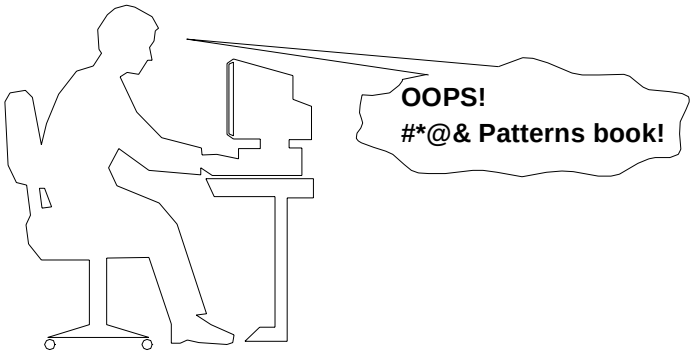
Command Processor

The *Command Processor* design pattern separates the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

Example A text editor usually provides a way to deal with mistakes made by the user. A simple example is undoing the most recent change. A more attractive solution is to enable the undoing of multiple changes. We want to develop such an editor. For the purpose of this discussion let us call it TEDDI.



The design of TEDDI includes a multi-level undo mechanism and allows for future enhancements, such as the addition of new features or a batch mode of operation.

The user interface of TEDDI offers several means of interaction, such as keyboard input or pop-up menus. The program has to define one or several *callback* procedures that are automatically called for every human-computer interaction.

Context Applications that need flexible and extensible user interfaces, or applications that provide services related to the execution of user functions, such as scheduling or undo.

Problem An application that includes a large set of features benefits from a well-structured solution for mapping its interface to its internal functionality. This allows you to support different modes of user interaction, such as pop-up menus for novices, keyboard shortcuts for more experienced users, or external control of the application via a scripting language.

You often need to implement services that go beyond the core functionality of the system for the execution of user requests. Examples are undo, redo, macros for grouping requests, logging of activity, or request scheduling and suspension.

The following *forces* shape the solution:

- Different users like to work with an application in different ways.
- Enhancements of the application should not break existing code.
- Additional services such as undo should be implemented consistently for all requests.

Solution The *Command Processor* pattern builds on the Command design pattern in [GHJV95]. Both patterns follow the idea of encapsulating requests into objects. Whenever a user calls a specific function of the application, the request is turned into a *command* object. The Command Processor pattern illustrates more specifically how command objects are managed. The See Also section discusses further differences between the Command pattern and the Command Processor pattern.

A central component of our pattern description, the *command processor*, takes care of all command objects. The command processor schedules the execution of commands, may store them for later undo, and may provide other services such as logging the sequence of commands for testing purposes. Each command object delegates the execution of its task to *supplier* components within the functional core of the application.

Structure The *abstract command* component defines the interface of all command objects. As a minimum this interface consists of a procedure to execute a command. The additional services implemented by the command processor require further interface procedures for all command objects. The abstract command class of TEDDI, for example, defines an additional undo method.

For each user function we derive a *command component* from the abstract command. A command component implements the interface of the abstract command by using zero or more *supplier components*. The commands of TEDDI save the state of associated supplier components prior to execution, and restore it in case of undo. For example, the delete command is responsible for storing the text deleted and its position in the document.

Class Abstract Command	***Collaborators*** -
Responsibility • Defines a uniform interface to execute commands. • Extends the interface for services of the command processor, such as undo and logging.	

Class Command	***Collaborators*** • Supplier
Responsibility • Encapsulates a function request. • Implements interface of abstract command. • Uses suppliers to perform a request.	

The *controller* represents the interface of the application. It accepts requests, such as 'paste text,' and creates the corresponding command objects. The command objects are then delivered to the command processor for execution. The controller of TEDDI maintains the event loop and maps incoming events to command objects.

The *command processor* manages command objects, schedules them and starts their execution. It is the key component that implements additional services related to the execution of commands. The command processor remains independent of specific commands because it only uses the abstract command interface. In the case of our TEDDI word processor, the command processor also stores already-performed commands for later undo.

