Respuestas a preguntas teóricas ISW1

- Programming As Theory Building

¿Cómo relacionaría lo escrito por Naur con un sistema de organización de trabajo como el Taylorismo?

Un sistema de organización de trabajo como el taylorista piensa principalmente en la producción que genera ese trabajo, de hecho el foco está puesto en aumentar la producción. Para Naur la programación debe ser vista como lo opuesto a ese tipo de visiones en las que el foco está puesto en la producción de un programa y otros textos, lo cual parece ser una noción bastante generalizada. La programación es para Naur la construcción de una teoría que no se puede limitar dentro de los cánones de producción de un producto sino de un conocimiento y como tal es de índole intelectual.

¿Qué habría que modificar en la concepción de Naur para que esta sea compatible con la visión de que la programación puede ser en el futuro desarrollada por completo por agentes artificiales?

Naur hace hincapié en el concepto de la programación como construcción de teoría. La noción de teoría en la que se basa se construye sobre la naturaleza de la actividad intelectual y la forma particular en que esta se diferencia y supera la inteligencia. El ejercicio de la inteligencia supone la capacidad de seguir ciertas reglas para la realización de una tarea, lo cual sería lo esperable por agentes artificiales. Pero el ejercicio intelectual de la construcción de teoría supone no sólo el conocimiento para hacerlo inteligentemente sino también ser capaz de explicarla, responder preguntas al respecto, discutir sobre ella, etc. Compatibilizar la visión de Naur con el desarrollo de programas por parte de agentes artificiales implicaría que dichas inteligencias logren acercarse al desarrollo de tareas intelectuales, o bien la noción de teoría que utilice Naur sea más laxa y descarte la intelectualidad involucrada en el proceso.

- The Design of Everyday Things

Elija dos entes de la realidad (pueden ser objetos físicos del mundo real o incluso software) y clasifíquelos por: Visibilidad de las partes a operar, Visibilidad de Mapeo, Feedback, Modelo Conceptual y Diseño Natural (y sus affordances). Por ej. hablando de Modelo Conceptual, una tijera tiene uno bueno, un reloj digital uno inexistente, y la heladera con freezer de Dan uno malo. Explique y justifique cada una de las 5 apreciaciones de la clasificación. No puede utilizar ninguno de los ejemplos dados en el libro o los videos de Norman.

Bicicleta de carrera: Las partes a operar son bien visibles (los pedales, el asiento, el manubrio). El mapeo también es directo: los pedales traccionan sobre la rueda trasera, cada freno (si tuviese más de uno) acciona sobre cada una de las ruedas, los cambios también pueden ser

dos, uno para cada disco. El feedback es inmediato, accionar los pedales pone en movimiento la bicicleta, utilizar los frenos disminuye la velocidad, usar los cambios modifica la cadencia del pedaleo. Y cada una de estas operaciones tiene un feedback visual a su vez inmediato. El modelo conceptual suele estar bien establecido (probablemente por exposición cultural más que nada) y sería difícil y extraño encontrar una bicicleta (al menos comercial) cuyo diseño vaya en contra de este. Los affordances también contribuyen a la relación entre el diseño y la operabilidad: sea que lo primero que se descubre sea el sillín, los pedales o el manubrio, una vez que se establece la relación entre la parte y el usuario las restantes no tienen mucha alternativa.

Ojotas: Estos objetos tienen un buen modelo conceptual al igual que la tijera. Generalmente vienen con una división para separar el dedo gordo del resto, este diseño ergonómico a la forma del pie a su vez daría un feedback inmediato si las ojotas fuesen colocadas en los pies equivocados, ¿quién no metió su pie en la ojota equivocada solo para descubrir que tenía la mitad del pie afuera? De la misma manera una ojota dada vuelta no sería de utilidad ya que no se acomodaría al pie, y si bien alguna vez muchos juegan con llevarlas en las manos, usar ojotas como guantes no da buenos resultados a la hora de realizar actividades de motricidad fina. Los modelos conceptuales de las piezas de vestimenta suelen estar bien arraigados culturalmente, solo rompiendo en ámbitos de la haute couture o moda de pasarelas. En cuanto a los 'affordances' de las mismas suelen ser de material que invitan a ser llevadas en los pies: no hay ojotas de seda (las cuales tal vez invitarían más a ser llevadas a las manos); el espacio para acomodar los dedos es justo para ellos mismos y no habría manera de intentar pasar un brazo sin cierta dificultad. También son habitualmente de algún material resistente no demasiado noble, unas ojotas de cuero o terciopelo no serían muy amigas de la playa y la humedad, por ejemplo.

