Table of Contents

John Lewis IT Software Engineering Principles

Through our combined experience of building and releasing software we have discovered and come to value the following principles. These principles are not hard and fast rules but rather some things that we apply and use to guide us on a daily basis. These principles could also be used in a 'discovery phase' when selecting a new product to purchase.

The basic anatomy of a principle is:

- Name - This is the essence of the rule
- Statement - Unambiguously communicate the fundamental rule
- Rationale - The business benefits of adhering to the principle
- Implications - The requirements both for the business and IT for carrying out the principle

The concept of creating principles in this manner has been borrowed from TOGAF's Architectural Principles

Printable version

A PDF version is available.

Invitation to contribute

If you're involved in the software engineering process at John Lewis, then we want your input! The principles are maintained in a GitLab project tech-council/engineering-principles.

For correction of typos, fixing broken links, or anything similarly unlikely to raise any debate, please fork the project, make the change and raise a merge request.

For more significant adjustments to an existing principle, do the same, but perhaps anticipate some discussion on the merge request.

For suggestions of new principles, fundamental challenges to existing ones, please raise an issue

Build Differentiators

If functionality is a differentiator for the Partnership then we should build over buy and/or customise. Also check if there is an existing system that can be used rather than building a new one

Rationale

Customisation of COTs packages can be costly in comparison to building equivalent functionality. It also has a longer standing impact of the future pace of change maintaining and developing on top of frequently complex customisations that requires specialist knowledge.

Implications

- Extract what we can from existing COTs packages
- New functional differentiators should be developed outside of COTs packages.
- Use COTS for what they are good for and compose systems around them.
- Legacy systems may need to have a wrapper layer so it can be strangled and replaced in the future

Design for Pace of Change

For components that change often, favour designing for pace of change over re-use

Rationale

For systems that change often, designing for re-use can inhibit rapid change by creating dependencies and bottlenecks. If a system changes less frequently it's maintenance cost becomes more important and designing for reuse can help reduce this.

Code that is designed for re-use is generally harder to use and more expensive to create than code designed for a single purpose[1].

Implications

- Focus on re-use as it emerges through evolution
- Be sceptical of components that look similar but aren't the same
- Domain specific component sharing is an anti-pattern; code re-use between bounded contexts is a hazard to be avoided[2].
- Systems should expose common business functions through APIs.
- If re-used code becomes a bottleneck remove the coupling by forking or duplicating the code[1].
- Features such as monitoring, logging, service discovery, authentication/authorisation often need to be implemented consistently across all services/applications. Use Service Templates to define which libraries should be used by all services[1].
- Shared components require robust tests, sufficient documentation and ongoing maintenance; a website to host generated docs may be useful.

> [1]. [Building Evolutionary Architectures: Support Constant Change by Neal Ford, Patrick Kua, and Rebecca Parsons] (http://shop.oreilly.com/product/0636920080237.do) ↩

> [2]. [Domain Driven Design by Eric Evans] (http://dddcommunity.org/book/evans_2003/) ↩

Examples

- A good example of code reuse is logging functionality or a service that provides business functionality, for example a VAT calculator

Evolutionary Systems

Systems and architectures should be designed and built to evolve independently from other systems.

Rationale

At this point in time we know less than what we'll know in 6 months. Our software should be able to evolve as we learn more. It should evolve both at a code level and also at a service level.

Implications

- Systems and processes will need to be exposed through clearly defined, open APIs so that the underlying implementation can change and evolve without impacting the wider system. (Note: this does not mean that a component or system's *internal* APIs necessarily need to be approached in the same way.)
- This will rely heavily on the principle of Maintainability to validate that behaviour hasn't mistakenly changed or broken.
- Consider patterns such as TolerantReader and the Principle of Robustness
- These patterns will give us more robust systems and allow systems to evolve independently of each other. One system can change and it will not take down others.
- Sometimes gradual/incremental change is insufficient as requirements change, so consider a Sacrificial Architecture as a pattern. Don't be afraid to throw away and re-write, as this is often faster and will lead to better quality. If we have kept things Small and Simple this should not turn into a painful Big Rewrite.

