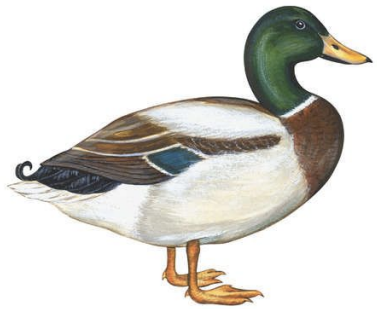
ADDITIONAL MATERIAL FOR INFSEN01-2

DR ANNA V. KONONOVA

LECTURE 2. VISITOR PATTERN. MAY 9, 2017

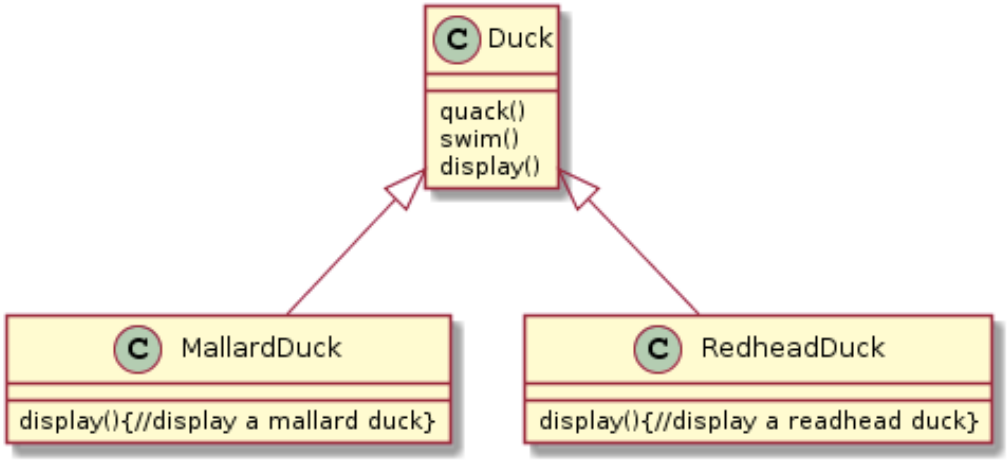
- Reminder: **Read all official slides as well**
- Photo opportunity!
- Last week, we have clarified terminology, remembered basics and principles of OOP, seen where design patterns come from, and where exactly they can help, reviewed sources and consequences of design choices, understood diversity of design patterns, looked at some how-tos and examples
- **Today** we will: quickly revise last lesson's duck example, study the first design pattern and try applying it to some new example problem



WAKE UP BREAK. NATURAL EVOLUTION OF REQUIREMENTS

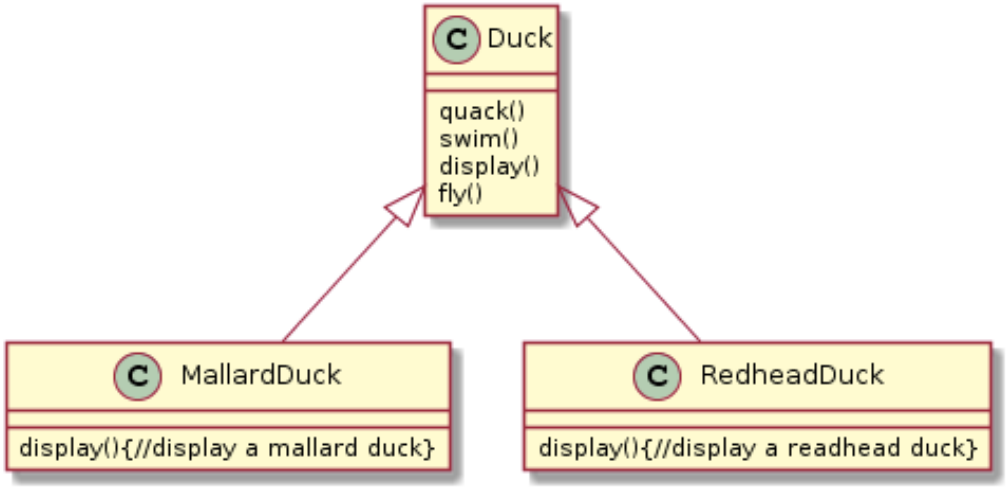
- Duck pond simulator game. A variety of ducks species swims around and makes quacking noises. ➤ superclass Duck with methods quack(), swim() and display() and two subclasses: MallardDuck and RedheadDuck both of which redefine display() method to show an appropriate duck.
- Now we need ducks to fly. How would you redesign the classes?
- A new type of the duck, a rubber duck, is introduced. Which seems to be able to fly now. Can you think of any change for your design?
- However, rumour has it that soon the simulator will also have wooden decoy ducks
- But what if ducks suddenly change the way they fly? All subclasses will need a change!
- Design pattern? Not yet! Just an OO principle (which?)
- "Encapsulate the change" principle. How to use it here?
- Duck superclass looks stable. The change comes from flying and quacking. And now?
- Two separate sets of classes: flying behaviour and quacking behaviours.
- We can now even assign behaviours run-time
- Looks like another OO principle is coming to play here as well. Which one?
- "Program to interface, not an implementation".
- An interface for each behaviour and each implementation of behaviour implements one of those interfaces ➤ behaviour classes!
- Now the superclass uses a behaviour represented by an interface. An actual implementation is not locked into sub-classes
- Draw and explain the new design

