A Command-Line Based Task Management System, Group 24

Wang Ruijie, Zeng Tianyi, Zhu Jin Shun and Liu Yuyang

Deparment of Computing, The Hong Kong Polytechnic University

Introduction

The Command-Line Based Task Management System (TMS) is aimed at achieving, and has realized the following expectations:

- The creation, modification, record and display of user-defined tasks;
- The definement, management and utilization of user-defined criteria for task filtering;
- The friendly interaction through a command-line user interface.

Introduction

The Command-Line Based Task Management System (TMS) is aimed at achieving, and has realized the following expectations:

- The creation, modification, record and display of user-defined tasks;
- The definement, management and utilization of user-defined criteria for task filtering;
- The friendly interaction through a command-line user interface.

Introduction

The Command-Line Based Task Management System (TMS) is aimed at achieving, and has realized the following expectations:

- The creation, modification, record and display of user-defined tasks;
- The definement, management and utilization of user-defined criteria for task filtering;
- The friendly interaction through a command-line user interface.

System Architecture

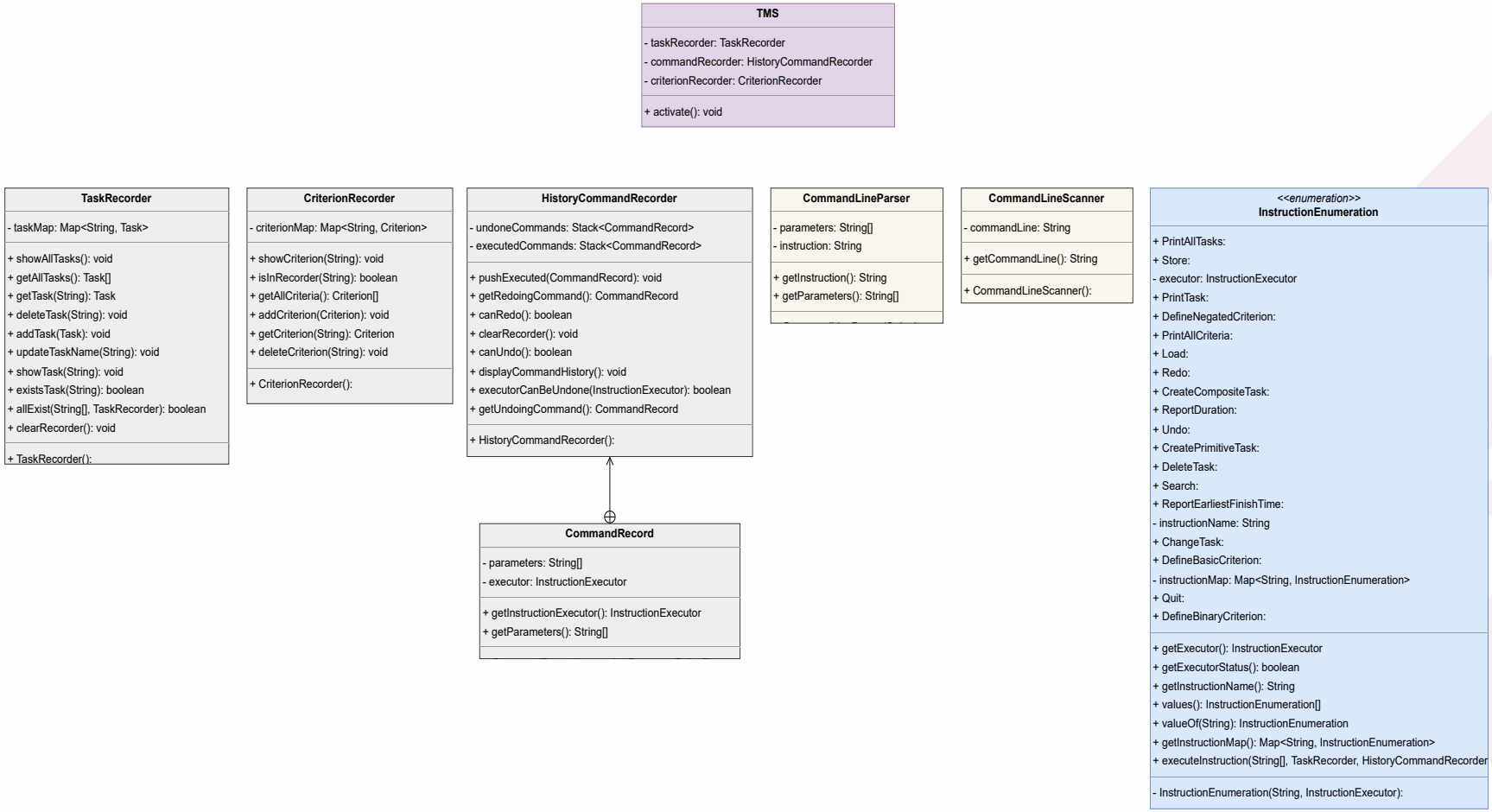

Figure 1: System architecture

System Constructs

In our design, the operation of TMS and the implementation of the functions rely on the necessary system constructs.

- A command-line scanner and a parser
  - *To acquire and analyze user input and divide the command-line into an instruction and parameters.*
- Three recorders: task recorder, criterion recorder and history command recorder
  - *To record and index all created objects and undoable input commands.*
  - *To provide methods of searching and deleting created objects.*
- The recorders have the same lifetime as the system itself, which the scanner and parser are repeatedly constructed for each user input.

System Constructs

In our design, the operation of TMS and the implementation of the functions rely on the necessary system constructs.

- A command-line scanner and a parser
  - *To acquire and analyze user input and divide the command-line into an instruction and parameters.*
- Three recorders: task recorder, criterion recorder and history command recorder
  - *To record and index all created objects and undoable input commands.*
  - *To provide methods of searching and deleting created objects.*
- The recorders have the same lifetime as the system itself, which the scanner and parser are repeatedly constructed for each user input.

