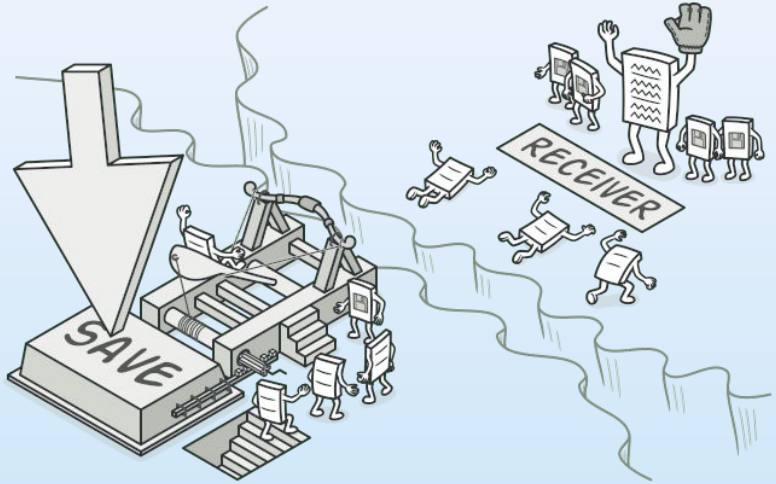
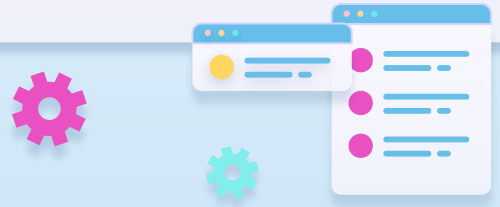
Patrón Command



Ángel Chaves Chinchila C12113
Camilo Suárez Sandí C17811

Tabla de contenidos

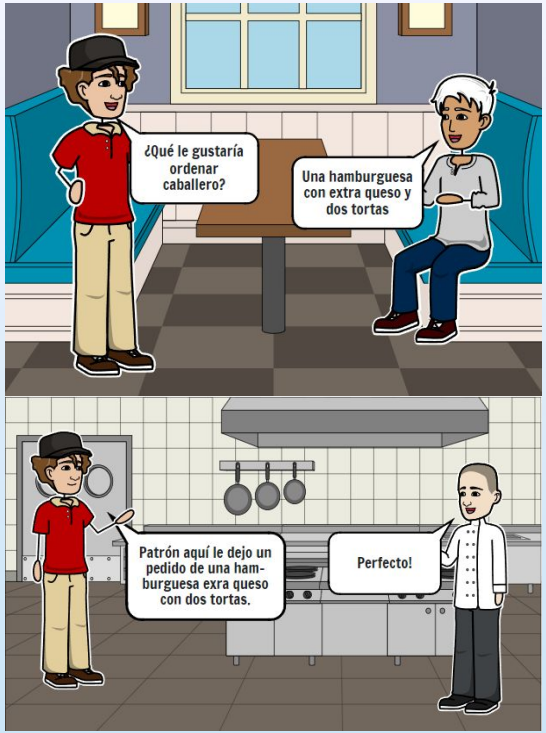


Descripción

El patrón Command encapsula una solicitud como un objeto, lo que le permite parametrizar otros objetos con diferentes solicitudes, poner en cola o registrar solicitudes y admitir operaciones que se pueden deshacer.

-The Gang of Four

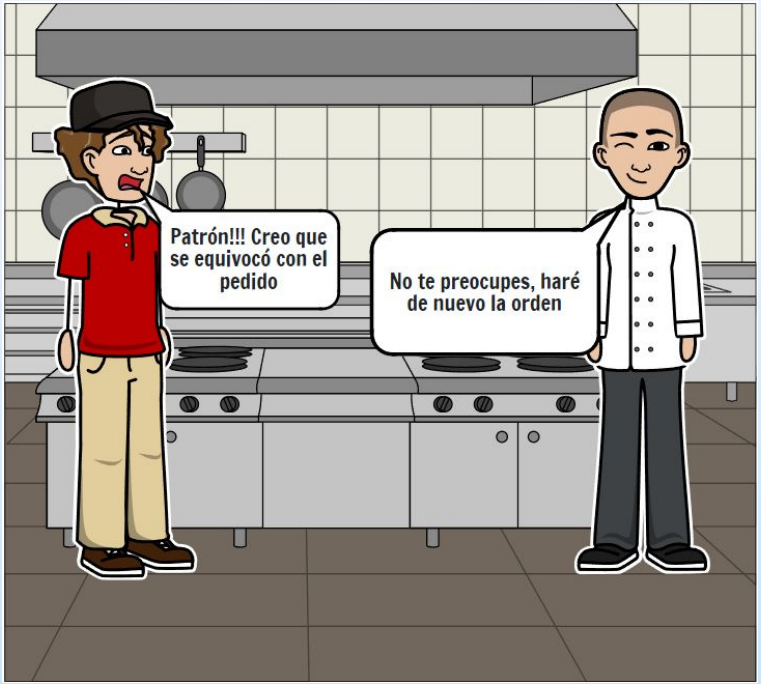

Me gustaría que me cocinaras una hamburguesa con extra queso y dos tortas
Hey!!! Fuera de mi cocina! >:(


¿Qué le gustaría ordenar caballero?
Una hamburguesa con extra queso y dos tortas
Patrón aquí le dejo un pedido de una hamburguesa exra queso con dos tortas.
Perfecto!

Uso de colas


¿Qué le gustaría ordenar caballero?
Una hamburguesa con extra queso y dos tortas
Yo quiero...
Nosotros queremos....
Qué le gustaría ordenar?
Patrón le traigo más ordenes
Uy no, son muchas. Dejeme las ordenes ahí en la mesa, ahorita las atiendo

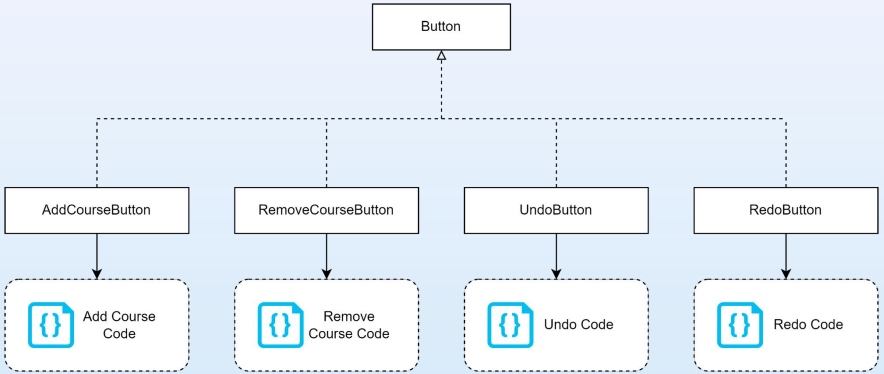
Admitir operaciones que se pueden deshacer (*undo*).



