Verdaderas

Los bloques en Smalltalk son ejemplos de clausuras léxicas.

Puedo utilizar clausuras léxicas para indicar como se debe continuar cuando esté listo el resultado de una operación que se ejecuta de manera asincrónica (callback)

Seaside es un Framework para construir aplicaciones web siguiendo el estilo “paginas de servidor” (server pages). Es decir, cada vez que el usuario oprime un link, un botón, o envía un formulario, toda la pagina se recarga.

En un bloque de Smalltalk se puede hacer referencia a variables del contexto donde fue definido, por ejemplo, variables temporales o parámetros del método donde se define el bloque.

Puedo utilizar clausuras léxicas cuando quiero tener funciones que toman otras funciones como parámetro o que retornan funciones que luego voy a utilizar (por ejemplo, iteración)

Es posible cambiar la clase que Seaside usa para manejar las sesiones (es un hotspot del framework). Es necesario Subclasificar WASession y luego enviar el mensaje #sessionClass: a la aplicación.

Las clases en un framework se relacionan (herencia, conocimiento, envío de mensajes) de manera que resuelven la mayor parte del problema en cuestión – conforman un todo

Un framework es una aplicación “semicompleta”, “reusable”, que puede ser especializada para producir aplicaciones a medida

Cuando utilizamos librerías de clases, nuestro código controla/usa a los objetos de las librerías

Cada vez que un nuevo usuario ingresa al sistema se crea una nueva instancia de la sesión y del componente raíz (el de inicio)

Refactoring permite deshacerse de una jerarquía mal diseñada.

El refactoring Extract Method permite crear un método más reusable.

Se usa test doublé para no acarrear errores del objeto que provee los datos de entrada del método a testear.

El smell “Feature Envy” (Envidia de Atributo) marca un problema de mala asignación de responsabilidades.

La deuda técnica mide la acumulación de smells de código.

El patrón de diseño Adapter se enfoca en el problema de adaptación de interfases incompatibles

Usamos un patrón específico del catálogo del libro de Gamma cuando la naturaleza de nuestro problema de diseño coincide con la aplicabilidad del patrón

La "Estructura" de un patrón según la forma de descripción del libro de Gamma y que habitualmente mostramos en UML nos muestras los roles que cumplen las diferentes clases (y objetos) que participan en el patrón

Usamos el patrón Composite para cuando queremos representar estructuras parte-todo de objetos ignorando las diferencias entre ellos

Cuando usamos el template method en una aplicación (supongamos que lo llamamos t) t invoca operaciónes o1,o2, o3 que tienen diferentes implementaciones en cada clase concreta

El patrón Decorator se utiliza cuando subclasificar es impráctico.

El Proxy virtual se utiliza para representar objetos que se encuentran en una base de datos.

Respecto a la comparación entre patrones, tanto el patrón Composite como el Decorator tienen una estructura de clases similares que permite el anidamiento de objetos.

Cuando usamos el State el objeto contexto (que recibe los mensajes) los delega en su estado actual que es quien tiene el comportamiento correspondiente

Las plantillas y ganchos pueden implementarse utilizando herencia o composición como mecanismos para separar los aspectos comunes de los aspectos variables en un framework.

Con respecto a refactoring hacia patrones la sobre-ingeniería, es decir, tener un diseño sofisticado y complejo, es tan peligrosa como la baja-ingeniería, es decir, tener un diseño pobre

Si implemento plantillas y ganchos aplicando herencia, al implementar los métodos gancho puedo utilizar las variables de instancia y todo el comportamiento heredado de la clase abstracta (donde se implementa la plantilla)

La explosión en el número de subclases y la duplicación de código evidencian las limitaciones de la herencia como estrategia para separar plantillas y ganchos.

Cuando ya tengo código existente, el problema que soluciona Template Method es el relacionado con el smell “Duplicate Code” o Código duplicado

No es necesario escribir ni editar archivos «html para hacer una aplicación en Seaside. Todo el htm de la aplicación es generado por el framework a partir de código escrito en Smalltalk.

