

9
2

Patrones de diseño

Introducción

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2024-2025

Problemas repetidos

Los ingenieros de software se enfrentan cada día a multitud de problemas de distinto calibre.

La efectividad de un ingeniero se mide por su rapidez y acierto en la diagnosis, identificación y resolución de tales problemas.

**El mejor ingeniero es el que
más reutiliza la misma solución
—matizada— para resolver
problemas similares.**

No reinventar la rueda

La orientación a objetos propugna no reinventar la rueda en la pura codificación respecto de la resolución de problemas.

¿Por qué, entonces, reinventarla para el ataque genérico a problemas comunes de diseño e implementación?

Debe existir alguna forma de comunicar al resto de los arquitectos software los resultados encontrados tras mucho esfuerzo por algunos de ellos.

**Se necesita, en definitiva, algún
esquema de documentación que
permita tal comunicación.**

Adaptación al cambio

Diseñar software orientado a objetos es difícil.

Pero diseñar software orientado a
objetos reutilizable lo es aún más.

*You must find pertinent objects,
factor them into classes at the
right granularity, define class
interfaces and inheritance
hierarchies, and establish key
relationships among them.*

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley Longman Publishing.

**El diseño debe ser específico
del problema que tenemos
entre manos.**

Pero también lo suficientemente
general como para abordar futuros
problemas y necesidades.

Queremos evitar el rediseño.

O, al menos, minimizarlo.

En definitiva . . .

**Perseguimos un diseño
flexible.**

Es casi imposible dar con él la primera vez.

*Yet experienced object-oriented
designers do make good designs.
Meanwhile new designers are
overwhelmed by the options
available and tend to fall back
on non-object-oriented
techniques they've used before.*

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley Longman Publishing.

It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley Longman Publishing.

Que tienden a reutilizar
soluciones que se han probado
útiles en el pasado.

Y que es lo que les hace ser expertos.

*You'll find recurring patterns
of classes and communicating
objects in many object-oriented
systems.*

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley Longman Publishing.

These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience.

A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

We all know the value of design experience. How many times have you had design déjà-vu—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it.



youtube.com

YouTube ES

Buscar

9+

PRACTICAL
CREATIVITY

Raph Koster
GDC Next 2014

GDC NEXT

Practical Creativity

GDC 524 K suscriptores

Suscribirme

10 K Compartir Clip ...

Todos GDC Videojuego independiente >

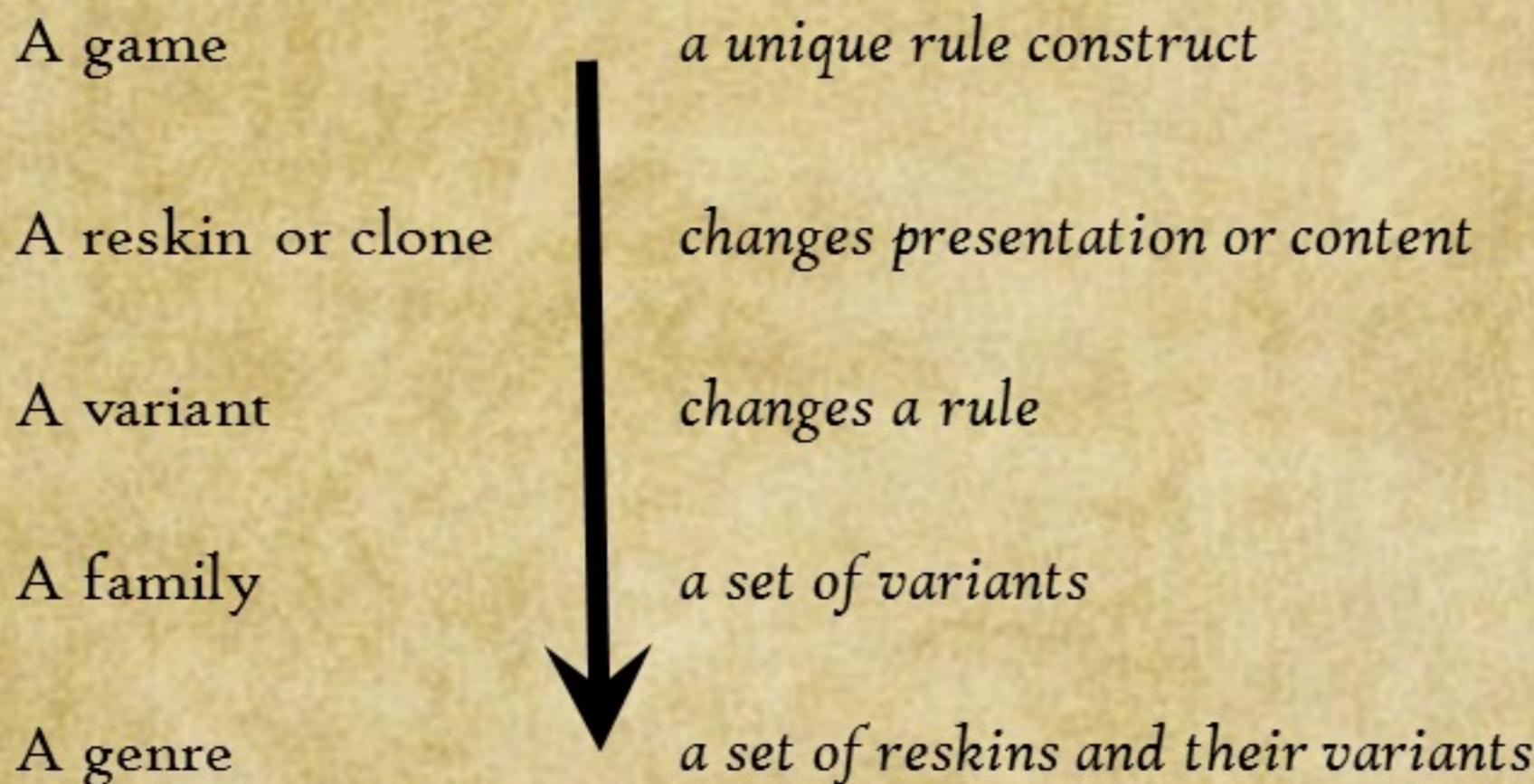
309.612 visualizaciones 15 ene 2017

In this 2014 GDC Next session, MMD designer Raph Koster explains what science tells us about creativity, and offers practical straightforward steps that any game designer or developer can make use of in order to get more creative.

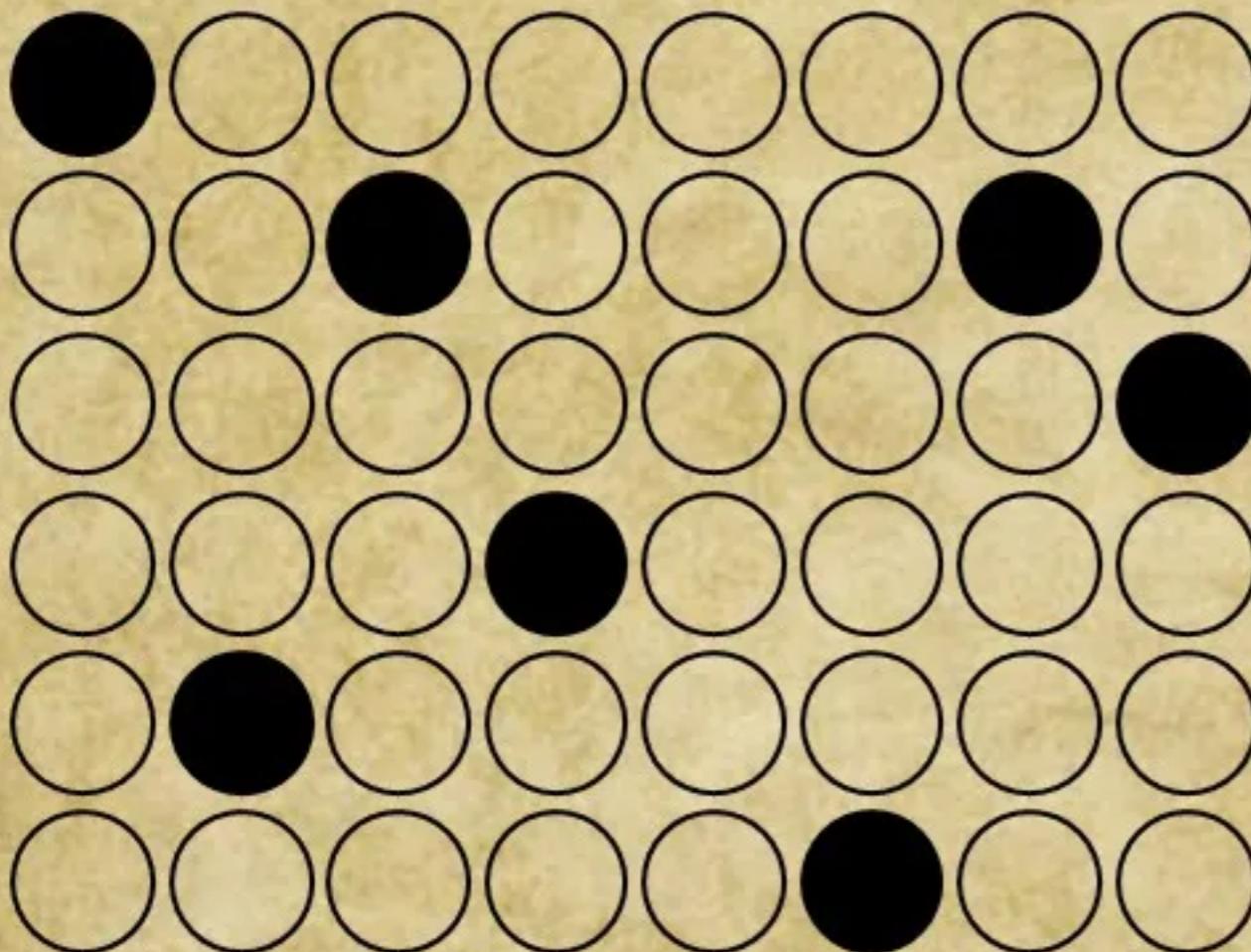
5 Signs of an Inexperienced Self-Taught Developer (and ho...
Travis Media 205 K visualizaciones hace 8...

