Design Patterns

Software Engineering, Summer 2012

Andreas Zeller, Saarland University

Design Patterns

In this chapter we will examine typical usage scenarios of object-oriented programming - the so called *design patterns*.

The name *design pattern* was coined by the architect Christopher Alexander:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

A pattern is a *template* that can be used in many different situations.

Patterns in Architecture: *Window Place*

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them

In every room where you spend any length of time during the day, make at least one window into a “window place”

low sill

place

Window place

Patterns in Software Design

In our case design patterns are

descriptions of communicating objects and classes, that have been adapted to solve a design problem in a specific context.

The Elements of a Pattern

Patterns are usually combined into *catalogues*: manuals that contain patterns for future reuse.

Every pattern can be described by at least four characteristics:

- Name
- Problem
- Solution
- Consequences

The Elements of a Pattern (2)

The name of the pattern is used to describe the design problem and its solution in one or two words.

it enables us to

- design things on a higher level of abstraction
- use it under this name in the documentation
- speak of it

The problem describes when the pattern is used.

- it describes the problem and its context
- can describe certain design problems
- can contain certain *operational conditions*

The Elements of a Pattern (3)

The solution describes the parts the design consists of, their relations, responsibilities and collaborations - in short, the *structure* and *participants*:

- not a description of a concrete design or an implementation
- but rather an *abstract description* of a design problem, and how a general interaction of elements solves it

The consequences are results, benefits and drawbacks of a pattern:

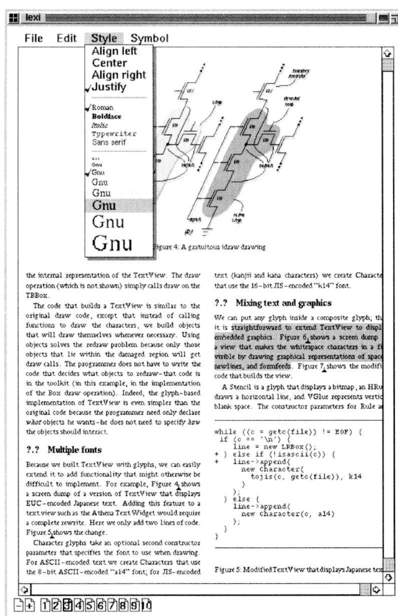
- *assessments of resource usage* (memory usage, running time)
- influence on *flexibility*, *extendiblility*, and *portability*

Case Study: Text Editor *Lexi*

Let's consider the design of a "what you see is what you get" ("WYSIWYG") text editor called *Lexi*.

Lexi is able to combine text and graphics in a multitude of possible layouts.

Let's examine some design patterns that can be used to solve problems in *Lexi* and similar applications.



Challenges

Document structure. How is the document stored internally?

Formatting. How does *Lexi* order text and graphics as lines and polygons?

Support for multiple user interfaces. *Lexi* should be as independent of concrete windowing systems as possible.

User actions. There should be a unified method of accessing *Lexi's* functionality and undoing changes.

Each of these design problems (and their solutions) is illustrated by one or multiple design patterns.

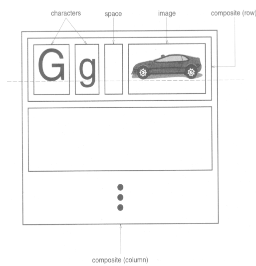
Displaying Structure - Composite Pattern

A *document* is an arrangement of basic graphical elements like glyphs, lines, polygons etc.

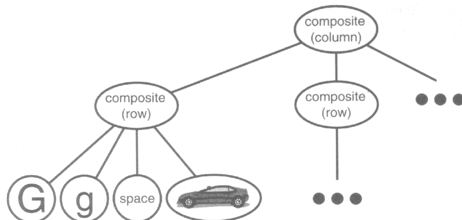
These are combined into *structures* - rows, columns, figures, and other substructures.

Such hierarchically ordered information is usually stored by means of *recursive composition* - simpler elements are combined into more complex ones.

