Design Patterns Using OpenEdge

Mauricio dos Santos
Principal Consultant
Progress Software Corp.
Professional Services

Audience for this course

This course is for experienced ABL developers who have a good understanding of Object Oriented development and want to achieve a higher level of software quality from design to development, maintenance and deployment.

Course prerequisites

Before you begin this course, you should have experience with:

- Basic Object Oriented understanding from a developer's point of view
- Writing ABL code to:
  - Create simple abstract classes and interfaces
  - Create simple concrete classes
- Developer Studio for OpenEdge to create projects, procedures, classes and edit ABL code.
- Progress Application server for OpenEdge (PAS for OpenEdge)
  - Modifying a launch configuration
  - Starting and stopping a PAS for OpenEdge instance

Introduce yourself

- Your name and your job.
- Name of your company and its type of business.
- Your technical background.
- Any prior experience with Progress Software products?
- What would you like to learn from this course?

Learning objectives for this course

After taking this class, you should be able to:

- Understand and explain each design principle presented.
- Have a more profound and concise understanding of key object oriented concepts.

System and software requirements for this course

- Hardware/platform
  - Windows 64 bit
  - 2 Gb RAM
  - 100 Mb disk space for course files
  - 6 Gb disk space for Progress OpenEdge install files and installation
- Progress Developer Studio for OpenEdge 11.7 or later
  - Must install 64 bit version of Developer Studio with one of:
    - Developer Studio for OpenEdge license
    - OpenEdge Developer Kit (OEDK) Classroom Edition (no license required)
- See **ExerciseSetup** for instructions to follow before you begin the exercises of this course

Agenda

- Key Object Oriented Concepts
- Why Design Patterns?
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Key Object Oriented Concepts

- Classes, Subclasses, Types and Subtypes
- Interfaces, Abstract and Concrete Classes
- Abstraction and Implementation
- Inheritance and Composition
- Encapsulation: members can be private, protected, public, static
- Dynamic Binding and Polymorphism (LSP)
- S <: T
- Object Oriented Design

What Are Design Patterns?

- Proven ways of designing and developing object oriented applications so that they become highly reusable and flexible
- Based on design principles they favor composition over inheritance, try to maximize OCP, SRP and all the others
- Made possible mostly since S <: T
- Class patterns vs. object patterns
  - Class patterns use inheritance
  - Object patterns rely on composition

Why Design Patterns?

- Promote communication
- Stimulate new ideas
- Education (everybody else is doing it)
- Achieve higher levels of:
  - Encapsulation
  - Granularity
  - Dependency (less, that is)
  - Flexibility
  - Performance
  - Evolution
  - Reusability
  - Maintenance
  - Deployment (release process)

Types of Design Patterns

- Creational
  - Facilitate, or otherwise automate or abstract, the creation (or instantiation) of objects for a given purpose or application
- Structural
  - Use classes and objects to create certain structures whose goal is to solve certain problems
- Behavioral
  - Help create and organize algorithms (behavior or logic) and the communication between objects

Creational Patterns

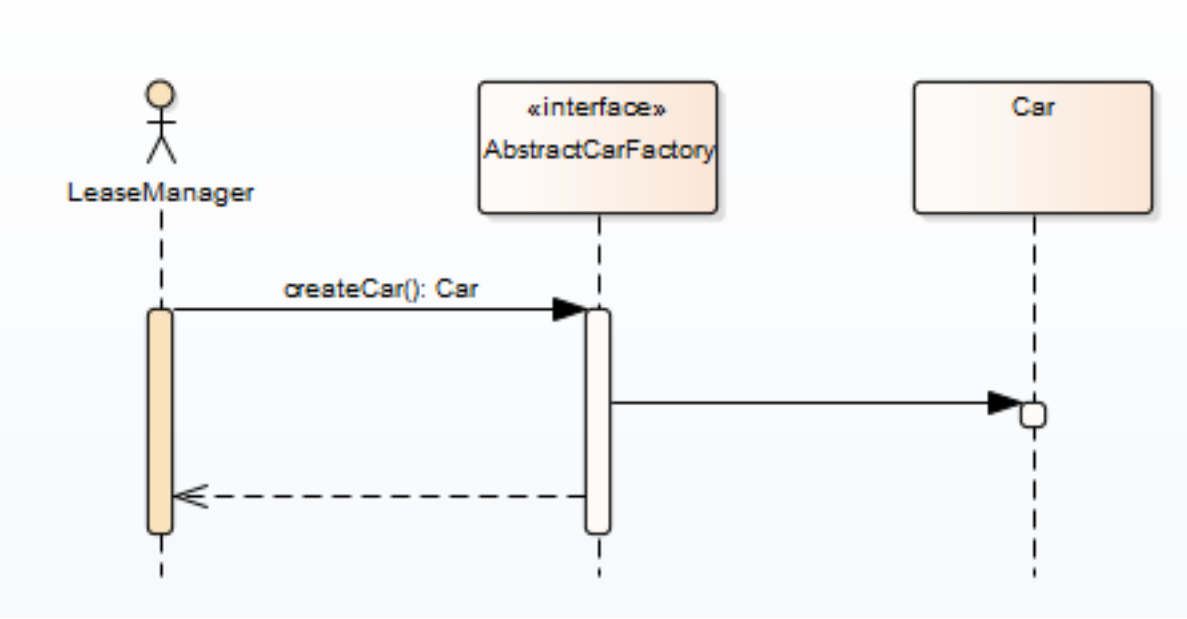
- Abstract Factory
- Builder
- Factory Method
- Singleton

Abstract Factory

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes
- Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. This applies to both specific objects as well as the factories themselves
- May result in extra initial code
- Two main concepts:
  - Factory: creates *product* objects
  - Product: object created by factory