You're not stupid - How to learn

3 A hierarchy

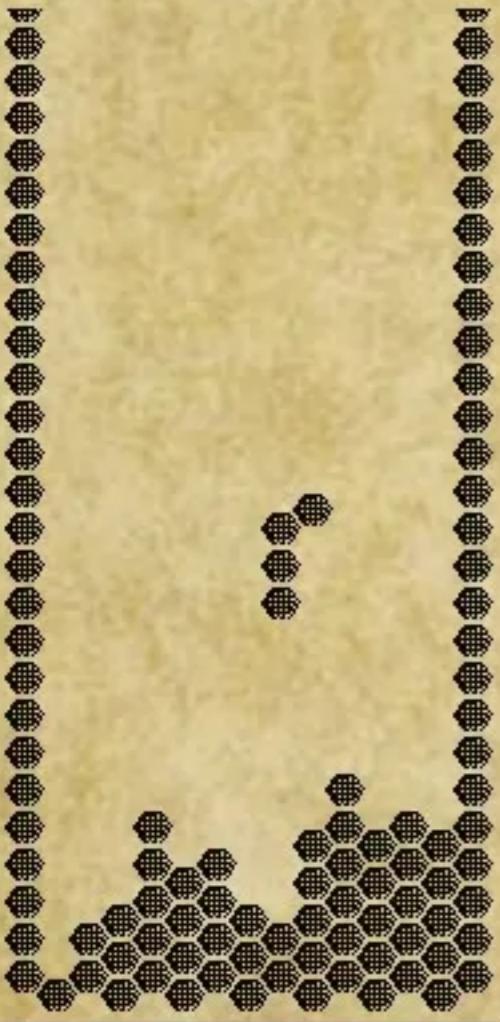


3 Merge a mechanic

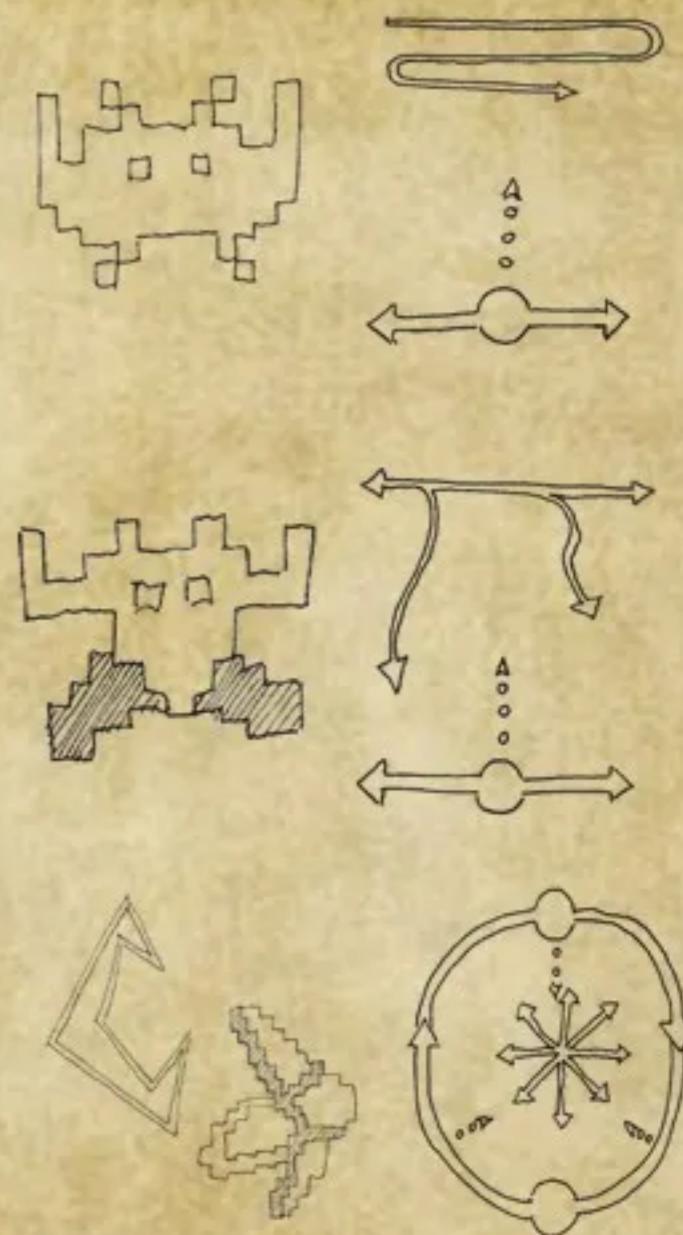


- “Visit every node on the graph” is *Pac-Man*.
- But if you mark only certain nodes as ones you must visit, it changes into a game of “pick up objects” as in *Lode Runner* or *Jumpman*.
- Make the locations hidden information, and you invented the “secrets” system.

3 Change topologies



- A lot of game evolution is driven simply by changing the shape of the graph.
 - Blokus to Trigon or Gemblo
 - Wrapping a game on a torus or visually bending a plane.









GESUND

72

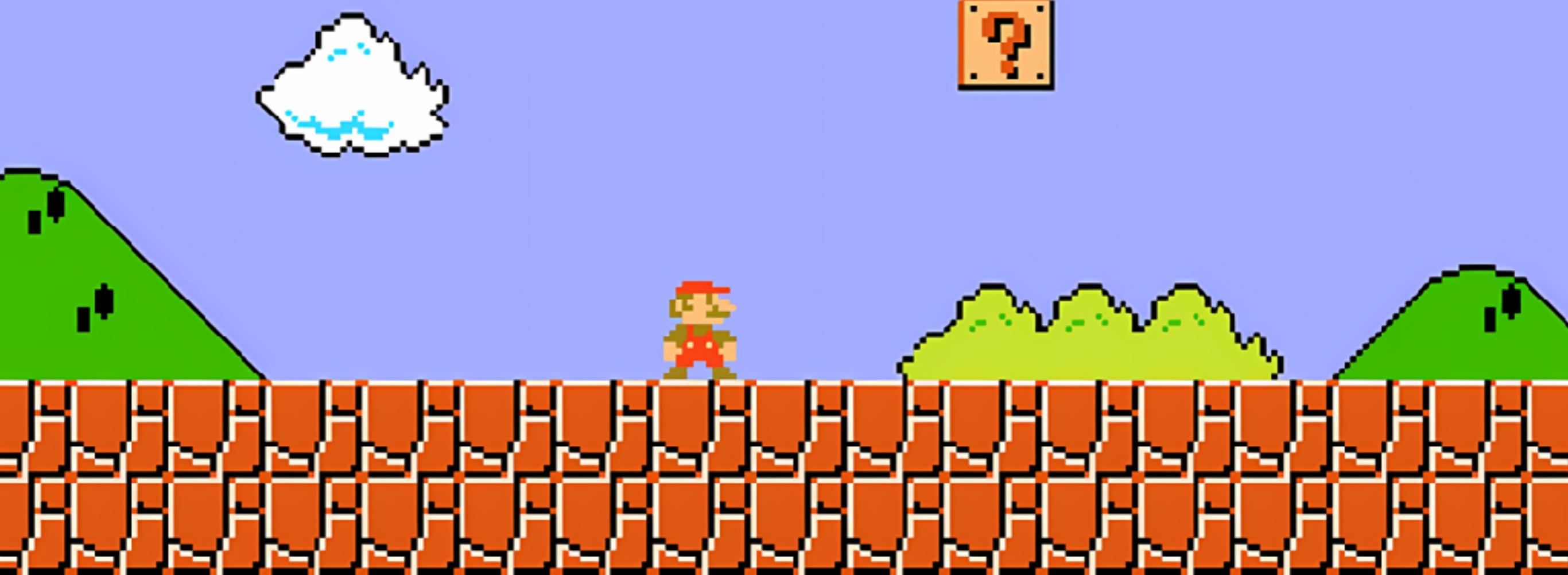
ANZUG

0





SUPER
MARIO BROS.



Ejemplos de problemas típicos

¿Cómo representar estados con objetos?

¿Cómo añadir responsabilidades dinámicamente a un objeto?

Muchos diseños OO resuelven esos
y otros muchos problemas similares.

Que son genéricos, no específicos de una
aplicación concreta

¿No hay forma de documentar ese conocimiento
de alguna forma?

Para eso surgen los
patrones de diseño.

Patterns help you build on the collective experience of skilled software engineers. They capture existing, well-proven experience and help to promote good design practise.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns. Volume 1*. Chichester, England: John Wiley & Sons

What is so exciting about patterns? It is probably the fact that they constitute a ‘grass roots’ effort to build on the collective experience of skilled designers and software engineers. Such experts already have solutions to many recurring design problems. Patterns capture these proven solutions in an easily-available and, hopefully, well-written form.

Qué es un patrón
de diseño

Cada patrón describe un problema que se repite una y otra vez en nuestro entorno, y describe el núcleo de la solución a dicho problema, de modo que pueda usarse esta solución millones de veces, aunque siempre de forma ligeramente distinta.

—Christopher Alexander (1977)

A Pattern Language

Towns • Buildings • Construction



Christopher Alexander

Sara Ishikawa • Murray Silverstein

WITH

Max Jacobson • Ingrid Fiksdahl-King

Shlomo Angel

El arquitecto Christopher Alexander, en su libro *A Pattern Language. Towns, Buildings, Construction* (Oxford University Press, 1977) presenta 253 patrones referidos al modo de diseñar edificios y ciudades de forma que tenga en cuenta las necesidades de sus habitantes.

Se le conoce como el libro AIS (las iniciales de los tres primeros autores).

**Los patrones de diseño
software se consideran
deudores de los trabajos
de Alexander.**

**La calidad
sin nombre**

The
Timeless Way of
Building



Christopher Alexander

The Timeless Way of Building

Alexander

*

Oxford

La calidad sin nombre

¿Existe en verdad una parte común en los buenos diseños, a veces tan dispares entre sí?

Christopher Alexander así lo afirma, y da a esta parte la elusiva calificación de «la calidad que no se puede nombrar».

Alexander sostiene que existe un «algo innombrable» que no puede ser modelado únicamente por medio de un conjunto arbitrario de requisitos. Los sistemas poseerían, así, una esencia cualitativa que les otorgaría verdadera identidad y equilibraría sus fuerzas internas.

Algunos ejemplos

* * *

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

A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease
[...]

If the room contains no window which is a “place”, a person in the room will be torn between two forces:

1. He wants to sit down and be comfortable.
2. He is drawn toward the light.

* * *

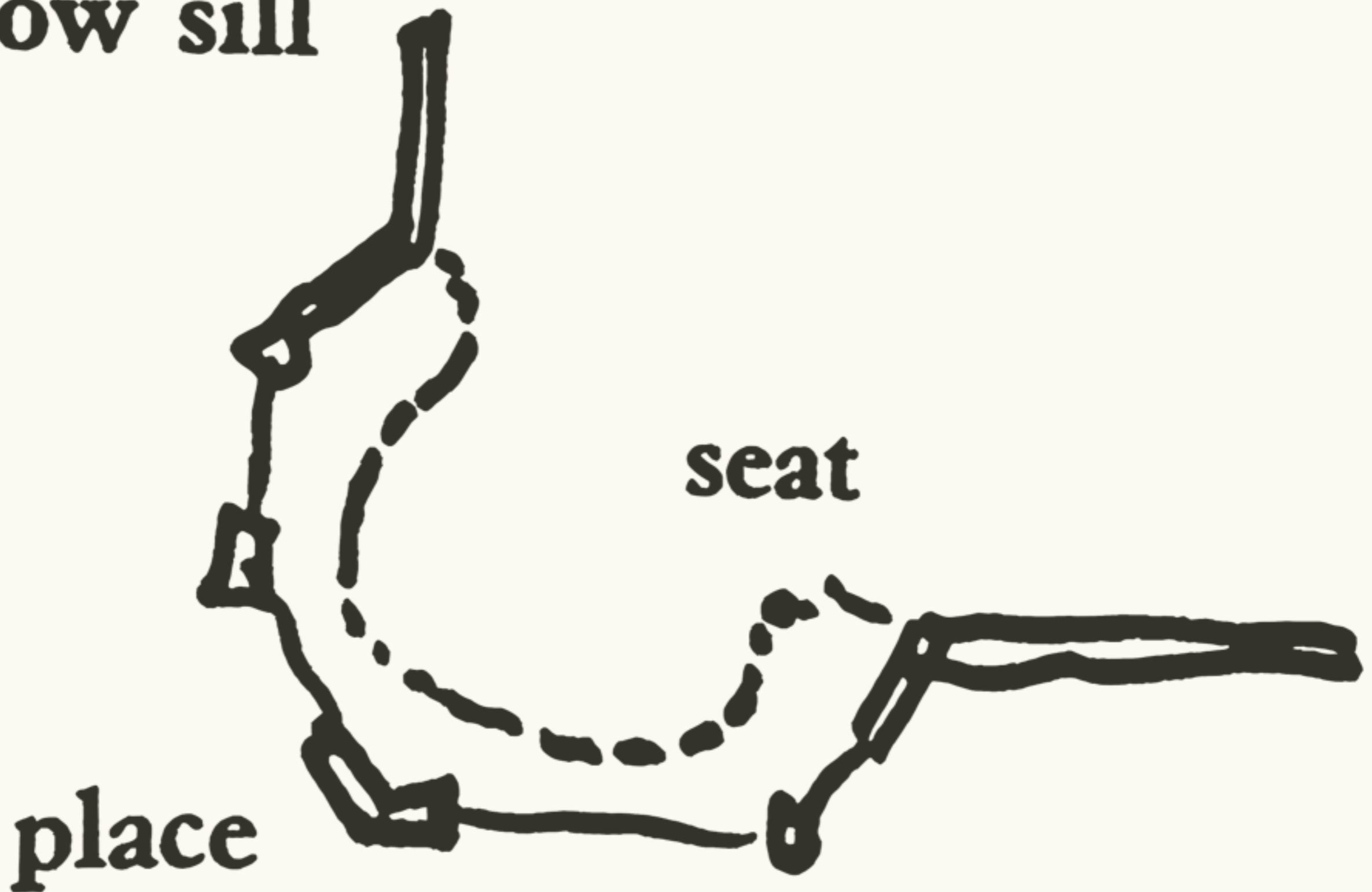
Obviously, if the comfortable places —those places in the room where you most want to sit— are away from the windows, there is no way of overcoming this conflict [...]