Problema

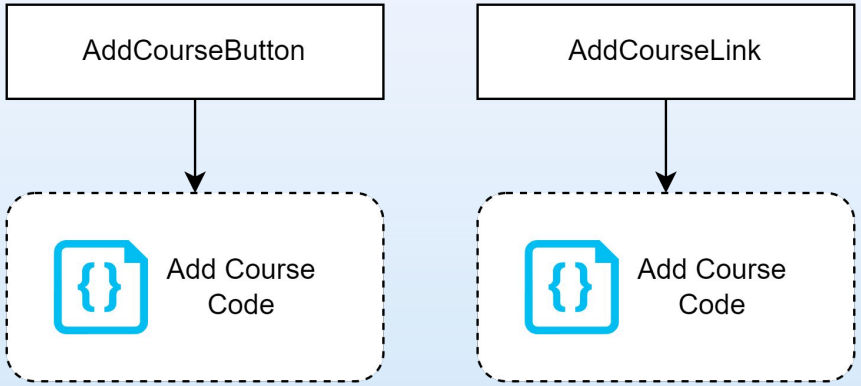
Problema

- Realizar solicitudes a objetos sin conocimiento previo de la operaciones o el receptor.
- Sistema de matrícula universitaria con operaciones de agregar y borrar.
- Añadir operaciones de deshacer, rehacer y permitir futuras operaciones sin alterar las existentes.
- Botones correspondientes a cada operación.



Problema

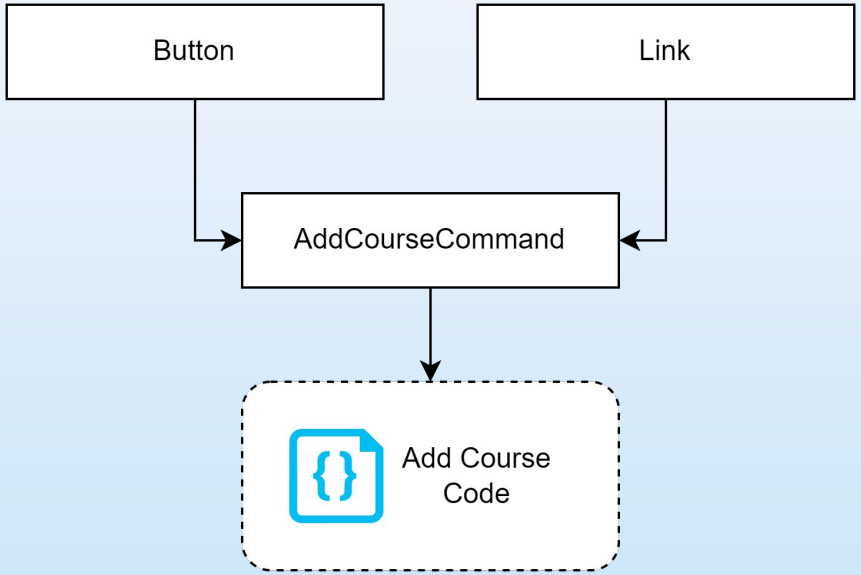
- Creación de múltiples clases y necesidad de cambios en cada modificación de la clase base.
- Dependencia del código de la interfaz con la lógica de negocio.
- Problemas de redundancia de código.
- Complicaciones adicionales debido a la existencia de otras clases con las mismas operaciones.



Solución

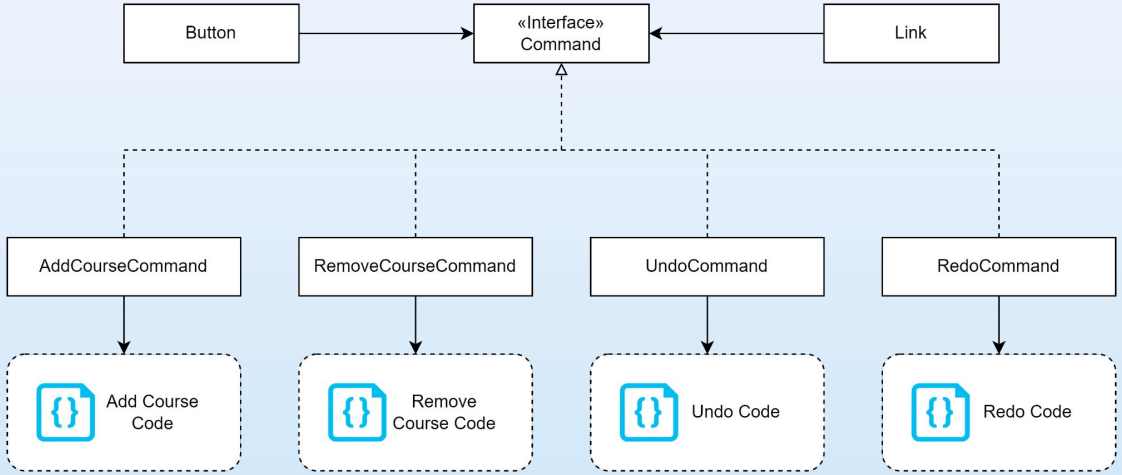
Solución

- Encapsulación de solicitudes como objetos.
- Ejecución de diferentes solicitudes con parámetros variables.
- Capacidad de encolar o registrar solicitudes.
- Operaciones de deshacer y rehacer.

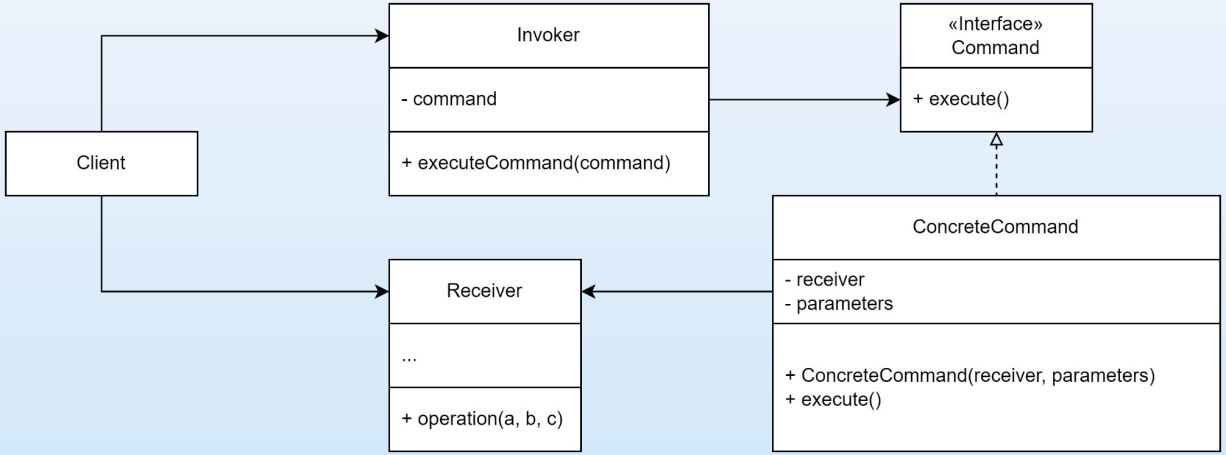


Solución

- Implementación de la misma interfaz.
- No es necesario crear múltiples subclases.
- Otras clases pueden implementar la misma operación utilizando los comandos adecuados.
- Evita la redundancia de código y simplifica la implementación.

