# Visitor Pattern

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

## Design Aspect of Visitor

Operations that can be applied to objects without changing their classes

## Outline

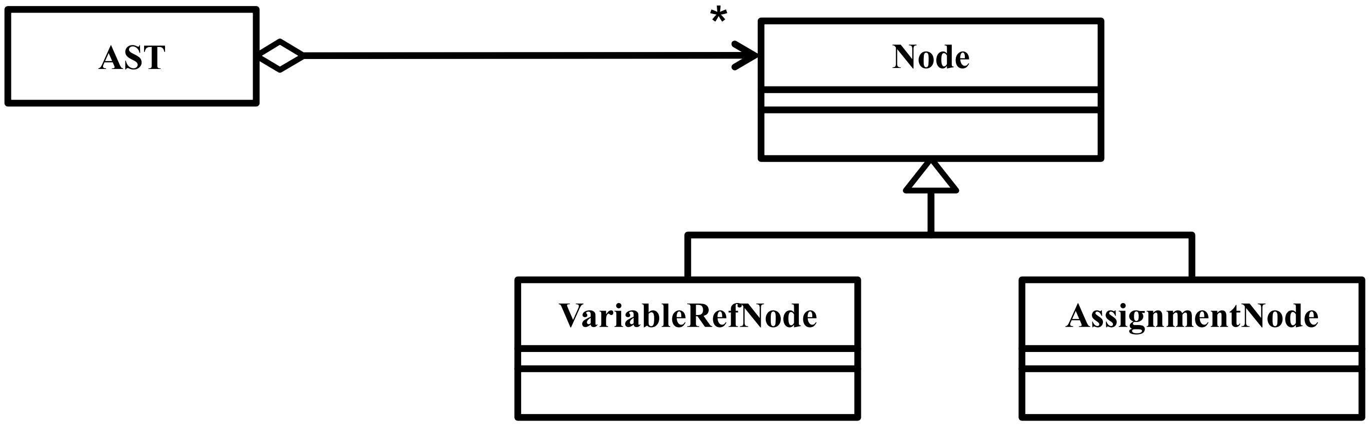
- Compiler and AST Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Recurrent Problems
- Intent
- Visitor Pattern Structure
- Double-Dispatch
- Nutrition Retrieval from A Restaurant Menu: Another Example
- Equipment Power Consumption: Another Example

# Compiler and AST (Visitor)

Shin-Jie Lee (李信杰)
Assistant Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

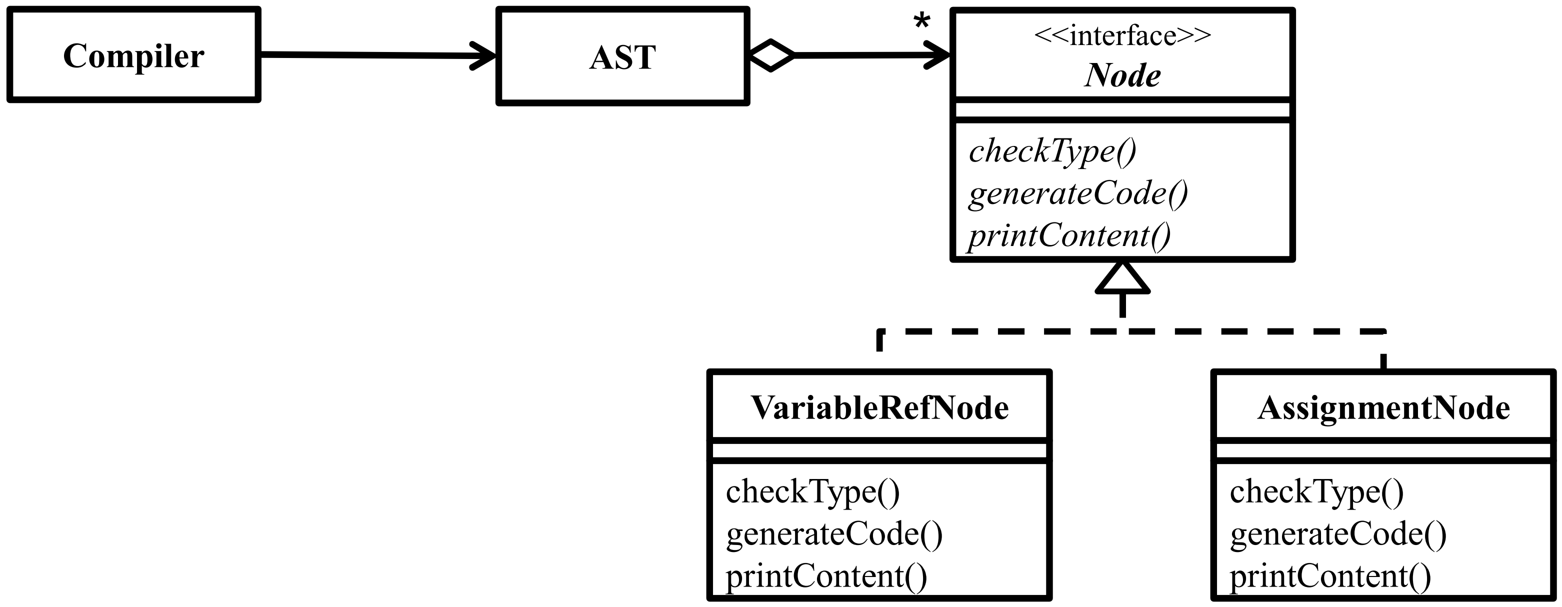
## Requirements Statements$_1$

- There are several nodes in an abstract syntax tree (AST), such as VariableRefNode and AssignmentNode, which represent respective parts in source code and keep the code information.

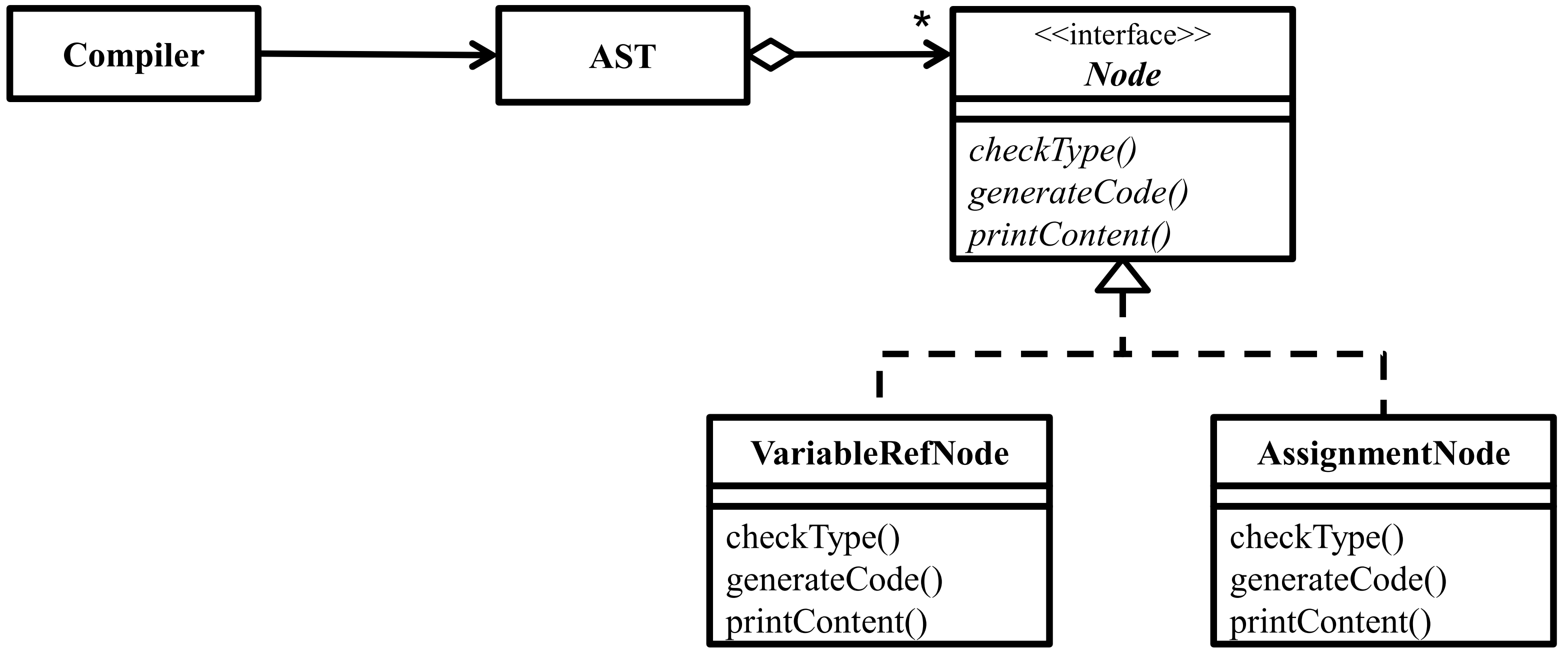


## Requirements Statements$_2$

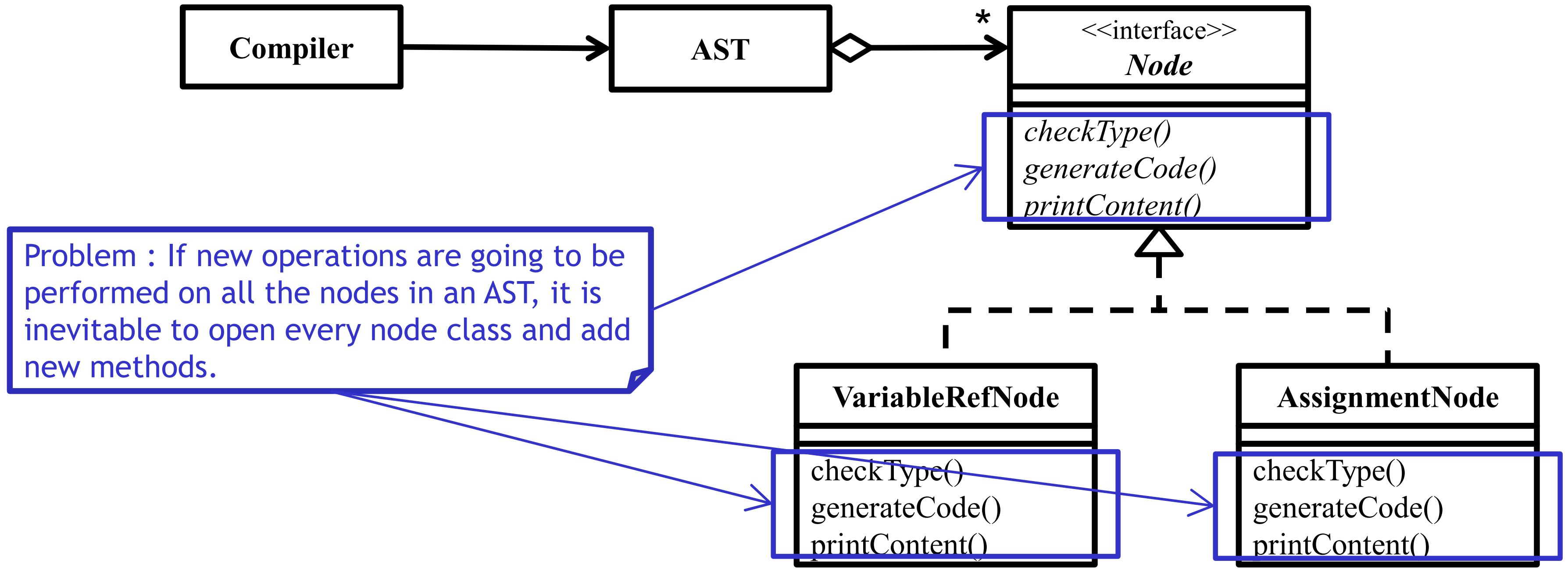
- Each node currently provides three interfaces for the compiler to use in order to check its type, generate code and print out the content.