Puedo utilizar clausuras léxicas para indicar como se debe continuar cuando esté listo el resultado de una operación que se ejecuta de manera asincrónica (callback)

Puedo utilizar clausuras léxicas cuando quiero tener funciones que toman otras funciones como parámetro o que retornan funciones que luego voy a utilizar (por ejemplo, iteración)

Los bloques en Smalltalk son ejemplos de clausuras léxicas.

Para pintar en el canvas (lienzo) utilizo pinceles. A los pinceles le envío el mensaje #with: para indicarle qué quiero que pinte. El parámetro puede ser un String o un bloque. Hay pinceles para pintar hipervínculos, párrafos, listas, etc.

Todas las aplicaciones construidas con un mismo framework tienen aspectos en común que no podemos cambiar (esos aspectos constituyen el Frozenspot)

Es  posible cambiar la clase que Seaside usa para manejar las sesiones (es un hotspot del framework). Es necesario Subclasificar WASession y luego enviar el mensaje #sessionClass: a la aplicación.

El código que hace algo con la interfaz de usuario, solo debe estar relacionado a la interfaz de usuario. Le llamamos “código de presentación”

Un framework es un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas

Cada clase en una librería resuelve un problema concreto, es independiente del contexto de uso, no espera nada de nuestro código y es generalmente independiente de otras clases en la librería

Permite diseñar en forma incremental, partiendo de un diseño simple y esperando ver cómo funciona en su contexto antes de seguir avanzando.

Crear un método más reusable.

Crear el fixture, ejercitarlo, comprobar el resultado y eliminar el fixture.

Ante un cambio de requerimientos debo modificar código en muchas clases.

Chequean precondiciones con el análisis del árbol de sintaxis y luego aplican los cambios.

Cuando queremos representar estructuras parte-todo de objetos ignorando las diferencias entre ellos

Un objeto a de A conozca una instancia ad de AD la cual a su vez conozca la instancia c de C con la cual se comunica

t invoca operaciónes o1,o2, o3 que tienen diferentes implementaciones en cada clase concreta

Nos muestras los roles que cumplen las diferentes clases (y objetos) que participan en el patrón

Los objetos de A conocen una instancia de AD. Las instancias de AD conocen una instancia de C. Las instancias de AD responden al ménsaje m (con un método m) enviando el mensaje m1 a la instancia de C y retornan el resultado al objeto de A que les envió m

El patrón Decorator permite tener mayor flexibilidad en tiempo de ejecución que una solución con herencia

El propósito de Proxy es proporcionar un intermediario para controlar el acceso al objeto real

El patrón Strategy cambia la funcionalidad del objeto por dentro y el Decorator la cambia por fuera

El objeto contexto (que recibe los mensajes) los delega en su estado actual que es quien tiene el comportamiento correspondiente

Tenemos un método que puede tener múltiples implementaciones y no queremos tomar la decisión de cual usar con una estructura IF

Las plantillas y ganchos pueden implementarse utilizando herencia o composición como mecanismos para separar los aspectos comunes de los aspectos variables en un framework

Cuando aplico un refactoring hacia un patrón determinado, puedo quedar a mitad de camino con la mecánica y ya haber logrado una mejora suficiente en el diseño.

Un punto a favor de la implementación de plantillas y ganchos por composición es que permite cambiar el comportamiento del objeto que implementa la plantilla en tiempo de ejecución.

La explosión en el número de subclases y la duplicación de código evidencian las limitaciones de la herencia como estrategia para separar plantillas y ganchos.

Elimina el código duplicado pero puede causar la creación de muchos métodos vacíos o con comportamiento por defecto forzados por el Template Method

El patron “Template Method”, aplicado en un punto de extensión de un framework produce inversión de control.

La plantilla del patrón “Template Method” utilizada para generar un punto de extensión en framework, constituye parte del frozenspot

La estrategia de plantillas y ganchos para introducir puntos de extensión en un framework puede implementarse en base a herencia o en base a composición de objetos.