Abstract Factory — Configuration

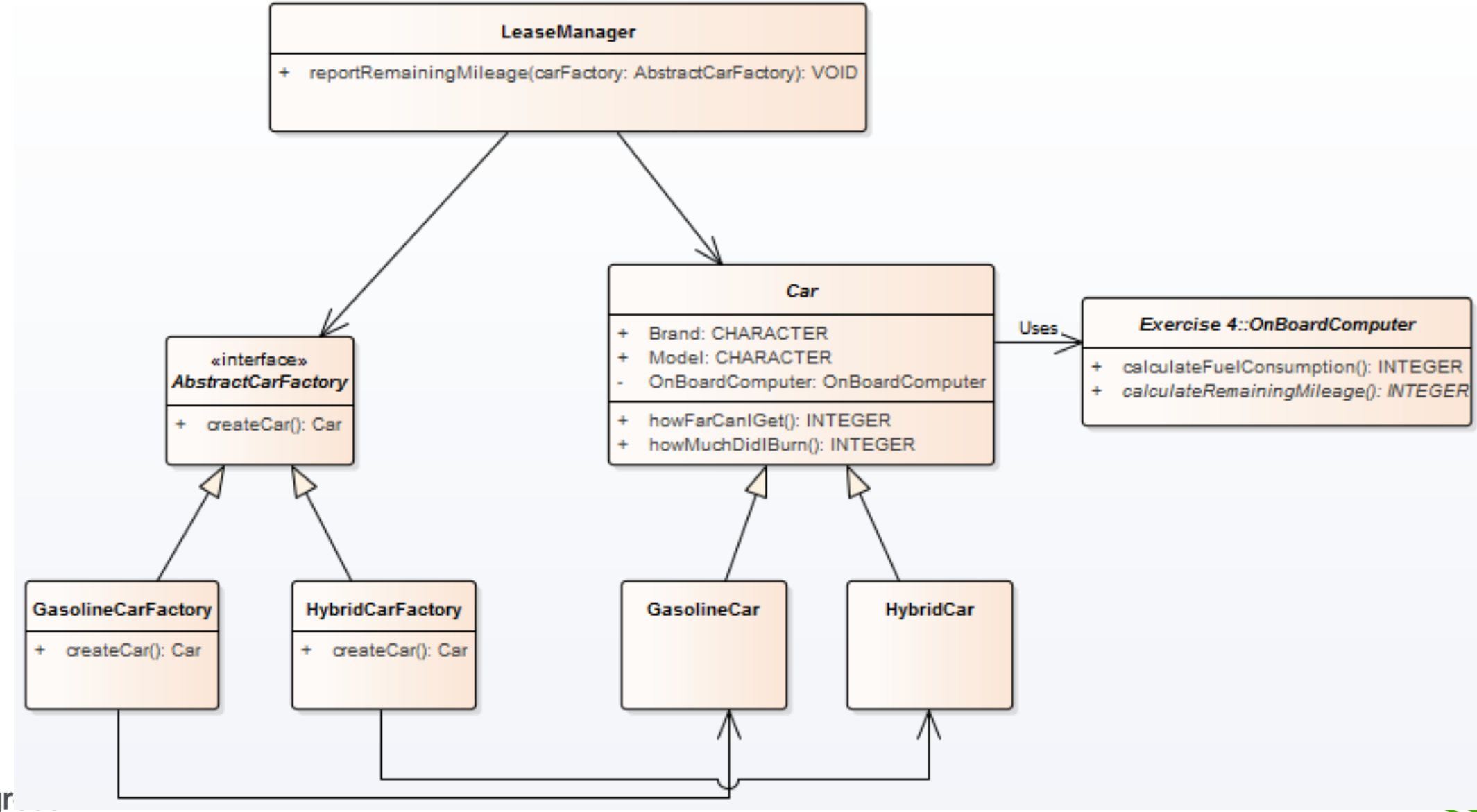
- A client uses an abstract factory
- Concrete factory determined at runtime e.g. configuration or user input
- The factory instantiates the objects as required by client



Abstract Factory — Exercise

- See Exercises folder, exercise 5
- See Bonus Exercise DesignPatternsOpenedge.AbstractFactory

Abstract Factory — Exercise Class Diagram



Abstract Factory — Exercise Bonus

- New requirement: lease manager now needs to be able to lease electric cars
- Extend application to fulfill new requirement without affecting lease manager
- Update class diagram (design)
- Write new classes (develop)

Builder

- Builder pattern builds a complex object using simple objects with a step by step approach
- When and why
  - Complex objects
  - Multiple ways to instantiate a complex object
- Solves the telescoping constructor anti-pattern
- Create an empty object then set its attributes
- Two main concepts:
  - Factory: creates *product* objects
  - Product: object created by factory

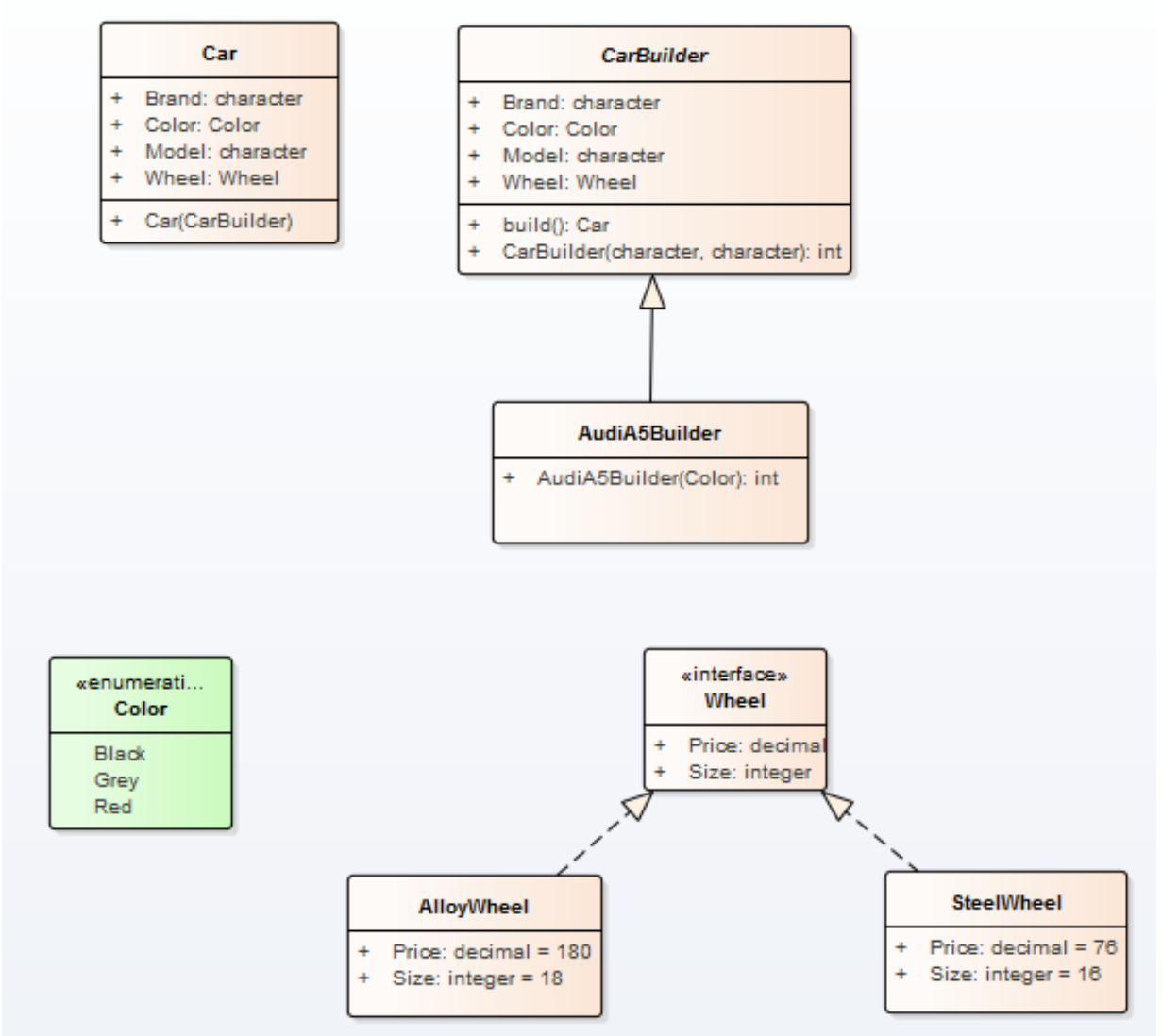
Builder — Configuration

- An abstract builder
- Concrete builders for each specific product (object) — they subclass the abstract builder
- Product's constructor (method) receives the concrete builder as its super type (abstract builder)
- Client instantiates desired builder and asks it for product

Builder — Exercise

- See Exercises folder, exercise 6
- See Bonus Exercise DesignPatternsOpenedge.Builder

