EXERCISES – Software Engineering (Ian Sommerville)

Chapter 1

1 **Explain why professional software that is developed for a customer is not simply the programs that have been developed and delivered.**
Professional software includes not only the programs that have been developed and delivered, but also:

a. Associated documentation, such as user or system documentation. User documentation should explain how to use the programs, and system documentation should go into more technical details about how the systems work together. The latter is usually targeted at software developers, and other technical roles.
b. Associated libraries.
c. Support websites.
d. Configuration data that are necessary to make these programs work.

Indeed, professional software might include multiple separate programs and configuration files working together as a system. This is an important distinction between professional and amateur development: the latter does not assume maintainability, useability, nor dependability.

2 **What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?**
A generic software product is intended for the open market, whereas a custom software product has been designed for a specific customer / business case. The main distinction between generic software product development and custom software development, is that, for the latter, the developers must adhere to specifications that are determined by the customer. In contrast, for generic products, the organisation that develops the software controls the software specification, making it more flexible for developers: if they run into development problems, they can rethink what is to be developed.

In practice, this distinction might mean that users of generic software products must rely on the organisation that has developed the product. If the organisation did not follow good software engineering practices, for example, the product may be unreliable, or unsecure, putting the final users at higher risk. In addition, users might need to adapt their processes to fit the software rather than having software tailored to their processes. Furthermore, generic products might not perfectly

match all users' specific needs – with users normally having limited influence over product features compared to custom software clients. Of course, there may also be potential benefits, such as lower costs, more extensive testing due to larger user base, and potentially more regular updates.

3 **What are the four important attributes that all professional software should have? Suggest four other attributes that may sometimes be significant.**
Four important attributes that all professional software should have, are:

a. **Acceptability** – software must be understandable, usable, and compatible with other systems users may utilise.
b. **Dependability and security** – dependability includes a range of characteristics including reliability, security and safety. In the event of system failures, dependable software should not cause physical or economic damage. Malicious users should be prevented from exploiting the system by dependable software.
c. **Efficiency** – software should not make wasteful use of system resources such as memory and processor cycles.
d. **Maintainability** – software should be written in a way that would allow for the system to meet the changing needs of the business. Change is the only constant in software development.

Four other attributes that may sometimes be significant in professional software are:

e. **Compliance** – software must be designed to adhere to data privacy and retention regulations, such as the European Union's General Data Protection Regulation (GDPR). For example, if under GDPR's jurisdiction, software must process data lawfully, fairly, and in a transparent manner. Organisations developing software must clearly communicate how they collect and use personal data.
f. **Sustainability** – this attribute could be related to efficiency, but it goes further in making environmental sustainability as a priority. Organisations should take their software carbon footprint into account in the way it is designed, developed, and deployed and by rethinking some aspects of how the data centres that provide cloud-based services operate. For example, sustainable coding practices include optimising algorithms, reducing code complexity, and minimising the use of resources. Carbon-efficient software results from multiple factors – from selecting energy-efficient programming languages and platforms to "green" architecture design and DevOps. For example, interpreted languages consume 48x times more energy than compiled ones.
g. **Scalability** – this is the property of a system to handle a growing amount of work. In many cases it refers to the software's ability to handle increased workloads while adding users and removing them with minimal cost impact.

h. **Testability** – software testability is the degree to which a software artifact (e.g. a software system, module, requirement, or design document) supports testing in each test context. If the testability of an artifact is high, then finding faults in the system (if any) by means of testing is easier.

4 Apart from the challenges of heterogeneity, business and social change and trust and security, suggest other problems and challenges that software engineering is likely to face in the 21st century.

One of the greatest challenges software engineering faces in the 21st century is the Artificial Intelligence (AI) market hype. Over-reliance on AI tools by developers there's a risk of homogenised solutions being more vulnerable to systematic exploitation putting cybersecurity at risk. In addition, this may erode fundamental programming skills and deep technical understanding, making debugging and testing harder. Finally, training these large AI models requires massive computational resources, creating tension between advancing AI capabilities and environmental sustainability.