Elements in a Document

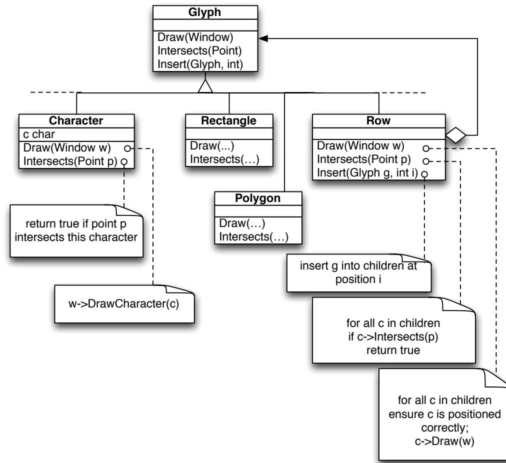


For each important element there is an individual object.



Glyphs

We define an abstract superclass Glyph for all objects that can occur in a document.



Glyphs (2)

Each glyph knows

- how to draw itself (by means of the Draw() method). This abstract method is implemented in concrete subclasses of Glyph.
- how much space it takes up (like in the Intersects() method).
- its children and parent (like in the Insert() method).

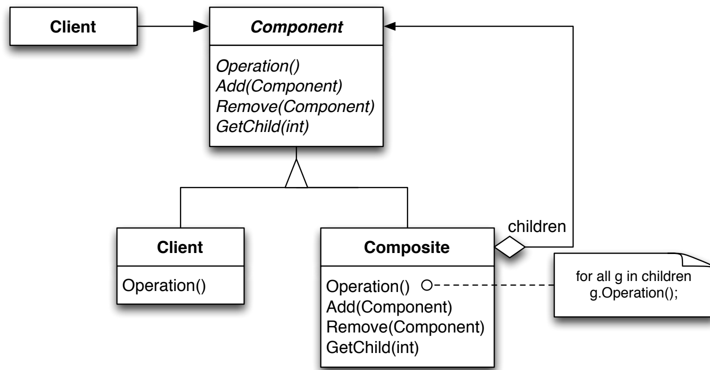
The class hierarchy of the *Glyph* class is an instance of the *composite* pattern.

The Composite Pattern

Problem Use the composite pattern if

- you want to express a part-of-a-whole hierarchy
- the application ignores differences between composed and simple objects

Structure



Participants

Component (Glyph)

- defines the interface for all objects (simple and composed)
- implements the default behavior for the common interface (where applicable)
- defines the interface for accessing and managing of subcomponents (children)

Leaf (e.g. rectangle, line, text)

- provides for basic objects; a leaf doesn't have any children
- defines common behavior of basic elements

Participants (2)

Composite (e.g. picture, column)

- defines common behavior of composed objects (those with children)
- stores subcomponents (children)
- implements methods for accessing children as per interface of *Component*

Client (User)

- manages objects by means of the *Component* interface

Consequences

The composite pattern

- defines class hierarchies consisting of composed and basic components
- simplifies the user: he can use basic and composed objects in the same way; he doesn't (and shouldn't) know whether he is handling a simple or complex object.

Consequences (2)

The composite pattern

- simplifies adding of new kinds of elements
- can *generalize* the design too much: for example, the fact that a certain composed element has a fixed number of children, or only certain kinds of children can only be checked at runtime (and not at compile time). ⇐ *This is a drawback!*

Other known fields of application: expressions, instruction sequences

Encapsulating of Algorithms - Strategy Pattern

Lexi has to wrap the text in rows and combine rows into columns - as the user wishes it.

This is the task of the formatting algorithm.

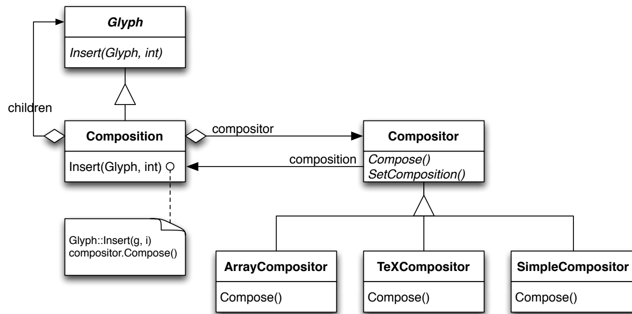
Lexi must support multiple formatting algorithms e.g.

