Arquitectura y Diseño de Software

Por Danilo Domínguez, PhD

Acerca de mí

@danilo04

@danilo04

- Danilo Domínguez Pérez
- Egresado de la UTP – Ing. en Sistemas y Computación
- Ingeniero de Software Senior en ADR Technologies
- Miembro de Flosspa
- Maestría en Ciencias Computacionales del Rochester Institute of Technology
- Doctorado en Ciencias Computacionales de Iowa State University
    - Investigación en Análisis y Testing de Aplicaciones Móviles

https://www.linkedin.com/in/danilo-dominguez-perez

Agenda

- Definición de Arquitectura de Software
- Componentes de Arquitectura (Architectural Drivers)
  - Fiabilidad
  - Mantenibilidad
- Complejidad en el Diseño de Software

¿Qué es Arquitectura de Software?

Arquitectura como **un nombre**	Arquitectura como **un verbo**
▪ Relacionado con estructura ▪ Descomposición de un producto en una colección de elementos más pequeños ▪ Y la relación de estos elementos	▪ Traducir **requerimientos funcionales** , **atributos de calidad** , **restricciones** en **una solución**

Software Architecture for Developers (Simon Brown, 2018)

Tipos de Arquitecturas

Arquitectura de Software

Arquitectura de Sistemas + Arquitectura de Aplicaciones

Arquitectura de Software

Arquitectura de Aplicaciones

- Deployable unit
- Puede ser un monolito, un microservicio, etc.
- Ejemplos:
  - Android App
  - Django App
  - Flask App

Arquitectura de Sistemas

- Compuesto de múltiple deployable units
- Ejemplos:
  - Flask App + PostgreSQL + Kafka + iOS APP
  - Microservicios (Mesh)

Componentes de Arquitectura

Architectural Drivers

Componentes de Arquitectura de Software

Restricciones de Negocio

- Marcan los límites del proyecto
- **Tiempo:** tiempo límite para desarrollar proyecto
- **Presupuesto:** proyectos tienen un presupuesto límite
- **Equipo:** miembros del grupo de diseño e implementación

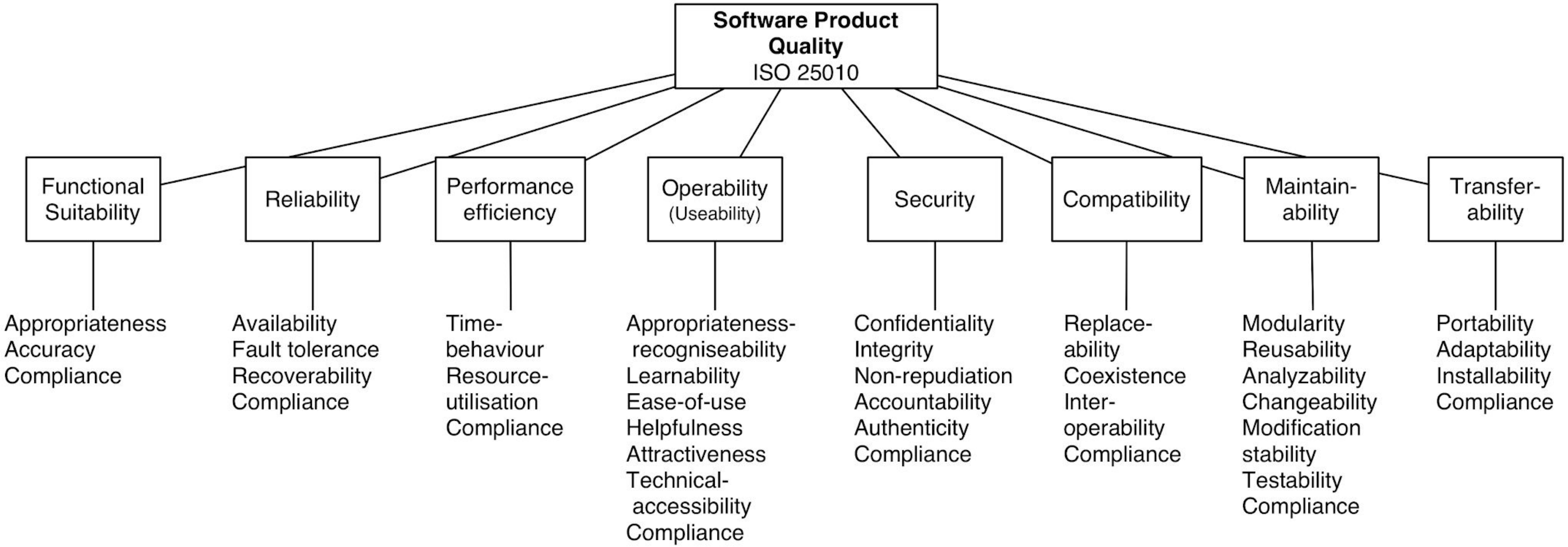
Restricciones Técnicas

- Decisiones tomadas por el equipo de desarrollo o stakeholders del proyecto
- **Lenguaje de Programación**
- **Soporte de plataformas:** e.g. solo iPhone 11+ y Android 10+
- **Librerías o Framework** : e.g. utilizaremos flutter y Django