DUCK POND GAME: NEW DESIGN EVOLUTION



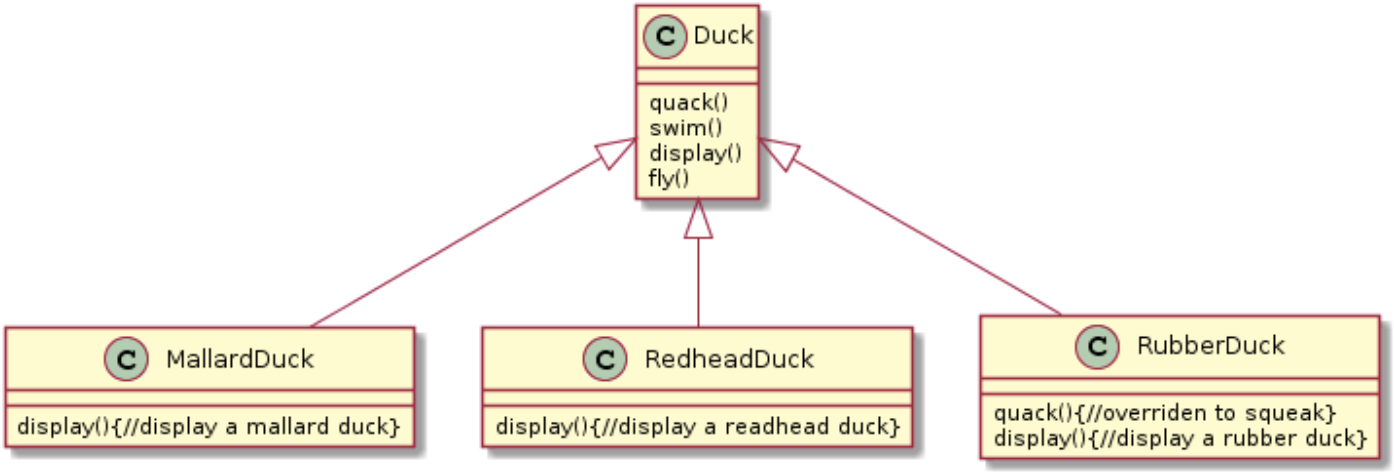
Duck pond simulator game. A variety of ducks species swims around and makes quacking noises. ➢ superclass Duck with methods quack(), swim() and display() and two subclasses: MallardDuck and RedheadDuck both of which redefine display() method to show an appropriate duck.

DUCK POND GAME: NEW DESIGN EVOLUTION



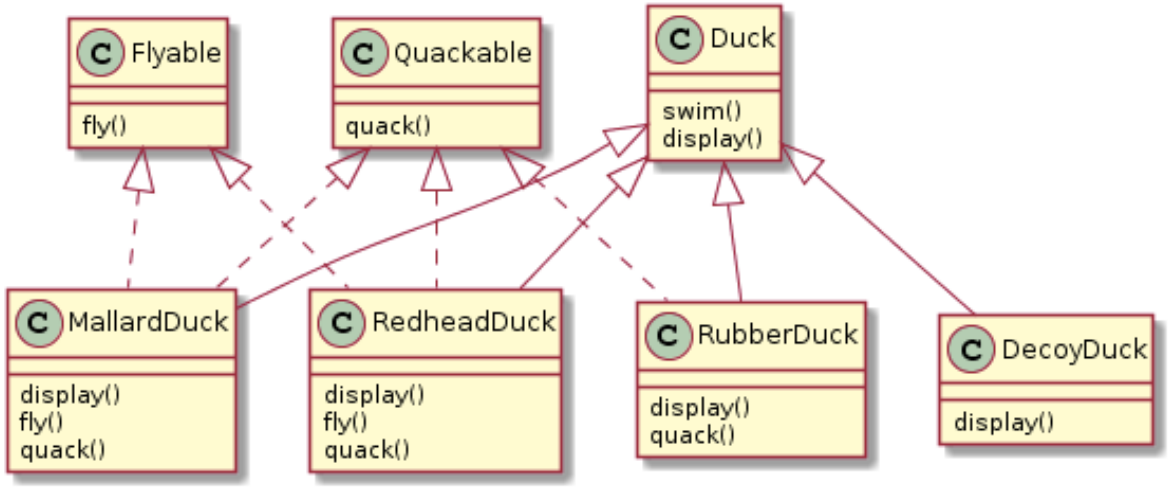
Now we need ducks to fly. How would you redesign the classes?

