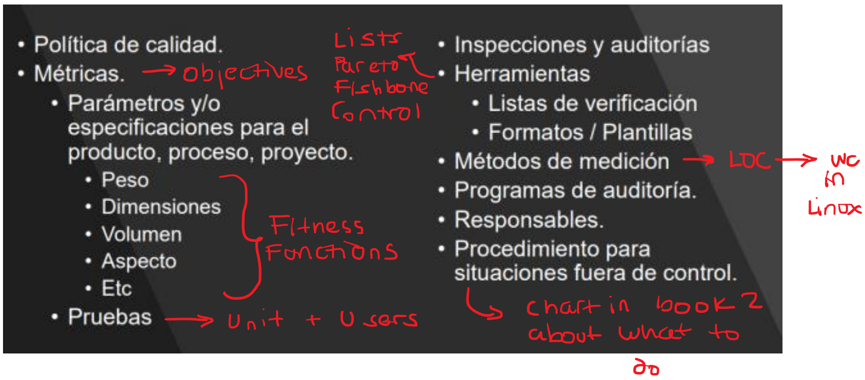
Applications In Software

Saturday, September 17, 2022 3:10 PM



B1 The four key metrics:

1) Deployment Frequency : number of unit changes

Not a count : total deploys in a time period ↳ # of commits → +0 for days with no deploys

2) Lead Time for changes : time that a dev's changes take to be pushed.

3) Change Failure Rate : The proportion of changes that cause a failure

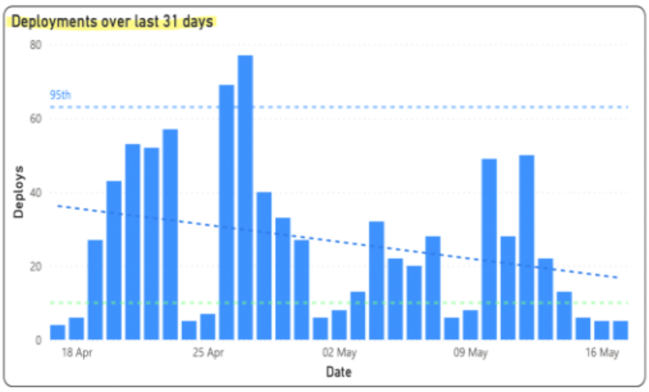
36 deployments a day, dev also resolved 2 issues } CFR = 2/36 = 5.56%.

4) Time to restore service: How long it takes a dev to be aware of a failure, diagnose it and fix it.

↳ the time a failure ticket takes from being created to being closed.

↳ calculate the mean of a 120 day period, for example

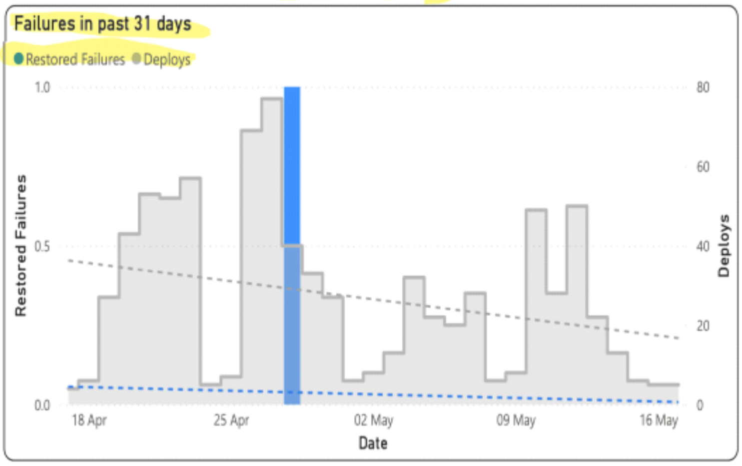
Charts



total	Today	Average
821 deps 31 days	5 deps	26 per day

2) Chart: Lead Time for changes NO

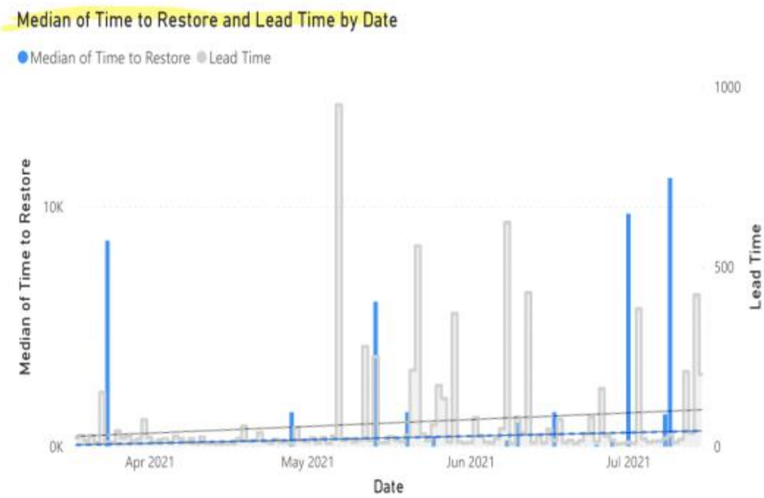
3) Change failure Rate



Y-axis : zero failures in a 24-hour period
↳ Very clear where there are problems

Failures	Deploys	Change Failure Rate
4	821	0.12%
Active Failures	in 31 days	% of failures

