Strategy (Gamma et al.)

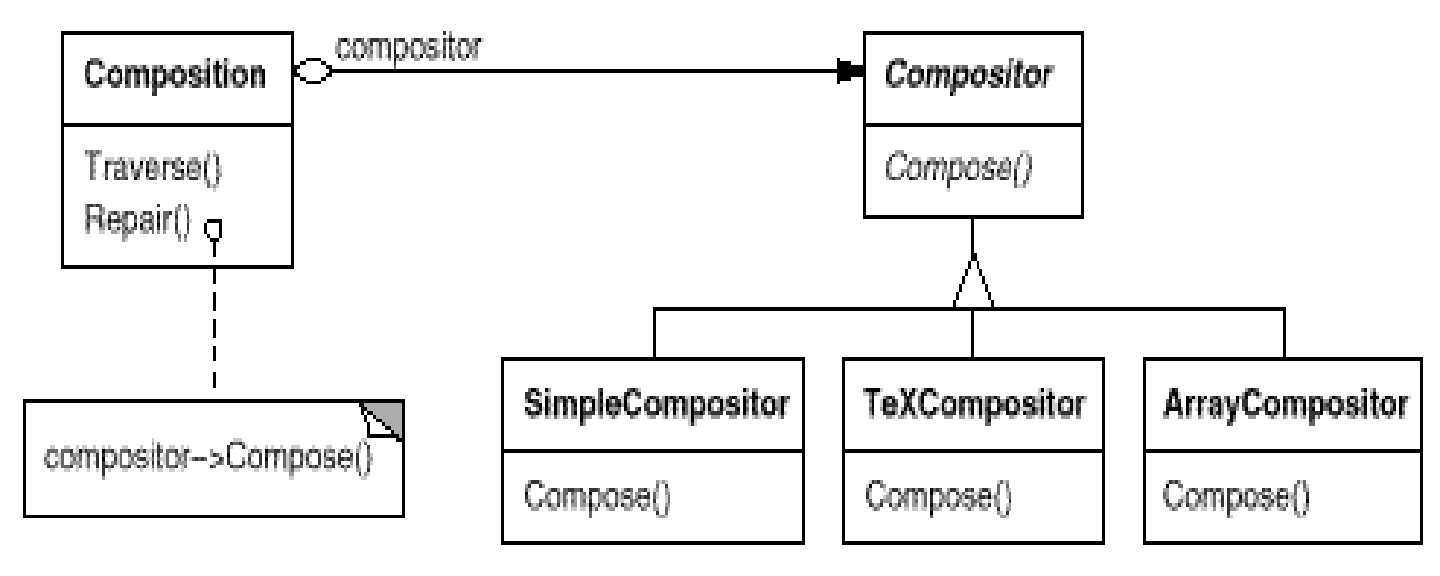
- Define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
- Motivación: Existen varios algoritmos para dividir un párrafo en líneas. Incluir cada uno en el cliente que lo necesita no es conveniente por varias razones.
  - Los clientes que lo necesitan se hacen más complejos: hace crecer sus clases y las hace por lo tanto más difícil de mantener, especialmente si deben mantener varios algoritmos de división en líneas
  - Distintos algoritmos son útiles en momentos distintos. Para qué soportarlos todos si no se usarán todos?
  - Es difícil agregar nuevos algoritmos si son parte del cliente que los usa.
- Esto se evita definiendo clases que encapsulan los distintos algoritmos de división de un párrafo en líneas. Un algortimo encapsulado de esta manera es llamado estrategia.

Strategy

- En este caso se quiere disponer de tres estrategias:
  - SimpleCompositor: va determinando un corte de línea (linebreak) cada vez
  - TexCompositor: implementa el algoritmo que usa Tex (latex) para encontrar los linebreaks. Esta estrategia los trata de optimizar globalmente, es decir, en el párrafo completo cada vez.
  - ArrayCompositor: implementa una estrategia que selecciona breaks de tal manera que cada fila tenga un número fijo de items (palabras)
- Se definen dos clases: Compositor, una clase abstracta cuyas subclases implementan las distintas estrategias, y Composition, la cual referencia un objeto Compositor. Cada vez que reformatea un texto, delega esta responsabilidad en el Compositor object .