DUCK POND GAME: NEW DESIGN EVOLUTION



A new type of the duck, a rubber duck, is introduced. Which seems to be able to fly now. Can you think of any change for your design?

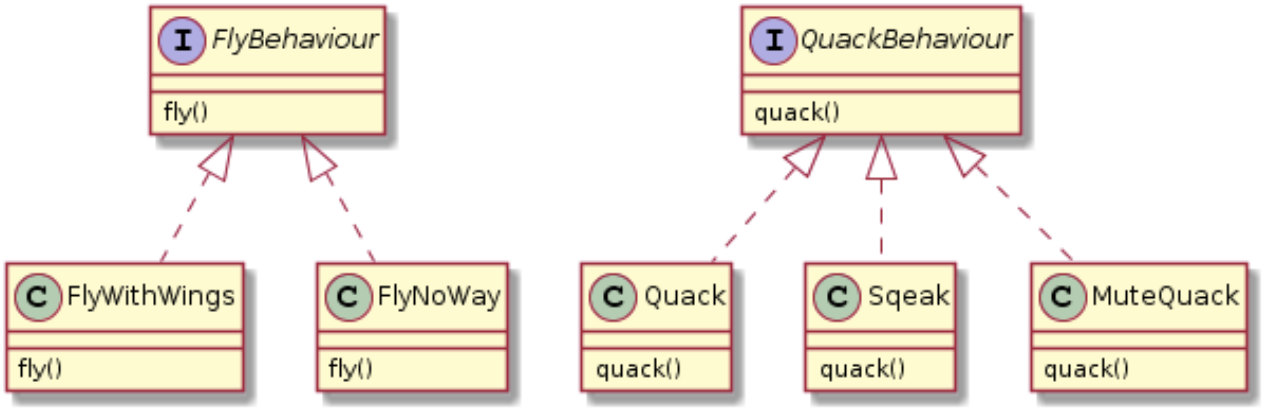
DUCK POND GAME: NEW DESIGN EVOLUTION



Introduce Flyable() interface with fly() method which flying ducks can implement. And Quackable() interface too, with quack() method

But what if ducks suddenly change the way they fly? All subclasses will need a change!