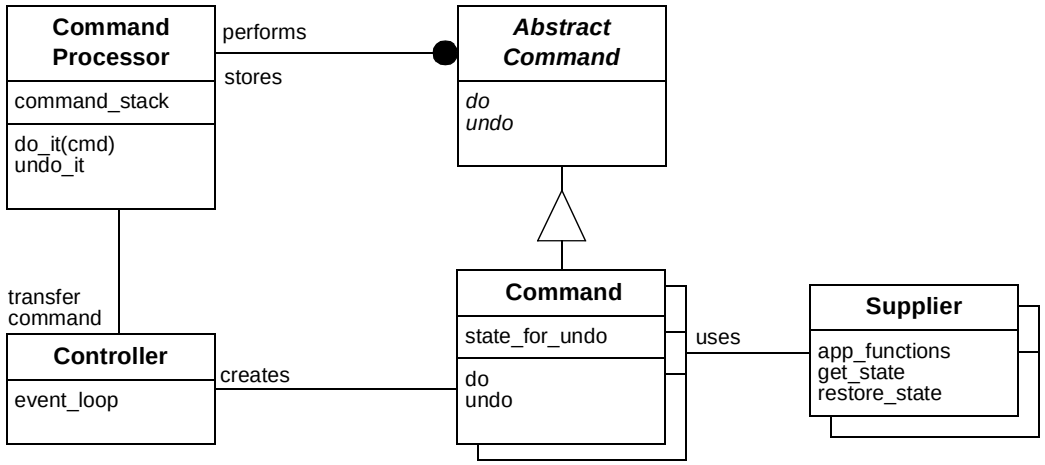
The *supplier* components provide most of the functionality required to execute concrete commands (that is, those related to the concrete command class, as opposed to the abstract command class). Related commands often share supplier components. When an undo mechanism is required, a supplier usually provides a means to save and restore its internal state. The component implementing the internal text representation is the main supplier in TEDDI.

Class Controller	*Collaborators*
Responsibility • Accepts service requests. • Translates requests into commands. • Transfers commands to command processor.	• Command Processor • Command

Class Command Processor	*Collaborators*
Responsibility • Activates command execution. • Maintains command objects. • Provides additional services related to command execution.	• Abstract Command

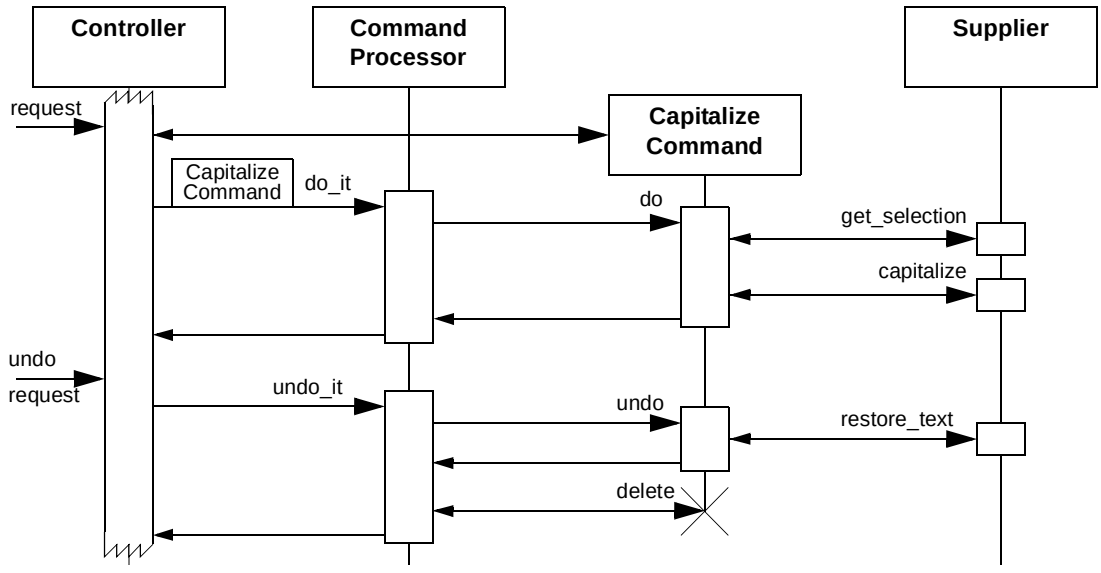
Class Supplier	*Collaborators*
Responsibility • Provides application specific functionality	-

The following diagram shows the principal relationships between the components of the pattern. It demonstrates undo as an example of an additional service provided by the command processor.