4) Time to restore Service



↳ Values plotted over a longer timescale (120 days) so that we could see improvement against a metric with fewer data points. Co-plot with lead time for context

Failures	Restored Failures	MTTR
Ø	14	1420
Active Failures	Sum of Restored Failures	Median of Time To Res...

Fitness Function: an objective function used to summarize how close a prospective design solution is to achieving the set aims. The fitness function defines the target metric

→ Functional Test : whether a system can create a new customer.

→ Architectural Test : whether a system can create 10 customers while also achieving an architectural / qualitative goal.

↳ creates a metric for how fast those 10 were created and if it was under 10 milliseconds.

Unit Test Coverage of 90%.

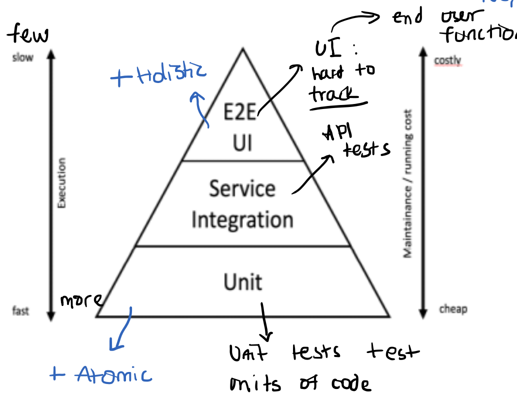
Fitness function 2.2

Integration test errors = 0% (when network latency is 10s for third-party API call); Execute on each nightly integration test build; Fail when integration test fails; Fail when test execution duration is > 10 minutes (standard execution time, without network latency is below 5 minutes)

Atomic Fitness Funcs: verify partial aspects
Holistic : broader feedback. Large part of the system is working good.

→ Tests execute automatically by a certain trigger or scheduled (nightly)

Static FF : check code for a fixed value
Dynamic FF : target response time for 10,000 users : 50 millisecs
100,000 users : 100 millisecs



The more tests you can have the more confidence you gain.
↓
but
cost of implementation

Top level holistic tests : checkout rate per minute
Revenue per minute
logins per minute
} deviations show detect arrivals

Bottom level (Unit tests) : Execution is triggered on each push to the source control system

Classify a test:
1) Breadth of feedback : atomic
2) Ex trigger : triggered
3) Ex location : CI
↳ triggered on each push to the source control
4) Metric type : specific value (>90%)
5) Quality att requirement : maintainability
6) Static or dynamic : static

provided by the following table:

Timeframe of day	Min revenue (per min)
01:00 AM - 05:00 AM	€ 200
05:01 AM - 07:00 AM	€ 400
07:01 AM - 09:00 AM	€ 600
09:01 AM - 11:30 AM	€ 900
11:31 AM - 01:30 PM	€ 1100
01:31 PM - 05:30 PM	€ 950
05:31 PM - 07:30 PM	€ 1500
07:31 PM - 09:00 PM	€ 750
09:01 PM - 00:59 AM	€ 300

test set containing the 5 main end user use cases (login, put item to cart, remove item from cart, view cart, checkout). The system performs all actions and responds within 100ms. Fail when test case fails; Fail when system doesn't perform actions and respond < 100ms

Technical debt: additional work later in time

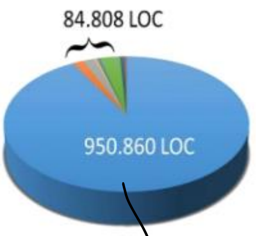
Modularity Maturity Index (MMI): to compare the technical debt in the arch of many systems
Modularity: a principle by Parnas (1970s) where a module should contain only one design decision,
And that the data structure for this decision should be local to the module.

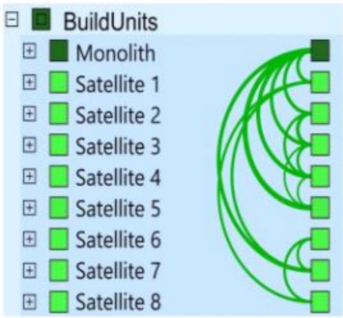
- Modules are units in a software system
- Program units that combine arbitrary, unrelated elements are not accepted.
- A modular system = low technical debt, low unnecessary complexity.

