Conceptos que vi en finales.
HERENCIA: Programación por diferencia. Es una forma de vinculación de componentes.
DELEGACIÓN: También es un tipo de programación por diferencia.

Interfaces: Conjunto de mensajes que una clase ofrece a sus clientes. Todos sus métodos son públicos y abstractos. En JAVA podemos hacer que una clase implemente varias interfaces. En Smalltalk no se utilizan interfaces porque es de comprobación dinámica (variables no tienen tipo, los objetos sí), hace comprobaciones a la hora de la ejecución, en cambio JAVA es de tipado estático.

Persistencia: Característica que puede tener un dato o estado que le permite sobrevivir al proceso que lo creó. Un POO debe permitir que los objetos persistan para mantener "su vida más allá del programa". Gracias a la persistencia, un objeto puede conservar su estado en un medio de almacenamiento permanente.

- Persistencia nativa: provista por la plataforma. Resuelve referencias circulares, no es extensible ni reparable (info en binario) ni portable a otros lenguajes de manera sencilla, no es óptima en cuanto tamaño..
- Persistencia no nativa: programada a mano o provista por una biblioteca externa. Es más compleja pero también más versátil.

Java implementa de forma nativa la persistencia.

Serialización: consiste en convertir la representación de un objeto en un stream (flujo o secuencia) de bytes. Reconstruir un objeto a partir de un stream de bytes se denomina deserialización. Luego de la serialización: lo envío por red, mantengo en memoria, envío a una impresora, etc. EN JAVA se debe implementar la clase Serializable que no tiene métodos, sirve para avisarle a la máquina virtual que la clase puede serializarse. En Smalltalk también se implementa serialización nativa.

Persistencia y Serialización: Para persistir debo primero serializar, serializar no implica persistir.

Invariantes: Los invariantes son condiciones que debe cumplir un objeto durante toda su existencia. Suelen expresarse en forma de post y pre condiciones.

Polimorfismo: Capacidad que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje.

RTTI: Real Time Type Information. Es preguntarle a la instancia de qué clase es (instance of) Esta práctica compromete la extensibilidad de una manera que el polimorfismo lo resuelve automáticamente. Viola el principio de Open/Close. Esto se obtiene en tiempo de ejecución.

Metaclase: Es una clase cuyas instancias también son clases. En Smalltalk son anónimas (ej: CuentaCorriente class)

Multiple choice: Behavior no es una metaclase. Esta es la clase padre de Class cuyo padre es Object en Smalltalk.

Reflexión y xUnit: xUnit crea una instancia de la clase para testear y llamar sucesivamente a los métodos que encuentra en ella que comienzan con la palabra test o en Java se usan las notaciones @Test. Para esto utiliza la técnica de Reflexión.

Reflexión: Es la posibilidad que tiene un programa de examinar y modificar su estructura y comportamiento en tiempo de ejecución.

Diseño: Actividad de desarrollo de software que se centra en la manera en la que resolvemos problemas.

- Alta cohesión: clases con pocas responsabilidades. Idealmente una sola (Single Responsability).
- Bajo Acoplamiento: la clase debe depender de pocas clases.

MVC: patrón de separación de incumbencias. Patrón macro que consiste en separar la aplicación en tres partes: Modelo (con bajo acoplamiento a los otros dos), vista (Tarea: observar el modelo para actualizar las variaciones) y controlador (manejo global de la aplicación).

Un patrón es una solución no trivial a un problema en un determinado contexto. Para que sea útil, debe ser una solución probada a un problema que aparece con frecuencia. Se aplica a una sociedad de objetos y/o clases. De alguna manera, un patrón de diseño se resuelve mediante una colaboración, y consta tanto de aspectos estructurales como de comportamiento.

- Iterator. Iteradores fuera de la colección. De esta forma, no interesa saber la estructura interna de la colección (y esta última no se ocupa de desarrollar el desplazamiento entre cada una) Tiene alta cohesión, única responsabilidad e inversión de dependencia porque utilizamos el iterador mediante una interfaz.

- Template Method: No se puede redefinir el método plantilla (representa las invariantes del algoritmo) porque tergiversa la intención del patrón (la macro sería mutable), los otros métodos pueden ser abstractos o tener una implementación por defecto.

- Factory Method: Encapsula la creación de nuevos objetos de la misma clase abstracta o interfaz. También se puede implementar con Reflexión.

- Observer:

- Factory Method: También usa Reflexión.

Double dispatch no es un patrón sino una técnica.

xUnit es un framework. Dicho de otra manera, una biblioteca está lista para ser usada mediante la invocación de sus servicios, mientras que un framework es un programa que invoca a los comportamientos definidos por el programador.

USING JAVA REFLECTION

Permite que un programa Java en ejecución se examine o introspectiva sobre sí mismo y manipule las propiedades internas del programa.
¿Qué se puede hacer?