Si le hace falta, puede subir imágenes del ente elegido (por ej. si quiere plantear la clasificación de una puerta o un exprimidor de jugo "ganador" de un premio de diseño")

- Design Principles Behind Smalltalk

¿Qué modalidad usaron los desarrolladores del proyecto Smalltalk?

Los desarrolladores siguieron una metodología de iteraciones incrementales con ciclos de dos a cuatro años que tiene un paralelismo con el métodos científico en tanto dicho ciclo se comprende de: hacer una observación, formular una teoría, hacer una predicción que pueda ser probada y repetir el ciclo.

¿En qué dos aspectos decidieron enfocarse para investigar?

Uno de los aspectos en los cuales se enfocó la investigación fue la búsqueda y generación de un lenguaje de descripción (lenguaje de programación) que sirva de interfaz entre los modelos que habitan en la mente humana y aquellos que se producen en la computadora. El otro

aspecto era la búsqueda de un lenguaje de interacción (interfaz de usuario) que supiera hacer coincidir la comunicación humana con la que se produce en la computadora.

El paper enumera rápidamente dos principios en los que se basa el diseño de Smalltalk. ¿Cuál es el tercero y que significa?

El tercer propósito tiene que ver con la razón de existir de un lenguaje. Un lenguaje es principalmente un entorno en el cual se puede producir la comunicación. El hecho de querer exteriorizar la comunicación, involucrar a un otro, hace necesaria la construcción de un marco en el cual deben haber reglas para que esa interacción se produzca correctamente. Siendo unos de los pilares de la investigación el lograr un lenguaje para comunicar la mente humana y la computadora, es entendible que Smalltalk sea comprendido entonces como un lenguaje orientado a pasaje de mensajes. Tanto su lenguaje como su interfaz fueron concebidos con el fin de darle soporte a la comunicación entre la mente humana y la computadora.

- Self: The Power Of Simplicity

Indique cómo logra 'Self' uniformar el concepto de estado y comportamiento.

No distingue estado de comportamiento, Self uniforma estos conceptos mediante prototipación. A diferencia de los lenguajes de clasificación donde las clases definen comportamiento y el estado queda en las instancias, no hay clases ni variables. En Self, mediante la creación prototípica de objetos, la delegación define el comportamiento y el estado. Luego cada objeto puede redefinirlo y extenderlo.

Explique cómo lograr encapsulamiento utilizando 'Self'. Por ejemplo para cambiar un número fijo (como el 17) por un número aleatorio al hacer una llamada a self.

En el caso de Self, no hay colaboradores internos como en Smalltalk, el acceso a los mismos es a través de mensajes. No hay otra forma de acceder al slot que contiene al colaborador más que haciéndolo a través de alguno de los mensajes que pueden o no estar definidos: uno para acceder y otro para escribir. De esta manera, la representación de los objetos queda oculta, contenida en slots y accesible mediante mensajes.

Para hacer el cambio por un número aleatorio, habría que remover el mensaje que escribe el slot "x:" (para que el valor no pueda ser modificado) y reemplazar los contenidos del slot correspondiente al mensaje "x" por el código que genera un número aleatorio.

Explique cómo se crean objetos en 'Self' y compárelo con el paradigma de clasificación aristotélico.

En un lenguaje de clasificación un objeto se crea mediante la instancia obtenida por la definición que da la clase correspondiente. En un lenguaje de prototipos como Self, un objeto

se crea por clonación (se hace una copia) de un prototipo y luego se define el comportamiento que lo haga único. En Self cualquier objeto puede ser copiado.
En el paradigma de clasificación aristotélico hay dos mundos separados, el sensible y el inteligible, el de las ideas y el de las instancias. Cualquier instancia se origina siempre a partir de una idea. En Self no se distinguen ideas de instancias, todos los objetos que existen tienen una relación de parentesco con un objeto prototípico ("object").
En el paradigma de clasificación las relaciones entre instancias son dos, a qué idea pertenece y a cuál se asemeja. En Self las relaciones se simplifican y basta con entender de quién se hereda, el comportamiento está dado por la relación "hijo de".
En Self pueden existir objetos que pertenezcan o no a una idea común, todos son prototipos de algún otro. Sin embargo, en el paradigma de clasificación cualquier instancia existe porque pertenece a una idea a la cual responde.