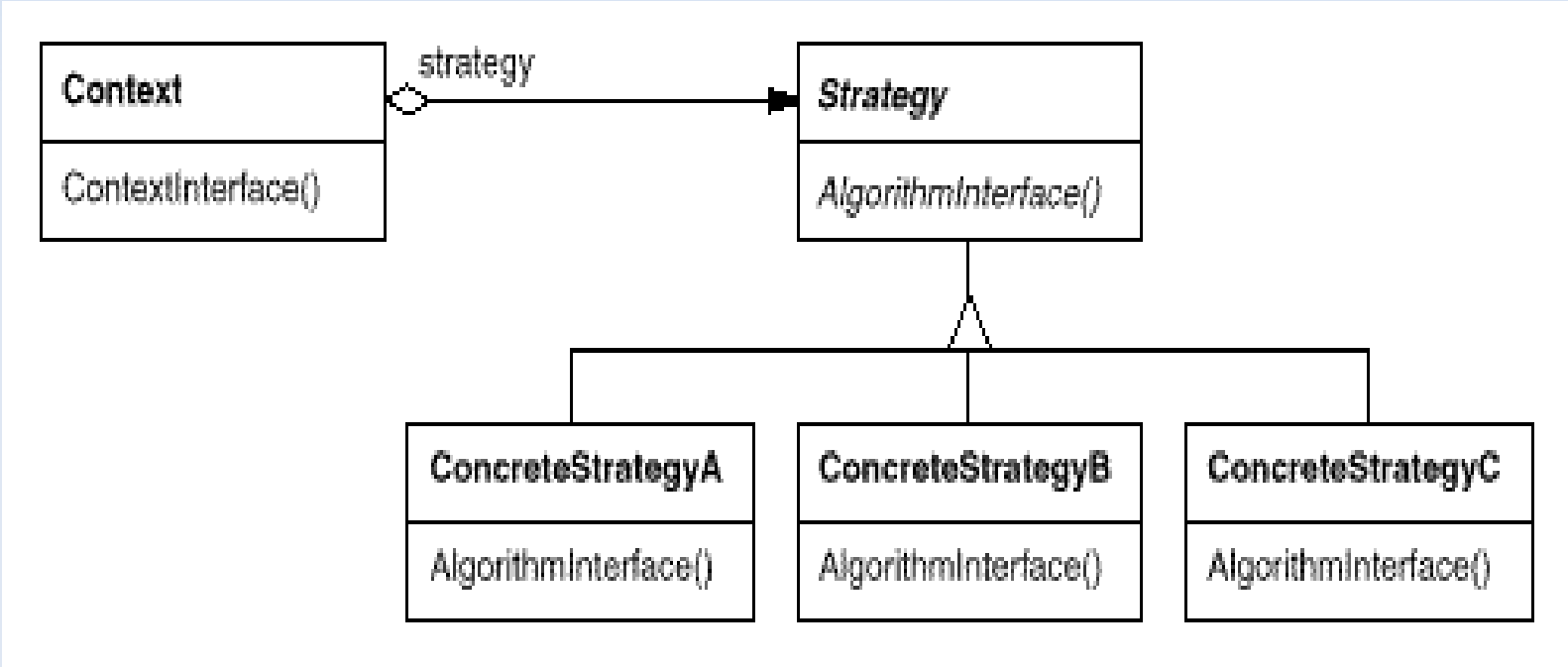
Strategy

Ejemplo:



Strategy

- Estructura:



Strategy

- Consecuencias:
- Strategy elimina instrucciones condicionales. No se necesita usar if-else para seleccionar la estrategia a usar.
- Eligiendo una implementación distinta: Strategy puede modelar distintas implementaciones para el mismo comportamiento. El cliente puede escoger entre estrategias con distinto desempeño tiempo y espacio.
- El cliente debe conocer que estrategias disponibles para asi elegir la más adecuada.
- Overhead de comunicacion entre una estrategia y el contexto. La interfaz de Strategy debe ser compartida por todas las estrategias sean simples o complejas.

Strategy

Implementación:

- Las interfaces del contexto y las estrategias deben permitir un intercambio eficiente de la información que necesitan. En ejemplo presentado, toda la información se pasa a traves de parámetros, haciendo que tanto algoritmos complejos como simples deban recibirla. Otra alternativa sería pasar el contexto como parámetro a la estrategia y que ésta le pregunte la información necesaria. Esto último requeriría de una interfaz más elaborada del contexto.