Requerimientos Funcionales

- Representan **funciones (e.g. features)** de la aplicación a desarrollar
- Diferentes métodos para definirlos: **casos de uso o historias de usuario**
- Limitados por atributos de calidad

Atributos de Calidad (aka Requerimientos No Funcionales)



Fuente: https://faq.arc42.org/questions/C-1-2/

Fiabilidad

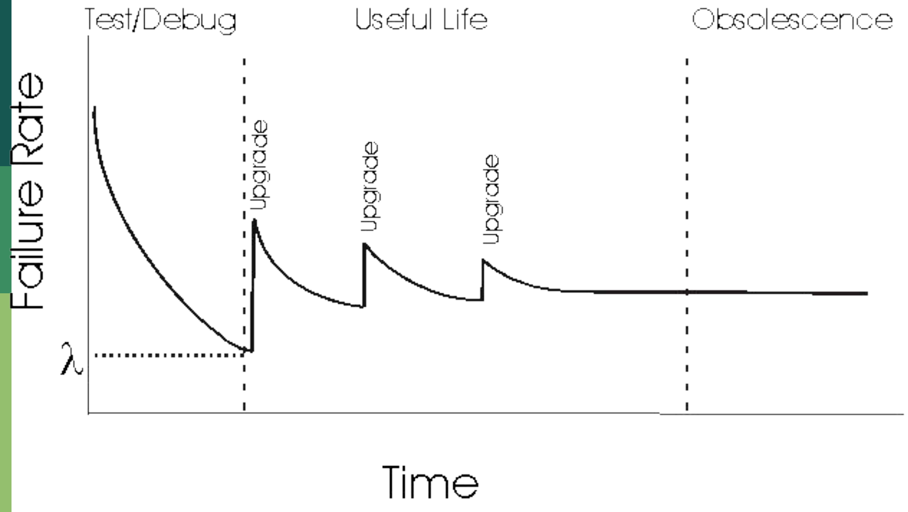
Reliability

“Fiabilidad (Reliability) de Software es definido: la probabilidad de cero fallas en la operación de un software durante un tiempo específico en un ambiente específico”

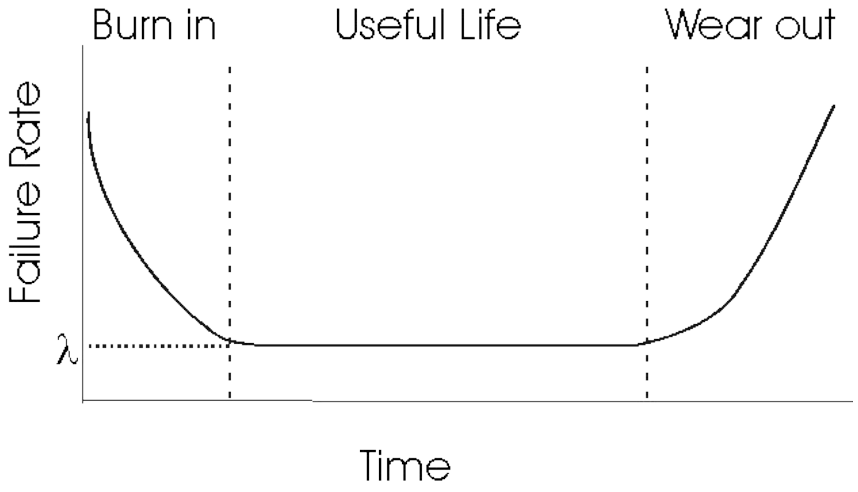
ANSI

Software Reliability vs Hardware Reliability

Software



Hardware



Fuente: https://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/

Mantenibilidad

Mantenibilidad

"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."

IEEE Standard Glossary of Software Engineering Terminology

- Facilidad para extender o modificar una aplicación o sistema
  - Nuevo feature
  - Fix bug
  - Nuevos ambientes

Complejidad en Diseño de Software

Complejidad en Desarrollo de Software

No Silver Bullet (Fred Brooks, 1996)

“Much of the complexity a software engineer must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which is interfaces must confirm.”