- Multiple Choice

    - ***El paper presenta la situación en la que una expresión posee*** más de una variable independientemente polimórficas

    - **Para que dos objetos sean polimórficos es necesario pedir que** ninguna de las anteriores

    - **La solución presentada en el paper (Double-Dispatch) utiliza polimorfismo para** evitar el chequeo de tipos (isMemberOf)

    - **Teniendo en cuenta el ejemplo del paper (Graphical Objects y Display Ports). La solución final, donde los nuevos métodos se encuentran en las clases ports** es igual de buena que la solución donde los métodos se encuentran en las clases graphical y solo depende de una decisión de diseño.

- Polymorphic Hierarchies

Compare la definición de polimorfismo visto en clase con la presentada en el paper.

Según la definición de polimorfismo vista en clase, un conjunto de objetos son polimórficos respecto de un conjunto de mensajes si cada objeto del primero responde semánticamente igual a los mensajes del segundo. De acuerdo a Bobby Woolf el polimorfismo entre dos objetos ocurre cuando ambos comparten no solo la misma interfaz (habla de una 'core interface') sino que además se comportan de la misma manera.

Con respecto a la igualdad semántica vista en clase, esto se refiere a mensajes que tienen el mismo nombre, reciben colaboradores polimórficos y devuelven objetos polimórficos, de esta

manera el 'qué' entre objetos polimórficos es el mismo. Si entendemos el 'qué' como la interfaz y el comportamiento entonces ambas definiciones son equivalentes.

Explique la siguiente afirmación del autor: Template Class is similar to the Template Method pattern.

Son similares en cuanto a que ambos definen interfaces postergando los detalles de sus implementaciones a las subclases. Si bien ambos templates se componen de una clase principal y otras subclases implementadoras, una Template Class define una interfaz para una clase y un Template Method para un método.

Todos los objetos que sepan responder al mensaje value y value: serán polimórficos.

Falso. No alcanza con que sepan responder a los mismos mensajes. Como vimos en clase con el ejemplo de los mensajes #value y #value: utilizados a modo de accessors (get y set) de colaboradores internos y los mismos para bloques closure con objetivos e implementaciones totalmente distintos. En este ejemplo si bien ambos objetos entienden los mensajes, la semántica es diferente, no responden al mismo 'qué', por lo tanto, no son polimórficos.

En una jerarquía de clases, las clases más arriba en la jerarquía deben definir el cómo, es decir, deben proveer la implementación de los métodos y luego las subclases solo deben tener un comentario que diga "See superimplementor."

Falso. El autor propone mediante el patrón de Template Class que existan clases más arriba que definan los comportamientos comunes pero postergan la implementación, dejando la responsabilidad a las clases más bajas, las que realmente "hacen".
El autor propone esta jerarquización con templates sin buscar subsanar problemáticas de código repetido sino de polimorfismo. Si las clases de más arriba definiesen el cómo, las clases que heredasen de estas estarían atadas a implementaciones determinadas, por lo que no serían polimórficas.

- Null Object Pattern

¿Qué técnica usa el Patrón Null Object para evitar el uso de ifs?

Define un objeto polimórfico, el 'null object', que teniendo la misma interfaz que el objeto que busca reemplazar, no hace nada. Para esto hay que armar una nueva jerarquía basada en el objeto que se busca reemplazar. El objeto nulo y el original serán subclases de una misma clase abstracta que definirá los mensajes que deberán saber responder ambos.

Explique por qué la solución del patrón es superadora a la de usar nil en el ejemplo del Controller.

En el ejemplo, propone como primera solución crear un controlador read-only, pero este seguirá procesando el input aunque sin hacer nada con ese input. Esta no sería una buena opción ya que no necesita hacer ningún procesamiento para no hacer nada. La otra solución es reemplazar la instancia del controlador por nil. Esto tampoco es una buena alternativa, porque en cada lugar donde se accediera al controller habría que llenar el código con condicionales, lo cual lo haría más complejo.
Usando el 'null object' los clientes pueden tratar uniformemente colaboradores reales y nulos, sin necesidad de agregar lógica para el control de flujo, lo cual incrementaría la complejidad accidental. Además, el 'do nothing' queda encapsulado en un objeto.

El patrón de diseño explica que estructuralmente el Null Object debe tener un ancestro común con la jerarquía de real objects. ¿Qué opina de dicha restricción?