However, this environmental impact is not specific to AI-based systems. Modern software systems, especially data centres, consume enormous amounts of electricity. Software engineers must design more energy-efficient systems to reduce carbon emissions. Another example is that constant software upgrades often drive hardware obsolescence, contributing to e-waste. Developing software that can run efficiently on older hardware could help address this issue, but it won't solve it. As climate change accelerates, software engineering practices will need to incorporate sustainability principles throughout the development lifecycle.

Finally, the rapid pace of technological change creates constant demand for new skills the educational systems often struggle to provide. Communications between industry, the academic and public sectors need to be consolidated to develop programmes tailored at developing these skills.

Beyond the educational challenges, software engineering must also contend with the growing complexity of modern systems. As software continues to expand in scale and scope, managing ultra-large-scale systems with billions of lines of code across distributed environments poses unprecedented complexity challenges for testing, maintenance, and evolution. Additionally, emerging technologies such as quantum computing will require entirely new programming paradigms, security approaches, and ways of thinking about computation that current educational frameworks aren't prepared to address.

Perhaps most concerning are the ethical dimensions of increasingly autonomous software systems. As software makes more decisions affecting human lives – from healthcare resource allocation to autonomous vehicles to criminal justice

risk assessments – software engineers face complex ethical considerations regarding fairness, transparency, and accountability. These ethical challenges require not just technical solutions but interdisciplinary approaches that bring together computer scientists, ethicists, legal experts, and domain specialists to ensure software serves humanity's best interests.

5 **Based on your own knowledge of some of the application types discussed in Section 1.1.2, explain, with examples, why different application types require specialized software engineering techniques to support their design and development.**

Different application types demand specialised software engineering techniques because each category faces distinct challenges, constraints, and requirements that cannot be adequately addressed through a one-size-fits-all approach. The fundamental differences in operating environments, user expectations, safety implications, and technical constraints necessitate tailored development methodologies.

Embedded control systems exemplify this need for specialisation most clearly. Consider the software controlling anti-lock braking in a motor car versus a word processing application - the embedded system operates under severe constraints that fundamentally alter the development approach. The automotive software must respond to sensor inputs within microseconds, operate reliably in extreme temperatures, consume minimal power, and never fail catastrophically. These constraints demand specialised techniques such as real-time programming methodologies, extensive hardware-software co-design, formal verification methods, and exhaustive testing protocols that simulate thousands of driving scenarios. The development process must also account for the fact that once the software is burnt into ROM and installed in vehicles, updates become extraordinarily expensive, potentially requiring costly product recalls. Consequently, embedded systems engineering emphasises front-loaded verification and validation, with mathematical proofs of correctness often replacing the iterative development approaches suitable for other application types.

In stark contrast, interactive transaction-based applications such as e-commerce platforms or online banking systems require entirely different specialised techniques. These systems must gracefully handle thousands of concurrent users, maintain data consistency across distributed databases, and provide responsive user experiences despite complex business logic running behind the scenes. The development techniques here focus on scalability patterns, database transaction management, security protocols, and user experience optimisation. Unlike embedded systems where requirements are typically fixed early, web-based applications operate in rapidly changing business environments where requirements evolve continuously. This drives the adoption

of agile development methodologies, continuous integration and deployment pipelines, and extensive automated testing frameworks that can validate functionality across multiple browsers and devices. The development teams must also master specialised skills in web technologies, distributed systems architecture, and performance optimisation techniques that would be irrelevant for embedded systems developers.

Entertainment systems, particularly modern video games, demand yet another set of specialised techniques that bridge creative and technical disciplines. Game development requires close collaboration between programmers, artists, designers, and audio specialists, necessitating specialised project management approaches and development tools. The software must deliver consistent frame rates whilst rendering complex 3D graphics, managing artificial intelligence for multiple characters, and responding to real-time user input. This drives the adoption of specialised game engines, performance profiling tools, and iterative prototyping methodologies where playability testing begins early in development. Game development also embraces specialised techniques such as level editors, asset pipeline management, and platform-specific optimisation that would be unnecessary overhead in business applications. The tolerance for occasional bugs differs dramatically from safety-critical systems - a minor glitch in a game might be acceptable if it doesn't break core gameplay, whereas any anomaly in medical device software could prove fatal.