Therefore:

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

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. 1977. Window Place. In *A Pattern Language* (pp.). New York: Oxford University Press

low sill



seat

place

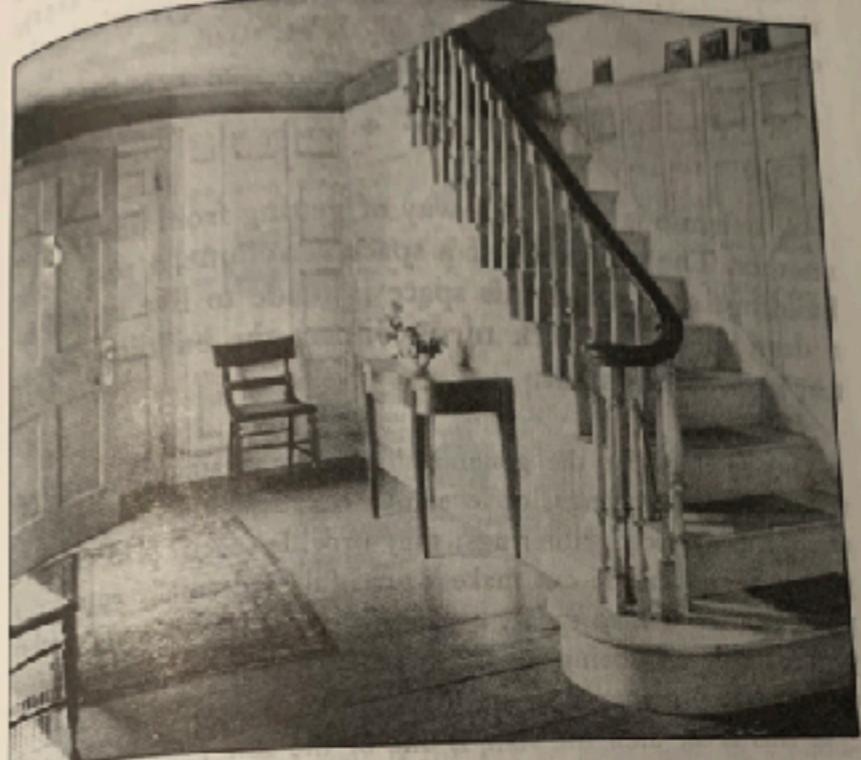








133 STAIRCASE AS A STAGE



... if the entrances are in position—MAIN ENTRANCE (110); and the pattern of movement through the building is established—THE FLOW THROUGH ROOMS (131), SHORT PASSAGES (132), the main stairs must be put in and given an appropriate social character.

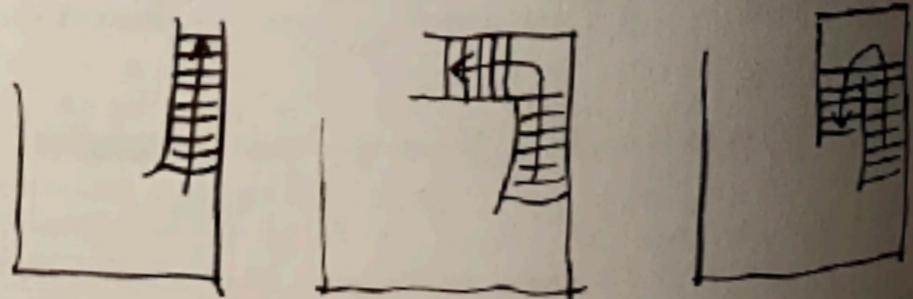
* * *

A staircase is not just a way of getting from one floor to another. The stair is itself a space, a volume, a part of the building; and unless this space is made to live, it will be a dead spot, and work to disconnect the building and to tear its processes apart.

Our feelings for the general shape of the stair are based on this conjecture: changes of level play a crucial role at many moments during social gatherings; they provide special places to sit, a place where someone can make a graceful or dramatic entrance, a place from which to speak, a place from which to look at other people while also being seen, a place which increases face to face contact when many people are together.

If this is so, then the stair is one of the few places in a building which is capable of providing for this requirement, since it is almost the only place in a building where a transition between levels occurs naturally.

This suggests that the stair always be made rather open to the room below it, embracing the room, coming down around the outer perimeter of the room, so that the stairs together with the room form a socially connected space. Stairs that are enclosed in stairwells or stairs that are free standing and chop up the space

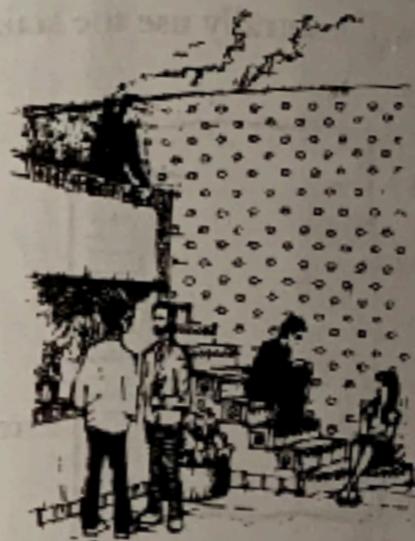


Examples of stair rooms.

133 STAIRCASE AS A STAGE

below, do not have this character at all. But straight stairs, stairs that follow the contour of the walls below, or stairs that double back can all be made to work this way.

Furthermore, the first four or five steps are the places where people are most likely to sit if the stair is working well. To support this fact, make the bottom of the staircase flare out, widen the steps, and make them comfortable to sit on.



Stair seats.

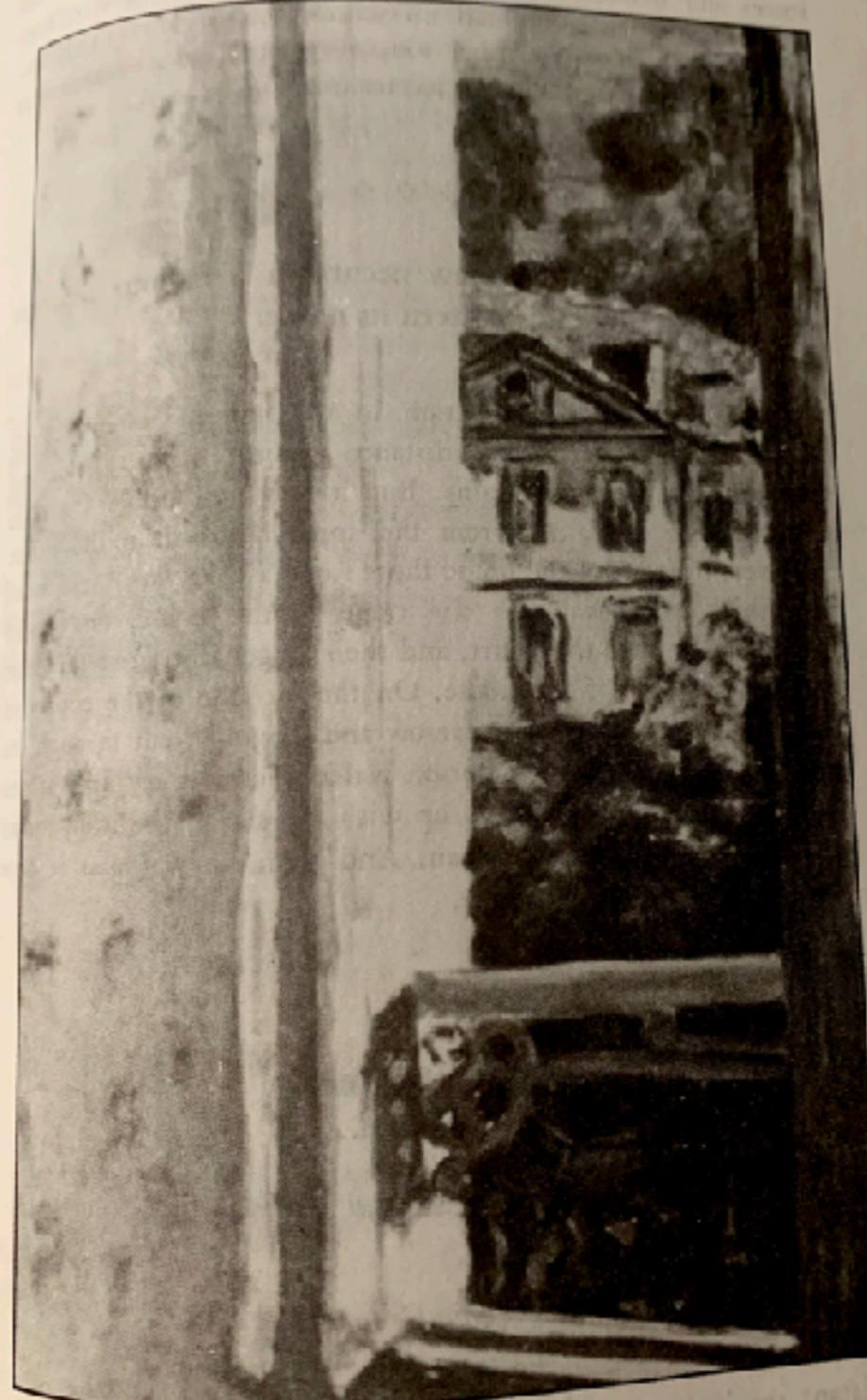
Finally, we must decide where to place the stair. On the one hand, of course, the stair is the key to movement in a building. It must therefore be visible from the front door; and, in a building with many different rooms upstairs, it must be in a position which commands as many of these rooms as possible, so that it forms a kind of axis people can keep clearly in their minds.

However, if the stair is too near the door, it will be so public that its position will undermine the vital social character we have described. Instead, we suggest that the stair be clear, and central, yes—but in the common area of the building, a little further back from the front door than usual. Not usually in the ENTRANCE ROOM (130), but in the COMMON AREA AT THE HEART (129). Then it will be clear and visible, and also keep its necessary social character.

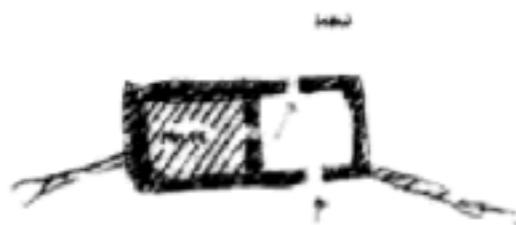
Therefore:

Place the main stair in a key position, central and vis-

134 ZEN VIEW*



A Buddhist monk lived high in the mountains, in a small stone house. Far, far in the distance was the ocean, visible and beautiful from the mountains. But it was not visible from the monk's house itself, nor from the approach road to the house. However, in front of the house there stood a courtyard surrounded by a thick stone wall. As one came to the house, one passed through a gate into this court, and then diagonally across the court to the front door of the house. On the far side of the courtyard there was a slit in the wall, narrow and diagonal, cut through the thickness of the wall. As a person walked across the court, at one spot, where his position lined up with the slit in the wall, for an instant, he could see the ocean. And then he was past it once again, and went into the house.



The monk's house.

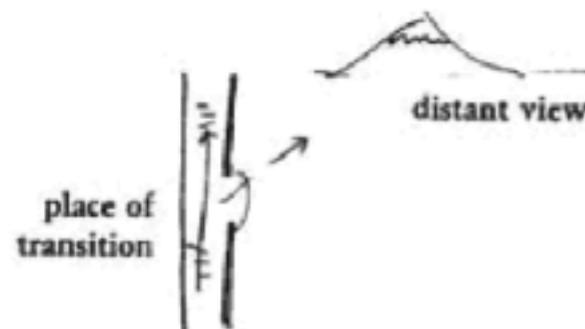
What is it that happens in this courtyard? The view of the distant sea is so restrained that it stays alive forever. Who, that has ever seen that view, can ever forget it? Its power will never fade. Even for the man who lives there, coming past that view day after day for fifty years, it will still be alive.