Esta restricción no aplica a lenguajes dinámicamente tipados donde sólo sería necesario que el NullObject sea polimórfico con el objeto que se busca reemplazar. En un lenguaje estáticamente tipado si es necesario definir un ancestro común para permitir que los objetos de la misma jerarquía sean intercambiables.

- Object Recursion

¿Qué intenta resolver el patrón Object Recursion?

Intenta resolver escenarios en los cuales se necesita enviar colaboraciones a varios objetos para obtener un resultado, evitando tener que enviar mensajes individuales y favoreciendo la comunicación existente entre los mismos. Para ello realiza un procesamiento distribuído de un pedido sobre una estructura mediante delegación polimórfica. Por ejemplo, se puede utilizar para pasar un mensaje a una estructura vinculada donde no se conoce el destino final o para mandar el mismo mensaje a todos los nodos de la estructura o para distribuir comportamiento en la misma.

Presente un ejemplo (diferente al del paper) que se pueda abordar con dicho patrón

En el caso de una pila con un protocolo estándar de push, pop y top, para conocer el tamaño de la misma habría que desapilar cada uno de los elementos hasta llegar a la base de la misma o mantener un contador que se actualice ante cada pop y push. En este caso queda claro que las opciones implican o bien alterar el estado de la pila con el fin de saber su tamaño o bien agregar lógica a los mensajes mencionados. Si se implementa con objetos, polimórficos al mensaje size, que modelen el primer elemento de la pila (base la pila) y luego el resto (elementos apilables), se puede enviar el mensaje size al tope de la pila y luego que se produzca la recursión entre ellos hasta la base (el caso base). El resultado de dichas colaboraciones entre los objetos contenedores de la pila devolverá el resultado sin necesidad de alterar el estado de la misma.

¿Cómo se relaciona el patrón Object Recursion con el patrón Decorator?

El decorator puede ser visto como un caso de object recursion, en el cual el objeto decorador delega los mensajes al decorado agregando comportamiento, esto implica recursión en un solo nivel (o el nivel de decoraciones que tenga). Sin embargo, un objeto recursor no necesariamente tiene que decorar un objeto, solamente delegar los mensajes a sus componentes de manera recursiva.

- No Silver Bullet – Essence and Accident in Software Engineering

Describa las características inalienables al software que Brooks presenta

Las características que se presentan inalienables al software son: ***complejidad****,* ***conformidad****,* ***adaptabilidad*** *e* ***invisibilidad****:*
La ***complejidad*** *del software surge de la gran cantidad de partes únicas interactuando en un sistema. Las partes son únicas porque se encapsulan como funciones, subrutinas u objetos, y se invocan según sea necesario en lugar de replicarse; una diferencia rotunda con computadoras, edificios o automóviles donde abundan las partes repetidas.*
La ***conformidad*** *aparece como una dificultad más en tanto el software debe cumplir con las especificaciones exactas en la representación de cada parte, en las interfaces con otras partes internas y en las conexiones con el entorno en el que opera. El software no es creado de manera aislada, tiene que adherir a limitaciones del mundo real, hardware preexistente, componentes de terceros, regulaciones gubernamentales, sistemas legacy, etc.*
La ***adaptabilidad*** *y el cambio son características inalienables al software en tanto es fácil modificarlo y cambiarlo y a la vez eso no lo hace fácil. La* ***complejidad*** *y la necesidad de* ***conformidad*** *hacen que cualquier modificación del software cuente con un mínimo bastante grande de dificultad.*
En tanto el software no tiene propiedades físicas, puede considerarse ***invisible****. Y, aunque los efectos de su ejecución sean percibidos por alguno de los cinco sentidos, el software es en esencia intangible y por eso es que existen tantas formas distintas de representación que intentan dar visibilidad a lo invisible.*

¿Por qué desarrollar software es esencialmente complejo según Brooks?

De acuerdo al paper, Brooks diferencia dos tipos de complejidad en relación al software: la accidental y la esencial. La complejidad accidental se relaciona con los problemas que los ingenieros crean y pueden arreglar. En cambio la complejidad esencial tiene por origen el problema en sí mismo que se intenta solucionar y no hay forma de quitarla del medio.
De hecho Brooks enumera cómo el pasar de los años nos ha quitado de encima mucha de esa complejidad accidental con distintas herramientas y que hoy los programadores dedican la mayor parte del tiempo a atender cuestiones de la complejidad esencial. Incluso considerando avances como los lenguajes de alto nivel, no se han reducido en órdenes de magnitud significativos los tiempos que implican el desarrollo de software.

