private fields that hold instances of the abstract product classes. When a client is instantiated, a concrete factory object is passed to the constructor and used to populate the private fields.
- **AbstractFactory**. This is an abstract base class for the concrete factory classes that will generate new sets of related objects. A method is included for each type of object that will be instantiated. Generally, an abstract class will be used so that other standard functionality can be included and inherited by all the concrete factory subclasses.
- **ConcreteFactory A/B**. Inheriting from the AbstractFactory class, this class overrides the methods that generate the objects required by the client. There may be many, depending on the design of the software system.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Abstract Factory Pattern

- **AbstractProduct (A/B)**. This abstract class is the base class for the types of object that a factory can create. One base typoe exists for each of the distinct types of product required by the client.
- **ConcreteProduct (A/B)**. Multiple subclasses of the ConcreteProduct classes are defined, each with a specific functionality. These classes are generated by the AbstractFactory to populate the client.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Abstract Factory Pattern

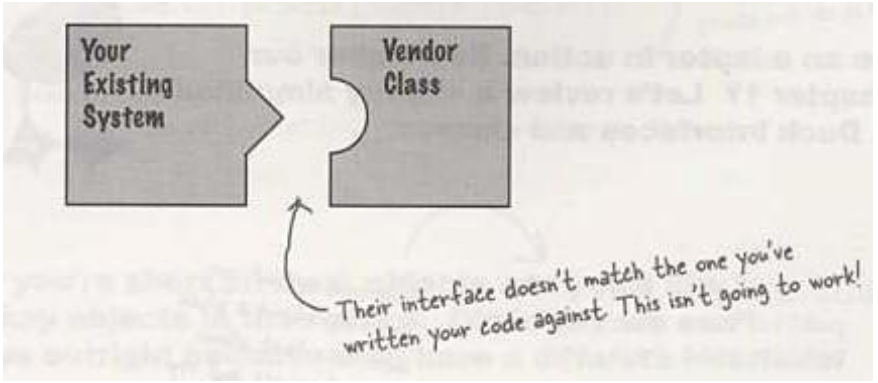
- Usage Examples:
  - Offering software that automatically runs on multiple platforms
  - Online purchasing/shipping system for different levels of product

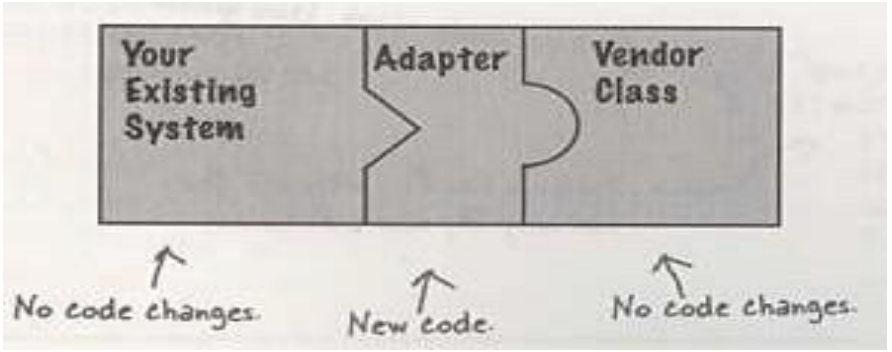
Georgia State University | COLLEGE OF ARTS & SCIENCES

Pattern: Adapter

- Suppose a client objects expect a certain interface to be provided by called object.
- However, the interface of the called object differs from what the clients expect, and cannot be changed.
- How may this situation be improved, i.e. how may the interface satisfy the clients?

Georgia State University | COLLEGE OF ARTS & SCIENCES





Transforms the interface of the target class into another interface that the client classes expect.

Pattern: Adapter

- Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- Used to provide a new interface to existing legacy (i.e. old) software components (aka interface engineering, re-engineering).
- Adapter also known as a wrapper

Georgia State University | COLLEGE OF ARTS & SCIENCES

Pattern: Adapter

Context:

1. Want to use an existing class without modifying it – call it the “legacy” class
2. But context in which you want to use the legacy class requires conformance to a target (i.e. new) interface that is different from that which the legacy provides.
3. However, the target interface and legacy interface are conceptually related.

Solution:

1. Define an adapter class that implements the target interface.
2. The adapter class holds a reference (delegates) to the legacy class. It translates (new) target requests (or method calls) from client classes to (old) legacy requests (or method calls).