- a fast, imprecise ("quick-and-dirty") algorithm for the WYSIWYG view
- a slow and precise one for printing

In accordance with the separation of interests, the formatting algorithm must be independent of the document structure.

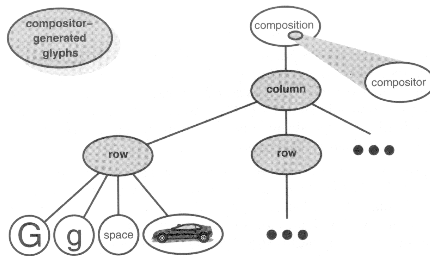
Formatting Algorithms

We define a separate class hierarchy for objects that encapsulate certain formatting algorithms. The root of this hierarchy is the Compositor abstract class with a general interface; every subclass implements a concrete formatting algorithm.



Formatting Algorithms (2)

Every Compositor traverses the document structure and possibly inserts new (composed) Glyphs:



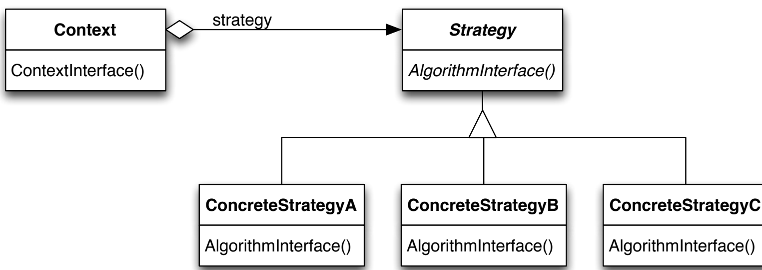
This is an instance of the strategy pattern.

Strategy Pattern

Problem Use the strategy pattern if

- multiple connected classes differ only in behavior
- different variants of an algorithm are needed
- an algorithm uses data that shall be concealed from the user

Structure



Participants

Strategy (Compositor)

- defines a common interface for all supported algorithms

ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)

- implements the algorithm as per Strategy interface

Context (Composition)

- is configured with a ConcreteStrategy object
- references a Strategy object
- can define an interface that makes data available to Strategy

Consequences

The strategy pattern

- makes conditional statements unnecessary (e.g. if simple-composition then... else if tex.composition...)
- helps to identify the common functionality of all the algorithms
- enables the user to choose a strategy...
- ... but burdens him with a choice of strategy!
- can lead to a communication overhead: data has to be provided even if the chosen strategy doesn't make use of it

Other fields of application: code optimization, memory allocation, routing algorithms

User Actions - Command Pattern

Lexi's functionality is accessible in multiple ways: you can manipulate the WYSIWYG representation (enter text, move the cursor, select text), and you can choose additional actions via menus, panels, and hotkeys.

We don't want to bind any action to a specific user interface because

- there may be multiple ways to initiate the same action (you can navigate to the next page via a panel, a menu entry, and a keystroke)
- maybe we want to change the interface at some later time

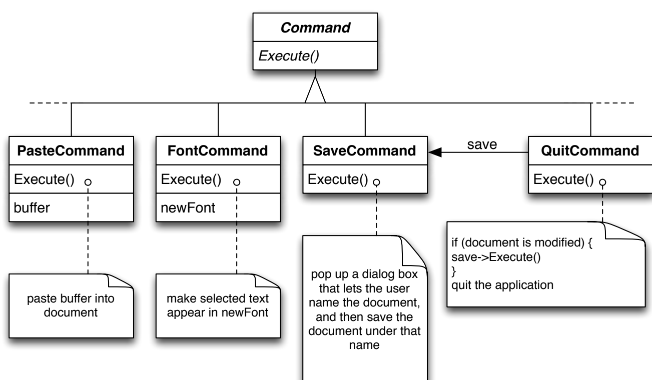
User Actions (2)

To complicate things even more, we want to enable *undoing* and *redoing* of multiple actions.