Dynamics The following diagram shows a typical scenario of the Command Processor pattern implementing an undo mechanism. A request to capitalize a selected word arrives, is performed and then undone. The following steps occur:

- The controller accepts the request from the user within its event loop and creates a 'capitalize' command object.
- The controller transfers the new command object to the command processor for execution and further handling.
- The command processor activates the execution of the command and stores it for later undo.
- The capitalize command retrieves the currently-selected text from its supplier, stores the text and its position in the document, and asks the supplier to actually capitalize the selection.
- After accepting an undo request, the controller transfers this request to the command processor. The command processor invokes the undo procedure of the most recent command.
- The capitalize command resets the supplier to the previous state, by replacing the saved text in its original position
- If no further activity is required or possible of the command, the command processor deletes the command object.



Implementation To implement this pattern, carry out the following steps:

1 *Define the interface of the abstract command.* The abstract command class hides the details of all specific commands. This class always specifies the abstract method required to execute a command. It also defines the methods necessary to implement the additional services offered by the command processor. An example is a method 'getNameAndParameters' for logging commands.

➥ For the undo mechanism in TEDDI we distinguish three types of commands. They are modeled as an enumeration, because the command type may change dynamically, as shown in step 3:

No change. A command that requires no undo. Cursor movement falls into this category.

Normal. A command that can be undone. Substitution of a word in text is an example of a normal command.

No undo. A command that cannot be undone, and which prevents the undo of previously performed normal commands.

If we want our text to become 'politically correct' and replace all occurrences of 'he' by 'he/she', TEDDI would need to store all corresponding locations in the document to enable later undo. The potentially high storage requirement of global replacements is the main reason why commands belong to the category 'no undo'.

```
class AbstractCommand {
public:
     enum CmdType { no_change, normal, no_undo };
     virtual ~AbstractCommand();
     virtual void doit();
     virtual void undo();
     CmdType getType() const { return type;}
     virtual String getName() const { return "NONAME";}
          // gives name of command for selection
          // in undo/redo menu
protected:
     CmdType type;
     AbstractCommand(CmdType t=no_change): type(t){}
};
```

The method getName() is used to display the most recent command to the user when he selects 'undo'. ❑

2 *Design the command components* for each type of request that the application supports. There are several options for binding a command to its suppliers. The supplier component can be hard-coded within the command, or the controller can provide the supplier to the command constructor as a parameter. An example of the second situation is a multi-document editor in which a command is connected to a specific document object.

➥ The 'delete' command of TEDDI takes the object representing the text as its first parameter. The range of characters to delete is specified by two additional parameters:

```
class DeleteCmd : public AbstractCommand {
public:
     DeleteCmd(TEDDI_Text *t, int start, int end)
          : AbstractCommand(normal) , mytext(t) ,
               from (start) , to (end) {/*...*/}
     virtual ~DeleteCmd();
     virtual void doit();
          // delete characters in mytext
          // between from and to and save them in delstr
     virtual void undo();
          // insert delstr again at position from
     String getName() const { return "DELETE " + delstr;}
protected:
     TEDDI_Text *mytext;// plan for multiple text buffers
     int from,to;          // range of characters to delete
     String delstr;        // save deleted text for undo
};
```

The implementation of the method doit() calls the method deleteText() of the TEDDI_Text supplier object. ❑

A command object may ask the user for further parameters. The TEDDI 'load text file' command, for example, activates a dialog to request the name of the file to be loaded. In this situation the event-handling system must deliver user input to the command, rather than to the controller. Commands that require user interaction during their creation or execution therefore call for additional care. The design of the event-handling system—which is outside the scope of this pattern—must be able to handle such situations.

Undoable commands can use the Memento pattern [GHJV95] to store the state of their supplier for later undo without violating encapsulation.

3 *Increase flexibility by providing macro commands* that combine several successive commands. Apply the Composite pattern [GHJV95] to implement such a macro command component.

