Resumen papers Leveroni

Advertencia: Ningún paper no-práctico dice nada útil ni tiene mucho sentido leerlos, son todas falopeadas filosóficas de exactas. Recomiendo fuertemente no leer los primeros 3.

In god we trust

Modern software engineering

Por más que no lo parezca, no hay mucho más que esto, el paper se repite un montón de veces, diciendo efectivamente nada.

Introducción

- La ingeniería en software es un proceso de descubrimiento y exploración, por lo tanto los ing. en software tienen que convertirse expertos en aprender
- Propone aplicar el método científico a la ing. en software para organizarnos y no saltar a conclusiones inapropiadas
- **Def:** Software engineering is the application of an empirical, scientific approach to finding efficient, economic solutions to practical problems in software.
- Es decir que tenemos que manejar la complejidad de nuestros sistemas de forma tal que tengamos la habilidad de aprender nuevas cosas y adaptarnos a ellas.
- Tenemos que volvernos expertos en **aprender y manejar complejidad creciente**
- Para convertirnos en expertos en **aprender** hay 5 claves a seguir:
    - Iteraciones
    - Feedback
    - Incrementalismo
    - Experimentación
    - Empiriscismo
- Los sistemas complejos surgen de pequeños y controlados pasos, no de una de nuestra imaginación, estas cosas nos dejan explorar y descubrir
- Para convertirnos en expertos en **manejar complejidad:**
    - Modularidad
    - Cohesión
    - Separación de responsabilidades
    - Abstracción
    - Tener el mínimo acoplamiento posible

Reclaiming “Software Engineering”

- Acá que ande no signifique que sea bueno, necesitamos prácticas que nos permitan hacer software más rápido, sino no son engineering y tenemos que volarlas

How to make progress

- Hacer software es una práctica compleja, no podemos andar reinventando como hacerlo por cada proyecto, necesitamos una serie de principios comunes y cierta disciplina que nos guíe. Vamos a aplicar principios científicos a nuestra práctica para ello.

Design engineering, not production engineering

- No se puede iterar rápido cuando el producto es físico, porque cambiar algo como un puente no es trivial, pero con lo digital si.
- Cuando testeamos nuestro producto testeamos la cosa real, no un modelo esotérico de la realidad como puede ser cuando simulan/testean lanzamientos de cohetes por ejemplo.

Any industry change?

- Básicamente, no ha habido muchos cambios en cómo **pensamos** el software, se ha mantenido.

The importance of measurement

- Menciona que los equipos de trabajo chicos performean mejor.
- Hay dos medidas usadas, **stability and throughput**, se creó un modelo que en base a ellas predice que tan bueno será el equipo de trabajo.
- Stability:
    - Change failure rate: Que tan frecuentemente un cambio rompe cosas
    - Recovery failure time: Cuánto tiempo toma arreglar un error
- Throughput:
    - Lead time: Tiempo que una linea de código pasa de idea a funcionar
    - Frecuencia: Que tan frecuentemente los cambios son pusheados a producción

Programming as theory building

Introduction

- Programar es tener un insight, teoría, acerca del tema que se está trabajando. Es decir, es llegar a conclusiones acerca del dominio del problema.

Programing and the programmer's knowledge

- El aspecto principal de la programación es que los programadores construyan y adquieran conocimiento acerca de un determinado tema.
- Ni el código ni la documentación son suficientes para entender un programa. Pone un ejemplo de extender el uso de un compilador hecho por A, luego B lo toma, lee todo y no usa ningún tipo de facilidad que ya venía hecha, consulta con A y fixean todo. Pero luego alguien toma lo que hace B y sigue de ahí, aca se ve el problema.

Ryle's notion of theory

- Si yo hice algo es porque considero que está bien, a eso llegue basado en mi aprendizaje a lo largo de mi vida. Por lo tanto la teoría de uno es irremplazable y no puede ser explicada simplemente con una serie de reglas.

The theory built by the programmer

- La teoría que construye el programador es de cómo ciertos aspectos de la realidad van a ser handeleados por una computadora.
- Teoría >>> Documentación/Código
    - Porque puede explicar el por que de cada parte del programa
    - Explica cómo match con la realidad cada parte del programa
    - Sabe como modificarlo para extenderlo siendo coherente con lo previamente hecho

Problems and costs of program modifications

- No es bajo porque modificar un programa es mas que editar texto
- Es costoso porque por lo general las modificaciones no las hace quien tiene la teoría

Program Life, Death and Revival

- Cuando se disuelve el equipo que lo hizo muere la teoría
- Revivirlo es re-construir su teoría con un grupo nuevo de programadores. Es imposible reconstruirla realmente, se puede hacer algo similar.