Additionally, we want to be able to record and play back *macros* (instruction sequences).

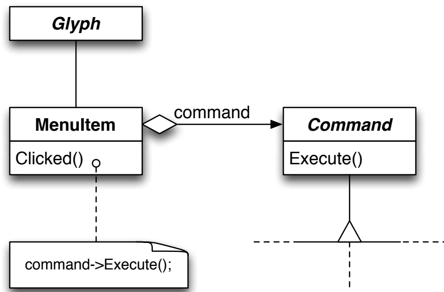
User Actions (3)

Therefore we define a *Command* class hierarchy what encapsulates the user actions.



User Actions (4)

Specific glyphs can be bound to user actions; they are executed when the glyph is activated.



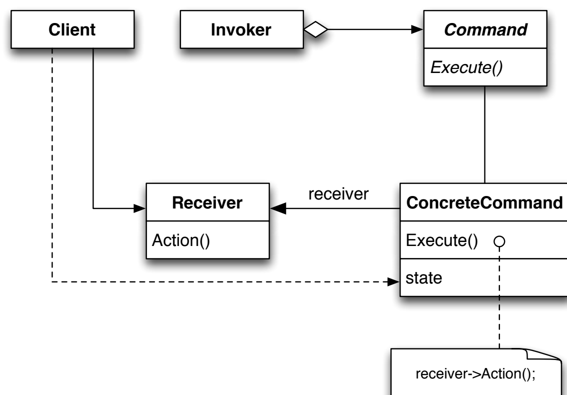
This is an instance of the *command* pattern.

Command Pattern

Problem Use the command pattern if you want to

- parameterize objects with the action to be performed
- trigger, enqueue, and execute instructions at different points in time
- support undoing of instructions
- log changes to be able to restore data after a crash

Structure



Participants

Command

- defines the interface to execute an action

ConcreteCommand (PasteCommand, OpenCommand)

- defines a coupling between a receiving object and an action
- implements Execute() by calling appropriate methods on the receiver

Client (User, Application)

- creates a ConcreteCommand object and sets a receiver

Participants (2)

Invoker (Caller, MenuItem)

- ask the instruction to execute its action

Receiver (Document, Application)

- knows how the methods, that are coupled with an action, are to be executed. Any class can be a receiver.

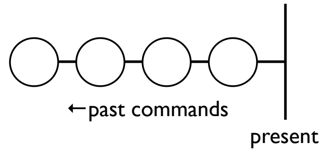
Consequences

The command pattern

- decouples the object that triggers an action from the object that knows how to execute it
- implements Commands as *first-class* objects that can be handled and extended like any other object
- allows to combine Commands from other Commands
- makes it easy to add new Commands because existing classes don't have to be changed

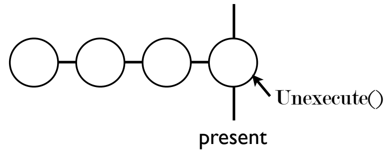
Undoing Commands

With the help of a Command-Log we can easily implement command undoing. It looks like this:



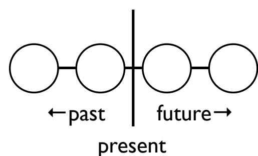
Undoing Commands (2)

To undo the last command we call Unexecute() on the last command. This means that each command has to store enough state data to be able to undo itself.



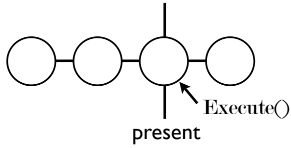
Undoing Commands (3)

After undoing, we move the "Present-Line" one command to the left. If the user chooses to undo another command we end up in this state:



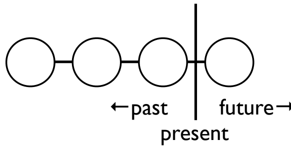
Undoing Commands (4)

To redo a command, we simply have to call Execute() on the current command...