➥ In TEDDI we implement a macro command class, to allow user-defined shortcuts to frequently-used command sequences:

```
class MacroCmd : public AbstractCommand {
public:
     MacroCmd(String name, AbstractCommand *first)
         : AbstractCommand( first->getType()),
                                macroname(name){/*...*/}
     virtual ~MacroCmd();
     virtual void doit();
          // do every command in cmdlist
     virtual void undo();
          // undo all commands in cmdlist in reverse order
     virtual void finish(); // delete commands in cmdlist
     void add(AbstractCommand *next) {
          cmdlist.append(next);
          if (next->getType() == no_undo) type = no_undo;
          /*... */}
     String getName() const { return macroname;}
protected:
     String macroname;
     OrderedCollection<AbstractCommand*> cmdlist;
};
```

The command type of a MacroCmd depends on the commands that are added to the macro. An appended command of type no_undo will prevent the undo of the complete macro command. The undo function otherwise iterates through cmdlist in reverse order undoing all normal commands and skipping all commands of type no_change. ❑

4 *Implement the controller component.* Command objects are created by the controller, for example with the help of the 'creational' patterns Abstract Factory and Prototype [GHJV95]. However, since the controller is already decoupled from the supplier components, this additional decoupling of controller and commands is optional. A generic menu controller provides an example of the application of the Prototype pattern. Such a controller contains a command prototype object for each menu entry, and passes a copy of this object to the command processor whenever the user selects the menu entry. If such a menu controller can be dynamically configured with macro command objects, we can easily implement user-defined menu extensions.

➥ In TEDDI user interaction is handled by callback procedures in the controller. A callback creates the corresponding command object and passes it to the command processor. TEDDI uses a global variable theCP that refers to the single command processor component.

```
void TEDDI_controller::deleteButtonPressed(){
     AbstractCommand *delcmd =
          new DeleteWordCommand(
               this->getCursor(),// pass cursor position
               this->getText());    // pass text
     theCP->perform(delcmd);
}
```

On start-up the callback deleteButtonPressed() is registered with the event-handling system. ❑

5 *Implement access to the additional services of the command processor.* A user-accessible additional service is normally implemented by a specific command class. The command processor supplies the functionality for the 'do' method. Directly calling the interface of the command processor is also an option. Other intrinsic services such as logging of commands are performed automatically by the command processor.

➥ The class UndoCommand provides access to the undo mechanism of TEDDI. The implementation of this class cooperates with the internals of the command processor and is thus declared a friend to it. Note that UndoCommand objects must not be stored by the command processor, and fall in the category no_change.

```
class UndoCommand : public AbstractCommand {
public:
     UndoCommand()
          : AbstractCommand(no_change){}
     virtual ~UndoCommand();
     virtual void doit() { theCP->undo_lastcmd(); }
};
```

The method doit() of UndoCommand asks the command processor to undo the last normal command executed. A class RedoCommand provides the inverse functionality. Its method doit() makes the command processor re-execute the undone command. ❑

6 *Implement the command processor component.* The command processor receives command objects from the controller and takes responsibility for them. For each command object, the command processor starts the execution by calling the do method. A command processor implemented in C++, for example, is responsible for deleting command objects that are no longer useful.

Apply the Singleton design pattern [GHJV95] to ensure that only one command processor exists.

➥ For TEDDI we implement a multi-level undo/redo with two stacks, one for performed commands and one for undone commands:

```
class CommandProcessor {
public:
     CommandProcessor();
     virtual ~CommandProcessor();
     virtual void do_cmd(AbstractCommand *cmd){
          // do cmd and push it on donestack
          cmd->doit();
          switch(cmd->getType()){
          case AbstractCommand::normal:
               donestack.push(cmd); break;
          case AbstractCommand::no_undo:
               donestack.make_empty();
               undonestack.make_empty();
               // Fall through:
          case AbstractCommand::no_change:
               // take responsibility for command objects:
               delete cmd;
               break;
          }
     }
     friend class UndoCommand; // special relationship
     friend class RedoCommand; // special relationship
private:
     // this method is only used by UndoCommand
     virtual void undo_lastcmd();
          // pop cmd from donestack,
          // undo it, and push it on undonestack
     // this method is only used by RedoCommand
     virtual void redo_lastundone(){
          AbstractCommand *last = undonestack.pop();
          if (last) this->do_cmd(last);
     }
private:
     Stack<AbstractCommand*> donestack,undonestack;
};
```

❑

Variants

Spread controller functionality. In this variant the role of the controller can be distributed over several components. For example, each user interface element such as a menu button could create a command object when activated. However, the role of the controller is not restricted to components of the graphical user interface.

Combination with Interpreter pattern. In this variant a scripting language provides a programmable interface to an application. The parser component of the script interpreter takes the role of the controller. Apply the Interpreter pattern [GHJV95] and build the abstract syntax tree from command objects. The command processor is the client in the Interpreter pattern. It carries out interpretation by activating the commands.