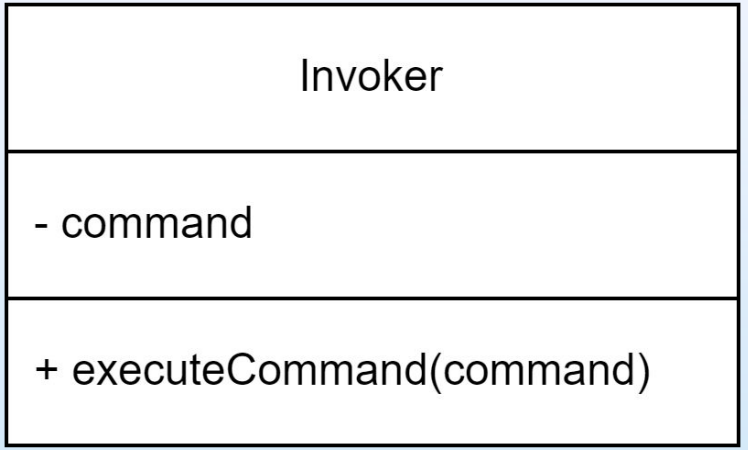


Estructura


Invoker
- command
+ executeCommand(command)
«Interface»
Command
+ execute()
Client
ConcreteCommand
- receiver
- parameters
+ ConcreteCommand(receiver, parameters)
+ execute()
Receiver
...
+ operation(a, b, c)

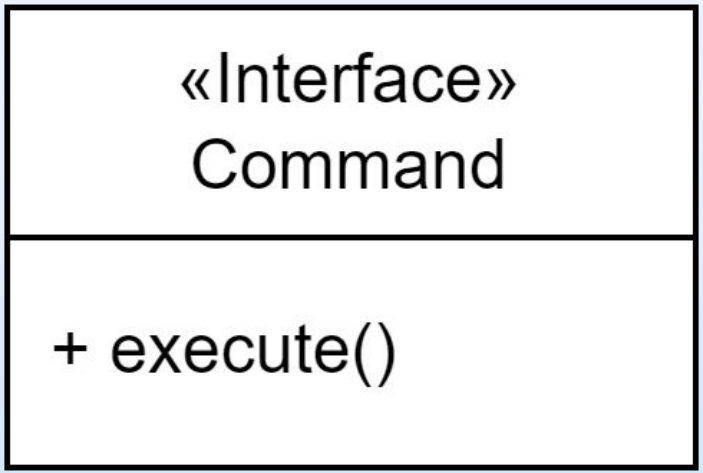
Clase Invoker

- Utiliza el objeto Command para realizar solicitudes.
- Enviar las solicitudes ejecutando el comando correspondiente.
- Guarda una referencia al objeto Command.
- No crea el objeto Command, se crea en el cliente.



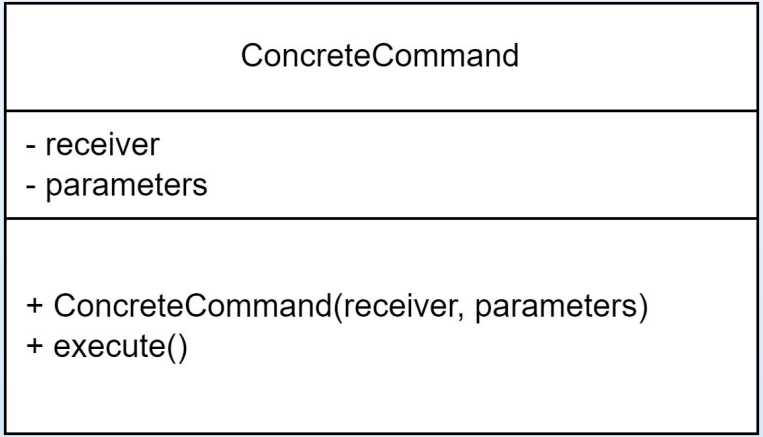
Interfaz Command

- Declara la interfaz para ejecutar una operación.
- Permite ejecutar el comando.



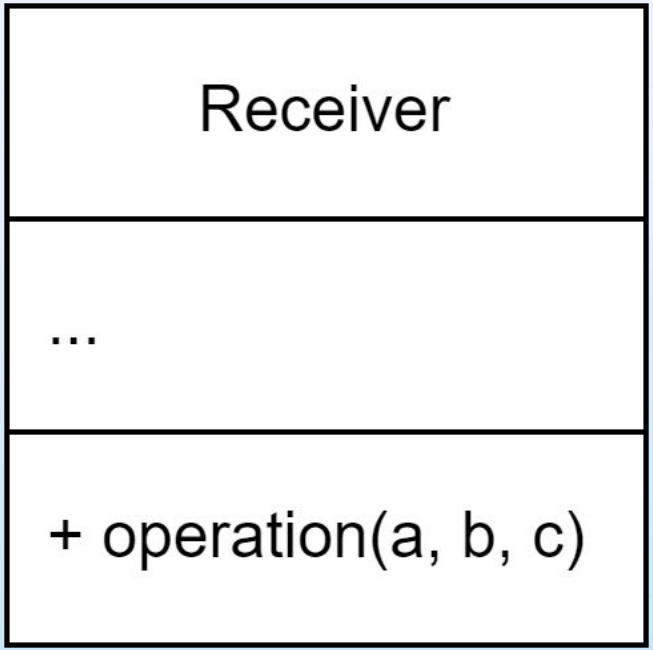
Clases Concretas Command

- Establecen el enlace entre el objeto Receiver y una acción específica.
- Método para ejecutar la operación en el Receiver.
- Varias solicitudes son implementadas según sea necesario.
- Los parámetros pueden ser declarados como atributos en la clase Command.



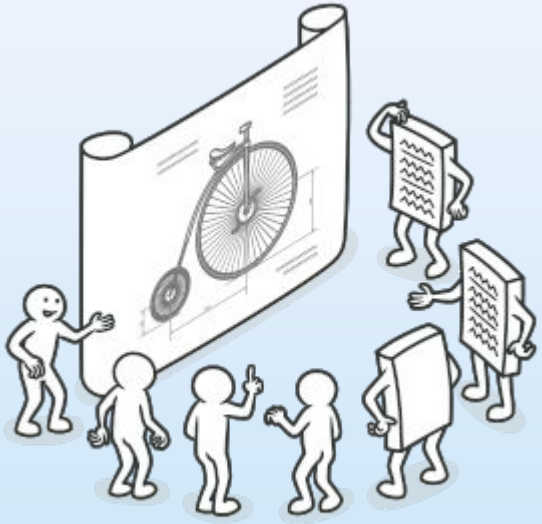
Clase Receiver

- Conoce cómo realizar las operaciones asociadas a una solicitud.
- Puede ser cualquier clase.
- Contiene parte de la lógica de negocio.
- Los comandos configuran la forma en que se envía la solicitud al Receiver.