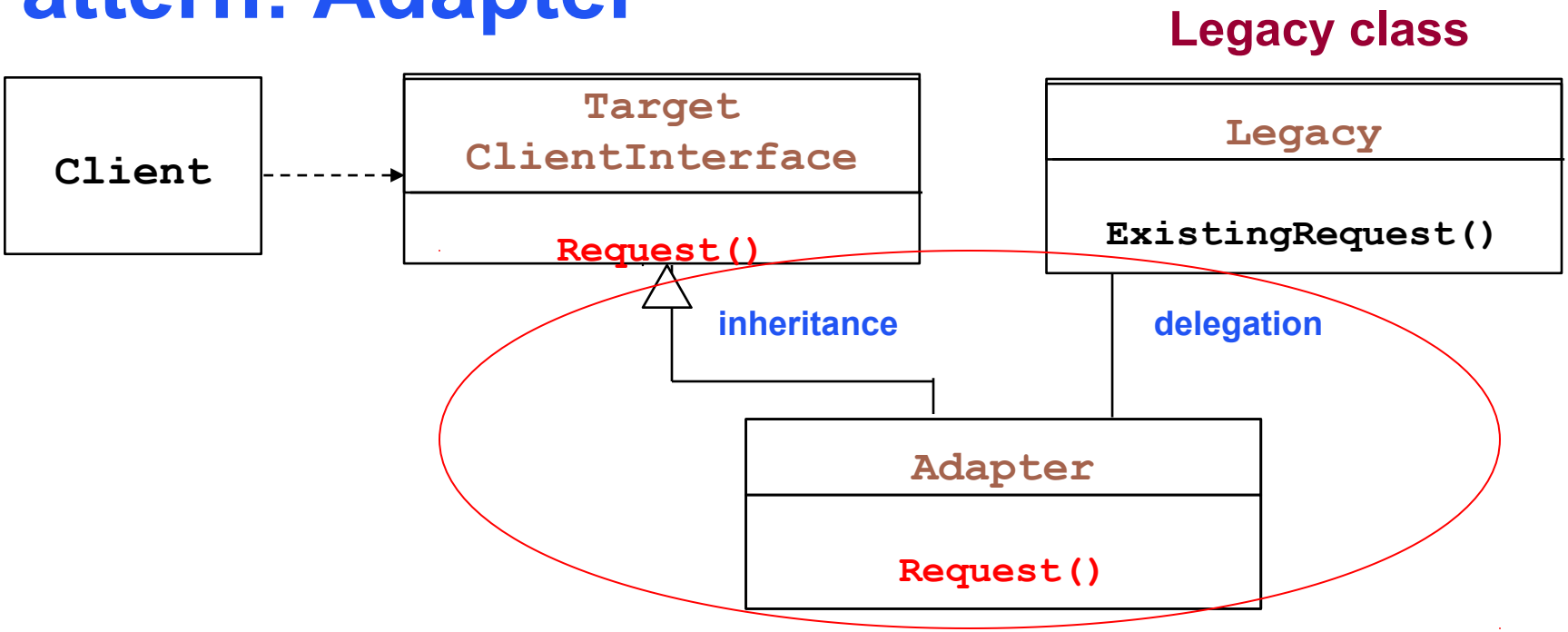
Georgia State University | COLLEGE OF ARTS & SCIENCES

Pattern: Adapter
Participants of the pattern.

- Target: Defines the application-specific interface that clients use.
- Client: Collaborates with objects conforming to the target interface.
- Legacy: Defines an existing interface that needs adapting.
- Adapter: Adapts the interface of the legacy class to the target interface.