# Initial Design

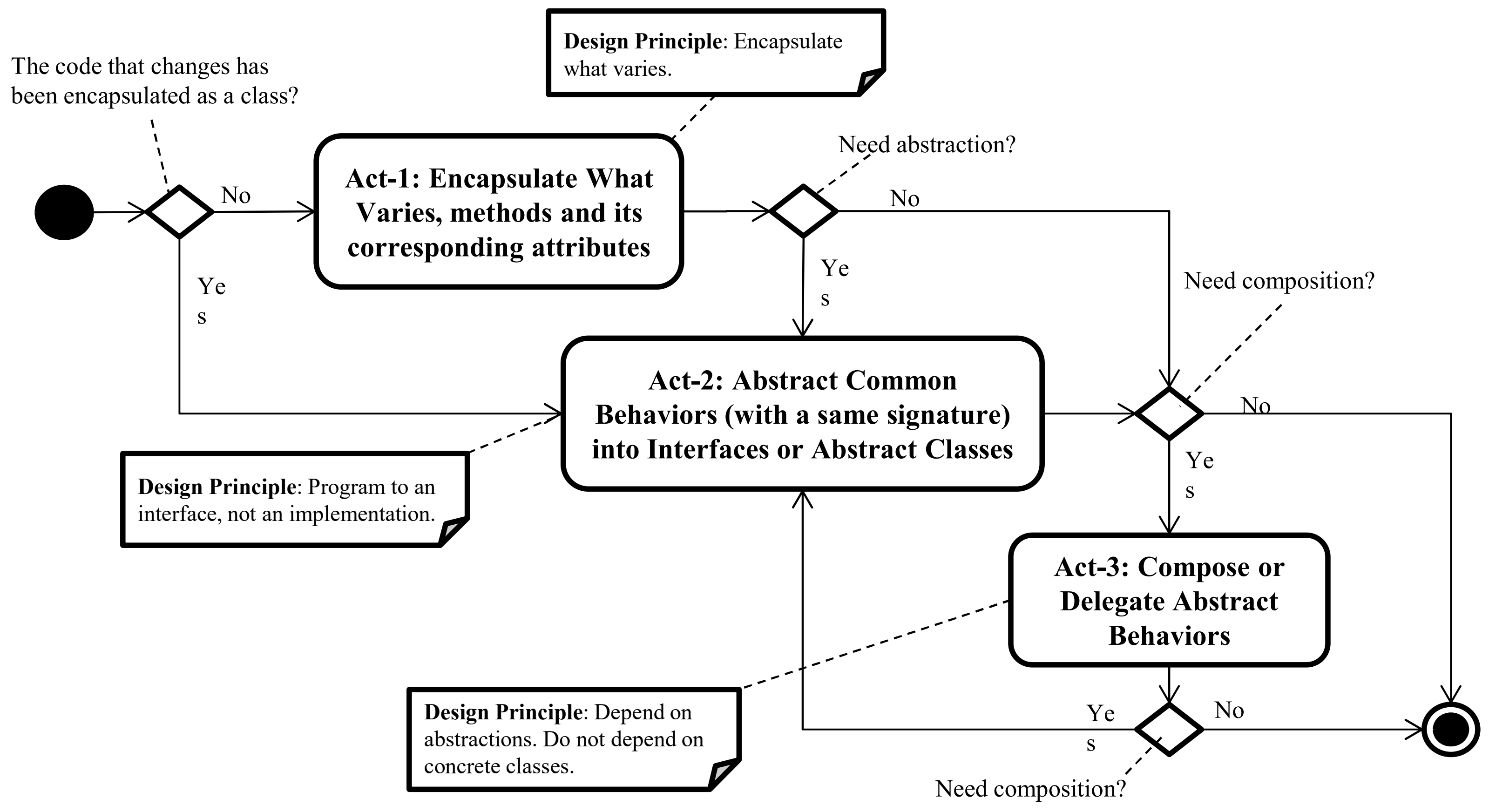


# Problem with the Initial Design



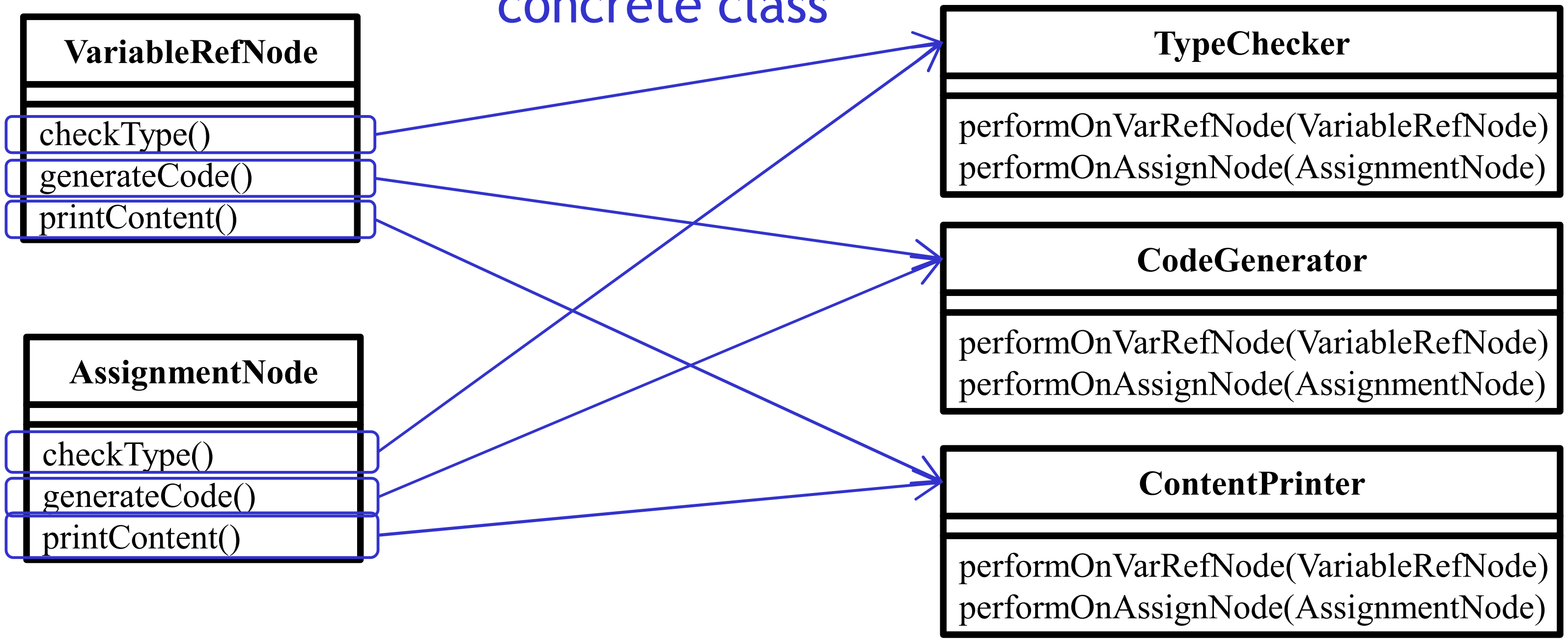
Problem : If new operations are going to be performed on all the nodes in an AST, it is inevitable to open every node class and add new methods.

# Design Process for Change

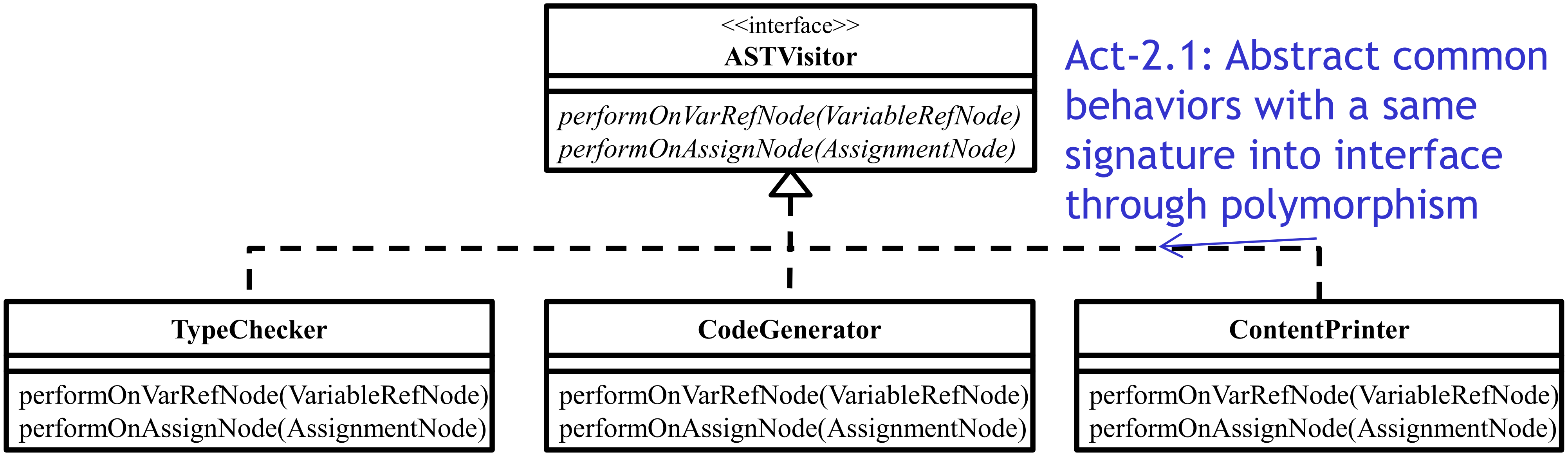

The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Ye s
Ye s
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Ye s
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Ye s
No
Need composition?

# Act-1: Encapsulate What Varies

Act-1.2: Encapsulate a method into a concrete class


VariableRefNode
checkType()
generateCode()
printContent()
AssignmentNode
checkType()
generateCode()
printContent()
TypeChecker
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
CodeGenerator
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
ContentPrinter
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

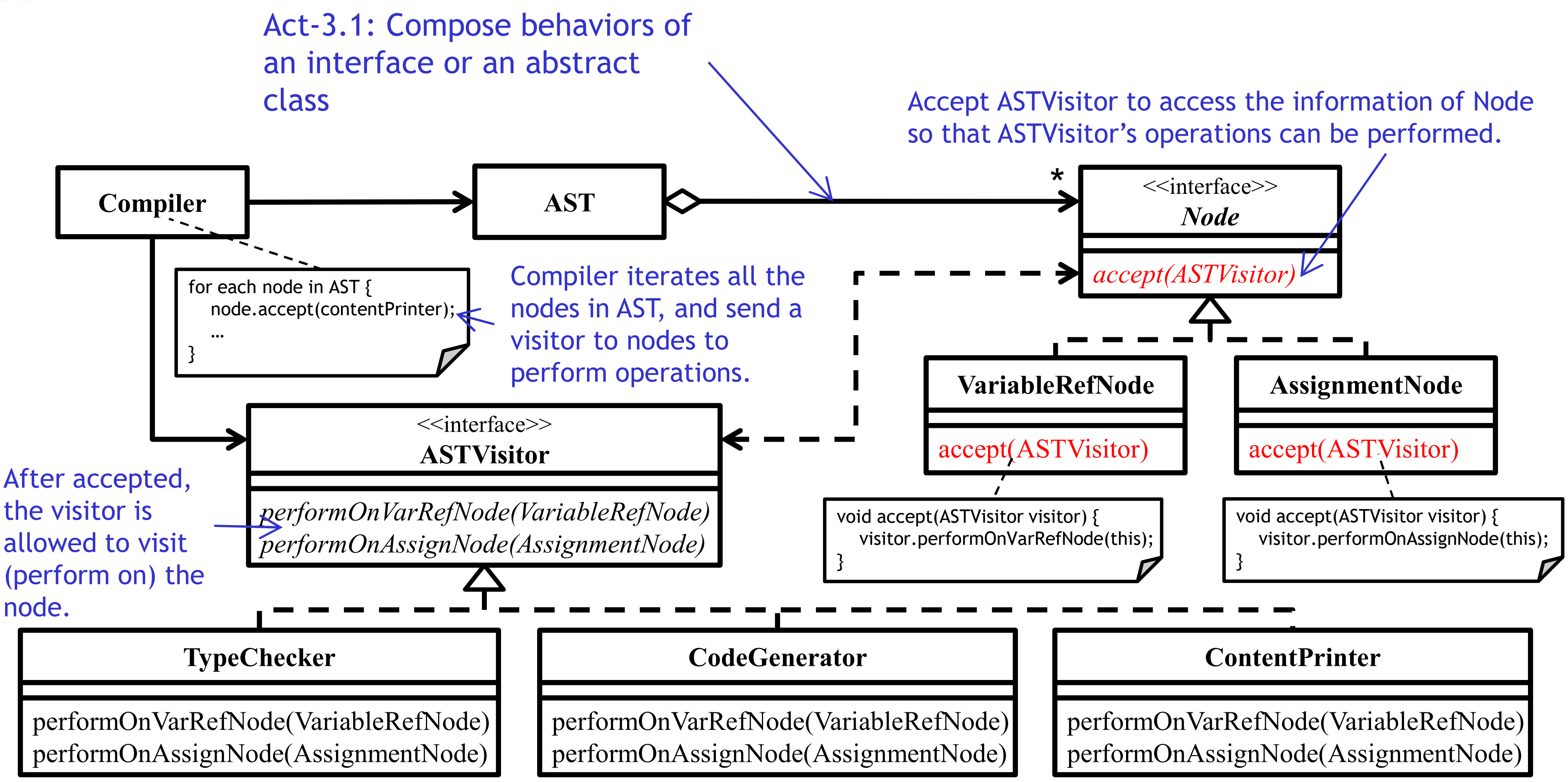
# Act-2: Abstract Common Behaviors



<<interface>>
ASTVisitor
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
TypeChecker
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
CodeGenerator
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
ContentPrinter
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