Un punto de extensión de un framework implementado en base a plantillas y ganchos, por composición de objetos, permite alterar (cambiar) comportamiento en tiempo de ejecución.

Componer objetos en estructuras de árbol para representar jerarquías parte-todo es un indicador de aplicabilidad del patrón Composite.

Que los clientes tratan uniformemente a objetos individuales y a composiciones es un indicador de aplicabilidad del patrón Composite

Para agregar responsabilidades a objetos en forma dinámica y transparente, es decir, sin afectar a otros objetos es un indicador de aplicabilidad del patrón Decorator

Un método largo indica la necesidad de hacer refactoring

El refactoring “Extract Method” (Extraer Método) puede aplicarse para eliminar el mal olor “método largo”

El refactoring es un proceso iterativo y continuo de mantenimiento de código

El refactoring no debe alterar la funcionalidad ofrecida por el artefacto de software al que se aplica

El refactoring mejora al diseño de un artefacto de software

Falsas

Cuando se define un bloque en Smalltalk dentro de un método, no se puede hacer referencia a variables de instancia; solo es posible hacer referencia a variables temporales y parámetros del método.

Los bloques son muy potentes, fácil de usar y de depurar. Siempre que puedo utilizo bloques para parametrizar el comportamiento de una clase (por ejemplo, en lugar de definir subclases).

Cuando se evalúa un bloque en Smalltalk las variables del contexto a las que hace referencia se pasan al bloque por copia. Por lo tanto los cambios a dichas variables fuera del bloque no afectan el valor de la variable dentro del bloque.

Cuando se define un bloque en Smalltalk dentro de un método, no se puede hacer referencia a variables de instancia; solo es posible hacer referencia a variables temporales y parámetros del método.

Seaside es un framework para construir aplicaciones móviles (de carga progresiva). Es decir, las aplicaciones que se descargan completamente al dispositivo móvil y solo se conectan al servidor para intercambiar datos.

Para hacer una aplicación en Seaside creo archivos de plantillas escribiendo código HTML que incluye expresiones Smalltalk que serán procesadas por el servidor.

Los bloques son muy potentes, fácil de usar y de depurar. Siempre que puedo utilizo bloques para parametrizar el comportamiento de una clase (por ejemplo, en lugar de definir subclases).

Cuando se evalúa un bloque en Smalltalk las variables del contexto a las que hace referencia se pasan al bloque por copia. Por lo tanto los cambios a dichas variables fuera del bloque no afectan el valor de la variable dentro del bloque.

Cuando se evalúa un bloque en Smalltalk las variables del contexto a las que hace referencia se pasan al bloque por copia. Por lo tanto los cambios a dichas variables fuera del bloque no afectan el valor de la variable dentro del bloque.

En un bloque que se define dentro de un método no se puede hacer referencia a self.

Cuando un componente Seaside quiere pasar el control a otro componente le envía el mensaje #call: y se pasa a si mismo como parámetro.

Para que los componentes Seaside en distintas sesiones compartan un mismo modelo, subclasifico la clase WASession e implemento su método #initialize para que instancie al modelo.

Llamamos “código de presentación” a todo modelo de objetos del dominio que representa reportes o informes aunque el mismo sea independiente de la interfaz de usuario.

Cuando utilizamos una librería de clases el código de la librería controla al nuestro (llamamos a esto inversión de control)

Con Seaside, cada vez que un nuevo usuario ingresa a la aplicación se crea una nueva instancia de la sesión. Sin embargo todos los usuarios comparten una misma instancia del componente raíz (el de inicio). A esto se le llama Singleton o instancia única. Incorrecta

Las clases en una librería de clases se relacionan (herencia, conocimiento, envío de mensajes) de manera que resuelven la mayor parte del problema en cuestión – conforman un todo

Cuando utilizamos frameworks, nuestro código controla/usa a los objetos del framework (llamamos a esto flujo del control)

Una librería de clases es una aplicación “semicompleta”, “reusable”, que puede ser especializada para producir aplicaciones a medida.

Una librería de clases es un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas.

Una librería de clases es un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas .

Cuando utilizamos frameworks, nuestro código controla/usa a los objetos del framework (llamamos a esto flujo del control)

Una librería de clases es una aplicación “semicompleta”, “reusable”, que puede ser especializada para producir aplicaciones a medida

Un componente que activa a otro debe pasarle los objetos que necesita. Una posibilidad es que deje la información en variables globales (recomendable). Otra es que deje lo necesario en el objeto Session (no tan recomendable). O podría enviarle mensajes con la información antes de activarlo (peor alternativa).

En aplicaciones con interfaz gráfica, el código de dominio puede interactuar (enviar mensajes) con el código de presentación si se utiliza un framework de caja negra.

Llamamos “código de presentación” a todo modelo de objetos del dominio que representa reportes o informes aunque el mismo sea independiente de la interfaz de usuario.

Refactoring permite eliminar bugs en el código.

Refactoring permite agregar nueva funcionalidad en forma ordenada.

El refactoring Extract Method permite mover métodos en una jerarquía.

El refactoring Extract Method permite reasignar responsabilidades entre clases.

Se usa test doublé para para no acarrear errores en la superclase de la clase a testear.

Se usa test doublé para para no duplicar el código que crea los objetos del fixture de un test.

El smell “Feature Envy” (Envidia de Atributo) marca un problema de mal uso de variables de instancia y temporales.

El smell “Feature Envy” (Envidia de Atributo) marca un problema de mala jerarquía de herencia (envidia entre clase y subclase).

La deuda técnica mide la cobertura de los tests de unidad.

La deuda técnica mide la cantidad de refactorings aplicados.

La deuda técnica mide funcionalidad que falta implementar.

El patrón de diseño Adapter permite que enriquezcamos la interfaz de una clase con nueva funcionalidad

El patrón de diseño Adapter nos provee objetos reusables que podemos usar para adaptar interfaces

Usamos un patrón específico del catálogo del libro de Gamma cuando la jerarquía de clases en nuestra aplicación se parece mucho a la estructura del patrón

Usamos un patrón específico del catálogo del libro de Gamma cuando en nuestro problema aparece un requerimiento cuyo nombre coincide exactamente con el nombre del patrón

La "Estructura" de un patrón según la forma de descripción del libro de Gamma y que habitualmente mostramos en UML contiene clases que debemos instanciar para poder usar el patrón en ejemplos concretos

La "Estructura" de un patrón según la forma de descripción del libro de Gamma y que habitualmente mostramos en UML debe ser copiada y pasteada en nuestro sistema para poder usarla

Usamos el patrón Composite para representar un tipo de herencia específica entre el todo y las partes que permita que las partes hereden del todo

Usamos el patrón Composite para cuando tenemos relaciones jerárquicas como las que existen en una organización (director, jefe, empleado)

Cuando usamos el template method en una aplicación (supongamos que lo llamamos t) tenemos que escribir métodos t en cada sub-clase e invocar a t usando super

Cuando usamos el template method en una aplicación (supongamos que lo llamamos t) Las operaciones primitivas o1, 02 y 03 definidas en las sub-clases usan a t mediante el mecanismo de herencia

Cuando usamos el patrón Decorator y debe agregarse un nuevo decorador, se le pide al objeto decorado que lo agregue.

Cuando usamos el patrón Decorator, el objetos decorado conoce a todos sus decoradores.

El propósito de Proxy es agregar responsabilidades a un objeto dinámicamente.

El Proxy de protección le delega al objeto real la responsabilidad de chequear si el cliente tiene los permisos necesarios para accederlo.

Respecto a la comparación entre patrones, el patrón Adapter cambia la funcionalidad del objeto por dentro y el Decorator la cambia por fuera.

Respecto a la comparación entre patrones, tanto el patrón Composite como el Decorator forman una estructura de árbol en sus instancias.