Builder — Exercise Class Diagram



Factory Method

- Defines an interface for creating an object but let subclasses decide which class to instantiate
- Lets a class defer instantiation it uses to subclasses
- A.k.a. Virtual Constructor
- Allows the subclasses to choose the type of objects to create
- Promotes loose coupling by eliminating the need to bind application specific classes into the main client code
- Client code interacts solely with the resultant interface or abstract class so that it will work with any classes that implement the interface or extend the abstract class
- One of the most commonly used patterns
- Delegate the construction to a specific factory or a generic factory

Factory Method — Exercise

- See Exercises folder, exercises 7 A and 7B
- See Bonus Exercise DesignPatternsOpenedge.FactoryMethod

Singleton

- Restricts the instantiation of a class to one object
- An implementation of the singleton pattern must, therefore:
  - ensure that only one instance of the singleton class ever exists
  - provide global access to that instance
- Typically, this is done by:
  - declaring all constructors of the class to be private
  - providing a static method that returns a reference to the instance

Singleton — Exercise

- See Exercises folder, exercise 8
- See Bonus Exercise DesignPatternsOpenedge.Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

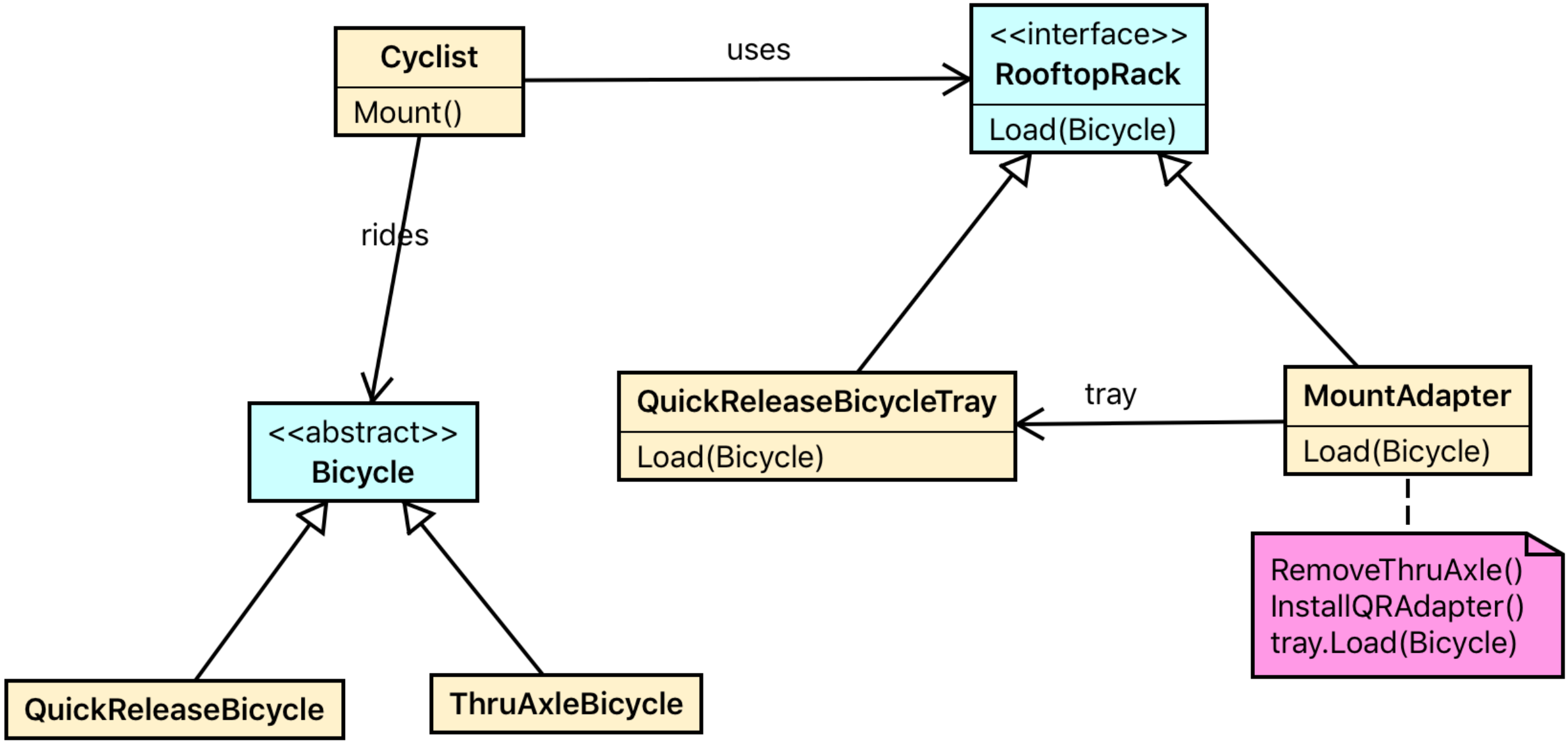
Adapter

- A.k.a. Wrapper
- Often an existing class cannot be reused because its interface does not conform to the interface a client requires and changing it is not possible or desirable
- A separate Adapter class converts the (incompatible) interface of a class (Adaptee) into another interface (Target) the client can use
- Use Adapter to work with, i.e. reuse, a class that does not have the required interface

Adapter — Exercise

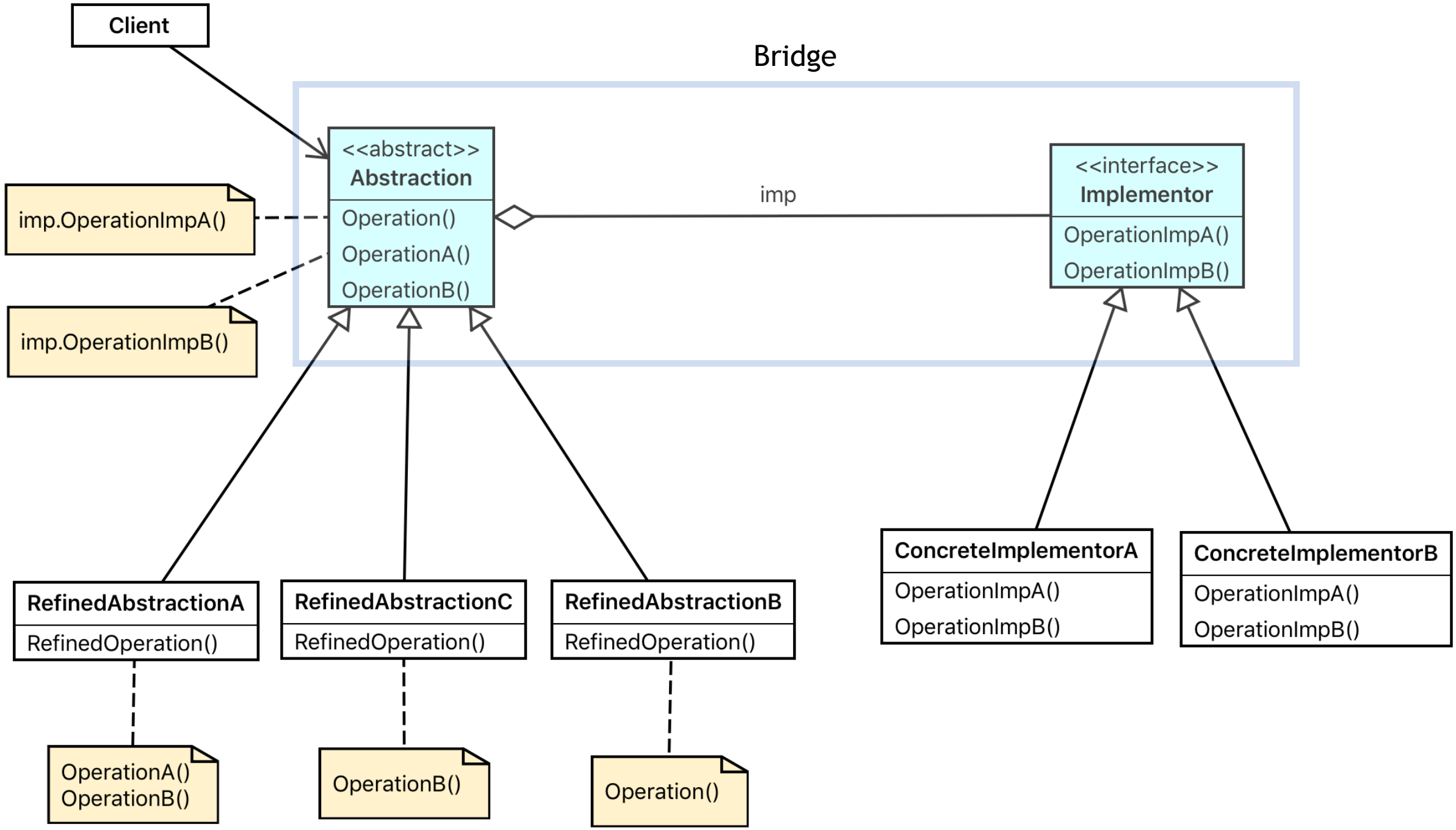
- See Exercises folder, exercise 9 A
- See Bonus Exercise DesignPatternsOpenedge.Adapter.Cycling (picture and class diagram on two next pages)
- See Bonus Exercise DesignPatternsOpenedge.Adapter.Editor

