How To Manage Coupling

Introduction

The ability to change software is the defining characteristic of its quality. If you can't change the code, the only way to get the rest right is by hoping to be perfect first time.

Coupling is what decides how easy that change is going to be, and the cost of change is dominated by how much things are tangled together. Manage coupling well and a system stays adaptable, fast to evolve and safe to release. Manage it badly and changes ripple across your system, tests break, teams wait on teams.

Unmanaged Coupling Is The Enemy

If there were no coupling, the parts of our system could not talk to each other. We need coupling. The discipline is in choosing which coupling we accept and how we manage it.

A useful rule: strong coupling is acceptable when it is stable. SQL barely changes from year to year, so depending on it, may be a reasonable choice. Depending on this week's version of our application schema is not.

Problems arise when we have coupling that is unstable, unintended or invisible.

The Five Types of Coupling

Michael Nygard identifies five different types of coupling:

Operational	The consumer cannot run without the provider. An application that cannot start without its database is operationally coupled to that database. This kind of coupling is sometimes unavoidable but it bites hard when many systems share one provider. A flaky shared service brings down everything that depends on it.
Developmental	Two components have to be changed together. A modification in one forces coordinated modifications in the other. This is the coupling that shared code creates. When two services share a function, they become developmentally coupled through that common connection,, neither can evolve without the other also changing in-step.
Semantic	Two components share concepts – what a "customer" means, what an "order" contains. Even with no code in common, they must agree on meaning.
Functional	Different parts of the system address similar problems in different ways – two slightly different implementations of "calculate discount", two competing notions of "valid email". The system has more than one answer to the same question.
Incidental	Coupling that exists for no good reason at all. e.g. a module reaches across the system to grab a value it should not know about, rather than accepting that value as a parameter of some kind.

NB: Event-driven systems deliberately trade away operational and developmental coupling and concentrate the remaining coupling in the shape of messages.

How Coupling Shows Up

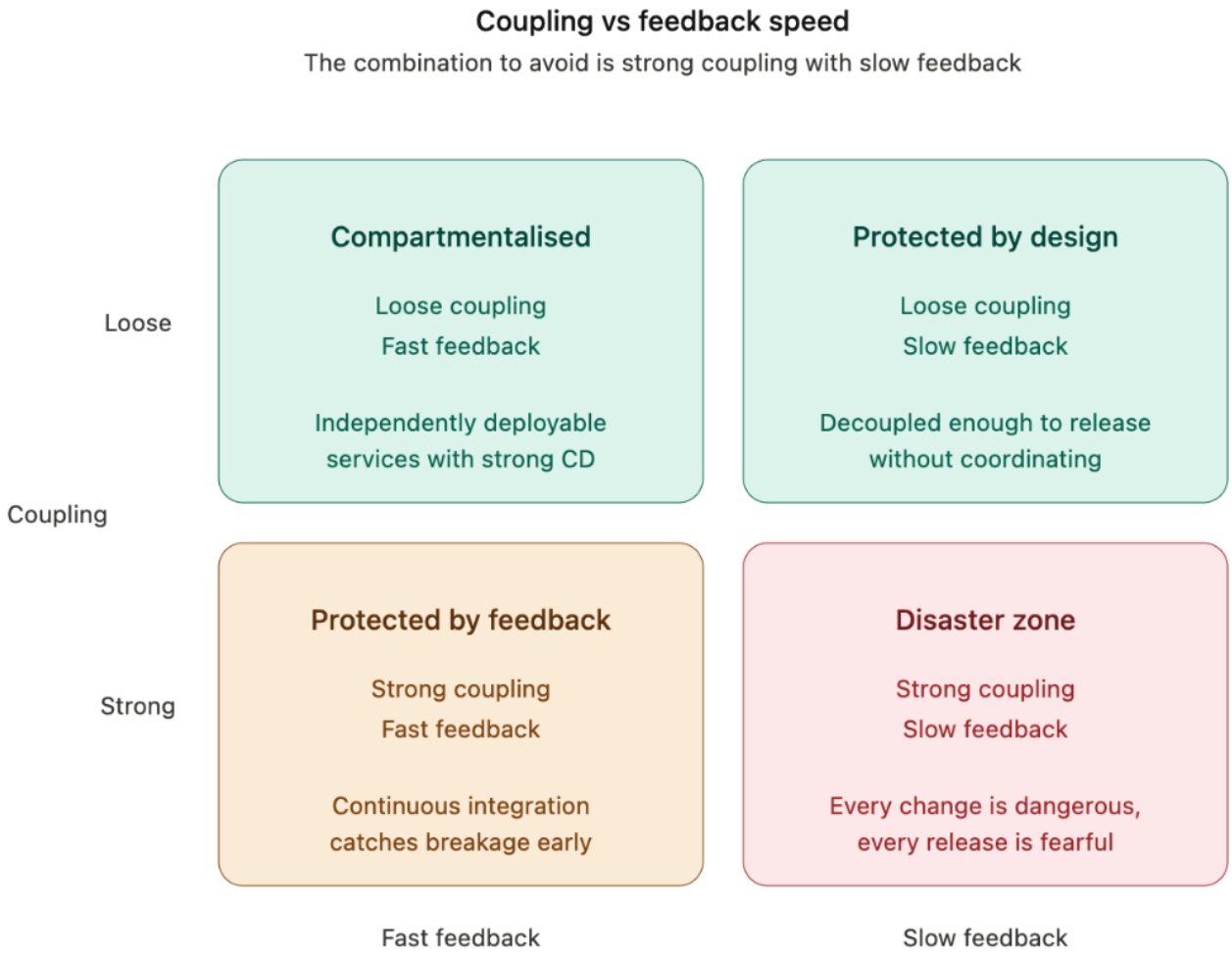
Coupling shows up in:

- Slow builds. Build times are a physical measurement of coupling. Everything-depends-on-everything architectures build slowly.
- Complex test setup. If a test needs paragraphs of setup before you can exercise the code, the design is telling you something. Hard-to-write tests are a signal of over-coupling.
- Brittle tests. Tests that break when an internal class is renamed, or when a private collaborator changes, are coupled to implementation rather than behaviour.
- Teams blocked on each other. Tightly coupled teams move at the pace of the slowest team. If your work cannot progress without depending on someone outside your team doing something, whatever the nature of the "something", the system is most likely too coupled.
- Lockstep platform releases. Shared platform code that forces every dependent service to take a new version, whether they want it or not, is one of the worst forms of coupling.
- Long parameter lists. A method with eight parameters is doing too much, knows too much, or depends on too much. Get nervous above four; refuse above six.
- Slow release feedback. Anything that takes more than a day to confirm a change is safe is a feedback problem caused by coupling somewhere.

How To Reduce Coupling

- Hide Information Behind APIs: Services should keep secrets. Public APIs exist to give you room to change implementation without breaking consumers.
- Use Ports and Adapters at Boundaries: Treat boundaries between modules, teams and services as special places that need careful guarding. Translate inputs from other teams at the edge, and treat them as something that can break your system. Validate inputs!
- Announce, Don't Command: Event-driven design inverts the usual command-style of programming. Rather than calling another component to do something, announce that something has happened and let listeners react.
- Be Parsimonious Consuming, Generous Producing: When you read from another system, take only what you need: every field you consume is a coupling you have accepted. When you produce a message, include the full story: adding new fields later is easy, removing them is a breaking change.
- Prefer Tagged Data Formats: JSON and XML couple consumers to tag names, not byte positions. That lets producers add or reorder fields without breaking consumers. Fixed binary formats couple everyone to layout, and layout is brittle.
- Use TDD as a Design Forcing Function: If a test is hard to write, do not power through, listen to the signal. Hard-to-write tests almost always mean the code under test depends on too much. Refactor the design until the test is easy to write.
- Apply the Implementation-Swap Test: Ask, if I threw away this implementation and rewrote it from scratch, would my tests still make sense? If yes, the tests are coupled to behaviour. If not, they are coupled to implementation.
- Tighten What's Stable, Loosen What's Uncertain: Couple tightly to things you understand well and that change slowly. Couple loosely to things you are still figuring out.

The Disaster Zone



There are two primary tools for managing coupling: design and speed of feedback. You can cope with a more coupled system if your feedback is fast enough, continuous integration is essentially a tool for safely working on highly coupled code. You can cope with slow feedback if your system is sufficiently decoupled, independently deployable services do not need each other's permission to release.

The combination that does not work is strongly coupled systems with slow feedback. That is the disaster zone. Changes are dangerous, releases are fearful and nobody is sure what will break next.

- New systems should start with continuous integration, even if the long-term plan is to decouple into separate services. Until you understand the right abstractions, CI is your defence against coupling you have not yet learned to see.
- Treating a highly coupled system like it is a set of microservices is disastrous. If components share state, schemas, or concepts, deploying them independently will hurt you.
- Two valid strategies exist: compartmentalise by design, or protect through fast feedback. Pick deliberately. Do not drift into the disaster zone by accident.

MSE Enables Decoupling

We can use the principles of Modern Software Engineering – modularity, cohesion, separation of concerns, abstraction – to address coupling:

- Small modules decouple parts of the system from each other.
- High cohesion decouples unrelated concerns by keeping related things together and helps us to avoid the "feature-envy" anti-pattern.
- Separation of concerns decouples different reasons for change.
- Abstraction decouples consumers from implementation by hiding information.

Key Takeaways

- Coupling is the biggest single factor in the cost of change, and the cost of change is the cost of software.
- You cannot eliminate coupling. You can only decide which coupling you accept and manage the rest.
- Strong coupling is fine when it is stable. Unstable, accidental and hidden coupling is what hurts you.
- Name your coupling. Operational, developmental, semantic, functional and incidental coupling each have different remedies.
- The disaster zone is strongly coupled systems with slow feedback. Escape it by decoupling, by speeding up feedback, or both.
- Hard-to-write tests are a coupling signal. So are slow builds, blocked teams, and long parameter lists.
- Announce, don't command. Hide information. Translate at boundaries.
- Modularity, cohesion, separation of concerns and abstraction are all in service of managing coupling.

Further Learning, Reading & Viewing

Modern SWEngineering on YouTube – https://www.youtube.com/@ModernSoftwareEngineeringYT

Coupling Is The Biggest Challenge In Software Engineering – https://youtu.be/plMttQWztRM

How To Build Big Software With Small Agile Teams – https://youtu.be/cpVLzcjCB-s

Training By Dave Farley:

Modern Software Engineering Courses – https://courses.cd.training/

Free Tutorial "Design to Manage Complexity" – https://courses.cd.training/courses/take/design-to-manage-complexity

Books:

"Release It!" by Michael Nygard (the source of the five-type coupling taxonomy) – https://pragprog.com/titles/mnee2/release-it-second-edition/

"Modern Software Engineering" by Dave Farley – https://amzn.to/3rjCnXn

"The Software Developers' Guidebook" by Dave Farley – https://leanpub.com/softwaredevelopersguidebook