Una Introducción a los Patrones de Software

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM.
jloa@fciencias.unam.mx

Resumen

Los *Patrones de Software* son una forma de literatura desarrollada para comunicar conocimiento experto acerca del diseño y construcción de sistemas de software. Los más útiles son aquéllos que responden a los problemas estructurales y aquéllos que han sido cuidadosamente descritos para ser entendibles.

Los patrones pueden ayudar a una organización a capturar sus principales aptitudes, conservar la memoria de diseño dentro de la organización y proveer de un formato para compartir ideas que muy frecuentemente se mantienen solo por algunas personas dentro de la propia organización. La literatura de patrones provee de una base en la cual estas ideas pueden efectivamente reunirse y compartirse. Esta introducción tiene como meta principal dar a conocer los objetivos y valores desarrollados durante varios años de discusión y reflexión dentro de la Comunidad de Patrones.

Índice

1. Patrones de Software – Una Introducción

El interés sobre Patrones de Software ha dado lugar a una de las comunidades con mayor crecimiento en el área de diseño de software. Los Patrones de Software han llamado la atención de muchos desarrolladores y diseñadores de la industria del software del mismo modo que anteriormente muchas otras técnicas emergentes, como la programación estructurada, la abstracción de datos y la programación Orientada a Objetos, fueron consideradas "clave" para el desarrollo de software. Sin embargo, la mayoría de estas técnicas fallaron (de un modo u otro) en cumplir con las expectativas que generaron en un principio. Las razones han sido variadas: en parte porque eran ideas expuestas al mercado prematuramente, en parte debido al oportunismo de algunas empresas, y en parte por la espera (o, mas bien, desesperación) del mercado, que busca ansiosamente una solución única a todos los problemas de desarrollo de software. Esta receta de decepción y desinformación impulsa a la vez nuevas espectativas. Así, un ciclo vicioso continúa una vez mas.

Ante esta situación, durante ya algunos años el grupo de *Hillside* [1] ha desarrollado una visión en el diseño de software que en sí misma da acomodo y celebra la variedad de visiones individuales. El diálogo y la discusión se han centrado en buena medida alrededor de *qué es* y *qué no es* un patrón, y respecto a *qué es* y *qué no es* un *buen* patrón. Tal labor no pretende limitar el diálogo y la discusión, sino contribuir a la difusión del concepto. La Comunidad de Patrones valora la diversidad: hay muchas cosas que pueden verse como patrones, y que comparten elementos eseciales de técnicas, principios y valores, que han comprobado ser útiles para los programadores. De hecho, hay varias comunidades de patrones, algunas muy cercanas y otras muy alejadas entre sí. Aquí se intenta capturar las prácticas y las bases y la (por cierto, no muy bien definida) línea principal de la cultura de patrones.

Este escrito es tanto una disculpa como una defensa sobre Patrones de Software. Cuando se discute sobre patrones en público, es necesario establecer espectativas razonables sobre los mismos. La mayoría de las técnicas nuevas están avaladas por expectativas que muestran lo que tales técnicas pueden proveer, y el peligro para los Patrones de Software, como se menciona anteriormente, es que pueden caer en la trampa de generar falsas expectati-

[1] *Hillside* es un grupo que aglomera a diseñadores de software activos en la búsqueda de patrones como soluciones a problemas de la industria del software. Su página Web se halla en `http://hillside.net/`

vas. En general, los Patrones de Software se refieren a cuestiones críticas y centrales en el diseño y desarrollo de software, lo que les da su importancia natural. Sin embargo, pueden considerarse sólo como una documentación más, que depende de la perspicacia de quienes los crean y los usan. Es importante considerarlos como un elemento más de la caja de herramientas del diseñador de software. Su éxito depende de las personas, y particularmente, de los aspectos más humanos del diseño y desarrollo de software y su cultura. Los Patrones de Software se proponen a fin de acentuar el valor de las personas en el proceso de diseño.

A pesar de que este escrito es la labor de un solo autor, al mismo tiempo reflejan el trabajo de docenas de individuos. Toma directamente las contribuciones de quienes "escriben sobre patrones" (*pattern writers*) y quienes "piensan sobre patrones" (*pattern thinkers*), lo que hace que el material vaya más allá de la esfera de pensamiento del autor. De hecho, las siguientes páginas contienen tal vez más ideas y conceptos de varios colegas que de los propios; el autor simplemente provee del ordenamiento y cohesión entre ellos.

Esto es consistente con otra meta importante de la Comunidad de Patrones: se prefiere leer, usar y escribir patrones en vez de hablar o discutir acerca de patrones. La abstracción prematura es peligrosa; fácilmente distrae de los objetivos que se aspiran. Este documento es un intento inicial de abstraer los principios de la Comunidad de Patrones y explorar fronteras de sus sistema de valores. Sin embargo, está fuertemente basado en la realidad.

1.1. Una Nota sobre Alexander

Los Patrones de Software tienen su origen en la arquitectura y diseño de edificios, particularmente en el trabajo del arquitecto británico Christopher Alexander. La mayoría de las publicaciones sobre patrones le hacer referencia. Alexander ha introducido varios términos en el vocabulario de Patrones de Software: *fuerzas*, el propio término *patrón* y el concepto de *lenguaje de patrones*.

Sin embargo, la mayor parte de la Comunidad de Patrones se ha alejado de las interpretaciones literales del trabajo de Alexander dentro del desarrollo de software. Por ejemplo, el desarrollo de software parece realizarse iterativamente, mientras que Alexander propone un desarrollo en orden monolítico y progresivo. Todas las analogías fallan en algún punto. Por tanto, mientras algunos de lor primeros pioneros en el desarrollo de software con

patrones buscaban exhaustivamente analogías con el trabajo de Alexander, la literatura de Patrones de Software parece alejarse y tomar un rumbo propio fuera de las estructuras y formas de Alexander. La experiencia con patrones se ha desarrollado abarcando otras fuentes, que incluyen algunos trabajos iniciales en computación y programación.

El gran legado de Alexander a la Comunidad de Patrones es su visión y sistema de valores, pero su visión es tan extraña para la mayoría de la práctica en desarrollo de software que comúnmente se pierde en la dimensión técnica de los patrones. Un objetivo de este documento es subrayar el sistema de valores, a fin de ser más ampliamente entendido y apreciado.

1.2. Organización del documento

Este documento se organiza alrededor de preguntas centrales sobre patrones, intentando acercarse cada vez más a cuestiones más refinadas y avanzadas al final. Las secciones de este documento son:

- *¿Qué es un Patrón?* Esta sección presenta las definiciones más recurrentes de patrón, tanto históricamente como aquéllas usadas dentro de los Patrones de Software. Intenta proveer una explicación de los patrones y su importancia.
- *¿Qué son Lenguajes de Patrones?* A partir de lo explicado en la sección anterior, esta sección define las colecciones de patrones que se conocen con el nombre genérico de Lenguajes de Patrones, que representan la aplicación más importante de patrones al diseño de sistemas.
- *Dominios de Patrones.* Esta sección provee algunos ejemplos de desarrollo de software que actualmente aplican patrones.
- *Clasificando Patrones.* Habiendo cientos de patrones disponibles ¿cómo encontrar aquél que el diseñador busca? Esta sección investiga los principios organizacionales que actualmente se utilizan para Patrones de Software.
- *Pragmatismo en Patrones.* Esta sección resume los beneficios de negocio de los patrones, y cómo introducir patrones dentro de una cultura de desarrollo.
- *Generatividad.* Generatividad es lo que distingue a los patrones de las reglas. Se trata de un componente sutil pero importante de la cultura de patrones.

- *El Sistema de Valores de Patrones.* Dentro de la Comunidad de Patrones ha surgido un sistema de valores que intenta restaurar la dignidad del programador como persona. En esta sección se presentan varias cuestiones legales y éticas al respecto.

- *Historia.* En esta sección se presenta cómo los patrones se han desarrollado desde los primeros trabajos de Alexander hasta las labores actuales dentro de la Arquitectura de Software; es un resumen de las personas y los eventos a la fecha.

2. ¿Qué es un Patrón?

Un patrón es una pieza de literatura que describe un problema de diseño y una solución general para tal problema en un contexto particular. Alexander lo define de la siguiente forma [4]:

> *Cada patrón es una regla de tres partes, la cual expresa la relación entre un cierto contexto, un problema y una solución.*
>
> El patrón es, en resumen, al mismo tiempo una cosa, que sucede en el mundo, y una regla que nos dice cómo crear esa cosa, y cuándo debemos crearla. Es tanto un proceso como una cosa; es tanto una descripción de una cosa que está viva como una descripción del proceso el cual genera esa cosa.

Puede relacionarse esta definición con patrones de costura. De tal modo, es posible decir cómo hacer una prenda de vestir mediante especificar el camino de las tijeras através de la tela en términos de ańgulos y longitudes de corte. O podría utilizarse una forma determinada previamente, es decir, un patrón. Usando esta descripción, es muy posible que no se sepa qué se está construyendo, o si se ha construido lo correcto cuando se ha finalizado. Sin embargo, en cierto modo el patrón predice el producto: es la regla para hacerlo, pero también, y en muchos aspectos, es el producto mismo.

Otro ejemplo, ya en el área de diseño de software, es el documento que describe un arreglo particular de desacoplamiento en programación Orientada a Objectos, conocido como el patrón *Bridge* [23]; de hecho, a cada instancia de tal arreglo en el programa se le llama también un *Bridge*.

Alexander también menciona que [4]:

> Estos patrones en nuestras mentes son, más o menos, imágenes mentales de patrones en el mundo: son representaciones abstractas de las mismas reglas morfológicas que definen a los patrones en el mundo.
>
> Sin embargo, en un aspecto son diferentes. Los patrones en el mundo meramente existen. Pero los mismo patrones en nuestras mentes son dinámicos. Tienen fuerza. Son generativos. Nos dicen qué hacer; nos dicen como debemos, o podemos, generarlos; y también nos dicen que bajo ciertas circunstancias, debemos crearlos.
>
> Cada patrón es una regla la cual describe qué se debe hacer para generar la entidad que define.

Anteriormente, se cita que Alexander se refiere a los patrones como una regla de tres partes. Sin embargo, los patrones son más que sólo la regla: comprenden la literatura que los describe y que ayuda a desarrollarlos para su uso.

Ward Cunningham añade la siguiente metáfora, describiendo que los patrones son más como una receta que un plan:

> Me agrada hacer una distinción entre lo que es un plan y una receta. Un plan puede obtenerse mediante Ingeniería Inversa a partir de la construcción, pero una recta no puede (fácilmente) obtenerse por Ingeniería Inversa a partir de un pastel. Nuestro genoma es una receta, no un plan. Las recetas parecen servir mejor como un esquema de sistemas adaptables complejos.

Observamos patrones en estructuras cotidianas: en edificios, en tráfico vehicular, en organizaciones sociales, y en nuestro software. Alexander extrajo sus patrones de edificios y estructuras de poblaciones de numerosas culturas. Creía que mediante documentar estos patrones, podría ayudar a las personas a conformar las construcciones de su comunidad para vivir plenamente. Los patrones son estructuras que han evolucionado através de los años para cubrir necesidades culturales. Son reglas, dirigidas por principios, que sirven a necesidades humanas y sociales. Los patrones son también la documentación de esas estructuras. Por lo tanto, un patrón es tanto una cosa que sucede en el mundo, como la regla para crear tal cosa.

Los patrones han evolucionado hacia un pequeño número de formas literarias. En algún sentido, un patrón es solo documentación. Por otro lado, un patrón es solo documentación tal y como la poesía es solo litearatura.

Además, en la Comunidad de Patrones se considera que la comunicación humana es el "cuello de botella" en el desarrollo de software. Si la forma literaria de los patrones puede ayudar a los programadores a comunicarse con sus clientes, sus usuarios, y entre sí, entonces a la vez ayudan a llenar una necesidad crucial del desarrollo de software actual.

Los Patrones de Software no son un método de diseño completo; capturan prácticas importantes que ocurren durante el desarrollo de software por métodos convencionales, pero que normalmente no están documentadas por tales métodos convencionales. No son una herramienta CASE; se enfocan más en las actividades humanas de diseño que en las transformaciones que ciegamente pueden ser automatizadas. No son Inteligencia Artificial; celebran y alientan la inteligencia humana, que separa a las personas de las computadoras.

Un ejemplo de patrón, obtenido del libro *A Pattern Language* de Alexander [3] se presenta a continuación:

> *A Place to Wait*
>
> *El proceso de esperar tiene conflictos inherentes en sí*
>
> Por un lado, en cualquier momento en que las personas se encuentran esperando a un médico, un avión, una cita de negocios, etc., tiene incertidumbres implícitas, que hacen inevitable que las personas deban pasar perdiendo el tiempo, esperando y no haciendo nada.
>
> Por otro lado, normalmente las personas no pueden pasar disfrutando este tiempo. Debido a que es impredecible, deben permanecer pendientes de una llamada o una puerta. Esto es porque no se sabe exactamente cuándo será su turno, por lo que no pueden salir a caminar o sentarse afuera...
>
> Por lo tanto:
>
> *En los lugares donde las personas terminan esperando, se debe crear una situación que haga una espera positiva. Conjunte la espera con alguna otra actividad como leer el periódico, tomar café, jugar billar; algo que haga pensar a las personas que no están simplemente esperando. Y también lo opuesto: hágase un lugar en que una persona esperando entre en un retiro, en calma, en silencio positivo.*

También hay patrones en el software que escribimos. Los patrones son pares *problema-solución* dentro del diseño. Los más importantes capturan

estructuras, prácticas y técnicas como elementos clave dentro de un campo dado, pero que no han sido aún ámpliamente conocidos.

2.1. La Relación de los Patrones con...

Los Patrones de Software son parecidos a muchos otros formalismos de diseño, pero se distinguen de los formalismos más comunes de maneras sutiles. Panu Viljamaa lo explica de la siguiente forma [36]:

> Los patrones están relacionados con, pero son diferentes a: paradigmas, expresiones idiomáticas, principios, heurísticas, arquitecturas, *frameworks* y modelos.
>
> Podría decirse que un paradigma es un patrón muy abstracto, o un estilo de trabajar que puede seguirse consistentemente durante el desarrollo de un sistema. Una expresión idiomática es una manera, típica y específica de un lenguaje, de usar y combinar bloques elementales de construcción. Un principio es una invariante que se mantiene en forma global, o siempre; puede ser un sinónimo de regla de diseño. Una heurística ayuda en la toma de decisiones, sin afirmar una bondad absoluta de las acciones sugeridas. Las heurísticas pueden utilizarse para la selección entre múltiples patrones. La arquitectura se refiere a la estructura total de una aplicación, posiblemente descrita por los muchos patrones involucrados. Los patrones han sido llamados micro-arquitecturas. Los *frameworks* se refieren a colecciones de clases concretas trabajando juntas para lograr una tarea parametrizable dada. Los modelos describen una colaboración única y coordinada entre múltiples participantes (las clases de un *framework* pueden servir en múltiples modelos simultáneamente). Los modelos podrían ser la cosa más cercana a la formalización de patrones.

2.2. ¿Porqué necesitamos capturar patrones?

La mayoría concuerda en que los patrones son una disciplina que intenta capturar información de diseño que es importante y empírica. Sin embargo, los patrones tienen un énfasis diferente que el reuso de programas o los catálogos de diseño: los patrones tienden a capturar abstracciones más amplias. En términos de Alexander, los patrones se conforman como lapsos en la memoria de la cultura contemporanea de diseño de software, y capturan la estructura no inmediatamente aparente del código o de la mayoría de los documentos de diseño.

- **Los patrones capturan prácticas poco claras pero importantes.** Los patrones capturan prácticas establecidas que permanecen po-

co claras en la amplia práctica dentro de un dominio dado. La intuitividad de los patrones resulta paradójica. Muchos patrones tienen sus orígenes en el trabajo de quienes adaptan inicialmente una nueva tecnología o quienes son los primeros arquitectos de un sistema. Muchos de estos patrones atacan problemas de manera sutil, lo que los hace difíciles de expresar como *frameworks* de las contrucciones predominantes en términos de la tecnología del sistema. Por ejemplo, en C++, el conteo de referencias sería un patrón, mientras que las características específicas del lenguaje usadas para su implementación son tan solo eso, características del lenguaje, independientes del conteo de referencias *per se*. El mismo Bjarne Stroustrup previó la necesidad de contar referencias en C++. Muchas características como lenguaje de C++, tales como los constructores, destructores (y que sirven a otros fines también) y la sobrecarga de la asignación anticipan esta necesidad. Pero cuando las personas aprenden por primera vez C++, lo hace en términos de las partes del lenguaje (clases, procedimientos y objetos) o en términos de principios de diseño Orientado a Objetos, que guían hacia una partición del problema en clases, pero que hacen poco notoria la necesidad del conteo de referencias.

Los patrones trabajan a muchos niveles de detalle. Se tiende a pensar en el conteo de referencias como un detalle, pero aquellos programadores que no saben cómo implementarlo apropiadamente nunca llegarán a ser buenos programadores en C++. Aun cuando el conteo de referencias es una construcción de bajo nivel, se pueden utilizar patrones para capturar sus principios clave de diseño en forma abstracta. Un patrón es abstracto porque es una aproximación a la solución de un problema a un nivel general, aun cuando la solución misma pueda requerir detalles. Una buena solución tiene el suficiente detalle, de manera que el diseñador sabe qué hacer, pero a la vez, es suficientemente general para utilizarse en un contexto más amplio.

- **Los patrones capturan la estructura oculta.** Los patrones se extienden a lo largo de las particiones predominantes de un sistema. Muchos patrones de Alexander mencionan la relación entre la calle y las casas (por ejemplo, "Transición de Entrada"), entre casa y casa (por ejemplo, "Línea de Casas"), entre cuarto y cuarto (por ejemplo, "Alturas de Techo Variadas"), o entre los cuartos y sus interfaces ("Luz en Ambos Lados de Cada Cuarto"). Rara vez se enfocan en las propiedades de un solo artefacto arquitectónico. Los Patrones de Software son

de la misma forma: resuelven problemas de sistema y relaciones que se obscurecen por una perspectiva del interior de cualquiera de las partes. Las técnicas iniciales de diseño Orientado a Objetos magnificaban esta vista miope de los sistemas, y los métodos actuales parecen no hacerlo mejor. Los Patrones de Software complementan los métodos de diseño Orientado a Objetos para capturar componentes importantes que ocurren más allá de los objetos. Estos componentes de la arquitectura y diseño son más grandes que cualquier bloque de construcción como procedimientos y objetos. Tales componentes de construcción no son claros para el desarrollador debido a que los "materiales de construcción" (objetos, módulos y procedimientos) no los manifiestan. Algunos de estos patrones son sumamente intrincados y quizá muy detallados, como el conteo de referencias. Sin embargo, estos patrones permean la estructura del sistema de software que ellos mismos ayudan a crear. El arquitecto se ocupa de colocar tales estructuras en sitios para satisfacer una necesidad, pero tales estructuras y el razonamiento de su creación y aplicación se pierden fácilmente en la historia. Los patrones capturan este tipo de estructuras y decisiones.

Citando de nuevo a Alexander [4]:

> *El diseño se piensa frecuentemente como un proceso de síntesis, un proceso de colocar cosas juntas, un proceso de combinación.*
>
> De acuerdo con esta visión, un todo es creado mediante colocar partes juntas. Las partes vienen primero: y la forma del todo viene después.
>
> *Pero es imposible formar nada que tenga el carácter de natural mediante añadir partes pre-formadas.*
>
> Cuando las partes son modulares y hechas antes que el todo, por definición entonces, son idénticas, y es imposible que sean únicas, de acuerdo con su posición en el todo.

Más tarde, en el mismo libro, Alexander comenta que [4]:

> Para que el edificio esté vivo, sus detalles de construcción deben ser únicos y adecuados a sus circunstancias individuales tan cuidadosamente como las partes más grandes. Los detalles de un edificio no pueden ser hechos vivos cuando se hacen de partes modulares.