Known Uses

ET++ [WGM88] provides a framework of command processors that support unlimited, bounded, and single undo and redo. The abstract class `Command` implements a state machine to track the execution state of each command. This state machine is used to check if a command is performed or undone. The controller role is distributed over the event-handler object hierarchy of an ET++ application.

MacApp [App89] uses the Command Processor design pattern to provide undoable operations.

InterViews [LCITV92] includes an action class that is an abstract base class providing the functionality of a command component.

ATM-P [ATM93] implements a simplified version of the Command Processor pattern. It uses a hierarchy of command classes to pass command objects around, sometimes across process boundaries. The receiver of a command object decides how and when to execute it. Each process implements its own command processor.

SICAT [SICAT95] implements the Command Processor pattern to provide a well-defined undo facility in the control program and the graphical SDL editors.

Consequences The Command-Processor pattern provides the following **benefits**:

Flexibility in the way requests are activated. Different user interface elements for requesting a function can generate the same kind of command object. It is thus easy to remap user input to application functionality. This helps to create an application interface that can be adapted to user preferences. An example is a text editor that provides different control modes such as a WordStar or an emacs keyboard.

Flexibility in the number and functionality of requests. The controller and command processor are implemented independently of the functionality of individual commands. Changing the implementation of a command or introducing new command classes does not affect the command processor or other unrelated parts of the application. For example, it is possible to build more complex commands from existing ones. In addition to a macro mechanism, such compound commands can be pre-programmed, and thus extend the application without modifying the functional core.

Programming execution-related services. The central command processor easily allows the addition of services related to command execution. An advanced command processor can log or store commands to a file for later examination or replay. A command processor can queue commands and schedule them at a later time. This is useful if commands should execute at a specified time, if they are handled according to priority, or if they will execute in a separate thread of control. An additional example is a single command processor shared by several concurrent applications that provides a transaction control mechanism with logging and rollback of commands.

Testability at application level. The command processor is an ideal entry point for application testing. If combined with the Interpreter pattern [GHJV95] as in the second variant above, regression tests can be written in the scripting language and applied after changes to the functional core. Furthermore, logging of command objects executed by the command processor allows you to analyze error situations. If the sequence of executed commands is stored persistently, it can be re-applied after error correction, or reused for regression testing.

Concurrency. The Command Processor design pattern allows commands to be executed in separate threads of control. Responsiveness improves, because the controller does not wait for the execution of a command to finish. However, this calls for synchronization when the global variables of the application, for example in a supplier component, are accessed by several commands executing in parallel.

The Command Processor pattern imposes some **liabilities**:

Efficiency loss. As with all patterns that decouple components, the additional indirection costs storage and time. A controller that performs a service request directly does not impose an efficiency penalty. However, extending such a direct controller with new requests, changing the implementation of a service, or implementing an undo mechanism all require more effort.

Potential for an excessive number of command classes. An application with rich functionality may lead to many command classes. You can handle the complexity of this situation in a number of ways:

- By grouping commands around abstractions.
- By unifying very simple command classes by passing the supplier object as a parameter.
- By pre-programmed macro-command objects that rely on the combination of few low-level commands.

Complexity in acquiring command parameters. Some command objects retrieve additional parameters from the user prior to or during their execution. This situation complicates the event-handling mechanism, which needs to deliver events to different destinations, such as the controller and some activated command object.

See also The Command Processor pattern builds on the *Command* design pattern in [GHJV95]. Both patterns depict the idea of encapsulating service requests into command objects. Command Processor contributes more details of the handling of command objects. The controller of the Command Processor pattern takes the role of the client in the Command pattern. The controller decides which command to use and creates a new command object for each user request.

In the Command pattern, however, the client configures an invoker with a command object that can be executed for several user

requests. The command processor receives command objects from the controller and takes the role of the invoker, executing command objects. The controller from the Command Processor pattern takes the role of the client. The suppliers of the Command Processor pattern correspond to receivers, but we do not require exactly one supplier for a command.

Credits Studying the CommandProcessor classes of ET++ [WGM88] initially motivated this pattern description. The Siemens SICAT team [SICAT95] pointed out the problems with event handling that occur when a command acquires additional parameters from the user during execution.