System Constructs

In our design, the operation of TMS and the implementation of the functions rely on the necessary system constructs.

- A command-line scanner and a parser
  - *To acquire and analyze user input and divide the command-line into an instruction and parameters.*
- Three recorders: task recorder, criterion recorder and history command recorder
  - *To record and index all created objects and undoable input commands.*
  - *To provide methods of searching and deleting created objects.*
- The recorders have the same lifetime as the system itself, which the scanner and parser are repeatedly constructed for each user input.

System Constructs

In our design, the operation of TMS and the implementation of the functions rely on the necessary system constructs.

- A command-line scanner and a parser
  - *To acquire and analyze user input and divide the command-line into an instruction and parameters.*
- Three recorders: task recorder, criterion recorder and history command recorder
  - *To record and index all created objects and undoable input commands.*
  - *To provide methods of searching and deleting created objects.*
- The recorders have the same lifetime as the system itself, which the scanner and parser are repeatedly constructed for each user input.

System Constructs

In our design, the operation of TMS and the implementation of the functions rely on the necessary system constructs.

- A command-line scanner and a parser
  - *To acquire and analyze user input and divide the command-line into an instruction and parameters.*
- Three recorders: task recorder, criterion recorder and history command recorder
  - *To record and index all created objects and undoable input commands.*
  - *To provide methods of searching and deleting created objects.*
- The recorders have the same lifetime as the system itself, which the scanner and parser are repeatedly constructed for each user input.

System Constructs

In our design, the operation of TMS and the implementation of the functions rely on the necessary system constructs.

- A command-line scanner and a parser
  - *To acquire and analyze user input and divide the command-line into an instruction and parameters.*
- Three recorders: task recorder, criterion recorder and history command recorder
  - *To record and index all created objects and undoable input commands.*
  - *To provide methods of searching and deleting created objects.*
- The recorders have the same lifetime as the system itself, which the scanner and parser are repeatedly constructed for each user input.

