Software Engineering Principles
Understandability

Implemented on (date)	20/05/2024
Approved by (name & role)	Head of Software Practices (Fahad Anwar) In consultation with TAG (Technical Advisory Group).
Last review on (date)	16/05/2024
Reviewed by	TAG (Technical Advisory Group)
Next review due on (date)	16/05/2025
Principle owner (name)	Head of Software Practices In consultation with Technical Advisory Group (TAG) representing Software Engineers, Cloud Division, TISS and Security Division.
Principle owner (division)	Digital Services and Technology (DST)
Main point of contact (name)	Fahad Anwar Software Engineering Head of Practice
Status	Approved

Principle Review Record

This Review Record is to be completed on each time a review is conducted. Its purpose is to maintain a record of reviews, recording who conducted the review, the date of the review and the outcome of the review (fit for purpose, amendment required, principle no longer required, etc).

This principle is to be reviewed annually.

Review No	Review Conducted By	Review Date	Review Outcome
01			

Amendment Details

Date	Amendment Summary	Amended by	Version

RASCI (For detail please – RASCI Information document)

Responsible	G6 Program Managers through Technical Leads, G7 and SEO's
Accountable	Head of Software Practices
Supportive	Head of Cloud Functions (Amazon, GCP, Azure) SAIM SIRA Software Engineering Community of Practice (SE-CoP)
Consulted	Technical Advisory Group (TAG) representing Software Engineers, Cloud Division, TISS and Security Division.
Informed	Senior Leadership Team Software Engineering Community

	SAIM SIRA Design Authority Chair

Understandability

Each codebase must be understandable and easy to change by new developers with minimal experience of the application.

Rationale

Almost all software continues to require changes through its life, whether to fix bugs, add new features or address security vulnerabilities. Sooner or later, most software ends up being maintained by people other than those who created it. In order to be able to change the software a developer needs to be able to understand it — or at least the part of it that they need to change.

Systems that are considered too difficult, too expensive or too risky to change will either become a limiting factor on business change or eventually lead to a Big Rewrite, which can cost millions and take several years.

Instead, we should ensure that applications' codebases, as well as the systems that result from the interactions between them, are easily maintainable *and kept that way*, so that they are able to change with the business requirements. Even the process of replacing or decommissioning a system will benefit greatly from its current workings being understandable.

Implications

- Software must be created with a clear set of tests which help the reader understand the application, and tests can be run easily (ideally with single command).
- Appropriate levels of documentation will need to be created and updated, with a preference for detailed descriptions of functional behaviour to be implemented in code as tests. This includes maintaining a README/CONTRIBUTE describing the application and documenting the commands required to build, test and run it.
- Different audiences may require different artefacts for effective maintenance, change planning and risk management. This may include: the code, the commit history, JIRA tickets, cross-functional stories, architectural descriptions, runbooks, ADR (Architecture Decision Records) or formal documentation.
- Stale documentation is worse than no documentation, so investment will be required to ensure it is up to date. Maintained reference documentation and transient delivery documentation should be kept separate, so anyone can quickly understand what they can trust. Therefore, regular scheduled checks for documentation accuracy is required to ensure it remains reliable and up-to-date.
- Knowledge sharing can be split into two parts, the "WHATs" and the "WHYs". The WHATs should be evident from the code, and focused reference documentation, but the WHYs should be carefully documented as decisions are made. Decision records, code comments and/or appropriately named tests can all be useful in this case. Handover of the WHYs should be done to all levels, not just to developers.
- Each codebase should have a published set of code style guidelines, preferably importable into your IDE or text editor. At a minimum: encoding, line breaks and

tabbing standards should be defined, ideally using a non-IDE-specific (or widely-supported) tool such as editorconfig.
- Developers should refactor code and tests when needed, reassured by the presence of the rapid feedback they will receive if they break something.
- An application should be consistent in applying a programming style throughout - such as procedural, functional or object-oriented. Many languages encourage a particular style, in which case this should be adopted in order to play to its strengths.

Questions to be considered

- Are your services independently deployable and testable?
- Are you building a distributed monolith[1]? This is mostly undesirable and can result from a tightly couple microservices architecture.
- Are your services sliced based on technical nature? It is mostly desirable for services to be sliced based on your business domain.
- Are you building nano-services? The nano-services can add lot of complexity and network overhead.
- Are your services independent or are they integrating on the database layer? Independent services are more desirable.
- Are you documenting architecture decisions?
- Are you using consistent styling supported by appropriate tools?

Motivation for the principles

- https://www.gov.uk/guidance/the-technology-code-of-practice
- https://engineering-principles.jlp.engineering/
- https://github.com/otto-de/tech_manifest
- https://www.gov.uk/government/publications/security-policy-framework/hmg-security-policy-framework

[1] A distributed monolith is the result of splitting a monolith into multiple services, that are heavily dependent on each other, without adopting the patterns needed for distributed systems.