El Patrón de Software *Mediator* se refiere a la relación entre dos objetos, que frecuentemente atraviesa la partición intuitiva de un sistema en objetos [23]:

Nombre
Mediator

Intención
Define un objeto que encapsula cómo un conjunto de objetos interactúan. Mediator promueve un acoplamiento débil mediante no permitir que los objetos se refieran entre sí explícitamente, permitiendo al programador variar su interacción en forma independiente.

Motivación
El diseño Orientado a Objetos alienta la distribución del comportamiento entre objetos. Tal distribución puede resultar en una estructura de objetos con muchas relaciones entre objetos; en el peor de los casos, cada objeto termina reconociendo a todos los demás.

Aun cuando particionar un sistema en muchos objetos generalmente mejora la reusabilidad, la proliferación de interrelaciones tiende a reducirla de nuevo. Muchas interrelaciones hacen menos probable que un objeto pueda trabajar sin el apoyo de los otros, y el sistema actúa como si fuera monolítico. Más aun, puede ser difícil hacer cambios en el comportamiento del sistema en forma significativa, ya que el comportamiento se encuentra distribuido entre muchos objetos. Como resultado, el programador se ve forzado a definir sub-clases para adecuar el comportamiento del sistema.

Se puede evitar estos problemas mediante encapsular el comportamiento colectivo en un objeto **mediador** por separado. Un mediador es responsable de controlar y coordinar las interacciones de un grupo de objetos. El mediador sirve como un intermediario que previene a los objetos en el grupo de referirse entre sí explícitamente. Cada objeto sólo conoce al mediador, lo que reduce el número de interrelaciones.

Consecuencias

1. Limita la generación de sub-clases
2. Desacopla objetos cooperativos
3. Simplifica los protocolos entre objetos
4. Abstrae la cooperación entre objetos
5. Centraliza el control

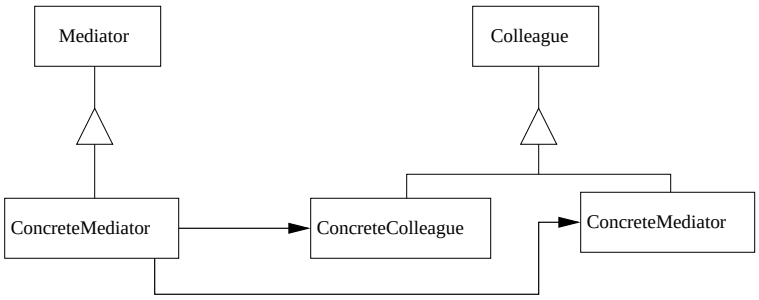


Figura 1: Un diagrama de clases para el Patrón *Mediator*.

2.3. Las Formas de los Patrones

Los patrones se consideran formas literarias, de la misma manera en que los sonetos, las novelas o los cuentos son formas literarias. La forma sirve a un propósito: introducir al lector a un problema, describir el contexto cuando el problema puede aparecer, analizar el problema, y presentar y dilucidar una solución.

Muchos documentos pueden clasificarse como patrones bajo esta descripción, y la cuestión de la forma se encuentra íntimamente relacionada con la definición de patrón. La definición prevaleciente: una solución a un problema en un contexto evoca elementos de la forma. Hay una gran variedad de formas de patrones establecidas. Las siguientes secciones son un resumen de las formas de uso común hasta ahora.

2.3.1. Las secciones

Se puede escribir un buen patrón mediante asegurarse de que los objetivos de cada sección se logren. Escribir un gran patrón es, sin embargo, mucho más holístico que eso. El patrón debe funcionar como una sola pieza de literatura, pero las secciones y sus atributos merecen la investigación igualmente del estudiante de patrones y del crítico de patrones.

En esta sección se exponen las ideas básicas de varias formas de uso común para describir patrones. Combina los antecedentes y algunas ayudas para entender, y en su caso desarrollar, cada una de las secciones de un patrón. Estas secciones también permiten colocar a los componentes del patrón en perspectiva. La solución es obviamente el corazón del patrón. Sin embargo, una representación gráfica (un dibujo) y la descripción de fuerzas

son también importantes.

Una lista común (y tal vez mínima) de secciones para describir un patrón es la siguiente:

- Nombre
- Resumen
- Contexto
- Problema
- Fuerzas
- Solución
- Representación gráfica
- Consecuencias

Nombre

Los nombres son importantes en muchas culturas, y siempre han aparecido de manera relevante en programación. Todo buen diseñador de software Orientado a Objetos entiende la importancia de escoger nombres adecuados para sus clases. La selección de los nombres de los patrones también tiene una importancia, por al menos dos razones.

Primero, los nombres de los patrones son una de las entidades que inicialmente encuentra un diseñador de software cuando busca una solución. Si el nombre del patrón representa el significado del propio patrón de una manera clara y adecuada, el diseñador tiene mayor oportunidad de hallar un patrón que se ajuste a sus necesidades, aún en un lenguaje de patrones o catálogo de patrones poco familiar. Por ejemplo, nombres como *Ambassador* o *Remote Proxy* evocan imágenes de cómo interactúan sus componentes (en general, objetos).

Segundo, los nombres de los patrones tienden rápidamente a formar parte del vocabulario de un equipo de diseño. Nombres cortos, bien escogidos, son útiles para la comunicación entre desarrolladores. Por ejemplo, el patrón *Riding Over Transients* [1] es un patrón general que trata con eventos transitorios: si el diseñador busca una solución a errores transitorios, el nombre

puede llamar la atención del diseñador, alentando una lectura más profunda. Otros nombres descriptivos de patrones son *Half-Object plus Protocol*, *Window Per Task* y *Exceptional Value*.

Muchos patrones toman nombres interesantes evocando analogías obscuras o cuestiones relevantes e históricas. Se pueden hallar nombres de patrones y nombres de lenguajes de patrones como *Leaky Bucket Counter*, *Fool Me Once* y *Caterpillar's Fate*, que conllevan significados profundos cuando se explican. Tal vez debido a que estos nombres tienen lazos culturales muy fuertes, son utilizados como facilitadores de la comunicación. Sin embargo, estos nombres resultan una desventaja para el diseñador novato que no está familiarizado con tales elementos culturales. Los novatos normalmente buscan nombres que describan algún elementos o cualidad del problema o la solución.

El uso de uno o varios alias puede ayudar a resolver este problema. *Leaky Bucket Counter* puede tener un alias de *Riding Over Even Transients*: el primer nombre evoca una poderosa analogía para aquellos que conocen el patrón, y el segundo nombre sirve para la búsqueda de soluciones por parte del novato.

¿Debería el nombre de un patrón ser un sustantivo, un verbo, una frase sustantiva o una frase verbal? ¿debería caracterizar el problema o la solución? Se pueden encontrar patrones que sigue ambas aproximaciones. Por ejemplo, los patrones de Alexander son predominantemente sustantivos y frases sustantivas, con algunas excepciones, pero con variaciones que funcionan adecuadamente. Por ejemplo:

- *A Place to Wait* [3] describe el contexto para la solución.
- *Capacity Bottlenecks* [28] describe el problema.
- *Bridge* [23] describe la solución.
- *Chicken and Egg* [5] describe el problema.
- *Gatekeeper* [16] describe un componente clave de la solución.

Por otro lado, algunas frases verbales son las siguientes:

- *Fool Me Once* y *Minimize Human Intervention* [1] describen la solución.

- *Developing in Pairs* [16] describe la solución.
- *Reception Welcomes You* [3] es un nombre de un patrón de Alexander que utiliza un verbo; describe la solución.
- *Identify the Nouns* [20] describe el problema.

Resumen

El resumen es una frase o enunciado que brevemente describe lo que el patrón hace, normalmente describiendo la cuestión o problema de diseño que resuelve. Conforme el diseñador revisa los patrones en búsqueda de una solución para un problema específico, el resumen provee de información sobre patrones prometedores. Frecuentemente no es parte del cuerpo del patrón, pero puede ser usado para facilitar al diseñador encontrar un patrón que satisfaga sus necesidades.

Algunos patrones no cuentan con un resumen, pero incorporan una sección llamada Intención (*Intent*), la cual hace las veces del resumen. Fue usado por primera vez en la forma GoF (que se describe más adelante), y tiene una analogía directa con la pregunta que aparece al principio de muchas formas de patrones (por ejemplo, las preguntas que aparecen al principio de los patrones de Kent Beck en la forma Portland, que también se describe más adelante).

Contexto

El contexto se refiere a la situación cuando el problema ocurre, y se hace explícito en la descripción del patrón. Incluye, por ejemplo, una historia de patrones que han sido aplicados antes que el patrón actual fuera considerado. Especifica también elementos de la situación como tamaño, entorno, mercado, lenguaje de programación, o cualquier cosa que, si se cambia, invalidaría al patrón.

El contexto de los patrones es crucial para el éxito de un lenguaje de patrones, que es una colección de patrones que trabajan juntos para resolver problemas a nivel sistema (los lenguajes de patrones se describen más a detalle en la Sección 3). Se puede pensar en un patrón como el balanceo entre fuerzas para un problema en un contexto, resultando en un nuevo

contexto. De tal forma, los contextos entretejen a los patrones en la forma de un lenguaje de patrones.

Es difícil escribir (o describir al menos) un contexto. La sección de contexto va madurando con la experiencia: conforme los diseñadores van encontrando situaciones especiales que invalidan al patrón, el contexto se extiende, lo que lo hace más restrictivo. En ocasiones, se encuentran oportunidades para extender a los patrones en nuevos contextos, capturando tal experiencia en la sección de contexto.

Conforme los patrones evolucionan, quien escribe el patrón debe ser cuidadoso para mantener una sección de contexto clara. El vocabulario de todo dominio evoluciona con un mayor entendimiento, y se puede hacer al contexto una sección con descripciones cada vez más precisas.

Problema

La sección del problema describe lo que se desea resolver. Un enunciado del problema conciso ayuda al diseñador a decidir si se continúa leyendo, y frecuentemente sirve como índice primario para la selección del patrón.

Algunas formas reducen el problema a una sola pregunta o a una formulación breve del problema; muchos de los patrones en la forma Coplien (que se explica en la siguiente sección) presentan esta característica. En otras formas, el enunciado del problema es un pequeño ensayo que motiva o ilustra la necesidad. Por ejemplo, la forma Alexander combina las secciones del problema y fuerzas.

Fuerzas

Los patrones no son reglas a seguir ciegamente; se deben entender y adecuar a las necesidades de cada problema. Es posible entender las necesidades de cada problema si se entienden las fuerzas (los "pros" y "contras") presentes, a la vez que se entiende la solución como el balance de tales fuerzas. Partiendo de esto, se puede intuir mucho de la lógica detrás del patrón. Por tal razón, las fuerzas son esencialmente el punto focal del patrón. Alexander, en sus obras iniciales, enfatiza que las fuerzas son cruciales para entender los sistemas [2]:

> Nadie se vuelve un mejor diseñador mediante... seguir cualquier método ciegamente; si se trata de entender la idea que se pueden crear patrones

> abstractos mediante estudiar la implicación del sistema limitado de fuerzas, y se puede crear nuevas formas por la libre combinación de estos patrones y entender que esto sólo funcionará si los patrones que se definen tratan con fuerzas del sistema cuya interacción interna es muy densa, y cuya interacción con otras fuerzas en el mundo es muy débil, entonces en el proceso de intentar crear tales patrones uno mismo, se llega a la idea central de la cual se trata este libro.

Desde una perspectiva pragmática, las fuerzas ayudan al diseñador a entender cómo aplicar un patrón efectivamente. Un solo patrón puede ser aplicado un millón de veces sin hacer exactamente la misma cosa dos veces. La clave en tal distinción recae en las fuerzas.

El propio término *fuerzas* denota la herencia del área de arquitectura que tienen los patrones. Un arquitecto diseña arcos y muros para balancear las fuerzas de gravedad con las fuerzas de las uniones de una estructura, de tal modo que la propia estructura está balanceada y centrada. Las fuerzas en balance soportan un sistema firme y estructuralmente completo. En software, se utiliza el término de fuerza de manera figurativa, ya que no hay fuerzas físicas que se deban balancear. Aun el propio Alexander usa el término fuerza de manera figurativa, particularmente cuando se ocupa de balancear las fuerzas de la estética y comodidad humanas con la estructura física de una población o un edificio. Los grandes arquitectos pueden balancear todas estas fuerzas, es decir, no optimizan la parte estética a costa de la factibilidad, utilidad o costo en la implementación, una cuestión práctica que parece perderse en muchas escuelas de arquitectura moderna. Conforme los patrones maduran, toman en cuenta además de las fuerzas mecánicas, las fuerzas humanas.

La descripción de las fuerzas debe amplificar e ilustrar el enunciado del problema, ya que es sólo a través de las fuerzas que se puede apreciar el problema. En la forma Alexander, las fuerzas son propiamente parte del enunciado del problema, y no una sección por separado. Otras formas separan las fuerzas para aclarar al lector la exposición de los “pros” y “contras”.

Las fuerzas determinan porqué un problema es difícil. Si un diseñador entiende las fuerzas de un patrón, la solidez de la solución se hace obvia. Alexander dice al respecto [2]:

> Es casi imposible caracterizar una casa que se ajuste a su contexto. Aun así es la cosa más fácil en el mundo nombrar los tipos específicos

de desajustes que no permiten un buen ajuste. Una cocina que es difícil de limpiar, la falta de un sitio de estacionamiento, los niños jugando donde pueden ser arrollados por un automóvil, goteras, espacios llenos de personas o falta de privacidad, la parrilla al nivel que echa aceite hirviendo directo a los ojos, la perilla dorada de plástico que decepciona espectativas, la puerta principal que no se puede encontrar, todos son desajustes entre la casa y las vidas de los habitantes a los que debería ajustarse. Estos desajustes son las fuerzas que deben darle forma, y no deben confurndirse. Porque se expresan en forma negativa, son específicas y suficientemente tangibles como para hablar de ellas.

Considérese el siguiente patrón de Gerard Meszaros, que es particularmente provocativo en cuanto a su sistema de fuerzas [28]:

Leaky Bucket of Credits

Problema

¿Cómo puede saber un procesador si otro procesador es capaz de ocuparse de más trabajo?

Fuerzas

Para que un dispositivo periférico sea capaz de rechazar más trabajo cuando el sistema se encuentra sobrecargado, debe ser al mismo tiempo capaz de reconocer cuando se encuentra en tal estado. Pero tener al procesador como cuello de botella consumiendo ciclos valiosos de tiempo para informar a los periféricos (que pueden ser potencialmente muchos) reduciría aún más su capacidad. Y ¿qué sucede si llega a estar tan cargado que puede enviar un mensaje solicitando que no se le envíe más trabajo?

Solución

El procesador le comunica al periférico cuando es capaz de aceptar mas trabajo, y lo hace mediante enviar "créditos" al dispositivo. Cada periférico rastrea un "cubo de créditos" que recibe del procesador. Conforme se envían peticiones al procesador, o simplemente pasa el tiempo, el cubo va "goteando" hasta que se vacía. Cuando el sistema no está a toda su capacidad, el cubo se vuelve a llenar continuamente por créditos nuevos enviados desde el procesador; sin embargo, si el sistema ya está a toda su capacidad, el procesador no enviará más créditos, y los periféricos retendrán el trabajo que llegue.

Las fuerzas, por sí mismas, informan al lector si se trata de un problema difícil o hasta un problema insoluble. Las fuerzas "jalan" al diseño en dos o más direcciones, ayudando al proceso de pensamiento a explorar todas las opciones y posibilidades, y a considerar los rincones obscuros del diseño.

La solución balancea las fuerzas, pero a la vez significa algo más que solo resolverlas.

Solución

Una buena solución tiene suficiente detalle como para informar al diseñador qué hacer, pero es lo suficientemente general para considerar un contexto más amplio. La solución debe resolver lo enunciado en el problema. Algunos patrones proveen sólo soluciones parciales que abren rutas hacia otros patrones, los cuales tienden a balancear las fuerzas no resueltas.

Si un patrón es literatura, entonces puede verse como una obra teatral en la que la solución debe proveer una cierta catarsis. El contexto introduce a los personajes y la circunstancia; las fuerzas proveen la trama, y la solución provee una resolución a la tensión que se va creando debido al conflicto de las fuerzas. Esta descripción teatral permite enfatizar la importancia de la solución dentro de la descripción de un patrón.

Representación gráfica

Alexander sostiene que una representación gráfica (un *sketch*) es la esencia del patrón. En *Notes on the Synthesis of Form*, uno de sus trabajos iniciales, hace la siguiente afirmación [2]:

> Estos diagramas, los cuales, en mi trabajo más reciente, he estado llamando patrones, son la clave para el proceso de crear la forma..., la mayoría del poder de lo que he escrito recae en el poder de estos diagramas...
>
> Poincaré una vez dijo: los sociólogos discuten métodos sociológicos; los físicos discuten física. Me encanta esta afirmación. El estudio del método por sí mismo es siempre estéril, y la gente que ha tratado [este libro] como si fuera acerca de un método de diseño casi siempre ha perdido el punto acerca de los diagramas y su gran importancia, porque se obsesionan con los detalles del método que propongo para obtener los diagramas.

También, en su libro *The Timeless Way of Building*, Alexander afirma simplemente [4]:

> Si no puedes dibujar un diagrama de ello, no es un patrón.

¿Qué se dibuja en un Patrón de Software? Cualquier cosa que se piense pueda ayudar al diseñador a entender la relación entre las partes. La representación gráfica generalmente conlleva *estructura*. El libro *Design Patterns* [23] utiliza diagramas en OMT (una notación ampliamente aceptada para diseño de software) para presentar ejemplos de estructuras de la solución para cada uno de sus patrones. Los diagrams de interacción, que ilustran el orden de los eventos u otro tipo de dinámicas entre componentes, son igualmente útiles.

Los arquitectos clásicos no dibujaban acerca de nociones abstractas de arte por su propio gusto, sino más bien se limitaban a estructuras familiares que servían bien a la cultura. En tal sentido, la arquitectura es diferente a la pintura. Y los constructores de hogares confortables y comunes generalmente no seguían un plan maestro pre-escrito, sino ensamblaban lodo, paja, piedras o ladrillos para construir primero los cimientos, luego las paredes y puertas, en seguida las ventanas, después el techo, y finalmente las paredes interiores. Pre-ordenar todas estas estructuras hace imposible para el constructor el manejo de la interacción entre una casa que crece y su ambiente: luz, viento y sombras de las estructuras vecinas. Aun cuando se pueden prever tales problemas, los dibujos bidimensionales iniciales no pueden capturar la expresión de la estructura completa.

Los documentos para la especificación y arquitectura de software son análogos a los planes de construcción. Considérese la "Ley de Alberti" [33]:

> He aquí otra desventaja: los dibujos hermosos pueden volverse fines en sí mismos. Frecuentemente, si el dibujo engaña, no es sólo el observador quien queda encantado sino también el autor, quien es víctima de su propio artificio. Alberti entendió este peligro y señaló que los arquitectos no deben tratar de imitar a los pintores, produciendo obras realistas. El propósito del dibujo arquitectónico, de acuerdo con él, es meramente ilustrar la relación de las varias partes... Alberti entendió, como muchos arquitectos hoy día no lo hacen, que las reglas de dibujo y las reglas de construcción no son las mismas, y que la maestría en las primeras no asegura el éxito en las últimas.

Alexander dice algo muy similar [4]:

> Es esencial, por lo tanto, que el constructor construya sólo a partir de dibujos a grandes rasgos, y que realicen los patrones detallados a partir de los dibujos de acuerdo a los procesos dados en el lenguaje de patrones en su mente.

Es por esto que, más que una representación o especificación gráfica detallada, se prefieren las representaciones esquemáticas. La mayoría de los lectores tienden a interpretar diagramas refinados muy literalmente. Y aún se puede decir mucho acerca de diagramas hechos a mano que no tienen ángulos o líneas rectas. Este tipo de representaciones esquemáticas parecen animar al diseñador a desarrollar la solución a mano.