DUCK POND GAME: NEW DESIGN EVOLUTION



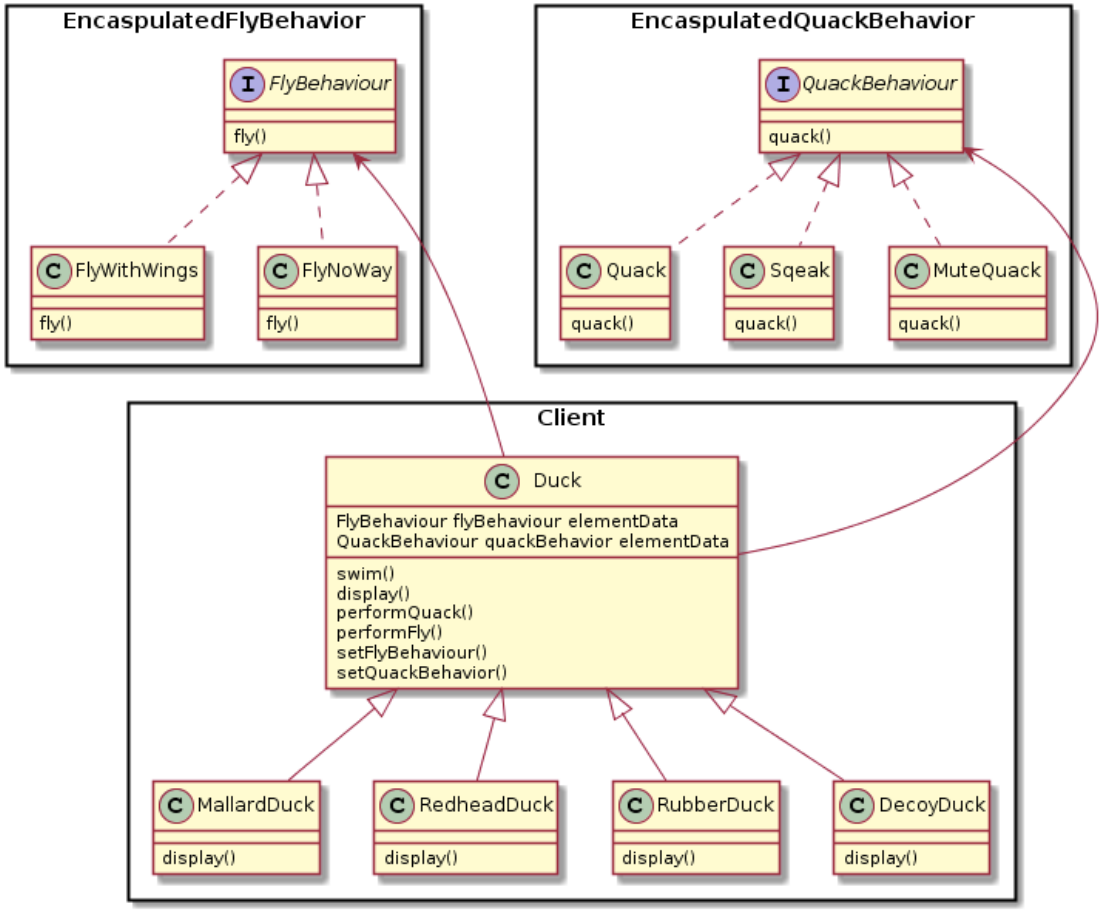
Duck superclass looks stable. The change comes from flying and quacking.

Separate them in two separate classes! Actually two separate sets of classes: flying behaviour and quacking behaviours. This kind of modification will later also accommodate squeaking and being mute.

DUCK POND GAME: NEW DESIGN EVOLUTION

- We can now even assign behaviours run-time, can we?
- Looks like another OO principle is coming to play here as well:"Program to interface, not an implementation".
- An interface for each behaviour and each implementation of behaviour implements one of those interfaces ➢ behaviour classes!
- Now the superclass uses a behaviour represented by an interface. An actual implementation is not locked into sub-classes

