**Concurrencia**

Concurrencia es tratar de **lidiar** con muchas cosas a la vez. Es la composición de la ejecución de “cosas” independientes. Es sobre la estructura de los problemas, descomponer tareas en tareas más pequeñas.

Paralelismo es **hacer** muchas cosas a la vez. Es ejecutar tareas en simultáneo.

**Concurrencia no implica paralelismo.**

**Thread:** secuencia independiente de instrucciones ejecutándose dentro de un programa.

**Problemas:**

**Race condition:** se da cuando varios threads pueden acceder a recursos compartidos (código). El resultado del programa depende de cómo se intercalen los threads.

**Critical section:** sección de código que necesita ser ejecutada en forma **atómica** por un solo hilo a la vez.

**Locks:** se basa en el uso de una variable de **exclusión mutua (mutex).**

**Monitores:** objetos thread-safe, sus métodos están sincronizados **(mutex).**

**Conditional variable:** mecanismo de bloqueo con una señalización.

**Deadlock:** aparece cuando entre dos o más threads uno obtiene un recurso y no lo libera generando un bloqueo.

**¿Cómo se implementa?**

1. **Heredar de Thread (Template Method)**
2. **Implementar Runnable**

**Consideraciones:**

* **Debuggear es complicado**
* **Agregar println() puede alterar el resultado**
* **Demasiada sincronización perdemos la ventaja de usar threads**
* **Poca sincronización genera errores difíciles de detectar**

**Clase Mocking**

**Utilidad:**

* **Integración de componentes**
* **Mitigar inestabilidad de casos de pruebas**
* **Aislar componentes**
* **Paralelizacion de desarrollos interdependientes**
* **Simular Aleatoriedad**

Favorecer **stubs** antes que mocks. Reemplazar stubs con implementaciones concretas.

Favorecer **FakeObjects** sobre stubs si existe código específico para la prueba

Solo viven en ambientes de prueba

Si buscamos probar comportamiento (envío de mensajes). Utilizar **MockObject**

**MVC (Modelo-Vista-Controlador)**

**Mecanismo de despacho**

**¿Qué? Disponibilizar software**

**¿Cómo?**

* **Procastinar decisiones no escenciales**
* **Cohesión**
* **Escritura de pruebas**

**Modelo-Vista-Controlador:** patrón de arquitectura diseñado para la construcción de interfaces de usuario. La principal motivación es la separación de responsabilidades.

Tres tipos de objetos:

* **Modelo:** lógica y entidades de negocio o dominio de nuestra aplicación
* **Vista:** formas en que los objetos del modelo se muestran y representan al usuario
* **Controlador:** definen cómo la interfaz de usuario reacciona a las acciones del usuario.

**Patrón Observer**

**Intención:** definir una dependencia uno-a-muchos entre objetos de manera que cuando un objeto cambie su estado, todas sus dependencias sean notificadas y actualizadas automáticamente.

**Motivación:** mantener la consistencia entre objetos que dependen entre sí, reduciendo el acoplamiento y preservando la reusabilidad.

En criollo: permite a un objeto ser “observado” (sujeto) por un conjunto de otros objetos (observador). Cuando el sujeto sufre un cambio en su estado, “notifica” a los observadores a través de un método especial para ello.

**Patrón Strategy**

Visible en la relación Vista-Controlador. Se puede pensar el controlador como una instancia del Strategy a usar por la Vista frente al input o acción de un usuario.

**Principios de Diseño**

Conjunto de reglas que favorecen y/o fomentan la escritura de código extensible y mantenible, minimizando los posibles códigos afectados. Uno de estos es Tell-don´t-Ask y otro son los S.O.L.I.D.

**Principios SOLID**

* **SRP: principio de responsabilidad única**

Una clase debe tener una única razón para cambiar. Una refactorización o un arreglo de un bug no es una razón de cambio. Las razones de cambio son cambios de los requisitos.

* **OCP:** principio de abierto/cerrado

Las clases deben estar abiertas para la extensión, pero cerradas para su modificación. Se debe poder cambiar el comportamiento sin modificar el código ya existente. Se puede lograr utilizando herencia o delegación.

* **LSP:** principio de sustitución de Liskov

Las clases heredadas deben poder ser utilizadas a través de su clase madre sin la necesidad de que el usuario sepa la diferencia.

* **ISP:** principio de segregación de la interfaz

Los clientes no deben ser forzados a depender de métodos que no utilizan. Muchas interfaces específicas son mejores que una interfaz de propósito general. Es necesario aplicarlo cuando se tiene una clase con varios métodos, de los cuales solamente me interesan algunos.

* **DIP:** principio de inversión de dependencias

Se debe depender de las abstracciones y no de las implementaciones. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.