Habiendo dicho esto, la creación de software queda claro que es compleja por su naturaleza. Incluso si se inventara un lenguaje de alto nivel que operara al nivel del dominio del problema, programar seguiría siendo una tarea compleja porque aún había que definir las relaciones entre entidades del mundo real, identificar casos excepcionales, anticipar todas las posibles transiciones y casos, etc.

- Decorator, Adapter, Proxy

Describa similitudes y diferencias entre Decorator y Adapter.

Si bien ambos patrones son considerados estructurales, el patrón decorator es diferente del adapter en tanto el decorator solo cambia las responsabilidades del objeto y no su interfaz. Es por esto que resulta más transparente y le permite aplicar una composición recursiva. En cambio un adaptador le da al objeto adaptado una interfaz completamente nueva, permitiendo interactuar a objetos que no estaban preparados para comunicarse entre ellos. En cuanto a las similitudes, entendemos que ambos "envuelven" al objeto "decoratee" (que decoran) o "adaptee" (al cual adaptan), de manera de convertir la composición en una alternativa a la subclasificación.

¿Qué ventajas y desventajas tiene implementar un Proxy polimórfico sobre uno no polimórfico?

El proxy polimórfico responde a todos los mensajes a los que el objeto al cual está sirviendo de proxy sabe responder. La desventaja es que dicha implementación requiere la elaboración de una lógica más abarcativa que la necesaria para que el objeto sirva de proxy. Para que sea realmente polimórfico se puede realizar una delegación de los mensajes que el proxy no entiende hacia el objeto al que sirve de proxy. Para ello la clase a la que el proxy pertenece debe heredar de ProtoObject, no simplemente de Object, con el fin de evitar circularidad en el "look up" del mensaje potencialmente no entendido. La ventaja de implementarlo de esta manera es que el reemplazo en código existente es más limpio en tanto no necesita otros cambios más que al reemplazar la instanciación del objeto por la versión correspondiente de su proxy.

¿Qué es el problema de la "identidad" o de "self"? ¿En cuáles de estos patrones aplica?

El problema de la identidad surge en los escenarios en que objetos que ya participan de colaboraciones son "envueltos" por nuevos objetos. Hay dos modalidades para enviar mensajes en colaboraciones, o bien se delegan (delegation) o se reenvían (forwarding) y se diferencian entre sí por la identidad que toma "self". Este caso se da particularmente cuando se aplica el patrón Decorator. Una desventaja del mismo es que la identidad del decoratee queda oculta y acceder directamente a la misma es problemático. Este patrón se usa para agregar comportamiento y responsabilidades a un objeto que no son inherentes a su identidad, de hecho cuando un objeto es envuelto por un "wrapper", se proyecta una identidad distinta (la del wrapper mismo) y es por eso que se denota como un problema.

- Future

Multiple Choice

El patrón Future se utiliza para *poder programar asincronicamente, paralelizando la ejecución y sincronizando los resultados..*

Explique por qué son importantes las promesas en un lenguaje fuertemente basado en eventos como Javascript.

Al estar basado en eventos los mismos pueden tener distintos tiempos de respuesta. En ese caso es necesario contar con objetos a los cuales se les puedan enviar mensajes sin que el progreso de la ejecución quede interrumpida a la espera de los resultados de los mismos. Usualmente dichas promesas son creadas junto con callbacks, conjuntos de colaboraciones que deben ejecutarse una vez que dicho proceso asincrónico termine.

Explique qué es un future transparente o polimórfico. ¿Qué ventajas tiene con respecto a uno que no comparte el mismo protocolo que el objeto a proxiar?

El patrón de futuro ofrece una abstracción que encapsula un procesamiento en forma diferida. Manejar estos procesos directamente requiere el conocimiento de conceptos como la concurrencia y sus consecuencias. La noción de un futuro transparente permite esconder la complejidad con una abstracción que es polimórfica al objeto y hace sencillo su utilización en tanto difiere del objeto al que sirve de proxy en su instanciación y el resto de código será indistinto se trate de un futuro o no. Sin embargo, hacerlo de esta manera deviene en una mayor complejidad accidental en tanto requiere mayor cantidad de mensajes y métodos para soportar el polimorfismo.

- State

Dé un ejemplo ORIGINAL de un dominio donde pueda caracterizar varios estados de un objeto cuyo comportamiento dependa del estado en que se encuentre (el ejemplo de la bibliografía es una conexión que puede tener los estados establecida, escuchando o cerrada).