Open Integration Standards

Components must be accessible through open, non-proprietary standards (with a preference for HTTP & in-particular REST)

Rationale

Proprietary standards and applications inhibit open integration and extensibility. In order to create a platform for innovation we need to embrace open communication standards. Embracing the best practices of the web gives not only the desired openness but also the scalability.

Implications

- Products that require complex integration technologies must be wrapped with an isolation layer or, ultimately, replaced with something that does not rely on closed communication protocols.
- Services will be required to provide the ability to discover all capabilities that they offer through their API
- Standards that are derived from working software and not just working groups are preferred. Battle tested standards used by several organisations tend to be more useful than over-engineerd hot air. Most succesfull standards tend to be the ones that are adopted by Open Source communities.

Scale Horizontally

System components should be horizontally scalable where possible.

Rationale

It is cheaper to scale services by adding inexpensive resources like commodity servers or database nodes rather than buying larger and larger pieces of hardware which at a given point are unable to scale further. By embracing horizontal scalability early and designing our systems to work in this manner we eliminate the cost of changing at a later date.

Implications

- State cannot be stored in-memory but rather must be persisted to the client or a shared datastore.
- It is not always possible to scale horizontally eg for traditional RDBMS increased performance is achieved through better hardware or specifications
- Services should strive to be idempotent

Small and Simple

Build systems small and simple

Rationale

Monolithic applications can be successful, but increasingly people are feeling frustrations with them - especially as more applications are being deployed to the cloud. Change cycles are tied together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource. Small and simple allows us to scale more efficiently.

Building small and simple add also makes systems easier to reason about, things that are easier to reason about with less code tend to generate fewer bugs and complex production issues.

Implications

- Create a suite of small systems
- Build systems around business capabilities
- Services are independently deployable and independently testable
- Keep them light-weight
- Reduce burden of more smaller systems with automation
- Designed for failure
- Published Interfaces
- Small applications need to be appropriately coupled

Examples

- Financial processing processing (COFP) for OCCO is self contained in one service so complex business logic related to that is tightly coupled and separate from other order processing tasks.
- Wishlist service for jl.com is a separate service.

Smarts in the Nodes not the Network

Sometimes referred to a "Smart nodes and dumb pipes", meaning that Systems aim to be de-coupled and cohesive as possible, and not centrally choreographed.

Rationale

Composing larger systems around simpler and smaller systems also means needing to own their communication with other systems as well. To allow systems to evolve over time and keep up with the businesses desired pace of change, having them own their communication allows them to change quickly rather than negotiating a change to the pipes that connect them.

Implications

- Prefer open protocols (E.g. HTTP, AMQP)
- Services acting like unix-filters, accepting requests, applying logic and returning a response.

Examples

- A bad example is VAT calculation is done in place-order service within ESB instead of in a separate service (anti-pattern). Makes the calculation less transparent of how VAT is calculated.
- Good example is the use of APIs with BFF

Cloud Native

Build systems that are native to cloud environments.

Rationale

To support a rapid pace of change, scaling and simplification to operation of many smaller systems.

Implications

- Prefer open and cloud tooling for configuration and deployment
- Considering auto-scaling in the design of applications
- Build tolerant of unique cloud failure conditions
- Requires in-house capability to deploy using cloud native infrastructure

Examples

- Space data capture and Beauty CRM are examples of cloud native. Not completely painless because tools used to provision cloud infrastructure would need work but Kubernetes is available on other cloud platforms and containerisation enables consistency across whole path to live. From nothing to live for Beauty CRM was days rather than > 6 weeks just to get an internal test server. DISCO - elastic search is another example.
- 1JL was designed as cloud native as reasonably possible at the time but are constrained by some dependencies of internal network.
- Bad example .. Large packages such as ATG are not cloud ready/enabled which restricts hosting option
- Good reading - https://12factor.net

Production Ready

Systems are built to be production ready, tolerant of failure, self healing and be able to be monitored.