Consecuencias

Las consecuencias se refieren al contexto resultante, que es como una conclusión del patrón. Las consecuencias comunican:

- qué fuerzas han sido resueltas o balanceadas
- cuáles nuevos problemas pueden surgir a partir de la aplicación del patrón
- qué patrones relacionados pueden seguir

Cada patrón se desarrolla para transformar a un sistema en un contexto, para producir un nuevo sistema dentro de un nuevo contexto. El contexto resultante de un patrón muchas veces es la entrada a otros patrones que le siguen. Los contextos, entonces, sirven como liga entre patrones relacionados dentro de un lenguaje de patrones.

2.3.2. Formas comunes

Un patrón es una forma literaria. Durante los últimos años, se han descubierto y han evolucionado varias formas comunes para describir patrones. Esta sección resume las formas de patrones más populares, haciendo referencia a quienes las originaron.

Forma Alexander

La forma Alexander se presenta en el trabajo de Christopher Alexander. De hecho, se le considera como la forma "original" de los patrones. Las secciones que considera un patrón en la forma Alexander no están fuertemente delimitadas. La estructura sintáctica más notable es un "Por lo tanto" (*Therefore*) que inmediatamente precede a la solución. Otros elementos de esta

forma se presentan comúnmente en las demás formas: una descripción clara del problema, una discusión de las fuerzas, la solución, y un razonamiento.

En esta forma, cada patrón usualmente comienza con un párrafo introductorio que enumera los patrones que ya se han aplicado anteriormente, para dar una referencia al patrón actual. El patrón mismo comienza con un nombre y una "designación de confianza" de cero, una, o dos estrellas. Los patrones con dos estrellas son quéllos en los cuales los autores tienen mayor confianza, dada su fundamentación empírica. Los patrones con menos estrellas pueden tener una significancia social fuerte, pero son más especulativos.

Alexander describe su forma de la siguiente manera [3]:

> Por conveniencia y claridad, cada patrón tiene el mismo formato. Primero, hay una figura, que muestra un ejemplo arquetípico de tal patrón. Segundo, después de la figura, cada patrón tiene un párrafo introductorio, el cual establece el contexto para el patrón mediante examinar cómo ayuda a completar ciertos patrones de mayor escala. Entonces, se colocan tres diamantes para marcar el inicio del problema. Después de los diamantes hay un encabezado, en negrilla. Este encabezado da la esencia del problema en uno o dos enunciados. Después del encabezado viene el cuerpo del problema. Esta es la sección más larga, y en ella se describe los antecedentes empíricos del patrón, la evidencia de su validez, el rango de maneras diferentes en que el patrón se manifiesta en un edificio, etc. Enseguida, y también en negrilla, se presenta la solución como el corazón mismo del patrón, el cual describe el campo de relaciones físicas y sociales que se requieren resolver en el problema ya enunciado y dado el contexto ya descrito. Esta solución es siempre descrita en términos de una instrucción a fin de que se sepa exactamente qué se necesita hacer para construir el patrón. Entonces, después de la solución, se coloca un diagrama que muestra la solución en forma gráfica, con etiquetas que indican sus componentes principales.
>
> Después del diagrama se colocan otros tres diamantes para mostrar que el cuerpo principal del patrón ha terminado. Y finalmente, después de los diamantes, hay un párrafo que liga al patrón con todos aquellos patrones más pequeños en escala dentro del lenguaje, que se requieren para completar este patrón, para embellecerlo, para llenarlo.

¿Porqué Alexander adopta esta forma? Da como argumento el siguiente [3]:

> Hay dos propósitos esenciales destrás de este formato. Primero, presentar cada patrón conectado a otros patrones, de modo que se pueda

cubrir el conjunto de todos los 253 patrones como un todo, como un lenguaje, dentro del cual se pueden crear una infinita variedad de combinaciones. Segundo, para presentar el problema y la solución de cada patrón de tal manera que se pueda juzgarse por uno mismo, y modificarlo sin perder la esencia que es central en él.

Forma GoF

La forma GoF (de *Gang of Four*) se establece en el libro *Design Patterns* [23]. Como se describe en este libro, la forma GoF tiene las siguientes secciones:

Nombre del patrón y clasificación: El nombre del patrón conlleva la esencia del patrón suscintamente. Un buen nombre es vital, porque se volverá parte de su vocabulario de diseño...

Intención: Un enunciado corto que responde las siguientes preguntas: ¿Qué hace el patrón de diseño? ¿Cuál es su razonamiento y su intención? ¿Qué cuestión particular de diseño o problema soluciona?

También conocido como: Otros nombres conocidos del patrón, si existen.

Motivación: Un escenario que ilustra un problema de diseño y cómo la estructura de clases u objetos en el patrón resuelve el problema. El escenario ayuda a entender una descripción más abstracta del patrón, más adelante.

Aplicabilidad: ¿En qué situaciones el patrón de diseño puede aplicarse? ¿Cuáles son ejemplos de diseños pobres que el patrón puede resolver? ¿Cómo reconocer tales situaciones?

Estructura: Una representación gráfica de las clases en el patrón utilizando la notación basada en el *Object Modeling Technique* (OMT) [32]. También se utilizan diagramas de interacción [24, 10] para ilustrar las secuencias de peticiones y colaboraciones entre objetos...

Participantes: Las clases y/o objetos participantes en el patrón de diseño, y sus responsabilidades.

Colaboraciones: ¿Cómo colaboran los participantes para llevar a cabo sus responsabilidades?

Consecuencias: ¿Cómo el patrón logra sus objetivos? ¿Cuáles son los pros y contras y el resultado de usar el patrón? ¿Qué aspectos de la estructura del sistema permite variar independientemente?

Implementación: ¿Qué problemas, ayudas, o técnicas deben considerarse cuando se implementa el patrón? ¿Son cuestiones específicas de un lenguaje de programación?

Muestra de Código: Fragmentos de código que ilustran cómo se puede implementar el patrón en C++ y Smalltalk.

Usos conocidos: Ejemplos del patrón encontrados en sistemas reales. Se incluyen al menos dos ejemplos de dominios diferentes.

Patrones relacionados: ¿Qué patrones de diseño están cercanamente relacionados con este patrón? ¿Cuáles son sus diferencias principales? ¿Con qué otros patrones debe utilizarse este patrón?

La forma GoF está hecha principalmente para diseños de software Orientado a Objetos.

Forma POSA

De manera similar a la forma GoF, la forma POSA (de *Pattern-Oriented Software Architecture*) se establce en el libro de Buschmann et al. [12]. La forma POSA tiene las siguientes secciones:

Nombre: El nombre y un resumen corto del patrón.

También conocido como: Otros nombres para el patrón, si hay algún otro conocido.

Ejemplo: Un ejemplo del mundo real en el que se demuestre la existencia del problema y la necesidad del patrón...

Contexto: La situación en la cual el patrón se aplica.

Problema: El problema que el patrón resuelve, incluyendo una discusión de sus fuerzas asociadas.

Solución: El principio fundamental de la solución que sirve como base al patrón.

Estructura: Una especificación detallada de los aspectos estructurales del patrón, que incluye un diagrama de clases en OMT [32].

Dinámica: Escenarios típicos que describen el comportamiento en tiempo de ejecución del patrón. Los escenarios se ilustran con diagramas de secuencia de mensajes entre objetos [32].

Implementación: Guías para la implementación del patrón...

Ejemplo resuelto: Discusión sobre cualquier aspecto importante para resolver el ejemplo que no se haya cubierto en las secciones Solución, Estructura, Dinámica e Implementación.

Variantes: Una breve descripción de las variantes o especializaciones del patrón.

Usos conocidos: Ejemplos de uso del patrón, tomados de sistemas existentes.

Consecuencias: Los beneficios que provee el patrón, y cualquier potencial desventaja.

Véase también: Referencias a patrones que resuleven problemas similares, y a patrones que ayudan a refinar el patrón que se describe.

La forma POSA tiene muchas similitudes con la forma GoF, siendo ambas las más utilizadas dentro de la descripción de Patrones de Software. Ambas tienen su origen en la comunidad de programación Orientada a Objetos. Sin embargo, ambas formas pueden utilizarse para describir soluciones exitosas en aplicaciones no necesariamente Orientadas a Objetos.

Forma Coplien

La forma Coplien también refleja los elementos básicos en la descripción de patrones, presentes en la forma Alexander. Esta forma delinea secciones de patrones con títulos, e incluye:

El nombre del patrón: la forma Coplien comúnmente usa sustantivos para los nombres de los patrones, pero también se admiten frases verbales. Esto concuerda con la forma Alexander.

El problema: el problema es frecuentemente enunciado como una pregunta o reto de diseño. Esto es análogo a la sección en la forma Alexander que sigue después de los tres diamantes.

El contexto: Se trata de una descripción del contexto en el cual el problema puede ocurrir, y cómo se aplica la solución. Esto es similar al párrafo introductorio de Alexander en el que se presenta el contexto.

Las fuerzas: las fuerzas describen los pros y contras del patrón de diseño; qué "jala" a las decisiones de diseño en diferentes direcciones, y hacia diferentes soluciones de diseño. Esto corresponde a la descripción a fondo en la forma Alexander del problema.

La solución: la solución explica cómo resolver el problema, igual que en la descripción en negritas de la forma Alexander. Una representación gráfica simple (*sketch*) acompaña a la solución.

Un razonamiento: ¿porqué funciona el patrón? ¿cuál es la historia detrás de él? Esto se extrae del resto de la descripción para no sobrecargar la solución. Como sección, su objetivo es llamar la atención hacia la importancia y principios detrás del patrón; se considera una fuente de aprendizaje más que una fuente de acción.

Contexto resultante: esta sección describe cuáles fuerzas resuelve el patrón, y cuáles otras permanecen sin resolverse. Además, apunta hacia otros patrones que podrían ser los siguientes en considerarse.

2.3.3. Relaciones entre las partes de un Patrón

Normalmente, se espera que la descripción en la sección de la solución balancee los elementos expuestos en la sección del problema. Sin embargo, varios patrones pueden proponerse para resolver un mismo problema, y también, es posible que una solución resuelva múltiples problemas.

Como ejemplo, considérese el siguiente modismo (o *idiom*, un patrón de bajo nivel de abstracción), conocido como *Counted Body* [15]:

Nombre: Counted Body Idiom

Problema: Simular la semántica de la asignación de Smalltalk en C++.

Contexto: Un diseño se ha transformado en un par de clases *handle/body* en C++.

Fuerzas:

- La asignación en C++ se define recursivamente como una asignación miembro a miembro con una copia como terminación de la recursión; sería más eficiente y más similar a Smalltalk si el copiado fuera religado.
- La copia profunda de un *body* en general es una operación costosa.

- La copia puede evitarse usando apuntadores y referencias, pero éstos dejan el problema de quién es el responsable respecto al retorno del espacio de memoria del objeto, dejando además una distinción sólo a nivel usuario entre tipos interconstruidos y tipos definidos por el usuario.

- Compartir las partes de *body* en la asignación es semáticamente incorrecto si alguna de ellas se modifica mediante algún *handle* durante la copia.

Solución: Añadir un conteo de referencias a la clase *body* para facilitar la administración de memoria.

La administración de la memoria se añade a la clase *handle*, particularmente a la implementación de su inicialización, asignación, copiado y destrucción.

Se debe considerar en toda operación que modifique el estado de un *body* para romper su compartición mediante hacer su propia copia, lo que decrementa el conteo de referencias al *body* original.

Contexto resultante: Se evita una copia gratuita, guiándose por una implementación más eficiente.

La compartición se rompe cuando el estado del *body* se modifica mediante un *handle*.

La compartición se preserva en los casos comunes como paso de parámetros y otros.

Se evitan apuntadores y referencias especiales. Se aproxima más a una semántica de tipo Smalltalk.

Razonamiento de diseño: El conteo de referencias es eficiente y reparte cualquier retraso a través de la ejecución en tiempo real de programas.

Compárese este patrón con el modismo *Detached Counted Handle/Body* [15], que se presenta en la siguiente sección. Ambos patrones resuelven un mismo problema: administración de memoria. Ambos resuelven el problema mediante separar la implementación de la interface. ¿Porqué son diferentes? Cada uno tiene un contexto diferente, que se dilucidan de la descripción de las fuerzas. En el modismo *Detached Counted Handle/Body* se encuentra que la clase a ser administrada es una clase dada en una biblioteca de clases, a la cual no se puede añadir un miembro para el conteo de referencias. En

su contexto resultante, se describen preocupaciones respecto a los retrasos durante las operaciones en memoria y fragmentación, que no son problemas particulares en el modismo ordinario y original *Handle/Body* [15] (que es similar al *Counted Body*, pero sin un conteo de referencias).

Ahora bien, compárese el patrón *Counted Body* con el patrón *Bridge* [23]:

> **Intención**
>
> Desacoplar una abstracción de su implementación, de modo que las dos puedan variar independientemente.
>
> **También conocido como**
>
> *Handle/Body*
>
> **Motivación**
>
> Cuando una abstracción puede tener una de varias posibles implementaciones, la forma general de acomodarlas es utilizando herencia. Una clase abstracta define las interfaces a la abstracción, y subclases concretas la implementan de diferentes maneras. Sin embargo, esta aproximación no es siempre lo suficientemente flexible. La herencia liga una implementación a la abstracción permanentemente, lo que hace difícil de modificar, extender o reusar abstracciones e implementaciones independientemente.
>
> El patrón *Bridge* resuelve estos problemas mediante acotar la abstracción y su implementación en jerarquías de clase separadas.
>
> **Aplicabilidad**
>
> Use el patrón *Bridge* cuando:
>
> - Se desea evitar ligado permanente entre la abstracción y su implementación.
> - Tanto las abstracciones como sus implementaciones deben ser extensibles mediante subclases.
> - Cambios en la implementación de una abstracción no deben tener impacto en los clientes, es decir, su código no debe ser recompilado.
> - En C++, se desea ocultar completamente la implementación de una abstracción de sus clientes.
> - Se tiene una proliferación de clases en una jerarquía. Esta indica la necesidad de dividir la descripción de un objeto en dos partes: abstracción e implementación.

- Se desea compartir la implementación entre múltiples objetos (tal vez usando conteo de referencias), y este hecho debe estar oculto a los clientes. Un ejemplo simple se muestra en la clase *String* de Coplien [15], en la cual múltiples objetos comparten una misma representación de cadenas de caracteres (*StringRep*).

Estos patrones están claramente relacionados. Sin embargo, el patrón *Bridge* describe el problema desde la perspectiva de la solución: la necesidad de separar la implementación de la interface; el conteo de referencias se encuentra hasta el final de la lista de aplicabilidad.

El problema que se resuelve en el *Counted Body Idiom* es simular la semática de asignación de Smalltalk en C++. El *Detached Counted Handle/Body Idiom* resuelve un problema similar: provee de clases con el mismo comportamiento razonable que se espera de los tipos primitivos. En muchos contextos, la solución del problema es separar la interface de la implementación, que es el problema clave resuelto por el patrón *Bridge*.

Muchas de estas sutilezas dan origen a la imprecisión del lenguaje natural y la complejidad del diseño. La investigación en un futuro puede proveer de bases mejores para una "lingüística" de patrones, que ayude a regularizar las relaciones entre los espacios de problemas y soluciones.

2.4. Los Patrones y los Paradigmas

Uno de los más importantes retos en el diseño de sistemas es manejar la complejidad. La complejidad se ataca con la abstracción. Mucho de la labor de diseño se ocupa de encontrar las abstracciones adecuadas en un sistema, particionando el sistema en trozos manejables mentalmente.

Para dividir un sistema en partes, se usan un conjunto consistente de guías, principios, reglas y herramientas. Un paradigma es una visión del mundo que comprende un conjunto consistente de reglas y herramientas que se utilizan para particionar un sistema en abstracciones manejables. Provee de principios organizativos amplios que ayudan en la estructuración de sistemas partiendo de una comprensión amplia de su funcionalidad, particularmente en la posible ausencia de experiencia o expertez de un dominio.

Normalmente, hay una gran cantidad de "huecos" en el espacio de diseño que la mayoría de los paradigmas (y sus métodos de diseño asociados) dejan sin llenar. Los grandes diseñadores saben cómo llenar esos huecos, mediante

utilizar su intuición o recurriendo a sus experiencias pasadas. Son este tipo de ideas las que se desea capturar como patrones: estructuras de diseño que pueden regularizarse fácilmente en un método.

Conforme la tecnología progresa, los patrones de hoy se convierten en los paradigmas de mañana; los paradigmas de mañana se convierten en los lenguajes de programación de la próxima semana. Los trucos de diseño de los diseñadores de los primeros compiladores están actualmente regularizados en herramientas como *yacc* y *bison*. Y sin embargo, algunos patrones parecen desafiar la regularización através del tiempo. Muchos de los patrones de telecomunicaciones son todavía parte de la práctica empírica de diseño. Hay un costo por mantener tal conocimiento a nivel empírico: es difícil utilizar y mantener sistemas desarrollados por otros. Los patrones tienen como objetivo el obtener tal conocimiento, y regularizarlo para su evolución posterior.

3. ¿Qué son Lenguajes de Patrones?

Un lenguaje de patrones es una colección de patrones que se complementan entre sí para generar un sistema. Un patrón aislado resuelve un problema de diseño aislado; un lenguaje de patrones construye un sistema. Es mediante lenguajes de patrones que los patrones logran su objetivo.

El término "lenguaje de patrones" proviene de la arquitectura de edificios, popularizada por Alexander. Se puede comparar un lenguaje de patrones con un lenguaje natural. El inglés puede generar enunciados con significado en inglés, mientras que un lenguaje de patrones sobre la forma de los datos de errores de entrada (como el lenguaje de patrones CHECKS) puede generar una arquitectura de manejo de errores para las interfaces humano-computadora que se ajusten a su contexto.

Este sentido de la palabra "lenguaje" no se utiliza comúnmente en computación. Un lenguaje de patrones no debe confundirse con un lenguaje de programación. Un lenguaje de patrones es una pieza de literatura que describe una arquitectura, un diseño, un *framework* o cualquier otra estructura observada en el software. Tiene una estructura, pero no el mismo nivel de estructura formal que se encuentra en los lenguajes de programación. El término "lenguaje de patrones" ha sido fuente de confusión por esto mismo, lo que ha provocado que algunos autores de patrones prefieran el uso de "sistema de patrones" [12].

Los lenguajes de patrones están cercanamente relacionados con la noción

de D.L. Parnas de familia de software (*software family*) [30]. Una familia de software comprende a varios miembros relacionados entre sí por sus "comunalidades" (lo que es común entre ellos), y se distinguen entre sí por sus "variabilidades" (lo que varía entre ellos). Considerando esto, se puede pensar en un lenguaje de patrones como una colección de reglas para construir todos los miembros de una familia de software, y sólamente miembros de esa familia. Ejemplos en la arquitectura de edificios, que Alexander presenta como lenguajes de patrones, son las casas en Cape Cod, las granjas en Bernese Oberland, las casas de piedra del sur de Italia, y otros géneros de construcciones [4].

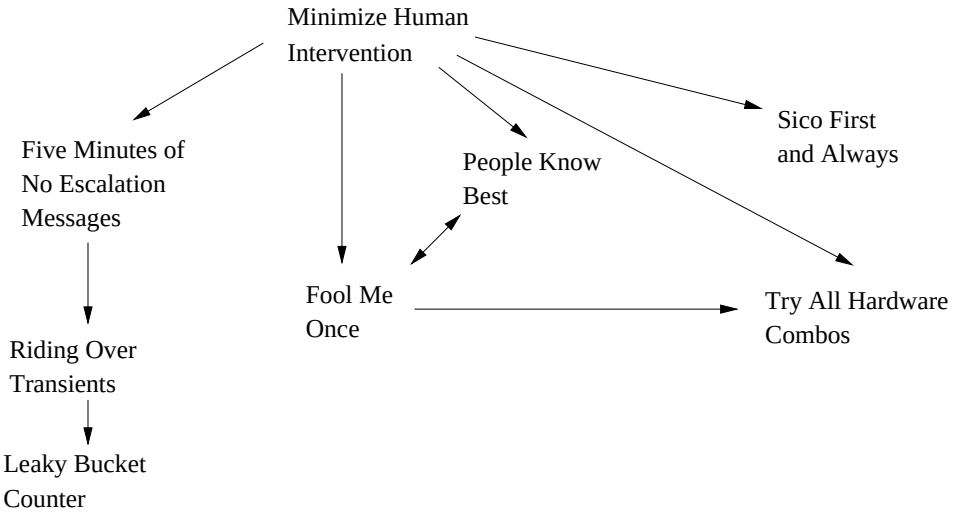
Un lenguaje de patrones no es tan solo un árbol de decisión de patrones. Esto se debe parcialmente a que los patrones en un lenguaje de patrones forman un grafo acíclico dirigido, no una jerarquía. El número de caminos distintos através de un lenguaje de patrones es muy grande.