Task and Criterion

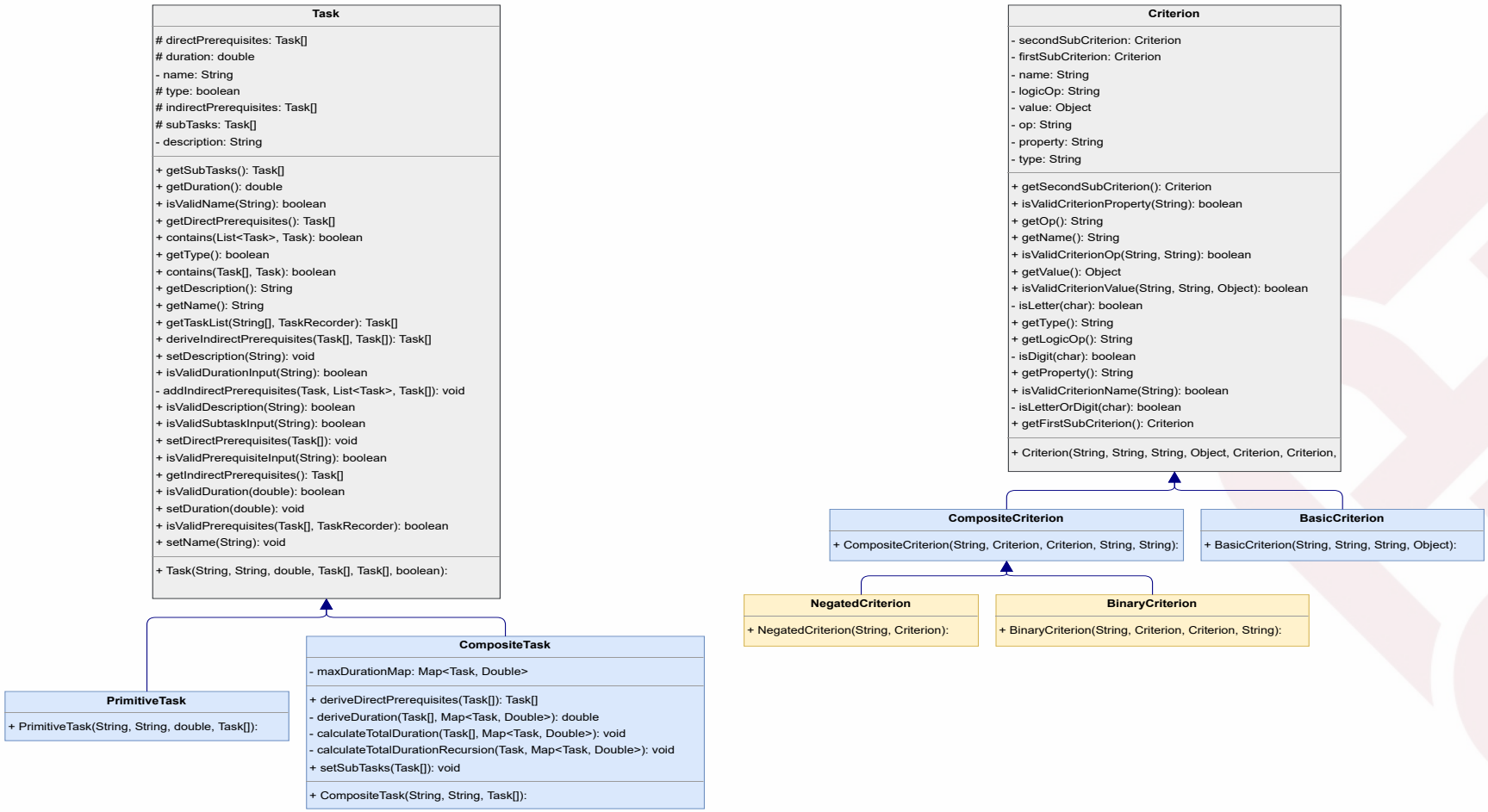

Figure 2: Task and Criterion

Task and Criterion

The tasks and criteria are fundamental objects that will be created by the user.

- The system defines the classes of Task and Criterion for the instantiation and operation on the user-defined objects.
- Inheritance is used for the variation of the meaning of the objects.
  - *PrimitiveTask and CompositeTask inherit from Task;*
  - *BasicCriterion and CompositeCriterion inherit from Criterion;*
  - *NegatedCriterion and BinaryCriterion inherit from CompositeCriterion;*
  - *Varied fields and methods are provided for each of them.*

Task and Criterion

The tasks and criteria are fundamental objects that will be created by the user.

- The system defines the classes of Task and Criterion for the instantiation and operation on the user-defined objects.
- Inheritance is used for the variation of the meaning of the objects.
  - *PrimitiveTask and CompositeTask inherit from Task;*
  - *BasicCriterion and CompositeCriterion inherit from Criterion;*
  - *NegatedCriterion and BinaryCriterion inherit from CompositeCriterion;*
  - *Varied fields and methods are provided for each of them.*

Task and Criterion

The tasks and criteria are fundamental objects that will be created by the user.