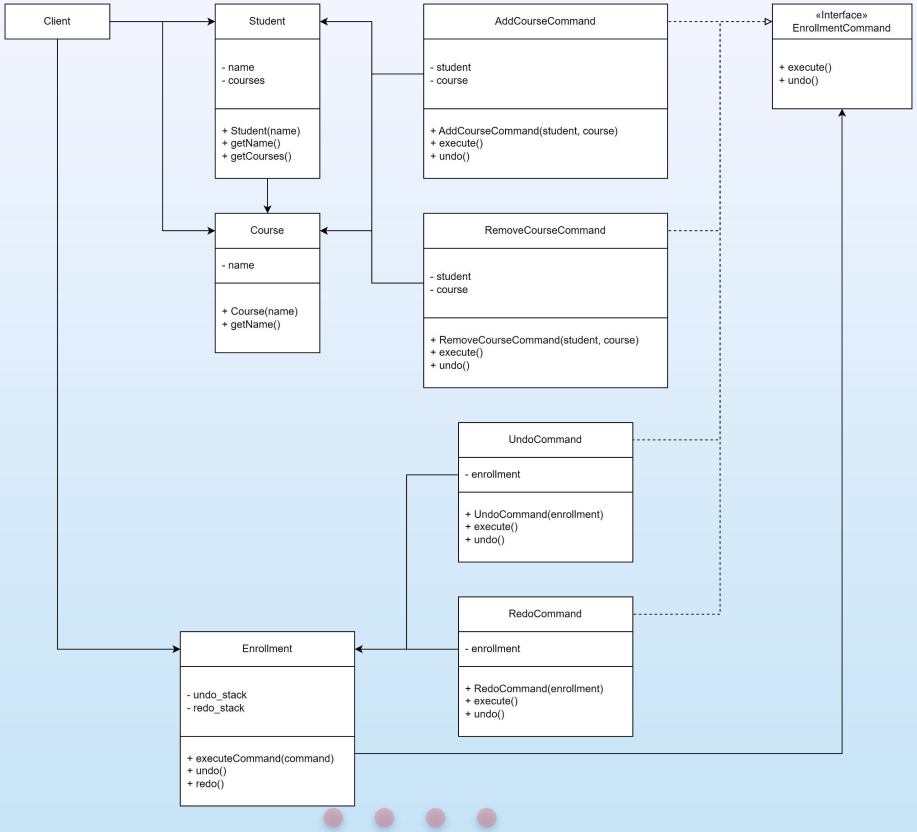
Client

- Guarda un objeto concreto Command y proporciona el objeto Receiver correspondiente.
- Guarda el objeto Command.
- El objeto Invoker realiza la solicitud ejecutando el comando.
- El objeto Command ejecuta las operaciones necesarias en el Receiver.
- Crea y configura objetos Command, proporcionando parámetros necesarios.
- Posible asociación con múltiples objetos Invoker.

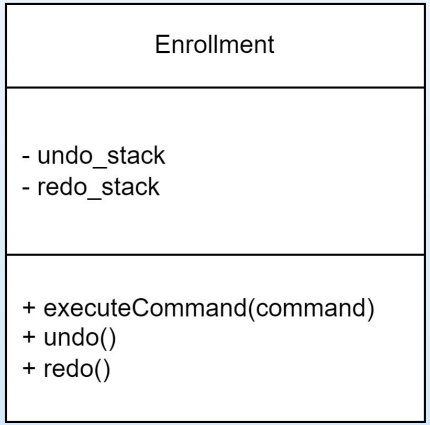


Ejemplo en código

Estructura

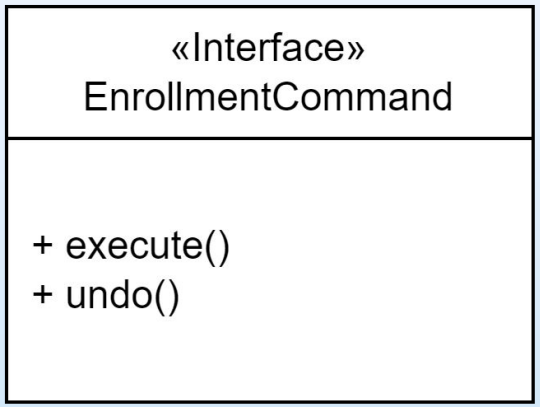

Client
Student
- name
- courses
+ Student(name)
+ getName()
+ getCourses()
Course
- name
+ Course(name)
+ getName()
AddCourseCommand
- student
- course
+ AddCourseCommand(student, course)
+ execute()
+ undo()
RemoveCourseCommand
- student
- course
+ RemoveCourseCommand(student, course)
+ execute()
+ undo()
«Interface»
EnrollmentCommand
+ execute()
+ undo()
UndoCommand
- enrollment
+ UndoCommand(enrollment)
+ execute()
+ undo()
RedoCommand
- enrollment
+ RedoCommand(enrollment)
+ execute()
+ undo()
Enrollment
- undo_stack
- redo_stack
+ executeCommand(command)
+ undo()
+ redo()

Clase Invoker



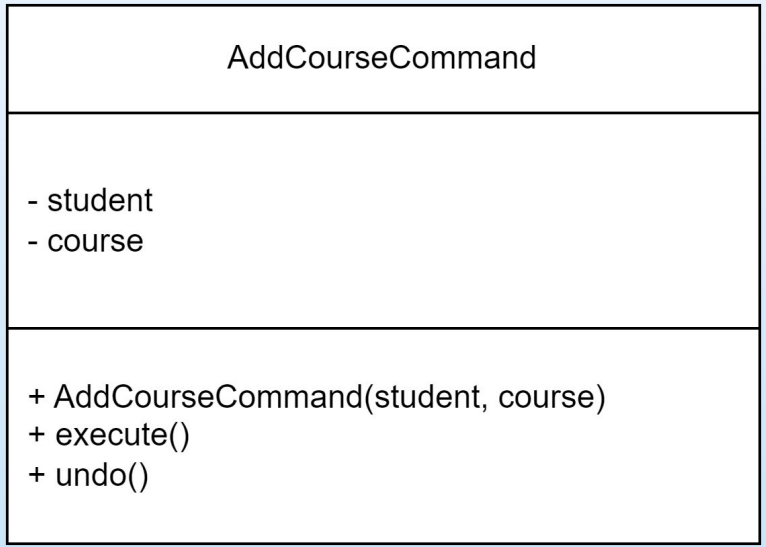

```
class Enrollment {
 private:
  std::stack<EnrollmentCommand*> undo_stack;
  std::stack<EnrollmentCommand*> redo_stack;

 public:
  void executeCommand(EnrollmentCommand* command) {
    command->execute();
    this->undo_stack.push(command);
  }

  void undo() {
    if (!this->undo_stack.empty()) {
      auto* command = this->undo_stack.top();
      command->undo();
      this->undo_stack.pop();
      this->redo_stack.push(command);
    }
  }

  void redo() {
    if (!this->redo_stack.empty()) {
      auto* command = this->redo_stack.top();
      command->execute();
      this->redo_stack.pop();
      this->undo_stack.push(command);
    }
  }
};
```

Interfaz Command



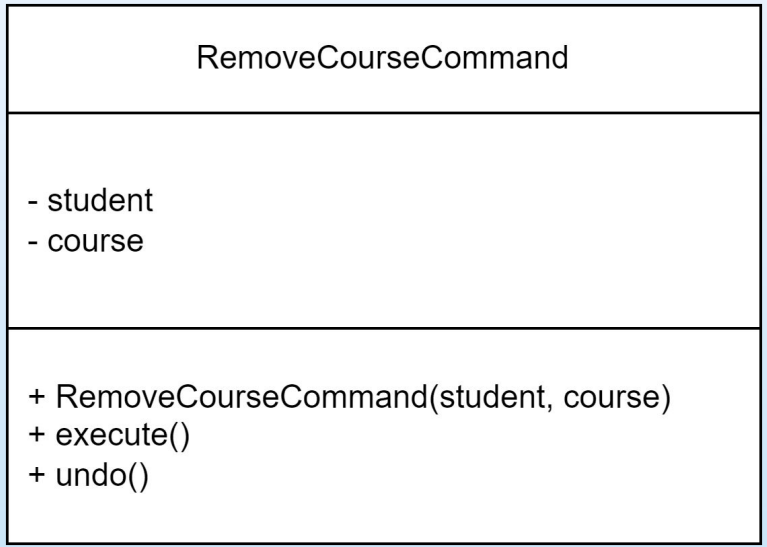

```
1  class EnrollmentCommand {
2   public:
3    virtual void execute() = 0;
4
5    virtual void undo() = 0;
6  };
```