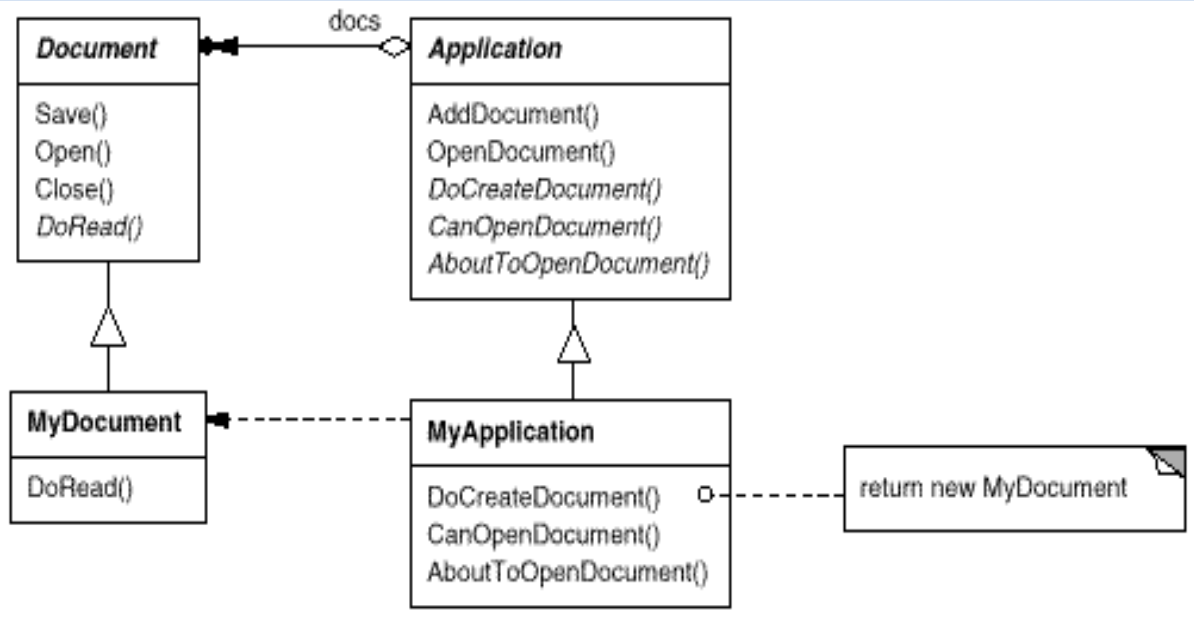
- Las estrategias pueden ser implementadas con templates. Esta técnica es aplicable si (1) la estrategía puede ser seleccionada en tiempo de compilación y (2) no necesita ser cambiada en tiempo de ejecución. Aquí no es necesario definir una clase abstracta para implementar las estrategias.
- Estrategias opcionales. Es posible que el contexto tenga implementada una estrategia por default. En caso que el cliente no le pase una estrategia, usa la que tiene definida internamente.

Template method (Gamma et al.)

- Define el esqueleto de un algoritmo en una operación delegando algunos pasos a las subclases
- **Motivación:** Se tiene una clase abstracta Aplication responsable de abrir documentos existentes en un formato externo, (por ejemplo un archivo), y una clase abstracta Document que representa la información en un documento una vez que es leida de un archivo. Una aplicación puede crear clases concretas de ambas: por ejemplo, una aplicación de dibuja define las subclases DrawAplication y DrawDocument, respectivamente. La clase abstracta Aplication define un algoritmo para abrir y leer un documento en su operación openDocument. Esta define cada paso:
  - verifica que el documento pueda ser abierto
  - crea el documento específico (dependiente de la aplicación)
  - agrega el documento al conjunto de documentos
  - leer el documento desde un archivo

Template method

Ejemplo:



Template method

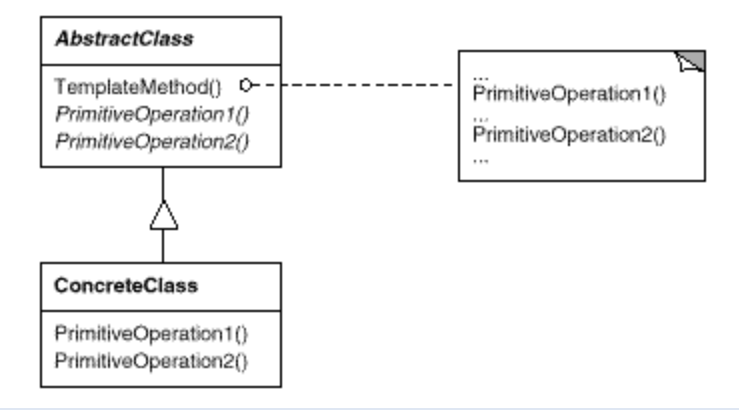
```
void Application::openDocument(const char* name){
    if( !canOpenDocument(name) ){
        return;
    }
    Document* doc = doCreateDocument();  //redefinida en subclases de Application
    docs->addDocument(doc);
    aboutToOpenDocument(doc); //redefinida en subclases de Application
    doc->doRead();
}
```

Este tipo de métodos como openDocument son los que se definen como **Template Method**

Template Method

- Aplicabilidad:
  - implementar las partes invariantes de un algoritmo una vez y dejar que las subclases implementen el comportamiento que puede variar
  - cuando un comportamiento común entre las subclases puede ser factorizado y localizado en una clase común para evitar duplicación

Estructura:



Template method

- Consecuencias: Los métodos templates son:
  - fundamentales para reusar código
  - muestran una estructura de control invertida: la clase padre llama a operaciones de las clases hijos y no al revés.
  - generalmente llaman a uno de los siguientes tipos de operaciones:
    - operaciones concretas (de una clase concreta o de un cliente)
    - operaciones concretas de una clase abstracta
    - operaciones abstractas
    - operaciones “hooks”. Esta proveen una implementación por default que las subclases pueden redefinir si lo lo desean. Normalmente esta operación no hace nada.

Template method

Implementación:

- Las operaciones abstractas que un método template usa pueden ser declaradas protected. Esto asegura que sólo un método template puede llamarlas. El método template NO debe ser redefinido.
- En la medida de lo posible, minimizar el número de métodos que deben ser redefinidos en las subclases.
- Puede ser útil usar una convención para nombrar los métodos que deben ser redefinidos. Por ejemplo con do: doCreateDocument, doRead, etc...

Patrones relacionados:

- Los factory methods son frecuentemente llamados por los métodos templates. En el ejemplo de motivación, se llama a doCreateDocument.
- Los métodos templates usan herencia para variar parte de un algoritmo. Estrategia usa delegación para variar un algoritmo completo.

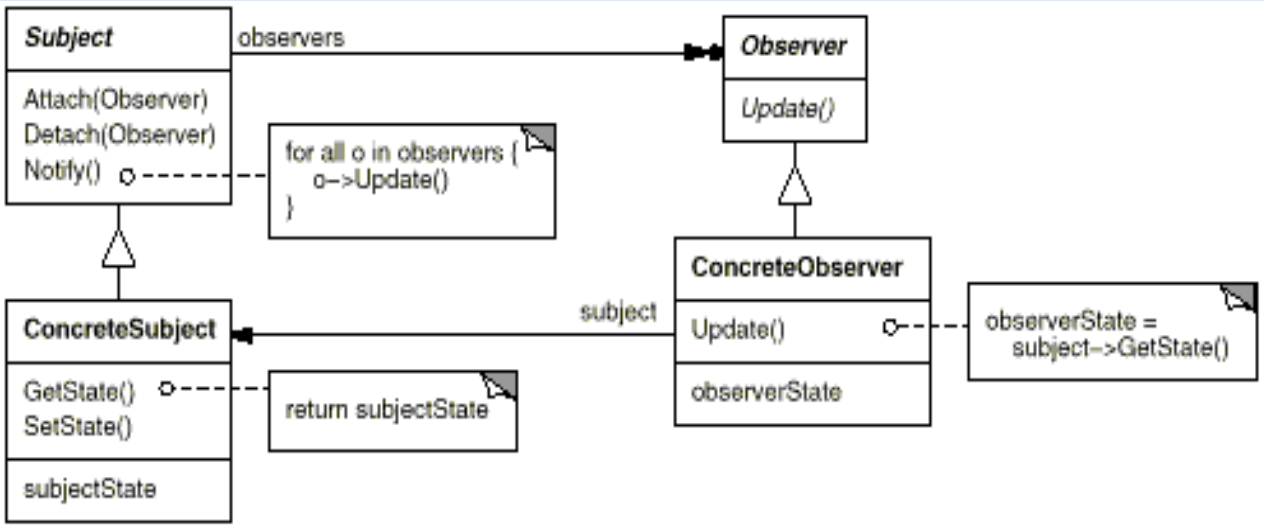
Observer (Gamma et al)