# Act-3: Compose Abstract Behaviors



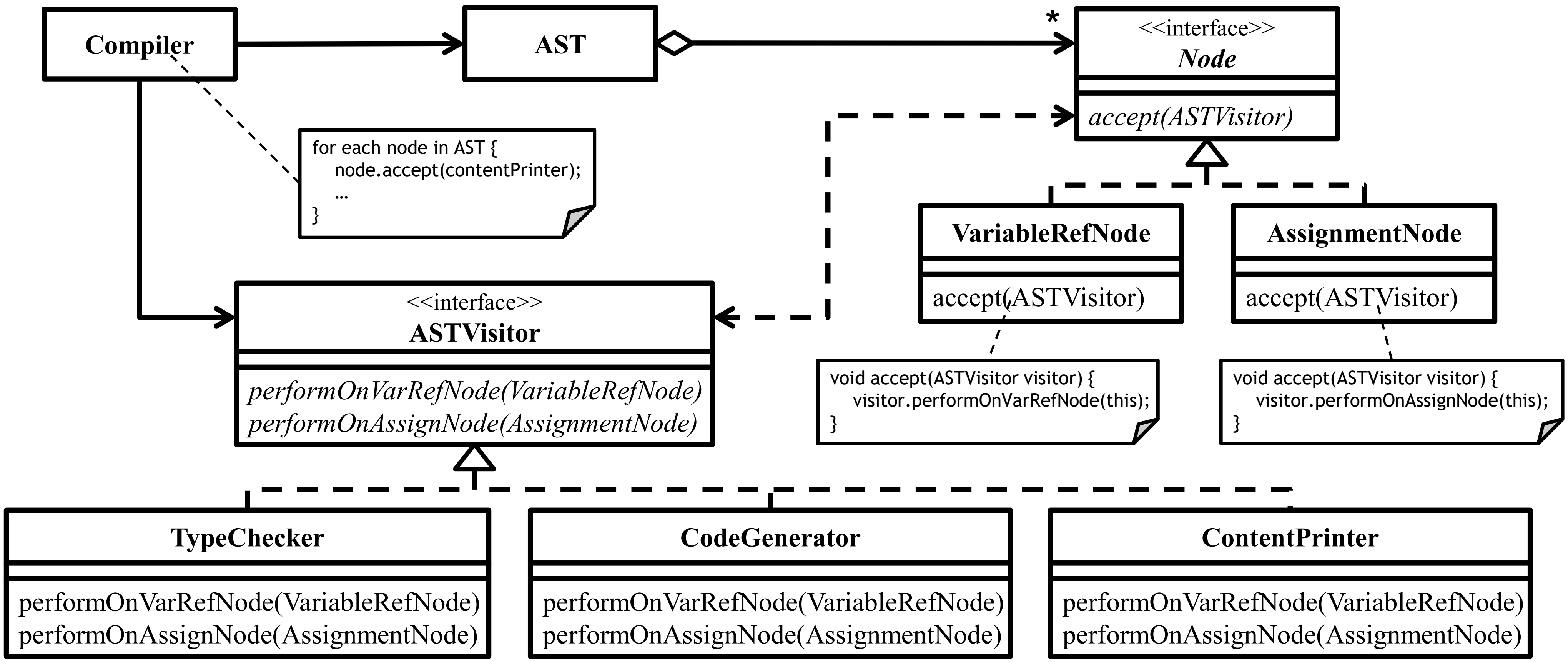
Act-3.1: Compose behaviors of an interface or an abstract class
Accept ASTVisitor to access the information of Node so that ASTVisitor's operations can be performed.
Compiler
AST
*
<<interface>>
Node
accept(ASTVisitor)
Compiler iterates all the nodes in AST, and send a visitor to nodes to perform operations.
VariableRefNode
accept(ASTVisitor)
AssignmentNode
accept(ASTVisitor)
<<interface>>
ASTVisitor
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
After accepted, the visitor is allowed to visit (perform on) the node.
TypeChecker
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
CodeGenerator
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
ContentPrinter
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

```
for each node in AST {
    node.accept(contentPrinter);
    …
}
```

```
void accept(ASTVisitor visitor) {
    visitor.performOnVarRefNode(this);
}
```

```
void accept(ASTVisitor visitor) {
    visitor.performOnAssignNode(this);
}
```

# Refactored Design after Design Process



Compiler
AST
*
<<interface>>
Node
accept(ASTVisitor)
for each node in AST {
node.accept(contentPrinter);
…
}
VariableRefNode
accept(ASTVisitor)
AssignmentNode
accept(ASTVisitor)
<<interface>>
ASTVisitor
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
void accept(ASTVisitor visitor) {
visitor.performOnVarRefNode(this);
}
void accept(ASTVisitor visitor) {
visitor.performOnAssignNode(this);
}
TypeChecker
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
CodeGenerator
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
ContentPrinter
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

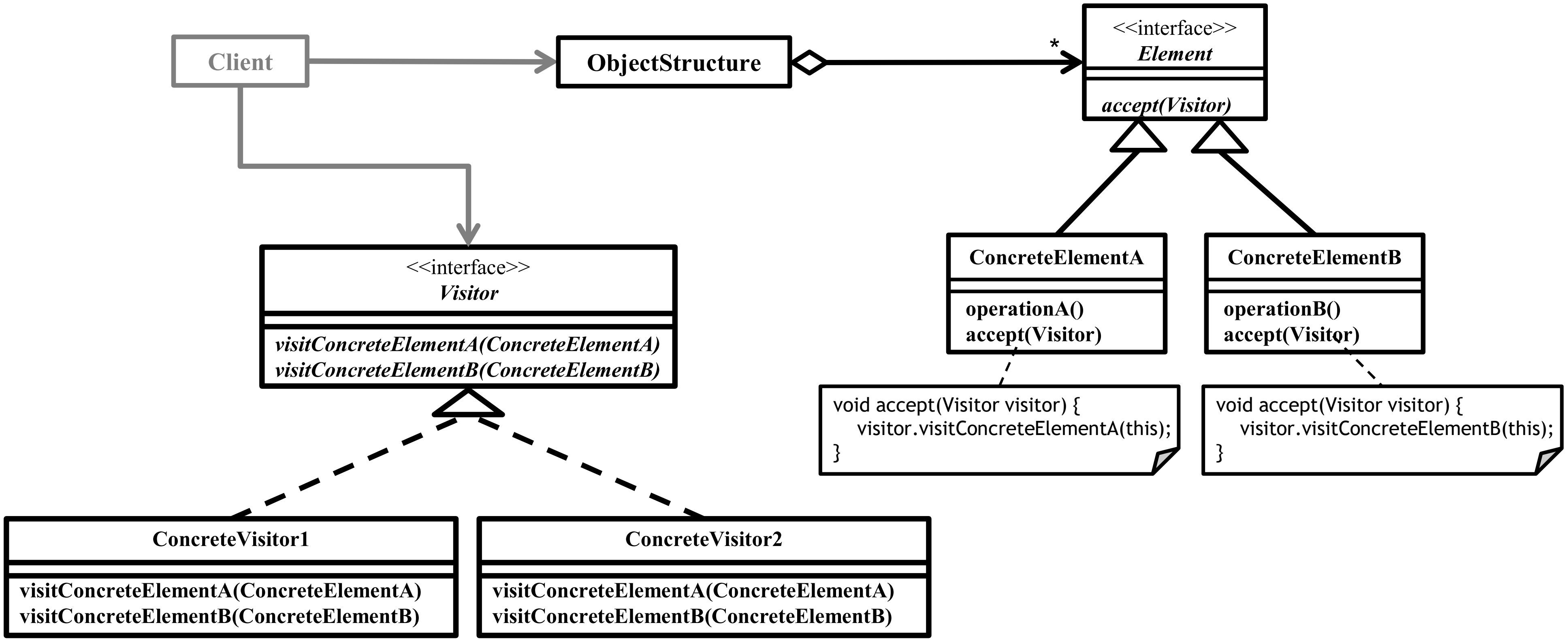
## Recurrent Problem

- The problem is that distributing all these operations across the various classes in an object structure leads to a system that's hard to understand, maintain, and change. Moreover, adding a new operation usually requires recompiling all of these classes.

## Intent

- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

# Visitor Pattern Structure$_1$



Client
ObjectStructure
*
<<interface>>
Element
accept(Visitor)
<<interface>>
Visitor
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteElementA
operationA()
accept(Visitor)
ConcreteElementB
operationB()
accept(Visitor)
ConcreteVisitor1
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

```
void accept(Visitor visitor) {
    visitor.visitConcreteElementA(this);
}
```

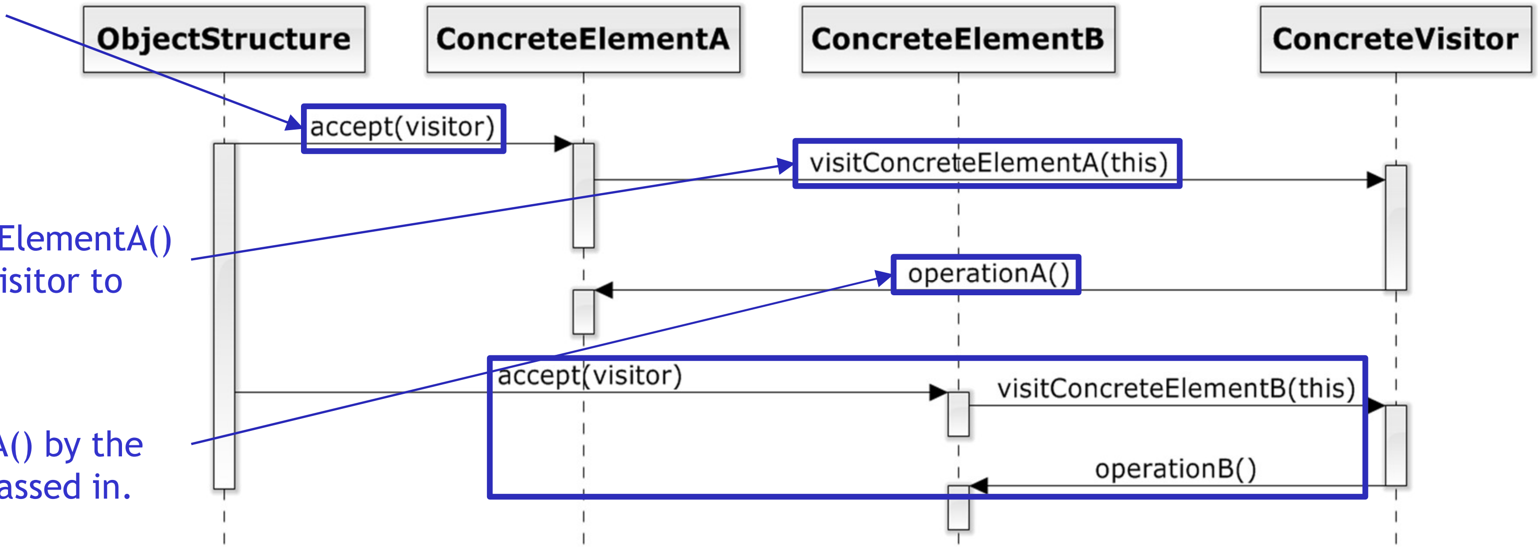
```
void accept(Visitor visitor) {
    visitor.visitConcreteElementB(this);
}
```

# Visitor Pattern Structure$_2$

1. ObjectStructure invokes ConcreteEmelemtA's accept() and passes a ConcreteVisitor instance.

2. ConcreteElementA invokes ConcreteVisitor's visitConcreteElementA() and passes itself to allow the visitor to access its state.

3. ConcreteVisitor invokes ConcreteElementA's operationA() by the ConcreteElementA reference passed in.



4~6. The same process as 1~3 for ConcreteElementB.

# Visitor Pattern Structure$_3$

| | Instantiation | Use | Termination |
|---|---|---|---|
| **Visitor** | X | ConcreteElement invokes Visitor's visit method through polymorphism. | X |
| **ConcreteVisitor** | Client | Client passes ConcreteVisitor to ObjectStructure, and ObjectStructure invokes the accept() of Element with the ConcreteVisitor. In the accept() of Element, the visit method of ConcreteVisitor is invoked and Element passes itself to the visit method so that the visitor can **access the state of Element.** | Client |
| **Element** | X | Element provides accept() that allows ObjectStructure to pass Visitor to Element. | X |
| **ConcreteElement** | Don't Care | **ConcreteElement realizes the accept() to allow Visitor accessing the state of ConcreteElement**. | Don't Care |
| **ObjectStructure** | Don't Care | ObjectStructure that consists of multiple Elements invokes the accept() of Element and passes ConcreteVisitor to Element. | Don't Care |

## Double-Dispatch

- "Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of two receivers.

- accept() is a double-dispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Double-dispatching lets visitors request different operations on each class of element.

- Instead of binding operations statically into the Element interface, you can consolidate the operations in a Visitor and use accept() to do the binding at run-time.

- Extending the Element interface amounts to defining one new Visitor subclass rather than many new Element subclasses.