Los lenguajes de patrones colocan a los patrones individuales en contexto. Alexander dice al respecto [4]:

> Cada patrón entonces, depende tanto de los patrones más pequeños que contiene, como en los patrones más grandes entre los que se contiene...
>
> Y es la red de estas conexiones entre patrones lo que crea al lenguaje.
>
> En esta red, las ligas entre los patrones son casi tanto una parte del lenguaje como los patrones mismos.
>
> Es, en verdad, la estructura de esta red lo que le da sentido a los patrones individuales, porque los ancla, y ayuda a completarlos.
>
> Pero aún cuando tenga los patrones conectados entre sí, en una red, de modo que formen un lenguaje, ¿cómo sé si el lenguaje es uno bueno?
>
> El lenguaje es uno bueno, capaz de hacer algo un todo, cuando es morfológicamente y funcionalmente completo.
>
> El lenguaje es morfológicamente completo cuando puedo visualizar muy concretamente la clase de edificios que genera.
>
> Y el lenguaje es funcionalmente completo cuando el sistema de patrones que define es completamente capaz de permitir todas sus fuerzas internas para que se resuelvan a sí mismas.

3.1. Un Ejemplo de un Lenguaje de Patrones de Software

AT&T ha ensamblado un gran lenguaje de patrones que captura las prácticas expertas de uno de sus dominios clave de negocio. Ese gran lenguaje de patrones puede dividirse en muchos lenguajes auto-consistentes en un entorno más modesto. Uno de esos lenguajes fue publicado por Adams et al. [1]. Incluye los patrones *Leaky Bucket Counter* y *Riding Over Transients*, como patrones pequeños que complementan al patrón *Minimize Human Intervention*. La estructura del lenguaje de patrones se muestra en la siguiente figura [1]:



Más adelante, se examinan los dos patrones en la esquina inferior izquierda del diagrama en mayor detalle. Por ahora, se describe a continuación el patrón *Five Minutes of No Escalation Messages* [1]:

> **Nombre:** *Five Minutes of No Escalation Messages*
>
> **Problema:** Los mensajes de la computadora se salen rápidamente de la pantalla: la interface humano-computadora se satura con reportes de error que se salen de la pantalla, consumiendo recursos tan solo por la actividad intensa de despliegue.
>
> **Contexto:** Cualquier sistema tolerante a fallas que contínuamente se ejecuta con escalamiento, donde condiciones de transición pueden estar presentes.
>
> **Fuerzas:**

- No hay sentido en perder tiempo o en reducir el nivel de servicio tratando de resolver un problema que desaparecerá por sí mismo.
- Muchos problemas se resuelven si se les da el tiempo suficiente.
- Es indeseable que se utilicen todos los recursos tan solo para desplegar mensajes.
- No se desea sobresaltar al usuario al hacerle pensar que el sistema está fuera de control (*Minimize Human Intervention*).
- La única acción del usuario relacionada con el escalamiento de mensajes puede ser inapropiada para el objetivo de preservar la salud del sistema.
- Hay otros sistemas de cómputo monitoreando las acciones realizadas. Estos sistemas pueden manejar un mayor volumen de mensajes.

Solución: Cuando se toma la primera acción que puede llevar a un exceso de mensajes, se despliega el mensaje. Entonces, periódicamente se despliega un mensaje de actualización sobre el estado. Si la condición anormal termina, se despliega un mensaje informando que todo ha vuelto a la normalidad. Sin embargo, no se despliega un mensaje por cada cambio de estado (véase *Riding Over Transients*).

Contínuamente se comunica el estado y las acciones tomadas al flujo de monitoreo del sistema de cómputo durante este periodo.

Por ejemplo, cuando el interruptor 4ESS entra al primer nivel de carga del sistema, se envía un mensaje al usuario. No se envían mas mensajes por los siguientes 5 minutos, aún cuando puede haber escalación en la condición. Después de los 5 minutos, se despliega un mensaje indicando el estado. Cuando la condición termina, se despliega un mensaje al respecto.

Contexto resultante: El operador del sistema no se sobresalta al ver un gran número de mensajes de error. Mensajes y medidas entre máquinas mantienen un registro para evaluación posterior, así como mantienen las acciones del sistema visibles a las personas que trabajan con él. En el ejemplo de sobrecarga del 4ESS, los contadores continúan el seguimiento de la dinámica de sobrecarga; algún sistema de soporte da a la vez seguimiento a estos contadores.

Otros mensajes, que no se relacionan con la situación de escalamiento que produce demasiados mensajes, se despliegan como si el sistema se encontrara en operación normal. Por tanto, el funcionamiento normal del sistema no se ve adversamente afectado por el volumen de mensajes.

Nótese el conflicto con *People Know Best*.

Este patrón hace referencia al patrón *Riding Over Transients* como uno de los mecanismos primarios para su implementación; a su vez, *Riding Over Transients* emplea el patrón *Leaky Bucket Counter*. De hecho, se podría utilizar *Leaky Bucket Counter* para implementar *Five Minutes of No Escalation Messages* para variaciones respecto a los 5 minutos.

Lo que hace a estos patrones un lenguaje es el hecho de que pueden hacer referencias entre ellos, a fin de cooperar para resolver un problema más amplio, que en este caso se refiere a minimizar la intervención humana en un sistema. Muchos de los patrones en este lenguaje también aparecen en otros lenguajes de patrones. *Leaky Bucket Counters* es un patrón versatil que puede utilizarse en muchos niveles de diseño.

Un solo patrón resuelve un solo problema en un forma dada y general. La diferencia de contextos, entonces, sugiere la existencia de diferentes soluciones para lo que sería de otra forma el mismo problema. Cada contexto, entonces, merece un patrón diferente. Un lenguaje de patrones puede presentar múltiples patrones para el mismo problema, aun cuando cada solución se ajusta a un contexto diferente. *People Know Best* intenta resolver el problema de confiablidad de manera muy diferente a otros patrones dentro del mismo lenguaje de patrones [1]:

> **Nombre:** *People Know Best*
>
> **Problema:** ¿Cómo balancear entre la automatización y la autoridad y responsabilidad humana?
>
> **Contexto:** Sistemas de alta confiabilidad y ejecución continua, en donde el sistema mismo trata de recuperarse de condiciones de error.
>
> **Fuerzas:**
>
> - Las personas tienen un sentido subjetivo bastante bueno del paso del tiempo, y cómo se relaciona con la probabilidad de una falla seria, o cómo la falla va a ser percibida por el cliente.
> - El sistema se crea para recuperarse de casos de falla (*Minimize Human Intervention*).
> - Las personas sienten que deben intervenir.
> - La mayoría de los errores de sistema pueden originarse por un error humano.

Solución: Supóngase que las personas saben mejor qué hacer, particularmente la gente de mantenimiento. Diseñe el sistema de modo que se permita a los usuarios con conocimiento tomar control en lugar del sistema automático.

Contexto Resultante: Las personas se sienten en control de la situación. Sin embargo, también son responsables de sus acciones.

Esto puede ser una regla absoluta: las personas sienten una necesidad de intervenir. No hay una solución perfecta para este problema, y el patrón no puede resolver todas las fuerzas adecuadamente. El patrón *Fool Me Once* es una solución parcial, respecto a que no le da al ser humano oportunidad de intervenir.

Nótese la tensión entre este patrón y *Minimize Human Intervention*.

Obsérvese que el patrón menciona explícitamente la tensión, y que ésta apunta a patrones alternativos que se aplican en diferentes contextos. De hecho, ¿cómo se ve *Minimize Human Intervention*? [1]:

Nombre: *Minimize Human Intervention*

Problema: La historia muestra que las personas causan la mayoría de los problemas en sistemas de ejecución contínua (acciones erradas, sistemas errados, botones errados).

Contexto: Sistemas digitales de alta confiabilidad contínuamente en ejecución, donde los tiempos fuera, tal vez inducidos por los seres humanos u otros sistemas, deben minimizarse.

Fuerzas:

- Los seres humanos son realmente inteligentes; las máquinas no lo son. Los humanos son mejores en detectar patrones de comportamiento del sistema, especialmente entre ocurrencias aparentemente aleatorias separadas por el tiempo (*People Know Best*).
- Las máquinas son buenas para orquestar una estrategia global bien pensada, mientras que los humanos no.
- Los seres humanos son falibles; las computadoras son frecuentemente menos falibles.
- Los humanos sienten la necesidad de intervenir si no pueden ver que el sistema hace serios intentos por restaurarse. Los tiempos humanos de reacción y decisión son muy lentos (por ordenes de magnitud) comparados con los procesadores de las computadoras.
- Un sistema silencioso es un sistema muerto.

- Los operadores humanos se aburren con la vigilancia contínua y pueden ignorar o no notar eventos críticos.
- El procesamiento normal o las fallas pueden ocurrir tan rápido que la inclusión del operador humano no es factible.

Solución: Deje a las máquinas tratar de hacer todo por sí mismas, dejando a los seres humanos sólo como un acto de desesperación o último recurso.

Contexto resultante: Un sistema es menos susceptible a errores humanos. Esto hace a los clientes del sistema más felices. En muchas administraciones, la compensación al operador del sistema se basa en la disponibilidad del sistema, así que esta estrategia realmente mejora la parte del operador.

Razonamiento de diseño: Empíricamente, una fracción desproporcionada de fallas en sistemas de alta disponibilidad se deben a errores del operador, no del sistema. Mediante minimizar la intervención humana, la disponibilidad del sistema puede mejorarse. La intervención humana puede reducirse mediante construir estrategias que prevengan al ser humano de actuar forzosamente; véanse los patrones como *Fool Me Once*, *Leaky Bucket Counters*, y *Five Minutes of No Escalation Messages*.

Nótese la tensión entre este patrón y *People Know Best*.

Este patrón de alto nivel apunta claramente a patrones más pequeños que lo refinan y completan para contextos más específicos. Tal relación entre patrones hace a este un lenguaje con una estructura mucho mejor que tan solo una colección de patrones débilmente acoplados.

Muchos de los patrones en este lenguaje apuntan hacia *People Know Best* como un opuesto, y aun así, el lenguaje de patrones pone en claro que este patrón no puede ser ignorado. Cada patrón instruye a hacer algo específico, pero un diseño completo requiere la aplicación balanceada de todos estos patrones en conjunto. Dos sistemas construidos con estos patrones no serán iguales debido a que su diseño y construcción depende del balance entre requerimientos de negocio y de la propia aplicación.

Este lenguaje, como un todo, genera una familia de arquitecturas. La mayoría de los sistemas modernos de telecomunicación usan estos u otros patrones relacionados. Estos patrones definen un género de software, no en términos de protocolos, interfaces o requerimientos, sino en términos de

estructuras básicas y mecanismos que surgen de las necesidades de la organización. Las vidas tanto de los clientes como de la gente que mantiene estos sistemas se mejora através de estos patrones. Las compañías han comenzado a reconocer los lenguajes de patrones detrás de sus sistemas, y los están usando para documentación y construcción de sistemas [6].

3.2. Lenguajes de Patrones comparados con Catálogos de Patrones

Los populares patrones de Gamma et al. [23] forman un catálogo de patrones, pero rigurosamente no forman un lenguaje de patrones. Es decir, los patrones no están lo suficientemente completos para generar todos los programas en un dominio (los programas Orientados a Objetos son el dominio de los patrones de Gamma et al.).

Lo anterior no significa que los patrones en un catálogo no tengan una estructura. Gamma et al. ponen de manifiesto la estructura de su catálogo de patrones en su libro, dentro de la cubierta; otros han descrito otras formas de clasificar los patrones de Gamma et al. [41].

Los catálogos de patrones, y particularmente el trabajo de Gamma et al., son la fuente más común de patrones de uso actualmente. El libro de Gamma et al. apareción muy a tiempo: lo suficiente tarde como para capturar las prácticas probadas en el paradigma de objetos, y suficientemente temprano para ser útil al enorme grupo de desarrolladores de objetos que emergieron. Tales patrones básicos muy probablemente permanecerán como una parte clave de la literatura del desarrollo de software por varios años, ya que sus problemas y soluciones son atemporales. Pero se puede ir un paso más allá: estos patrones no surgen por características del sistema. Es muy probable que vayan apareciendo lenguajes de patrones en dominios de aplicación particulares, como por ejemplo, diseño cliente/servidor, procesamiento distribuido de transacciones financieras, telecomunicaciones tolerantes a fallas, y muchas mas. Aún cuando un lenguaje de patrones manejable no puede apoyar todos los aspectos del diseño de sistemas en general, puede hacer una buena labor dentro de un dominio bien delimitado. Los lenguajes de patrones continuarán creciendo en importancia conforme la disciplina madure.

4. Dominios de Patrones

Los Patrones de Software surgen a partir de los patrones de Alexander para diseño urbano y arquitectura de edificios. Se han adaptado los principios y valores (pero no los patrones en sí) para la producción de software. El salto de arquitectura de edificios a arquitectura de software es intuitiva para muchos: ambas se ocupan de estructuras. Pero los patrones pueden y tienen que ser usados en disciplinas de software menos estructuradas, como el desarrollo de procesos y la capacitación. Algunos patrones (véase, por ejemplo [9]) combinan múltiples dominios.

En seguida, se presenta un patrón de proceso que captura una práctica recurrente en equipos de desarrollo de software de alto desempeño [16, 20]:

> **Nombre:** *Mercenary Analist*
>
> **Problema:** El soporte de una notación de diseño así como la documentación del proyecto relacionado son labores muy tediosas para las personas que directamente contribuyen en la producción de artefactos.
>
> **Contexto:** Se proponen los roles para una organización. La organización existe en un contexto en el que los revisores externos, clientes y desarrolladores internos esperan utilizar la documentación del proyecto para entender la arquitectura del sistema y su funcionamiento interno (la documentación para el usuario se considera separadamente).
>
> **Fuerzas:**
>
> - Si los desarrolladores realizan su propia documentación, no les permite hacer el trabajo para el que fueron contratados.
> - La documentación es frecuentemente una actividad de sólo escritura.
> - Los ingenieros comúnmente no tienen una buena habilidad para comunicarse.
> - Los arquitectos se han vuelto víctimas de la elegancia de sus propios dibujos (véase el razonamiento de diseño).
>
> **Solución:** Contrate un escritor técnico que tenga experiencia en los dominios necesarios pero que no tenga interés en el diseño mismo. Esta persona capturará el diseño utilizando una notación adecuada y dará formato y publicará el diseño para su revisión y uso en la organización.

La documentación misma debe mantenerse en línea siempre que sea posible. Debe mantenerse actualizada (y por tanto, el trabajo de "analista mercenario" es un empleo de tiempo completo), y debe relacionarse con los escenarios propuestos por el cliente (como en el patrón *Scenarios Define Problem*).

Contexto resultante: El éxito de este patrón depende en encontrar un agente adecuadamente capaz para cumplir el rol de analista mercenario. Si el patrón es exitoso, el nuevo contexto define un proyecto cuyo progreso puede ser revisado (el patrón *Review the Architecture*) y monitoreado por la comunidad de expertos fuera del proyecto.

Razonamiento de diseño: El Quattro Pro para Windows de Borland y muchos proyectos en AT&T han utilizado este patrón. Sin embargo, resulta difícil encontrar a las personas con las habilidades necesarias para llevar a cabo este rol.

Rybczynski menciona al respecto [33]:

He aquí otra desventaja: los dibujos detallados y bellos pueden volverse un fin en sí mismos. Frecuentemente, si los dibujos eran meritorios, no era sólamente el observador quien quedaba encantado con ellos, sino el propio elaborador, quien se vuelve víctima de su propio artificio. Alberti comprendió este peligro y señaló que los arquitectos no deben intentar imitar a los pintores, produciendo dibujos que parezcan vivos. El propósito de los dibujos arquitectónicos, de acuerdo con él, era meramente ilustrar la relación entre las varias partes... Alberti entendió, lo que muchos otros arquitectos de hoy no entienden, que las reglas del dibujo y las reglas de la construcción no son únicas ni las mismas, y que la maestría en la primera no asegura el éxito en la segunda.

En seguida, se presenta otro patrón para entrenamiento, proveniente del lenguaje de patrones de Dana Anthony, de la Knowledge Systems Corporation, Inc. [5]:

Chicken and Egg

También conocido como *Need to Know*, *Simplified Mutual Prerequisites*, e *Illusion of Understanding*.

Problema: Dos conceptos son ambos prerequisitos uno del otro. Es una situación del huevo o la gallina: un estudiante que no conoce A no entenderá B; pero un estudiante que no sabe B, no entenderá A.

Restricciones y fuerzas: Se podría explicar un concepto y luego el otro, pero en el punto intermedio, todos estarían confundidos. Mucha gente, si se confunde, deja de intentar entender; esto invalida una

aproximación de sólo seguir adelante. Se podría entonces intentar simplificar un concepto sólo para explicar el otro, pero también muchas personas no aceptan que se les de la información incompleta, aun por su propio bien.

Solución: Dé a los estudiantes la ilusión de entender, mediante explicar ambos conceptos A y B muy superficialmente, pero esencialmente. Itérese la explicación una y otra vez, cada vez en mayor detalle. Asegúrese de mantener la ilusión de entendimiento en cada iteración.

Patrones relacionados: El patrón *Mix New and Old* se relaciona con este patrón. Mientras se itera alrededor de dos conceptos o tópicos, mézclese nuevo material de cada concepto con una revisión del material ya cubierto. También es posible variar el estilo de explicación en cada iteración.

Los patrones pueden ser útiles en cualquier dominio de problema-solución con el legado de la experiencia. Los dominios definidos precisamente, como aquéllos cuya estructura puede ser fácilmente capturada, se prestan para integrarse en lenguajes de patrones.

5. Clasificando Patrones

Las jerarquías son una de las formas predominantes de estructurar y abstraer elementos conceptuales. Comúnmente, en programación se observan jerarquías procedurales en el diseño estructurado, y jerarquías de herencia en el diseño Orientado a Objetos.

Los modelos de categorización de patrones iniciales caen naturalmente dentro de una jerarquía. Anteriormente, se han mencionado algunas categorías de Patrones de Software. Conforme los patrones se van entendiendo mejor, se observa que tales categorizaciones sencillas resultan no satisfactorias para una clasificación. Es por ello que se requiere otro modelo de organización para los patrones.

5.1. Tres niveles de Patrones

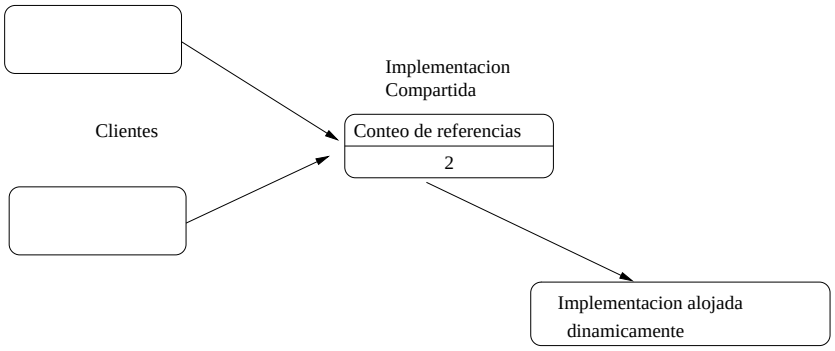
La Comunidad de Patrones ha convenido en un esquema de capas respecto a las categorías de patrones. Las capas permiten diferenciar niveles de abstracción, con Modismos (*Idioms*) en la base, Patrones de Diseño (*Design Patterns*) en el medio, y Patrones Arquitectónicos (*Architectural Patterns*) en la parte más alta. Esta sección describe estas categorías.