Adapter — Bonus Example


Cyclist
Mount()
uses
<<interface>>
RooftopRack
Load(Bicycle)
rides
<<abstract>>
Bicycle
QuickReleaseBicycleTray
Load(Bicycle)
tray
MountAdapter
Load(Bicycle)
RemoveThruAxle()
InstallQRAdapter()
tray.Load(Bicycle)
QuickReleaseBicycle
ThruAxleBicycle

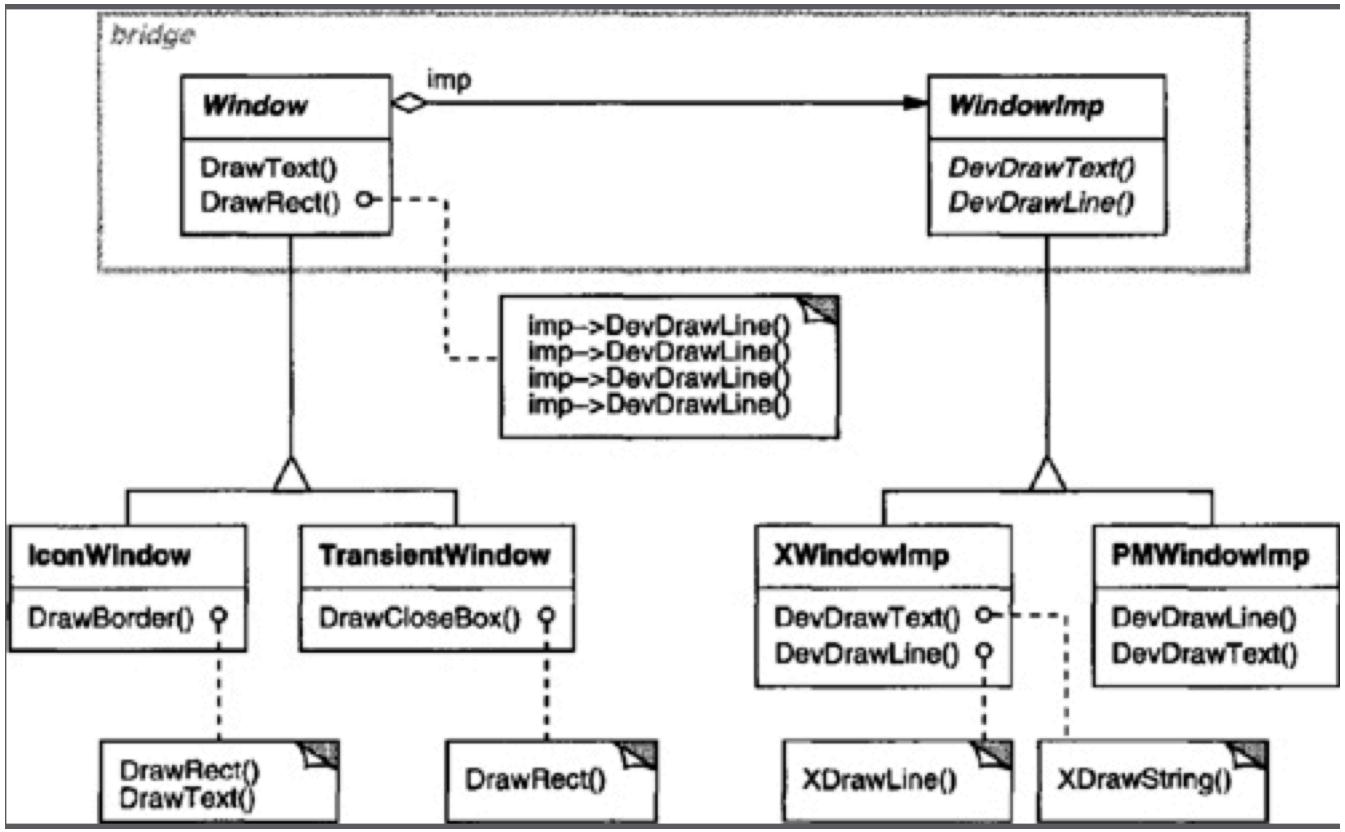
Bridge

- Decouple an abstraction from its implementation so that the two can vary independently
- Like Adapter but by design
- Change implementations without affecting client, without recompile
- Allow selecting and/or switching an implementation at run-time
- Solve “nested generalizations” — proliferation of classes or excess subclassing


Client
Bridge
<<abstract>>
Abstraction
Operation()
OperationA()
OperationB()
imp
<<interface>>
Implementor
OperationImpA()
OperationImpB()
imp.OperationImpA()
imp.OperationImpB()
RefinedAbstractionA
RefinedOperation()
RefinedAbstractionC
RefinedOperation()
RefinedAbstractionB
RefinedOperation()
ConcreteImplementorA
OperationImpA()
OperationImpB()
ConcreteImplementorB
OperationImpA()
OperationImpB()
OperationA()
OperationB()
OperationB()
Operation()