Undoing Commands (5)

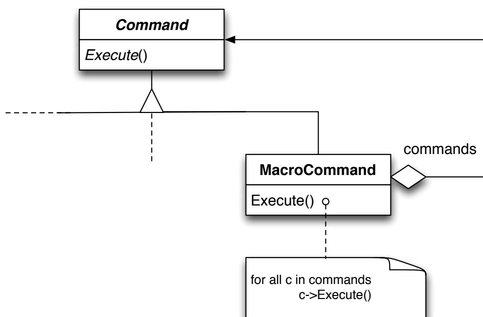
... and move the "Present-Line" one command to the right, so the next call to Execute() will redo the next command.



This way the user can navigate back and forth in time depending on how far he has to go to correct an error.

Macros

Lastly, let's consider an implementation of macros (instruction sequences). We use the command pattern and create a MacroCommand class that contains multiple command and can execute them successively:



If we add an Unexecute() method to the MacroCommand class, then we can undo macros like any other command.

In Summary

With *Lexi* we have familiarized ourselves with the following design patterns:

- *Composite* for representation of the internal document structure
- *Strategy* for support of multiple formatting algorithms
- *Command* for undoing commands and creating macros

None of these patterns are limited to a concrete field of application; they are also insufficient to solve every possible design problem.

In Summary (2)

In summary, design patterns offer:

A common design vocabulary. Design patterns offer a common design vocabulary for software engineers for communicating, documenting, and exchanging design alternatives.

Documentation and learning help. The most large object-oriented systems use design patterns. Design patterns help to understand such systems.

An extension of existing methods. Design patterns concentrate the experience of experts - independently of the design method.

"The best designs will use many design patterns that dovetail and intertwine to produce a greater whole."

Case Study: Spreadsheet

	A	B
1	1	11 = A1 + A2
2	10	111 = B1 + A3
3	100	

A *spreadsheet* consists of *m x n* cells.

Cells are either empty or they have *content*.

Contents can be *numbers, texts*, or *formulas*.

There are multiple *contents* for a formula (that serve as operands)

Each *formula* has a *result* (a content)

Object Model

Spreadsheet
1
has ▸
0..*
Cell
+get_value(): Content
+enter_data(s: string)
cell
1
content
0..1
Content
+get_value(): Content
+enter_data(s: string)
-notify()
0..* operand
1 result
formula
1
1
Number
-value: double
+enter_data(s:string)
Text
-value: string
+enter_data(s:string)
Formula
+get_value(): Content
+enter_data(s: string)
-refresh()

Relationships between Objects

	A	B
1	1	11 = A1 + A2
2	10	111 = B1 + A3
3	100	

a1: Number
value = 1
operand
a2: Number
value = 10
operand
b1: Formula
b1 = a1 + a2
result
b1': Number
value = 11
operand
a3: Number
value = 100
operand
b2: Formula
b2 = b1 + a3
result
b2': Number
value = 111

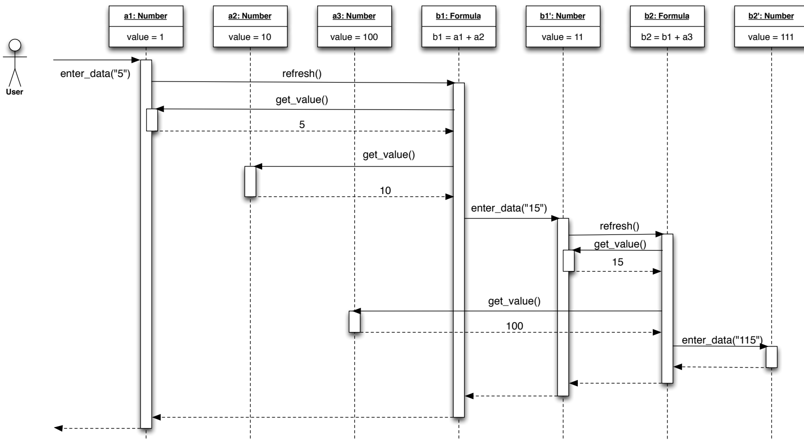
State Chart

The method enter_data() of the Content class examines whether the actual value has changed. If it has, every Formula that has this Content as an operand is notified by means of the method notify().

constant value
enter_data()
changed value
[new value = old value]
notify()

Sequence Diagram

Example: Let the spreadsheet be filled out as just described; now the value of cell A1 is changed from 1 to 5.