5.1.1. Modismos (*Idioms*)

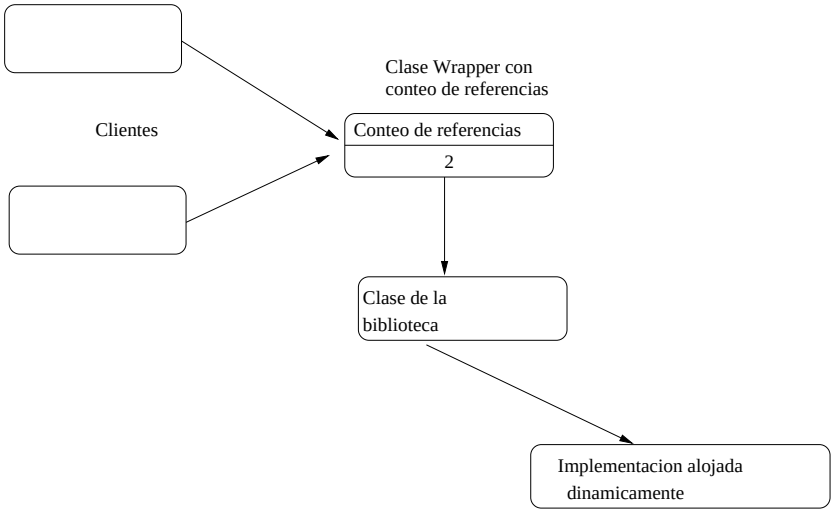
Los modismos son patrones de bajo nivel que dependen de la tecnología de implementación específica, tal como el lenguaje de programación. Los modismos se encuentran entre los primeros Patrones de Software publicados, emergiendo a finales de 1991 [15], aun cuando tales modismos iniciales no se expresaron en forma de patrón.

A continuación se presenta un modismo en forma de patrón, basado en sugerencias de Andrew Koenig y Bjarne Stroustrup, quienes refieren tener amplia experiencia con este patrón [15]:

> **Nombre:** *Detached Counted Handle/Body*
>
> Muchos programadores en C++ expresan tipos cuya implementación usa memoria dinámica. Los programadores frecuentemente crean tales tipos, y los colocan en bibliotecas sin añadir el código que hace a esto tipos "comportarse bien" como los tipos primitivos. La solución estándar, el patrón *Counted Body*, incluye una cuenta de referencias en una implementación compartida que se maneja mediante una clase *handle*:

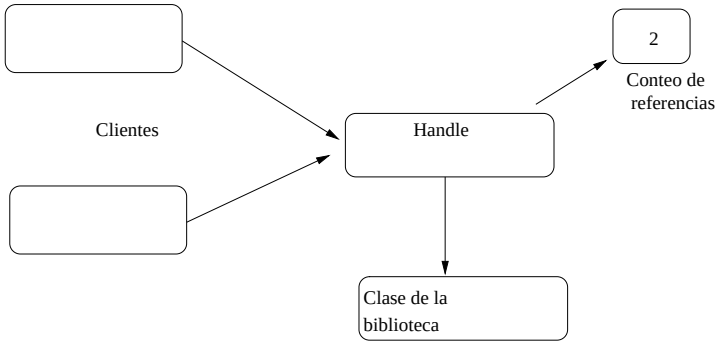


> Sin embargo, no es aconsejable añadir un conteo de referencias a la abstracción dentro de una biblioteca, ya que en ella sólo se tiene código objeto y un archivo *header*. Se puede resolver este problema con un nivel más de indirección:



Sin embargo, esto añade un ciclo a cada referencia, y esto puede ser costoso.

Por lo tanto **Asocie un conteo de compartición y un** *body* **compartido por separado con cada instancia de una abstracción común** *handle***:**



Ahora es posible accesar el *body* con un solo nivel de indirección, mientras se tiene una sola indirección para el conteo.

Los *handles* resultan algo más costosos de copiar que en el patrón *Counted Body*, la fragmentación de la memoria puede incrementarse, y el tiempo de construcción inicial puede ser mayor debido a la necesidad de alojar varios bloques en memoria.

5.1.2. Patrones de Diseño (*Design Patterns*)

Bridge es un Patrón de Diseño. Los Patrones de Diseño hicieron su entrada con el libro de Gamma et al. [23]. Los Patrones de Diseño están a un nivel más amplio que los modismos. Sus secciones son independientes del lenguaje de programación, de modo que se les considera prácticas generales de diseño para las clases comunes de problemas de software.

Los Patrones de Diseño de Gamma et al. [23] capturan las prácticas de diseño Orientado a Objetos, independientemente de un lenguaje de programación particular. Se les considera microarquitecturas: estructuras mayores que los objetos, pero no sean lo suficientemente grandes como para ser la organización básica a nivel de un sistema de software.

5.1.3. Patrones Arquitectónicos (*Architectural Patterns*)

También es posible encontrar patrones al nivel de sistemas de software. Un Patrón Arquitectónico es aquél que describe la estructura y funcionamiento básicos de un sistema de software, como un todo.

Por otro lado, un *framework* es el cuerpo parcialmente completo de un código diseñado para extenderse para una aplicación particular.Hay una relación cercana entre Patrones Arquitectónicos y *frameworks*. La mayoría de estos últimos se construyen a partir de su descripción como Patrones Arquitectónicos que conjuntan partes y mecanismos del sistema de software.

Un ejemplo de un Patrón Arquitectónico puede presentarse mediante un *framework* cuyo autor, Stephen Edwards llamó *Streams*:

> **Problema:** Este patrón permite a los diseñadores concentrarse en el flujo de datos de una pieza compleja de software sin preocuparse en las técnicas que los componentes individuales usen para distribuir la carga computacional.
>
> **Contexto:** El patrón se aplica cuando se utilizan la mayoría de los lenguajes imperativos. Los *streams* son más efectivos cuando la arquitectura de un sub-sistema de software se captura mejor mediante notar el flujo de datos en él. Así, los *streams* funcionan naturalmente en una organización *pipes and filters*, como modelo conceptual de la operación del programa.

Fuerzas:

- Los mecanismos de comunicación que serán usados entre componentes o sub-sistemas pueden tener un impacto substancial tanto en mantenibilidad como en adaptabilidad.
- Componentes diferentes pueden escoger estrategias de procesamiento diferentes.
- Lograr la composición de componentes independientes o semi-independientes es crítico para la mantenibilidad y adaptabilidad.

Solución: Modele cada componente en el sistema como un flujo de datos.

Otro ejemplo es el patrón *Client/Server* [39]:

Problema: Un *framework* cliente/servidor Orientado a Objetos, para comunicarse con un sistema de *hosts* reducido a sus más elementales términos, debe permitir a su usuario localizar objetos de su interés, examinarlos, y cambiarlos (o crearlos).

Solución: El patrón *Client/Server* es una agregación de los patrones *Searching*, *Viewing* y *Updating/Creating*. Un *framework* exitoso depende de la calidad con la cual estos tres patrones se implementen, que a la vez depende recursivamente de la calidad con la cual los patrones que los constituyen se implementen.

Este patrón se construye a partir de otros patrones que pueden hallarse en la misma referencia.

5.1.4. Problemas con los tres niveles

Clasificar a los Patrones de Software en los tres niveles de abstracción de modismos, patrones de diseño o patrones arquitectónicos es arbitrario, en el sentido de que la abstracción se realiza a lo largo de un contínuo sin límites claros. Esto hace a la clasificación muy subjetiva.

Pero también hay una cuestión más de fondo. Muchos patrones trascienden los tres niveles de arquitectura. Considérese el patrón *Model-View-Controller*, que comenzón como un artefacto idiomático en Smalltalk. Conforme se fue comprendiendo mejor, se convirtió en un elemento de diseño Orientado a Objetos para las interfaces de usuario. Buschmann et al. [12] lo presentan como un patrón arquitectónico porque representa el principio primario de estructuración al nivel más alto de varios sistemas. Los patrones tan amplios son difíciles de categorizar de acuerdo con esta taxonomía de tres niveles.

5.2. Otras aproximaciones al Escalamiento

Los autores de patrones han utilizado otras técnicas para organizar patrones. Una aproximación común ha sido utilizar "patrones paraguas" que conjuntan múltiples otros patrones. Los patrones más pequeños pueden ser disjuntos, siendo el patrón paraguas la única relación entre ellos.

5.3. Antipatrones (*Anti-patterns*)

Los Antipatrones (*Antipatterns*) son una forma de literatura escrita en forma de patrón de prácticas que no funcionan o que son destructivas. Los antipatrones fueron independientemente desarrollados por Sam Adams, Andrew Koenig y James Coplien. Muchos antipatrones documentan los razonamientos hechos por diseñadores inexpertos en la sección de Fuerzas. Considérese por ejemplo, el antipatrón *Egalitarian Compensation*:

> **Nombre:** *Egalitarian Compensation*
>
> **Problema:** Proveer motivación apropiada para el éxito.
>
> **Contexto:** Una comunidad de desarrolladores que enfrentan horarios nocturnos en un mercado de altas ganancias.
>
> **Fuerzas:**
>
> - Los premios motivan a aquéllos que los reciben, pero pueden frustrar a sus compañeros.
> - Se desea fomentar la cohesión en el equipo, construir identidad en el mismo y un comportamiento de equipo en general.
>
> **Solución supuesta:** Todo el equipo (como unidad social) debe recibir premios comparables, evitando desmotivar a los individuos que pueden medir su valor mediante su salario en relación con sus compañeros.
>
> **Contexto resultante:** Una organización donde las personas se sienten aceptadas como compañeros porque, financieramente, lo son. Sin embargo, aun emergerán líderes y en consencuencia habrá un distribución inequitativa del trabajo; tal distribución no es conmensurable con una compensación.
>
> Las personas averiguan esto, y pierden su motivación para ser mejores. El patrón tiene el efecto opuesto.
>
> **Razonamiento de diseño:** Nótese que este patrón difiere de *Compensate Success* [16] en sólo un detalle: que quienes contribuyen de forma

> sobresaliente no reciben premios excepcionales. Premios importantes para sólo algunos individuos puede desmotivar a sus compañeros, pero recompensar a un equipo permite remover el aspecto personal de este problema, y ayuda a establecer el mecanismo como un motivador.

Este patrón puede verse como la contraparte de *Compensate Success* [16], pero no todo patrón puede tener un antipatrón.

Los antipatrones no proveen una resolución de fuerzas como los patrones, y resultan peligrosos como herramientas de enseñanza; la pedagogía se construye a partir de ejemplos positivos que el estudiante puede recordar, en lugar de ejemplos negativos. Los antipatrones pueden ser adecuados como herramientas de diagóstico para entender los problemas de un sistema.

5.4. Meta-patrones (*Meta-patterns*)

En el libro de Wolfgang Pree [31] se introduce el término meta-patrón (*meta-pattern*). Podría considerarse como un error de nombre: no hay mucho "más allá" de los patrones, aun cuando son bloques básicos de construcción con los cuales se pueden construir otros patrones. En particular, se pueden considerar a los meta-patrones de Pree como generalizaciones de aspectos estructurales clave (de la implementación) de muchos patrones de Gamma et al. [23].

Un aspecto interesante del trabajo de Pree es el concepto de "punto caliente" (*hotspot*), que son los puntos dentro de un *framework* donde tiene lugar la parametrización. Pree ve a los *hotspots* como lugares cruciales donde aplicar patrones.

Para una revisión amplia y provocativa de meta-patrones en general, se puede consultar a Volk [38].

5.5. Patrones y Estrategias

El libro *Object Models: Strategies, Patterns and Applications* de Coad, North y Mayfield [13] usa el término patrón para referirse a *templates* para el diseño Orientado a Objetos. Sin embargo, el libro no tiene mucha relación con la disciplina de patrones, y los patrones no cuentan con razonamientos de diseño efectivos ni criterios para la aplicación que son cuestiones centrales para la Comunidad de Patrones.

Para una revisión de este trabajo que explora más allá su relación con patrones, véase las opiniones de Berczuk [8].

6. Pragmatismo en Patrones

El presente documento se enfoca en principios, filosofías y valores que soportan a los patrones. Antes de adentrarse más hacia el tema de generatividad y el sistema de valores de patrones, es oportuno mencionar algunas de las técnicas administrativas de la comunidad. ¿Qué debe esperarse que los patrones hagan por uno?, ¿y qué se espera que uno haga por los patrones?

6.1. El Objetivo de los Patrones

Reducción de costos, satisfacción del cliente, productividad, y reducción del intervalo de desarrollo son tan solo algunos de las muchas metas en el desarrollo de software. Los patrones contribuyen indirectamente para lograr muchas de estas metas:

- *Productividad.* Mediante proveer experiencia en un dominio, los patrones sirven como "atajos", reduciendo el intervalo de descubrimiento de muchas estructuras de diseño. El descubrimiento incluye las actividades de un diseñador para averiguar cómo el sistema actual funciona, y sirve como base para cambios de mantenimiento.

  De manera importante, los patrones evitan el re-trabajo que proviene de decisiones inexpertas de diseño. Como un ejemplo, los programadores que no entienden modismos como el *Counted Body* permanecerán un largo tiempo convergiendo a una solución, o emplearán soluciones que son menos mantenibles, o simplemente no funcionan.

- *Intervalo de desarrollo.* Muchos Patrones de Software son una forma de reuso a nivel diseño. Los patrones pueden reducir la cantidad de tiempo requerido para construir las estructuras de solución, ya que permite a los diseñadores el uso de trozos de diseño que son más grandes que las funciones o los objetos.

  Los patrones también proveen de guías en la estructura de sistemas existentes, facilitando al diseñador inexperto entender y navegar através del software existente.

- *Costos.* La reducción de costos sigue una ruta directa a partir de la reducción del intervalo de desarrollo.

- *Satisfacción del cliente.* La satisfacción del cliente es por mucho una consecuencia de los demás factores.

Recientemente, un gran proyecto en AT&T documentó sus patrones arquitectónicos clave como parte de un proyecto de documentación de arquitecturas. Esto patrones probaron ser de un valor estratégico para el proyecto siguiente, tanto porque educaron a la nueva comunidad de desarrollo acerca de *qué* necesitaba hacer el nuevo sistema, como del *cómo* lograr la funcionalidad requerida.

Ciertamente, la distancia es variable. Los patrones en sí no prometen ninguno de estos beneficios. Son una herramienta que amplifica la experiencia humana y el poder de tecnologías aplicables, prácticas de una buena administración, y prácticas y oportunidades de negocio.

6.2. Lo que no pueden hacer los Patrones

- Ya que los patrones capturan experiencia, algunos patrones importantes para dominios nuevos de negocios normalmente no se encuentran disponibles para el desarrollo de un primer sistema. Se puede discutir que muchos patrones importantes trascienden dominios, y que nuevos dominios de negocio puede construirse a partir de los patrones de otros dominios. Esto es hasta cierto punto cierto, particularmente para patrones generales como los patrones de diseño Orientados a Objetos, patrones para procesamiento distribuido, patrones cliente/servidor, y algunos similares. Pero cada nuevo dominio requiere de nuevos patrones que no pueden desarrollarse fuera del dominio. Tales patrones emergerán eventualmente con la experiencia en el dominio.

- Los patrones son guías para las personas, no para las máquinas. No generan código, y no se les puede encasillar en una herramienta CASE. Son un tipo de literatura que ayuda a las personas en los procesos de toma de decisiones. Los patrones no deben, no pueden, y no reemplazarán al diseñador humano.

- Aun cuando los patrones permiten el uso del conocimiento de expertos reconocidos, no pueden convertir a cualquiera en un experto. Los patrones sirven en tal caso para prevenir errores comunes de diseñadores inexpertos, invitándoles a considerar nuevos principios. Los patrones en sí mismos no pueden inculcar el sentido intuitivo de la estética de los verdaderos grandes diseñadores; esto es intuitivo, casi instintivo. Quizá la experiencia a largo plazo con los patrones pueda ayudar a los nuevos diseñadores a evolucionar al nivel experto, del mismo modo

que cualquier experiencia de largo plazo puede guiar a las personas correctas a ser expertos.

6.3. Un Programa de Patrones

Los patrones no son un método de diseño, pero pueden ser utilizados como un elemento adjunto a cualquier método de diseño. Los patrones no requieren de grandes ajustes y cambios en una organización o proceso, pero si dependen de una cultura de apoyo. Esta cultura incluye el entrenamiento en patrones, la mineria de patrones, la publicación de patrones y la aplicación de patrones.

6.3.1. Entrenamiento en Patrones

El entrenamiento se basa en describir a quienes desean incorporarse a la Comunidad de Patrones algo de la historia de patrones, el problema que los patrones intentan resolver, cómo capturar los logros intelectuales de una organización en un cuerpo de literatura (particularmente los relacionados con la arquitectura). Comúnmente, se introducen las formas de los patrones y sus partes (problema, contexto, solución, contexto resultante, etc.).

Para explicar a los diseñadores de software patrones arquitectónicos, frecuentemente es necesario introducirlos a algunas bases de arquitectura. Se enfatiza que la arquitectura se refiere a las relaciones entre partes, y no solamente a interfaces y cohesión. También, se hace énfasis en que la arquitectura sirve a necesidades humanas: para liberar a los equipos de desarrollo para trabajar tan independientemente como sea posible, desacoplados de otros grupos mediante líneas de fuerza arquitectónicas.

Pero el aspecto clave del entrenamiento se realiza cuando los estudiantes escriben patrones. Se les solicita pensar en una cosa que saben que es importante pero que no es ampliamente apreciada. Se les pide que trabajen con las fuerzas, y expongan su esfuerzo en forma de patrón. Este ejercicio tiene varios propósitos: primero, muestra la importancia de las fuerzas al estudiante; segundo, provee al estudiante de un recorrido alrededor de los patrones, de modo que puedan leerlos más instruidamente; tercero, ayuda al estudiante a darse cuenta que todos tienen algo importante con que contribuir, lo que combate la noción de que los patrones deben ser esotéricos o rimbombantes. Este ejercicio permite introducir a aspectos importantes del sistema de valores de patrones.

6.3.2. Minería de Patrones (*Pattern Mining*)

Normalmente, la mayoría de los desarrolladores utilizan patrones comunes de la industria, como podrían ser los patrones de Gamma et al. [23]. Sin embargo, existen muchos otros Patrones de Software fuera de este catálogo. Todo desarrollador u organización han descubierto patrones que se han refinado durante años de evolución, y que son aspectos importantes que se desean encontrar y documentar.

Un ejercicio de minería de patrones debe enfocarse en los aciertos intelectuales (o de negocio) claves. Pregunte a la empresa: ¿cuáles son las estrategias técnicas clave que han hecho grande a la compañía (disponibilidad de productos, retorno de inversiones, desempeño consistente, por ejemplo)? Acérquese a los expertos en estas áreas y pregúnteles las particularidades; muchas de sus respuestas serán patrones, y muchas otras puede transformarse fácilmente en patrones. Esto puede sonar presuntuoso, pero intentarlo ha resultado para muchas organizaciones que usan patrones por primera vez. El ejercicio de búsqueda del conocimiento es similar al utilizado por "ingenieros del conocimiento", y está lleno de los mismos problemas. Esto se soluciona mediante emplear expertos jóvenes como intérpretes durante las entrevistas.

Se podría tambien tener una aproximación orientada de abajo hacia arriba para la minería de patrones. Primero, se identifican las áreas que hacen falta en la organización. Se busca a quienes saben cómo manejar esas áreas adecuadamente, y se obtienen patrones de ellos. Por ejemplo, es posible que un solo administrador de proyecto en una organización sea el único que cumple con su planeación consistentemente, de modo que es posible obtener de esta persona sus patrones, a fin de averiguar qué le permite tener éxito donde otros fallan.

Es crucial obtener patrones de expertos, organizaciones o productos con un largo historial de registros comprobados.