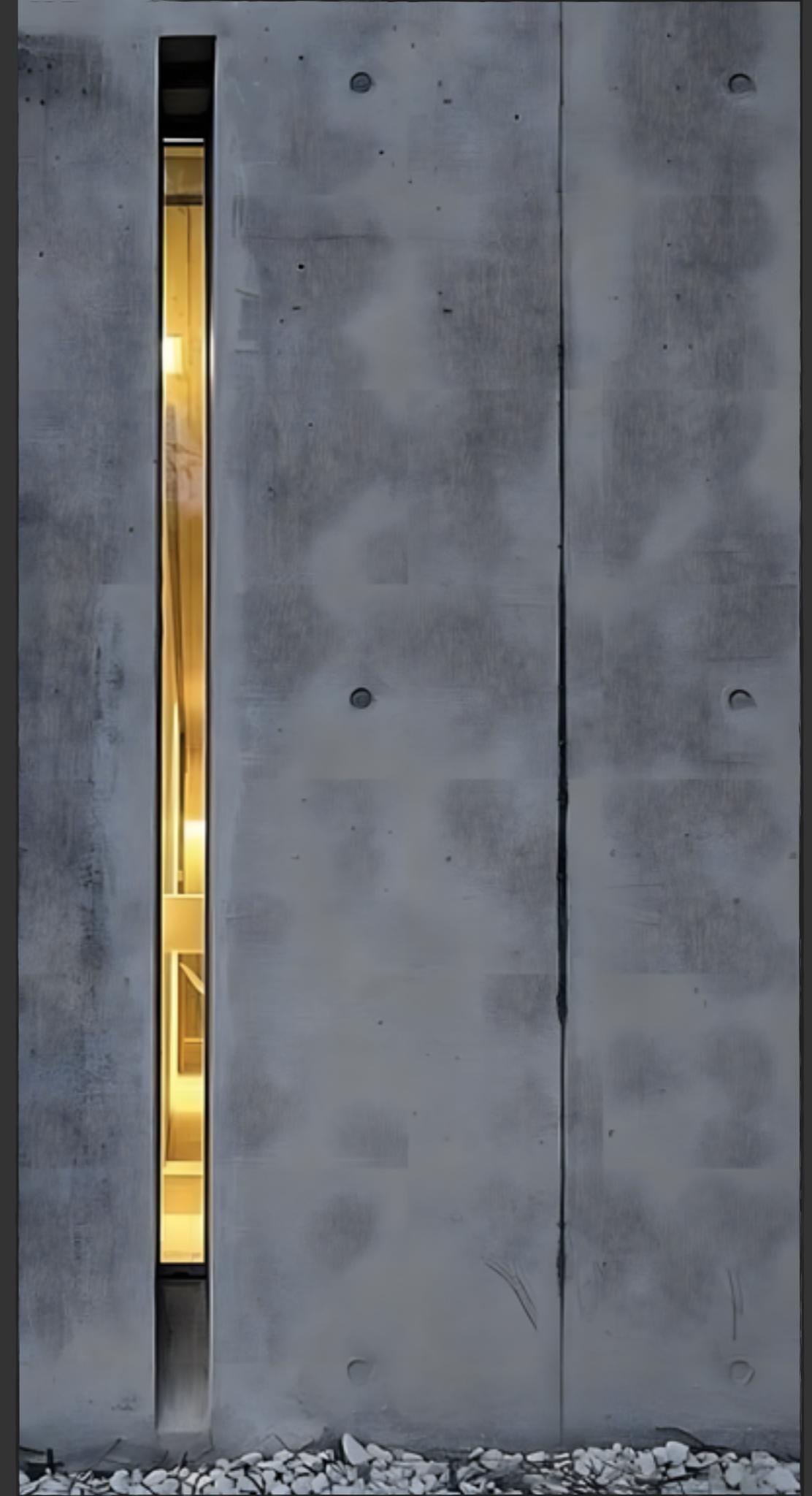
This is the essence of the problem with any view. It is a beautiful thing. One wants to enjoy it and drink it in every day. But the more open it is, the more obvious, the more it shouts, the sooner it will fade. Gradually it will become part of the building, like the wallpaper; and the intensity of its beauty will no longer be accessible to the people who live there.



Therefore:

If there is a beautiful view, don't spoil it by building huge windows that gape incessantly at it. Instead, put the windows which look onto the view at places of transition along paths, in hallways, in entry ways, on stairs, between rooms. If the view window is correctly placed, people will see a glimpse of the distant view as they come up to the window or pass it: but the view is never visible from the places where people stay.





Un ejemplo rural



Si nos fijamos en las construcciones de una determinada zona rural observaremos que todas ellas poseen apariencias parejas (por ejemplo, tejados de pizarra con gran pendiente, una determinada orientación, etcétera), pese a que los requisitos personales por fuerza han debido ser distintos. De alguna manera la esencia del diseño se ha copiado de una construcción a otra, y a esta esencia se pliegan de forma natural los diversos requisitos. Diríase aquí que existe un «patrón» que soluciona de forma simple y efectiva los problemas de construcción en tal zona.

Historia



History Of Patterns

In 1987, Ward and Kent were consulting with Tektronix's [SemiconductorTestSystemsGroup](#) that was having troubles finishing a design. They decided to try out the pattern stuff they'd been studying. Like Alexander who said the occupiers of a building should design it, Ward and Kent let representatives of the users (a trainer and a field engineer) finish the design.

Ward came up with a five pattern "language" that helped the novice designers take advantage of Smalltalk's strengths and avoid its weaknesses:

- [WindowPerTask](#)
- [FewPanes](#)
- [StandardPanes](#)
- [NounsAndVerbs](#)
- [ShortMenus](#)

Ward and Kent were amazed at the (admittedly spartan) elegance of the interface their users designed. They reported the results of this experiment at OOPSLA 87 in Orlando [4]. They wrote up a panel position, and presented at [NormKerth](#)'s workshop on *Where do objects come from?* They talked patterns until they were blue in the face, and got a lot of agreement, but without more concrete patterns nobody was signing up.

Meanwhile, [ErichGamma](#) was busy writing and reflecting about object-oriented design in ET++ as part of his PhD thesis. Erich had realized that recurring design structures or patterns were important. The question really was how do you capture and communicate them.

[BruceAnderson](#) gave a talk at TOOLS 90 at which [ErichGamma](#) was present; Erich liked the talk. Bruce gave a paper at EcoopOopsla90 (Ottawa) and ran a BOF called *Toward an Architecture Handbook* where he, [ErichGamma](#), [RichardHelm](#), and others got into discussions about patterns. That was the first time that Richard and Erich met, and they realized they shared common ideas about the key ideas behind writing reusable OO software.

Just prior to ECOOP'91 [ErichGamma](#) and [RichardHelm](#), sitting on a rooftop in Zurich on a sweltering summer's day, put together the very humble beginnings of the catalog of patterns that would eventually become [DesignPatterns](#). There they identified many patterns including such familiar and unfamiliar patterns as

- Composite



Ward Cunningham



Kent Beck

Using Pattern Languages for Object-Oriented Programs

Kent Beck, Apple Computer, Inc.

Ward Cunningham, Tektronix, Inc.

Technical Report No. CR-87-43

September 17, 1987

Submitted to the OOPSLA-87 workshop on the
Specification and Design for Object-Oriented Programming.

Abstract

We outline our adaptation of Pattern Language to object-oriented programming. We summarize a system of five patterns we have successfully used for designing window-based user interfaces and present in slightly more detail a single pattern drawn from our current effort to record a complete pattern language for object-oriented programs.

The search for an appropriate methodology for object-oriented programming has seen the usual rehash of tired old ideas, but the fact is that OOP is so different that no mere force-fit of structured analysis or entity-relationship methods will provide access to the potential inherent in OOP. In particular, neither of these methods address the user interface design issues that have obviously become of paramount importance. In addition, while E-R seems to be "object-oriented" it is not suited to the dynamic nature of objects as in Smalltalk and encourages the use of a global perspective while designing, a sure lose in object-oriented programming.

We propose a radical shift in the burden of design and implementation, using concepts adapted from the work of Christopher Alexander, an architect and founder of the Center for Environmental Structures. Alexander proposes homes and offices be designed and built by their eventual occupants. These people, he reasons, know best their requirements for a particular structure. We agree, and make the same argument for computer programs. Computer users should write their own programs. The idea sounds foolish when one considers the size and complexity of both buildings and programs, and the years of training for the design professions. Yet Alexander offers a convincing scenario. It revolves around a concept called a "pattern language."

A pattern language guides a designer by providing workable solutions to all of the problems known to arise in the course of design. It is a sequence of bits of knowledge written in a style and arranged in an order which leads a designer to ask (and answer) the right questions at the right time. Alexander encodes these bits of knowledge in written patterns, each sharing the same structure. Each has a statement of a problem, a summary of circumstances creating the problem and, most important, a solution that works in those circumstances. A pattern language collects the patterns for a complete structure, a residential building for example, or an interactive computer program. Within a pattern language, patterns connect to other patterns where decisions made in one influence the others. A written pattern includes these connections as prologue and epilogue. Alexander has shown that nontrivial languages can be organized without cycles in their influence and that this allows the design process to proceed without any need for reversing prior decisions [Alex77].



Erich Gamma

Elementos de un patrón

Cada patrón es una regla de tres partes, que expresa una relación entre un contexto determinado, un problema y una solución.

—Christopher Alexander (1979)

Contexto

Describe las situaciones en las que se da el problema.

Problema

El problema recurrente que ocurre en dicho contexto.

Comienza con una descripción general que capture su esencia: ¿cuál es la cuestión concreta de diseño que debemos resolver?

El patrón Modelo/Vista/Controlador, por ejemplo, aborda el problema del cambio en las interfaces de usuario.

Problema

El problema recurrente que ocurre en dicho contexto.

Ese enunciado general se completa con un conjunto de fuerzas implicadas.

Tomado originalmente de la arquitectura y de Christopher Alexander, la comunidad de patrones utiliza el término fuerza para denotar cualquier aspecto del problema que deba tenerse en cuenta a la hora de resolverlo, como por ejemplo:

Requisitos que debe cumplir la solución

Restricciones a considerar

Propiedades deseables

Así, el Modelo/Vista/Controlador especifica dos fuerzas:

Debería ser fácil cambiar la interfaz de usuario.

Sin que el núcleo principal con la funcionalidad del programa se vea afectado por dicho cambio.

En general, las fuerzas discuten el problema desde varios puntos de vista y ayudan a comprender sus detalles. Las fuerzas pueden complementarse o contradecirse.

Solución

Cómo resolver dicho problema recurrente.

Es decir, cómo equilibrar ese conjunto de fuerzas.

En el diseño de software, dicha solución incluye dos aspectos:

La estructura estática del patrón

El comportamiento en tiempo de ejecución

La estructura describe los aspectos estáticos del patrón

Las clases participantes y sus relaciones.

Sirven como bloques de construcción, cada uno con una responsabilidad definida.

El comportamiento, los aspectos dinámicos

¿Cómo colaboran los participantes?

¿Cómo se reparten el trabajo?

¿Cómo se comunican entre ellos?

**Los patrones
son un
esquema**

Un patrón software proporciona un esquema general

No es un artefacto implementado, un prototipo para ser usado tal cual.

Es un bloque de construcción mental para ser implementado en un sinfín de aplicaciones y contextos diferentes.

No habrá dos implementaciones idénticas

Después de aplicar el patrón, se identificará su estructura y los papeles (roles) desempeñados por las clases participantes, pero éstas estarán ajustadas a las necesidades de cada problema concreto.