Systems of systems present perhaps the most complex specialisation requirements, as they must integrate multiple independent software systems that may have been developed by different organisations using different technologies and standards. The development of such systems requires specialised techniques in enterprise architecture, service-oriented design, and systems integration. The engineering challenges focus less on individual components and more on managing interfaces, ensuring interoperability, and coordinating evolution across multiple subsystems with different lifecycles and governance structures. These projects demand specialised project management techniques that can coordinate multiple teams, sophisticated configuration management approaches, and extensive integration testing strategies.

The batch processing systems used for tasks such as monthly billing or salary processing require specialised techniques optimised for throughput rather than responsiveness. Development focuses on data processing pipelines, error recovery mechanisms, and resource utilisation optimisation. The testing approaches emphasise data validation and processing accuracy over user interface concerns, whilst the deployment strategies must handle large-scale data migrations and ensure transactional integrity across potentially millions of records.

These diverse requirements demonstrate why software engineering cannot rely on universal methodologies. Each application type operates under different assumptions about user interaction patterns, performance requirements, safety implications, and operational constraints. The specialised techniques evolved in each domain represent accumulated wisdom about what works effectively within those specific contexts. Attempting to develop an embedded automotive control system using web development methodologies would likely result in software that cannot meet real-time constraints, whilst applying embedded systems' rigorous formal verification techniques to a social media application would prove unnecessarily expensive and slow. The art of software engineering lies in selecting and adapting techniques that match the specific challenges and constraints of each application domain.

6 Explain why the fundamental software engineering principles of process, dependability, requirements management and reuse are relevant to all types of software system.

The four fundamental software engineering principles of process, dependability, requirements management, and reuse are universally relevant because they address core challenges that transcend specific application domains or technologies. These principles tackle the inherent complexities of software development that exist regardless of whether you're building a mobile game, a medical device, or an enterprise system.

Process is essential across all software types because it provides the structured framework needed to transform abstract ideas into working solutions. Consider the contrast between developing an insulin pump control system and creating a digital learning environment - whilst the technical challenges differ dramatically, both require systematic approaches to move from initial concept through to deployment. The insulin pump demands rigorous validation processes due to its safety-critical nature, whilst the learning environment needs iterative processes to accommodate evolving educational requirements. Without established processes, even simple applications can spiral into chaos as teams struggle to coordinate efforts, manage dependencies, and maintain quality standards. The process acts rather like a well-planned journey - you need a route whether you're walking to the local shops or embarking on a cross-continental expedition.

Dependability transcends application boundaries because users must be able to trust software to behave correctly, regardless of its purpose. A weather monitoring system in the wilderness must reliably collect and transmit data despite harsh environmental conditions, just as a mental health patient information system must consistently protect sensitive data and remain available when clinicians need it. The consequences of unreliability vary - a game crash might frustrate a user temporarily, whilst a failure in the insulin pump could be life-threatening - but in all cases, dependability affects user confidence and

system value. Modern software systems are increasingly interconnected, meaning that failures can cascade across system boundaries, making dependability a shared responsibility across all software types.

Requirements management is universally critical because it bridges the gap between what stakeholders envision and what developers build. Whether we're developing a bespoke air traffic control system or a generic productivity application, success depends on accurately capturing, analysing, and managing what the system should accomplish. The mental health case study illustrates how complex legal and regulatory requirements must be carefully managed alongside functional needs, whilst the learning environment demonstrates how requirements must evolve to accommodate different user groups and educational contexts. Poor requirements management leads to the same problems across all domains: missed expectations, scope creep, budget overruns, and ultimately, systems that fail to deliver value.

Reuse provides universal benefits because it leverages existing solutions to common problems, reducing development time, costs, and risks across all application types. The digital learning environment exemplifies this principle by integrating existing services rather than building everything from scratch, whilst even the embedded insulin pump benefits from reusing proven algorithms and components. Whether you're developing stand-alone applications, web-based systems, or embedded controls, certain patterns and solutions recur - user authentication, data validation, communication protocols, and user interface elements. By embracing reuse, developers avoid reinventing the wheel and can focus their efforts on the unique aspects that deliver specific business value.