# Nutrition Retrieval from A Restaurant Menu (Visitor)

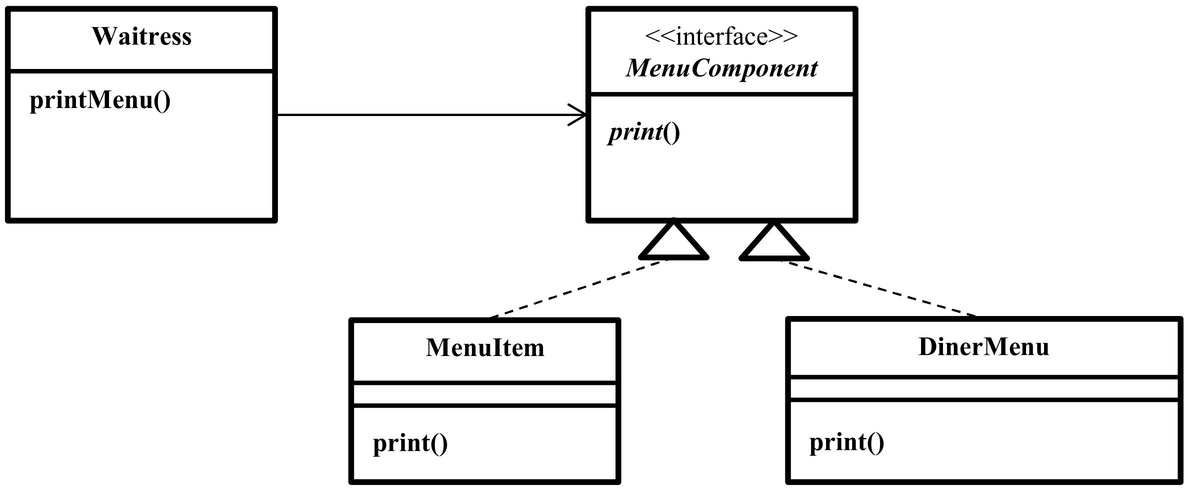
Shin-Jie Lee (李信杰)
Assistant Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

## Requirements Statements

- The menu components of the Diner restaurant which comprises menu items and diner menus can be printed by a waitress.
- Each diner menu consists of several menu items.
- The Diner restaurant would like to provide calories, protein and carbs information for each menu item.

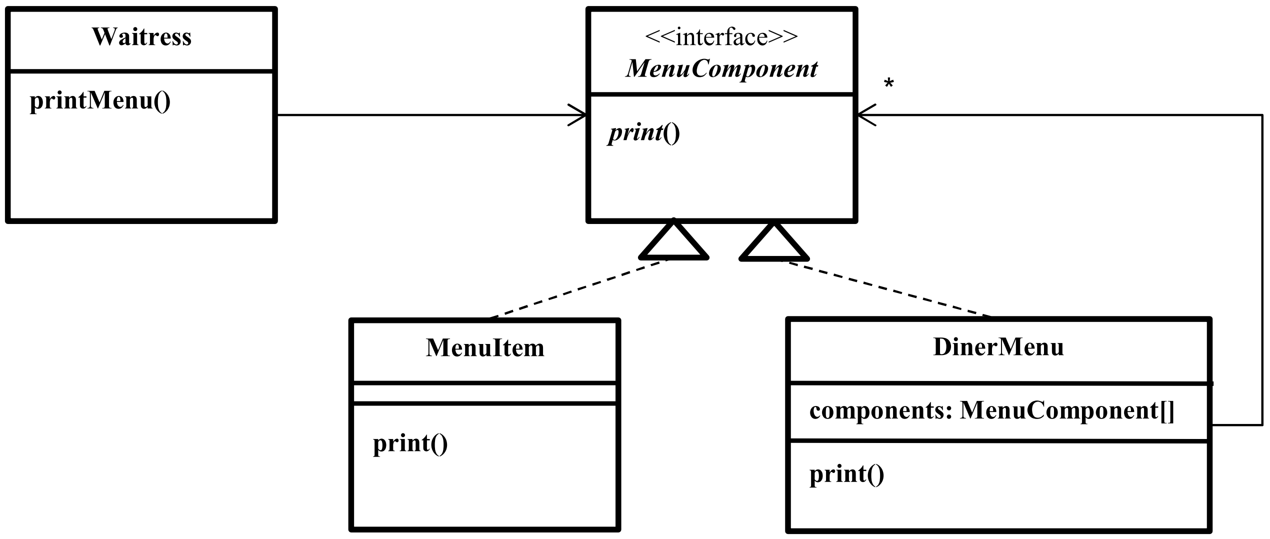
## Requirements Statements$_1$

- The menu components of the Diner restaurant which comprises menu items and diner menus can be printed by a waitress.



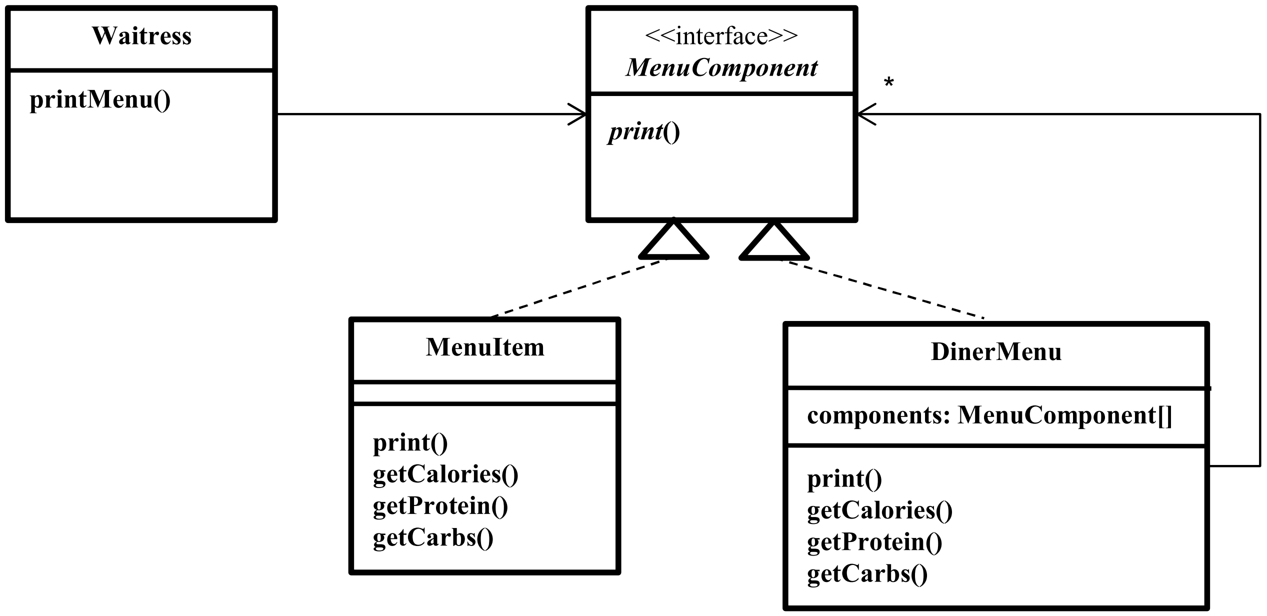
## Requirements Statements$_2$

- Each diner menu consists of several menu items.

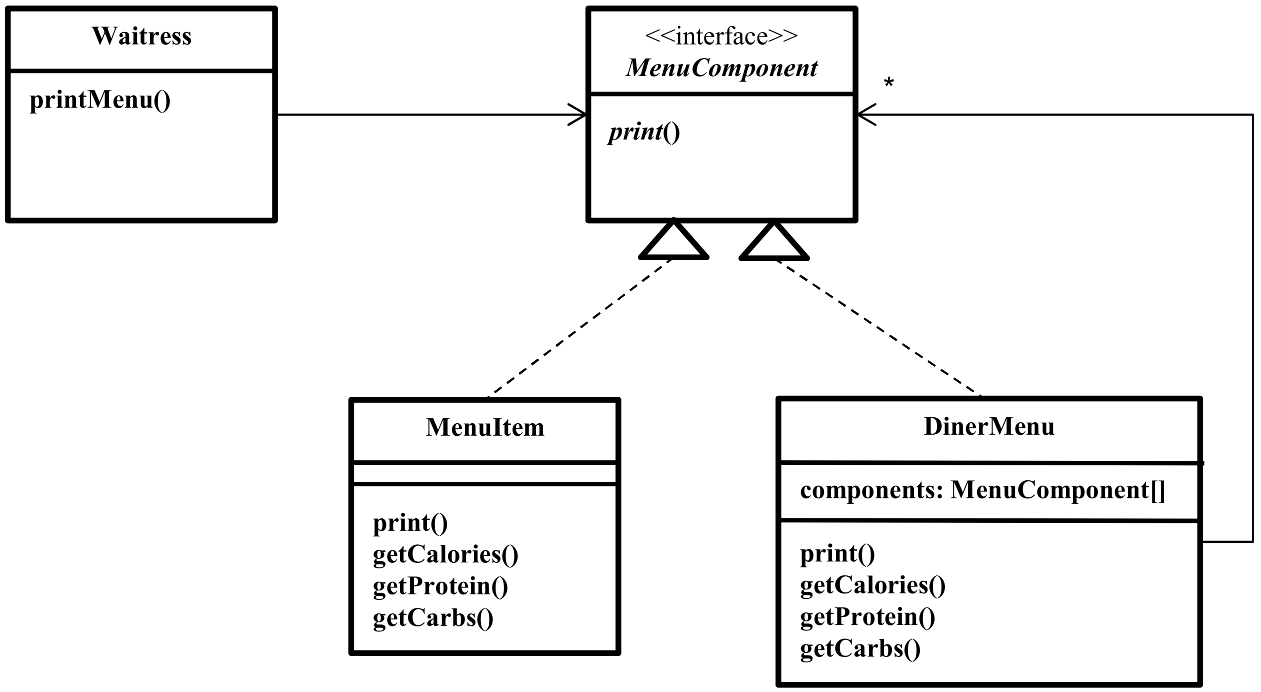


## Requirements Statements$_3$

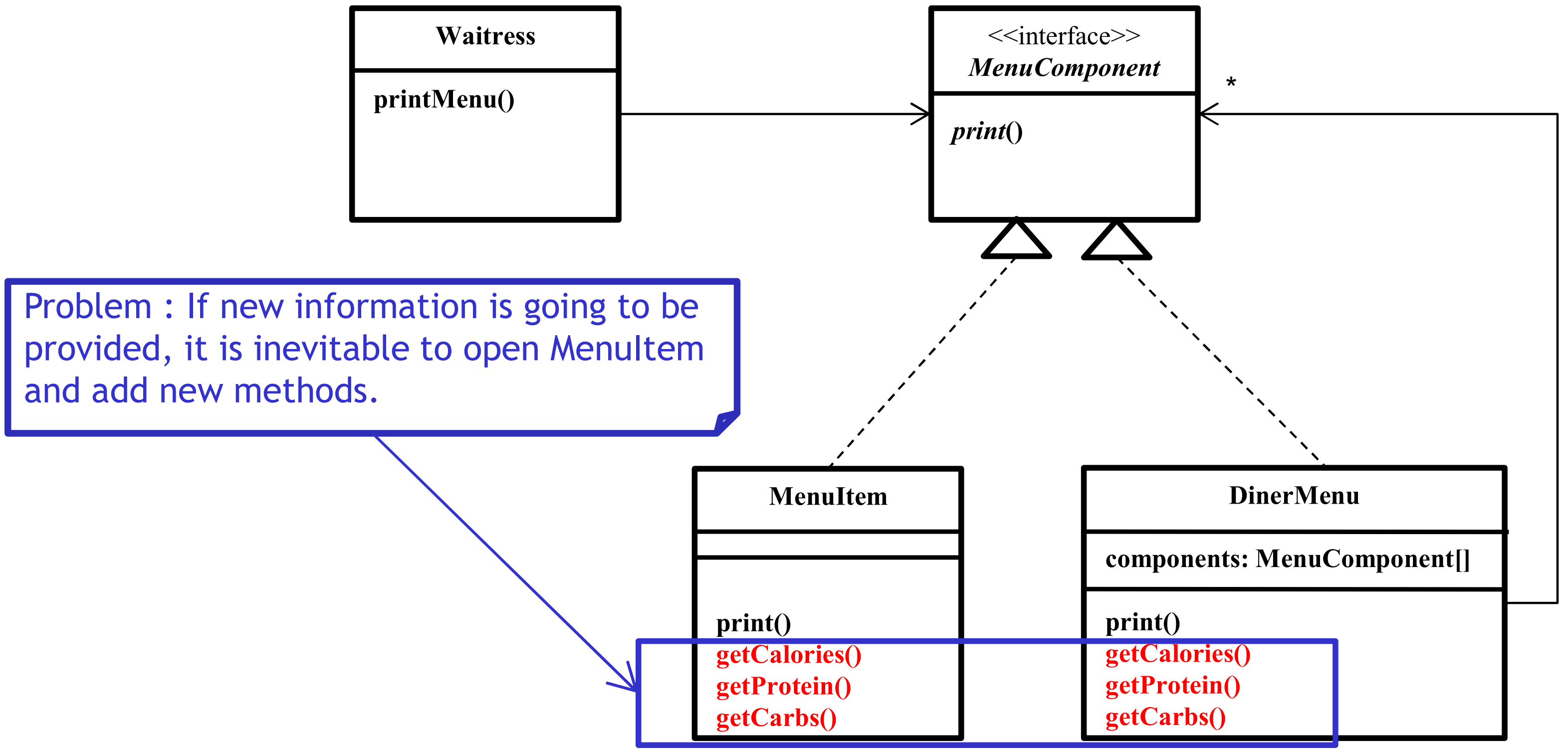
- The Diner restaurant would like to provide calories, protein and carbs information for each menu item and each DinerMenu.