Categorías de patrones

Atendiendo a su nivel de abstracción

Arquitectónicos

De diseño

De implementación (modismos o «idioms»)

Atendiendo a su nivel de abstracción

Arquitectónicos

De diseño

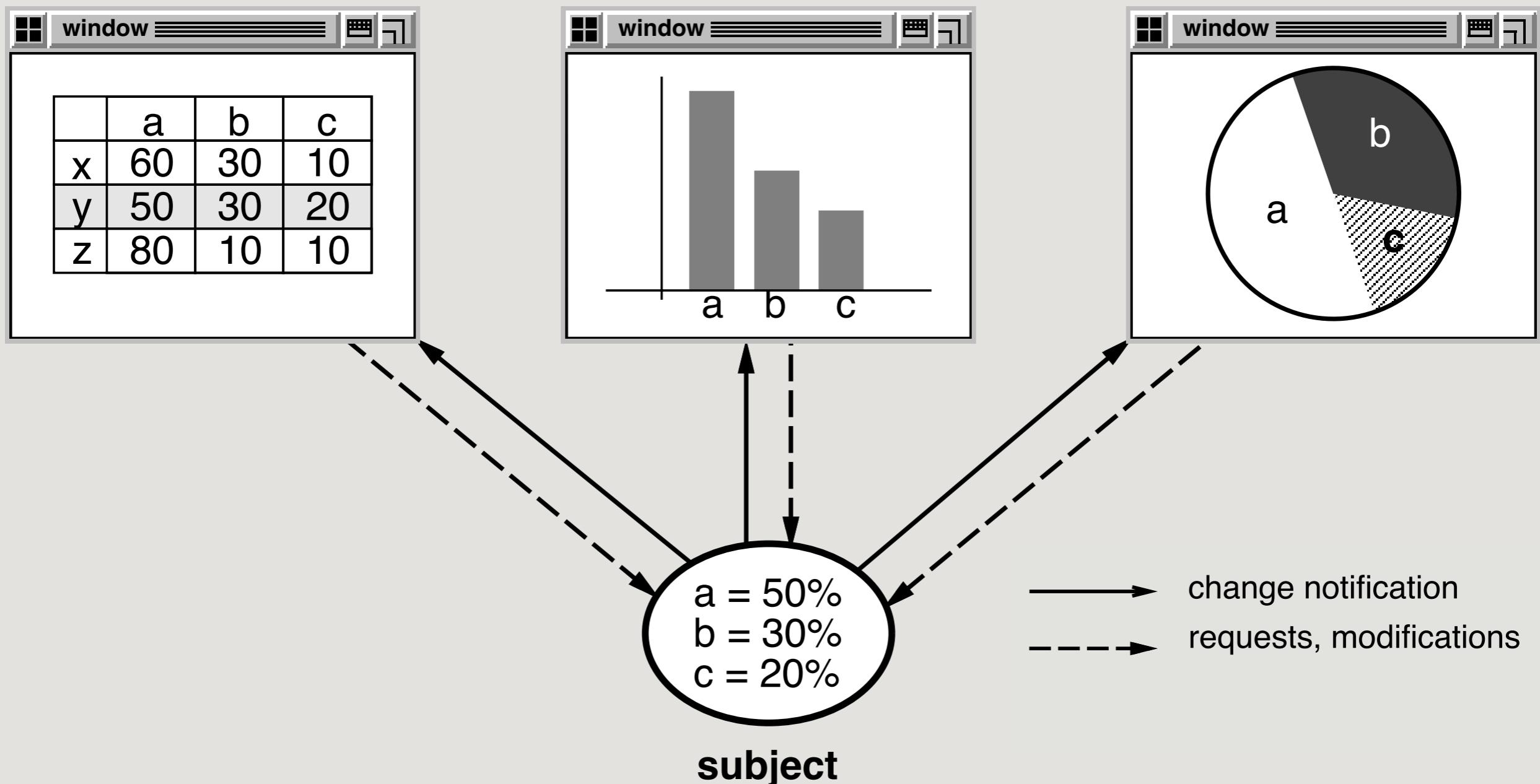
De implementación (modismos o «idioms»)

*Situados a un nivel de abstracción
más alto, describen la
arquitectura, la estructura de un
sistema en torno a subsistemas y
las relaciones entre ellos*

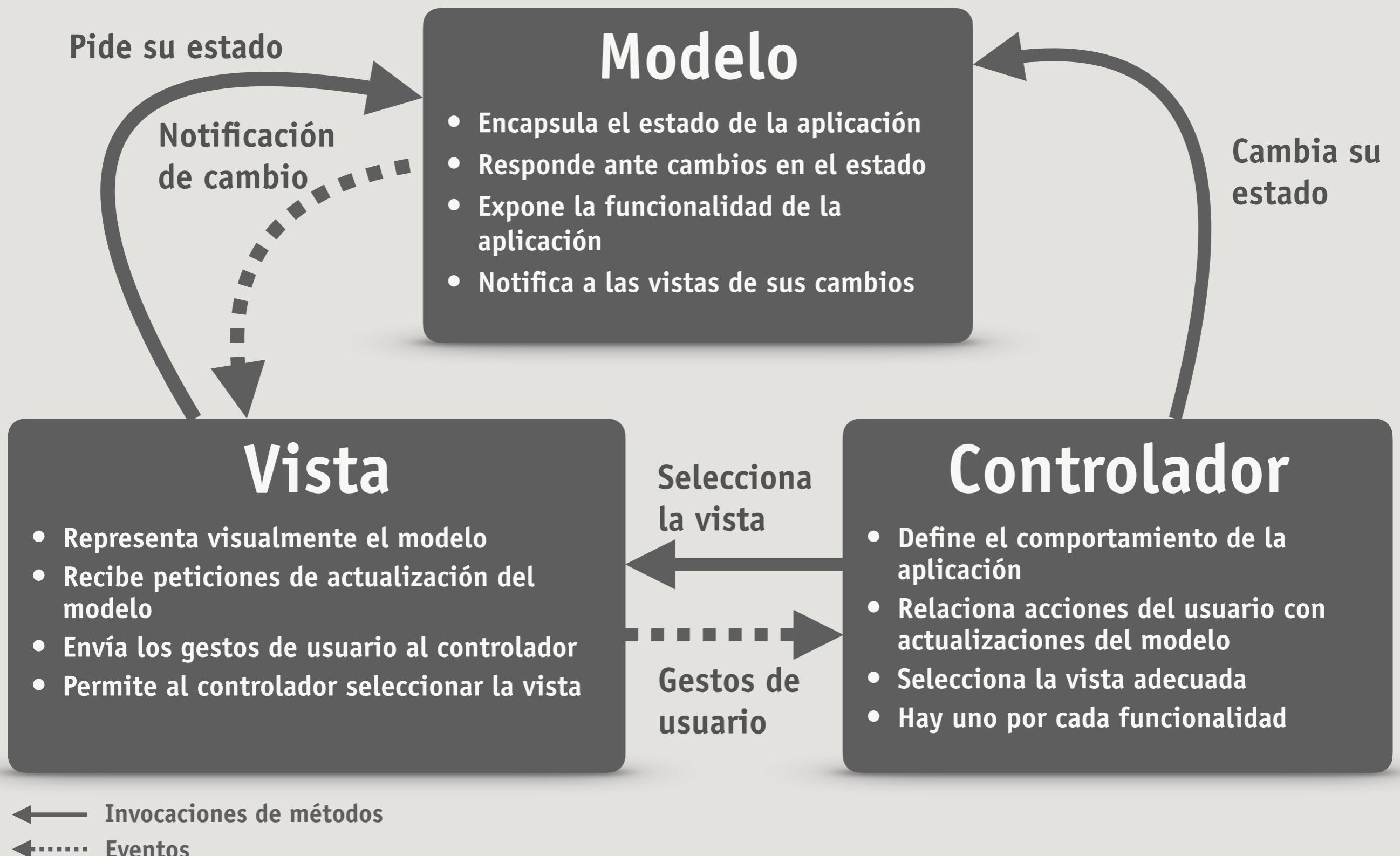
Ejemplo

Model/View/Controller (MVC)

observers



La tríada MVC



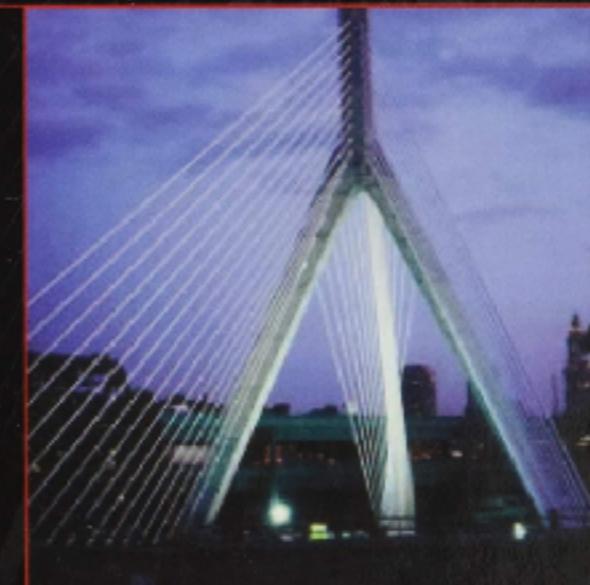
The Addison-Wesley Signature Series

PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE

MARTIN FOWLER

WITH CONTRIBUTIONS BY

DAVID RICE,
MATTHEW FOEMMEL,
EDWARD HIEATT,
ROBERT MEE, AND
RANDY STAFFORD



MARTIN FOWLER
SIGNATURE
BOOK



core
J2EE™ PATTERNS
Best Practices and Design Strategies
Second Edition



DEEPAK ALUR
JOHN CRUPI
DAN MALKS

Prentice Hall PTR, Upper Saddle River, NJ 07458
www.phptr.com

Sun Microsystems Press
A Prentice Hall Title

Atendiendo a su nivel de abstracción

Arquitectónicos

De diseño

De implementación (modismos o «idioms»)

Se sitúan a un nivel de abstracción medio, de diseño.

Suelen consistir en unas pocas clases e interfaces, sus responsabilidades y las relaciones y colaboraciones entre ellas.

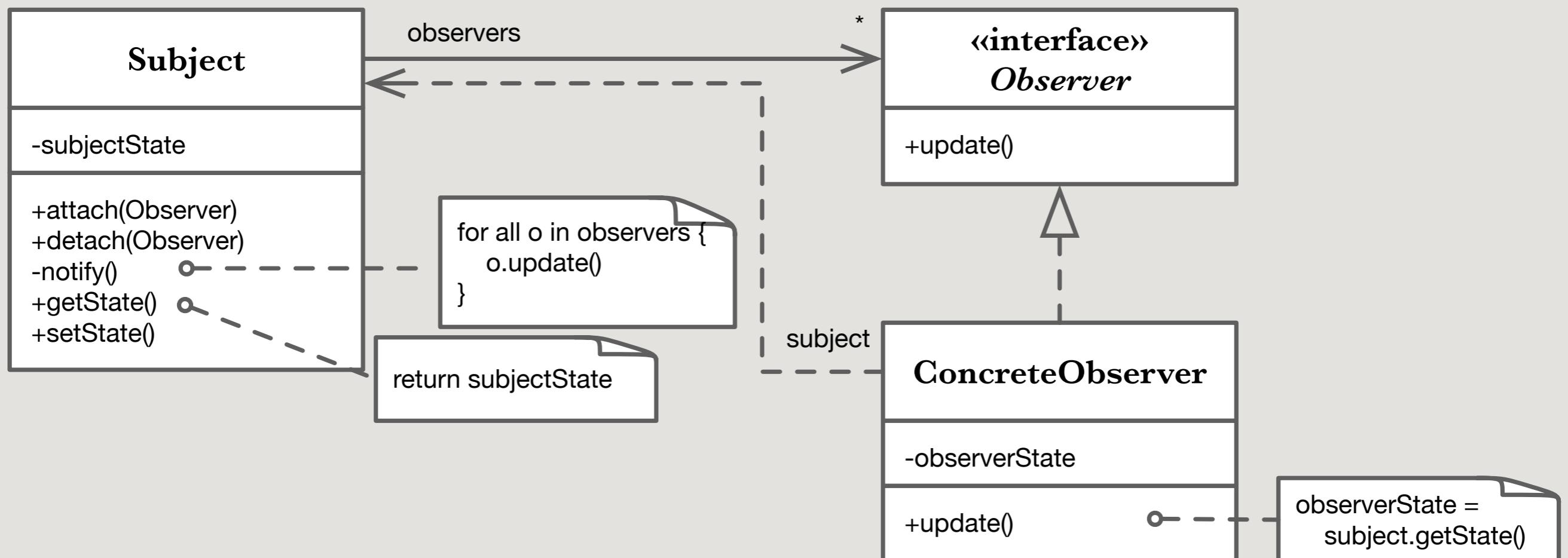
De menor escala que los patrones arquitectónicos, pero tienden a ser independientes de un lenguaje de programación concreto.