These principles work synergistically across all software domains because they address fundamental human and technical realities: the need for coordination and structure (process), the requirement for reliability and trust (dependability), the challenge of translating human needs into technical specifications (requirements management), and the economic imperative to build upon existing knowledge and solutions (reuse). Regardless of whether we're building safety-critical embedded systems or entertainment applications, these principles provide the foundation for professional software development that delivers value whilst managing complexity and risk.

7 Explain how the universal use of the web has changed software systems and software systems engineering.

The universal adoption of the web has fundamentally transformed software engineering from building isolated, standalone applications to creating interconnected, distributed systems. This architectural revolution moved us from desktop applications that operated like self-contained islands to complex client-server and multi-service architectures spanning multiple data centres and organisations. The shift required engineers to master entirely new concerns like

network latency, fault tolerance, data consistency across distributed components, and security threats such as cross-site scripting and distributed denial-of-service attacks that simply didn't exist in isolated desktop environments.

These architectural changes drove corresponding transformations in development practices and user expectations. The web's rapid pace and global accessibility necessitated agile methodologies, continuous integration, and frequent deployments that replaced the traditional cycle of extensive upfront planning followed by infrequent major releases. Users now expect software to be accessible anywhere, on any device, without installation, while providing real-time updates and seamless synchronisation. This has made user experience design integral to system architecture rather than an afterthought, while simultaneously requiring engineers to design for varying network conditions and device capabilities.

The web also enabled new deployment models and collaboration patterns that continue to reshape the field. Software-as-a-Service architectures allow centralised updates that immediately benefit all users, enabling sophisticated A/B testing and real-time monitoring but creating new availability challenges. Global teams can now collaborate through distributed version control and cloud-based development environments, accelerating open-source development but requiring new approaches to dependency management and code quality across distributed contributors. The need to serve millions of concurrent users has driven innovations in scaling, caching, and cloud computing, while new web-enabled business models like subscription services and advertising-supported platforms require fundamentally different technical capabilities than traditional licensed software.

Ultimately, the web transformed software engineering from creating discrete products to building continuously evolving, interconnected services that must operate reliably in a complex, constantly changing networked environment where technical, business, and user experience concerns are deeply intertwined.

8 Discuss whether professional engineers should be licensed in the same way as doctors or lawyers.

The question of whether professional engineers should be licensed like doctors or lawyers touches on fundamental issues of public safety, professional accountability, and the unique characteristics of engineering practice. This debate becomes particularly complex when we consider software engineering, which has grown rapidly without the traditional regulatory frameworks that govern other professions.

The strongest argument for engineering licensure centres on public safety and accountability. Just as we require doctors to be licensed because they make life-and-death decisions, engineers increasingly design systems that directly impact

public welfare. Consider that software controls everything from medical devices and autonomous vehicles to financial systems and power grids. When an engineer makes a critical error in designing flight control software or a bridge monitoring system, the consequences can be catastrophic. Licensing would establish clear standards of competence, create mechanisms for accountability when things go wrong, and provide a framework for ongoing professional development and ethical oversight.

Traditional engineering disciplines like civil and mechanical engineering already have well-established licensing systems through Professional Engineer (PE) certification. These systems require formal education, supervised experience, and demonstrated competency through rigorous examinations. They also establish clear ethical guidelines and provide mechanisms for disciplinary action when engineers violate professional standards. This model has generally served the public well in fields where engineering decisions have direct physical consequences.

However, software engineering presents unique challenges that complicate the licensing model. The pace of technological change in software far exceeds that in traditional engineering fields. Programming languages, frameworks, and methodologies evolve rapidly, making it difficult to establish stable, long-term competency standards. Unlike civil engineering, where the fundamental principles of structural mechanics remain relatively constant, software engineering involves constantly shifting technological foundations. A licensing system that takes years to update could quickly become obsolete and potentially stifle innovation.