Bridge — Exercise

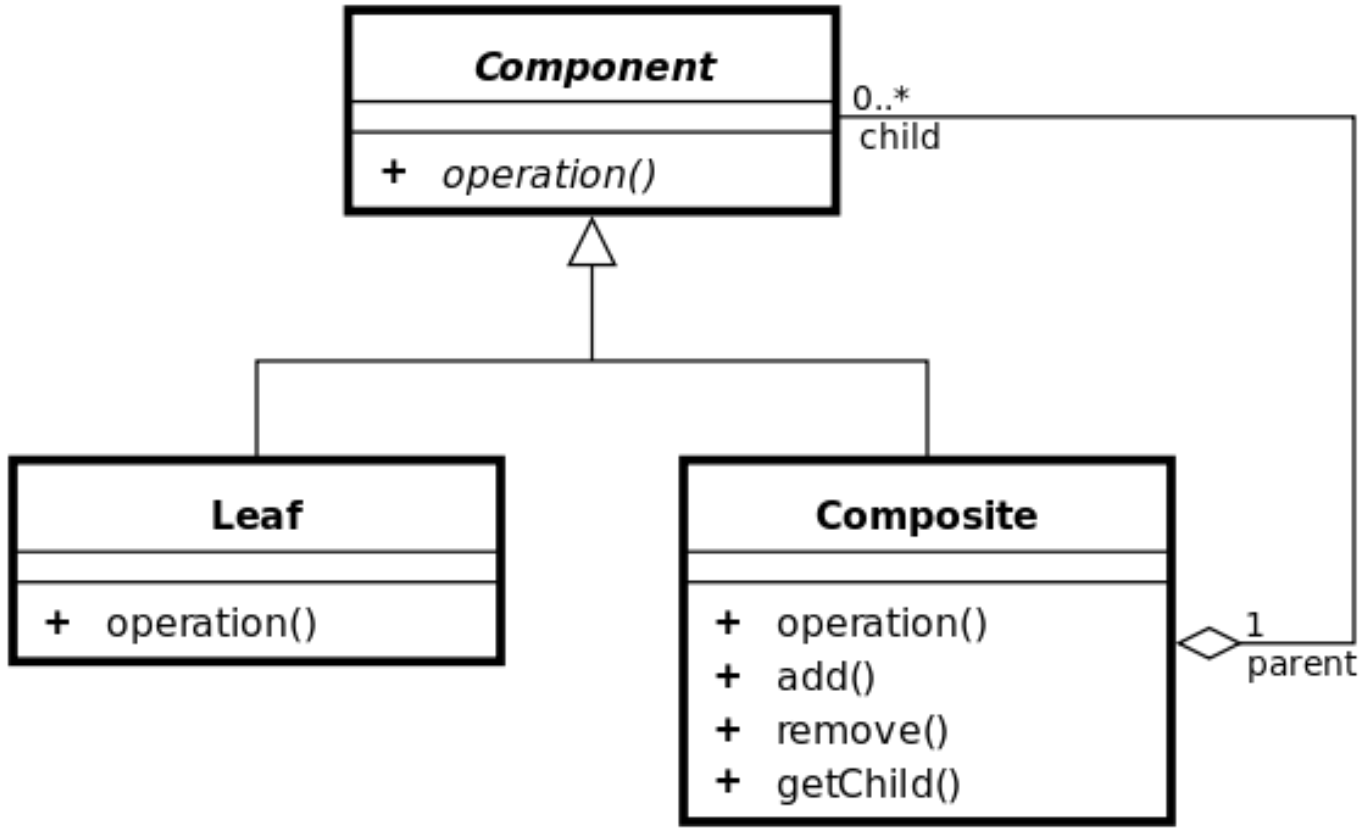
- See Bonus Exercise DesignPatternsOpenedge.Bridge (class diagram below)



Composite

- The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object
- The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies
- Implementing the composite pattern lets clients treat individual objects and compositions uniformly

Composite — Configuration



Composite Operations

- Where should the specific Composite operations (e.g. add) reside in?
  - The Component interface?
  - A Component abstract class?
  - The Composite class?

Composite — Exercise

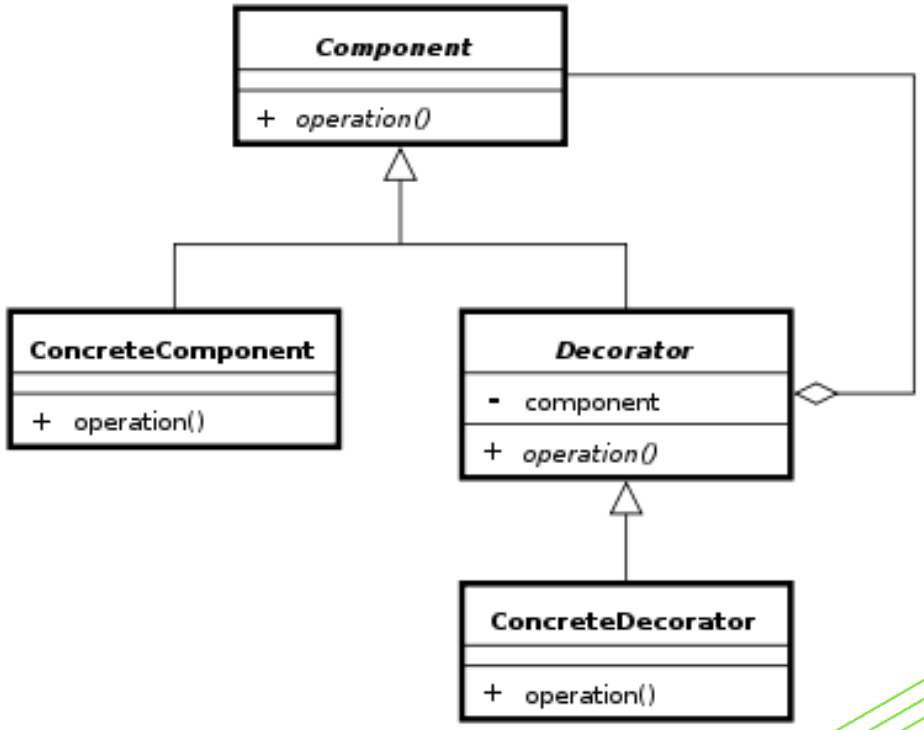
- See Exercises folder, exercise 10

Decorator

- Decorator is an alternative to subclassing
- Subclassing is bound to the class at compile time
- … and affects ALL instances of the original class
- A decorator can do this at run time
- … for selective objects
- “Decorates" an object with additional behavior determined at run time
- Uses object composition instead of subclassing

Decorator — Configuration

- Subclass the original Component class into a Decorator class (see UML diagram);
- In the Decorator class, add a Component pointer as a field;
- In the Decorator class, pass a Component to the Decorator constructor to initialize the Component pointer;
- In the Decorator class, forward all Component methods to the Component pointer; and
- In the ConcreteDecorator class, override any Component method(s) whose behavior needs to be modified.
- In OO ABL this “decoration” looks like:
  NEW MenuBranch(NEW MenuElement("Root"))