Georgia State University | COLLEGE OF ARTS & SCIENCES

Pattern: Adapter



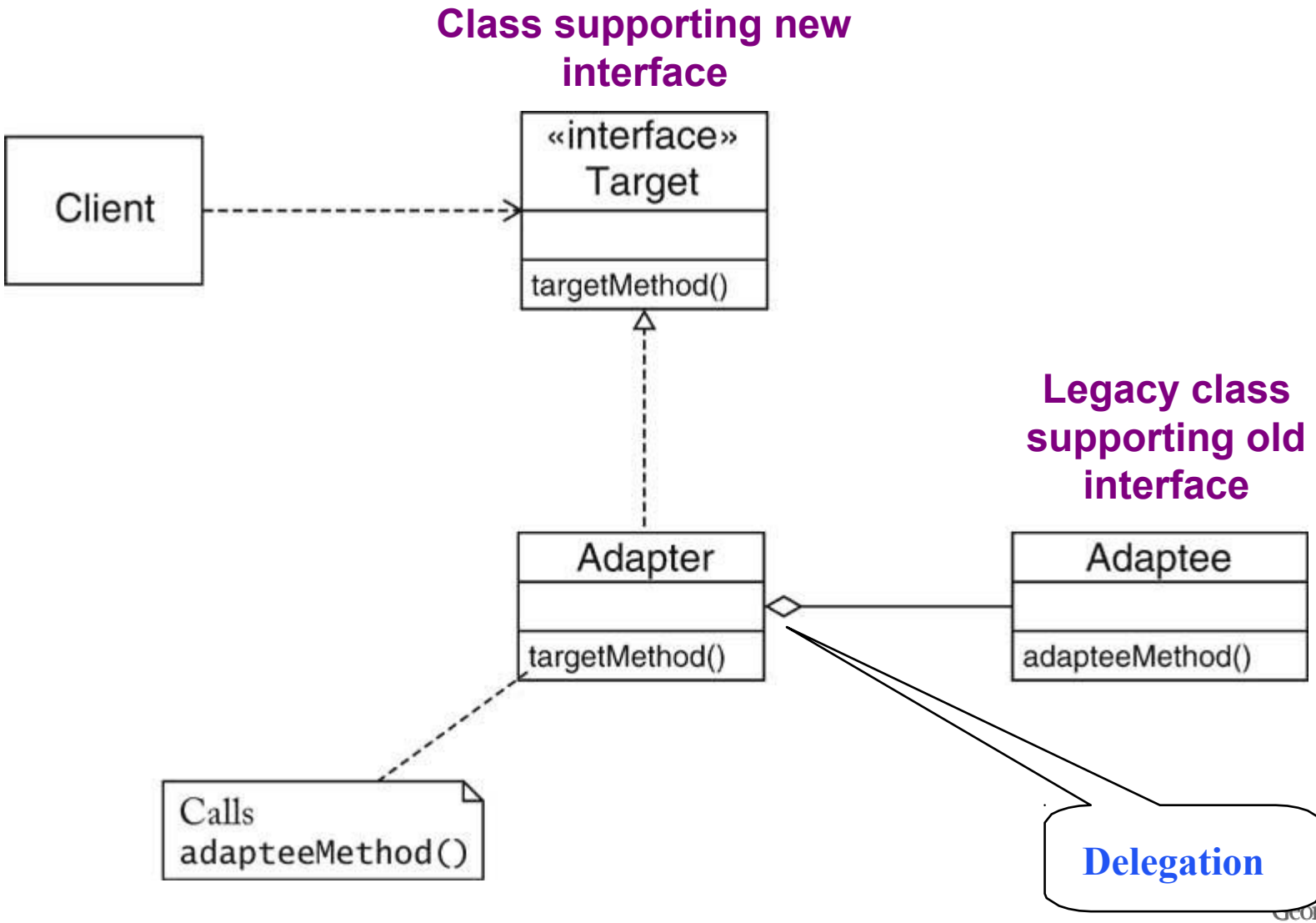
Bind an `Adapter` class with the `Legacy` class

The Adapter class implements the `ClientInterface` expected by the client. It then delegates requests from the client to the `Legacy` class and performs any necessary conversion.

`ClientInterface` could be a Java/C# `interface`, or an `abstract` class

Georgia State University | COLLEGE OF ARTS & SCIENCES

Interface inheritance is use to specify the interface of the Adapter class – and then delegation is used to reference the Legacy class



Georgia State University | COLLEGE OF ARTS & SCIENCES

Example

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter…

```
public interface Duck
{
    public void quack();
    public void fly();
}


public interface Turkey
{
    public void gobble();
    public void fly();
}
```

Turkeys do not quack, they gobble. Turkeys can fly but for only short distances.

How can we adapt the `Turkey` class to behave like a `Duck` class?

Georgia State University | COLLEGE OF ARTS & SCIENCES

Adapt the `Turkey` class to have the same interface as `Duck`.

i.e. have methods `quack()` and `fly()`

```
public class TurkeyAdapter implements Duck
{
    private Turkey _turkey;
    public TurkeyAdapter( Turkey turkey )
    {
        _turkey = turkey;
    }
    public void quack()
    {
        _turkey.gobble();
    }
    public void fly()
    {
        for( int i = 0; i < 5; i++ )
        {
            _turkey.fly();
        }
    }
}
```

duck quack becomes turkey gobble

duck fly becomes five times turkey fly

Georgia State University | COLLEGE OF ARTS & SCIENCES

Attributions

1. Pankhuree Srivastava, https://www.slideshare.net/pankhureesrivastava/software-design-patterns