DUCK POND GAME: NEW DESIGN EVOLUTION



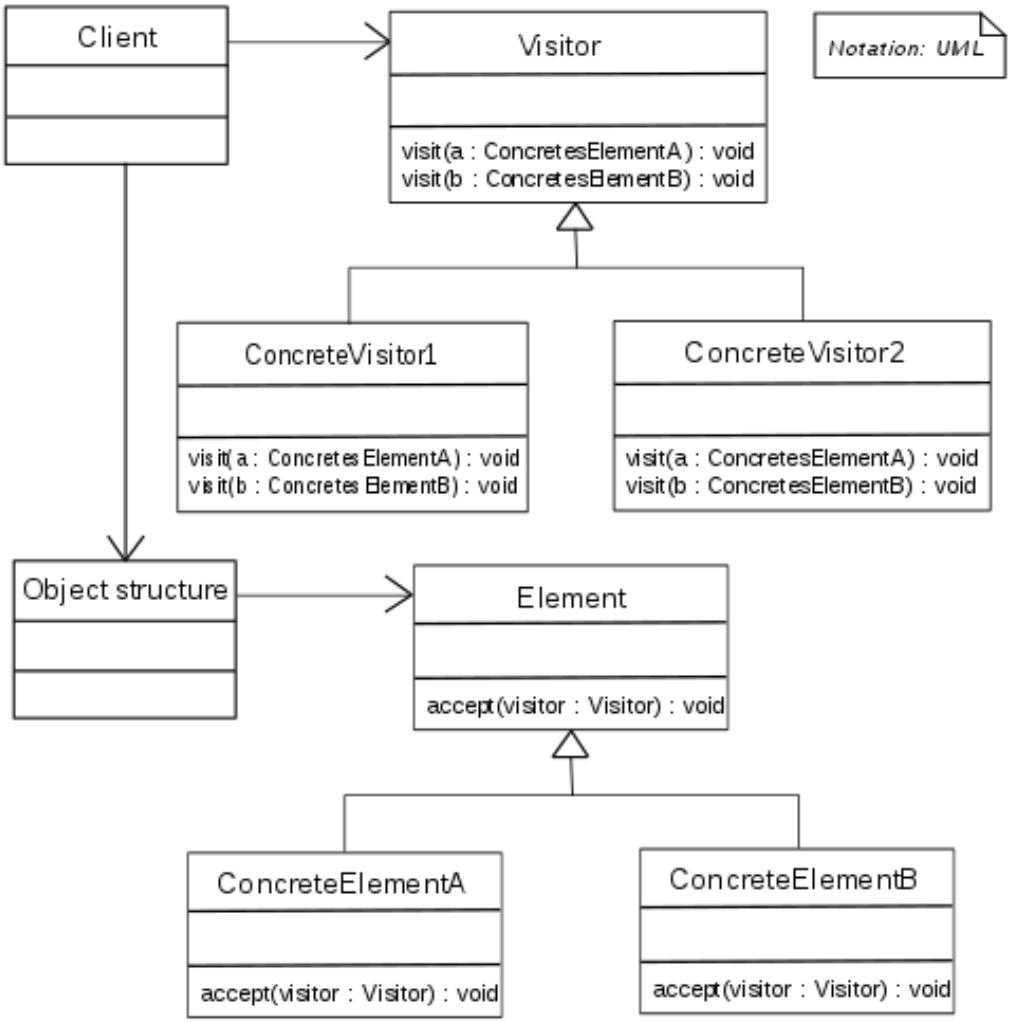
VISITOR: INTENT

- Setting the ducks aside, let us start with the topics of our design patterns course
- Behavioural pattern **Visitor**
- **Intent**: represent an operator to be performed on the elements of an object structure. Visitor allows a definition of a new operation without changing the classes of the elements on which it operates
- **In other words**: numerous distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. Avoiding "polluting" the node classes with these operations is desired. Therefore, operational logic is separated in a different class
- **Example**: a shopping cart in an online supermarket where different type of items can be added; clicking on checkout button calculates the total amount to be paid
- Imagine we have the following shopping list: 1. GoF book; 2. notepad; 3. pen; 4. banana; 5. bottled water (everything you might need to get for studying for this course)
- Each of the items has a price and a weight

VISITOR: MOTIVATION

- Distributing operations across nodes is hard to understand, maintain and modify
- Possible heterogeneous nature of the nodes will require to query the type of each node and cast the pointer to the correct type before performing the required operation
- Adding new operation will require recompiling all classes
- ➢ packaging operations from all node classes in a separate object called a **visitor** and passing it to the nodes of aggregate structure as they are being traversed
- When a node **accepts** the visitor, it sends a request to the visitor that encodes the class of this node. This request also includes the node as an argument. The visitor then executes the operation for this node.