- Define una dependencia entre uno y varios objetos de tal manera que cuando un objeto cambia de estado, todos los objetos dependientes de él son notificados y actualizados automáticamente.
- **Motivación:** Muchas toolkits separan los aspectos de presentación de los datos (modelo) de la aplicación, pues ambos deberían poder ser usados de manera independiente. Por ejemplo, una planilla de cálculo y un gráfico de barras no saben de la existencia del otro pero ellos se comportan como si se conocieran. Cuando un usuario cambia los valores en la planilla de cálculo, ambos reflejan los cambios, y viceversa. Este comportamiento implica que ambos dependen de la información y que ambos deben ser notificados cuando ésta cambia. El número de objetos dependientes puede ser grande.
- El patrón observer describe este comportamiento.

Observer

Los objetos claves de este patrón son: **subject** y **observer**. El subject puede tener un número muy grande de observadores. Todos los observadores deben ser notificados cuando un subject cambia de estado.

Estructura:



Observer

- **Aplicabilidad:**
  - cuando una abstracción tiene dos aspectos y una depende de la otra. Encapsulando estos aspectos en objetos separados permite reusarlos de manera independiente
  - Cuando un cambio en un objeto requiere cambios en otros y no se sabe cuantos objetos necesitan ser cambiados
  - Cuando un objeto debe notificar a otros sin conocer la identidad de esos objetos. Se desea evitar un alto grado de acoplamiento
- **Implementación:**
  - Observar más de un subject. Puede tener sentido que en algunas situaciones un observador dependa de más de un subject y por lo tanto requiera saber quien le notifica el cambio. Para esto basta con pasar como parámetro el método update el subject que genera el update.

Observer

- Quién genera le update? Quién llama a notify?
  - Cada cambio a las variables de instancia del subject podría generar una llamada a notify
    - ventaja: el cliente no tiene que acordarse de llamar a notify
    - desventaja: operaciones consecutivas pueden generar updates consecutivos lo cual puede ser muy ineficiente
  - Dejar que el cliente llame a notify cuando corresponda.
    - ventaja es que puede esperar una serie de cambios de estados del subject antes de generar un notify
    - desventaja: olvidarse
- Borrar un subject no debe generar referencias inexistentes en sus observadores. Una forma de hacer esto es que el subject notifique a todos los observadores que será eliminado, de tal manera que ellos puedan re-inicializar su referencia a él.

Observer

- Asegurar que el estado del subject consistente antes de enviar un notify. Por lo tanto, no se debe enviar un notify entre instrucciones que cambian el estado del subject, sino al final.
- Evitar protocolos de update que dependan del observador. En un extremo están los modelos push, en donde los subjects envian a los observadores información detallada sobre el cambio, lo quieran o no. En el otro extremo está el modelo pull, en donde el subject envía solo la notificación mínima y cada observer pregunta por los cambios que le interesan.
  - El modelo push asume que los subjects saben todas las necesidades de sus observers
  - El modelo pull enfatiza la ignoracia de los subjects sobre las necesidades de sus observadores. Este puede ser ineficiente pues el observador debe averiguar que cambió en el subject. (Este modelo es el que se muestra en el patrón)