Complejidad en Desarrollo de Software

Essential Complexity

- Relacionado a la naturaleza del problema
- Lógica esencial del problema
- Es inherente al problema, no se puede reducir

Accidental Complexity

- Retos que los mismos ingenieros se asignan
- Se puede evitar o reducir
- E.g. Arquitecturas complejas, librerías complejas

No Silver Bullet (Fred Brooks, 1996)

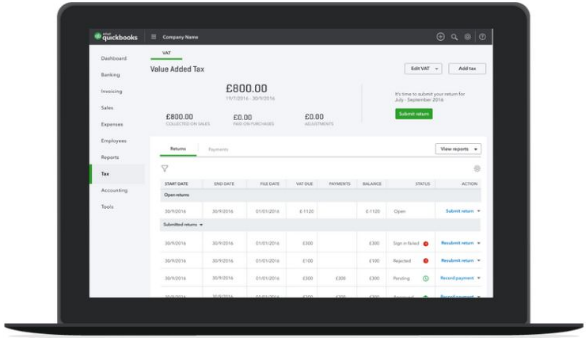
Essential Complexity

- Complejidad a nivel del dominio del problema
- Los algoritmos son difíciles de entender
- Sistemas simples crecen hasta sistemas complejos
- **Algoritmos diseñados de procesos definidos por clientes, managers o políticos caprichosos**
  - No hay una definición clara de requerimientos
  - Se mantienen en constante cambio
  - Procesos complejos

https://wiki.c2.com/?EssentialComplexity

¿Cuál software es más complejo para construir?

Chess engine that can consistently win against grandmasters?



Accounting system that calculates VAT in UK and France?

Fuente: http://tpetricek.github.io/Talks/2019/swe-principles/newcrafts/#/3/1

¿Agile?

Software Aspects of Strategic Defense Systems (David Parnas, 1985)

“Complex software can only be mastered if it is developed progressively, with the aid of extensive testing, and then operated more or less continually in a somewhat lenient and forgiving environment.”

REPORTS

SOFTWARE ASPECTS OF STRATEGIC DEFENSE SYSTEMS

A former member of the SDIO Panel on Computing in Support of Battle Management explains why he believes the “star wars” effort will not achieve its stated goals.

DAVID LORGE PARNAS

On 28 June 1985, David Lorge Parnas, a respected computer scientist who has consulted extensively on United States defense projects, resigned from the Panel on Computing in Support of Battle Management, convened by the Strategic Defense Initiative Organization (SDIO). With his letter of resignation, he submitted eight short essays explaining why he believed the software required by the Strategic Defense Initiative would not be trustworthy. Excerpts from Dr. Parnas's letter and the accompanying papers have appeared widely in the press. The Editors of American Scientist believed that it would be useful to the scientific community to publish these essays in their entirety to stimulate scientific discussion of the feasibility of the project. As part of the activity of the Forum on Risks to the Public in the use of computer systems the Editors of Communications *are pleased to reprint these essays.**

This report comprises eight short papers that were completed while I was a member of the Panel on Computing in Support of Battle Management, convened by the Strategic Defense Initiative Organization (SDIO). SDIO is part of the Office of the U.S. Secretary of Defense. The panel was asked to identify the computer science problems that would have to be solved before an effective antiballistic missile (ABM) system could be deployed. It is clear to everyone that computers must play a critical role in the systems that SDIO is considering. The essays that constitute this report were written to organize my thoughts on these topics and were submitted to SDIO with my resignation from the panel.

My conclusions are not based on political or policy judgments. Unlike many other academic critics of the SDI effort, I have not, in the past, objected to defense efforts or defense-sponsored research. I have been deeply involved in such research and have consulted extensively on defense projects. My conclusions are based on more than 20 years of research on software engineering, including more than 8 years of work on real-time software used in military aircraft. They are based on familiarity with both operational military software and computer science research. My conclusions are based on characteristics peculiar to this particular effort, not objections to weapons development in general.

I am publishing the papers that accompanied my letter of resignation so that interested people can understand why many computer scientists believe that systems of the sort being considered by the SDIO cannot be built. These essays address the software engineering aspects of SDIO and the organization of engineering research. They avoid political issues; those have been widely discussed elsewhere, and I have nothing to add.

In these essays I have attempted to avoid technical jargon, and readers need not be computer programmers to understand them. They may be read in any order.

The individual essays explain:

1. The fundamental technological differences between software engineering and other areas of engineering and why software is unreliable;
2. The properties of the proposed SDI software that make it unattainable;
3. Why the techniques commonly used to build military software are inadequate for this job;
4. The nature of research in software engineering, and why the improvements that it can effect will not be

* Reprinted by permission of *American Scientist*, Journal of Sigma Xi: Software Aspects of Strategic Defense Systems. David Lorge Parnas. Vol. 73, No. 5, pp. 432–440.