# Initial Design

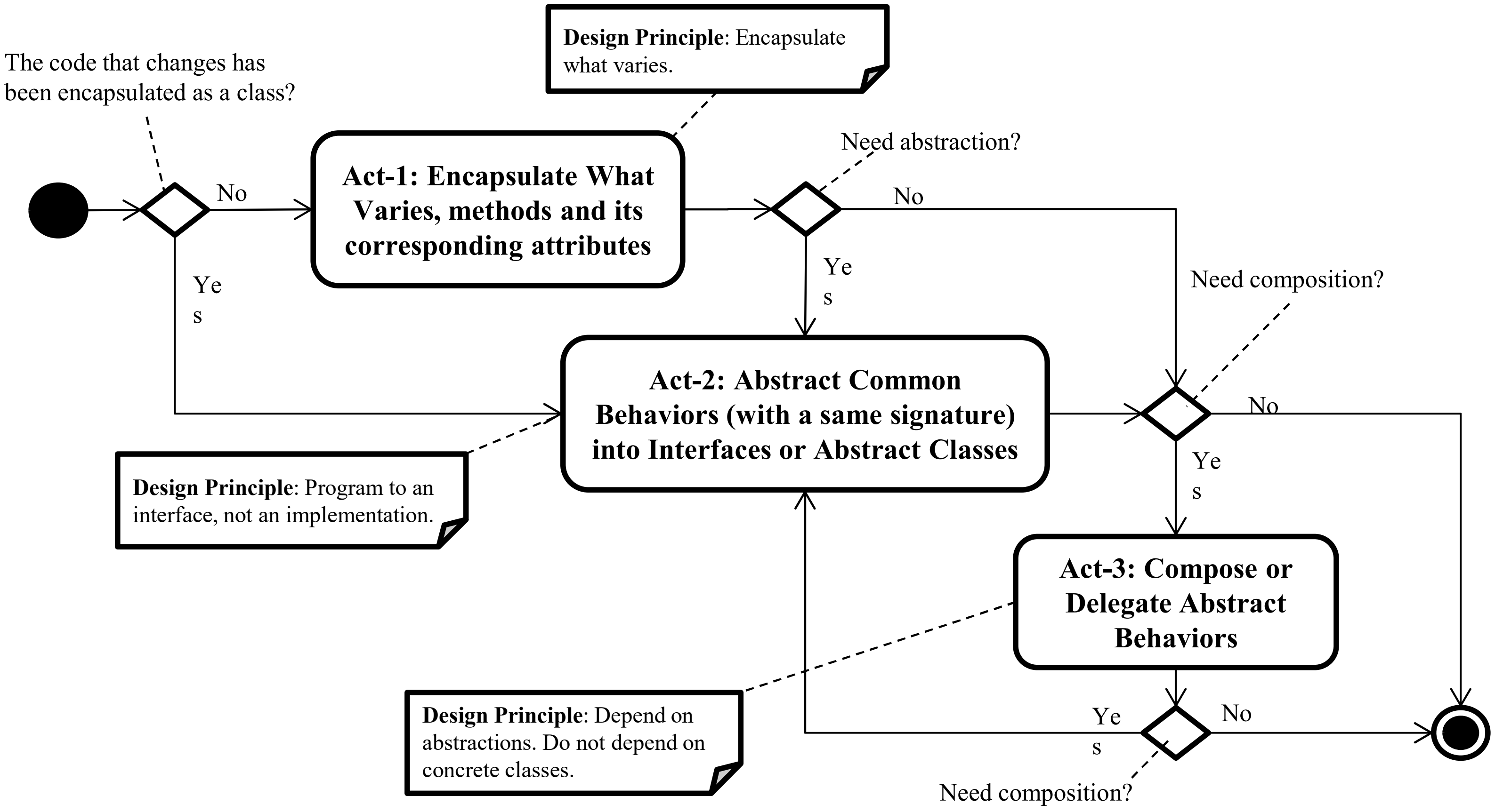


# The Problem with the Initial Design



Problem : If new information is going to be provided, it is inevitable to open MenuItem and add new methods.

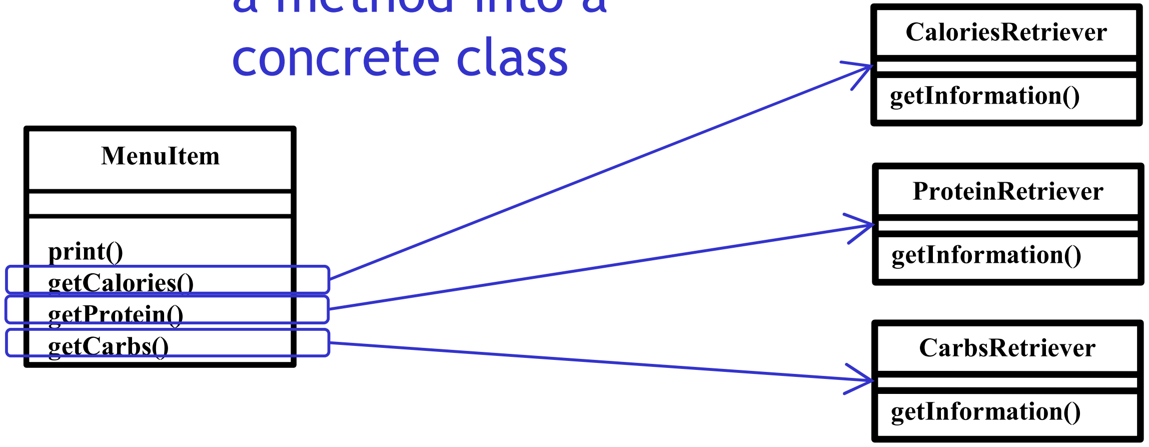
# Design Process for Change



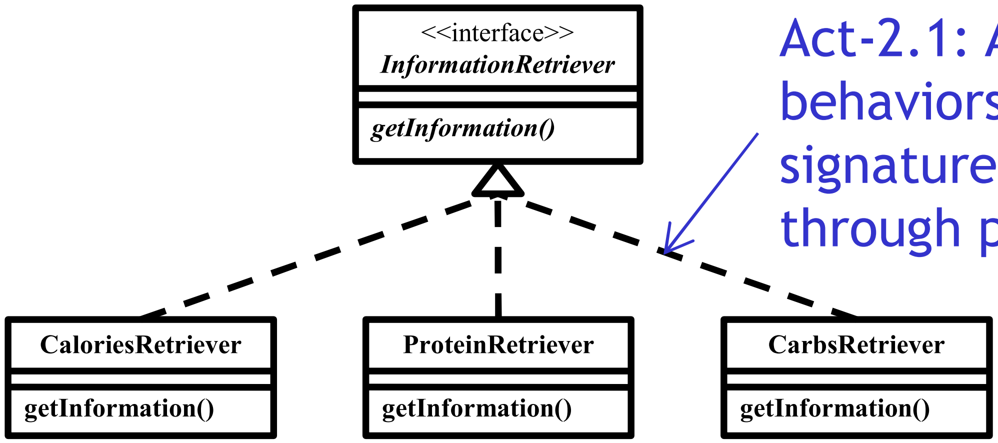
The code that changes has been encapsulated as a class?
No
Yes
Design Principle: Encapsulate what varies.
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