Rationale

Adopting the building of more smaller systems, and utilising cloud environments means handling failure is more important than ever. If a system is self healing, it must also raise alerts which provide enough information to investigate the problem.

Implications

- Release It will be used as the starting point for what defines Production Ready
- The system should have sufficient monitoring and understand and report on its health and the health of systems that it depends upon in a consistent way.
- Production monitoring should be available to all!
- Teams and business owners will need to work together to identify relevant business and technical KPIs for their product.
- Dashboard and alerting will need to be implemented for the most important KPIs
- Not all risks to production readiness are analysable in advance so (in addition to checking KPIs) exploratory testing should be used to expose new information about software behaviour
- Applications should be built with a diversity of stakeholders in mind. Operability and supportability are important in most contexts, but see [https://en.wikipedia.org/wiki/List_of_system_quality_attributes] for a list of other software 'illities' that may need particular consideration in your context.

Examples

- The 1JL Category Service, Elastic search service and apps built on the digital platform have been built with monitoring and self healing in mind using Grafana and Prometheus

Collaborate Early and Often

Different disciplines from business, architecture, delivery, operations must work together from idea to release.

Rationale

Functions working in isolation are unable to account for all scenarios when delivering software. Only with all disciplines working together with no barriers between them can the delivery process run smoothly. This will reduce communications overhead, decrease lead time and increase the delivery of business value.

Implications

- Cross-functional teams will require representatives from all functions with the common purpose of delivering valuable, working software to customers.
- We will need to establish a low friction mechanism that enables interested parties to contribute to the delivery of solutions.
- Teams will need to form around products rather than around roles and functions this may impact the structure of the organisation.

Keep pace with Technological Change

Use the latest, most appropriate, tools and technologies that solve the business problem.

Rationale

Technology advances at an ever increasing rate. Pragmatically adopting the latest tools and technologies will allow us to provide innovative new products and services to our customers and business and compete more effectively with our competitors.

Implications

- The principle of Evolutionary Systems is essential in order for us to update and upgrade systems without impacting the broader ecosystem.
- The best people to choose tools and technologies are those that are closest to the problem being solved they will need a framework / guidelines that outlines how to adopt new technologies consistently.
- The architecture function will help align technologies and identify synergies where appropriate.
- Regular patching, if possible, will help future software upgrades and maintainability and keep systems more secure.
- Technology selection conflicts will need to be resolved through a collaborative framework which is open to advancing technical excellence.
- We have created a Technology Radar that identifies and monitors the technologies that we believe will be relevant to our business. New tools and technologies should be filtered through this process
- There should be a community of interest around software engineering to share good practice and technology news.

Model the Business Domain

Terms, concepts and capabilities of the business should be reflected in the way we write, structure and deploy our code and systems.

Rationale

Using a common language throughout the team to describe business concepts and processes allows for a common understanding. Misunderstandings that lead to defects are easier to identify because the language that is used allows non-technical stakeholders to understand and correct it.

Structuring component systems to reflect and influence the organisational structure will allow for a more agile and responsive delivery as teams and systems are more closely aligned to the customer value that they provide.

Implications

- We will need to take advantage of Conway's Law (where software reflects the organisation that built it) to validate and influence both the systems we write and how we structure the organisation.
- A system should have a product/business owner identified so they can encourage and nuture a sense of ownership for future change and support. It will become their 'problem' if it breaks or requires changes to be made
- Where packages have their own model and terminology, we may need to create abstraction layers to map to the business model

Automate Repetitive Tasks

Repetitive tasks which occur on a frequent basis must be automated.

Rationale

Repetitive tasks such as deployment and testing that are manual are slow and prone to error. These tasks can be performed quickly, reliably and consistently by computers and should, therefore, be automated.

