Principles of sustainability to create viable systems

In our Software Innovation Studio course this semester, my team and I were tasked with integrating the principles of sustainability into the design and development of a software product. We chose to build an AI-assisted travel planning website that offered features like personalized itineraries, a digital travel journal, and AI-driven route optimization. Excited by the possibilities of AI, we prioritized adding novel functionalities over structural considerations. As development progressed, our codebase grew increasingly fragmented: components were tightly coupled, documentation was minimal, and new features were hastily layered onto existing ones without clear architectural boundaries. By mid-project, adding even minor updates required extensive debugging, and our system's responsiveness began to suffer. The technical debt accumulated so rapidly that we spent more time fixing bugs than developing new capabilities—a clear sign that our system lacked long-term viability.

Initially, I felt enthusiastic and proud of our "feature-rich" prototype. However, as technical debt accumulated, frustration set in. I became anxious during team meetings when simple changes triggered unexpected bugs. I also felt a growing sense of guilt—especially after our instructor pointed out that our approach contradicted the very sustainability principles we were supposed to embody. The excitement I once felt about AI capabilities was replaced by concern: had we compromised long-term viability for short-term innovation? I started questioning whether our product could realistically be maintained beyond the semester, let alone scaled for real-world deployment.

I used to equate "good software" primarily with user-facing novelty and performance. I believed that if a product looked impressive and worked in demos, it was successful. This project challenged that assumption. I now recognize that sustainability in software isn't just about environmental impact—it's about maintainability, modularity, adaptability, and human resource efficiency over time. A system that can't be easily updated, understood, or scaled is fundamentally unsustainable, no matter how "smart" its features appear. Sustainable software must serve not only end users but also future developers who inherit the codebase.

The core problem was our misalignment between intention and execution. While we verbally committed to sustainable design, our actions prioritized speed and novelty. We lacked clear coding standards, version control discipline, and architectural foresight. Crucially, we failed to define what "sustainable software" meant in practical terms for our context. This ambiguity allowed us to rationalize poor design choices as "temporary" solutions, which quickly became permanent liabilities. The lack of regular code reviews and modular planning further exacerbated maintainability issues. We also neglected to consider the entire software lifecycle—focusing only on launch, not long-term operation or eventual decommissioning.

Reflecting on academic literature and course materials, I now understand that sustainable engineering—whether in civil infrastructure or software—requires proactive lifecycle thinking. Dowling et al. (2016) emphasize that sustainable systems must balance environmental, social, economic, and technical constraints from the outset. Similarly, Hicks, Crittenden, and Warhurst (2000) argue in their "Design for Decommissioning" framework that neglecting end-of-life considerations leads to higher future costs and environmental burdens. Although their context is chemical plants, the principle applies directly to software: if we don't design for future maintenance, adaptation, or graceful shutdown, we create technical liabilities.

Moreover, ISO/IEC 25010 explicitly identifies maintainability as a core quality attribute of sustainable software systems. Our failure to adopt modular design patterns like MVC or microservices meant our system violated this standard. From an ethical standpoint—as outlined in the Engineers Australia Code of Ethics (Dowling et al., 2016)—engineers have a duty to promote sustainability. Building unmaintainable software wastes developer time, organizational resources, and ultimately contradicts the triple bottom line of sustainability: people, planet, and profit.

This experience reshaped my understanding of sustainability in software engineering. I now see that viable systems are built on thoughtful architecture, not just flashy features. Moving forward, I intend to: (1) define concrete sustainability criteria at project inception—especially around modularity, documentation, and lifecycle planning; (2) enforce regular code reviews, architectural planning sessions, and adherence to design patterns; and (3) treat technical debt as a critical risk, not a minor inconvenience. In future team projects, I will advocate for "sustainable by design" principles from day one, ensuring every feature is evaluated not just for its immediate utility but for its long-term maintainability. Ultimately, I've learned that true innovation lies not in how many features you add, but in how well your system endures change—making sustainability not an optional add-on, but the very foundation of viable, ethical engineering.

Professional Practice within intercultural and global contexts

During my Software Development Studio course at the University of Technology Sydney (UTS), I was assigned to a multicultural team composed of students from India, Australia, the Middle East, and China. Over the semester, we were tasked with collaboratively designing and building a full-stack software application. From our first meeting, I observed pronounced differences in how teammates approached technical discussions: some confidently advocated for specific frameworks like React Native or Firebase based on prior projects or regional tech ecosystems, while others prioritized agile workflows or user-experience heuristics unfamiliar to me. As someone trained primarily in China's academic computing environment—where structured, top-down project planning is typical—I found myself unable to keep pace with the spontaneous, debate-driven brainstorming sessions. Consequently, I retreated into a narrow role: maintaining and adjusting the database according to others' specifications without offering input on architecture, usability, or scalability.