Como ejemplo original podemos considerar un ticket en un sistema de seguimiento de requerimientos. Los tickets pasan por distintos estados de un workflow: “new”, “in progress”, “resolved”, “closed”, “rejected”, etc. Cada uno de estos estados es un objeto polimórfico con los otros que define comportamientos distintos sobre los mismos mensajes. Por ejemplo, un mensaje que habilite y deshabilite la edición de ciertos campos. En el estado “new” será posible escribir todos los campos, luego de que el ticket se asigna a un responsable el estado es ahora “in progress”. Mientras el ticket está asignado no puede ser actualizado ningún campo salvo el que indique el progreso realizado hasta el momento. Al único estado al que puede pasar es

“resolved” o “rejected”. Cuando cambia a estos estados se habilita la opción para hacer modificaciones sobre su conclusión pero ya no puede cargarse progreso sobre el mismo.

A continuación complete con una palabra la línea punteada

a. Un objeto-estado proviene de una “jerarquía” de estados posibles.

b. La clase “padre” define la interfaz que todos los estados tienen.

Para el dominio elegido en la primer pregunta, dé un ejemplo de un mensaje que el objeto forwardea a su estado y que cambia su propio comportamiento.

Actualizar el progreso porcentual (mensaje updateProgress) de un ticket que está en estado “in progress” lo lleva en algún momento al 100%. Dicho cambio forwardea un mensaje que cambia de estado y evita que se siga respondiendo a dicho mensaje cambiando el comportamiento.

- Composite

¿Qué estructura representa un Composite?

Un composite representa una estructura arbórea. Representa el todo y las partes de manera polimórfica.

¿Qué patrón se utiliza para responder los mensajes de un Composite que son polimórficos con los componentes del mismo?

Object-recursion pattern. El patrón define jerarquías de clases que consisten en objetos primitivos y objetos compuestos. Los objetos primitivos pueden usarse para componer objetos más complejos los cuales a su vez también pueden ser utilizados para componer otros objetos compuestos de manera recursiva.

¿Quién debe definir el protocolo de "administración" del Composite? (mensajes para agregar y sacar componentes del composite)

En el libro de GoF se menciona el criterio para decidir la definición del protocolo como un trade-off entre transparencia y seguridad. Si se define en el tope de la jerarquía (más transparencia), las hojas (o clases primitivas) de la estructura podrían saber responder los mensajes de administración. Esto no es ideal pero tal vez es necesario para trabajar con el patrón en lenguajes estáticamente tipados. Definir la administración solo en las clases compuestas da una mayor seguridad porque cualquier intento de agregar o quitar objetos de las hojas dará un error en tiempo de compilación en lenguajes estáticamente tipados. Pero se pierde transparencia porque las hojas y los objetos compuestos tendrán distintos protocolos.

- Visitor

¿Cuáles son los patrones en los que se basa un Visitor?

Se basa en method object y en una generalización del double dispatch.

¿Cuál es el protocolo que se utiliza para la generalización de double dispatch?

En la generalización se establece un protocolo que debe incluir los mensajes de "visita" (visit) y "aceptación" (accept). El mensaje visit se implementa con un método que contiene la lógica que aplicará el visitor para un tipo de nodo concreto. El mensaje accept queda del lado de los nodos que se visitan y se implementa con un método que contiene la lógica de cómo navegar la estructura. En el caso del "portfolio tree printer", los mensajes visit se implementan con la lógica de cómo recorrer el árbol de cuentas de un portfolio, mientras los de accept tienen que ver con la vuelta del dispatch que defina cómo imprimir una cuenta.

Cuando un visitor recorre un Composite, ¿quién tiene la responsabilidad de decidir cómo recorrer el composite?

El patrón visitor representa una forma de separar un algoritmo respecto de la estructura del objeto en el cual desea operar. Por ejemplo si se busca crear una forma genérica de recorrer una estructura arbórea, como lo es un composite, sin tener que agregar nuevos mensajes a las clases que representan dicha estructura. El composite permite recorrer dicha estructura siempre y cuando se haga siempre de la misma manera. Al utilizar un visitor, si bien eso requiere conocimiento interno acerca de la estructura del composite, la solución queda genérica y si son necesarias otras formas de hacer el recorrido alcanza con crear un nuevo visitor. Desde luego la decisión no deja de ser un trade off entre la necesidad de recorrer la estructura de una manera distinta a la que propone el composite y cierta violación del encapsulamiento para poder recorrer la estructura de una manera alternativa.