Accidental Complexity

Cómo Reducir Accidental Complexity

Complejidad en el tiempo

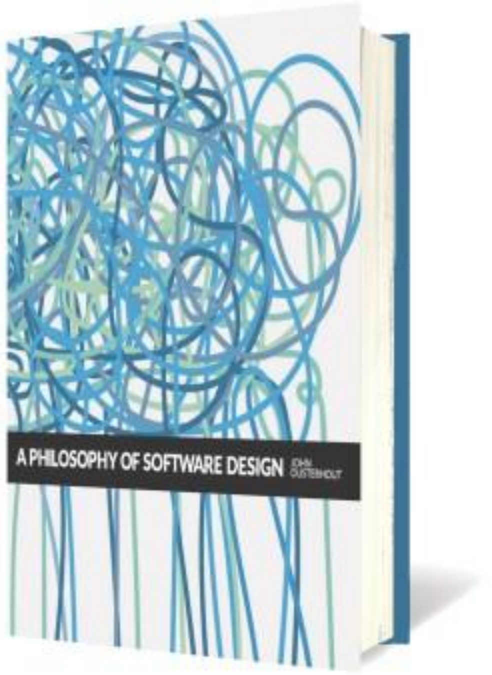
- A través del tiempo, un codebase acumula complejidad
- Es difícil para ingenieros de software mantener toda esa complejidad en su cerebro
- Dos factores contribuyen a reducir complejidad:
  1) Escribir código simple y obvio

  2) Diseño modular

Complejidad

A Philosophy of Software Design (John Ousterhout, 2018)

"Complexity is anything related to the structure of a software system that makes it hard to understand and modify the system"



Síntomas de Complejidad

- **Amplificación de cambio:** un simple cambio requiere cambios en múltiples lugares del codebase
- **Carga cognitiva:** cuánto los ingenieros deben saber para completar una tarea. E.g. APIs con muchos parámetros, variables globales, etc
- **Unknown unknowns:** no es claro que debo modificar para completar una tarea. Un buen diseño es hacer un sistema obvio

Causas de Complejidad

- La complejidad en software tiene dos causas principales: 1) dependencias, y 2) oscuridad
- **Dependencias:**
  - Dependencias entre módulos (acoplamiento)
  - Dependencias entre sistemas
  - Código altamente acoplado
- **Oscuridad:**
  - Relacionado con dependencias que no sabemos que existen
  - Al agregar un nuevo feature, existen dependencias oscuras?
- La complejidad es **incremental**
  - Una vez se ha acumulado en el tiempo, es difícil de eliminarla

Recomendaciones para Reducir Complejidad

Módulos deben ser profundos	Ocultación de Información

Módulos deben ser profundos

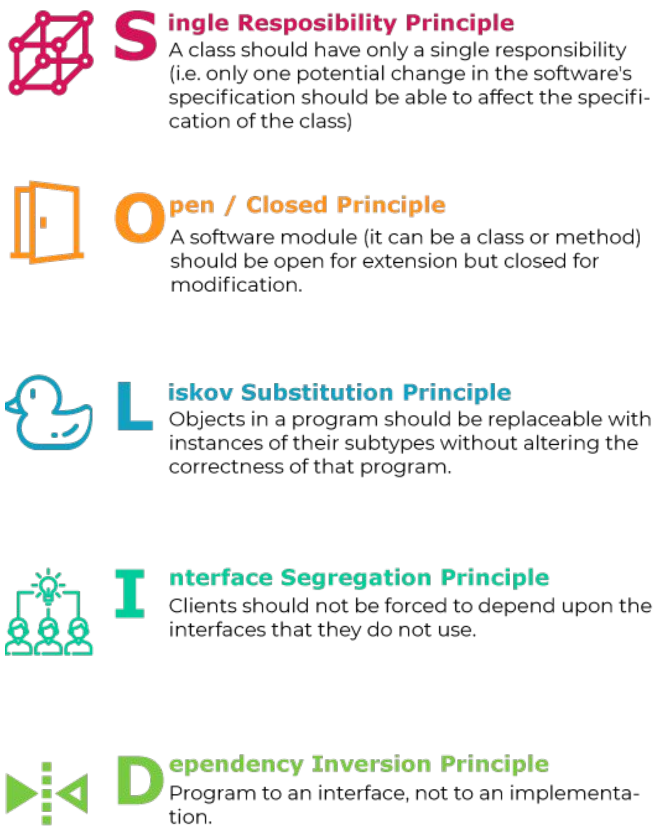
- Minimizar dependencias entre módulos
- La interfaz debe ser simple
- No incluir abstracciones que no son importantes

Ocultación de Información

- Ayuda a evolucionar el sistema
- La fuga de información entre módulos es un anti-patrón
- Diseñar para el caso más común

SOLID Principles

- Principios utilizados en OOP para mejorar el diseño de software
  - Mejorar comprensibilidad
  - Flexibilidad
  - Mantenibilidad
- Ayudan a mejorar métricas de calidad:
  - Low coupling
  - Strong encapsulation
  - High cohesion



PREGUNTAS