- Enumera los nombres de método de la clase con sus parámetros completos y tipos de retorno.
- Se puede simular un instanceof -> Class.isInstance()
- Obtener información sobre constructores. Se utiliza igual para obtener un número de clase pero sin valor de retorno.
- Los campos de datos (atributos) definidos en una clase y en una superclase. Se puede saber el tipo y el modificador (private, public static final).
- Invocar un método con un nombre específico en tiempo de ejecución.
- Crear nuevos objetos.
- Cambio de valores de campos. El valor de esto se deriva nuevamente de la naturaleza dinámica de la reflexión, donde un campo se puede buscar por su nombre en un programa en ejecución y luego cambiar su valor.
- Creación y manipulación de matrices de tamaño dinámico que no es necesario que se conozca en el momento de la compilación.

La reflexión de Java es útil porque admite la recuperación dinámica de información sobre clases y estructuras de datos por nombre, y permite su manipulación dentro de un programa Java en ejecución.

Corolario: Los frameworks xUnit hacen uso de reflexión.

INTEGRACIÓN CONTINUA - Fowler.

Integración del trabajo con frecuencia generalmente cada persona se integra al menos a diario, lo que lleva a múltiples integraciones por día. Cada integración se verifica mediante una compilación automatizada (incluida la prueba) para detectar errores de integración lo más rápido posible. Muchos equipos encuentran que este enfoque conduce a problemas de integración significativamente reducidos y permite que un equipo desarrolle software cohesivo más rápidamente.

Prácticas de integración continua:

- **Mantener un repositorio de fuente única.** (ej github) Debe ser lugar conocido para que todos obtengan el código fuente, **todo** debería estar en el repositorio (script de prueba, archivos de propiedades, script de instalación, biblioteca de terceros) regla de oro: si tomo una computadora sin nada (salvo SO y entorno de desarrollo JAVA) debo poder correr el programa sin problemas. También configuraciones IDE. Hay que mantener las ramas al mínimo y mantener todo en la línea principal la mayor parte del tiempo.
- **Automatizar la construcción.** (ej Ant) De debe incluir todo, sacar el esquema de la base de datos del repositorio y ponerlo en marcha en el entorno de ejecución. Una

buena herramienta de construcción analiza lo que debe cambiarse como parte del proceso.

- **Realice la autocomprobación de su construcción.** Para capturar errores puede usarse TDD que son pruebas automatizadas que pueden verificar una gran parte de la base del código. Las pruebas deben poder iniciarse con un simple comando y ser auto verificadas, al finalizar debe indicarse si la prueba falló. Para que una compilación sea autoevaluada, la falla de una prueba debería hacer que la compilación falle. TDD -> xUnit . Las pruebas no prueban la ausencia de errores. Las pruebas imperfectas que se ejecutan con frecuencia son mucho mejores que las pruebas perfectas que nunca se escriben.
- **Todos se comprometen con la línea principal todos los días.** La integración permite a los desarrolladores informar a otros desarrolladores sobre los cambios que han realizado. El único requisito previo para que un desarrollador se comprometa con la línea principal es que pueda compilar correctamente su código (pasar las pruebas de compilación). Ciclo de confirmación: primero actualizar nuestra copia de trabajo para que coincida con la línea principal, resolver cualquier conflicto que surja y luego construir en la máquina local. Si la compilación pasa son libres de comprometerse con la línea principal. De esta forma se descubre rápidamente si hay un conflicto entre dos desarrolladores.

  Si los desarrolladores se comprometen con la línea general cada hora, de forma recurrente, los errores son más fáciles de localizar y de arreglar al tener los conceptos claros y frescos.

- **Cada compromiso debe construir la línea principal en una máquina de integración.** Con las confirmaciones diarias se obtienen compilaciones probadas con frecuencia, pero en la práctica no siempre resulta porque las personas no actualizan y compilan antes de comprometerse o diferencias entre las máquinas de los desarrolladores. Se puede utilizar un servidor de integración continua (ej: Travis) o una compilación manual (similar a la compilación local que hace un desarrollador antes de la confirmación en el repositorio).

  A pesar de que hay empresas que realizan compilaciones regulares en un horario programador, como las noches, no es lo mismo que una compilación continua y no es suficiente para una integración continua. El objetivo de esta última es encontrar problemas lo antes posible.

- **Reparar las construcciones rotas de inmediato.** Si la compilación de la línea principal falla, debe repararse de inmediato. Este arreglo es de carácter urgente para todos los desarrolladores del proyecto. Una forma es revertir la última configuración en la línea principal, el equipo no debería hacer ninguna depuración en la línea principal rota a menos que la causa de la rotura sea obvia de inmediato.

- **Mantener la construcción rápida.** El objetivo de integración continua es proporcionar retroalimentación rápida. Una pauta de XP es una compilación de diez minutos.
  Compilaciones realizadas en secuencia:
  1) Compromiso con la línea principal: compilación de compromiso.

2) Compilación de confirmación. Se debe realizar rápidamente

   Paréntesis Vero: Lo relaciono con bajarlo del git, compilarlo, pasar las pruebas unitarias, de integración y después escribir mi código.