Observer

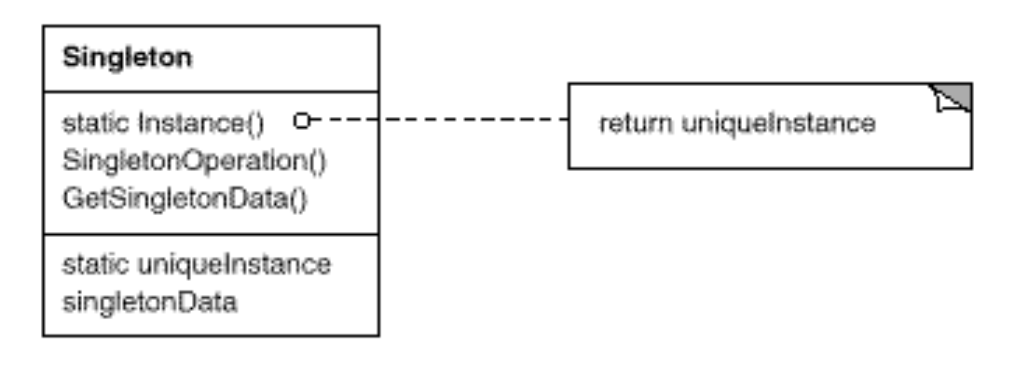
- Encapsulando semánticas de actualización complejas. Cuando la relación de dependencia entre subjects y observers es compleja puede ser necesario tener un objeto que maneje estas relaciones. Este objeto es llamado **ChangeManager**, cuyo propósito es minimizar el trabajo requerido para que los observers reflejen los cambios en sus subjects. Por ejemplo, si una operación involucra cambios en varios subjects interdependientes, puede ser necesario asegurar que sus observers sólo sean notificados una vez que todos los subjects han sido modificados para asi evitar que los observers sean notificados más de una vez. **ChangeManager** tiene tres responsabilidades:
  - mapea un subject a sus observadores y provee una interfaz para mantener este mapeo. Esto elimina la necesidad que los observadores mantengan una referencia al subject y viceversa
  - define una estrategia particular de update
  - actualiza/notifica a todos los observers dependientes ante el pedido de un subject

Observer

- ChangeManager es una instancia del patrón Mediator. En general hay un solo ChangeManager y es conocido globalmente, luego el patrón Singleton es tambień útil aquí.

Singleton

- Asegura que una clase tenga sólo una instancia y provee un punto global de acceso a ella
- Motivación: en algunas aplicaciones, una clase debe solo instanciarse una vez. Por ejemplo, pueden existir muchas impresoras pero una sola cola de impresión. Cómo se puede asegurar que una clase tenga una sola instancia y que ésta sea fácilmente accesible?
  - hacer que la misma clase almacene su única instancia.



Singleton

- Consecuencias:
  - acceso controlado a la única instancia
  - permite refinamiento de sus operaciones y representación. Un singleton puede tener subclases y es fácil configurar una aplicación con una instancia de sus subclases
  - permite un número variable de instancias. Este patrón puede ser modificado para permitir más de una instancia de la clase singleton, y por lo tanto puede ser usado para controlar el número de instancias que la aplicación desea usar
- Implementación:
  - Asegurando una única instancia. La forma más común de hacerlo es ocultar la operación que crea la instancia en una operación de la clase (función estática en java o c++). La variable de instancia que contiene la única instancia es también estática (ver libro)

Singleton

- Heredando de la clase Singleton. La variable de instancia que contiene la única instancia puede ser inicializada con una instancia de sus subclases. Una forma flexible de hacer esto es implementar un mecanismo de registro de Singletons. El registro mapea nombres y singletons. Existe una funcción lookUp para obtener el singleton asociado dado un nombre. El nombre del singleton a usar se puede obtener de una variable del ambiente. Donde se registran las clases singleton?
  - en su constructor y definir una instancia estática de esta subclase (c++) en el archivo que se define esta subclase. Esto hacer que quede accesible en el sistema.
- Patrones relacionados: Muchos patrones pueden ser implementados usando Singletons, por ejemplo Abstract Factory, Builder, y Prototype

Singleton (Código Java)

Una aplicación necesita generar números aleatorios y usarlos distintas partes, pero desea que una sola clase se los genere.

```
import java.util.Random;
public class SingleRandom{
    private SingleRandom(){generator = new Random();}
    public  void setSeed(int seed){ generator.setSeed(seed);}
    public int  nextInt(){return generator.nextInt();}
    public static SingleRandom getInstance(){ return instance;}
    private Random generator;
    private static SingleRandom instance = new SingleRandom();
}
```

Uso: `int randomNumber = SingleRandom.getInstance().nextInt();`