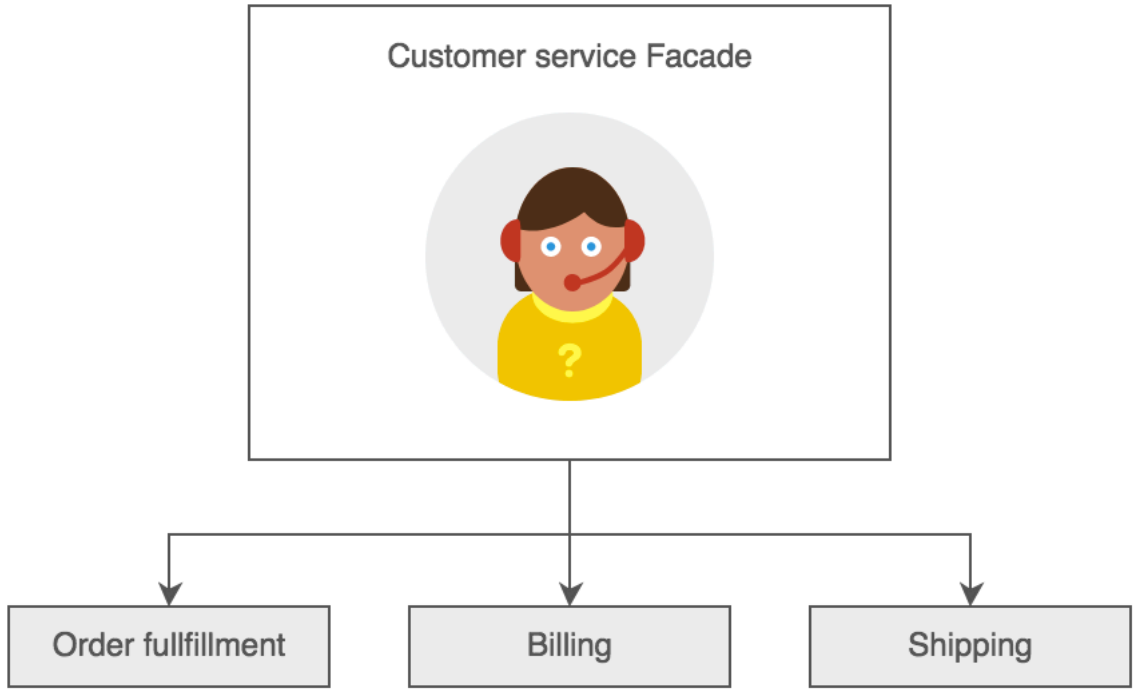


Decorator — Exercise

- See Exercises folder, exercise 11

Façade

- Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use
- Designed from the user perspective or use case
- Could be a singleton
- Typically wraps multiple objects



Façade — Exercise

- See Exercises folder, exercise 12

Flyweight

- Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of object state among multiple objects, instead of keeping it in each object
- An object has
  - Intrinsic state: unchangeable data
  - Extrinsic state: variable, context-specific, object data
- Example:
  - Create an object for every record in the database: results in thousands of objects
  - Instead, create a Business Entity that holds a dataset with all records

Proxy

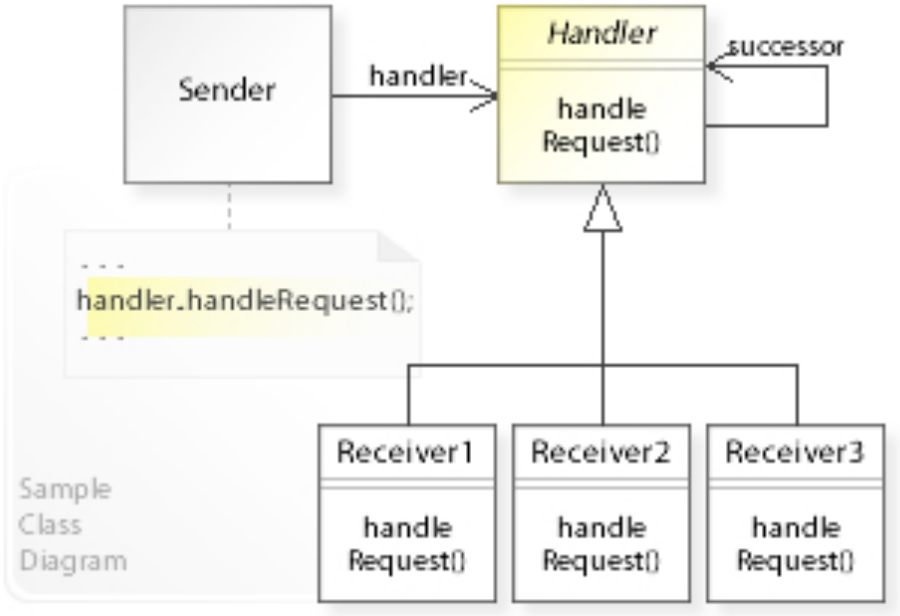
- A Proxy is a class functioning as an interface to something else
- A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scene
- E.g. a client-side object that calls its representative object on the server
- You will use this in Exercise 13 to call logic on an AppServer

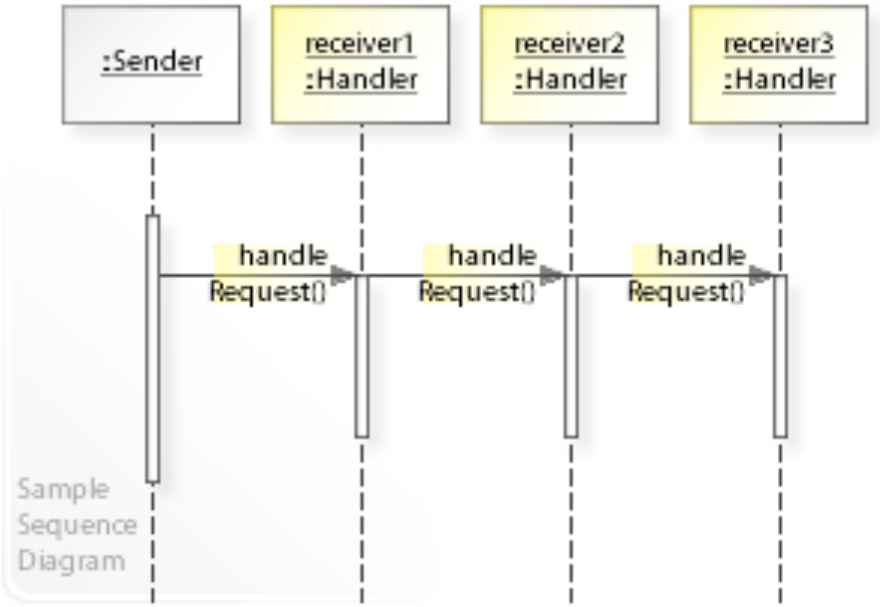
Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Chain of Responsibility

- A series of processing objects
- Based on Command objects
- Could become a Tree of Responsibility, e.g. An XML interpreter





Command

- Challenge: need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request
- The client that creates a command is not the same client that executes it
- Command can use Memento to maintain the state required for an undo-operation
- Two important aspects of the Command pattern: interface separation (the invoker is isolated from the receiver), time separation (stores a ready-to-go processing request to be started later — and even somewhere else when using Proxy)

Proxy, Chain, Command — Exercise

- See Exercises folder, exercise 13

Interpretor

- Provides a way to evaluate language grammar or expression
- E.g. SQL parsing, Rules engine, Domain Specific Language

Iterator

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- The traversal algorithm can be implemented in the Iterator OR in the Aggregator
- In the Aggregator: the Iterator acts as cursor
- In the Iterator: advantage is that different algorithms can be developed in subclasses
- To not violate the encapsulation principle, the Iterator should be an inner class of the Aggregator — not possible in ABL
- Therefore we need to implement the algorithms in the Aggregator. The Iterator only stores the state of the iterator

Iterator — Exercise

- See Exercises folder, exercise 14

Memento

- Purpose: need to restore an object back to its previous state (e.g. "undo"or "rollback" operations)
- Use a Memento object to manipulate state of an object
- The Originator (e.g. Car) knows how to save and restore its state to and from the Memento object
- The client (Caretaker) requests a Memento from the Originator