- The system defines the classes of Task and Criterion for the instantiation and operation on the user-defined objects.
- Inheritance is used for the variation of the meaning of the objects.
  - *PrimitiveTask and CompositeTask inherit from Task;*
  - *BasicCriterion and CompositeCriterion inherit from Criterion;*
  - *NegatedCriterion and BinaryCriterion inherit from CompositeCriterion;*
  - *Varied fields and methods are provided for each of them.*

Task and Criterion

The tasks and criteria are fundamental objects that will be created by the user.

- The system defines the classes of Task and Criterion for the instantiation and operation on the user-defined objects.
- Inheritance is used for the variation of the meaning of the objects.
  - *PrimitiveTask and CompositeTask inherit from Task;*
  - *BasicCriterion and CompositeCriterion inherit from Criterion;*
  - *NegatedCriterion and BinaryCriterion inherit from CompositeCriterion;*
  - *Varied fields and methods are provided for each of them.*

Task and Criterion

The tasks and criteria are fundamental objects that will be created by the user.

- The system defines the classes of Task and Criterion for the instantiation and operation on the user-defined objects.
- Inheritance is used for the variation of the meaning of the objects.
  - *PrimitiveTask and CompositeTask inherit from Task;*
  - *BasicCriterion and CompositeCriterion inherit from Criterion;*
  - *NegatedCriterion and BinaryCriterion inherit from CompositeCriterion;*
  - *Varied fields and methods are provided for each of them.*

Task and Criterion

The tasks and criteria are fundamental objects that will be created by the user.

- The system defines the classes of Task and Criterion for the instantiation and operation on the user-defined objects.
- Inheritance is used for the variation of the meaning of the objects.
  - *PrimitiveTask and CompositeTask inherit from Task;*
  - *BasicCriterion and CompositeCriterion inherit from Criterion;*
  - *NegatedCriterion and BinaryCriterion inherit from CompositeCriterion;*
  - *Varied fields and methods are provided for each of them.*