# Act-1: Encapsulate What Varies

Act-1.2: Encapsulate a method into a concrete class

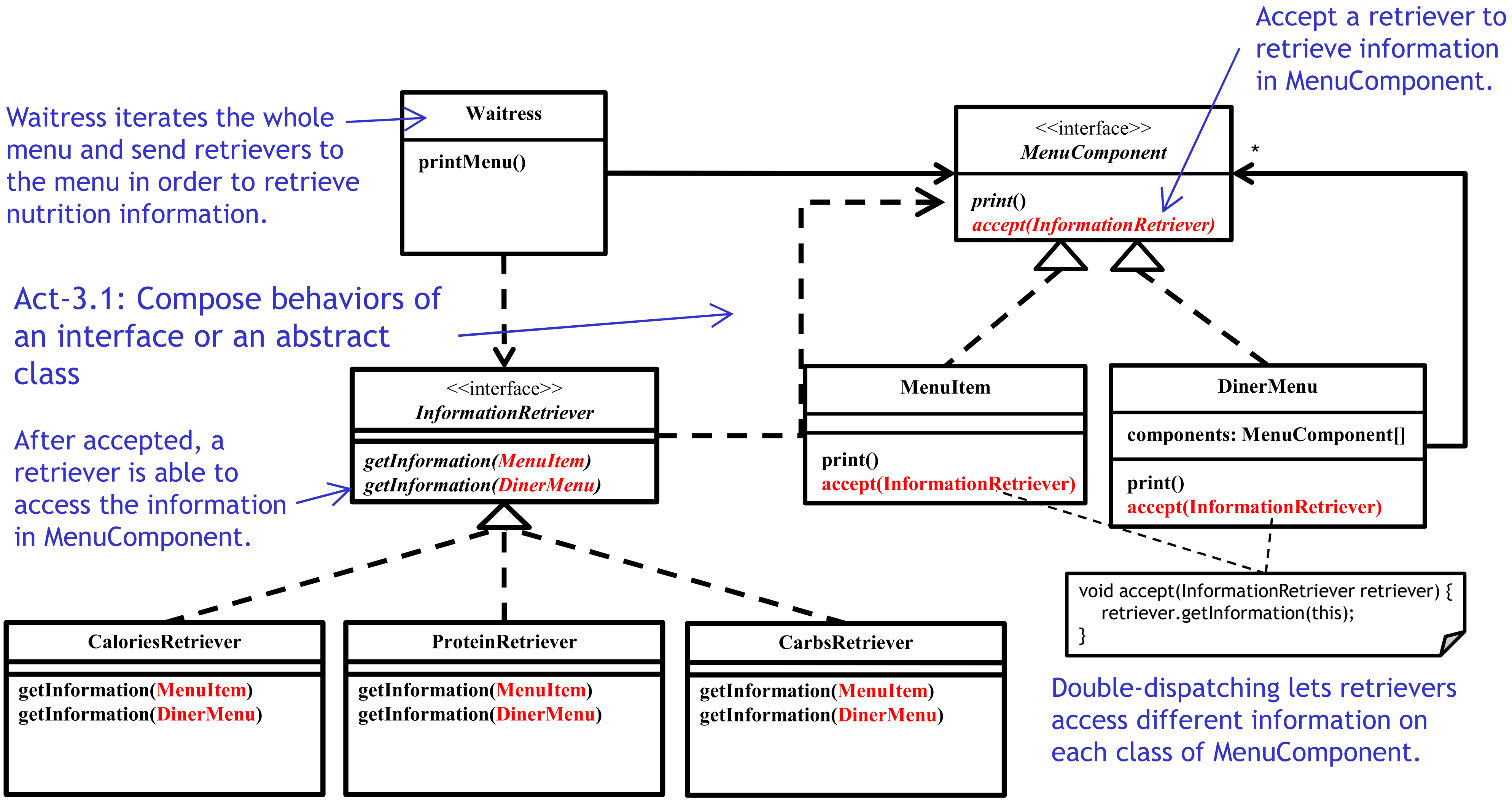


## Act-2: Abstract Common Behaviors



Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

# Act-3: Compose Abstract Behaviors

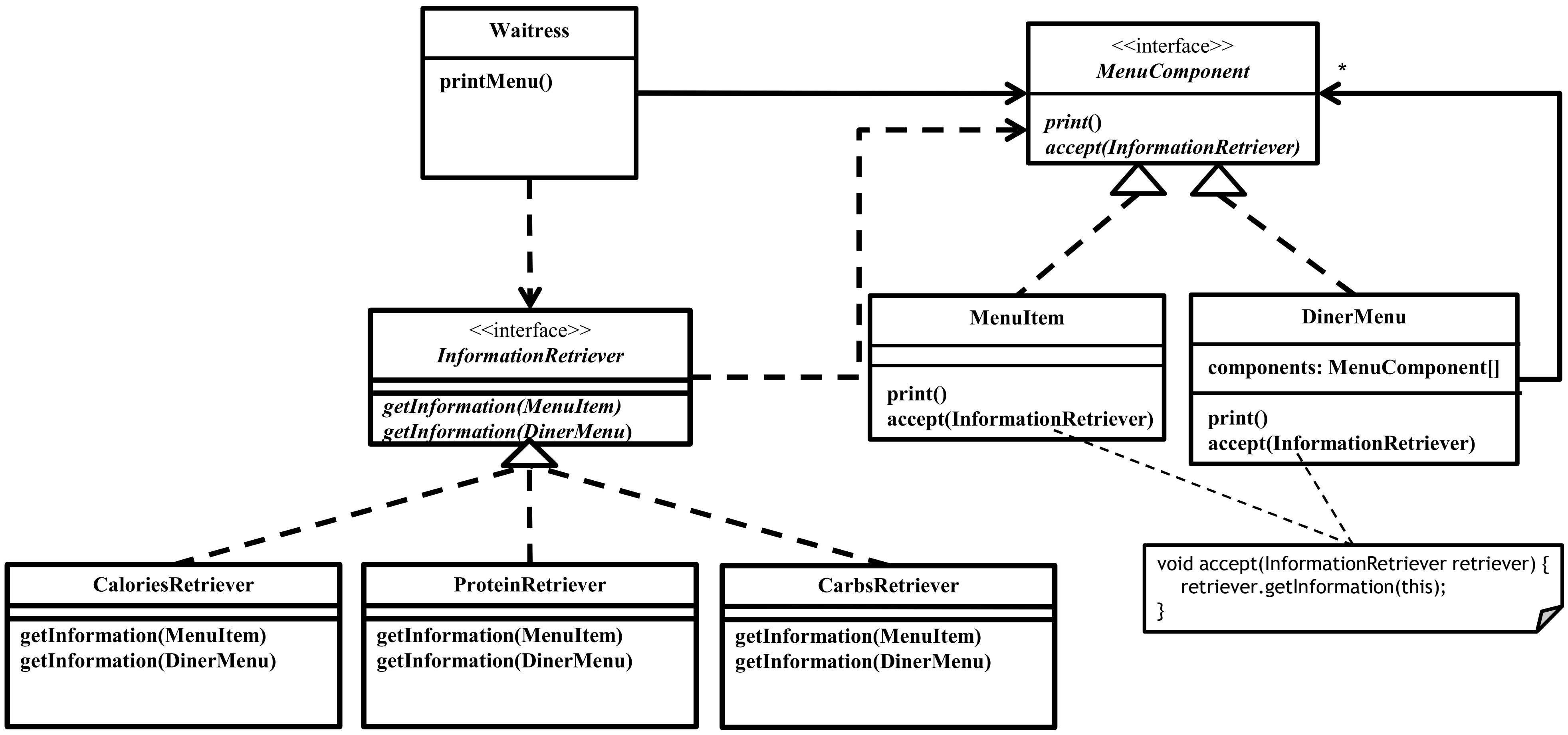


Accept a retriever to retrieve information in MenuComponent.
Waitress iterates the whole menu and send retrievers to the menu in order to retrieve nutrition information.
Waitress
printMenu()
<<interface>>
MenuComponent
print()
accept(InformationRetriever)
*
Act-3.1: Compose behaviors of an interface or an abstract class
<<interface>>
InformationRetriever
getInformation(MenuItem)
getInformation(DinerMenu)
After accepted, a retriever is able to access the information in MenuComponent.
MenuItem
print()
accept(InformationRetriever)
DinerMenu
components: MenuComponent[]
print()
accept(InformationRetriever)
CaloriesRetriever
getInformation(MenuItem)
getInformation(DinerMenu)
ProteinRetriever
getInformation(MenuItem)
getInformation(DinerMenu)
CarbsRetriever
getInformation(MenuItem)
getInformation(DinerMenu)

```
void accept(InformationRetriever retriever) {
    retriever.getInformation(this);
}
```

Double-dispatching lets retrievers access different information on each class of MenuComponent.

## Refactored Design after Design Process



```
void accept(InformationRetriever retriever) {
    retriever.getInformation(this);
}
```

# Equipment Power Consumption (Visitor)

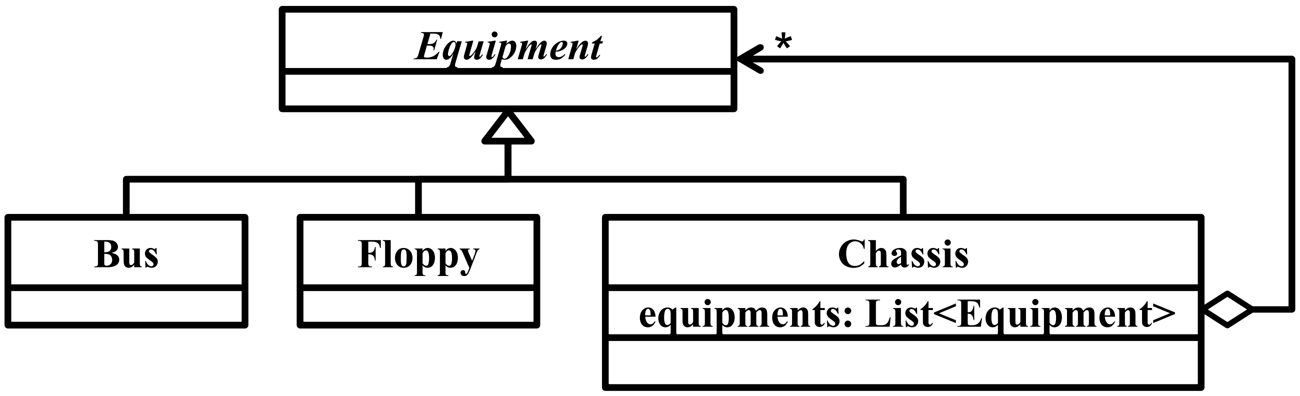
Shin-Jie Lee (李信杰)
Assistant Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

## Requirements Statements

- There are three types of equipment in the inventory, such as chassis, buses, and floppies. Among all the equipment, Chassis is composited of others.
- Chassis provides an interface for the creation of an iterator, which iterates all the equipment in one chassis with next and hasNext operations.
- Each equipment provides its power consumption and cost in addition.
- Chassis provides a sum of power consumption or cost for all its components.

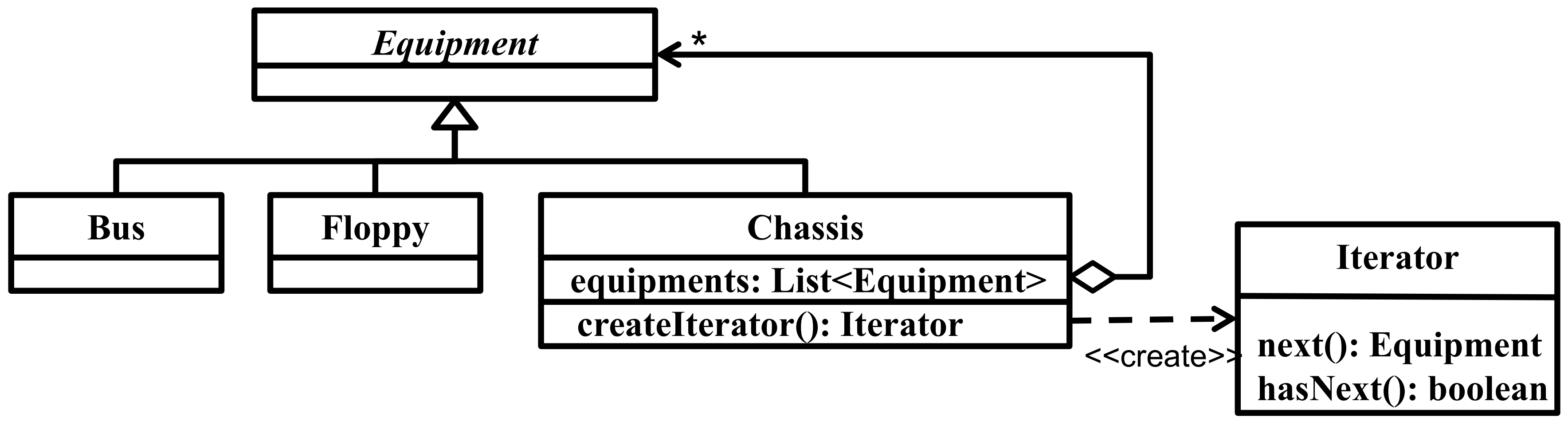
## Requirements Statements$_1$

- There are three types of equipment in the inventory, such as chassis, buses, floppies and drivers. Among all the equipment, Chassis is composited of others.



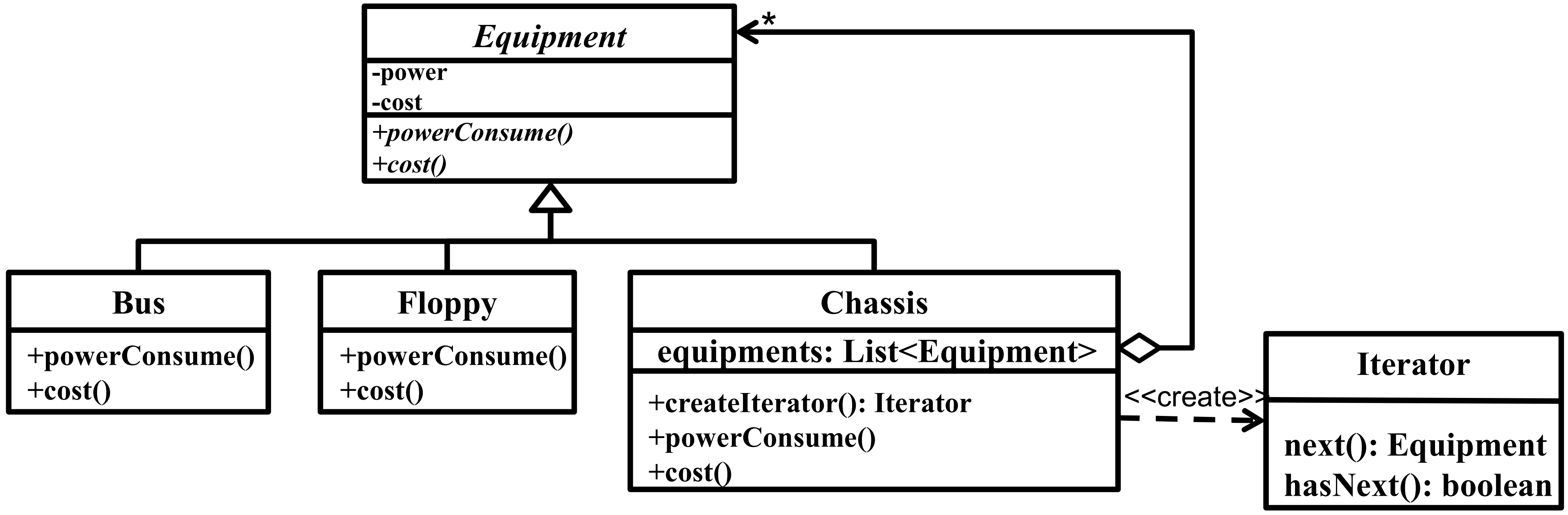
# Requirements Statements$_2$

- ❑ Chassis provides an interface for the creation of an iterator, which iterates all the equipments in one chassis with next and hasNext operations.



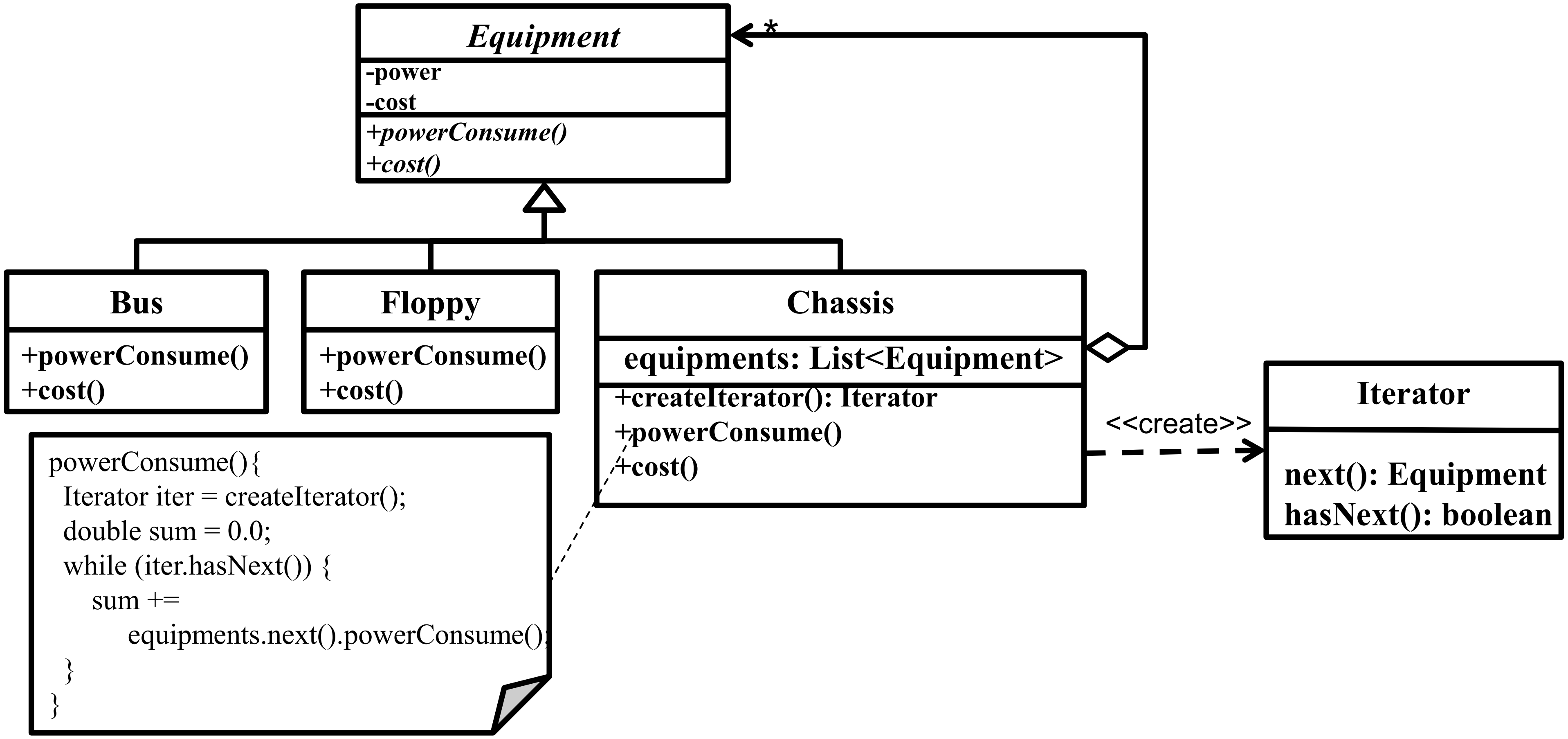
# Requirements Statements$_3$

- ❑ Each equipment provides its power consumption and cost in addition.



