ANEXO

PROGRAMA DE ASIGNATURA	
ASIGNATURA: Técnicas Avanzadas de Diseño de Software	**AÑO:** 2025
CARACTER: Optativa	**UBICACIÓN EN LA CARRERA:** 5° año 1° cuatrimestre
CARRERA: Licenciatura en Ciencias de la Computación	
REGIMEN: Cuatrimestral	**CARGA HORARIA:** 120 horas.

FUNDAMENTACIÓN Y OBJETIVOS

Fundamentos

En la actualidad, el diseño de software enfrenta retos significativos debido a la creciente complejidad de los sistemas y la necesidad de mantener estándares de calidad. Este programa está diseñado para ofrecer a los/as participantes una comprensión profunda de las técnicas avanzadas en diseño de software, abarcando tanto los principios teóricos, las prácticas aplicadas, análisis de software real y tendencias futuras de modelos generativos e inteligencia artificial.

Si bien algunos conceptos pueden haber sido introducidos en otras asignaturas, este curso los retoma por necesidad y profundiza desde una perspectiva integradora y aplicada, enfocándose en su implementación en sistemas complejos del mundo real y en el contexto de las tendencias tecnológicas emergentes, proporcionando así una visión más orientada a la práctica profesional actual.

Objetivos

Comprender la importancia de los principios fundamentales del diseño de software para crear sistemas robustos y mantenibles, así como las causas y efectos de la entropía en el software y las estrategias modernas de diseño para mitigar su impacto en la calidad y escalabilidad. Adquirir herramientas y metodologías para asegurar la calidad del software a través de buenas prácticas, fomentando la aplicación de patrones de diseño y heurísticas que optimicen la mantenibilidad y el rendimiento del código. Dotarse de herramientas teórico-prácticas para el desarrollo de software para grandes volúmenes de datos y así como proyectos que integren modelos modernos de inteligencia artificial.

CONTENIDO

Fundamentos diseño y entropía del software

Revisita el diseño de software, su definición y objetivos. Los principios fundamentales del buen diseño. Contexto histórico desde la creación de las primeras computadoras, hasta el cloud computing e inteligencia artificial. Las lecciones aprendidas de "The Mythical Man-Month" de Frederick Brooks. Ley de Brooks. Complejidad de la comunicación en equipos grandes. El efecto del segundo sistema. Entropía el contexto del software. Entropía como fenómeno físico. Causas del aumento de la entropía: cambios frecuentes, falta de documentación, deuda técnica, inconsistencias en el diseño, código legacy, complejidad creciente. Impacto en la mantenibilidad y escalabilidad. El buen diseño como estrategias de mitigación. Casos de estudio: Análisis de impacto y lecciones aprendidas de cada caso. Desafíos continuos en el diseño de software. Ley de Wirth: crecimiento de la complejidad del software vs. métodos para manejarla.

Calidad

Calidad total, definición de calidad de software. Importancia de la calidad en el desarrollo de software. Conceptos fundamentales: verificación y validación. Deuda técnica. Tipificación de casos de code smells y de refactoring. Herramientas y métricas de análisis estático y dinámico de código. Análisis de sistemas de versionado de código para el estudio de tendencias. Triangulación de

problemas de rendimiento. Análisis de cobertura de código y pruebas de mutación. Testing avanzado: Tests basados en propiedades, fuzzy testing, test de comportamiento (behavioral testing). Análisis de sensibilidad paramétrico. Administración de datos de tests (Test data management). Generación de datos ficticios (Faker) y anonimización.

Diseño

Principios de simplicidad, no necesidad y no repetición (KISS, DRY y YAGNI). Patrones de diseño para creación, comportamiento y estructura de objetos y funciones, y su implicancia en mantenibilidad, seguridad y rendimiento del código. Heurísticas de diseño orientado a objetos: inmutabilidad; lenguaje ubicuo y metáforas del dominio; valores nulos (null object pattern); Arquitectura del sistema en el código; entidades del dominio como objetos; invarianza de objetos; encapsulamiento; objetos cohesivos y completos. Clean architecture, clean code principles, boy scout rule, command query separation (CQS), single responsibility principle (SRP) para arquitectura, dependency rule, humble object pattern, transformation priority premise, four rules of simple design y STABLE patterns. Aplicación de estos principios en el diseño y desarrollo de software de calidad. SOLID, CUPID y GRASP. Patrones para ciencia de datos: Estimador-transformador (template method), pipeline/composite y accessor.