Proporcionan un esquema para refinar los subsistemas o componentes de un sistema de software o las relaciones entre ellos.

Describen una estructura recurrente de componentes comunicados que resuelve un problema general de diseño dentro de un contexto particular.

Ejemplo

Observer



Atendiendo a su nivel de abstracción

Arquitectónicos

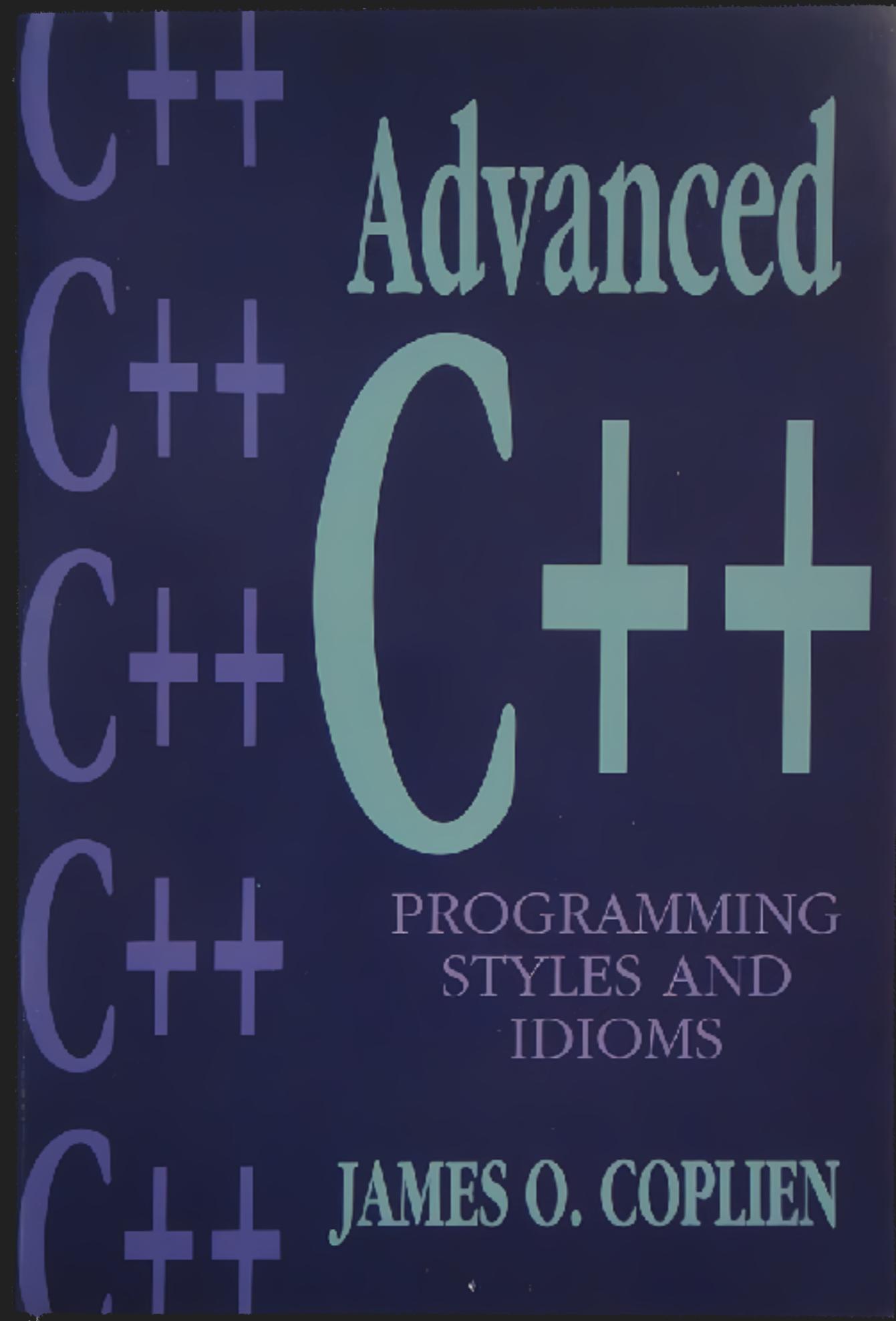
De diseño

De implementación (modismos o «idioms»)

Son específicos de un lenguaje de programación. Un modismo describe cómo implementar aspectos concretos de los componentes o las relaciones entre ellos utilizando las características del lenguaje en cuestión.

Contribuyen a dar un estilo uniforme, consistente, a los programas escritos en ese lenguaje

Al final, su conocimiento es lo que distingue a los buenos programadores en un determinado lenguaje de aquéllos que simplemente conocen su sintaxis.



Propiedades de los patrones

Un patrón responde a un problema de diseño que se repite frecuentemente en determinadas situaciones, y presenta una solución a dicho problema.

Los patrones sirven para documentar la experiencia previa, los diseños que ya se probaron útiles.

¡No se inventan ni se crean artificialmente!

Al contrario, destilan el conocimiento de diseño adquirido por los profesionales experimentados y proporcionan un medio para representar dicho conocimiento y facilitar su reutilización.

En lugar de que el conocimiento exista sólo en las cabezas de unos pocos expertos, los patrones hacen que esté disponible de forma más general.

Deben haberse utilizado en campos de aplicación muy diferentes (para ser útil, un patrón de diseño debe ser aplicable a más de unos pocos dominios de problemas).

Identifican y especifican abstracciones que están por encima de clases y objetos o componentes individuales.

Típicamente, un patrón describe varios componentes, clases u objetos, así como sus responsabilidades y relaciones, y el modo en que cooperan.

Todos los participantes resuelven juntos el problema que aborda el patrón.

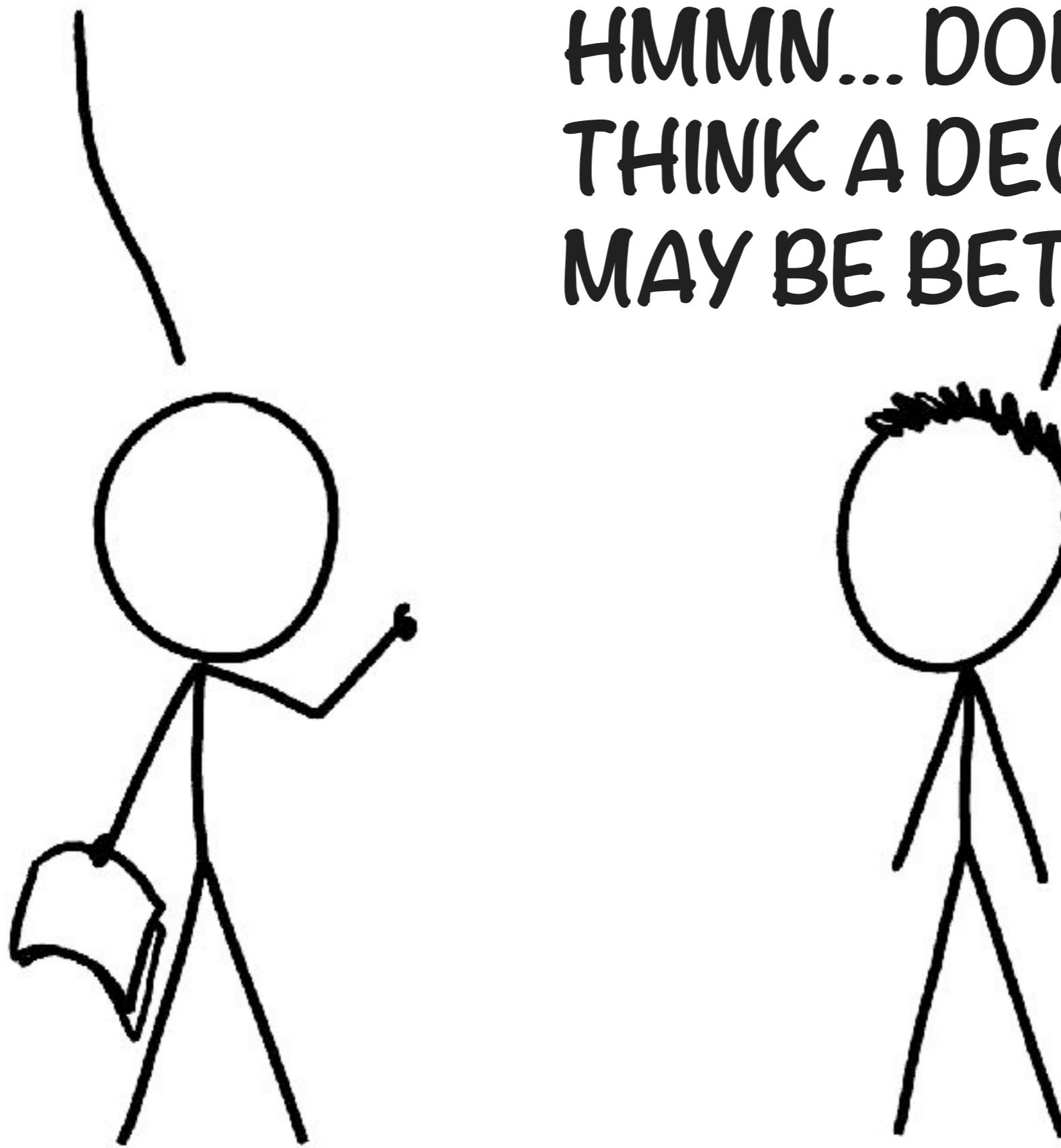
*Proporcionan un vocabulario de
diseño común.*

Los nombres de los patrones, si se eligen con cuidado, pasan a formar parte de nuestro vocabulario de diseño.

Facilitando un debate eficaz sobre los problemas de diseño y sus soluciones.

Eliminan la necesidad de explicar una solución a un problema concreto con una descripción larga y complicada. En su lugar, podemos utilizar un nombre de patrón y explicar qué partes de una solución corresponden a qué componentes del patrón o a qué relaciones entre ellos.

WE SHOULD USE A STRATEGY HERE.



HMMN... DON'T YOU
THINK A DECORATOR
MAY BE BETTER?

Tal vez sea su principal
aportación.

*Permiten documentar nuestros
diseños.*

Ayudan a describir la visión que teníamos en mente cuando diseñamos el software.