Implications

- Automation requires an upfront and on-going investment that rapidly pays off but must be balanced with delivering business value
- Software and processes that are difficult to automate will need to be changed or replaced if, they require too much effort or are not amenable to being automated.
- The process of setting up a development machine in order to write code must be automated.
- Automation scripts must be treated as first class citizens and conform to all these engineering principles.
- Automation scripts help document your system and how it is put together

Examples

- AMO - automated reports
- Clarity reminders
- Abinitio
- Build processes - running tests.

Continuous Delivery

Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time. Systems will be structured to enable fast, automated feedback on quality and correctness.

Rationale

Continuous Integration usually refers to integrating, building, and testing code within the development environment. Continuous Delivery builds on this, dealing with the final stages required for production deployment. This is done to reduce deployment risk, show believable progress, and reduce the time to get feedback from real users.

Failures identified very soon after the cause are easiest to fix; details of the change will still be fresh in the engineer's mind. Slow-running tests will tend be skipped or run less frequently during development.

Implications

- We will keep software deployable throughout its lifecycle and prioritize keeping the software deployable over working on new features through constant investment.
- Fast feedback is required both on an engineer's machine for a specific component for production readiness, and in context with other components/applications as part of a continuous delivery pipeline.
- Delivery approach should be informed by risk and risk appetite in order to focus mitigation approaches. Risk can be mitigated through a combination of automated checks, deployment and release processes and exploratory testing.
- We can perform push-button deployments of any version of the software to any environment on demand
- We use trunk based development, mixing in branching by abstrations and/or feature branching where appropriate.
- We use a single trunk based CI build as the begining of our pipline.
- Applications and systems should be broken down into smaller components, each of which can be built and tested quickly.
- Applications should be capable of being built via a single command where possible.

Examples

- The Bermondsey project and John Lewis Ventures are very close to a full automated release. 1jl and Apigee have a couple of manual gates.
- Amazon / Netflix are at the expert level.
- Team Disco mitigate risk will a fully automated pipeline (build, test and deploy), they alert in production when metrics deviate from their expectations. This allows for easier rollbacks.

Consistent Environments

To have environments with homogenous application configuration, software, operating system, infrastructure and data where appropriate

Rationale

Less time will be taken to investigate issues caused by environmental discrepancies, this will allow for faster delivery of business requirements and give higher confidence in what we are delivering is correct.

Implications

- Configuration will need to be enshrined in version-controlled code
- Software packages that are not amenable to being consistently configured will be less desirable
- Software that is free or provides developer licenses will be preferable
- We will need to invest heavily in making data available and consistent across environments
  - Simple way of cleansing and scaling production data?
- Creating (and destroying) new environments will need to be as simple and as fast as possible and will rely heavily on the principle of Automating Repetitive Tasks (this will also require investment)
- We will require consistent configuration of infrastructure (including Networks, Firewalls and Servers)

Examples

- 1JL is a good example of this. A system really needs to have every instance on the cloud, be highly automated and have infrastructure as code.
- Bad examples of practices are changing things on the fly and not integrating with all the systems in every environment.

Maintainability

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

Rationale

Systems become unmaintainable through size, complexity and lack of up to date documentation. By keeping our components small and well tested we reduce complexity and increase maintainability. By combining these small, well understood components we are able to create large, complex yet maintainable systems

Implications

- Software must be created with a comprehensive suite of tests that can be run from the command line with a single command.
- Appropriate levels of documentation will need to be created and updated, with a preference for it to be enshrined in code as tests.
- Different audiences may require different artefacts for effective maintenance. This may include, the code, the commit history, JIRA or formal documentation.
- Stale documentation is worse than no documentation, so therefore investment will be required to ensure it is up to date.
- Knowledge transfer can be split into two parts, the 'WHATS' and the 'WHYS'. The WHATS should be evident from the code but the WHYS should be carefully documented. Code comments and/or appropriately named tests can be useful in this case. Handover of the WHYS should be done to all levels, not just to developers.
- README's should be created describing the application and documenting the command required to build, test and run the application
- Each codebase should have a published set of code style guidelines, preferably importable into your IDE or text editor. At a minimum, encoding, linebreaks and tabbing standards should be defined (take a look at editorconfig).
- Developers should refactor code and tests when needed, reassured by the presence of the rapid feedback they will receive if they break something