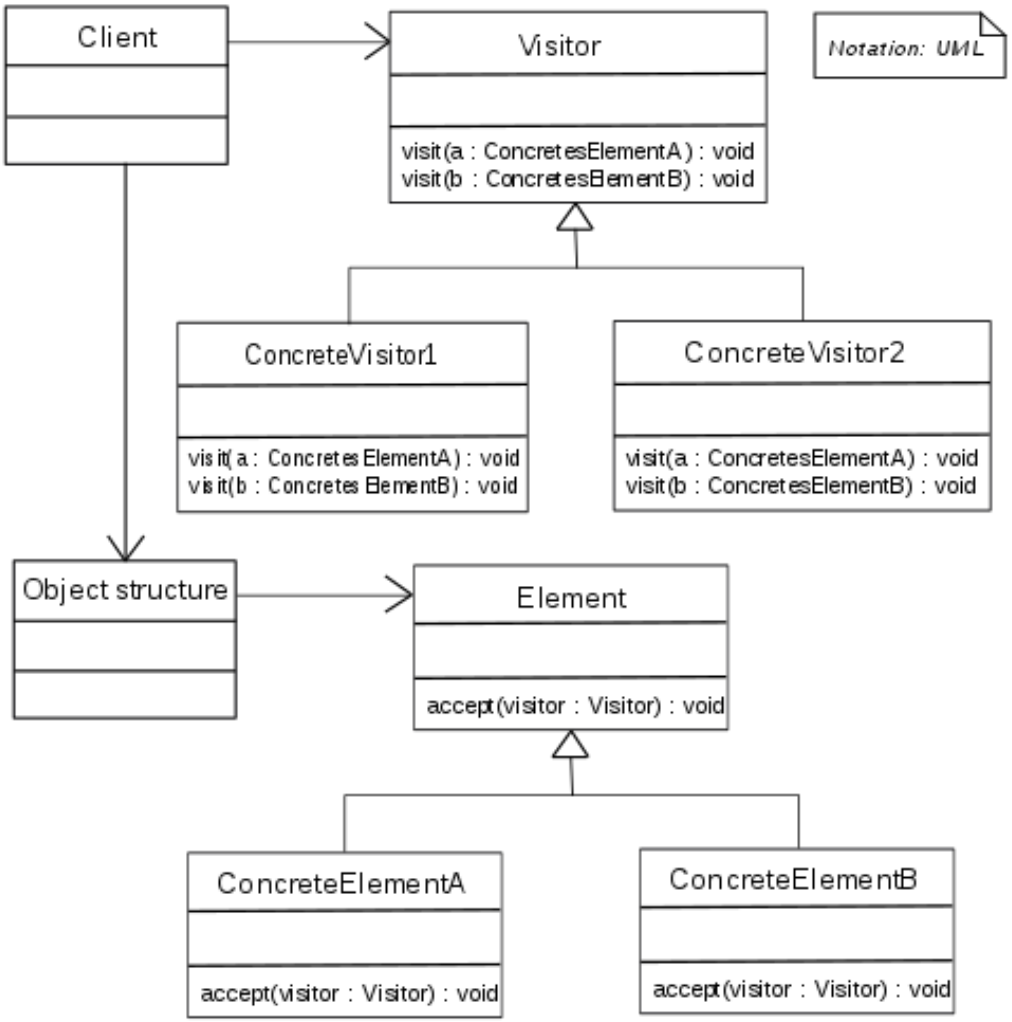
VISITOR: STRUCTURE



VISITOR: PARTICIPANTS

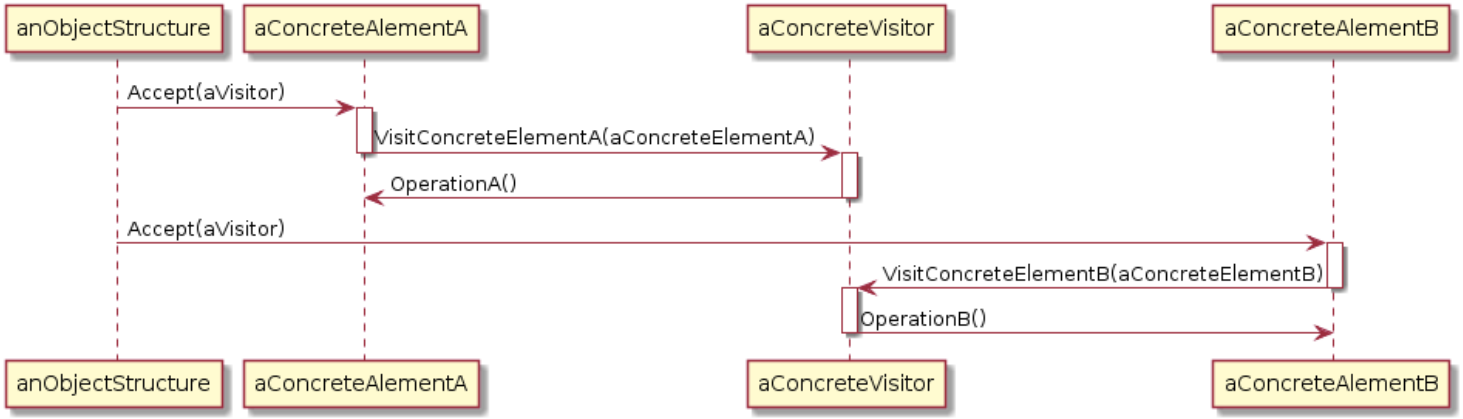
- Visitor: declares a Visit operation for each class of ConcreteElement in the object structure
- ConcreteVisitor: implements each operation declared by the Visitor
- Element: defines an Accept operation that takes a visitor as an argument
- ConcreteElement: implements an Accept operation that takes a visitor as an argument
- ObjectStructure: can enumerate its elements, may provide a high-level interface to allow the visitor to visit its elements