This retreat was driven by mounting anxiety and self-doubt. Although I possessed solid programming skills, the speed and informality of team conversations—often peppered with idioms, domain-specific acronyms, or culturally embedded assumptions about "best practices"—left me feeling intellectually out of sync. I feared that asking for clarification would slow the team down or mark me as incompetent. Worse, I worried that proposing an alternative approach might disrupt group harmony, a value deeply embedded in my upbringing. As weeks passed, my contributions grew increasingly mechanical: I coded what I was told, rarely questioned design choices, and avoided initiating conversations, even when I suspected potential data-modeling flaws. Internally, I felt like a ghost—physically present but functionally invisible.

Upon reflection, I realized this passivity was not about technical ability but cultural conditioning. In collectivist, high-context societies like China, communication prioritizes relational harmony over individual expression (Samovar, Porter & McDaniel, 2009, p. 215). Silence is often a sign of respect, not disengagement. I had unconsciously carried this norm into a low-context, individualistic team where explicit, assertive input is expected and valued. My assumption—that consensus emerges naturally through mutual understanding rather than open debate—prevented me from recognizing that my silence was being interpreted as apathy or lack of initiative.

The real challenge was not language fluency but a mismatch in communicative expectations and epistemic styles. I failed to see that my non-participation deprived the team of diverse perspectives and created integration risks between my backend work and the frontend logic. More critically, by conflating "listening" with "agreeing," I avoided necessary clarification, which later led to rework when API contracts between services didn't align. This wasn't merely a coordination failure—it was a breakdown in intercultural communication that compromised both product quality and team trust.

Bennett's (2004) Developmental Model of Intercultural Sensitivity (DMIS) helps diagnose my stance: I was in the *Minimization* stage, assuming that "good engineering" is culturally neutral and that shared technical goals would naturally override communication differences. However, Hofstede's cultural dimensions (Samovar et al., 2009, pp. 198–207) reveal a deeper tension: my high power distance and collectivism clashed with teammates from low power distance, individualistic cultures (e.g., Australia) who viewed equal participation as essential to fairness and innovation. Additionally, Hall's high-/low-context framework (Samovar et al., 2009, pp. 215–220) explains why I struggled: in my high-context background, much meaning is inferred from relationships and shared history, while my peers relied on explicit, verbalized information—a mismatch that rendered my silence ambiguous rather than respectful.

This experience transformed my understanding of global engineering practice. Technical skill alone is insufficient without intercultural communication competence. Inspired by Gentile's (2016) "Giving Voice to Values" approach, I now recognize the value of pre-scripting questions ("Could you walk me through why this framework fits our use case?") to engage constructively without appearing confrontational. Going forward, I will proactively co-create a "team communication charter" at project inception, establishing norms for feedback, decision-making, and knowledge sharing. I also plan to research dominant development paradigms in key global tech hubs to better anticipate cultural-technical assumptions. Most importantly, I aim to shift from passive adaptation to active co-creation—leveraging my cultural perspective as an asset, not a liability—in the collaborative spirit of truly global engineering.

References

Bennett, M. J. (2004). Becoming interculturally competent. In J. S. Wurzel (Ed.), *Toward multiculturalism: A reader in multicultural education* (pp. 43–56). Newton, MA: Intercultural Resource Corporation.

Dowling, D., Hadgraft, R., Carew, A., McCarthy, T., Hargreaves, D., & Baillie, C. (2016). *Engineering your future: An Australasian guide* (3rd ed.). Wiley.

Gentile, M. C. (2016, December 23). Talking about ethics across cultures. *Harvard Business Review*. https://hbr.org/2016/12/talking-about-ethics-across-cultures

Hicks, D. I., Crittenden, B. D., & Warhurst, A. C. (2000). Design for decommissioning: Addressing the future closure of chemical sites in the design of new plant. *Transactions of the Institution of Chemical Engineers, Part B: Process Safety and Environmental Protection, 78*(6), 465–479. https://doi.org/10.1205/095758200774251333

International Organization for Standardization. (2011). *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. ISO.

Samovar, L. A., Porter, R. E., & McDaniel, E. R. (2009). *Communication between cultures* (7th ed.). Wadsworth.