Memento — Exercise

- See Exercises folder, exercise 15
- Notice the serializer can only copy public properties
- As a bonus exercise use clone instead of serializer
  - Override the clone() method to copy all properties both public and private

Observer

- The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods
- OpenEdge has Publish and Subscribe built-in for Procedural and OO coding

```
[ publisher : ]event-name : { Subscribe | Unsubscribe }
  ( [subscriber : ]handler-method|
    [subscriber-handle , ]handler-procedure ) [ NO-ERROR ] .
```

```
DEFINE [event-modifiers] EVENT event-name
    { [ SIGNATURE ] VOID ( [parameter[ , parameter]...] )
    | [ DELEGATE ] [ CLASS ]dotNet-delegate-type} .
```

```
[ THIS-OBJECT : |class-type-name : ]
 event-name : Publish ( [parameter[ , parameter]...] ) [ NO-ERROR ].
```

Observer — Exercise

- See Exercises folder, exercise 16

Mediator

- Encapsulates communication between objects
- Competes with Observer
- But …. less reusable
- Could be used as a way to ensure communication with Observers (like a service bus)

State

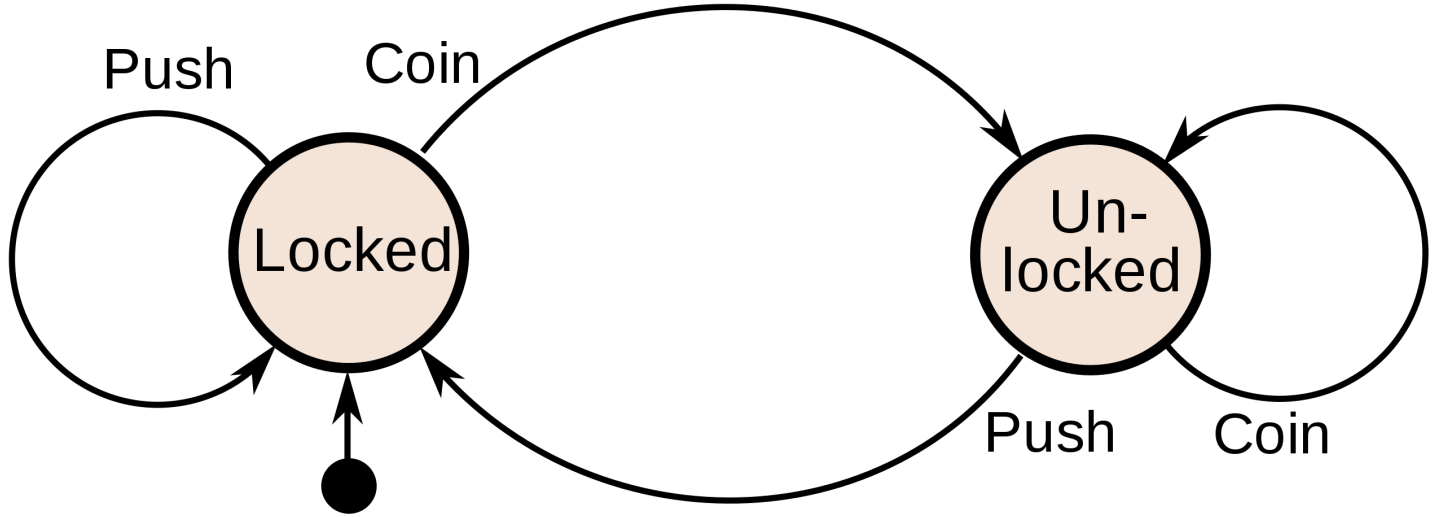
- The State pattern is a solution to the problem of how to make behavior depend on state
- We can use it to make a Finite State Machine: an abstract machine that can be in exactly one of a finite number of states at any given time.
- Consists of:
  - A ‘context’ class
  - A State abstract class
  - Concrete classes for the different states, these hold state-specific behavior
- The ‘context’ class holds a pointer to the current state, also used for changing state
- State changes are fired via Actions

State — Exercise

- See Exercises folder, exercise 17

State — Bonus Exercise

- Design and implement a set of classes to manage turnstiles
  - A turnstile has two states: Locked and Unlocked
  - It support two actions: Push and Coin
- Define the State class as abstract so action methods can be overridden (or not) — this will allow invalid actions to be performed without causing pointer errors



Strategy — Exercise

- See Exercises folder, exercise 4

Template Method

- Used to implement the invariant parts of an algorithm once and allow subclasses implement the behavior that can vary
- Useful for common classes that generalize behavior to avoid code duplication
- How: create an abstract class that implements methods with invariable logic and defines the abstract methods for the variable logic
- E.g. “hook” operations
- E.g. refactoring to generalize — *Opdyke and Johnson*