Cuando usamos el State chequea si en el estado actual puede o no responder dicho mensaje

Cuando usamos el State utiliza un template method que en las sub-clases determina cual comportamiento se debe ejecutar

Cuando usamos el Strategy una vez seteada una estrategia todos los objetos de la clase Context usarán dicha estrategia

Cuando usamos el Strategy el cliente envía la estrategia que quiere usar y el recepto decide como activarla

Cuando usamos el Strategy diferentes objetos de la clase Contexto pueden tener distintas estrategias asociadas en un momento determinado y estas se pueden cambiar dinámicamente

Si utilizo herencia, al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia definidas en la clase que implementa la plantilla.

Con herencia en la implementación de plantillas y ganchos se evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles

El patrón método plantilla (implementado como se describe en el libro de patrones) es la única forma de introducir variabilidad en un framework, y por eso decimos que es un generador de hotspots e inversión de control.

Con respecto a refactoring hacia patrones la sobre-ingeniería es preferible y recomendable sobre la baja o poca ingeniería

El patrón método plantilla (implementado como se describe en el libro de patrones) es la única forma de introducir variabilidad en un framework, y por eso decimos que es un generador de hotspots e inversión de control.

Si se implementa plantillas y ganchos por composición, NO es posible (o es muy complejo) cambiar en tiempo de ejecución el comportamiento implementado por los ganchos. Si se implementa por herencia, SI es posible.

Con herencia en la implementación de plantillas y ganchos se evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles

Si se implementa plantillas y ganchos por composición, NO es posible (o es muy complejo) cambiar en tiempo de ejecución el comportamiento implementado por los ganchos. Si se implementa por herencia, SI es posible.

Con herencia en la implementación de plantillas y ganchos se evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles

En el contexto de un framework, los usuarios del framework implementan métodos plantilla mientras que el desarrollador del framework implementa los ganchos correspondientes a esas plantillas.

Cuando ya tengo código existente, el problema que soluciona Template Method es el relacionado con el smell “Pull Up Method”.

Cuando ya tengo código existente, el problema que soluciona Template Method es el relacionado con el smell “Data Class” o Clase de datos.

Para hacer una aplicación en Seaside creo archivos de plantillas escribiendo código HTML que incluye expresiones Smalltalk que serán procesadas por el servidor.

Seaside es un framework para construir aplicaciones móviles (de carga progresiva). Es decir, las aplicaciones que se descargan completamente al dispositivo móvil y solo se conectan al servidor para intercambiar datos.

Cuando se define un bloque en Smalltalk dentro de un método, no se puede hacer referencia a variables de instancia; solo es posible hacer referencia a variables temporales y parámetros del método.

Cuando se evalúa un bloque en Smalltalk las variables del contexto a las que hace referencia se pasan al bloque por copia. Por lo tanto los cambios a dichas variables fuera del bloque no afectan el valor de la variable dentro del bloque.

Cuando se evalúa un bloque en Smalltalk las variables del contexto a las que hace referencia se pasan al bloque por copia. Por lo tanto los cambios a dichas variables fuera del bloque no afectan el valor de la variable dentro del bloque.

En un bloque que se define dentro de un método no se puede hacer referencia a self.

Cuando se define un bloque en Smalltalk dentro de un método, no se puede hacer referencia a variables de instancia; solo es posible hacer referencia a variables temporales y parámetros del método.

Los bloques son muy potentes, fácil de usar y de depurar. Siempre que puedo utilizo bloques para parametrizar el comportamiento de una clase (por ejemplo, en lugar de definir subclases).

Una aplicación Seaside consiste de componentes. Puedo utilizar como componente cualquier objeto que entienda el mensaje #printOn:, pero solo si antes lo registro en el servidor.

Para hacer una aplicación en Seaside creo archivos de plantillas escribiendo código HTML que incluye expresiones Smalltalk que serán procesadas por el servidor.

Cada clase en un framework resuelve un problema concreto, es independiente del contexto de uso, no espera nada de nuestro código y es generalmente independiente de otras clases en el framework.

Una librería de clases es una aplicación “semicompleta”, “reusable”, que puede ser especializada para producir aplicaciones a medida.

En aplicaciones con interfaz gráfica, el código de dominio puede interactuar (enviar mensajes) con el código de presentación si se utiliza un framework de caja negra.

Cuando un componente Seaside quiere pasar el control a otro componente le envía el mensaje #call: y se pasa a si mismo como parámetro.

Llamamos “código de presentación” a todo modelo de objetos del dominio que representa reportes o informes aunque el mismo sea independiente de la interfaz de usuario.

Para que los componentes Seaside en distintas sesiones compartan un mismo modelo, subclasifico la clase WASession e implemento su método #initialize para que instancie al modelo.

Para que los componentes Seaside en distintas sesiones compartan un mismo modelo, subclasifico la clase WASession e implemento su método #initialize para que instancie al modelo.

Cuando un componente Seaside quiere devolver el control al componente que lo activó, se envía a si mismo el mensaje #answer: con el componente activador como parámetro.

Cuando un componente Seaside quiere pasar el control a otro componente le envía el mensaje #call: y se pasa a si mismo como parámetro.

En aplicaciones con interfaz gráfica, el código de dominio puede interactuar (enviar mensajes) con el código de presentación si se utiliza un framework de caja negra.

Una librería de clases es un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas.

Una librería de clases es una aplicación “semicompleta”, “reusable”, que puede ser especializada para producir aplicaciones a medida.

Los frameworks orientados a objetos resuelven problemas comunes a la mayoría de las aplicaciones (p.e., manejo de archivos, funciones aritméticas, colecciones, fechas)

Cada clase en un framework resuelve un problema concreto, es independiente del contexto de uso, no espera nada de nuestro código y es generalmente independiente de otras clases en el framework.

Cuando utilizamos frameworks, nuestro código controla/usa a los objetos del framework (llamamos a esto flujo del control)

Permite diseñar el sistema completo con toda la complejidad que haga falta al principio y luego modificarlo según cómo funciona.

Mover métodos en una jerarquía.

Reasignar responsabilidades entre clases.

Escribir el test, codificar lo necesario para que el test pase y refactorizar.

Ejercitar la unidad y chequear.

Crear el fixture y comprobar el resultado de la creación.

La misma clase se modifica ante distintos cambios de requerimientos de diferentes maneras y por diferentes razones.

Una clase tenga baja cohesión.

Miden la deuda técnica que significa la cantidad de refactorings aplicados.

Chequean la semántica del programa antes de aplicar refactoring, por ejemplo, haciendo análisis de posibles valores de parámetros.

Realizan las transformaciones sin ningún tipo de chequeo previo, dejando el chequeo en manos del programador.

Para representar un tipo de herencia específica entre el todo y las partes que permita que las partes hereden del todo

Cuando tenemos relaciones jerárquicas como las que existen en una organización (director, jefe, empleado)

El objeto c de C con el cual necesitamos interactuar conozca una instancia ad de AD para realizar la adaptación

 Los objetos ad de AD puedan interactuar con objetos a de A para poder resolver los requerimientos de los objetos c de C

Tenemos que escribir métodos t en cada sub-clase e invocar a t usando super

 Las operaciones primitivas o1, 02 y 03 definidas en las sub-clases usan a t mediante el mecanismo de herencia

Contiene clases que debemos instanciar para poder usar el patrón en ejemplos concretos

Debe ser copiada y pasteada en nuestro sistema para poder usarla

Los objetos de A conocen objetos de C y dichos objetos conocen instancias de AD. El objeto de C que recibe el mensaje m1 delega dicho mensaje en la instancia de AD correspondiente, la cual resuelve el requerimiento m

Un objeto de A se comunica con el objeto de AD al que conoce el cual al recibir el mensaje m elige entre los objetos de C cual es el objeto que debe "adaptar" y le envía el mensaje m1

El propósito del patrón Decorator es generar una composición de objetos

Decorador y Decorado pertenecen a distintas jerarquías de clase.

El Proxy remoto se utiliza para representar objetos que se encuentran en una base de datos.

El Proxy virtual se utiliza para representar objetos en una arquitectura distribuida.

Los 3 wrappers tienen al wrapper en la misma jerarquía que el objeto real.

Los 3 wrappers permiten anidamiento.

Chequea si en el estado actual puede o no responder dicho mensaje

Utiliza un template method que en las sub-clases determina cual comportamiento se debe ejecutar

Hay varios comportamientos diferentes en un objeto y queremos poder seleccionar correctamente cual usar

Tenemos una jerarquía de clases con múltiples implementaciones de un método en las sub-clases y queremos usar un template method

El patrón método plantilla (implementado como se describe en el libro de patrones) es la única forma de introducir variabilidad en un framework, y por eso decimos que es un generador de hotspots e inversión de control.

Si utilizo herencia, al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia definidas en la clase que implementa la plantilla.

Con herencia en la implementación de plantillas y ganchos se evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles

Cuando aplico un refactoring hacia un patrón determinado, debo completar la mecánica hasta aplicar el patrón completamente.

En el contexto de un framework, los usuarios del framework implementan métodos plantilla mientras que el desarrollador del framework implementa los ganchos correspondientes a esas plantillas.

Si se implementa plantillas y ganchos por composición, NO es posible (o es muy complejo) cambiar en tiempo de ejecución el comportamiento implementado por los ganchos. Si se implementa por herencia, SI es posible.

 Cuando hablamos de implementar plantillas y ganchos por composición, nos referimos a combinar el patrón template method con el patrón composite.

Si se implementa plantillas y ganchos por composición, NO es posible (o es muy complejo) cambiar en tiempo de ejecución el comportamiento implementado por los ganchos. Si se implementa por herencia, SI es posible.

En el contexto de un framework, los usuarios del framework implementan métodos plantilla mientras que el desarrollador del framework implementa los ganchos correspondientes a esas plantillas.

Con herencia en la implementación de plantillas y ganchos se evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles

Todas positivas al eliminar la duplicación; no tiene contras.

Elimina el código duplicado pero puede causar Data class.

Siempre que se observa inversión de control en el punto de extensión de un framework, se debe a la existencia de un “Template Method” en su frozenspot

La plantilla del patrón “Template Method” en un punto de extensión de un framework, representa un hotspot

Dado un patrón de diseño cualquiera aplicado al desarrollo de un framework, el objeto con el rol “cliente” es parte del frozenspot, mientras que el objeto con el rol “sujeto” es parte de algún hotspot

Un punto de extensión de un framework implementado en base a plantillas y ganchos, por herencia, permite alterar (cambiar) comportamiento en tiempo de ejecución.

La estrategia de plantillas y ganchos para introducir puntos de extensión en un framework puede implementarse en base a herencia y con una clase abstracta que implementa la plantilla.

Convertir la interfaz de una clase a otra interfaz que los clientes esperan es un indicador de aplicabilidad del patrón Composite

Agregar comportamiento a un objeto dinámicamente y en forma transparente es un indicador de aplicabilidad del patrón Composite

Definir el esqueleto de un algoritmo en una operación, delegando algunos pasos a las subclases es un indicador de aplicabilidad del patrón Composite

Para mantener bajo control las extensiones que pueden hacer las subclases es un indicador de aplicabilidad del patrón Decorator

Para crear una clase reutilizable que coopera o cooperara en el futuro con otras no relacionadas, es decir, clases que no necesariamente tendrán interfaces (protocolos) compatibles es un indicador de aplicabilidad del patrón Decorator

Cuando el comportamiento en común entre varias subclases debe ser factorizado y ubicado en una superclase común para evitar duplicación de código es un indicador de aplicabilidad del patrón Decorator

“Comment every important line” (comentar cada línea importante) puede aplicarse para eliminar el mal olor método largo

El refactoring “Replace Temp With Query” (reemplazar temporal con consulta) puede aplicarse para eliminar el mal olor “método largo”

Un método es largo si tiene más de 6 líneas (o 160 caracteres)

Los refactorings (según Fowler) son estrategias que ayudan a eliminar errores frecuentes (bugs) del software

Según Fowler, solo debemos invertir esfuerzo en refactoring cuando el código se vuelve ilegible