6.3.3. Publicación de Patrones

Los patrones se publican tanto como documentos HTML en la red como por publicaciones convencionales. No hay mucho que decir sobre la publicación de patrones excepto que los hipermedios parecen ser particularmente valiosos como ayuda en la navegación, especialmente en lenguajes de patrones.

Más importante que el formato de publicación es el proceso de revisión que lleva a la publicación. Los buenos patrones son literatura "atemporal". "Atemporal" se refiere a la literatura con amplio atractivo que es fácilmente entendible, y que captura decisiones de diseño que han sido exitosas durante un largo periodo de tiempo.

Un aspecto importante del proceso de publicación de patrones es la revisión. Una buena revisión puede ayudar a asegurar que el patrón es legible, que tiene impacto en su audiencia, y que hace su labor. Los patrones son una forma de literatura, y merecen algo más que una revisión de diseño técnica que se hace a la mayoría de la documentación de software. El método de revisión de patrones aceptado por la comunidad ha sido originalmente propuesto por Richard Gabriel, y adaptado en las conferencias de patrones en forma de "talleres de escritores" (*writers' workshops*).

Los participantes en un taller de escritores son todos autores de patrones. El autor lee una parte de su patrón o patrones que considera esencial, y en seguida, guarda silencio, volviéndose "invisible" durante la sesión. Uno de los revisores (otro autor dentro del taller) intenta resumir el patrón. Se da entonces una discusión abierta sobre porqué se considera que el patrón funciona y todo aquello positivo que se puede observar. En seguida, los revisores pueden hacer sugerencias para mejorar el patrón, respecto a su contenido, forma, utilización, etc., tratando siempre de ser tan positivamente propositivos como sea posible. Finalmente, el autor "regresa" a la sesión, y puede pedir la aclaración de algunos puntos por parte de los revisores. Toda la sesión toma común y aproximadamente un lapso de una hora.

6.3.4. Aplicación de Patrones

Una vez que los patrones han sido publicados, pueden utilizarse como material de referencia por la comunidad de diseño de software. Sin embargo, en lugar de ser un "oráculo" para la resolución de problemas que surgen durante el diseño de software, es mejor considerar a los patrones como un tutorial para los diseñadores del dominio específico. En general, se espera que los diseñadores revisen superficialmente todos los catálogos de patrones de su dominio de interés al menos una vez. Conforme los problemas de diseño surgen, los diseñadores pueden volver a los patrones para una guía más detallada.

Una exposición y experiencia más amplia ayudan a refinar los patrones através del tiempo, mejorando su contexto y aclarando fuerzas y razona-

mientos de diseño. Un buen patrón es literatura viva, que va madurando con los años.

6.3.5. ¿Y de aquí, para dónde?

Esta sección ha sido expresamente acortada parcialmente debido a que resulta difícil poner por escrito los pasos correctos o el programa comprobado para lograr éxito con patrones. Cada cultura debe ajustar los patrones para sus necesidades. Mucho del proceso se va aprendiendo con experiencia. Aprender de otras experiencias puede ser también útil; como referencia, léase el artículo escrito por Beck et al. [6].

7. Generatividad

La mayoría de las estrategias de resolución de problemas intentan atacar directamente al problema. Al hacerlo, frecuentemente se ataca sólamente a los síntomas, dejando el problema de fondo sin resolución. Alexander entendió que las buenas soluciones a problemas arquitectónicos van al menos un nivel más a fondo. Las estructuras de un patrón no son en sí mismas soluciones, sino mas bien, *generan* soluciones. Los patrones que funcionan de esta manera se les llama *patrones generativos*. Un patrón generativo es un medio para permitir que el problema se resuelva a sí mismo en el tiempo [4]:

> 9. Esta calidad en edificios y poblaciones no puede hacerse, sino solo generarse indirectamente por las acciones ordinarias de las personas, tal y como una flor no puede hacerse, sino generarse de una semilla.

Más adelante, menciona que [4]:

> Un lenguaje ordinario como el inglés es un sistema que nos permite crear una infinita variedad de combinaciones unidimensionales de palabras, llamadas enunciados... Un lenguaje de patrones es un sistema que permite a sus usuarios crear una infinita variedad de combinaciones tridimensionales de patrones, a los que llamamos edificios, jardines, poblaciones...
>
> Así, como es el caso de los lenguajes naturales, el lenguaje de patrones es generativo. No sólo nos dice las reglas de composición, sino que nos muestra cómo construir tantas composiciones como deseemos, las cuales satisfagan las reglas.

Esta característica de generatividad es un aspecto importante de la calidad que Alexander busca en sus trabajos. Esta es tan elusiva que el mismo Alexander le llama "Calidad sin Nombre" (*Quality Without a Name*), y que se describe más adelante. Otras fuentes de filosofía oriental tambien mencionan este tipo de calidad [35]:

> ¿Qué, exactamente, significa decir que las estructuras generan patrones particulares de comportamiento? ...
>
> ... una característica fundamental de los sistemas humanos complejos... [es que] la causa y el efecto no se encuentran cercanos en tiempo y espacio. Por efectos, me refiero a los síntomas obvios que indican que hay problemas: abuso de drogas, desempleo, niños famélicos, ordenes que caen y ganancias que crecen. Por causas, me refiero a la interacción del sistema base que es el mayor responsable de generar los síntomas, y el cual, si se le reconoce, puede guiar a los cambios que producen mejoras a la larga. ¿Porqué es este un problema? porque la mayoría de nosotros suponemos que son [problemas] de la mayoría la mayor parte del tiempo, y que causa y efecto están cercanos en tiempo y espacio.

¿Porqué es importante la generatividad? Primero, como lo menciona Seng, la mayoría de los problemas reales van más allá de la superficie de sus síntomas, y se tiene que solucionar la mayoría de los problemas interesantes con comportamiento emergente. Segundo, un buen patrón es fruto del trabajo duro e intensa revisión y refinamiento. Los problemas simples pueden solucionarse mediante reglas simples, ya que las soluciones son más directas u obvias que las que encontramos en soluciones generativas. Ese es el objetivo de las formas de los patrones: capturar la atención del lector en soluciones generativas; entender los principios y valores para soluciones y comportamientos emergentes a largo plazo. Los patrones van más allá de una compostura rápida.

Los lenguajes de patrones, como un todo, pueden exhibir una generatividad de tipo *gestalt*: cada patrón del lenguaje CHECKS [19] resuelve un problema, pero el lenguaje como un todo resuelve un problema de ingeniería de software mucho mas amplio que el que se soluciona por un patrón aislado. Esta sorpresiva solución es una forma de generatividad. Otro ejemplo es *Caterpillar's fate* [25], que tiene la misma propiedad.

8. El Sistema de Valores de Patrones

Los Patrones de Software normalmente tienen un componente técnico fuerte, que deseablemente ha pasado por el proceso que se ha discutido, todo controlado por los principios que también se han mencionado. Estos principios incluyen la importancia del diseño y arquitectura, particularmente al extenderse más allá de la partición modular, principios de forma y organización, y otros principios profundos como la generatividad.

Muchos de estos principios obtinen su motivación de un sistema de valores en un nivel aún más profundo. En esta sección, se explora tal sistema de valores. Esta es una sección larga por dos razones: primero, este material se entiende o discute poco dentro de la comunidad de desarrollo de software, por lo que este documento provee de una oportunidad para tomar conciencia acerca de los patrones; segundo, este material es particularmente importante. Los valores que se mencionan aquí son el soporte de los principios, y los principios dan apoyo a las metas del desarrollo de software (que son similares en cualquier empresa humana).

Muchos de los valores provienen originalmente del trabajo de Alexander. Aun cuando muchos de los principios que Alexander propone en la arquitectura de edificios no se traducen directamente en la arquitectura de software, los valores humanos esenciales sí lo hacen. Estos valores provienen de la Comunidad de Patrones de software original, y cada vez se ligan menos con los valores de Alexander. Sin embargo, se puede encontrar estos valores en el ambiente de las conferencias de patrones y en mucha de la literatura de patrones. De hecho, son parte de la cultura emergente de patrones.

La pregunta es: ¿quienes usan patrones se adhieren a todos los detalles de este sistema de valores? Evidentemente, no. Muchos usuarios de patrones trabajan con la literatura existente simplemente como referencias de diseño. Pero los valores son una característica visible de la cultura de patrones, presente en talleres, conferencias y otras reuniones de autores, editores, revisores e investigadores de patrones, que conforman la línea principal de la disciplina de patrones.

8.1. La Calidad sin Nombre (*Quality Without a Name*)

Alexander busca la "Calidad sin Nombre" (*Quality Without a Name* o simplemente QWAN) en los patrones que captura en cuartos, edificios y poblaciones generados por esos patrones. En el dominio del software, la

Comunidad de Patrones busca una forma similar de calidad en el software que se construye con patrones.

¿Qué es QWAN? Es aquella sensación de satisfacción de quien crea algo que sabe está bien hecho. Muchos programadores han tenido tal sensación, particularmente cuando programan un módulo o sistema satisfactorio, en el código que simplemente se siente que está bien. Claro, definir tal calidad o nombrarla siquiera hace que se gaste esfuerzo, y se pierda de vista su importancia. Sin embargo, como tema recurrente en la literatura de patrones se tratan aspectos de lo que tal QWAN puede ser.

Uno de ellos, que es muy próximo a los objetivos de Alexander en arquitectura, es servir a necesidades humanas. Frecuentemente, se pasa por alto que todo software debe servir a necesidades humanas; la forma de los patrones, particularmente en las secciones de Fuerzas y Contexto resultante, son una oportunidad para llamar la atención a estas necesidades. Esto se retoma más adelante, cuando se mencionan las cuestiones estéticas, pero de hecho, va más allá de solo la estética, a la cuestión más amplia de comodidad humana.

La generatividad es otra parte importante de la QWAN, ya que refleja un entendimiento profundo del problema, y promueve una solución igualmente profunda.

Alexander hace la observación de que la QWAN tiene un aspecto triste, pues nos recuerda que nada es permanente. Reconocer que construir en un ambiente hoy sabiendo que ese ambiente cambiará en el tiempo, ayuda a considera la evolución dentro de la perspectiva de diseño. Esta evolución y conciencia del tiempo son elementos centrales de la QWAN.

La QWAN es ciertamente subjetiva, y puede significar algo diferente para cada diseñador. En el resto de esta sección se exploran otros valores que apoyan los límites de la QWAN, valores que son importantes dentro de la cultura de patrones.

8.2. Cosas Reales

Los patrones son acerca de cosas reales. Capturan práctica comprobada, en lugar de postulados, teorías, modelos o técnicas que se supone podrían funcionar.

Kent Beck cuenta una historia del taller de 1987, en donde tuvieron

lugar algunas discusiones iniciales sobre Patrones de Software [7]. Comenta que los patrones que presentó ofrecían principios de diseño desarrollados a partir de un éxito repetitivo al aplicarlos, principios que podían ser más ampliamente aplicados si las personas supieran de ellos. Otra persona que atendió al taller habló sobre la premisa de que aun cuando no hubiera una amplia experiencia con una técnica en particular, todo estaría bien si tan solo los ingenieros de software comenzaran a utilizar un conjunto particular de técnicas que llamó "las mejores prácticas", desarrolladas únicamente en la investigación académica de ese tiempo.

La Comunidad de Patrones ha tomado la posición de Beck, considerando lo que las personas hacen que funciona, y no predicando hacer algo en base solo a intuiciones, argumentos o aspiraciones propias. Esto no significa que la Comunidad de Patrones no tome en cuenta las contribuciones de la investigación académica, sino que toma en cuenta aquellas aportaciones apegadas a la realidad, y no a lo que se quisiera fuera la realidad.

Un buen patrón se basa en ejemplos reales, normalmente expuestos en la sección de Razonamiento de diseño; algunas otras formas de patrones tienen secciones propiamente para ejemplos. Como una pauta aceptada, un patrón se considera como tal si se presentan tres ejemplos que muestren tres implementaciones diferentes y exitosas.

A este respecto, la Comunidad de Patrones hace suya una afirmación que se atribuye a E.W. Dijkstra: "La abstracción prematura es el origen de todo mal" [2]. La sección de Solución de cualquier patrón debe enfocarse en cosas reales. Una solución debe instruir al lector para hacer algo específico y concreto.

Aun asi, lo concreto de la realidad parece no ser consistente con la abstracción. Sin embargo, abstracto no quiere decir vago. Una abstracción, aun cuando le haga falta detalle (por definición), puede ser precisa, aguda, e instructiva. Se puede hacer una analogía con una parte del diseño de software. Aun cuando la interface abstracta de la clase `String` distancía al usuario de la implementación de la clase, la interface es todo lo visible para el usuario. Junto con la documentación que explica su uso, la interface abstracta es suficiente para un uso efectivo. Un buen patrón debe comportarse del mismo modo: ser suficientemente abstracto para ser implementado un número indefinido de veces (sin necesidad de ser exactamente el mismo dos veces) y sin embargo, ser lo suficientemente específico para decirle al usuario qué hacer.

[2] *" Premature abstraction is the root of all evil"*

Richard Gabriel aconseja buscar más allá de la abstracción, en lo que conlleva dentro; es ahí donde muchos patrones importantes se albergan [22]:

> ¿Y qué hay acerca de la geometría? Alexander siempre vuelve a este punto. Y una de sus preguntas clave es esta: ¿qué es lo que dicta que una geometría contenga la QWAN?, ¿qué es esa cosa que falsamente se llama simplicidad?
>
> ¿Y qué corresponde a la geometría para nosotros?
>
> Creo que es el código en sí mismo. Muchos hablan de la necesidad de interfaces excelentes y los beneficios de separar la interface de la implementación, de modo que la implementación pueda variar. Pero pocas personas hablan seriamente acerca de la calidad de código en sí mismo. De hecho, muchos teóricos están muy dispuestos a amontonarlo en la categoría de las cosas que mejor no se discuten, algo que debe estar oculto de la vista de modo que pueda cambiarse en privado. Pero considérese la pregunta anterior de Alexander: la calidad proviene en casi la misma medida del arte y creatividad del constructor, cuyas manos más directamente forman la geometría que da a un edificio su calidad y carácter. ¿No es el constructor el propio codificador? ¿Y no es de la metodología de software clásica el poner el diseño en manos de analistas y diseñadores y poner la codificación en las manos de programadores y codificadores, a veces aquéllos a los que se les paga los más bajos salarios para hacer el trabajo menos importante?

8.3. Restricciones que liberan

La forma de los patrones restringe al escritor de patrones. El remover la preocupación de cómo se presenta un patrón libera a los solucionadores de problemas para enfocarse en otras cosas. Se considera que la forma de los patrones es mejor que simple lenguaje natural no-estructurado porque ofrece pautas. Pero también se considera un formato mejor que, por ejemplo, SGML. Las formas SGML no entienden de semática ni la relación entre las secciones del patrón, así que no pueden ofrecer una suficiente restricción. El principal beneficio de las formas SGML, y en particular de HTML, un derivado de SGML, es su amplia aceptación como un formato de publicación portable.

Un patrón terminado restringe al desarrollador. Un buen patrón ayuda al desarrollador a enfocarse en el problema mediante considerar fuerzas importantes. Esto previene al desarrollador de explorar innecesariamente callejones sin salida. Los desarrolladores de cualquier manera deben aplicar su

conocimiento y experiencia de diseño. Pero un buen patrón deja suficiente espacio para la adaptación creativa.

Alexander lo comenta así [4]:

> Las reglas del inglés promueven la creatividad porque nos ahorran tener que preocuparnos de combinaciones sin sentido de palabras... Un lenguaje de patrones hace lo mismo.

8.4. Arquitectura Participativa

Alexander creyó que las personas deben participar en el diseño de sus moradas en lugar de dejarlo a un arquitecto profesional. La arquitectura debe resolver necesidades humanas, y es mas problemático para un arquitecto asimilar la necesidad de un individuo dueño de preferencias personales y preferencias que surgen del contexto social, de lo que sería permitir que las personas derivaran sus diseños ellas mismas.

Este mismo espíritu se encuentra en la fundación de la Comunidad de Patrones de Software. Esta fue una de las primeras estrategias que Kent Beck y Ward Cunningham emplearon para sus patrones para diseño de interfaces gráficas: permitir al usuario del sistema diseñar su propia interface, igualmente de como Alexander propone que los ocupantes del edificio diseñen su edificio.

Este principio se toca en un patrón organizacional de Coplien [16]:

> **Nombre:** *Architect Also Implements...*
>
> **Problema:** Preservar la visión arquitectónica mediante la implementación.
>
> **Contexto:** Una organización de desarrolladores que requiere una dirección estratégica y técnica.
>
> **Fuerzas:** El control totalitario es visto por la mayoría de los equipos de desarrolladores como una medida draconiana. La información pertinente debe fluir a través de los roles pertinentes.
>
> **Solución:** Más allá de aconsejar y comunicarse con los desarrolladores, los arquitectos deben participar también en la implementación.
>
> **Contexto resultante:** Una organización de desarrolladores que percibe ventajas de la guía de los arquitectos, y que puede directamente avalar en sí misma experiencia arquitectónica.

Razonamiento de diseño: La importancia de este patrón surge de una experiencia en un proyecto. El equipo de desarrollo se componía de dos grupos separados geográficamente y con poca comunicación entre ellos. Aun cuando las responsabilidades generales y los roles estaban identificados, un grupo tenía la idea que los arquitectos también implementarían código; el otro no.

Rybczynski menciona [33]:

"Sería conveniente si la arquitectura pudiera definirse como cualquier edificio diseñado por un arquitecto. Pero ¿quién es un arquitecto? Aun cuando la *Academie Royale d'Architecture* de París fue fundada en 1671, las escuelas formales de arquitectura no aparecieron sino hasta el siglo XIX. El famoso *Cole des Beaux-Arts* fue fundado en 1816; la primera escuela de habla inglesa se funda en Londres, en 1847; el primer programa norteamericano, en el MIT, se estableción en 1868. Y a pesar de la existencia de escuelas profesionales, por mucho tiempo la relación entre la academia y la práctica permaneció ambigua. Es posible ser un arquitecto sin un grado universitario, y en algunos paises como Suiza, los arquitectos no tienen un monopolio legal sobre las construcciones. Por siglos, las diferencias entre maestros albañiles, albañiles, jornaleros, diletantes, hábiles amateurs y arquitectos estaba mal definida. Las grandes construcciones del Renacimiento, por ejemplo, fueron diseñados por una variedad de no-arquitectos. Brunellesehi tuvo como formación la de herrero; Miguel Angel era escultor; Leonardo era pintor; Alberti era abogado. Solo Bramante, que también se formó como pintor, había estudiado construcción formalmente. Todos estos hombre son conocidos como arquitectos porque, entre otras cosas, crearon arquitectura (una tautología que no explica nada)".

Así como este patrón busca en la arquitectura clásica motivos para animar a los arquitectos a ensuciarse las manos, la disciplina de patrones llama a los arquitectos de software para involucrarse en la implementación de sus diseños. Muchos de los métodos de desarrollo de software sufren de la falta de transferencia en sus diseños, lo que casi siempre asegura que el diseño original se pierda.

Cuando se capturan patrones de diseño y arquitectura, es importante capturarlos de forma que cualquier programador pueda utilizarlos. Esto es consistente con el objetivo de los patrones de capturar "cosas reales", y para preservar la dignidad de los programadores.

8.5. La Dignidad del Programador

Dado que los patrones son acerca de cosas reales, normalmente se refieren a las preocupaciones diarias del programador. Esto llama la atención a la elegancia fallida y el poder de las técnicas que dan forma al artefacto que se entrega al cliente. Tal énfasis rompe con el respeto desproporcionado que la disciplina de patrones tiene respecto a arquitectura y métodos formales. La mayoría de las culturas de desarrollo de software dan relevancia al arquitecto y al "metodólogo", pero mantienen a aquéllos que realmente generan el producto final muy abajo en la escala social. Estos valores parecen reflejar la era industrial y su estructura social en fábricas, donde los programadores son tan solo los trabajadores a lo largo de las líneas de ensamble. Muchas de esas culturas ven a sus programadores como poco mas que trabajadores manuales.