Clases Concretas Command



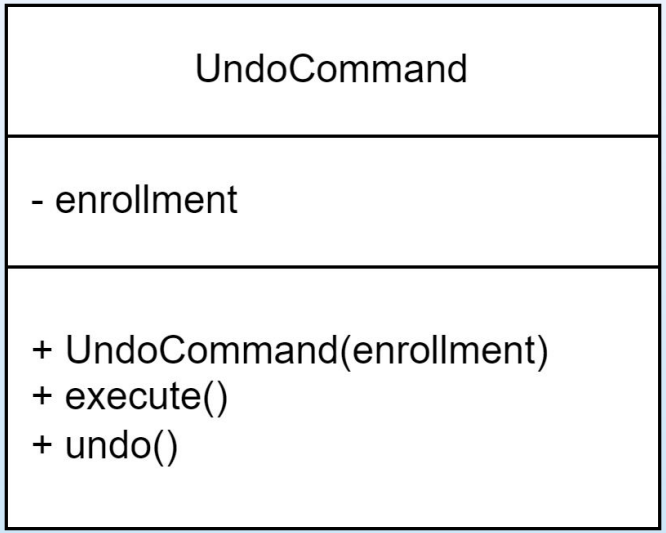

```cpp
class AddCourseCommand : public EnrollmentCommand {
 private:
  Student& student;
  Course& course;

 public:
  AddCourseCommand(Student& student, Course& course)
      : student(student), course(course) {
  }

  void execute() override {
    this->student.getCourses().push_back(this->course);
  }

  void undo() override {
    auto& courses = this->student.getCourses();

    for (auto iterator = courses.begin(); iterator != courses.end();
         ++iterator) {
      auto course = *iterator;

      if (this->course.getName() == course.getName()) {
        this->student.getCourses().erase(iterator);
        break;
      }
    }
  }
};
```

Clases Concretas Command



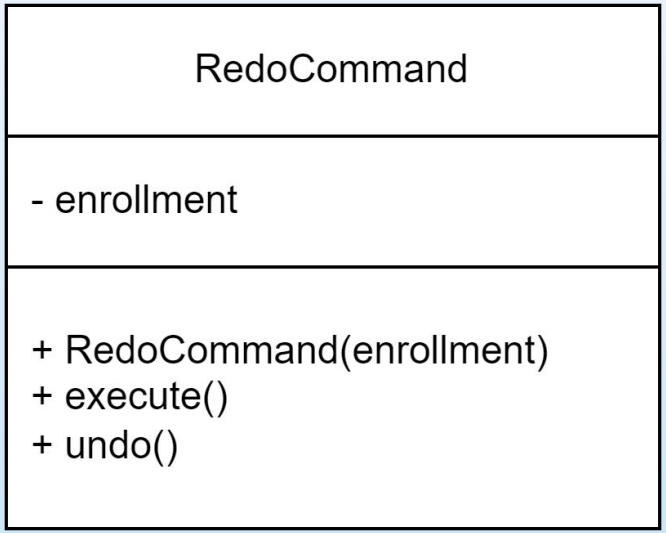

```cpp
class RemoveCourseCommand : public EnrollmentCommand {
 private:
  Student& student;
  Course& course;

 public:
  RemoveCourseCommand(Student& student, Course& course)
      : student(student), course(course) {
  }

  void execute() override {
    auto& courses = this->student.getCourses();

    for (auto iterator = courses.begin(); iterator != courses.end();
         ++iterator) {
      auto course = *iterator;

      if (this->course.getName() == course.getName()) {
        this->student.getCourses().erase(iterator);
        break;
      }
    }
  }

  void undo() override {
    this->student.getCourses().push_back(this->course);
  }
};
```

Clases Concretas Command



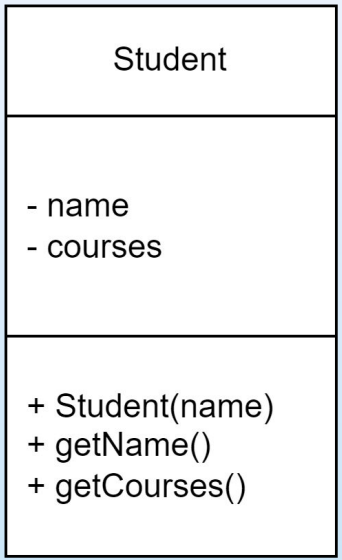

```
1  class UndoCommand : public EnrollmentCommand {
2   private:
3    Enrollment& enrollment;
4
5   public:
6    UndoCommand(Enrollment& enrollment) : enrollment(enrollment) {
7    }
8
9    void execute() override {
10     this->enrollment.undo();
11   }
12
13   void undo() override {
14   }
15 };
```

Clases Concretas Command



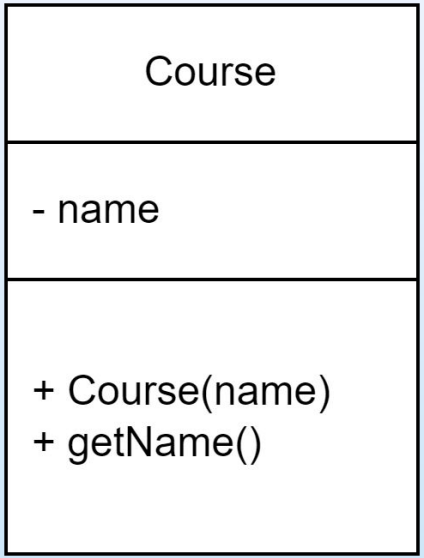

```
class RedoCommand : public EnrollmentCommand {
 private:
  Enrollment& enrollment;

 public:
  RedoCommand(Enrollment& enrollment) : enrollment(enrollment) {
  }

  void execute() override {
    this->enrollment.redo();
  }

  void undo() override {
  }
};
```

Clase Receiver




```cpp
class Student {
 private:
  std::string name;
  std::vector<Course> courses;

 public:
  Student(const std::string& name) : name(name) {
  }

  const std::string& getName() const {
    return this->name;
  }

  std::vector<Course>& getCourses() {
    return this->courses;
  }
};
```

Clase Receiver




```
1  class Course {
2   private:
3    std::string name;
4
5   public:
6    Course(const std::string& name) : name(name) {
7    }
8
9    const std::string& getName() const {
10     return this->name;
11   }
12 };
```