Instruction and Executor

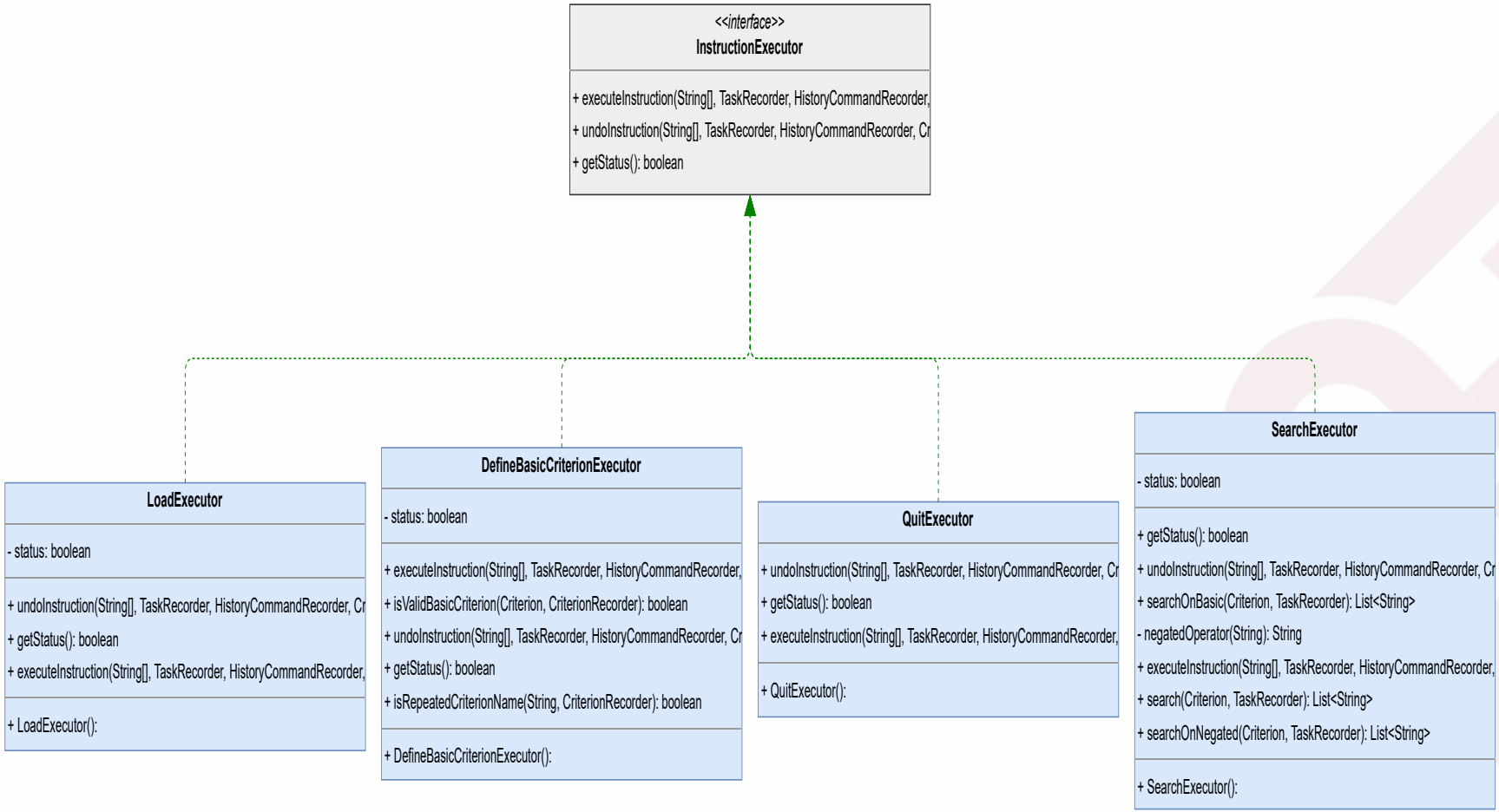

Figure 3: Instruction and Executor

Instruction and Executor

The first element of the user input is always the name of an instruction, and the others are parameters. They are conducted by a corresponding executor.

- Each executor is an implementation of the interface InstructionExecutor. The interface specifies the methods that each executor is expected to implement.
  - *executeInstruction, undoInstruction and getStatus.*
- The instructions are collected in an Enum class with the field of instructionName and the method of construction of the executor instance.
  - *With the parsed instructionName and parameters given, the system is able to select the correct executor to instantiate.*

Instruction and Executor

The first element of the user input is always the name of an instruction, and the others are parameters. They are conducted by a corresponding executor.

- Each executor is an implementation of the interface InstructionExecutor. The interface specifies the methods that each executor is expected to implement.
  - *executeInstruction, undoInstruction and getStatus.*
- The instructions are collected in an Enum class with the field of instructionName and the method of construction of the executor instance.
  - *With the parsed instructionName and parameters given, the system is able to select the correct executor to instantiate.*

Instruction and Executor

The first element of the user input is always the name of an instruction, and the others are parameters. They are conducted by a corresponding executor.

- Each executor is an implementation of the interface InstructionExecutor. The interface specifies the methods that each executor is expected to implement.
  - *executeInstruction, undoInstruction and getStatus.*
- The instructions are collected in an Enum class with the field of instructionName and the method of construction of the executor instance.
  - *With the parsed instructionName and parameters given, the system is able to select the correct executor to instantiate.*

Instruction and Executor

The first element of the user input is always the name of an instruction, and the others are parameters. They are conducted by a corresponding executor.

- Each executor is an implementation of the interface InstructionExecutor. The interface specifies the methods that each executor is expected to implement.
  - *executeInstruction, undoInstruction and getStatus.*
- The instructions are collected in an Enum class with the field of instructionName and the method of construction of the executor instance.
  - *With the parsed instructionName and parameters given, the system is able to select the correct executor to instantiate.*

Instruction and Executor

The executors of the instructions are defined in a inner package.

- Each executor realizes a requirement from the project description.
  - *E.g., CreatePrimitiveTaskExecutor, UndoExecutor, etc.*
- One command-line input equals to one instantiation of a executor.

Instruction and Executor

The executors of the instructions are defined in a inner package.

- Each executor realizes a requirement from the project description.
  - *E.g., CreatePrimitiveTaskExecutor, UndoExecutor, etc.*
- One command-line input equals to one instantiation of a executor.

Instruction and Executor

The executors of the instructions are defined in a inner package.

- Each executor realizes a requirement from the project description.
  - *E.g., CreatePrimitiveTaskExecutor, UndoExecutor, etc.*
- One command-line input equals to one instantiation of a executor.

Object-orientation Programming Discussion

The adaptation of object-orientation benefits our programming.

- Clear design and definement of the objects and their properties and behaviors.
- Easy modification and expansion of the code.

Object-orientation Programming Discussion

The adaptation of object-orientation benefits our programming.

- Clear design and definement of the objects and their properties and behaviors.
- Easy modification and expansion of the code.

Demonstration

Let's start the demonstration!