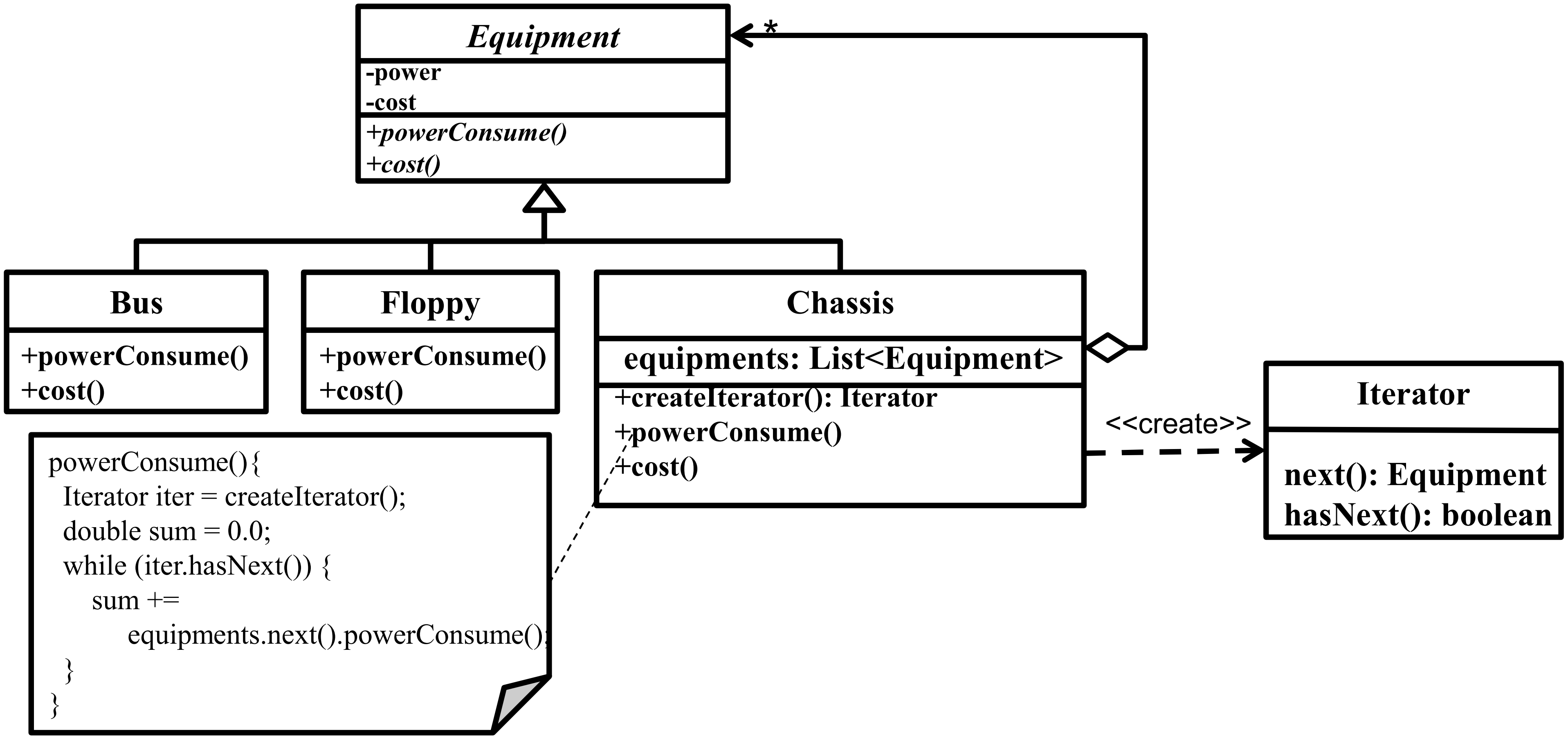
## Requirements Statements$_4$

- Chassis provides a sum of power consumption or cost for all its components.



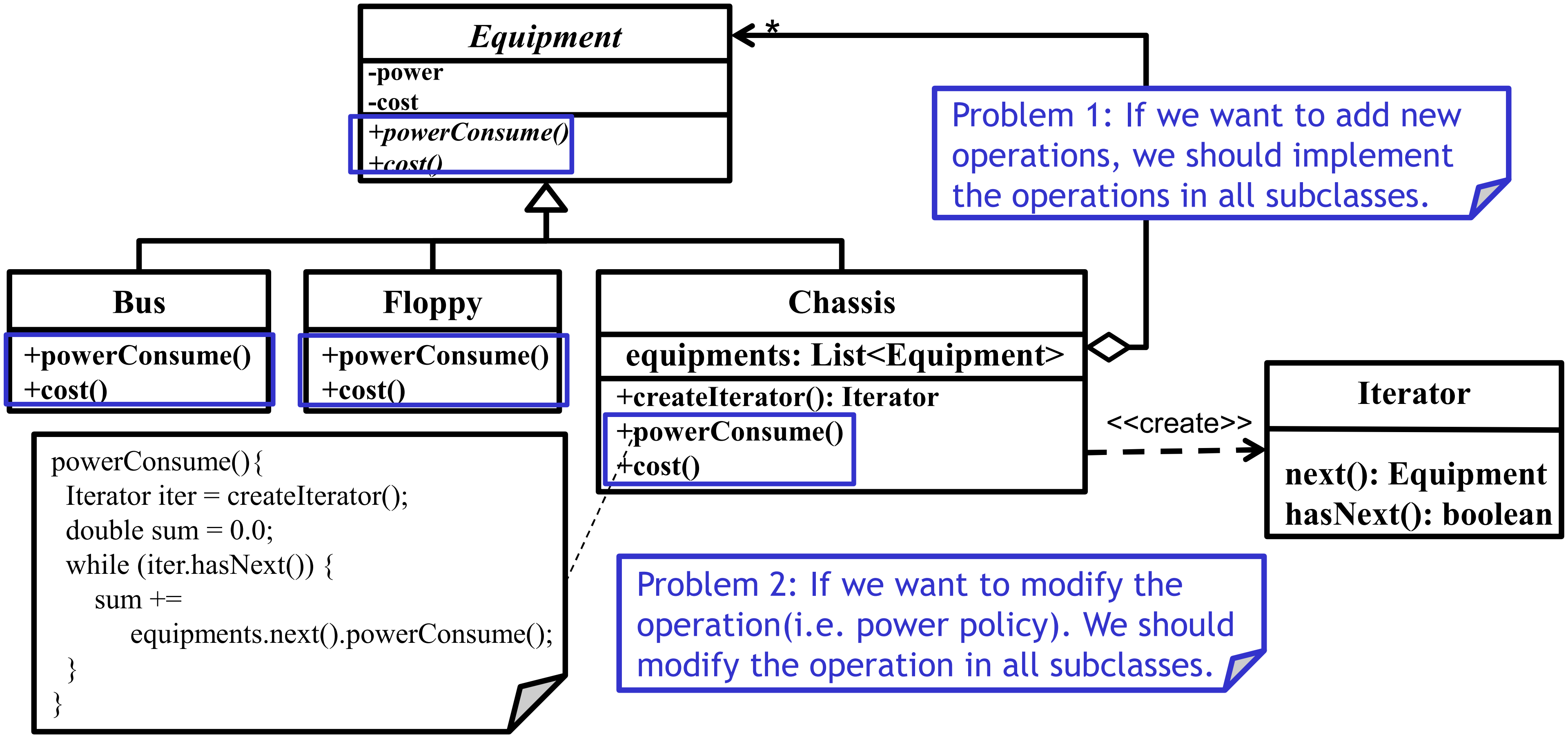
```
powerConsume(){
  Iterator iter = createIterator();
  double sum = 0.0;
  while (iter.hasNext()) {
      sum +=
           equipments.next().powerConsume();
  }
}
```

# Initial Design



```
powerConsume(){
  Iterator iter = createIterator();
  double sum = 0.0;
  while (iter.hasNext()) {
      sum +=
          equipments.next().powerConsume();
  }
}
```

# Problems with Initial Design

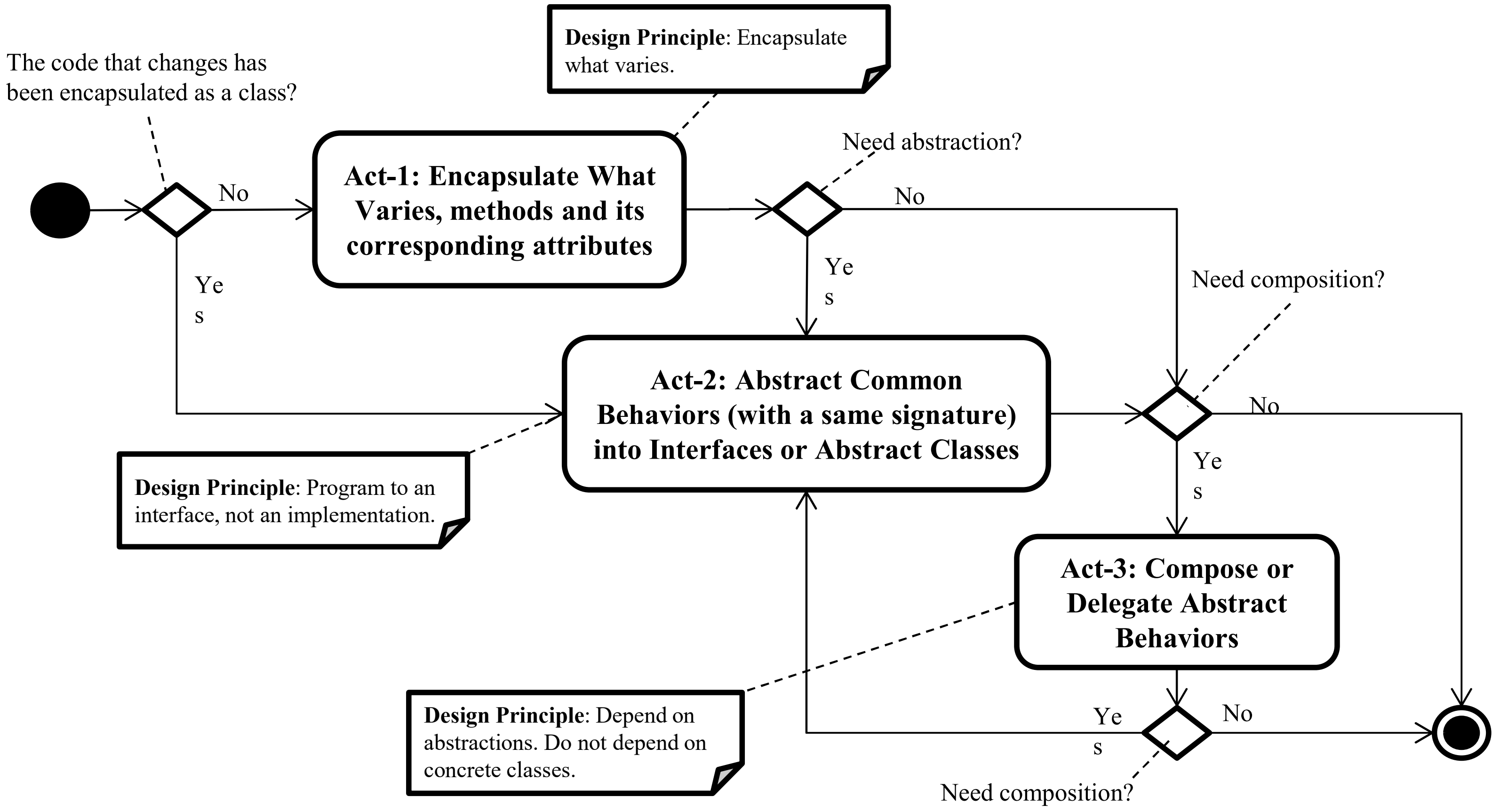


```
powerConsume(){
  Iterator iter = createIterator();
  double sum = 0.0;
  while (iter.hasNext()) {
      sum +=
          equipments.next().powerConsume();
  }
}
```