Units should contain sub-units that belong together

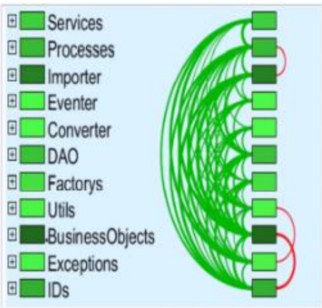
- Sub-units should have high cohesion, low coupling.
- Example: if subs of a module have higher coupling with other modules than with their sis and bros, Their cohesion is low, low modularity.

Units have names that can describe their tasks, show modularity. Vague names should be looked at.





unbalanced: candidates for decomposition for better modularity



Pattern Consistency
↳ connections
↳ If you implement a pattern, devs recognize quicker.

- 1. Modularity (45%) Hierarchy (30%) Pattern Consistency (25%)
  - 1.1. Domain and technical modularization (25%)
    - 1.1.1. Allocation of the source code to domain modules in % of the total source code
    - 1.1.2. Allocation of the source code to the technical layers in % of the total source code
    - 1.1.3. Size relationships of the domain modules ((LoC max / LoC min) / number)
    - 1.1.4. Size relationships of the technical layers ((LoC max / LoC min) / number)
    - 1.1.5. Domain modules, technical layers, packages, classes have clear responsibilities
    - 1.1.6. Mapping of the technical layers and domain modules through packages / namespaces or projects
  - 1.2. Internal Interfaces (10%)
    - 1.2.1. Domain or technical modules have interfaces (% violations)
    - 1.2.2. Mapping of the internal interfaces through packages / namespaces or projects
  - 1.3. Proportions (10%)
    - 1.3.1. % of the source code in large classes
    - 1.3.2. % of the source code in large methods
    - 1.3.3. % of the classes in large packages
    - 1.3.4. % of the methods of the system with a high cyclomatic complexity

The MMI is calculated by determining a number between 0 and 10 for each criterion with the Table
The resulting numbers per section (1.1,1.2,1.3) are added up and divided by the no. Of
Criteria in question. The MMI is recorded with the percentage of the principle so that a

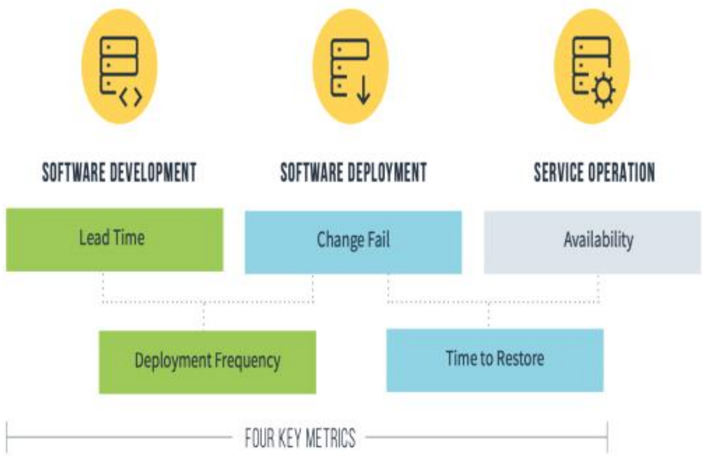
Number between 0 and 10 can be determined.

Section 0 1 2 3 4 5 6
1.1.1 <=54% >54% >58% >62% >66% >70% >74
1.1.2 <=75% >75% >77,5% >80% >82,5% >85% >87
1.1.3 >=7,5 <7,5 <5 <3,5 <2,5 <2 <1,5
1.1.4 >=16,5 <16,5 <11 <7,5 <5 <3,5 <2,5
1.1.5 No, partially, Yes, all
1.1.6 No, partially, Yes
1.2.1 >=6,5% <6,5% <4% <2,5% <1,5% <1% <0,6
1.2.2 No partially Yes
1.3.1 >=23% <23% <18% <13,5% <10,5% <8% <6%
1.3.2 >=23% <23% <18% <13,5% <10,5% <8% <6%
1.3.3 >=23% <23% <18% <13,5% <10,5% <8% <6%
1.3.4 >=3,6% <3,6% <2,6% <1,9% <1,4% <1% <0,7

- Rated between 8-10:
Low Technical Debt
- Rated between 4-8:
Quite a bit of tech debt
- Rated below 4:
Maintained with great effort,
Should consider worth upgrading
Or replacing the system

18 software systems that we assessed over a period of five years (X-axis). For each system, the size is shown in lines of code (size of the point) and the Modularity Maturity Index on a scale from 0 to 10 (Y-axis)

Conway's Law (Mel Conway):
Software architecture usually reflects the organisational structure of the company that creates and maintains it.



