Pattern: Command

Presented by: Rick Bradshaw
**Behavioral Patterns**

- Concerned with algorithms and the assignment of responsibility between objects. They describe not only the objects or classes but also the pattern of communication between them.
- Characterize complex control flow that is difficult to follow at run-time.

**Command Pattern: Intent**

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue, or log requests, and support undoable operations.
Command Pattern: Motivation/Applications

Motivation:
- Used when it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Applications:
- Object oriented replacement for “Call-back” functions
- specify, queue, and execute requests at different times
- Support “undo”
- Log changes to be replayed upon system crash
- Implement “transactional” systems
Command Pattern: Structure

- Command: declares an interface for executing an operation
- ConcreteClass:
  - Defines a binding between a Receiver and an Action()
  - Implements Execute by invoking the Action() from Receiver
- Client: creates a ConcreteCommand and sets the Receiver
- Invoker: asks the command to carry out the request
- Receiver: knows how to perform an Action()
  - Any class can act as a Receiver
Command Pattern: Consequences

- Decouples invoker from the object that performs the operation
- Can assemble multiple Commands into composite commands, like Macros/Transactions
- Easily change Commands without changing existing classes.
- If you are going to support “undo” you will need to possibly store extra state information in the ConcreteCommand object to ensure no loss or alteration of behavior
class Command{
    public:
        virtual void execute(void) =0;
        virtual ~Command(void){};
};

class Task : public Command {
    public:
        Task(string day, string task ){
            _task = task;
            _day = day;
        }
        void execute(void){
            cout << _day << "\t" << _task << endl;
        }
    private:
        string _task;
        string _day;
};

class TaskList{
    public:
        void add(Command *c) {
            commands.push_back(c);
        }
        void printTasks(void){
            for(vector<Command*>::size_type x=0;x<commands.size();x++){
                commands[x]->execute();
            }
        }
        void undo(void){
            if(commands.size() > 0) {
                commands.pop_back();
            } else {
                cout << "Can't undo" << endl;
            }
        }
    private:
        vector<Command*> commands;
};
int main(void) {
    TaskList todos;

    //Create each task
    Task first("Monday", "OO class");
    Task second("Tuesday", "Car appointment");
    Task third("Wednesday", "VHD meeting");
    Task fourth("Friday", "leave early");

    //Add tasks to TaskList
    cout << endl << "TODO List:" << endl;
    todos.add(&first);
    todos.add(&second);
    todos.add(&third);
    todos.printTasks();

    //Show an undo operation
    todos.undo();
    cout << endl << "TODO List:" << endl;
    todos.add(&fourth);
    todos.printTasks();
    return 0;
}