Optimización y gran volumen de datos

Proceso y costos de la optimización. Niveles de optimización (código fuente, diseño, tiempo de ejecución). Soluciones simples a problemas de optimización. Reemplazo por código de bajo nivel, compiladores justo a tiempo. Arreglos, data frames y actores para gran volumen de datos. Ray, Spark y Dask. Datos en alta dimensión (Xarray). Uso de patrones de diseño para el paralelismo y concurrencia (caso Joblib y Trio). Monkeypatching para paralelismo y rendimiento. Principios y aplicaciones del monkeypatching en bibliotecas de alto rendimiento. Caso de estudio: Modin y su enfoque para paralelizar Pandas. Grafos acíclicos dirigidos como patrón de diseño para pipelines (Luigi, Bonobo y Airflow). Sistemas de colas vs. sistemas de streaming. Casos: AMQP y Apache Kafka.

Diseño e inteligencia artificial

Proceso de diseño de aplicaciones que involucran modelos probabilísticos. Modelos generativos dentro del proceso de diseño y desarrollo. Reutilización de modelos (Transfer learning). IA en el Edge, Automatización del proceso de aprendizaje (AutoML), IA e Interpretabilidad, pruebas y validación de modelos, sistemas de recomendación avanzados, pruebas de robustez y seguridad.

BIBLIOGRAFÍA

BIBLIOGRAFÍA BÁSICA

- Brooks, F. P. (1975). *The mythical man-month: Essays on software engineering*. Reading, MA: Addison-Wesley.
- Fowler, M. (2018). *Refactoring: Improving the design of existing code- (2nd ed.). Boston, MA: Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.
- Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Upper Saddle River, NJ: Prentice Hall.
- Martin, R. C. (2011). *The clean coder: A code of conduct for professional programmers*. Upper Saddle River, NJ: Prentice Hall.
- Thomas, D., & Hunt, A. (2019). *The pragmatic programmer: Your journey to mastery- (20th Anniversary Edition). Boston, MA: Addison-Wesley.

BIBLIOGRAFÍA COMPLEMENTARIA

- Amershi, Saleema, et al. (2019). "Software engineering for machine learning: A case study." IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).

- Bagare, P., & Desyatnikov, R. (2018). Test Data Management in Software Testing Life Cycle-Business Need and Benefits in Functional, Performance, and Automation Testing.
- Buitinck, L., et al. (2013). API design for machine learning software: experiences from the scikit-learn project. arXiv preprint arXiv:1309.0238.
- Burkov, A. (2020). *Machine learning engineering*. True Positive Inc.
- Fink, G., & Bishop, M. (1997). Property-based testing: A new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes.
- Folk, M., Cheng, A., & Yates, K. (1999). HDF5: A file format and I/O library for high performance computing applications.
- Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media.
- Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler.
- McKinney, W. (2011). pandas: a foundational Python library for data analysis and statistics.
- Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling.
- Russell, S., & Norvig, P. (2020). *Artificial intelligence: A modern approach- (4th ed.). Pearson.
- Shvets, A. (2019). *Dive into design patterns*. Independently published.

EVALUACIÓN

FORMAS DE EVALUACIÓN

La evaluación consistirá en dos prácticos que involucran la aplicación de los conceptos teóricos y prácticos:

Análisis de un proyecto de software existente: El/La estudiante realizará un informe que incluya las primeras dos unidades de la materia, realizando mediciones de métricas y gráficos que representen la calidad del proyecto analizado.

Propuesta y reingeniería: Con el informe realizado, se elaborará una propuesta de mejora para el software y la reingeniería/refactorización de las críticas encontradas, aprovechando las técnicas de diseño aprendidas en las unidades 3, 4 y 5.

REGULARIDAD

Ccumplir un mínimo de 70% de asistencia a clases teóricas, prácticas, o de laboratorio y aprobar los dos informes.

PROMOCIÓN

No hay régimen de promoción.

CORRELATIVIDADES

Para cursar: Paradigmas de Programación (regularizada) e Ingeniería de Software I (regularizada)

Para Rendir: Paradigmas de Programación (regularizada) e Ingeniería de Software I (regularizada)