En la Comunidad de Patrones se busca reconocer la excelencia de diseñadores y programadores. Sus contribuciones dan forma a productos finales en mayor medida que aquéllas de los arquitectos.

Al mismo tiempo, la comunidad cree que los arquitectos tienen que mantenerse en contacto con la realidad de la implementación. Esto se ajusta directamente con la consideración de que los patrones se refieren a cosas reales; esto se ve reflejado en un patrón como *Architect Also Implements*.

8.6. Indiferencia por la Originalidad

Muchas técnicas y tecnologías nuevas en la computación intentan poner de manifiesto una distinción con sus predecesoras. Por ejemplo, quienes abogan por las técnicas Orientadas a Objetos frecuentemente las exaltan al compararlas con las técnicas de diseño procedural, mostrando su superioridad al resolver problemas bien conocidos.

Quienes han trabajado en la industria del software por algún tiempo pueden recordar varios ciclos en que aparece nueva tecnología, y pueden relatar que las expectativas que han emergido alrededor de la nueva tecnología siempre excedieron lo que después podía lograrse.

En la Comunidad de Patrones se busca capturar en forma de patrones las ideas comprobadas hace ya tiempo. Esto rompe con las normas culturales de la mayoría de las organizaciones de investigación y desarrollo, que premian la innovación. En la Comunidad de Patrones, en contraste, no se considera importante lo novedoso. Se tiene una indiferencia por la originalidad.

Esto no quiere decir que las soluciones novedosas deben ser rechazadas. Los patrones existentes pueden aplicarse en formas novedosas, como por ejemplo, la aplicación del patrón *Model-View-Controller* como estructura de la arquitectura de un sistema [12]. Ya que las ideas originales de hoy serán patrones el día de mañana, es importante tener siempre en cuenta los nuevos desarrollos.

Los patrones ayudan a reducir el riesgo de nuevos desarrollos mediante construir sobre fundamentos pasados que han comprobado realmente su utilidad en desarrollos exitosos anteriores. Un buen diseño balancea los patrones (en la estructura principal de la solución) con nuevas ténicas y métodos. Por ejemplo, sería altamente recomendable que el arquitecto de un sistema de telecomunicaciones tomara en cuenta lenguajes de patrones como el expuesto en la sección 3.1. Estos patrones dan forma a la estructura y mecanismos de manera que fundamentalmente apoyan las necesidades de varios negocios. Mas aun, estos patrones pueden perfectamente ser compatibles con implementaciones Orientadas a Objetos o basadas en reglas. Balancear entre nuevos y viejos patrones es tan solo un ejemplo de administración de riesgos, lo que siempre ha sido un objetivo común para los administradores de proyectos de software.

Esto pone en relevancia un detalle importante: los patrones no tienen una relación inmediata con Orientación a Objetos. Los patrones existen en las prácticas exitosas de muchas disciplinas previas; muchos de los patrones en esta introducción son de tal naturaleza. Comúnmente, estos patrones son compatibles con técnicas Orientadas a Objetos, pero no podrían ser reconocidos tan solo como técnicas Orientadas a Objetos. De hecho, los patrones funcionan bien fuera de un método de desarrollo de software, como podría ser en áreas de organización y entrenamiento, que poco tienen que ver con la Orientación a Objetos.

En algún punto, los viejos patrones tienden a desaparecer, y su sitio es ocupado por nuevos patrones. Tales cambios raramente son sutiles o incrementales; suceden cuando hay cambios fundamentales de paradigma. Los cambios de paradigma tienden a ser difíciles porque las personas tienen dificultad en dejar los viejos patrones con los que se sienten cómodos. Pero los cambios de paradigma también son poco frecuentes. Durante la mayoría del tiempo no sucede nada [40], y los principios probados y ciertos prevalecen sobre las novedades.

8.7. El Elemento Humano

Richard Gabriel resume la preocupación de la Comunidad de Patrones sobre las cuestiones humanas [22]:

> Intentamos atraer a las personas al diseño de software y al proceso de desarrollo. Creo que esta es la meta porque es la clave de la idea original de Alexander. Si se observan sus patrones, cada uno o casi todos se refieren en el contexto y las fuerzas en términos de lo que necesitan hacer las personas para vivir plenamente.

Todo software sirve a una necesidad humana a cierto nivel. Durante el diseño, tomar en cuenta estas cuestiones humanas provee de una perspectiva importante, quizá la perspectiva más importante respecto a la relevancia de las actividades de diseño y arquitectura. Esto se observa en el patrón de Gabriel *Simply Understood Code*, que se presenta más adelante. También hay un elemento humano en los patrones de la sección 3.1. Mientras que el lector casual puede pensar en estos patrones como arquitectónicos, el lector cuidadoso puede descubrir que muchos tienen poco que ver con la estructura de software, enfocándose más bien en comportamiento, necesidades y motivaciones humanas.

Muchos principios tradicionales y supuestamente técnicos contienen profundas fuerzas humanas. Buschmann y Meunier [11] incluyen cohesión y acoplamiento en su esquema de clasificación de patrones. Cohesión y acoplamiento son principios muy subjetivos y humanos del diseño de software. Al software mismo no le interesa la cohesión y el acoplamiento; si acaso, el desempeño del software podría beneficiarse de una alta cohesión que resuelva algunas ineficiencias de interfaces abstractas. Pero para los seres humanos, que sí enfocan su atención en la abstracción, la cohesión y el acoplamiento son benéficos para todos aquéllos que tengan después que dar mantenimiento al código. Las personas podrán trabajar independientemente mientras que su código esté lo suficientemente desacoplado del código de otros. Los patrones capturan no solo principios como cohesión y acoplamiento a nivel código, sino que también los incluyen en sus fuerzas y contextos resultantes, a nivel humano. Fallar a nivel humano mientras que se tiene éxito en el nivel técnico es perder el objetivo de un patrón, y particularmente respecto a los principios de cohesión y acoplamiento.

La estética, que se menciona a continuación, es también un aspecto importante del elemento humano en los patrones.

8.8. Estética

La estética del diseño mismo es un indicador de la mantenibilidad de un sistema. Un sistema que no puede ser fácilmente entendido, no evoluciona fácilmente. Un buen diseño apela a los aspectos humanos del desarrollo.

Los patrones consideran a la programación como si se tratara de una forma de literatura. La Comunidad de Patrones no ha sido, sin embargo, la primera en hacerlo; Knuth lo propone como la "programación literada" (*literate programming*) [26]. La programación para el desarrollo de software es una tarea creativa. Esto significa que no hay dos personas que resuelvan el mismo problema de la misma forma, o que la misma persona resuelva un problema dos veces de la misma manera.

La estética en la programación es importante porque se trata de un elemento humano. La estética en el código hace que el código mismo sea fácil de entender, y por ende, sea más mantenible. Considérese, por ejemplo, el siguiente patrón de Richard Gabriel [22]:

> *Simply Understood Code*
>
> Al nivel más bajo de un programa hay trozos de código. Esto son los "lugares" que deben entenderse para hacer cambios al programa de forma segura, y finalmente, entender un programa completamente requiere entender estos trozos de código.
>
> En muchas piezas de código el problema de desorientación es agudo. Las personas no tienen idea de para qué son cada uno de los componentes del código y experiementan un considerable estress mental como resultado.
>
> Suponga que escribe un trozo de código que no es tan complejo y que requiere documentación extensa, o por otro lado, no es lo suficientemente central para preocuparse por escribir la documentación, pues no vale la pena el esfuerzo, especialmente si el código es suficientemente claro por sí mismo. ¿Cómo debería ser la aproximación para escribir tal código?
>
> Las personas necesitan observar el código para entenderlo lo suficiente como para sentirse seguras de hacerle cambios. Pasar el tiempo saltando de ventana en ventana o recorriendo ventanas para arriba y para abajo, a fin de ver todas las porciones relevantes de código distrae la atención de entender el código y ganar confianza para modificarlo.
>
> Las personas pueden entender cosas más eficientemente si pueden leer-

> las en un orden de lectura natural; en general, esto es de izquierda a derecha y de arriba hacia abajo.
>
> Si el código no puede entenderse, puede accidentalmente "romperse".
>
> Por lo tanto, *arregle las partes importantes del código de manera que quepan en una sola página. Haga el código entendible para una persona leyendo de arriba hacia abajo. Que no sea necesario revisar repetidamente el código para entender cómo los datos se usan y cómo el control fluye.*
>
> Este patrón puede lograrse mediante el uso de los siguientes patrones: *Local Variables Defined and Used on One Page*, que intenta colocar las variables locales en una sola página; *Assign Variables Once*, que intenta minimizar las búsquedas en el código mediante cambiar las variables una sola vez; *Local Variables Reassigned Above their Uses*, que intenta hacer aparente el valor de las variables antes de que tal valor sea utilizado, mientras se busca de arriba hacia abajo; *Make Loops Apparent*, que ayuda a entender partes de un programa que no son lineales mientras se retiene la habilidad de buscar linearmente; y *Use Functions for Loops*, que empaqueta la estructura compleja de un ciclo que involucra varias variables de estado en trozos, cada uno de los cuales puede ser fácilmente entendido.

La parte humana de este patrón recae en el uso de términos como *seguridad*, *estress*, *cultura*, *entender*, etc. La comodidad humana emerge de la estética en el diseño.

La estética no se restringe solo al formateo de código. La estética puede encontrarse en una estructura coherente, interfaces intuitivas de módulos, o en una preocupación holística del sistema y su ambiente. Los más altos principios de la arquitectura se encuentran en la triada clásica de Vitrubio: utilidad, firmeza y gusto (ó estética).

Más aun, hay un aspecto muy obvio de la estética: la manera en que los usuarios ven al programa. Las interfaces gráficas estéticamente agradables hacen la labor de los usuarios menos pesada y más amena: los usuarios se frustran menos, son más productivos y quizá hasta puedan entretenerse con su trabajo.

Considérense los siguientes tres patrones, resumidos del lenguaje de patrones CHECKS de Ward Cunningham [19]. Estos patrones ostensivamente reflejan la estética en la interacción humano computadora:

1. *Whole Value*

   Cuando se parametriza o se cuantifica un modelo, queda un deseo intenso de expresar estos parámetros (tipo de cambio, calendario, periodos, números de teléfono) en las unidades más fundamentales de cómputo. No solamento esto ya no es necesario (esto era una práctica estándar en lenguajes con débil o nula abstracción), sino que realmente interfiere con la comunicación llana y propia entre las partes del programa y entre las partes del programa y sus usuarios. Ya que bits, números y caracteres pueden usarse para representar casi cualquier cosa, cada uno de ellos aisladamente significan casi nada.

   Por lo tanto, constrúyase valores especializados para cuantificar el modelo y úsense estos valores como argumentos en los mensajes y como unidades de entrada/salida.

2. *Exceptional Value*

   Un modelo normalmente se compone de un caso básico o abstracción que se especializa o refina para capturar la diversidad presente en el dominio. Sin embargo, habrá frecuentemente circunstancias en que la inclusión de todas las posibilidades en el dominio dentro de la jerarquía de clases es confusa, difícil o inapropiada. Es por ello que a veces es necesario extender el rango de un atributo más allá de lo que ofrece el patrón *Whole Value*. Considérese un encuestador que recolecta respuestas como "de acuerdo", "fuertemente de acuerdo", etc. Las respuestas que desafían la cuantificación, como "ilegible" o "rechazado", se representan mejor fuera del rango de valores, no importa lo difuso que sean. Sin embargo, las estructuras de un modelo deben guardar un lugar para este tipo de datos, ya que pueden aparecer más tarde. De hecho, tales datos son imposibles de evitar durante la creación (entrada de datos) de todos excepto los más sencillos de los modelos.

   Por lo tanto, úsese uno o más valores distinguidos para representar las circunstancias excepcionales. Los valores excepcionales deben ya sea aceptar todos los mensajes, respondiendo la mayoría de ellos con otro valor excepcional, o rechazar todos los mensajes.

3. *Meaningless Behavior*

   Dado que el patrón *Whole Value* se utiliza para cuantificar la lógica del dominio exhibiendo variaciones sutiles de compotamiento y que el patrón *Exceptional Value* puede aparecer en cualquier

> cómputo, es posible que los métodos que se programan tropiecen con circunstancias que no pueden preverse. Recuérdese que las reglas de negocio se aplican de manera selectiva, y que la evolución de las prácticas del negocio pueden "darle la vuelta" a esas reglas que se deben aplicar. En los modelos se intenta expresar la lógica del negocio sin más complejidad que en su concepción original o expresión actual.
>
> Por lo tanto, escriba métodos sin preocuparse de la posible falla. Espere que los dispositivos de entrada/salida que inician el cómputo se recuperen de la falla y continúen procesando. La salida permanecerá en blanco, ya que cualquier otra salida sería un intento de conectar un significado al comportamiento sin significado. Los usuarios interpretarán las salida en blanco significando que las entradas no aplican o que no hay salidas disponibles.

El usuario se benficia de interfaces limpias e intuitivas. Algunas interfaces sufren de contar con código que intenta ayudar al usuario demasiado, al presentar ventanas de error cada cierto tiempo en que la aplicación se tropieza con un error; este patrón intenta avitar eso.

Aun cuando se desarrolla para el diseño de interfaces gráficas, este patrón beneficia de igual modo al diseñador como al usuario final. El patrón indica no preocuparse por el manejo de errores: propagar problemas hasta el final de la cadena de evaluación (el usuario) y manejar los problemas justo antes que alcancen al usuario. Los patrones comentan tanto cómo propagar errores, como cómo atraparlos en el momento justo. Esto libera al programador de la carga congnitiva de la administración de errores conforme escriben código, y deja al código a ser más fácilmente mantenible después. Sin condicionales distribuidas por el código que chequen las condiciones de error, el código es mucho mas conciso y legible que si se utilizan técnicas de fuerza bruta.

La mayoría de los Patrones de Software son acerca de arquitectura de software, y los patrones ciertamente toman como inspiración el campo más antiguo de la arquitectura de edificios. La estética es clave para una buena arquitectura. Rybczynski menciona que la arquitectura es la suma de ingeniería y cultura; esto último se acepta en raras ocasiones dentro de los métodos de diseño. El mismo ilustra su afirmación con el siguiente ejemplo [34]:

> La comunicación del significado, más que la belleza, distingue a la arquitectura de la ingeniería. Un puente debe ser sólido, funcional y atractivo; una biblioteca pública debe ser todo esto, pero además lleva

> consigo cultura. Su arquitectura define una actitud hacia la lectura y celebra un sentido de orgullo cívico. Una biblioteca es más que un almacén de libros; es una evocación construida de un ideal intelectual.

De tal modo, es necesario aquí dar un significado al software dentro de la sociedad a la que sirve, y no solamente construir el equivalente mecánico. Los sistemas con significado sirven bien a sus usuarios. Durante la construcción de un sistema con significado, el constructor del sistema y el usuario del sistema se encuentran involucrados en (algunas veces implícito) diálogo donde cada uno debe entender los patrones del otro. Los autores de software deben entender el dominio y vocabulario del usuario; el usuario del software debe entender el significado de los movimientos del "ratón" y la presión de teclas que el autor del software ha mapeado para responder al vocabulario del usuario. Los patrones capturan los aspectos sutiles de este diálogo en un alto nivel, ayudando a la comunicación entre usuario y autor. Tales detalles distinguen de sistemas meramente funcionales a los sistemas que son agradables de usarse.

8.9. Enfoque Interdisciplinario

La Comunidad de Patrones toma sus bases de un campo externo: arquitectura de edificios y planeación urbana. Los Patrones de Software se están extendiendo a los dominios de organización, proceso, entrenamiento y otras áreas fuera del enfoque tradicional de desarrollo de software. Pero se aspira a ir más allá.

Hay una brecha tradicional y frecuentemente resaltada entre los dominios de los diseñadores de software y los dominios de sus clientes. Los diseñadores de software saben acerca de estructuras de datos, algoritmos, paradigmas y métodos; los clientes entienden las prácticas de su negocio, arte o pasatiempo. Los desarrolladores de software serán capaces de complacer las necesidades de sus clientes mediante entender las necesidades de sus dominios. Algo más allá, si se puede capturar los Patrones de Software importantes que se encuentran en soluciones útiles en dominios específicos, se puede proveer de esos patrones conforme comienzan a desarrolar su propia programación. Esto es consistente con la actuación del arquitecto que emplea los patrones de las arquitecturas predominantes de una cultura, de modo que las personas puedan materialmente participar en la construcción de sus propias viviendas, iglesias, fábricas y parques. Los patrones proveen de un vocabulario para lograr un encuentro con los clientes, pero es necesario primero encontrarse con los clientes en biología, telecomunicaciones, finanzas, aeroespacial

y otras disciplinas, a fin de trabajar con ellas en el desarrollo de un vocabulario.

El encuentro con el cliente es un tema popular en los círculos administrativos actuales. Se puede elaborar más en paralelismos que son menos directos, pero quizá más poderosos, mediante buscar estructuras de diseño que trascienden dominios.

Como ejemplo, considérese el lenguaje de patrones CHECKS, que se aplica al diseño de software que trabaja con interfaces interactivas de usuario. Este lenguaje de patrones es fundamentalmente el mismo que el estándar de la IEEE para punto flotante, con obvios paralelismos entre "no un número" (*not a number*) y "valor excepcional" (*exceptional value*).

Se pueden encontrar similitudes para el patrón *Leaky Bucket Counter* en otros dominios también. Este patrón realiza una función en el dominio del software muy similar al filtro paso-bajas en circuitos analógicos en sistemas de sonido. Estos patrones relacionados muchas veces apuntan a patrones más amplios que trascienden dominios; este es un pricipio organizacional fundamental para los sistemas de patrones.

Los patrones más poderosos son aquéllos que involucran cuestiones humanas y tecnológicas al mismo tiempo. En lugar de tratar de resolver los problemas en dos dominios con un solo patrón, se pueden entretejer los patrones de múltiples dominios en un lenguaje conjunto que integre las preocupaciones humanas con la tecnología. Steve Berczuk ha intentado esto al integrar sus primeros patrones de diseño con patrones organizacionales y de proceso [9, 16]:

> Steve incluye un contexto social como la motivación para un patrón. Observa que tanto los problemas organizacionales como los problemas tecnológicos figuran en el desarrollo de software para telemetría satelital debido a que tales proyectos emplean equipos de desarrollo geográficamente distribuidos. Estos proyectos no cuentan con una sola autoridad arquitectónica, haciendo difícil la aplicación del patrón *Architect Controls Product*. Ya que los grupos se encuentran geográficamente aislados y tienen intereses ampliamente divergentes, es importante localizar los cambios al software perteneciente a un equipo (*Code Ownership*, *Organization Follows Location*). La solución consiste en desacoplar a los equipos mediante interfaces arquitectónicas flexibles, empleando patrones como *Callback*, *Parser Builder* y *Hierarchy of Factories*. La arquitectura y organización resultante ayuda a reforzas patrones como *Developer Control Process* y *Code Ownership*.

8.10. Etica

La calidad del software es subjetiva, y la propiedad intelectual del software es un reto actual para la profesión legal. La cuestión ha ganado notoriedad por las demandas sobre apariencia y sensación (*look-and-feel*) que han involucrado a la *Free Software Foundation*, la *League for Programming Freedom*, y otros grupos. Técnicas, métodos y herramientas comparables desafían las medidas objetivas, pues varían dramáticamente en calidad.

Este es un mercado en curso al desastre, un mercado donde técnicas, métodos y herramientas no probadas pueden ganar una gran aceptación. Los patrones son particularmente sujetos a los mismos abusos, debido a que son tan subjetivos, y porque tienen el fin de informar a otros diseñadores y a neófitos.

Por estas razones, Norm Kerth ha lidereado una sesión ética en la primera conferencia de *Pattern Languajes of Programming* (PLoP'94), de modo que los patrones tengan una base ética fuerte desde sus primeros años en el dominio de desarrollo de software. Esta sección enumera algunos de los aspectos éticos del sistema de valores de la Comunidad de Patrones.