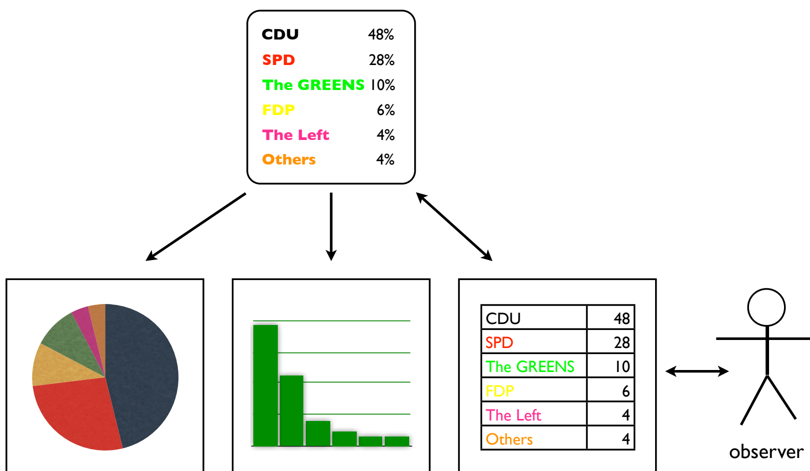


Model-View-Controller

The *Model-View-Controller* pattern is one of the best known and most common patterns in the architecture of *interactive* systems.

Example: Election Day

Let's examine an information system for elections that offers several different *views* on prognoses and results.



Problem

User interfaces are most frequently affected by changes.

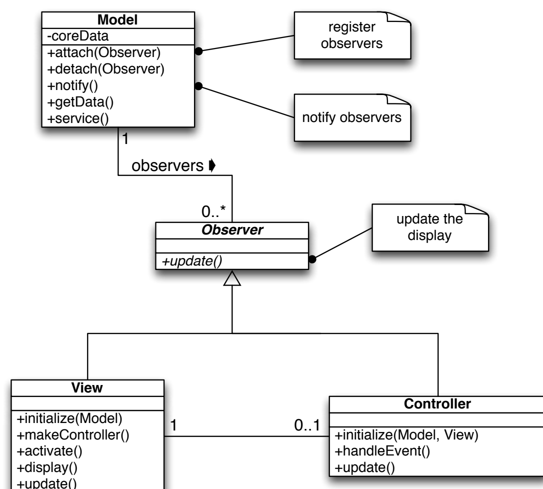
- How can I represent the same information in different ways?
- How can I guarantee that changes in the dataset will be instantly reflected in all views?
- How can I change the user interface? (possibly at runtime)
- How can I support multiple user interfaces without changing the core of the application?

Solution

The *Model-View-Controller* pattern splits the application into three parts:

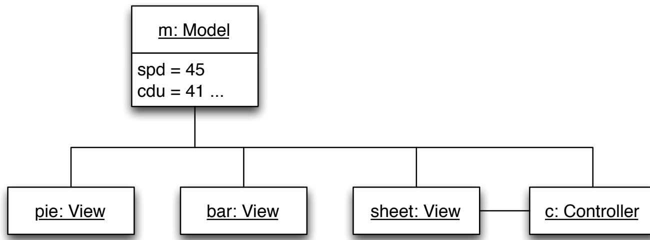
- The *model* is responsible for processing,
- The *view* takes care of output,
- The *controller* concerns itself with input

Structure



Structure (2)

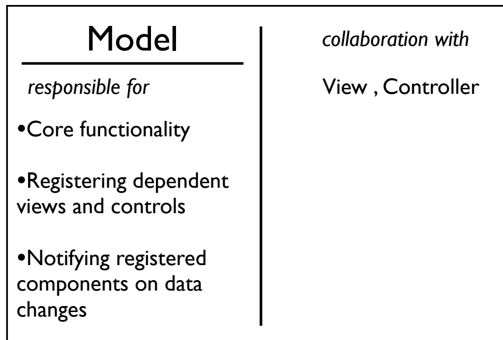
Each model can *register* multiple *observers* (= views and controllers).



