Bugsquashing:

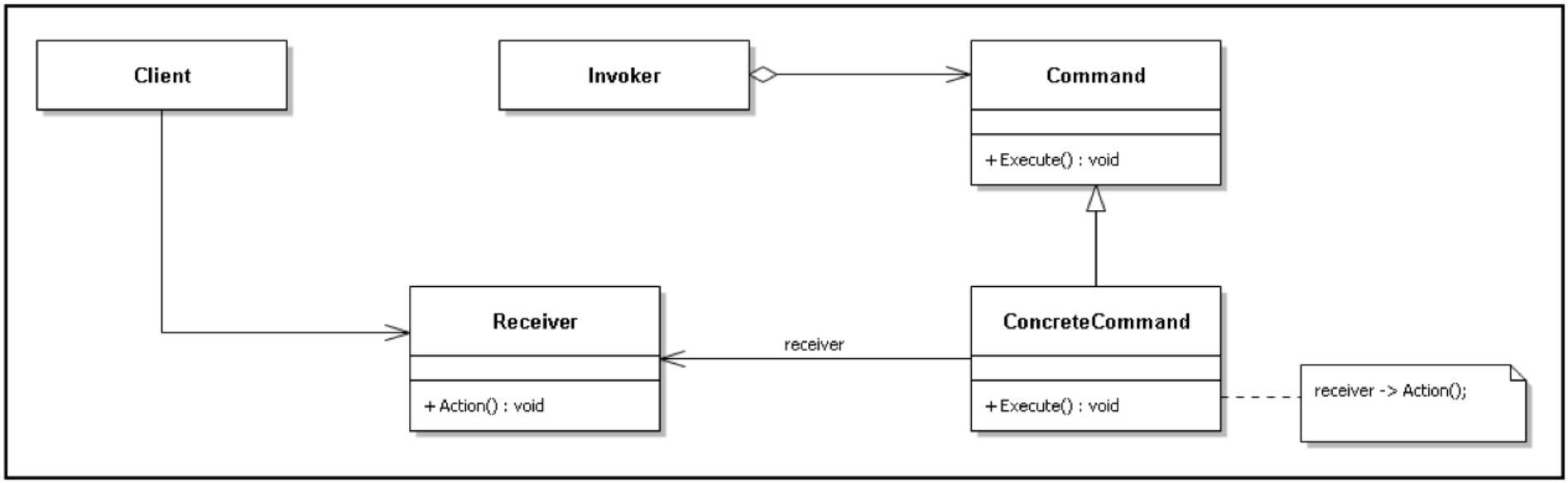
Command - Pattern

Andreas Fetzer

30.03.2011

dkfz.
DEUTSCHES
KREBSFORSCHUNGSZENTRUM
IN DER HELMHOLTZ-GEMEINSCHAFT

Class structure



- **Command**: Abstract superclass of all commands.
- **Concrete Command**: Specifies a concrete comand. Has the execute method in which the corresponding action on the receiver is performed. Must know the receiver and all neccessary information for the action.
- **Invoker**: Has the neccessary informations to invoke a concrete command on the receiver.
- **Receiver**: The concrete command is executed on the receiver.

Example

```
int main() {

    MBIEmployee employee;

    //Some work happens....
    SetUpDartclientCommand cmd;
    employee.SetNextCommand(&cmd);
    employee.DoSomething();
    //Some work happens....

    CoffeeMaschine maschine;
    double ml = 250;
    MakeCoffeeCommand drinkCoffeeCmd(maschine, ml);
    employee.SetNextCommand(& drinkCoffeeCmd);
    employee.DoSomething();
    return 0;
}
```

Example

```
int main() {
    class MBIEmployee {
public:
    MBIEmployee();
    virtual ~MBIEmployee();
    void SetNextCommand(Command* cmd);
    void DoSomething()
    {
        //logging, create undo info
        //can be any command
        command->execute();
    }

private:
    Command* command;
};
}
```

„INVOKER“

Example

```
int main() {
    cla
public:
    MBI
    vir
    voi
    voi
    {

    }

private
    Com
};
}
```

```
class MakeCoffeeCommand : public Command{

public:
    MakeCoffeeCommand(CoffeeMaschine, double);
    virtual ~MakeCoffeeCommand();

    void Execute()
    {
        coffeeMaschine.MakeCoffee(ml);
    }

private:
    CoffeeMaschine coffeeMaschine;
    double ml;
};
```

„CONCRETE COMMAND“

Example

```
int main() {
    cla
public:
    MBI
    vir
    voic
    voi
    {

    }

private:
    Comm
};
}
```

```
class MakeCoffeeCommand : public Command{
public:
    MakeCo
    virtua

    void E
    {
        cof
    }
private:
    Coffee
    double };
};
```