   Ejemplo si tengo pruebas de dos etapas y a la segunda detecta un error, la primera que consta de pruebas de confirmación debe modificarse para que pueda atrapar ese error.

- **Probar en un clon del entorno de producción.** (ej: máquinas virtuales)
- **Facilitar la obtención del último ejecutable.** Debe ser estable.
- **Todos pueden ver lo que está pasando.** Se debe poder comunicar el estado de la construcción de la línea principal.
- **Automatizar la implementación.** Para realizar integración continua, se necesitan varios entornos, uno para ejecutar pruebas de confirmación, uno o más para ejecutar pruebas secundarias. Dado que está moviendo ejecutables entre estos entornos varias veces al día, querrá hacerlo automáticamente. Por lo tanto, es importante tener scripts que le permitan implementar la aplicación en cualquier entorno fácilmente.

Beneficios de la integración contínua.

1) Reducción del riesgo, no pasar años integrando el proyecto al finalizar de desarrollar. Se sabe en qué punto está, qué funciona y qué no, los errores pendientes del sistema.
2) A pesar que las integraciones continuas no eliminan errores, las hace mucho más fáciles de encontrar y de eliminar. Es como un código de autoprueba. Si se introduce un error lo detecta rápidamente y es más fácil deshacerse de él, estos no se acumulan.
3) Los proyectos con integración continua tienden a tener menos errores, tanto en producción como en proceso. Está relacionado directamente con lo bueno que sea su conjunto de pruebas.

REEMPLAZAR LOS CONDICIONALES CON POLIMORFISMO.

¿Por qué refactorizar? Esta técnica puede ayudar si el código contiene operadores que realizan varias confirmaciones basadas en la clase del objeto o la interfaz que implementa, el valor de un atributo de un objeto, el resultado de llamar a un método de otro objeto.

Beneficios:

- Adhiere al principio tell-don’t-ask.
- Elimina el código duplicado. Deshacerse de condicionales casi identicos.
- Si se necesita una nueva variante de ejecución, todo lo que se necesita hacer es agregar una nueva subclase sin tocar el código existente (Open/Close)

Cómo refactorizar.

Se debe tener lista una jerarquía de clases que tendrá comportamientos alternativos.

ESTADO DEL ARTE Y TENDENCIAS EN TEST-DRIVEN DEVELOPMENT. - Fontela

UTDD: Unit Test-Driven Development. Test first pero con pruebas unitarias. Técnica de diseño detallado.
ATDD: Acceptance Test-Driven Development. TDD ampliado. Práctica de diseño de software que obtiene el producto a partir de pruebas unitarias. Se toma cada requerimiento en forma de user story y se construyen pruebas de aceptación del usuario que luego se automatizan. Técnica de diseño basada en requerimientos.
BDD: TDD mejorado. Behaviour Driven Development. Se prueban las especificaciones o el comportamiento, así se valida más fácilmente con clientes y especialistas del negocio. No se prueba solamente pequeñas porciones de código, sino la interacción de objetos en escenarios. Métodos que se puedan leer como oraciones y, cuando estas pruebas fallan, se lee textualmente cuál es el error.Técnica de diseño basado en dominio

ATDD vs BDD: No está claro el límite entre ambas.Ambas pusieron énfasis en que no eran pruebas de pequeñas porciones de código, sino de especificaciones de requerimientos ejecutables.

- UTDD: Facilita el buen diseño de clases.
- ATDD y BDD: facilitan construir el sistema correcto.

STDD: Story-Test Driven Development. La idea es usar ejemplos como parte de las especificaciones y que los mismos sirvan para probar la aplicación.Técnica de captura de requerimientos. No todo requerimiento puede llevarse a ejemplos en el sentido de STDD (ej: si una app debe generar números al azar, los ejemplos que escribamos nunca van a servir como pruebas de aceptación).

NDD: Need-Driven Development. El comportamiento de los objetos debería estar definido en términos de cómo envía mensajes a otros objetos, además de los resultados devueltos en respuestas a mensajes recibidos. Requiere la utilización de Mocks.

UTDD típicamente basad en diseño de clases, en el chequeo del estado de los objetos luego de un mensaje y se basa en un esquema *bottom-up*. BDD y STDD se basan en diseño general, enfocándose en el chequeo de comportamiento y se basa en el desarrollo *top-down.* NDD se encuentra en la mitad de ambos enfoques.

UTDD y NDD apuntan a la calidad interna. BDD y STDD apuntan a la calidad externa.

Pruebas de aceptación: STDD, ATDD y BDD. ¿Funciona el sistema de acuerdo a los requerimientos?
Pruebas de integración: NDD ¿cada parte del sistema interactúa correctamente con las demás partes? Tiene que ver con el bajo acoplamiento y las interfaces entre módulos.
Pruebas Unitarias: UTDD ¿Las clases hacen lo que deben hacer y es conveniente trabajar con ellas? Tiene que ver con la cohesión y el encapsulamiento.