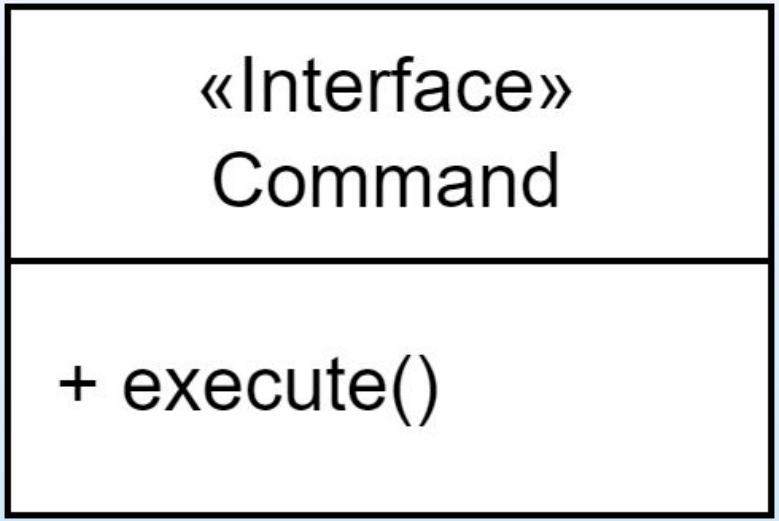
Client

```
1  int main() {
2    Student student("Martin Fowler");
3
4    Course english_course("English");
5    Course spanish_course("Spanish");
6
7    Enrollment enrollment;
8
9    enrollment.executeCommand(new AddCourseCommand(student, english_course));
10   enrollment.executeCommand(new AddCourseCommand(student, spanish_course));
11
12   enrollment.executeCommand(new UndoCommand(enrollment));
13
14   enrollment.executeCommand(new RedoCommand(enrollment));
15 }
16
```

Implementación

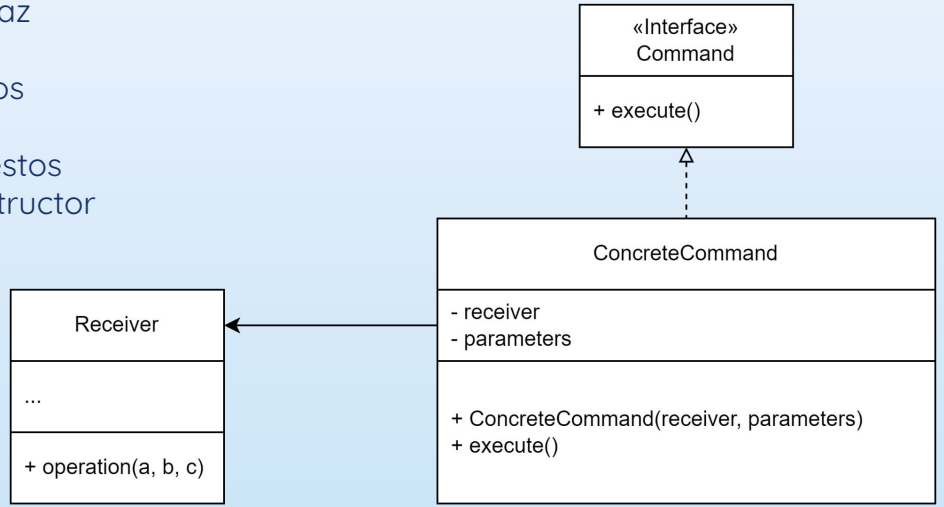
Pasos

- Declarar la interfaz Command con un método de ejecución único.



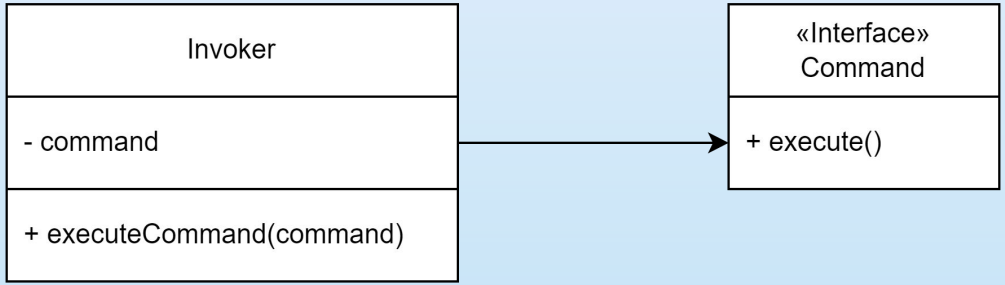
Pasos

- Extraer las solicitudes en clases concretas Command que implementan la interfaz Command. Cada clase debe tener un conjunto de atributos para guardar los argumentos de las solicitudes y una referencia al objeto Receiver. Todos estos valores deben inicializarse en el constructor de la clase concreta Command.



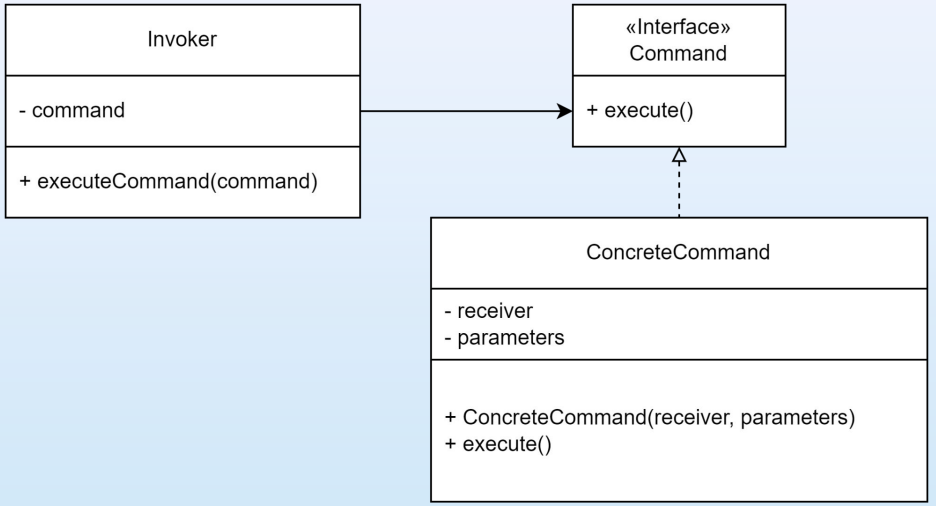
Pasos

- Identificar las clases que actúan como Invokers. Agregar atributos para guardar los comandos en estas clases. Los Invokers deben comunicarse con los comandos a través de la interfaz Command. Por lo general, los Invokers no crean los objetos Command por sí mismos, sino que los obtienen del código cliente.



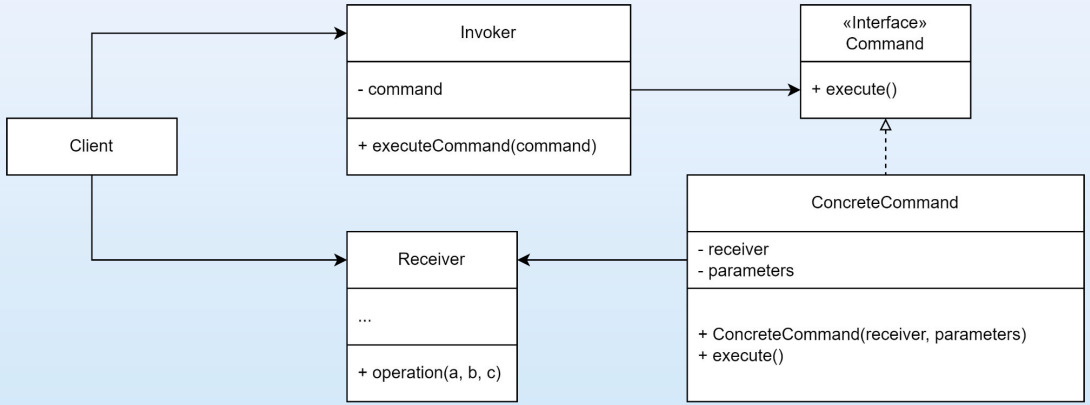
Pasos

- Modificar los Invokers para que ejecuten el comando en lugar de enviar directamente una solicitud al objeto Receiver.