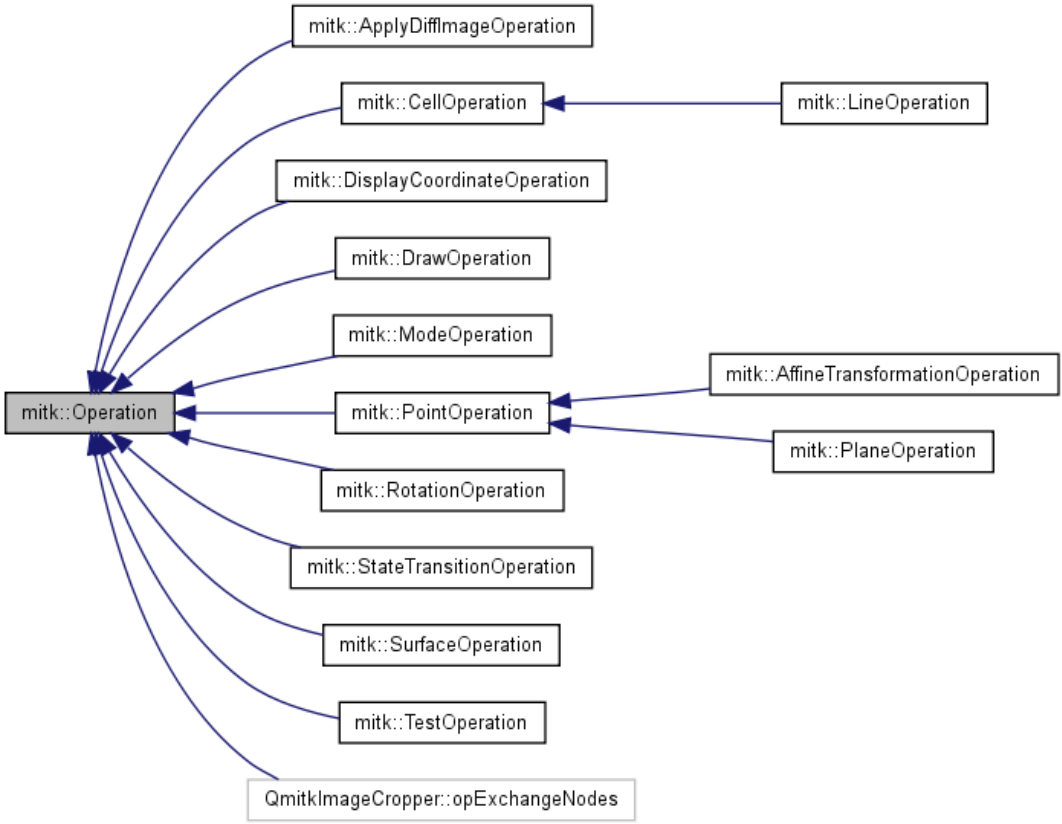
```
class CoffeeMaschine {
public:
    CoffeeMaschine();
    virtual ~CoffeeMaschine();

    void MakeCoffee(double ml)
    {
        std::cout<<"Bitte Schalen leeren!!";
    }
```

„RECEIVER“

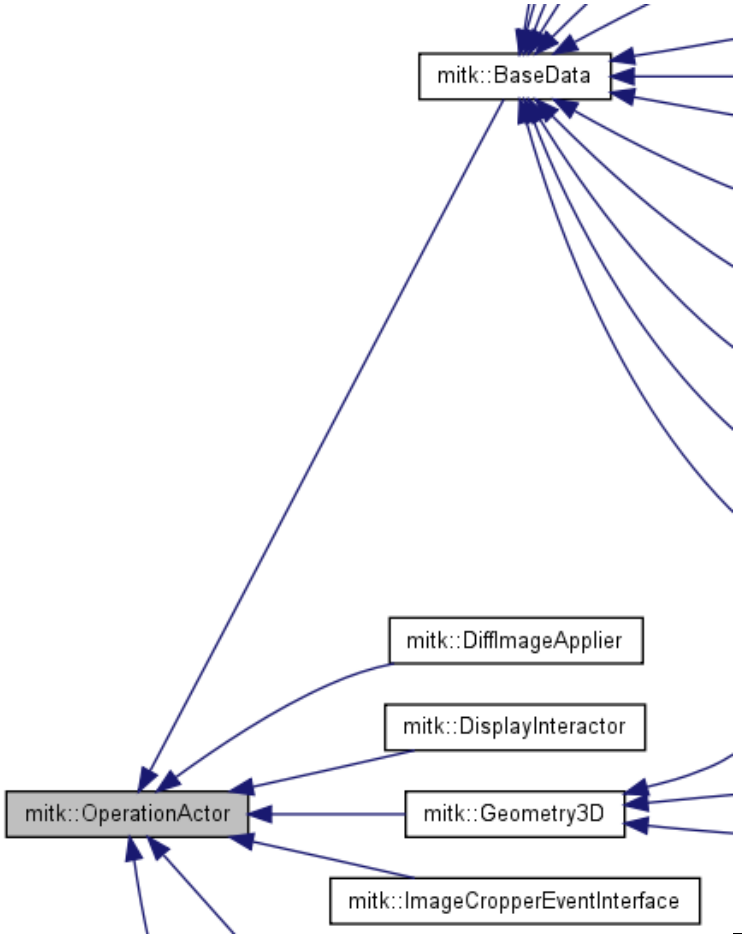
Commands in MITK

- Little bit different to the previous described pattern:
  - Command = mitk::Operation
  - has no execute method
  - just stores the neccessary informations
  - e.g. mitk::RotationOperation stores angle, vector and point of rotation
  - Enum for OperationType in mitkInteractionConst.h



Commands in MITK

- mitk::Operations are executed by subclasses of mitk::OperationActor
- OperationActor defines *ExecuteOperation* method
- e.g. SliceNavigationController creates a *RotationOperation*.
  - Geometry3D is derived from from OperationActor and has *ExecuteOperation* method.
- Inverse command strategy for undo/redo



Summary

- Behavioural pattern
- One object is used to encapsulate all the informations needed to call a certain method at later time:

  i.e.:
  - The owner of the specific method
  - neccessary parameter values for the method
- Uses:
  - Undo/Redo
  - Transactional behaviour (e.g. *rollback*)
  - GUI buttons and menu items
  - ...

Sources

- http://en.wikipedia.org/wiki/Command_pattern

- Design Patterns. Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides