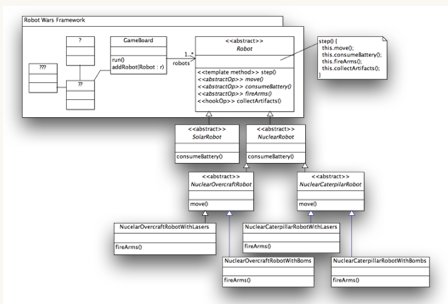
**Plantillas y ganchos**

**Objetivos**

Las plantillas y los ganchos son una estrategia de programación comúnmente utilizada para introducir puntos de variabilidad (hotspots) en los frameworks orientados a objetos. Pueden implementarse utilizando herencia o composición como mecanismos para separar los aspectos comunes de los aspectos variables. El objetivo de este material es que usted pueda reconocer ambas estrategias de implementación del concepto de plantilla y ganchos.



**Destinatarios**

Este material está destinado a desarrolladores de software y/o alumnos de carreras afines al desarrollo de software, interesados en conocer algunos de los mecanismos fundamentales del desarrollo de software reusable.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Preliminares

Antes de avanzar, asegúrese de que cuenta con los conocimientos requeridos para aprovechar al máximo este material.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

**Conocimientos previos**

**Conocimiento previo**

Para aprovechar este contenido usted debe estar familiarizado con los conceptos de: framework orientado a objetos, inversión de control, hotspot y frozenspot, y el patrón de diseño Método Plantilla.

A continuación se ofrecen un conjunto de preguntas que le ayudarán a confirmar que cuenta con los conocimientos previos. De no ser así, es recomendable que lea el material sugerido como lectura previa recomendada.

**Cuestionario SCORM**

¿Qué es el Método Plantilla?



Un patrón de diseño



Un método que se puede reutilizar como si fuera una plantilla



Un método de programación que se basa en construir plantillas

¿Cuál es el propósito del método plantilla?



Proporciona una plantilla para crear familias de objetos relacionados o que dependen entre sí, sin especificar clases concretas



Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.



Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes, por medio de una plantilla. Permite que cooperen clases que, de otra forma, no podrían hacerlo por tener interfaces incompatibles.

¿Qué es un hotspot?



Una parte crítica de un programa (hot-spot), que requiere ser mejorada



Una parte caliente (hot) de un diseño en la que se producen problemas frecuentes.



Un punto de variabilidad de un framework orientado a objetos

¿Qué es un frozenspot?



Un espacio con temperatura controlada para almacenar servidores



Una parte del programa que nunca se ejecuta (por eso está congelada, frozen)



El comportamiento invariable en un framework orientado a objetos



Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Lectura previa recomendada

Para refrescar los conceptos relativos al patrón Método Plantilla se recomienda leer el capitulo correspondiente del libro:

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software.

Si no tuviera acceso al mismo, la página [Método Plantilla](https://en.wikipedia.org/wiki/Template_method_pattern) en Wikipedia le será de utilidad. Para su comodidad, incluimos la versión pdf de dicho articulo (al día 16/12/2015) en la sección de ficheros adjuntos.

Para refrescar conceptos de frameworks, hotspots y frozenspots le recomendamos lea el artículo:

Roberts, Don, and Ralph Johnson. "Evolving frameworks: A pattern language for developing object-oriented frameworks." Pattern languages of program design 3 (1996): 471-486. 

En ese artículo los autores describen el proceso de desarrollo de frameworks. Hablan de la importancia de conocer bien el dominio antes de comenzar; hablan de blackbox y whitebox, hablan de hotspots, y hablan de herramientas para asistir a los programadores en el uso/instanciación del framework. Discute que patrones de diseño pueden ser de utilidad en cada paso del desarrollo del framework. El mismo puede descargarse [desde el sitio del autor](http://carfield.com.hk/document/software%20design/Evolving%20Frameworks:%20A%20Pattern%20Language.pdf) o de la sección de ficheros adjuntos, a continuación.

## Ficheros adjuntos

* [Evolving frameworks: A pattern language for developing object-oriented frameworks (pdf) (Ventana nueva)](https://catedras.linti.unlp.edu.ar/pluginfile.php/85076/mod_imscp/content/3/roberts96.pdf)
* [Template Method Design Pattern. Wikipedia. 16 de Diciembre de 2015 (Ventana nueva)](https://catedras.linti.unlp.edu.ar/pluginfile.php/85076/mod_imscp/content/3/Template%20method%20pattern.pdf)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Ejemplo conductor

Imagen ilustrativa: Roborally, el juego de mesa

A lo largo de este material trabajaremos sobre un mismo ejemplo. Se trata de un framework que permite generar distintas variantes de juegos de lucha de robots.

### Frozenspot

Todos los juegos generados con este framework trendrán los siguientes aspectos en común (hablamos del **frozenspot** del framework):

* Un tablero en forma de grilla
* Varios robots, los cuales estarán ubicados en alguna celda del tablero
* El juego se organiza en pasos. A cada paso del juego, corresponde un paso de cada uno de los robots en el tablero.
* Cada paso de un robot consiste de las siguientes acciones en estricto orden:
  + moverse en el tablero
  + consumir energía (actualizar su registro de energía)
  + disparar su armamento
  + recolectar lo que pudiera haber en la celda del tablero que ocupa

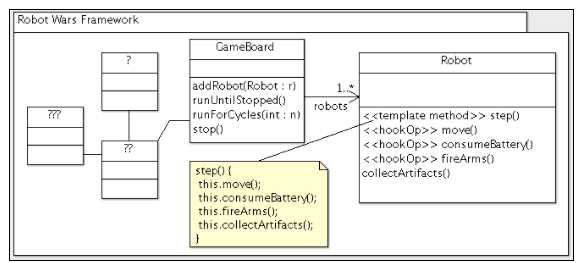
### Hostpots

Un framework como este seguramente contará con varios puntos de variabilidad (hostpots) que darán lugar a distintas variantes de juego. Algunos hotspots permiten variar características del tablero (no hablaremos de estos). Otros permiten obtener variantes de robots. Estos últimos son los que nos interesarán. El framework define tres hostpots que permiten obtener distintos tipos de robots.

1. locomoción: este hotspot permite definir de que forma se moverá el robot. Dos alternativas para este hotspot son locomoción con orugas (caterpillar) que permite subir grandes pendientes y locomoción sobre colchón de aire (overcraft) que permite circular por zonas pantanosas.
2. fuente de energía: este hotspot permite definir de que forma el robot obtendrá/almarenará/consumira su energía. Dos alternativas para este hotspot son los paneles solares con baterías (muy estables pero sensibles a la falta de luz solar) y el reactor nuclear (poco estable, pero garantiza la provisión energética)
3. armamento: este hotspot permite definir de que forma el robot atacará a otros. Dos alternativas para este hotspot son el láser (mucho alcance y precisión pero poco poder de daño) y las bombas (poco alcance y precisión, y mucho poder de daño)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Diseño general



La imagen muestra los aspectos mas interesantes de nuestro framework ejemplo. Nos interesan particularmente las clases GameBoard (el tablero del juego) y Robot. Los pasos básicos para construir un juego con nuestro framework son:

1) se instancia el GameBoard (lo que resultará en la instanciación y configuración de muchos otros objetos que hacen al frozenspot del framework).

2) se instancian varios Robots y se agregan al juego enviando el mensaje *addRobot()*al *gameBoard*.

3) se inicia el juego enviando al *gameBoard* el mensaje*runUntilStopped() .*Si se quiere detener el juego se envía al gameBoard el mensaje stop(). Alternativamente se puede activar el juego solo por un determinado numero de pasos enviando al gameBoard el mensaje runForCycles(int n).

### El método plantilla

En cada paso del juego (cada ciclo) el gameBoard recorre su colección de robots y a cada uno le envía el mensaje *step()*. Note que el método *step()* es donde se garantiza que el robot efectúe las acciones establecidas en las reglas del juego (frozenspot), en el orden establecido.

Pensando en el patron de diseño "método plantilla" uno podría implementar variantes de robots subclalsificando la clase Robot y redefiniendo los métodos *move(), consumeBattery(), y fireArms()*. De hecho, cada uno de estos métodos hace posible la implementación de uno de nuestros hostspots (recuerde que no siempre un hotspot se corresponde/implementa con un método o una clase, pero casualmente eso es lo que ocurre en nuestro diseño ejemplo)

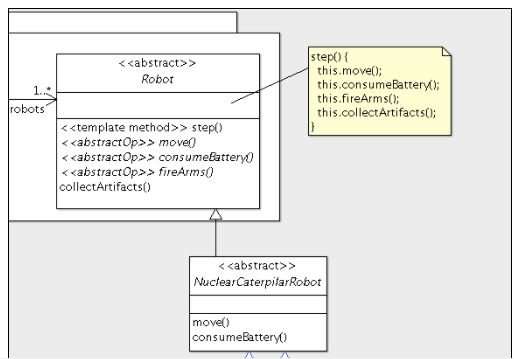
Subclasificar es una de las alternativas de diseño posibles cuando se utiliza la estrategia de la plantilla y los hooks para separar lo que se mantiene constante (fronzenspot) de lo que varía (los hotspots).  La otra estrategia es separar lo constante de lo que cambia componiendo objetos. A continuación veremos en detalle ambas alternativas.

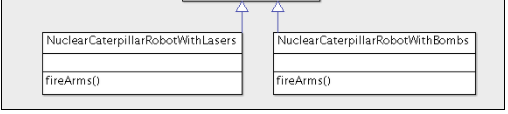
Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

**Plantillas y herencia**

El patron de diseño "metodo plantilla" documentado en el libro "Patrones de diseño" (de Gamma y otros) propone abstraer los pasos canónicos de la operación en un método implementado en la clase abstracta - en nuestro caso step() - y redefinir los pasos variables en sus subclases. De esta forma, la herencia es el mecanismo principal para separar lo que es constante, de lo que varía. El método plantilla es parte del frozenspot mientras que las operaciones abstractas (o los ganchos) dan lugar a la implementación de los hotspots.

Note, en la siguiente figura, como las subclases de Robot redefinen las operaciones abstractas dando lugar a variantes de robot.

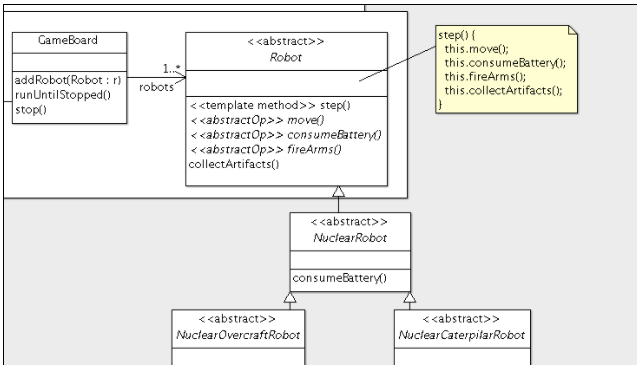


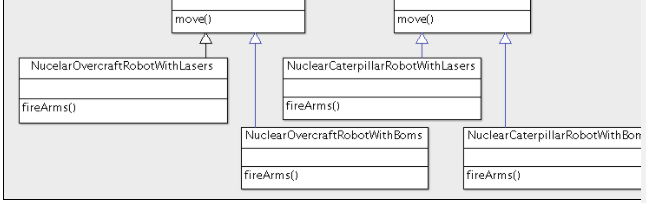


Tenemos dos tipos de Robots que pueden agregarse a nuestro Juego. Los robots con orugas, alimentados con energía nuclear y armados lasers (NuclearCaterpillarRobotWithLasers) y los robots con orugas, alimentados con energía nuclear y armados bombas (NuclearCaterpillarRobotWithBombs). Ambas clases de robots tienen el mismo comportamiento de locomoción (orugas) y relativo a su fuente de energía (nuclear) . Dicho comportamiento común se ha abstraído en la clase abstracta NuclearCaterpillarRobot. En las subclases concretas quedan la implementaciones del método fireArms(), dado que difieren una de otra.

La subclasificación es un mecanismo sencillo para especializar el comportamiento de nuestros robots. Las subclases no solo pueden redefinir el comportamiento heredado sino que tienen acceso a todas las variables de instancia del robot (las heredadas, si las hubieras y las definidas en la subclase).

Pero note en la siguiente imagen lo que sucede a medida que aparecen nuevas alternativas para cada hotspot y si se intentan combinar arbitrariamente (cosa que sería deseable).





En este diseño, se ha subclasificado la clase Robot con el objetivo de obtener cuatro clases de robots.

|  | fuente de energía | Locomoción | armamento |
| --- | --- | --- | --- |
| NuclearOvercraftRobotWithLasers | nuclear | colchón de aire | lasers |
| NuclearOvercraftRobotWithBombs | nuclear | colchón de aire | bombas |
| NuclearCaterpillarRobotWithLasers | nuclear | Orugas | lasers |
| NuclearCaterpillarRobotWithBombs | nuclear | Orugas | bombas |

Todas ellas son idénticas en lo que se refiere a fuente de energía (nuclear). La clase abstracta NuclearRobot abstrae ese comportamiento común, que queda encapsulado en el método consumeBattery() heredado por las 4 clases concretas de robot. Luego, dos de las clases de robots implementan locomoción por orugas (caterpillar). Este comportamiento común se encapsula en el método move() de la clase abstracta NuclearCaterpilarRobot. Las otras dos clases implementan locomoción por colchón de aire (overcraft). Ese comportamiento común se encapsula en el método move() de la clase abstracta NuclearOvercraftRobot.

Observe que el comportamiento de armamento se encapsula en el método fireArms(). Necesitamos que una de las subclases de NuclearCaterpillarRobot implemente fireArms() para obtener comportamiento de armamento con lasers (NuclearCaterpillarRobotWithLasers). Un método fireArms() idéntico se requiere en una de las subclases de NuclearOvercraftRobot. Algo similar ocurre con el método fireArms() correspondiente a la implementación de armamento con bombas. No hay forma de abstraer ese comportamiento en una clase abstracta (para evitar duplicar código) sin reformular toda la jerarquía y que ocurra algo similar con otro de los comportamientos variables.

La explosión en el numero de subclases y la duplicación de código evidencian las limitaciones de la herencia como estrategia para separar plantillas y ganchos. En situaciones como estas puede explorarse la composición como una estrategia alternativa.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Ejercicio - ejemplo

## Ficheros adjuntos

El archivo adjunto (un file-out de Pharo Smalltalk) ofrece una implementación muy simple del framework en base a lo visto en esta sección (es decir, utilizando herencia para separar la plantilla de los ganchos).

Al hacer el file-in, se debería abrir automáticamente un workspaces con los ejemplos de uso, y browsers en las clases involucradas. El código es muy simple, solo escribe mensajes en el Transcript.

Al mirar el código otra preguntense:

* ¿Que debo hacer si aparece una nueva fuente de energía (por ejemplo, Paneles solares con baterias)? ¿Cuantas y cuales clases debo agregar en caso de querer todas las variantes de robots posibles para este nuevo tipo de fuente de energía?
* ¿Puedo cambiarle, a un robot existente, el sistema de armas sin tener que instanciar el robot de nuevo?
* ¿Donde almacenaría usted el nivel de carga de la batería? ¿Que implicaría eso si antes de disparar el laser hay que garantizar que la fuente de energía puede satisfacer el consumo del arma?

**Descargas**:

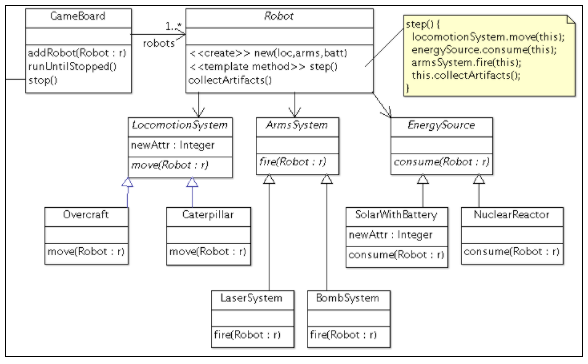
* [Implementación del framework de robots utilizando herencia (Ventana nueva)](https://catedras.linti.unlp.edu.ar/pluginfile.php/85076/mod_imscp/content/3/Hotspots-Inheritance.st.zip)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

**Plantillas y composición**

En la sección anterior vimos como el comportamiento constante (parte del frozenspot) se encapsula en el método plantilla y se separa del comportamiento variable (hotspots) que se implementa en los ganchos, en las subclases. Recuerde, que si bien el comportamiento constante se separa del comportamiento variable en métodos diferentes, al instanciar los robots, ambos comportamientos (grupos de métodos) pasan a formar parte de un único objeto (el robot con los métodos de su clase y los que hereda). Esto, unido al hecho de que la herencia simple no puede lidiar bien con la combinación y abstracción de comportamientos ortogonales, resultó en una explosión de clases y en la duplicación de código.

En diseño que muestra la figura profundiza mas la separación de lo que es constante y lo que varía. No solo lo separa en métodos diferentes, sino que también lo encapsula en objetos diferentes.



El primer cambio a observar en este diseño es que un robot ahora tiene tres partes. Un sistema de locomoción, un sistema de armamento, y una fuente de energía. Hay solo una clase de robot (la clase Robot). El constructor de la clase robot recibe tres parámetros, que corresponden a cada una de estas partes. Crear un tipo específico de robot implica decidir cuales de las alternativas se utilizará para cada parte.

Los sistemas de locomoción se implementan en la jerarquía de la clase LocomotionSystem. Se dispone, por ahora, de dos opciones: colchón de aire (Overcraft) y orugas (Caterpillar). Los sistemas de armas y las fuentes de energía se implementan en las jerarquías ArmsSystem y EnergySource respectivamente. Solo combinando esas clases pueden obtenerse los 8 tipos de robots posibles. El siguiente ejemplo crea un robot y lo configura con locomoción por colchón de aire, armamento laser, y fuente de energía solar.

*Robot robot = new Robot(new Overcraft(), new LaserSystem(), new SolarWithBattery());*

 Si apareciese un nuevo tipo de locomoción, tracción 4x4 por ejemplo, alcanza con agregar una subclase FourByFourWheels a LocomotionSystems. La nueva clase implementará el método move(Robot : r) como corresponda. Esto es una ventaja sobre el caso visto en la sección anterior (herencia)

Ahora, cuando un robot recibe el mensaje *step(),*sigue utilizando su método plantilla (que garantiza el cumplimiento de los pasos canónicos como lo marca la especificación del framework). Sin embargo el método plantilla ya no envía mensajes al objeto mismo (como lo hacía en el caso de herencia) sino que delega las operaciones en sus partes. Entonces, la plantilla está en el robot, y los ganchos en las partes.

Habrá notado que cuando el robot delega las operaciones gancho en sus partes, se pasa el mismo como parámetro. Esto es necesario porque las partes ya no tienen acceso a las variables de instancia del robot ni de las otras partes. En consecuencia al implementar los métodos ganchos (en las partes) solo se puede utilizar los métodos que el robot haga públicos. Esto resulta en un escenario mas restrictivo que el que utilizaba herencia.

Una punto a favor de la implementación por composición es que ahora podemos cambiar el comportamiento del robot en tiempo de ejecución. Alcanza con cambiar la parte afectada por aquella que provee el comportamiento deseado.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Ejercicio - ejemplo

## Ficheros adjuntos

El archivo adjunto (un file-out de Pharo Smalltalk) ofrece una implementación muy simple del framework en base a lo visto en esta sección (es decir, utilizando composición para separar la plantilla de los ganchos).

Al hacer el file-in, se debería abrir automáticamente un workspaces con los ejemplos de uso, y browsers en las clases involucradas. El código es muy simple, solo escribe mensajes en el Transcript.

Al mirar el código otra preguntense:

* ¿Que debo hacer si aparece una nueva fuente de energía (por ejemplo, ruedas con tracción 4x4)? ¿Cuántas y cuáles clases debo agregar en caso de querer todas las variantes de robots posibles para este nuevo tipo de sistema de locomoción?
* ¿Puedo cambiarle, a un robot existente, el sistema de armas sin tener que instanciar el robot de nuevo?
* ¿Donde almacenaría usted el nivel de carga de la batería? ¿Que implicaría eso si antes de disparar el laser hay que garantizar que la fuente de energía puede satisfacer el consumo del arma?

**Descargas**:

* [Implementación del framework de robots utilizando composición (Ventana nueva)](https://catedras.linti.unlp.edu.ar/pluginfile.php/85076/mod_imscp/content/3/Hotspots-Composition.st.zip)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Recapitulando

Resumen de lo que vimos:

* Las plantillas y los ganchos se pueden utilizar en un framework para separar lo que se mantiene constante (frozenspot) de lo que varía (hotspots).
* Las plantillas implementan lo que se mantiene constante, los ganchos lo que varía.
* Se puede utilizar herencia o composición para separar la plantilla de los ganchos.
* Si uso herencia, la plantilla se implementa en una clase abstracta y los ganchos en sus subclases (como el clásico patron Método Plantilla). Si uso composición, un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes.
* Con herencia:
  + es mas simple para casos con pocas alternativas y/o pocas combinaciones
  + al implementar los métodos gancho puedo utilizar las variables de instancia y todo el comportamiento heredado de la clase abstracta
  + si hay muchas variantes o combinaciones, empiezo a tener muchas clases y duplicación de código
* Con composición:
  + evita la duplicación de código y el creciente numero de clases cuando existen muchas alternativas y combinaciones posibles
  + al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia del objeto
  + puedo cambiar el comportamiento en tiempo de ejecución sin mayor dificultad

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Auto-evaluación

## Pregunta Verdadero-Falso

Marque con verdadero o falso cada una de las siguientes afirmaciones

Principio del formulario

Las plantillas son formas simples de frameworks orientados objetos

 Verdadero  Falso



Final del formulario

Principio del formulario

Las plantillas y los ganchos pueden utilizarse para separar la funcionalidad constante (frozenspot) de la funcionalidad variable (hotspots) en un framework.

 Verdadero  Falso



Final del formulario

Principio del formulario

Plantillas y ganchos pueden implementarse utilizando herencia, como se muestra en el patron de diseño Método Plantilla

 Verdadero  Falso



Final del formulario

Principio del formulario

Plantillas y ganchos no pueden implementarse utilizando composición (salvo que se utilicen clases abstractas)

 Verdadero  Falso



Final del formulario

Principio del formulario

Las plantillas implementan lo que se mantiene constante, los ganchos lo que varía.

 Verdadero  Falso



Final del formulario

Principio del formulario

Si uso composición, un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes.

 Verdadero  Falso



Final del formulario

Principio del formulario

Si uso herencia, la plantilla se implementa en una clase abstracta y los ganchos en sus subclases .

 Verdadero  Falso



Final del formulario

Principio del formulario

Con herencia se evita la duplicación de código y el creciente numero de clases cuando existen muchas alternativas y combinaciones posibles

 Verdadero  Falso



Final del formulario

Principio del formulario

SI utilizo herencia, al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia del objeto

 Verdadero  Falso



Final del formulario

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)

# Lectura recomendada

Este material se basa principalmente en un artículo científico que se puede consultar para complementar o reforzar lo estudiado (y se puede obtener a continuación).

El artículo "Hot-Spot-Driven Framework Development " [#pree95] presenta a los hotspots como la característica principal en la descripción/especificación de un framework. Los hotspots guían todo el proceso de desarrollo del mismo y actúan como documentación de uso. Habla de templates y hooks pero no se restringe a la interpretación que se hace en el patrón de diseño Método Plantilla sino que también habla de templates/hooks en la composición. Discute que impacto tiene esta diferencia en el comportamiento runtime de la aplicación. Discute también la relación entre patrones y hotspots. Introduce el concepto de Tarjeta de Hotpot para describir un hotspot (ya sea como requerimiento para quien desarrolla el framework o como documentación para el usuario). Define un proceso a en el cual, la definición de los hotspots guía el desarrollo del framework.

## Ficheros adjuntos

[#pree95]: W. Pree, “Hot-spot-driven framework development,” in Summer School on Reusable Architectures in Object-Oriented software Development, 1995, pp. 123–127.  Este articulo puede descargarse a continuación u obtenerse directamente desde CiteSeer siguiendo [este link](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.4910&rep=rep1&type=pdf)

* [Hot-spot-driven framework development (pdf) (Ventana nueva)](https://catedras.linti.unlp.edu.ar/pluginfile.php/85076/mod_imscp/content/3/pree95.pdf)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](http://creativecommons.org/licenses/by-sa/4.0/)