VISITOR: STRUCTURE



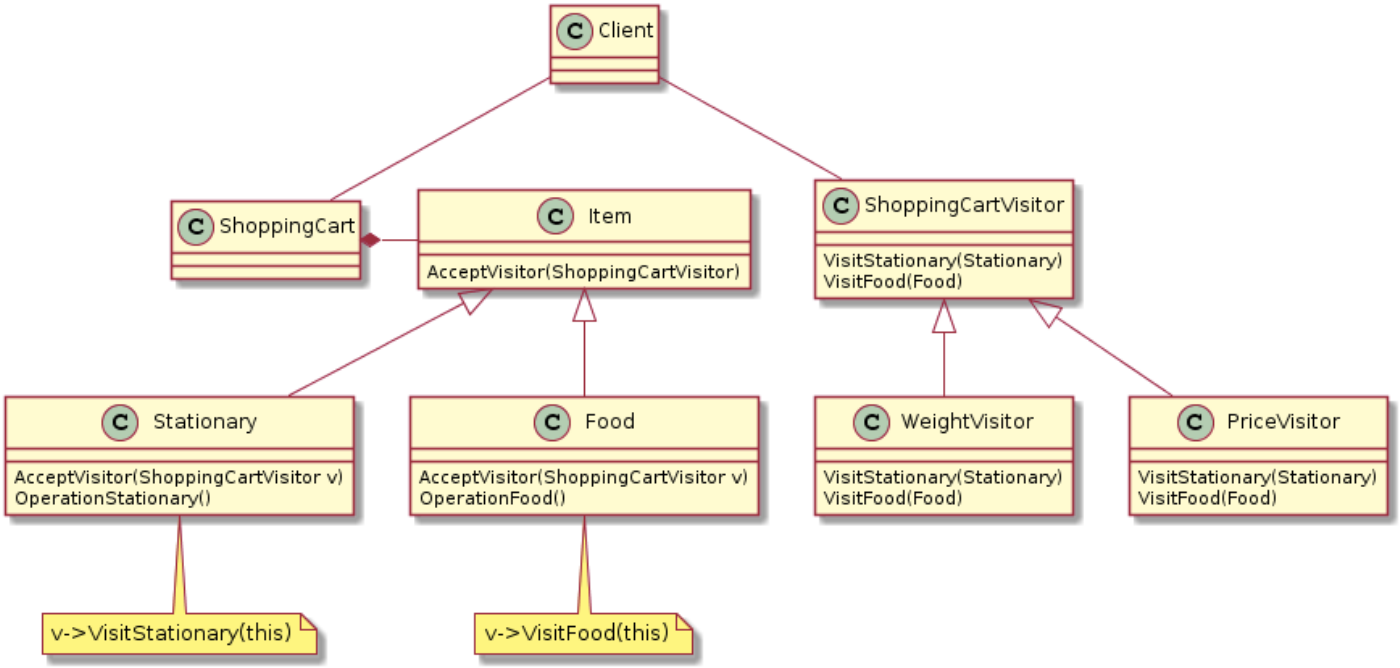
- Visitor
- ConcreteVisitor
- Element
- ConcreteElement
- ObjectStructure

VISITOR: COLLABORATIONS



- A client using Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each each element with the visitor
- When an element is visited it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operaation to let the visitor access its state if necessary

VISITOR: SHOPPING CART EXAMPLE



A shopping cart in an online supermarket where different type of items can be added; clicking on checkout button calculates the total amount to be paid

Shopping list: 1. GoF book; 2. notepad; 3. pen; 4. banana; 5. bottled water

Each of the items has a price and a weight

VISITOR: APPLICABILITY, CONSEQUENCES

- **Applicability**:

Object structure contains many classes of objects with differing interfaces and operations that need to be performed on the depend on concrete classes

Many distinct and unrelated operations to be performed on objects and class pollution must be avoided

Object classes rarely change and operations are added often

- **Consequences**:

Easy addition of new operations: simply by adding a new visitor

Gather related operations together, separate unrelated ones

Adding new ConcreteElement classes is hard (explain to me why)

Allows visiting across class hierarchies (unlike Iterator pattern)

Visitors can accumulating state

Possible breaking encapsulation due to the need to provide public operators to access element's internals

BEHAVIOURAL PATTERNS: COMMON FEATURES PRESENT IN VISITOR PATTERN

- Encapsulating variation

New objects that encapsulate the varying aspects and existing objects that use the new objects