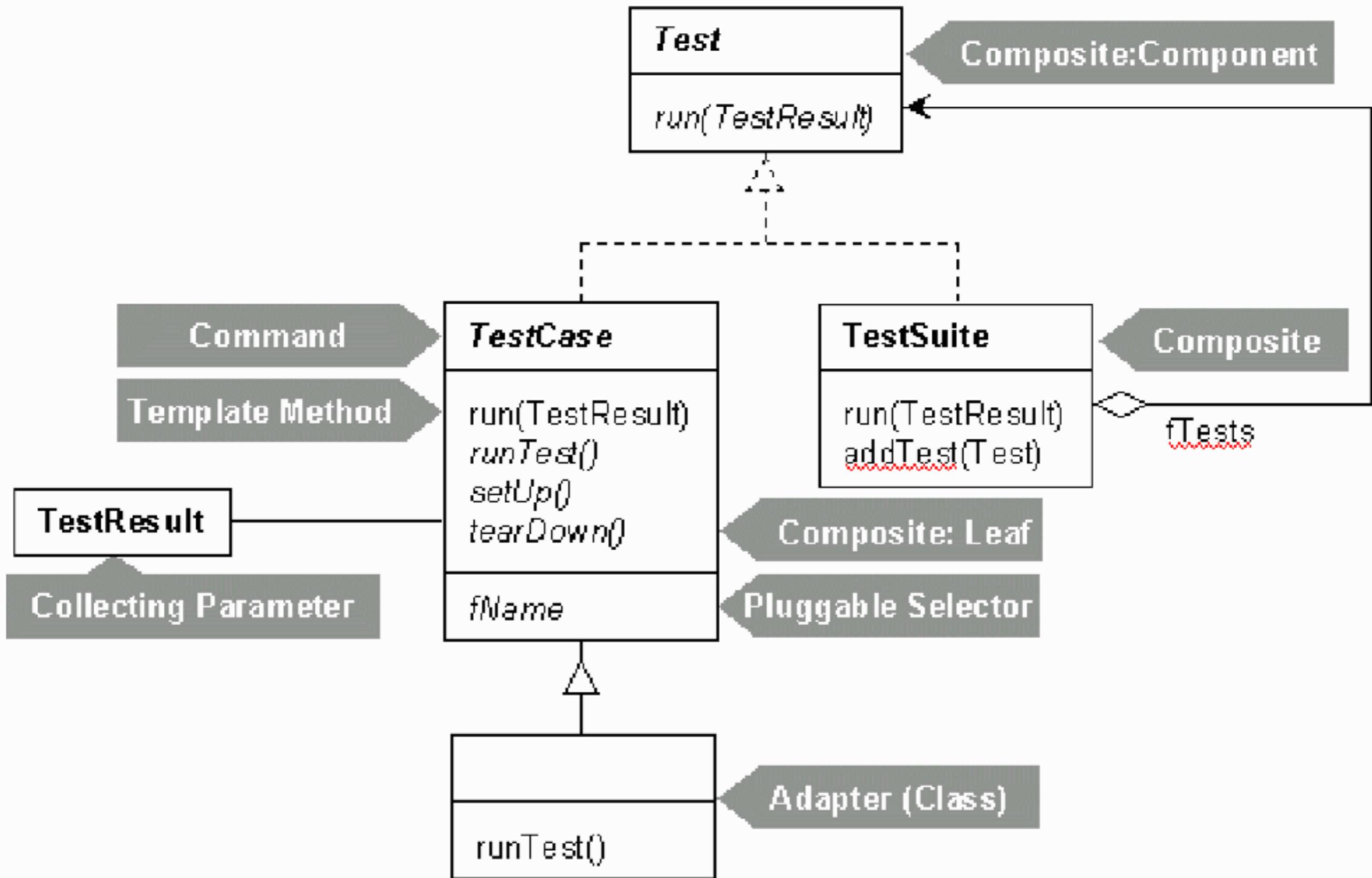
Disminuyendo así el riesgo de que sea malinterpretado y se viole esa visión en un futuro, al modificar el diseño, o por parte de los programadores al implementarlo.

JUnit: A Cook's Tour

Un buenísimo ejemplo de cómo documentar un diseño.

Eric Gamma y Kent Beck no sólo describen el diseño final en términos de los patrones de diseño implicados, sino que explican cada una de las razones y motivaciones que subyacen a las decisiones de diseño tomadas.

Este nivel de detalle hace que el documento deje de ser meramente técnico para ofrecer una visión más profunda, ayudando a los lectores a entender no sólo qué decisiones se tomaron, sino también por qué, un aspecto crucial para fomentar la comunicación y la colaboración en el diseño de software.



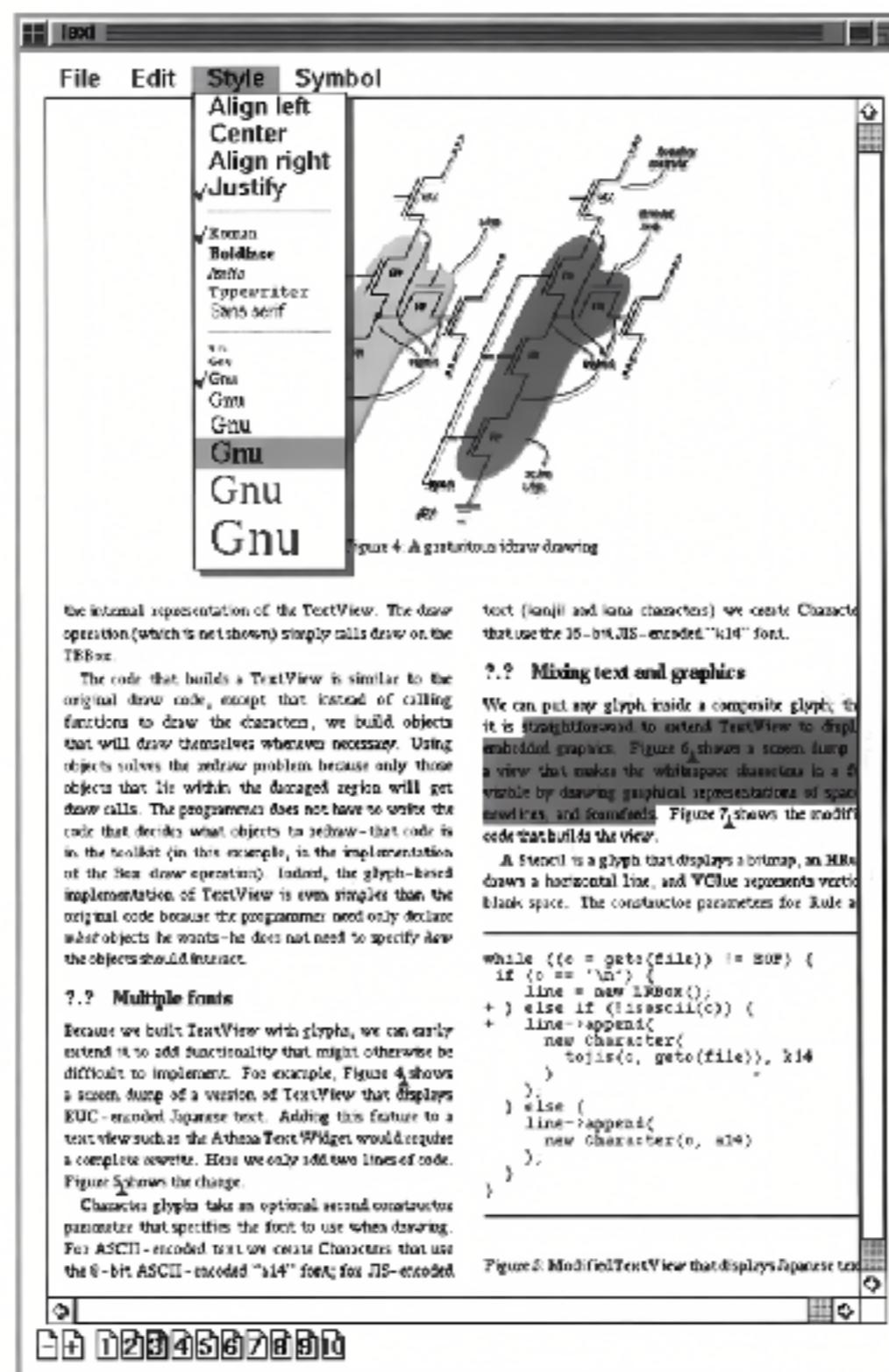


Figure 2.1: Lexi's user interface

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBBx.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw—that code is in the toolkit (in this example, in the implementation of the New draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare what objects he wants—he does not need to specify how the objects should interact.

7.2 Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded “>14” font; for JIS-encoded

text (kanji and kana characters) we create Characters that use the 16-bit JIS-encoded “>14” font.

7.2 Mixing text and graphics

We can put any glyph inside a composite glyph; this is straightforward to extend TextView to display embedded graphics. Figure 6 shows a screen dump of a view that makes the whitespace characters in a TextView draw graphical representations of space, newlines, and linefeeds. Figure 7 shows the modified code that builds the view.

A Stencil is a glyph that displays a bitmap; an HBar draws a horizontal line, and VClue represents vertical blank space. The constructor parameters for Rule are:

```
while ((c = gets(file)) != EOF) {
    if (c == '\n') {
        line = new LBox();
    } else if (!isascii(c)) {
        line->append(
            new Character(
                tojis(c, gets(file)), >14
            )
        );
    } else {
        line->append(
            new Character(c, >14)
        );
    }
}
```

Figure 5: Modified TextView that displays Japanese text

Otro excelente ejemplo sería el capítulo 2 del GoF, donde se explica el diseño de un procesador de textos, Lexi, en términos de patrones.

*Permiten construir software con
propiedades definidas.*

Los patrones proporcionan un esqueleto de comportamiento funcional y, por tanto, ayudan a implementar la funcionalidad de nuestra aplicación.

Además, los patrones abordan explícitamente los requisitos no funcionales de los sistemas informáticos, como que sean modificables, la fiabilidad, que sean comprobables o la reutilización.

*Ayudan a construir arquitecturas
complejas y heterogéneas.*

Los patrones actúan como bloques de construcción para construir diseños más complejos.

Este método de utilizar artefactos de diseño predefinidos favorece la rapidez y la calidad de nuestro diseño.

Comprender y aplicar los patrones de diseño ahorra tiempo en comparación con tener que buscar la solución por nuestra cuenta.

Esto no quiere decir que los patrones individuales vayan a ser necesariamente mejores que sus propias soluciones, pero, como mínimo, un sistema de patrones de este tipo puede ayudarnos a evaluar y valorar alternativas de diseño.

Los patrones ayudan a resolver problemas, pero no proporcionan soluciones completas.

Aunque un patrón determina la estructura básica de la solución a un problema de diseño concreto, no especifica una solución totalmente detallada.

Un patrón proporciona un esquema para una solución genérica a una familia de problemas, en lugar de un módulo prefabricado que puede utilizarse «tal cual». Debemos aplicar este esquema en función de las necesidades específicas del problema de diseño en cuestión.

*Los patrones ayudan a gestionar
la complejidad del software.*

Catálogos de patrones

Si aceptamos que los patrones pueden resultar útiles en el desarrollo de software, el siguiente paso es reunirlos en catálogos de forma que resulten accesibles mediante distintos criterios, pues lo que necesitamos no es tan solo la completa descripción de cada uno de los patrones sino, esencialmente, la correspondencia entre un problema real y un patrón (o conjunto de patrones) determinado.



La Curva de aprendizaje

Lo que se pretende con un catálogo de patrones no es favorecer al diseñador experto (que quizás no necesite en absoluto los patrones), sino más bien ayudar al diseñador novel a adquirir con cierta rapidez las habilidades de aquél, como también comunicar las decisiones de diseño de forma clara.



Un catálogo de patrones es un medio para comunicar la experiencia de forma efectiva, reduciendo lo que se conoce como «curva de aprendizaje» del diseño.



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 WLC Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

* ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Sobre el GoF

De Gang of Four o Banda de los cuatro, en alusión a sus autores.

Y tomado, a su vez, de la «revolución cultural» china.

Libro clásico por excelencia de patrones de diseño.

Puede ser un poco duro, al principio, pero es un libro que cualquier programador o diseñador de software debería leer.

¡Y que debería existir en cualquier empresa!

Catálogo de 23 patrones divididos en tres categorías:
de creación, estructurales y de comportamiento

Secciones de un patrón

Las notaciones gráficas no son suficientes.

Los patrones de diseño proporcionan un esquema documental consistente que se repite en todos ellos.

Cada patrón se divide en las siguientes secciones.

Nombre y clasificación

El nombre del patrón transmite suavemente su esencia. Un buen nombre es vital, porque formará parte de nuestro vocabulario de diseño.

Nombre y clasificación

La clasificación se refiere al alcance del patrón: si se aplica principalmente a clases o a objetos.

Intención

Una o dos frases que responden a las siguientes preguntas: ¿Qué hace el patrón de diseño? ¿Cuál es su justificación y propósito? ¿Qué cuestión o problema de diseño concreto aborda?

También conocido como

Otros nombres por los que se conoce al
patrón, si los hay.

Motivación

Un escenario que ilustra un problema de diseño y cómo las estructuras de clases y objetos del patrón resuelven el problema.

Ayudará a entender la descripción más abstracta del patrón que sigue.

Aplicabilidad

¿En qué situaciones puede aplicarse el patrón de diseño?

¿Qué ejemplos de diseños deficientes puede resolver el patrón?

¿Cómo reconocer estas situaciones?

Estructura

Un diagrama de clases que representa las clases e interfaces del patrón.

A veces también se utilizan diagramas de secuencia para ilustrar secuencias de peticiones y colaboraciones entre objetos.

Participantes

Las clases, interfaces u objetos que participan en el patrón de diseño, y sus responsabilidades.

Colaboraciones

Cómo colaboran los participantes para llevar a cabo sus responsabilidades.

Consecuencias

¿Qué consigue el patrón?

¿Cuáles son las ventajas e inconvenientes de aplicarlo?