The risk profiles between software engineering and traditional professions also differ significantly. While software errors can certainly cause harm, many software applications have relatively low stakes compared to medical procedures or legal decisions that can determine someone's freedom. A bug in a social media app, while potentially problematic, doesn't carry the same immediate life-or-death consequences as a surgical error or a criminal defence attorney's malpractice. This suggests that a one-size-fits-all licensing approach might be overly burdensome for large segments of the software industry.

Furthermore, the software industry has already developed alternative quality assurance mechanisms that may be more effective than traditional licensing. Code reviews, automated testing, continuous integration, open-source collaboration, and industry certifications provide ongoing quality control and knowledge sharing. The rapid iteration cycles common in software development allow for quick identification and correction of problems, unlike the more permanent nature of traditional engineering projects where errors are harder to fix after implementation.

There are also practical implementation challenges to consider. The software engineering workforce is global and highly mobile, making it difficult to establish consistent licensing standards across jurisdictions. Many successful software engineers are self-taught or have non-traditional educational backgrounds, which could be problematic under rigid licensing requirements that typically emphasise formal credentials.

A potential middle-ground approach might involve risk-based licensing, where engineers working on safety-critical systems like medical devices, autonomous vehicles, or infrastructure control systems would be required to obtain specialized licenses, while those working on lower-risk applications would remain unlicensed. This would provide public protection where it's most needed while avoiding unnecessary regulatory burden on routine software development.

Another alternative could be strengthening existing professional organisations and industry standards rather than implementing government licensing. Organisations like the ACM and IEEE could expand their role in setting ethical standards, providing continuing education, and offering professional certification programs that employers and clients could use to evaluate competency.

The question ultimately depends on whether we view engineering as fundamentally different from other professions in its relationship to public welfare, and whether the benefits of licensing would outweigh the potential costs in terms of innovation, accessibility, and practical implementation challenges.

9 For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.4, propose an appropriate example that illustrates that clause.

1. PUBLIC - Software engineers shall act consistently with the public interest. Example: A software engineer working on a mobile banking app discovers a security vulnerability that could allow unauthorised access to user accounts. Despite pressure from management to meet a tight release deadline, the engineer insists on delaying the launch until the vulnerability is fixed, prioritising public safety over commercial interests. This demonstrates putting the broader public good ahead of immediate business pressures.

2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest. Example: A developer is asked by their employer to implement a feature that would secretly collect additional user data beyond what's disclosed in the privacy policy. While this data could provide valuable business insights, the engineer refuses to implement the feature as designed and instead proposes a transparent, opt-in alternative that serves the employer's needs while respecting user rights and maintaining public trust.

3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. Example: A

software engineer working on medical device software insists on comprehensive unit testing, integration testing, and code reviews despite time pressures. They document their code thoroughly and follow established coding standards, ensuring that the life-critical software meets the highest quality standards. When a colleague suggests skipping some tests to save time, they explain why this would compromise the product's reliability and patient safety.

4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment. Example: During a project review, a senior developer is asked to estimate whether a complex feature can be completed in two weeks. Despite knowing that management wants to hear "yes" to meet marketing commitments, the engineer provides an honest assessment that the feature will require at least four weeks to implement properly, explaining the technical complexities involved and the risks of rushing the implementation.

5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. Example: A development manager refuses to implement "crunch time" practices that would require developers to work excessive overtime for weeks. Instead, they advocate for realistic project timelines, ensure their team has adequate resources for proper testing and code reviews, and create an environment where developers feel safe reporting potential issues without fear of retaliation or blame.

6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. Example: A software engineer discovers that a competitor's product contains serious security flaws that could harm users. Rather than exploiting this knowledge for competitive advantage, they responsibly disclose the vulnerabilities through appropriate channels. They also participate in professional conferences, contribute to open-source projects, and mentor junior developers, helping to elevate the standards and reputation of the entire profession.

7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues. Example: When a junior developer makes a mistake that causes a production issue, a senior engineer focuses on helping them understand what went wrong and how to prevent similar issues in the future, rather than blaming them publicly. They also ensure that credit for successful projects is shared appropriately among team members and advocate for colleagues who deserve recognition or promotion opportunities.