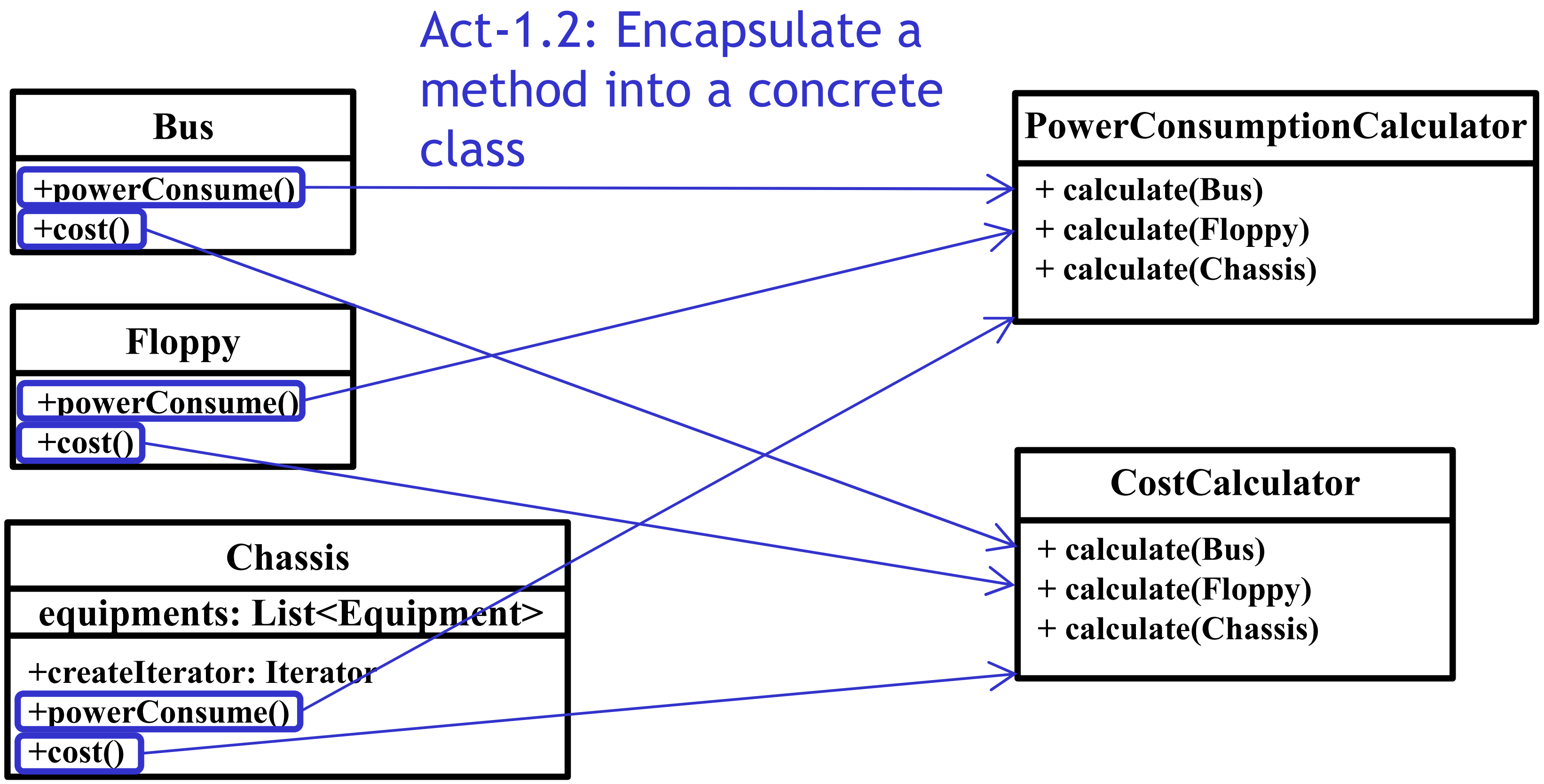
Problem 1: If we want to add new operations, we should implement the operations in all subclasses.

Problem 2: If we want to modify the operation(i.e. power policy). We should modify the operation in all subclasses.

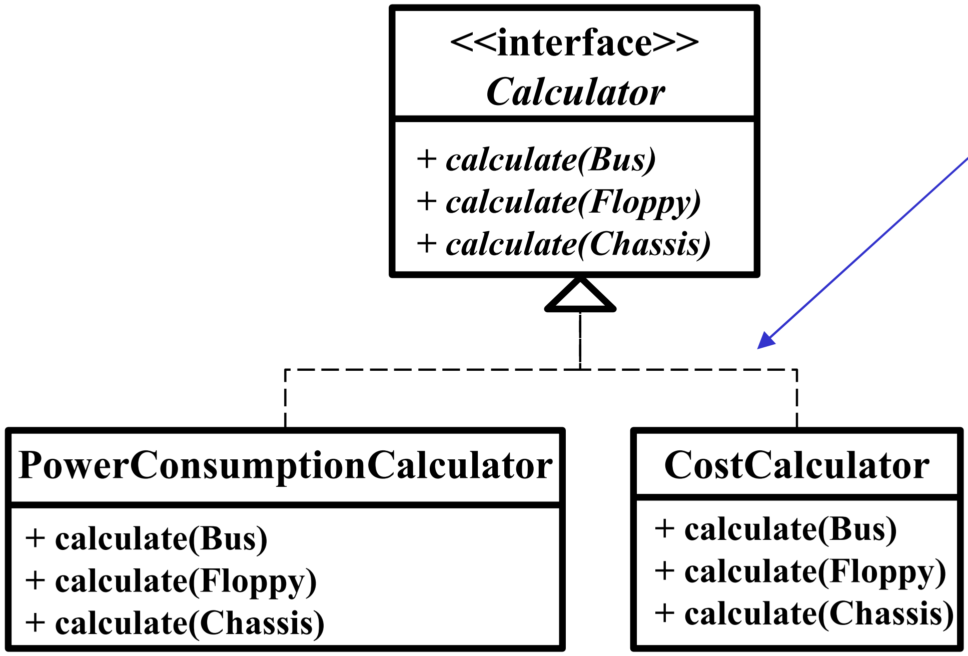
# Design Process for Change


Design Principle: Encapsulate what varies.
The code that changes has been encapsulated as a class?
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

# Act-1: Encapsulate What Varies

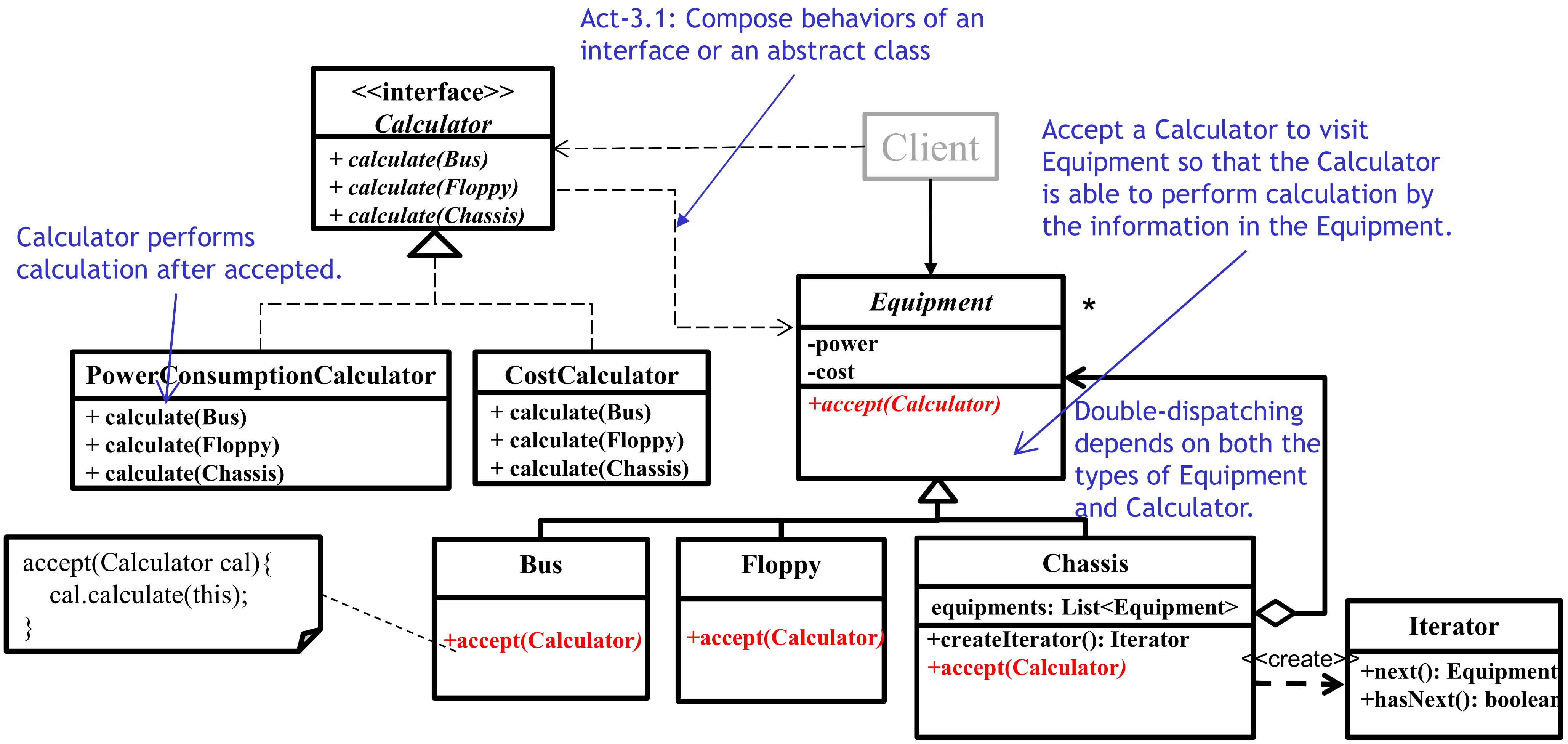


# Act-2: Abstract Common Behaviors

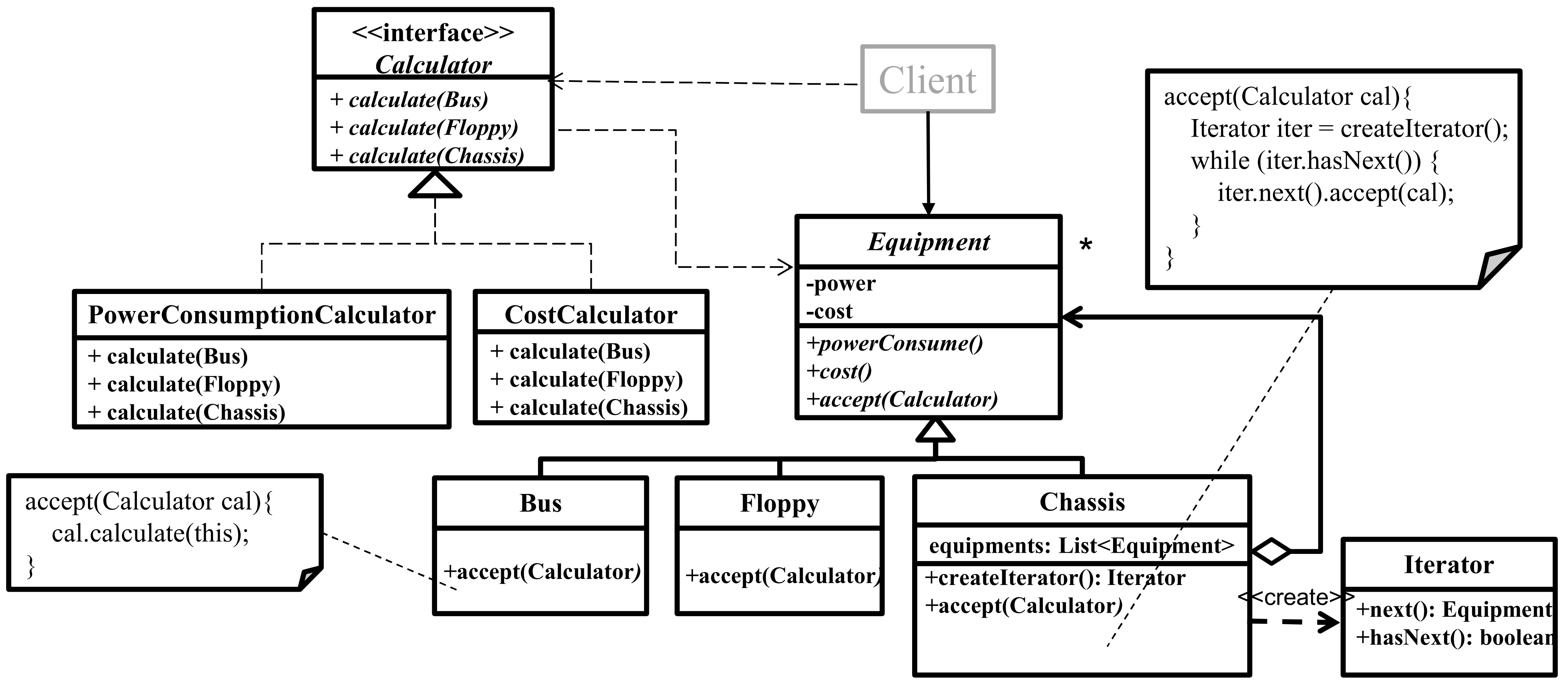


Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

# Act-3: Compose Abstract Behaviors



## Refactored Design after Design Process



<<interface>>
*Calculator*
+ *calculate(Bus)*
+ *calculate(Floppy)*
+ *calculate(Chassis)*
Client
accept(Calculator cal){
Iterator iter = createIterator();
while (iter.hasNext()) {
iter.next().accept(cal);
}
}
*Equipment*
*
-power
-cost
+*powerConsume()*
+*cost()*
+*accept(Calculator)*
PowerConsumptionCalculator
+ calculate(Bus)
+ calculate(Floppy)
+ calculate(Chassis)
CostCalculator
+ calculate(Bus)
+ calculate(Floppy)
+ calculate(Chassis)
accept(Calculator cal){
cal.calculate(this);
}
Bus
+accept(Calculator)
Floppy
+accept(Calculator)
Chassis
equipments: List<Equipment>
+createIterator(): Iterator
+accept(Calculator)
<<create>>
Iterator
+next(): Equipment
+hasNext(): boolean