¿Qué aspectos del sistema permite que varíen de forma independiente?

Implementación

Detalles de bajo nivel a tener en cuenta a la hora de implementar el patrón (trucos, técnicas, cuestiones específicas de tal o cual lenguaje...).

Código de ejemplo

Fragmentos de código (en C++ o Smalltalk) que ilustran cómo podría implementarse el patrón.

Usos conocidos

Ejemplos de uso del patrón en sistemas reales. Se incluyen al menos dos ejemplos de ámbitos diferentes.

Patrones relacionados

¿Qué patrones de diseño están estrechamente relacionados con este?

¿Cuáles son las diferencias importantes?

¿Con qué otros patrones suele aparecer?

Organización del catálogo

Según su
propósito

El primer criterio, según su propósito, refleja qué hace el patrón.

Pueden ser de creación, estructurales o de comportamiento.

De creación

Se refieren al proceso de creación de objetos.

Estructurales

Tratan con la composición de clases u objetos.

De comportamiento

Caracterizan cómo las clases u objetos interactúan y distribuyen su responsabilidad.

Según su ámbito

El segundo criterio, denominado **ámbito**,
especifica si el patrón se aplica
fundamentalmente a clases u objetos.

De clases

Tratan sobre todo con relaciones entre clases*
y sus subclases a través de la herencia.

O sea, estáticas, fijadas estáticamente en tiempo de compilación.

* En nuestro caso, también entre interfaces y las clases que las implementan.

De objetos

Tratan con relaciones entre objetos, que pueden cambiarse en tiempo de ejecución (dinámicas).

Aunque casi todos los patrones utilizan la herencia en cierta medida, los etiquetados como «de clases» se centran específicamente en las relaciones entre clases.

La mayor parte de los patrones entran en esta categoría.

Los patrones de creación de clases difieren alguna parte de la creación de objetos a las subclases, mientras que los patrones de creación de objetos la delegan en otro objeto.

Los patrones de clases estructurales utilizan la herencia para componer clases, mientras que los patrones de objetos estructurales describen formas de ensamblar objetos.

Los patrones de clases de comportamiento utilizan la herencia para describir algoritmos y flujos de control, mientras que los patrones de objetos de comportamiento describen cómo un grupo de objetos coopera para realizar una tarea que ningún objeto puede realizar por sí solo.

No son la única forma de organizar los patrones.

Algunos patrones se utilizan a menudo juntos. Por ejemplo, Composite se utiliza a menudo con Iterator o Visitor.

Algunos patrones son alternativos: Prototype es a menudo una alternativa a Factory Method.

Algunos patrones dan lugar a diseños similares aunque tengan intenciones diferentes. Por ejemplo, los diagramas de estructura de Composite y Decorator son parecidos.

Ámbito

Clases
Objetos

Propósito
De creación Estructurales De comport.

Factory Method Adapter

Abstract Factory
Builder
Prototype
Singleton

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Interpreter
Template Method

Chain of
Responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor

Ámbito

Clases
Objetos

Propósito
De creación Estructurales De comport.

Factory Method Adapter

Abstract Factory
Builder
Prototype
Singleton

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Interpreter
Template Method

Chain of
Responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor

Diseñar para el cambio

(Causas de rediseño)

La clave para maximizar la reutilización reside en anticiparse a las nuevas necesidades y a los cambios en las existentes, y en diseñar los sistemas de modo que puedan evolucionar en consecuencia.

Para diseñar el sistema de modo que sea robusto ante tales cambios, debemos considerar cómo podría tener que cambiar el sistema a lo largo de su vida útil. Un diseño que no tenga en cuenta el cambio corre el riesgo de tener que ser rediseñado en el futuro.

Estos cambios pueden implicar la redefinición y reimplementación de clases, la modificación del cliente y la repetición de pruebas. El rediseño afecta a muchas partes del sistema de software, y los cambios imprevistos son siempre caros.

Los patrones de diseño ayudan a evitar esto asegurando que un sistema pueda cambiar de formas establecidas.

Cada patrón de diseño permite que algún aspecto de la estructura del sistema varíe independientemente de otros aspectos, lo que hace que un sistema sea más robusto ante un tipo concreto de cambio.

Veamos cuáles son algunas de esas causas comunes de rediseño (junto con los patrones de diseño que las abordan).

*Crear un objeto especificando
explícitamente su clase.*

Especificar un nombre de clase al crear un objeto le compromete a una implementación concreta en lugar de a una interfaz concreta. Este compromiso puede complicar futuros cambios. Para evitarlo, crea objetos indirectamente.

Patrones de diseño:

Abstract Factory, Factory Method, Prototype

*Depender de operaciones
concretas.*

Cuando especifica una operación concreta,
se compromete a una forma de satisfacer
una solicitud.

Al evitar tener dichas llamadas concretas en el
código, se facilita el cambio de la forma en que se
satisface una petición, tanto en tiempo de
compilación como en tiempo de ejecución.

Design Patterns:

Command, Chain of Responsibility

*Dependencia de plataformas
hardware o software.*

El software que depende de una biblioteca o plataforma concreta será más difícil de portar a otras plataformas. Incluso puede ser difícil mantenerlo actualizado en su plataforma nativa.

Por tanto, es importante diseñar el sistema para limitar sus dependencias de la plataforma.

Design Patterns:

Abstract Factory, Adapter, Bridge

*Depender de implementaciones o
representaciones de objetos.*

Los clientes que conocen cómo se representa internamente, cómo se almacena, dónde se localiza o cómo se implementa un objeto tendrán que cambiar cuando cambie aquél.

Es preciso ocultar esta información a los clientes para prevenir los cambios en cascada.

Design Patterns:

Abstract Factory, Bridge, Memento, Proxy

Dependencias de algoritmos.

Los algoritmos es probable que cambien a lo largo del tiempo (para optimizarlos o porque se sustituyan por otro distinto).

Por ese motivo, los algoritmos que es probable que cambie deben estar aislados.

Design Patterns:

Strategy, Template Method, Visitor, Iterator, Builder

Fuerte acoplamiento.

Las clases estrechamente acopladas son difíciles de reutilizar de forma aislada, ya que dependen unas de otras.

El acoplamiento estrecho conduce a sistemas monolíticos, donde no se puede cambiar o eliminar una clase sin entender y cambiar muchas otras clases. El sistema se convierte en una masa amorfa difícil de aprender, portar y mantener.

Los patrones de diseño promueven el acoplamiento abstracto.

Design Patterns:

Abstract Factory, Command, Observer, Bridge, Chain of Responsibility, Facade, Mediator

*Extender funcionalidad
mediante la herencia.*

Aunque es fácil de entender y usar, la herencia de clases presenta no pocos inconvenientes.

En primer lugar, se define en tiempo de compilación, lo que hace imposible cambiar la implementación heredada de las clases padre en tiempo de ejecución.

Lo que es peor, las clases padre suelen definir al menos parte de la representación física de sus subclases.

Dado que la herencia expone una subclase a detalles de la implementación de su padre, a menudo se dice que «la herencia rompe la encapsulación».

La composición de objetos se define dinámicamente en tiempo de ejecución mediante objetos que adquieren referencias a otros objetos.

Como sólo se accede a los objetos a través de sus **interfaces**, no se rompe la encapsulación.

Cualquier objeto puede ser sustituido **en tiempo de ejecución** por otro siempre que tenga el mismo tipo.

Además, como la implementación de un objeto se escribirá en términos de **interfaces** de objetos, hay muchas **menos dependencias de implementación**.

Favorecer la composición de objetos frente a la herencia de clases ayuda a mantener cada clase encapsulada y centrada en una tarea.

Las clases y jerarquías de clases seguirán siendo pequeñas y será menos probable que crezcan hasta convertirse en monstruos inmanejables.

Por otro lado, un diseño basado en la composición de objetos tendrá más objetos, y el comportamiento del sistema dependerá de sus interrelaciones en lugar de estar definido en una sola clase.

*Favorecer la
composición de
objetos sobre la
herencia de clases.*

*Programar para una
interfaz, no para una
implementación.*

La composición de objetos en general y la delegación en particular ofrecen alternativas flexibles a la herencia para combinar comportamientos.

Por otro lado, un diseño basado en la composición de objetos tendrá más objetos, y el comportamiento del sistema dependerá de sus interrelaciones en lugar de estar definido en una sola clase.

Muchos patrones de diseño producen diseños en los que se puede introducir funcionalidad personalizada simplemente definiendo una subclase y componiendo sus instancias con otras ya existentes.

Design Patterns:

Composite, Decorator, Observer, Strategy, Bridge,
Chain of Responsibility

No se puede modificar las clases.

Algunas veces necesitamos cambiar una clase y no se puede.

O bien porque no es posible al no disponer del código fuente (por ejemplo, con una biblioteca de clases de terceros).

O porque requeriría cambiar montones de subclases existentes.

Hay patrones que nos permiten modificar clases bajo tales circunstancias.

Design Patterns:

Adapter, Decorator, Visitor

Cómo seleccionar un patrón de diseño

Pensemos cómo los patrones resuelven los problemas de diseño.

Cómo nos ayudan a encontrar objetos apropiados, determinar la granularidad de los objetos, especificar interfaces de objetos y otras cuestiones en las que los patrones resultan útiles.

Es decir, tengamos en cuenta todo lo aprendido hasta ahora, qué es lo que queremos conseguir en ese contexto concreto y cómo aplicar los principios clásicos y las guías y directrices que hemos dado en la asignatura.

Examinar sus intenciones.

Lee la intención de cada patrón para encontrar uno o varios que parezcan relevantes para el problema.

Estudiar cómo los patrones se
relacionan entre sí.

Estudiar patrones con propósitos parecidos

Cada capítulo del libro comienza con comentarios introductorios sobre los patrones y concluye con una sección en la que se comparan y contrastan.

Estas secciones ofrecen una visión de las similitudes y diferencias entre patrones de la misma categoría.

Tener claras las causas de rediseño.

Y mirar si nuestro problema tiene que ver con alguna de ellas.

A continuación, miremos los patrones que ayudan a evitar dicha causa.

Considerar qué es lo más probable que cambie en nuestro diseño.

El enfoque contrario a centrarse en las causas del rediseño. En lugar de considerar qué podría obligar a cambiar un diseño, hay que considerar qué queremos poder cambiar sin rediseñar.

Nos centraremos en otro principio clásico del D00 y en el que se basan muchos de los patrones.

*Encapsular y aislar
el concepto que varía.*

Cómo usar un patrón de diseño

Hacer una primera lectura
rápida del patrón.

Prestando especial atención a las secciones de
Aplicabilidad y Consecuencias para asegurarnos
de que el patrón es adecuado para el problema a
resolver.

A continuación, estudiar la estructura, participantes y colaboraciones.

Asegurándonos de entender las clases y objetos del patrón y cómo colaboran entre sí.

Echar un vistazo al código de ejemplo.

Nos puede ayudar a entender cómo implementar el patrón.

Aunque en vuestro caso tenéis algo probablemente mejor: el código de la práctica o prácticas de laboratorio donde se aplicó dicho patrón.

Elegir los nombres de participantes adecuados en el contexto de nuestra aplicación.

Adaptando los del patrón, que son demasiado abstractos para ser usados directamente.

Aunque a veces es útil incorporar el nombre del participante al nombre del dominio. Eso ayuda a que el patrón sea más explícito en la aplicación.

De todas formas, siempre que sea posible, preferiremos nombres que procedan del dominio, si las abstracciones representadas por los participantes en el patrón coinciden con algún concepto existente.

Definir las clases.

Declarar sus interfaces, establecer sus relaciones de herencia y definir las variables de instancia que representan datos y referencias a objetos.

Identificar las clases existentes a las que afectará el patrón y modificarlas en consecuencia.

Definir nombres específicos del dominio para las operaciones del patrón.

También en este caso, los nombres suelen depender de la aplicación.

Utilizaremos como guía las responsabilidades y colaboraciones asociadas a cada operación, siendo además coherentes en los convenios de nomenclatura.

Implementar las operaciones para llevar a cabo las responsabilidades y colaboraciones en el patrón.

La sección de implementación ofrece consejos que nos pueden guiar en la implementación.

El código de ejemplo también pueden ser de ayuda.