Tárgytematika / Course Description
Production Software Development

GKNM_MSTA092

Tárgyfelelős neve / Teacher's name: dr. Bakosi József
Félév / Semester: 2026/27/1
Beszámolási forma / Assesment: Folyamatos számonkérés
Tárgy heti óraszáma / Teaching hours(week): 2/2/0
Tárgy féléves óraszáma / Teaching hours(sem.):

OKTATÁS CÉLJA / AIM OF THE COURSE

Prepare students for professional, production-quality software development in industrial, research, and collaborative engineering environments. Students learn modern software development tools and practices, including version control, build systems, testing, debugging, software design, code quality, teamwork, code review, and responsible use of AI-assisted development tools. The course concludes with a short individual software project and oral presentation in which each student demonstrates the acquired production-software-development practices.

TANTÁRGY TARTALMA / DESCRIPTION

1. Key concepts, software engineering, types of software development
Concepts. The difference between computer science and software engineering. Computer vs. computational science. Research vs. production code. Why should you care? Production qualities: correctness, maintainability, reliability, portability, reproducibility and operability.

2. Tools for productivity
Terminal and terminal multiplexers. Shell utilities. File managers. Editors/IDEs. Code search and navigation. Symbolic and numeric math tools. Build and documentation automation. Testing tools and code-analysis tools. Linux-first workflow using Ubuntu LTS as the reference environment, while keeping tools and project structure portable. Git, libraries as tools, code-review tools and team communication tools.

3. Version control
Why use version control? Git, GitHub/GitLab and alternatives. Basic and intermediate usage. Proper commit messages. Branches, merge/pull requests, resolving conflicts, stash, tags, revert/reset, rebase, bisect and safe history rewriting.

4. Build systems
What are build systems? Why use build systems? Build correctness, performance, automation and portability. Parallel and distributed builds. Scalability. Off-the-shelf and custom build systems. CMake, GNU Make and Ninja. Modern CMake targets, presets, build types, dependency discovery and reproducible builds.

5. Documentation
Why document? Specifics for computational and production codes. Requirements, specification, design, implementation and interfaces. Verification/validation cases and user examples. Source-control history as documentation. Documenting and archiving team collaboration. Documenting code correctness and tracking code quality.

6. Testing, continuous integration, code quality

Why test? Unit, integration, regression, acceptance, verification and validation tests. Performance testing and regression. Static and dynamic code analysis. Test-driven development where appropriate. Testing frameworks and libraries. Build-system integration, fuzz testing and continuous integration: automated code review, builds and tests.

7. Debugging, instrumentation and profiling

Measurement of test/code coverage. Instrumentation. Memory-error and thread-error detection. Cache-, branch- and heap-profiling. Systematic debugging: reproduce, minimize, observe, hypothesize, test and verify. Logs, assertions, stack traces, core dumps, gdb/lldb, sanitizers, Valgrind, strace, perf and Git bisect. Memory corruption, undefined behaviour, races and deadlocks. Root-cause analysis and preventing recurrence.

8. Programming styles and maintainable code

Procedural, object-oriented, generic and functional programming styles. Languages and choosing the right tool for the job. Code correctness, power consumption, compile/runtime performance, maintainability, productivity and user/developer friendliness. Refactoring, readability, ownership and avoiding unnecessary complexity.

9. Software design, architecture and design patterns

Why design production code? Process, requirements, specifications, value and artifacts. Design principles and concepts. Modeling languages and design considerations. Design patterns and priorities for writing code. Practical examples: Strategy, Factory, Adapter, Observer, State, Command, RAII and dependency injection. Coupling, cohesion, module boundaries, anti-patterns and avoiding over-engineering.

10. Third-party libraries and dependency management

Pros and cons of third-party libraries. Libraries for computational code. Maintenance and upstream contributions. Selecting dependencies based on API quality, maintenance, licensing, portability, performance and security. Version pinning, update strategy, vulnerability awareness and software-supply-chain basics.

11. Software development in teams, effective communication

Effective team communication and archiving communication. Centralized vs. decentralized and synchronous vs. asynchronous communication. The Cathedral and the Bazaar. Available tools. Agile, waterfall, feature-driven and extreme-programming concepts. Issues, milestones, ownership, technical decisions, handover and distributed-team collaboration.

12. Code review

The importance of code review. The de facto standard open-source development process and pull/merge-request workflows. Code review in distributed teams. Review for correctness, maintainability, tests, design, performance and security.

13. User-friendly input, design partners and feedback-driven development

The importance of user-friendliness. Balance between user-friendly and developer-friendly, and between foolproof and expert-user input. Automated user input for verification, validation and uncertainty quantification. Working with design partners/pilot users: requirements discovery, prototypes, measurable acceptance criteria, structured feedback, prioritization and validation.

14. AI-assisted production software development

Modern AI coding assistants and coding agents for code understanding, implementation, refactoring, migration, tests, documentation and code review. Repository context, task decomposition and effective instructions. Production-code rules: the developer remains responsible for correctness; generated code must be understood, reviewed, built, tested and validated. Hallucinations, insecure code, outdated APIs, privacy/IP concerns and over-reliance. AI-assisted debugging and root-cause analysis. Applying AI-assisted development responsibly within a production-software-development workflow.

SZÁMONKÉRÉSI ÉS ÉRTÉKELÉSI RENDSZERE / ASSESMENT'S METHOD

ZÁRÓ EGYÉNI PROJEKT ÉS RÖVID SZÓBELI BEMUTATÓ / FINAL INDIVIDUAL PROJECT AND SHORT ORAL PRESENTATION

Instead of a traditional final examination, each student prepares a short individual software project. The project should

demonstrate the student's ability to independently apply relevant software engineering practices covered during the course, such as version control, build automation, testing and CI, debugging, software design, documentation, dependency management, and responsible use of AI-assisted development tools. The completed project must be submitted to the instructor at least one week before the scheduled assessment for review. Based on the submitted work and in-class activity, the instructor may offer a grade. During the short oral assessment, the student demonstrates the project and briefly explains the most important implementation, design and engineering decisions, including why the chosen solutions were used. The discussion may also cover relevant course topics as they appear in the student's own project.

SZÁMONKÉRÉSI ÉS ÉRTÉKELÉSI RENDSZERE / ASSESSMENT'S METHOD

Continuous assessment is complemented by a short individual final project and oral presentation instead of a traditional final examination. The project must be submitted at least one week before the scheduled assessment for instructor review; a grade may be offered on the basis of the submitted work and in-class activity. If an oral assessment is held, the student briefly demonstrates the project and explains the relevant technical and engineering decisions. Attendance/signature requirements may remain consistent with the existing course. Grades are primarily based on in-class activity, demonstrated engineering practice, and the individual final project.

Component / Suggested weight

In-class activity, exercises and participation 35%

Individual final project 40%

Short oral presentation and explanation of engineering decisions 25%

KÖTELEZŐ IRODALOM / OBLIGATORY MATERIAL

AJÁNLOTT IRODALOM / RECOMMENDED MATERIAL