8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. Example: A software engineer regularly attends workshops and conferences to stay current with new technologies and best practices. They also take courses on software ethics and security, participate in code review

processes to learn from others, and share their knowledge through blog posts or internal presentations. When they encounter ethical dilemmas, they seek guidance from professional organisations and discuss these challenges with colleagues to promote ethical awareness throughout their workplace.

10 To help counter terrorism, many countries are planning or have developed computer systems that track large numbers of their citizens and their actions. Clearly this has privacy implications. Discuss the ethics of working on the development of this type of system.

The core ethical tension revolves around the privacy versus security trade-off. The fundamental question is whether mass surveillance systems represent a proportionate response to terrorism threats. Let us consider the analogy of a neighbourhood watch program versus having cameras in every home - where do we draw the line between collective security and individual privacy? This tension becomes even more complex when we consider that different societies may legitimately reach different conclusions based on their values and experiences.

For example, Western liberal democracies typically emphasise individual rights, judicial oversight, and democratic accountability, with the EU's GDPR exemplifying this rights-focused approach. In contrast, authoritarian systems may prioritise collective security and state stability over individual privacy rights. Post-conflict societies may have different risk tolerances based on recent experiences with violence, while established democracies might view such systems as fundamentally incompatible with their constitutional principles.

For developers specifically, there are unique ethical considerations around professional responsibility. This includes understanding the full scope and potential uses of systems being built, whether the developer has meaningful input into safeguards and oversight mechanisms, and their ability to withdraw if the system is misused beyond its original intent. The technical design itself raises ethical questions about implementing privacy-by-design principles, building strong access controls and audit trails, and designing for transparency and accountability where legally permissible.

Rather than approaching this as a binary choice, we could perhaps consider examining graduated responses and safeguards. Key questions include whether mass surveillance is truly necessary or if targeted approaches would suffice, what legal and institutional oversight exists, whether citizens are aware of the system's scope, whether there's evidence these systems actually prevent terrorism, and whether the system can be scaled back if threats diminish. These considerations might lead to middle-ground approaches such as time-limited authorisations requiring regular renewal, strong judicial oversight requirements, technical measures that preserve some degree of anonymity, and clear legal frameworks defining acceptable use and prohibiting mission creep.

Chapter 2

1. **Suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems. Explain your answer according to the type of system being developed:**
    a. **A system to control antilock braking in a car**
    b. **A virtual reality system to support software maintenance**
    c. **A university accounting system that replaces an existing system**
    d. **An interactive travel planning system that helps users plan journeys with the lowest environmental impact**

For a system to control antilock braking in a car, the waterfall model would be most appropriate. This safety-critical embedded system requires extensive analysis, complete specification, and thorough verification before implementation begins. The hardware constraints mean that changes after deployment are extremely expensive, and regulatory requirements demand comprehensive documentation and testing. The waterfall model's emphasis on sequential phases with formal signoffs ensures all safety requirements are properly analysed and validated before proceeding to implementation.

A virtual reality system to support software maintenance would benefit from incremental development. VR interfaces are inherently complex, and user requirements are likely to evolve significantly as users experience the system. The innovative nature of VR applications means that complete requirements cannot be specified in advance, and rapid prototyping with user feedback is essential to develop effective interfaces. Incremental development allows for experimentation and refinement of both the VR interface concepts and the underlying maintenance support functionality.

For a university accounting system replacing an existing system, the integration and configuration model would be most suitable. University accounting processes are well-established and standardised, with proven commercial systems available. Rather than developing from scratch, configuring existing enterprise resource planning systems or specialised university management systems would reduce cost, risk, and development time whilst providing proven functionality that meets regulatory and audit requirements.

An interactive travel planning system focusing on environmental impact would be best developed using incremental development. User preferences for environmental factors, journey types, and interface design will evolve as users interact with the system. The system needs to incorporate complex algorithms for environmental impact calculation and route optimisation, which require iterative refinement based on real usage patterns. Market pressure for rapid

delivery also favours incremental approaches that can deliver basic functionality quickly whilst building more sophisticated features over time.