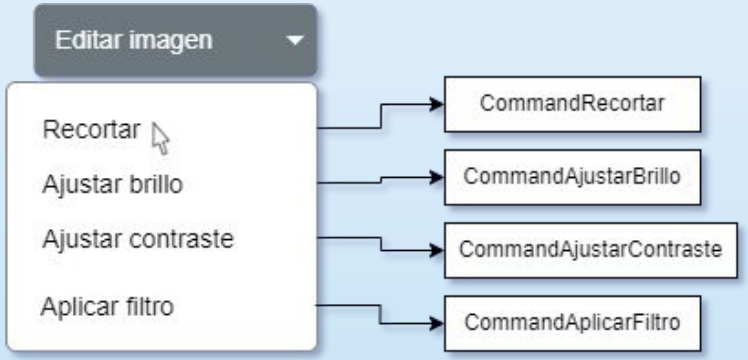
Pasos

- El cliente debe inicializar los objetos en el siguiente orden:
  - Crear los objetos Receivers.
  - Crear los comandos y asociarlos, si es necesario, con los objetos Receivers.
  - Crear los Invokers y asociarlos con los comandos específicos.



Aplicabilidad

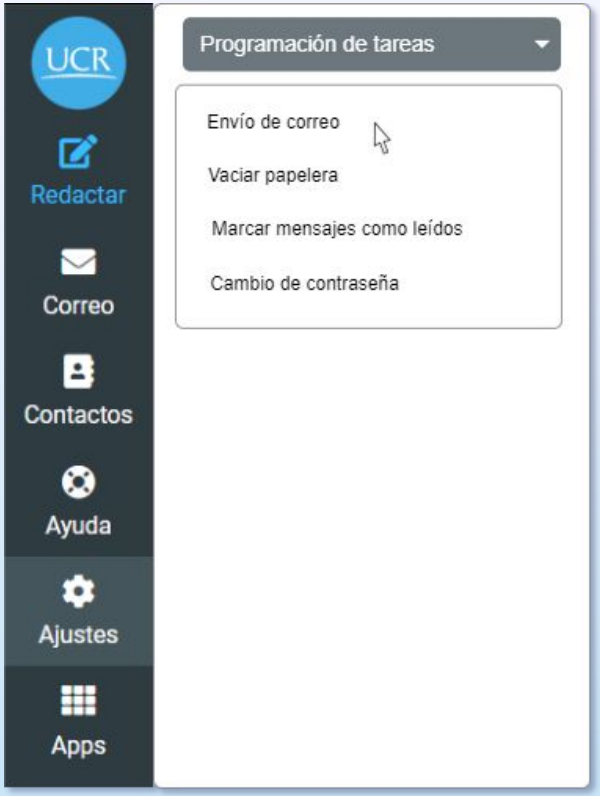
Cuando se desee parametrizar objetos con operaciones: El patrón Command puede convertir una llamada de método específico en un objeto independiente. Este cambio abre muchos usos interesantes: puede pasar comandos como argumentos de método, almacenarlos dentro de otros objetos, cambiar comandos vinculados en tiempo de ejecución, etc. [Ejemplo editor imagen y luces]



Aplicabilidad

Utilice el patrón Command cuando desee poner en cola operaciones, programar su ejecución o ejecutarlas de forma remota. [Ejemplo correo]

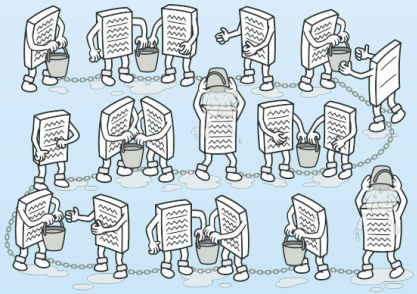
Utilice el patrón Command cuando desee implementar operaciones reversibles.



Relación con otros patrones

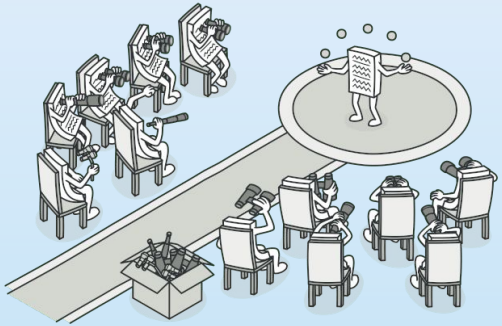
Cadena de responsabilidad

Pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la maneja.



Observer

Permite a los receptores suscribirse y darse de baja dinámicamente de las solicitudes de recepción.

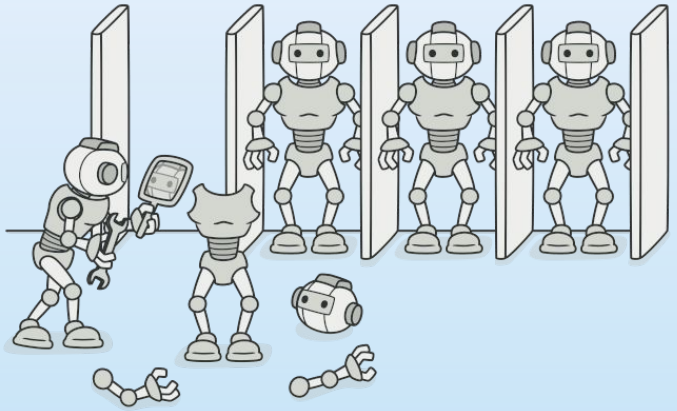


Memento

Se puede usar con Command para implementar *undo*. Los Commands realizan varias operaciones sobre un objeto destino y los Memento guardan el estado de ese objeto.

Prototype

Puede ayudar cuando necesite guardar copias de Comandos en el historial.



Consecuencias

Ventajas

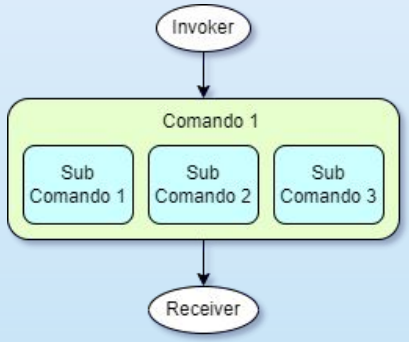
Facilita la implementación de operaciones **deshacer/rehacer**

Promueve la extensibilidad y la flexibilidad del código

Puede implementar la ejecución diferida de operaciones (**colas** y registros de comandos)

Desventajas

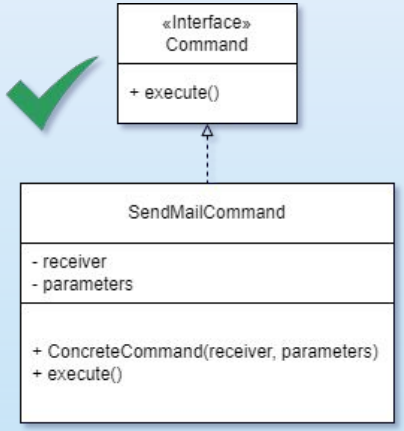
El código puede volverse más complicado ya que está introduciendo una capa completamente nueva entre remitentes y receptores.