Method and theory building

- No es coherente tener métodos para programar en el contexto del theory building point of view, cada teoría se construye de manera diferente.
- Le pega al paper anterior respecto a usar el método científico ya que no hay realmente un método que ayude a los científicos, esto está equivocado.

No silver bullet

Does it have to be hard? - Essential Difficulties

- Se dividen en dos las dificultades del software, accidentales: limitaciones del lenguaje, hardware, etc, no inherentes del software en sí y esenciales: inherentes del software en si.
- La parte más difícil es el diseño y testeo de la solución ideada
- Propiedades inherentes del software moderno:
    - Complexity: No escala de manera lineal, escalar un sistema no implica hacer más de lo mismo. Se incluyen todos los problemas de management aca tambien.
    - Conformity: Tener que seguir interfaces preestablecidas. Esto tiene mas que ver con tener que seguir ideas ya preestablecidas de como hacer software (me lo dijo en el final) mas que tener que seguir cosas tipo interfaces de frameworks o bases de datos (lo comento en clase)
    - Changeability: Constantemente cambia. Es extensible, se le agregan nuevas cosas. Sobrevive pasado el tiempo de vida del hardware.
    - Invisibility: Es un quilombo, la cantidad de dependencias que hay es muy grande, hace que se complique comunicar acerca del mismo.

Past breakthroughs solved accidental difficulties

- Formas en las que se fue eliminando la dificultad accidental del software:
    - Lenguajes de alto nivel: Me da igual toda la aparte de codigo maquina
    - Time-sharing: Literal lo que hace el scheduler, no tengo que esperar un montón de tiempo a ver el resultado de mi programa.
    - Entornos de programacion unificados: Unix, cosa como docker, etc. Todo lo que evite tener problemas de compatibilidad.

Hopes for the sliver

- Ada y otros lenguajes High level languages: Hacen lo mas legible posible el código
- POO: Ayuda a expresarse mejor con menos codigo
- AI
- Expert systems: Ayudan a encontrar errores
- Automatic programming: Literal GPT
- Graphical programming
- Me rindo esto no tiene ningún sentido

Promising attacks of the conceptual essence

- Todas las soluciones a lo accidental se ven limitada por la formula esta:

$$Time\ of\ task = \sum (Frequency)_i \ x\ (Time)_i$$

- Bay versus build: Poner mucha gente a laburar con laptops.
- Requirements refinement and rapid prototyping: Formas de prototipar sistemas mas rapido.

- Incremental development-grow, not build, software: Ir haciendo pasos que agreguen funcionalidad de a poco teniendo siempre un sistema funcional, porque eso es motivante.
- Great designers: Agarrar a los tipos que mejor diseñan y mentorearlos, enseñar buenas prácticas a quienes programan.

Polymorphic hierarchy

- Para que dos métodos sean polimórficos no solo deben tener el mismo nombre sino también el mismo propósito
- Para que dos clases sean polimórficas deben compartir una interfaz común base (no hace falta que sean todos los métodos, si en SortedCollection agrego algo tipo print reversed solo para esa sigue siendo polimórfica con UnorderedCollection)
- Template class pattern: para hacer una serie de mensajes polimórficos se crea una clase abstracta tipo interfaz, esto no tiene ningún sentido si existen interfaces, sino, es la forma de simularlas, no es muy buena que digamos, pero es lo que hay para no repetir código al pedo, ver trade offs de acuerdo al caso.
- Lo ideal es que la template clase sea flexible, muy reusable y extensible, si se hace para cosas no lo suficientemente genéricas/compartidas queda horrible.

Simple technique for handling multiple polymorphism

- Si yo tengo DisplayPort, PrinterPort, AlgoPort y tengo figuras Rectangulo, Ovalo, Circulo que se representan en distintos puertos segun lo que sean, en rectangulo yo tendria este codigo por ejemplo:

  ```
  Rectangulo displayOnPort: unPuerto
      if unPuerto isKindOf: DisplayPort:
          algo
      if unPuerto isKindOf: PrinterPort:
          algo
      …
  ```