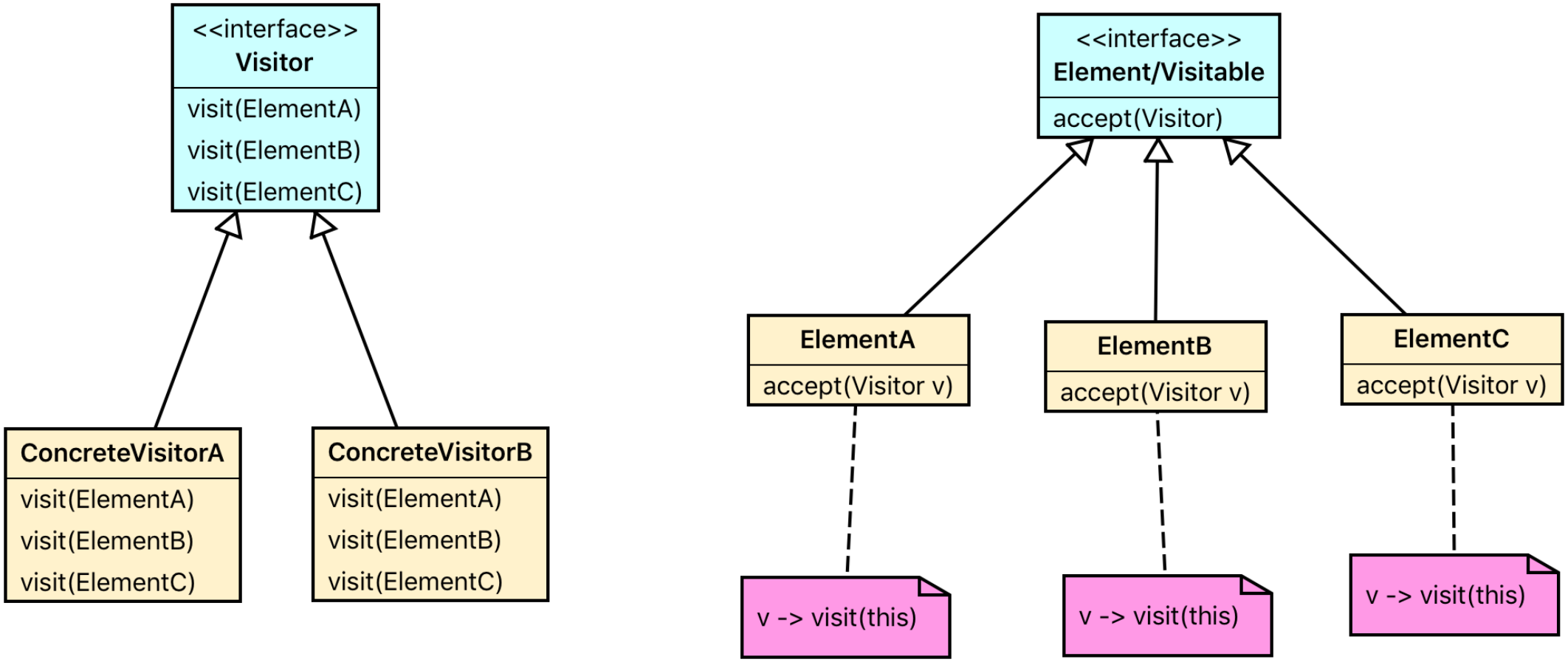
Template Method — Exercise

- See Exercises folder, exercise 4

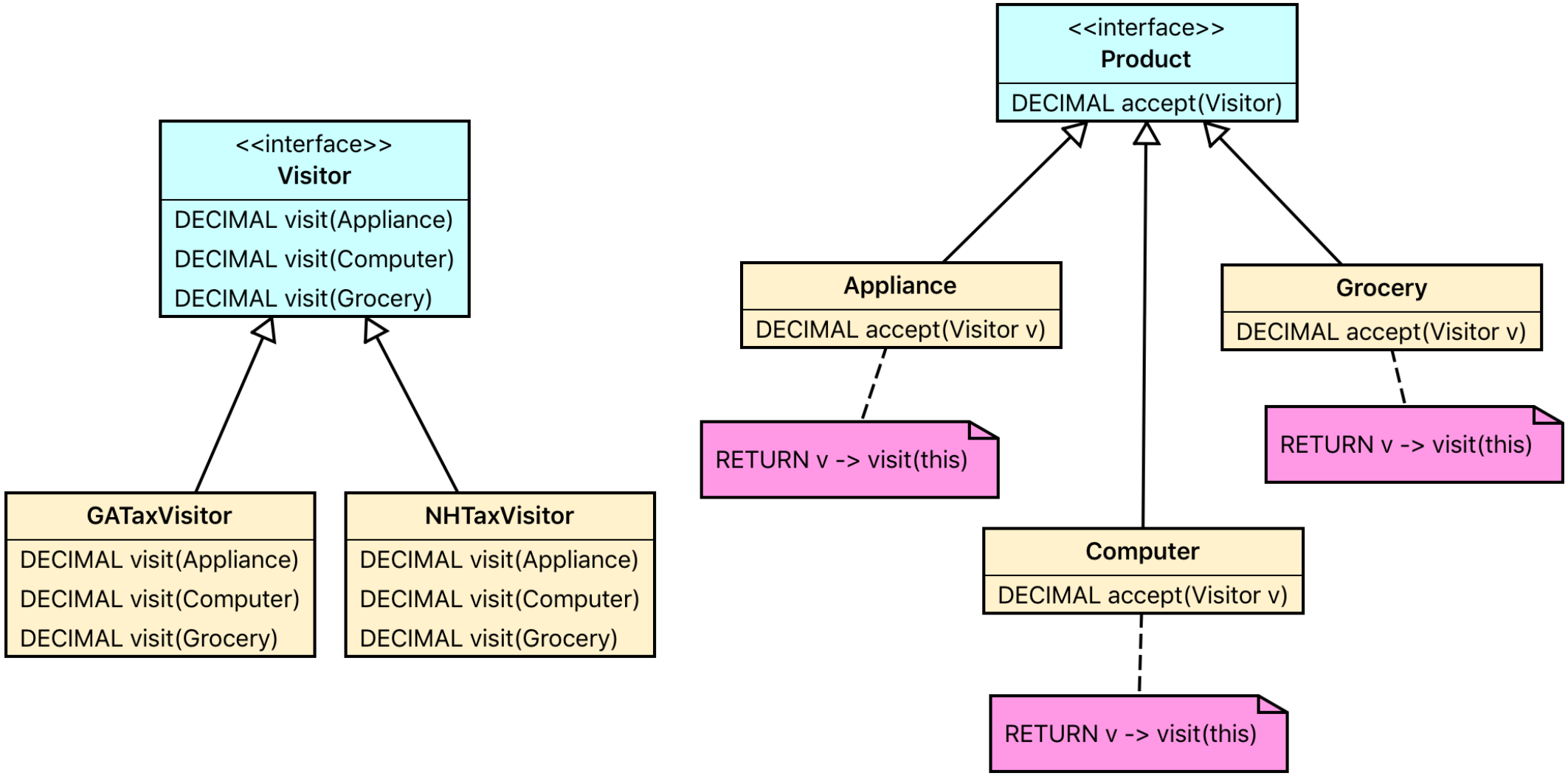
Visitor

- Visitor pattern allows a class to defer operations to a separate class which “visits” it back for properties and methods
- The operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits)
- Configured by Visitor interface with concrete visitors and a Visitable interface used by classes that need to be visited
- Useful for complex object structures regardless of type or hierarchy e.g. can be used when traversing tree view object structures or collections
- The actual logic will be implemented within the concrete visitor classes — on behalf of the visited classes
- Clients will specify which concrete visitor class (object) to use to visit certain elements
- Visitor relies on double dispatch mechanism to achieve its “visiting” goal: accept/visit
- Element objects “accept” a concrete visitor which in turn calls the “visit” method within the visitor passing itself along — double dispatch

Visitor — Class Diagram



Visitor — Exercise Class Diagram



Visitor — Exercise Client (Test)

```
DEFINE VARIABLE taxCalc AS TaxVisitor NO-UNDO.
DEFINE VARIABLE prod1   AS Product    NO-UNDO.
DEFINE VARIABLE prod2   AS Product    NO-UNDO.
DEFINE VARIABLE prod3   AS Product    NO-UNDO.
DEFINE VARIABLE total1  AS DECIMAL    NO-UNDO.
DEFINE VARIABLE total2  AS DECIMAL    NO-UNDO.
DEFINE VARIABLE total3  AS DECIMAL    NO-UNDO.

ASSIGN
    taxCalc = NEW GATaxVisitor()
    prod1   = NEW Grocery("Milk",3,5)
    prod2   = NEW Appliance("Refrigerator",1,1500)
    prod3   = NEW Computer("iMac",1,1200)
    total1  = prod1:accept(taxCalc)
    total2  = prod2:accept(taxCalc)
    total3  = prod3:accept(taxCalc).

MESSAGE prod1:productName STRING(total1,"$>>>,>>9.99") VIEW-AS ALERT-BOX.
MESSAGE prod2:productName STRING(total2,"$>>>,>>9.99") VIEW-AS ALERT-BOX.
MESSAGE prod3:productName STRING(total3,"$>>>,>>9.99") VIEW-AS ALERT-BOX.
```

Progress OpenEdge resources

- **Web site**

  https://www.progress.com/openedge

- **YouTube channel**

  https://www.youtube.com/channel/UCNxy1VY73tYJ7o_R25HjohQ/playlists?shelf_id=10&view=50&sort=dd

- **Documentation**

  http://documentation.progress.com/output/ua/OpenEdge_latest/

- **Technical support and knowledge base**

  https://www.progress.com/support/reference-guide

- **Developer community**

  https://community.progress.com

- **Progress eLearning community**

  https://wbt.progress.com