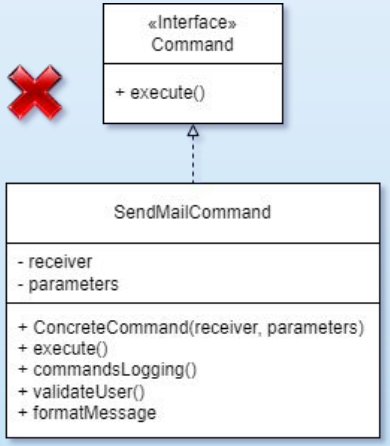


Principios de diseño

SOLID

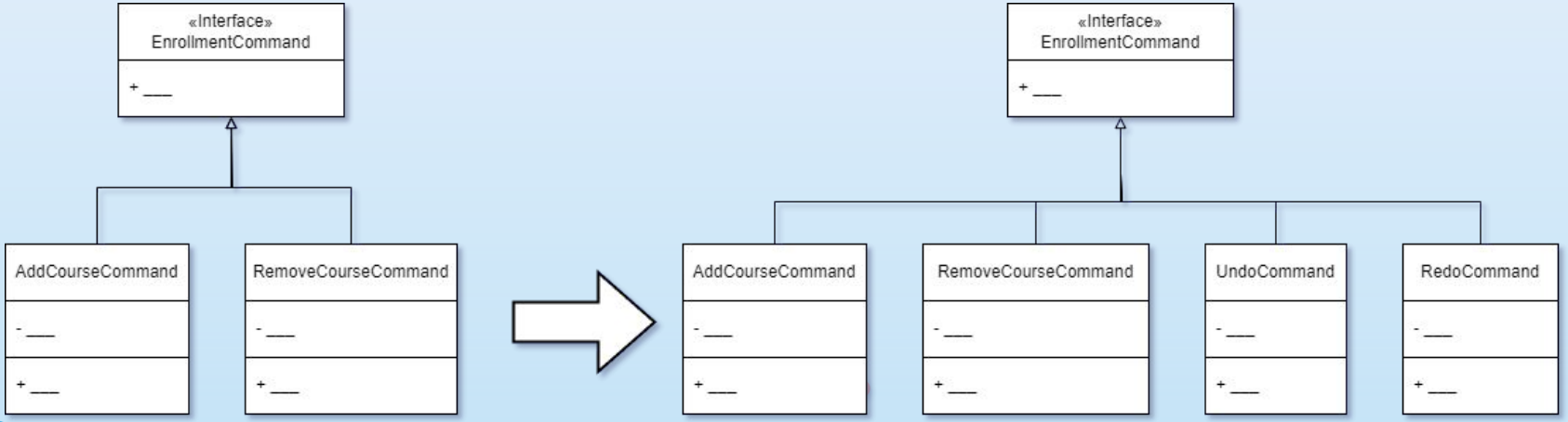
Principio de responsabilidad única (SRP): Cada comando tiene la responsabilidad de encapsular una operación específica y ejecutarla cuando sea necesario. **Podría incumplirse** si un comando también se encarga de realizar operaciones adicionales, como el registro de eventos, la notificación de otros objetos o la gestión del estado.





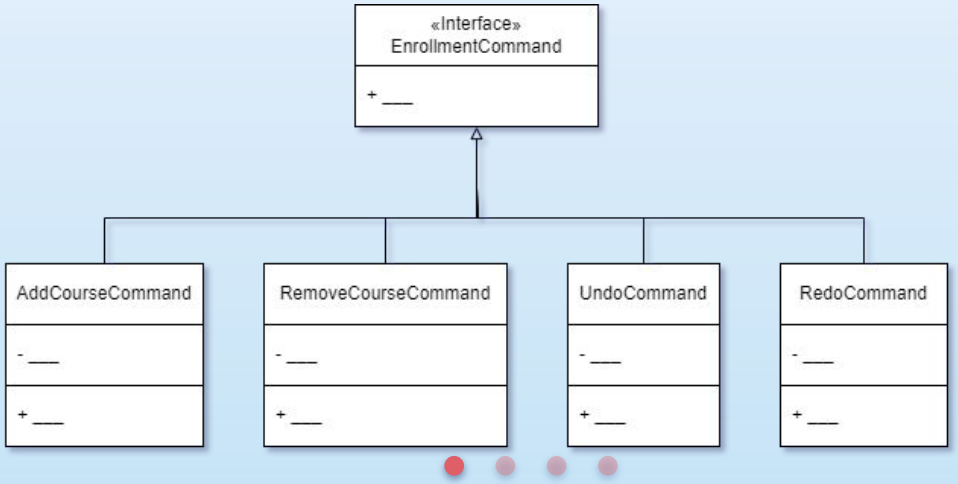
SOLID

Principio de Open/Closed (OCP): cumple con este principio al permitir la extensibilidad del código sin modificar el código existente. Se pueden agregar nuevos comandos sin alterar los objetos invocadores o los objetos receptores existentes. **No se estaría cumpliendo** si se debe estar modificando el objeto invocador o también la interfaz ICommand cada vez que se agreguen nuevos comandos.



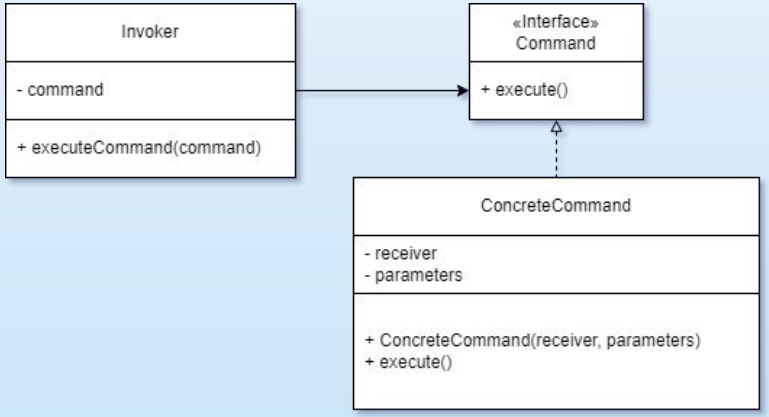
SOLID

Principio de segregación de interfaces: Cumple con este principio al definir una interfaz específica para los comandos. Cada comando implementa sólo los métodos necesarios para ejecutar la acción solicitada, evitando así la dependencia de métodos innecesarios. **Podría incumplirse** si se agregan métodos innecesarios a la interfaz.



SOLID

Principio de inversión de dependencia: cumple con este principio al permitir que los objetos invocadores no dependan directamente de los **objetos receptores** o de los **comandos concretos**. Los objetos invocadores solo conocen la interfaz común del comando.



KISS

DRY

OOP

Bibliografía

Refactoring Guru. (s.f.). Command. Recuperado de https://refactoring.guru/design-patterns/command

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

DigitalOcean. (s.f.). Command Design Pattern. Recuperado de https://www.digitalocean.com/community/tutorials/command-design-pattern

Spring Framework Guru. (s.f.). Command Pattern. Recuperado de https://springframework.guru/gang-of-four-design-patterns/command-pattern/

Banas, D. (2012, 10 de octubre). Design Patterns: Command Pattern [Video]. YouTube. Recuperado de https://www.youtube.com/watch?v=9qA5kw8dcSU

¡Muchas gracias!