**1. Maintainability**

**Why It Matters**

The **real-world complexity** of university systems grows over time. That’s because maintainability matters. **Evolving Requirements,** from the project brief, we already anticipate possible changes (e.g., new types of TA tasks, changes in proctoring logic & swap rules, or integration with other departments). A maintainable codebase makes it easier to accommodate these changes without breaking the entire system. **Long-Term Ownership** requires outlive our development team. If the department or future students maintain the system, it needs to be **easy to understand, extend, and debug**. Another important aspect is **bug fixes and enhancements:** A maintainable system should be easily modified to fix bugs or introduce enhancements (like new reporting tools or more intelligent assignment algorithms) without extensive rewrites. **Onboarding New Developers** is also important. Clear structure, good documentation, and modular design help new developers onboard quickly, reducing the ramp-up time when new contributors join.

**Trade-offs:** Although maintainability is really important, it does not come free cost. Having this maintainability feature results in some losing other aspects.

**Maintainability vs. Performance**

* Writing highly maintainable code often means favoring **clarity** over **efficiency**. For instance, we might use abstract layers or ORM (like Hibernate) to make code easier to read and modify — but this can come at the cost of raw performance compared to hand-optimized SQL or streamlined procedural logic

### Maintainability vs. Increased Productivity (Short-term)

Focusing on maintainability can **reduce short-term productivity** for us because:

* **More time is spent on structure and documentation:** Instead of just building features, we spend time creating abstractions, writing clean code, and documenting it for future developers.
* **We may avoid "quick wins":** Writing temporary code that "just works" is often faster, but maintainable design encourages longer-term thinking — leading to slower short-term delivery.
* **Extra work on modularity:** Maintainable code involves setting up proper architectures (like MVC), separating concerns — all of which require effort that doesn’t immediately show user-facing results.

**2. Rapid Development**

**Why It Matters**

We have **only two months for implementation**, and a functioning system by the deadline is more valuable than a exact-perfect one that never ships. Rapid development helps us **deliver working software fast**, iterate based on feedback, and meet our deadline. Since we are developing our app **deadline-driven,** without a deployable system, none of the features or architectural dreams matter. Rapid development enables us to reach the **working-app** fast. Also **agile feedback loops,** a working system — even a basic one — lets stakeholders (faculty, TAs, etc.) test features early, provide feedback, and refine the vision. By rapid development we can also **avoid burnout.** Spending too much time perfecting early features might leave us scrambling later. Rapid development emphasizes working smart and delivering just enough to move forward.

**Trade-offs** Same as maintainability, rapid-development comes with cost.

**Rapid Development vs. Functionality**

* When we prioritize speed, we often have to **defer or cut lower-priority features**, leading to reduced **functionality** in the initial version.
* This is a **classic trade-off**: We deliver **faster**, but the system might not fully cover all edge cases, exceptions, or nice-to-have features.

**Rapid Development vs. Reliability**

When we prioritize **Rapid Development**, we often sacrifice some level of **reliability** in the initial stages.

* **Less time for thorough testing:** We may skip writing comprehensive unit tests, integration tests, or corner case handling.
* **More shortcuts and assumptions:** We might hard-code certain values, ignore error-checking, or assume ideal user behavior just to “make it work.”
* **Higher risk of bugs or breakdowns** in real-world usage, especially under unexpected inputs or workflows.