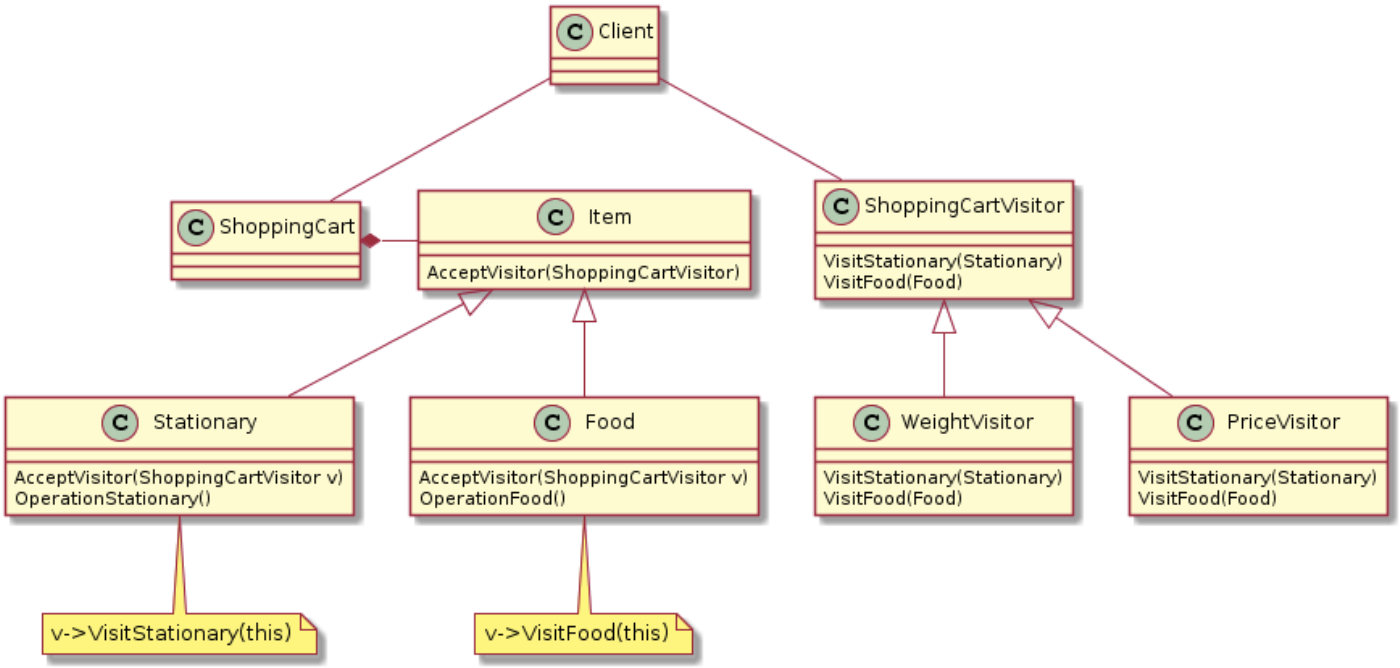
- Objects as arguments

A Visitor object is the argument to a polymorphic Accept operation on the objects it visits.

VISITOR: SHOPPING CART EXAMPLE

- Think whether you understand how Visitor Design pattern is applied to the Shopping cart example
- Write (snippets of) code that returns cost and weight of my earlier shopping list:

1. GoF book; 2. notepad; 3. pen; 4. banana; 5. bottled water



VISITOR: SHOPPING CART EXAMPLE

- Code

CONCLUSION

- **Today** we have revised last lesson duck example, studies the first design pattern, applied visitor pattern to shopping cart example

✓ Lesson 1: Introduction

✓ Lesson 2: Visitor pattern

⌛ Lesson 3: Iterator pattern

⌛ Lesson 4: Adapter pattern

⌛ Lesson 5: Abstract classes

⌛ Lesson 6: Abstract factory pattern

⌛ Lesson 7: Decorator pattern

⌛ Lesson 8: Revision

- Thanks for your attention!

BACK-UP SLIDE. VISITOR PATTERN: IMPLEMENTATION ASPECTS

- Double dispatch

Visitor pattern allows adding operations to classes without changing them. This is done via the technique called double dispatch.

Single dispatch: two criteria determine which operation will fulfill a request (the name of the request and the type of receiver)

Double dispatch: operation that gets executed depends on the kind of the request and the types of *two* receivers: choice of *Accept* operation depends on the type of the Visitor and type of the Element

Instead of binding operators statically into the Element interface, VDP allows consolidating operations in a Visitor and use *Accept* to do the binding runtime

- Who is responsible for traversing the object structure

A Visitor must visit all elements of the object structure. Traversal responsibility can reside in (a) the object structure, (b) the visitor, (c) separate iterator object (lecture 3)

- Back-up slides

BACK-UP SLIDES