  Lo cual queda horrible porque este es el problema que venia a resolver POO con el polimorfismo. Como hago?

  ```
  DisplayPort displayRectangulo:
      codigo para mostrar rectangulo

  PrinterPort displayRectangulo:
      codigo para mostrar rectangulo

  ….

  Rectangulo displayOnPort: unPuerto
  ```

unPuerto displayRectangulo: Self

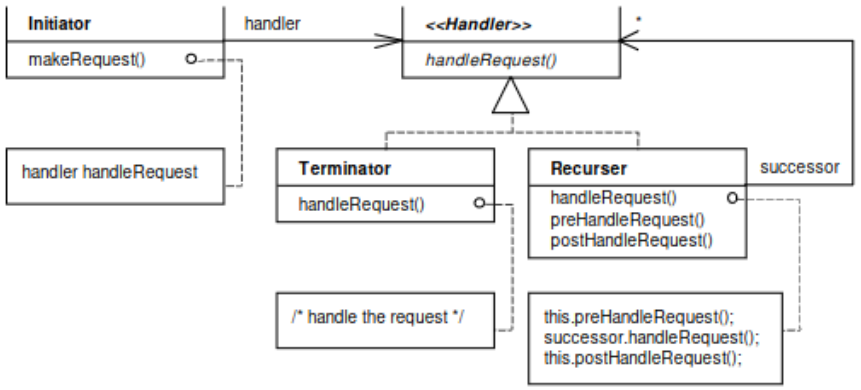
Pasando self como parametro el reciver gana informacion del tipo de dato y puede actuar de manera acorde. Queda mas modularizado el codigo, es mas extensible, y en la superclase puedo definir cada uno de estos como subclassResponsability.

Visitor, Composite y Object Recursion (En los TPs)

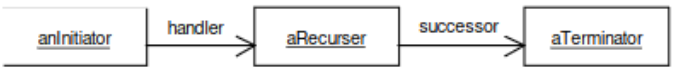
- Object recursion: El ejemplo que da es el de la comparacion, si yo le implemento la comparacion a mi objeto Engine (tiene horse_power: int, engine_size: int) le digo que haga self.horse_power == other.horse_power && self.engine_size == other.engine_size. Esto va a triggerear dos llamados a la comparacion de == con integer, lo cual va a devolver true o false.
- Esto es una forma generica de implementarlo, el ejemplo mas util es el de la igualdad, la suma con pandas es otro ejemplo de esto, la serializacion de objetos es otro.

Structure

The following class diagram shows the roles of the participants:



A typical object structure might look like this:



Object recursion, often referred to as recursive delegation, is a design pattern where an object delegates a task to a similar object of the same class, which in turn may delegate the task further to another object, and so on, until a base condition is met. This pattern is typically used in scenarios where a problem can be broken down into smaller, similar problems.

Here's a brief explanation:

1. **Initialization**: An initial object of a class is created and given a task.
2. **Delegation**: If the object cannot or should not handle the task entirely by itself, it creates another object of the same class and delegates the task to it.
3. **Recursion**: The newly created object follows the same logic: if it needs to, it delegates the task further by creating another object of the same class.
4. **Base Condition**: This process continues until a base condition is met, which is a scenario where the object can handle the task without needing to delegate it further.
5. **Completion**: Once the base condition is met, the task is completed, and the result is passed back up the chain of delegated objects.

This pattern is useful for tasks that can naturally be described by recursion, such as traversing a file system, processing nested data structures, or solving algorithmic problems that have recursive solutions (like tree traversals).

The Composite Pattern Explained and Implemented in Java | Structural Design Pat…
The Visitor Pattern Explained and Implemented in Java | Behavioral Design Patter…

- Composite: El portfolio era un composite, porque teníamos que calcular el balance total, y el portfolio podía tener otros portafolios o cuentas, ambos implementaban el método calculateBalance(), si era un portfolio, ejecutaba la operación para cada uno de los elementos que contenia (sin tener en cuenta la clase), si era una cuenta, directamente devolvió el balance

- El reporte era un visitor: Se hacía un reporte de una cuenta (ya sea portfolio o receptive account). A la cuenta se le pedían todas sus transacciones las cuales podrían ser depósitos o withdraws. Indistinto de la clase que sea, yo mando el mensaje transaccion.acceptNombreDelVisitor(self) para cada transacción, aceptar el visitor lo único que hace es llamar a visitor.reportNombreEspecificoDeLaTransaccion(self) para hacer un double dispatch, ahora el visitor sabe a quien está visitando y puede hacer la operación correspondiente.Todo esto es para separar la funcionalidad de hacer el reporte de las clases.

- La lógica es: Quiero separar funcionalidad de mis clases A (son varias concretas de una abstracta) => La muevo a una nueva clase B (unica), pero lo que termina ocurriendo, es que pierdo informacion de quien soy, porque si desde mi clase B intento llamar a mi funcionalidad para cada clase, lo que termina pasando es que

tendria que hacer es un for con un if por cada subclase, lo cual queda horrible => Uso double dispatch y le digo a mis clases A que acepten a mi visitor, y dentro de ese accept del visitor, lo unico que hago es agarrar y llamar al metodo que corresponda del visitor con self, para que el metodo del visitor pueda trabajar con mi clase especifica A