Examples

- Clean Code will be used as the starting point for what defines 'Clean'
- Small systems help a lot here.
- Keeping documentation with the code. Tests, swaggerfox. Examples are Payment, CategoryService
- Although these examples are from a particular context, they could easily be adapted for recording any decision or system info / rationale
- Lightweight Architecture Decision Records
- Decision record template / advice being trialled
- OCCO Architecture "agile documentation principles" (see attachment). Idea is to produce wiki reference docs that are always current - accessible to everyone for reads, but more careful with trust and quality checks for updates.
- OCCO also logs key decisions / rationales (business and IT) in JIRA with specific ticket types. Quality is variable but has proven its value many times over the last few years.

Performance Importance

System will be designed and delivered with a constant focus on the trend in performance characteristics.

Rationale

Without a continued focus on the performance impact of changes to systems we risk impact to both user experiences and more seriously, outages leading to impact of revenue.

Implications

- Every system should incorporate analysis of trend in the change of performance after every commit.
- Everyone needs awareness of how systems have been engineered meet it's performance criteria, as avoid introducing breaking changes.
- Some performance patterns:
  - Caching
  - Sharding
  - Scaling horizontally
  - Eventual consistency

Examples

- WebPIM has some JMeter examples

Release Early and Often

Prefer early and frequent releases of big functionality pieces to Big Bang releases.

Rationale

Early and frequent releases provides feedback earlier, code level integration is easier, course corrections can be made sooner and provides business value sooner. Problems are easier to identify and resolve in smaller changesets.

Implications

- Requires the automation of repetitive tasks.
- Encourages splitting big project into a set of smaller features.
- Requires switches to disable incomplete parts of functionality or to make a rollback.
- Requires regular assessment of feature flags and removal of the obsolete ones.
- No long-term development branches (and merges) needed.
- Requires an automated, version controlled, configuration management strategy
- This principle applies to all environments in the path to live, not just the production environment.

Examples

- Coming up with a very pure MVP is important a good example is Made To Measure Roller Blinds, where functionality was added slowly over a few weeks
- 1jl wishlists pure MVP
- OCCO - backend code in production, international 1man 2man before the site was ready
- A bad example was Payment where the code was complete way before it was needed

Security First

Systems will be designed and maintained with the assumption that they *will* be attacked and possibly compromised by **anybody**.

Rationale

The cost of handling a security breach[1] is significantly greater than the cost of hardening the system in the first place. If a breach does occur, we can minimise the scale and cost by detecting it as soon as possible, and having a well-planned response enables a fast time to close the breach and assess its impact.

By focussing on security from the beginning we mitigate not only the financial impact but also the reputational impact that would be incurred in the event of a breach.

Implications

- Teams should be aware of data privacy implications including policy and legal compliance
- Teams understand the security risks and model threat vectors
- Data will need to be secured both in transit and at rest.
- People at all points of the engineering process need strong awareness of security techniques and principles, and should apply these throughout the lifecycle of the software
- We will use the OWASP guidelines for software development as starting point to help people understand potential attack vectors and mitigation techniques.
- A framework is needed to classify systems and their data to determine the appropriate level of inspection and audit; this level may change as the system evolves. *link required - does anyone know if this exists?*
- Automated vulnerability analysis tools can help identify vulnerabilities but are not a replacement for a Security-First approach
- All HTTP communication should be done over SSL
- The system should be able to detect attacks
- Teams should have a clear strategy in place to respond to attacks and data breaches

---

[1]. A 2015 survey found the average security breach costs a large organisation £1.46 - £3.14 million (source). ↩