8.10.1. Valor Intelectual

Muchos patrones atacan problemas cuyas soluciones son difíciles de medir directamente. ¿Cómo medir la calidad de un diseño? A largo plazo, es posible medir qué tan bien satisface a un cliente y cómo permanece su estructura durante su evolución. Pero tales medidas son indicadores de venta, y no precisamente de lo que se hace el patrón. Los patrones capturan la perspicacia de expertos con un buen registro de éxitos. Aun cuando se apoyan mediante un razonamiento de diseño, la decisión de usar una técnica u otra proviene finalmente de una evaluación subjetiva. Y sin embargo, son estas joyas intelectuales las que a veces son las impresiones clave de diseño que permiten que un producto o una tecnología se desarrolle, y es necesario reconocer a aquéllos que se toman tiempo y esfuerzo en codificar estas técnicas como patrones.

Mary Shaw utiliza el término "valor intelectual" (*intelectual currency*) para describir un modelo para diseminar la literatura de innovación técnica. El valor normal provoca un empobrecimiento si se pierde. Ideas que se atribuyen como innovadoras valen más al originador cuando las publica, y es justa, generosa y precisamente reconocido. Se les recomienda a los escritores

de patrones y los usuarios de patrones citar a sus fuentes cuando usen o codifiquen patrones. Esto anima a los expertos a compartir sus secretos.

8.10.2. Cuestiones Legales

Muchas personas preguntan cómo los derechos de autor y las patentes se relacionan con los patrones. Los patrones pueden tener derechos de autor como cualquier pieza de literatura. Los patrones pueden decribir soluciones técnicas patentables independientemente de su expresión en forma de patrón.

Los patrones no son invenciones en sí mismos; en lugar de eso, codifican prácticas bien establecidas (aunque frecuentemente poco claras). Alguien puede ser identificado como el primero en pensar en un patrón; otro como el primero en usarlo; otro más como el primero en descubrir que es un patrón recurrente; otro como el primero en escribirlo en forma de patrón; otro como el primero en publicarlo; otros como los primeros diseminadores de trabajos derivados; etc. Cada uno de esto papeles puede formar parte de los derechos de propiedad intelectual.

Las cuestiones legales se regulan principalmente por precedencia y convención, así como por leyes, y los profesionales legales deben consultarse para obtener asistencia legal en cada una de las juridicciones.

8.10.3. En Contra del Oculatamiento de Información

El conocimiento es poder. Los individuos con conocimiento técnico clave consideran a su propia experiencia como una fuente de seguridad en sus trabajos, y quizá más profundamente como una fuente de seguridad e identidad personal.

Conforme se reunen patrones de expertos en la industria, es necesario hacerles sentir suficientemente seguros como para compartir su conocimiento con otros: ésta es la principal motivación para el principio de intercambio intelectual.

Más allá de la atribución, es necesario persuadir a los expertos que pueden ser todavía más efectivos si comparten sus ideas ampliamente. El conocimiento escrito puede lanzar a estos expertos a la posteridad en el tiempo, y mundialmente en el espacio.

Finalmente, ni siquiera los patrones pueden reemplazar a los expertos.

Se pueden capturar patrones de los expertos para ayudar a los diseñadores y programadores en sus labores diarias, amplificando lo que ya conocen, y potenciando las habilidades que tienen. Un verdadero experto usa patrones inconcientemente, conforme éstos se han convertido en parte de los procesos mentales del experto. Alexander llama a este estadío "la puerta" (*the gate*) [4]. Los patrones capturan los extremos de las estrategias intuitivas de los verdaderos expertos; mucho de su poder proviene de un nivel más profundo, accesible solo a los expertos que alcanzan dentro de sí mismos ideas que normalmente no son accesibles a los demás. Los patrones pueden ayudar a los diseñadores y programadores para evitar errores debido a su inexperiencia, pero no serán capaces de transformar a un carpintero en un Frank Lloyd Wright, o a cada diseñador de software en un Donald Edmund Knuth. Es todavía muy necesaria la experiencia cultivada por los expertos en el dominio.

8.10.4. No Exagerar

La exageración fue un tópico importante de discusión en la sesión de Norm Kerth. La exageración tiene dos fuentes: entusiasmo que proviene de defensores bien informados sobre patrones y expectaciones infladas de usuarios de patrones fuera de la Comunidad de Patrones, ya sea que estén bien intencionados o no. La exageración frecuentemente provoca expectativas más allá de lo que los patrones pueden lograr.

Mediante hacer a la exageración indeseable en la Comunidad de Patrones, se espera minimizar ambas fuentes de exageración, y aumentar las oportunidades de que las expectativas sean realistas respecto a lo que los patrones pueden lograr. Es inevitable que algunos oportunistas exageren las expectativas sobre lo que los patrones pueden lograr con el fin de vender sus productos, y de hecho, este problema está muy por afuera de la influencia de la Comunidad de Patrones.

Hay otra cuestión interesante respecto a la exageración: una indebida atención a tratar de reducir la exageración es una forma de exageración en sí misma.

9. Historia

El interés en Patrones de Software surge de las actividades de algunos de los principales diseñadores de software en la industria. Ya que la programa-

ción Orientada a Objetos han sido el foco del diseño y práctica de software desde la mitad de los años 1980, era natural que los patrones emergieran del diseño Orientado a Objetos. La liga entre patrones y objetos persiste a la fecha, aún cuando no hay nada intrínseco en la relación.

Algunos de los trabajos más tempranos con patrones se realizó por Ward Cunningham y Kent Beck, quienes trabajaban en aquel entonces en Tektronix. Kent Beck encontrón patrones en sus días como estudiante [3]:

> Descubrí patrones primero como estudiante en la Universidad de Oregon. Muchos estudiantes en la residencia para primer año estaban en la Escuela de Arquitectura. Fueron ellos quienes me apuntaron en la dirección de Christopher Alexander. Leí todo su libro *Timeless Way of Building* de pie en la librería de la universidad por varios meses.
>
> Trabajé para Tektronix durante año y medio cuando volví a cruzarme con Alexander de nuevo. Encontré un vieja copia de *Notes on the Sinthesis of the Form*. La opinión de Alexander respecto a los "metodologistas" en la introducción de la segunda edición resonó con mis propias ideas, llevándome de nuevo a *Timeless Way of Building*. Parecía que todo aquello que no le gustaba a Alexander de los arquitectos, no me gustaba a mí de los ingenieros de software. Convencí a Ward Cunningham que estabamos sobre algo grande.

En 1987, Ward Cunningham y Kent Beck eran consultores con un grupo que tenía problemas con el diseño de interfaces a usuario. Decidieron, mientras estaban en camino en la camioneta Vanagon de Ward, probar los patrones que habían estado estudiando. Alexander dice que los ocupantes de un edificio son quienes deben diseñarlo, de modo que de manera similar, los usuarios de un sistema deben diseñar sus interfaces a usuario. Ward presentó un lenguaje de patrones que les ayudó a tomar ventaja de las fortalezas de Smalltalk, y evitar sus debilidades. Los patrones fueron originalmente presentados en forma verbal. En seguida se presentan algunos extractos de esos patrones, tomados del sitio Web [4].

[3] Portland Pattern Repository: `http://c2.com/ppr/`
[4] Wiki-WikiWeb: `http://c2.com/cgi/wiki?WindowPerTask`

Window Per Task

Hágase una ventana específica por cada tarea que el usuario deba realizar. Toda la información necesaria para completar la tarea debe estar disponible en la ventana en unas cuantas sub-ventanas (*Few Panes*). Supóngase que cualquier tarea pre-requisito se ha completado (si no se han completado, el usuario simplimente cambiará de ventana).

Este patrón efectivamente hace un lado algunos problemas graves que el patrón *Model-View-Controler* tenía con ventanas mutuamente dependientes durante el proyecto Tek LT1000.

Los ingenieros del LT1000 sabían exactamente qué tareas realizaban sus usuarios. Estas eran solo cinco o seis.

Few Panes

Para entender las cosas complejas se debe frecuentemente ver desde diferentes puntos de vista. Por lo tanto, provéase estos puntos de vista (llamados simplemente "vistas") mediante dividir el área de una ventana por tarea (*Window Per Task*) en sub-ventanas.

Standard Panes

Es necesario aprender a operar en cada tipo de sub-ventana que se ofrezca por cada ventana.

Por lo tanto, preséntese cada sub-ventana en el formato ofrecido por una de las sub-ventanas. Para las pruebas del programa, deben limitarse a:

- Texto
- Lista
- Tabla
- Forma de onda

Nouns and Verbs

Las cosas existen mientras se da la acción.

Por lo tanto, póngase listas de cosas (nombres) que persisten durante la interacción en una sub-ventana. Póngase acciones (verbos) en menúes cortos (*Short Menus*) que aparezcan y desaparezcan conforme la acción comienza.

La única ocasión en que los especialistas del dominio tuvieron problemas para satisfacer este patrón fue en un menú que pensaron que debía incluir:

- Decimal
- Octal
- Hexadecimal

Se reconoció que en realidad estaban pensando en las opciones del menú como acciones y se les sugirió renombrarlas como:

- Ser Decimal
- Ser Octal
- Ser Hexadecimal

Short Menus

Los elementos de un menú que aparece deben buscarse visualmente varias veces.

Por lo tanto, háganse cortos, fijos y de un solo nivel.

Es interesante que este patrón fue fácilemente cumplido dadas las condiciones establecidas por una ventana por tarea (*Window Per Task*) y nombre y verbos (*Nouns and Verbs*).

Ambos autores quedaron sorprendidos de la elegancia de las interfaces que sus usuarios diseñaron. Los resultados de este experimento fueron reportados en OOPSLA'87, en Orlando, Florida. Ward presentó un panel [18], y ambos se presentaron en el taller de Norm Kerth *Where do objects come from?* [7]. Hablaron de patrones hasta el cansancio, pero sin otros patrones mas concretos, ya que nadie mas había atendido al taller.

Mientras tanto, Erich Gamma estaba muy ocupado escribiendo y reflexionando acerca del diseño Orientado a Objetos en ET++ como parte de su tesis de doctorado. Para ese momento, ET++ se había establecido ya como un ejemplo de programación en la comunidad de C++. Erich notó la importancia de las estructuras recurrentes de diseño, o patrones, de ET++. La pregunta realmente era: ¿cómo capturarlos y comunicarlos?

En 1990, en la conferencia ECOOP/OOPSLA en Ottawa, se dedicó una sesión llamada *Towards an Architecture Handbook*, donde Erich Gamma, Richard Helm, y otros se enfrascaron en una discusión sobre patrones.

Justo antes de ECOOP'91, Erich Gamma y Richard Helm, iniciaron el catálogo de patrones que eventualmente se convertirían en sus patrones de diseño. Identificaron muchos patrones, como por ejemplo:

- *Composite*
- *Decider*
- *Observer*
- *Constrainer*

Muchos de estos patrones lograron abrirse camino hasta el libro de *Design Patterns* [23]; muchos otros permanecieron inmaduros y sin publicarse hasta la fecha.

La situación comenzó a cambiar en un taller de OOPSLA en 1991. Coincidentalmente, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides se encontraron ahí; después, este mismo grupo se conocería como "el grupo de los cuatro" (*the Gang of Four*, o simplemente, GoF) que escribió el libro *Design Patterns* [23]. Muchos de quienes después formarían el grupo de Hillside estaban ahí: Ward y Kent, Desmond D'Souza, Norm Kerth, y otros como Doug Lea y Wolfgang Pree. El taller se repitió en 1992, en donde el GoF propiamente se formó. También, ahí se presentó la primer publicación sobre patrones de Frank Buschmann.

Antes de todo esto, en 1988, Jim Coplien comenzó a catalogar patrones específicos para C++, que nombró *idioms* (modismos), y que representan patrones de bajo nivel. Los primeros trabajos de tal labor se utilizaron para enseñar tecnología Orientada a Objetos en AT&T a principios de 1989. En septiembre de 1991, Coplien publicó su libro *Advanced C++ Programming Styles and Idioms* [15]. A partir de las mismas fechas, Peter Coad también ha intentado explorar patrones, ya que los menciona en una publicación que hizo en 1992 en el *Communications of the ACM* [14].

Por ese tiempo, el grupo de patrones comenzó a descrubrir sus intereses mutuos, buscando la oportunidad de llevar sus ideas más allá. En mayo de 1993, algunos de ellos se reunieron en un taller de diseño Orientado a Objetos, en Thornwood, Nueva York. Reflexionando y pensando fue la principal actividad del taller. Desmond D'Souza, Doug Lea, Kent Beck, Ralph Johnson, Ron Casselman y John Vlissides actuaron como moderadores.

En ese mismo año, en agosto, Kent y Grady Booch patrocinaron un retiro a una montaña en Colorado, en donde un grupo discutió sobre las bases de los Patrones de Software. Ward Cunningham, Ralph Johnson, Ken Auer, Hal Hildebrand, Grady Booch, Kent Beck y Jim Coplien analizaron las ideas de Alexander, buscando una unión entre objetos y patrones. Se aceptó iniciar con las bases propuestas por Erich Gamma, estudiando patrones Orientados a Objetos para usarlos en forma generativa en el sentido en que Alexander usaba sus patrones para planeación urbana y arquitectura de edificios. Se usó el término "generativo" para significar "creacional", a fin de distinguir entre algunos patrones de Gamma y las características de los patrones en general.

Para tener una mejor compresión de los patrones de Alexander, el grupo condujo un ejercicio al lado de una colina, que consistía en construir un edificio llamado *Center for Object-Oriented Programming*. La visión era diseñar un edificio donde desarrolladores de software pudieran acercarse a sus clientes y aprender de los demás, usando patrones como base de su diálogo. El grupo diseñó el edificio como lo sugiere Alexander: planeando el edificio en el propio sitio de construcción, aprovechando el paisaje, y mezclando el edificio con sus alrededores. Fue una experiencia reveladora. El grupo seguiría reuniéndose, para profundizar su entendimiento sobre patrones, y llevando los patrones a la industria del software. Comenzaron informalmente a referirse a sí mismos como grupo con el nombre de *Hillside*.

El grupo de Hillside continuó reuniéndose hasta que en abril de 1994 se planeó la primera conferencia sobre patrones: *Pattern Languages of Programming* (PLoP). Se buscó algo poco usual para la organización, ya que la mayoría estaba de acuerdo en correr riesgos con cosas nuevas. Esa fue la primera vez que Richard Gabriel se encontró con el grupo.

En agosto del mismo año, cerca de 80 personas se reunieron en Allerton Park, cerca de Monticello, Illinois, para la conferencia. Ward Cunningham y Ralph Johnson actuaron como presidentes del programa y de la conferencia respectivamente. Las memorias de la conferencia aparecieron hasta mayo de 1995, bajo el nombre de *Pattern Languages of Program Design* [16].

Mientras tanto, el GoF había logrado terminar su libro y enviarlo al editor. El mayor compendio de patrones reunido bajo el título de *Design Patterns: Elements of Reusable Objet-Oriented Software* [23]. Tan solo en la conferencia de OOPSLA de 1994 el libro vendió 750 copias, siete veces más que cualquier otro libro técnico que la editorial Addison-Wesley hubiera puesto a la venta en una conferencia.

Referencias

[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus (1996) *Fault Tolerant Telecommunication Patterns.* Pattern Languages of Program Design 2, Addison-Wesley.

[2] C. Alexander (1974) *Notes on the Synthesis of Form.* Hardvard University Press.

[3] C. Alexander *et al.* (1977) *A Pattern Language.* Oxford University Press.

[4] C. Alexander (1979) *The Timeless Way of Building.* Oxford University Press.

[5] D.L.G. Anthony (1996) *Patterns in Classroom Education.* Pattern Languages of Program Design 2, Addison-Wesley.

[6] K. Beck, R. Crocker, J. Coplien, L. Dominic, G. Meszaros, E. Paulisch, and J. Vlissides (1996) *Industrial Experience with Design Patterns.* Proceedings of ICSE 1996.

[7] K. Beck (1988) *Using a Pattern Language for Programming.* In Workshop on Specification and Design, ACM SIGPLAN Notices 23-5, Addenum to the Proceedings of OOPSLA 87.

[8] S. Berczuk (1996a) *Object Models: Strategies Patterns and Applications.* Book Review in Object-Oriented Systems, 43.

[9] S. Berczuk (1996b) *Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams.* Pattern Languages of Program Desi*Counted Body*gn 2, Addison-Wesley.

[10] G. Booch (1994) *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings.

[11] F. Buschmann and R. Meunier (1995) *A System of Patterns.* Pattern Languages of Program Design, Addison-Wesley.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal (1996) *Pattern-Oriented Software Architecture. A System of Patterns.* John Wiley & Sons, Ltd. Chichester, United Kingdom.

[13] P. Coad, D. North, and M. Mayfield (1995) *Objects Models: Strategies, Patterns and Applications.* Prentice-Hall.

[14] P. Coad (1995) *Object-Oriented Patterns.* Communications of the ACM, 35-9.

[15] J.O. Coplien (1992) *Advanced C++ Programming Styles and Idioms.* Addison-Wesley.

[16] J.O. Coplien (1995) *A Development Process Generative Pattern Language.* Pattern Languages Of Program Design, Addison-Wesley.

[17] J.O. Coplien (1996) *The Human Side of Patterns.* C++ Report 81=86, January, 1996.

[18] W. Cunningham (1988) *Panel on Desigm Methodology.* ACM SIGPLAN Notices 23-5, Addenum to the Proceedings of OOPSLA 87.

[19] W. Cunningham (1995) *The CHECKS Pattern Language of Information Integrity.* Pattern Languages of Program Design, Addison-Wesley.

[20] D.L. DeBruler (1995) *A Generative Pattern Language for Distributed Processing.* Pattern Languages of Program Design, Addison-Wesley..

[21] S.H. Edwards (1992) *Streams: A Pattern fro Pull-Driven Processing.* . Pattern Languages of Program Design, Addison-Wesley.

[22] R.P. Gabriel (1996) *Patterns of Software: Tales from the Software Community.* Oxford University Press.

[23] E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995) *Design Patterns. Elements of Reusable Object Oriented Software.* Addison-Wesley.

[24] I. Jacobson, M. Chisterson, P. Johnson, and G. Overgaard (1992) *Object-Oriented Software Engineering. A Use Case Driven Approach.* Addison-Wesley.

[25] N. Kerth (1995) *Caterpillar's Fate.* Pattern Languages of Program Design, Addison-Wesley. .

[26] D. Knuth (1991) *Literate Programming.* Center for the Study of Language and Information, Stanford, California.

[27] A.R. Koeing (1995) *Patterns and Antipatterns.* Journal of Object-Oriented Programming, 8-1.

[28] G. Meszaros (1996) *A Pattern Language for Improving Capacity of Real-time Systems.* Pattern Languages of Program Design 2, Addison-Wesley.

[29] M. Morgan (traductor) (1960) *Vitruvius. The Ten Books of Architecture.* Dover.

[30] D.L. Parnas (1976) *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering, SE-2: 1 9.

[31] W. Pree (1995) *Design Patterns for Object-oriented Software Development.* Addison-Wesley.

[32] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson (1991) *Object-Oriented Modeling and Design.* Prentice-Hall.

[33] W. Rybczynski (1989) *The Most Beautiful House in the World.* Viking.

[34] W. Rybczynski (1992) *Looking Around.* Viking.

[35] P. Seng (1990) *The Fifth Discipline: The Art and Practice of Learning Organizations.* Doubleday.

[36] P. Viljamaa (1995) *The Patterns Business: Impressions from PLoP 94.* ACM Software Engineering Notes 20-1.

[37] J. Vlissides, N. Kerth, and J. Coplien, eds. (1996) *Pattern Languages of Program Design 2.* Addison-Wesley.

[38] T. Volk (1995) *Metapatterns across Space, Time and Mind.* Columbia University Press.

[39] K. Wolf and C. Liu (1995) *New Clients with Old Servers: A Pattern Language for Client/Server Frameworks.* Pattern Languages of Program Design, Addison-Wesley.

[40] G.M. Weinberg and D. Weinberg (1988) *General Principles of System Design.* Dorset.

[41] W. Zimmer (1995) *Relations between Design Patterns.* Pattern Languages of Program Design, Addison-Wesley.