As soon as changes occur in the model, all registered observers are *notified*, and they update themselves accordingly.

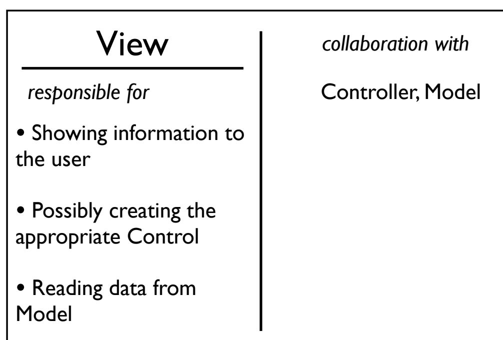
Participants

The **model** encapsulates core data and functionality; it is independent of any concrete output representation, or input behavior.



Participants (2)

The **view** displays information to the user. A model can have multiple views.



Participants (3)

The controller processes input and invokes the appropriate services of the view or the model; every controller is assigned to a single view; a model can have multiple controllers.

Controller	*collaboration with*
responsible for • Accepting user input • Matching inputs to service invocations (displaying services of the View, or services of the Model)	View, Model

Dynamic behavior

Consequences of the Model-View-Controller Pattern

Benefits

- multiple views of the same system
- synchronous views
- attachable views and controllers

Drawbacks

- increased complexity
- strong coupling between Model and View
- Strong coupling between Model and Controllers (can be avoided by means of the command pattern)

Known fields of application: GUI-libraries, Smalltalk, Microsoft Foundation Classes

Anti-Patterns

If the following patterns occur in your software project, you're doing it wrong!

Anti-Patterns: Programming

The Blob. One object("blob") has the majority of the responsibilities, while most of the others just store data or provide only primitive services.

Solution: refactoring

The Golden Hammer. A favorite solution ("Golden Hammer") is applied to every single problem: With a hammer, every problem looks like a nail.

Solution: improve level of education

Copy-and-Paste Programming. Code is reused in multiple places by being copied and adjusted. This causes a maintenance problem.

Solution: Black box reuse, identifying of common features.

Anti-Patterns: Programming (2)

Spaghetti Code. The code is mostly unstructured; it's neither particularly modular nor object-oriented; control flow is obscure.

Solution: Prevent by designing first, and only then implementing. Existing spaghetti code should be refactored.

Mushroom Management. Developers are kept away from users.

Solution: Improve contacts.

Anti-Patterns: Architecture

Vendor Lock-In. A system is dependent on a proprietary architecture or data format.

Solution: Improve portability, introduce abstractions.

Design by Committee. The typical anti-pattern of standardizing committees, that tend to satisfy every single participant, and create overly complex and ambivalent designs ("A camel is a horse designed by a committee").

Known examples: SQL and COBRA.

Solution: Improve group dynamics and meetings (teamwork)

Anti-Pattern: Architecture (2)

Reinvent the Wheel. Due to lack of knowledge about existing products and solutions, the wheel gets reinvented over and over, which leads to increased development costs and problems with deadlines.

Solution: Improve knowledge management.

Anti-Patterns: Management

Intellectual Violence. Someone who has mastered a new theory, technique or buzzwords, uses his knowledge to intimidate others.

Solution: Ask for clarification!

Project Mismanagement. The manager of the project is unable to make decisions.

Solution: Admit having the problem; set clear short-term goals.

Other Anti-Patterns

- *Lava Flow* (design changes frequently)
- *Boat Anchor* (a component has no apparent user)
- *Dead End* (a bought component that isn't supported any longer)
- *Swiss Army Knife* (a component that pretends to be able to do everything)