Also recommended: mean time to discover metric, is the average time between when

IT incident occurs and when someone discovers it. Trends in this metric tells whether
People are engaged and learnign from the past.
Example: A component cycle check. Consider the three components:

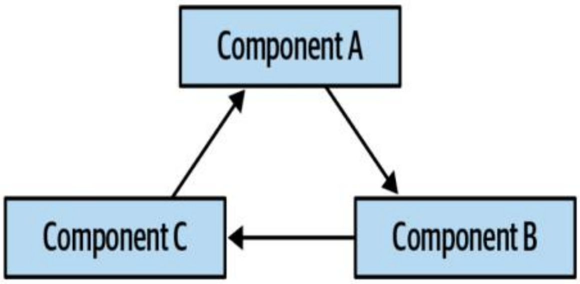


Figure 5-2. Three components involved in a cyclic relationship.

The cyclic dependency is an anti-pattern: when a developer tries to reuse
a component, each of the entangled components must also come.
Architects want to keep the number of cycles low.
Code CycleTest.beFreeOfCycles(component);
XP developers noted a correlation from past projects that more frequent integration led to fewer issues,
which led them to create continuous integration: every developer commits to the main line of
development at least once a day: merge conflicts arise and are resolved as quickly as they appear,
Instead of a final merging phase with a ball of mud.

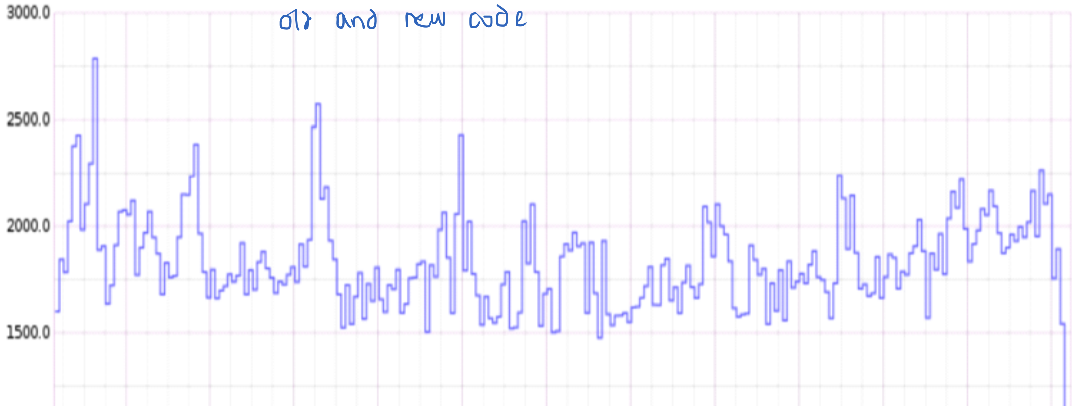
```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

Unit Tests can ensure communication
By Controller to be the only one, thus
Preserving architecture. Raise Warnings
Or raise exceptions.

Accuracy

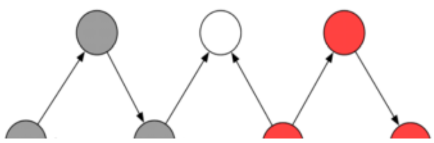
The number of times that the candidate and the control agree or disagree. View mismatches

old and new code

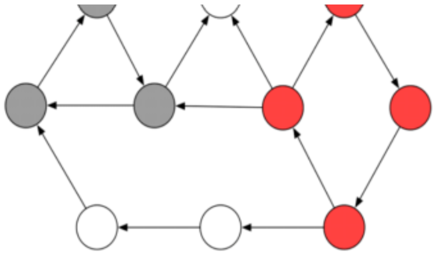


Entropy Kills Software:
Entropy, aka Structural Erosion, which results in the Big Ball of Mud, which is a badly tangled code
That has lost all architectural cohesion, a part that has a lot of undesirable dependencies between
Its parts that should otherwise be unrelated. Symptoms: a change in one part of the system breaks
Something in an unrelated part, lots of cyclic dependencies

A cycle group
The nodes in the graph are source files
The arrows are dependencies



A cycle group



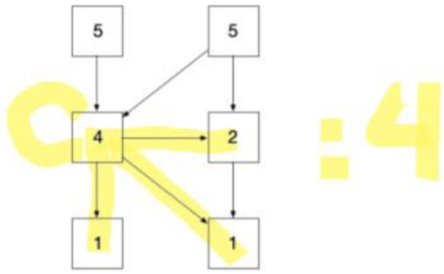
The nodes in the graph are source files
The arrows are dependencies
The image shows two cycle groups
Cycle groups, aka Code cancer
- Cycle groups make it impossible to

Test units.
- Modularity becomes impossible

Techniques to break cyclic groups:
- Dependency Inversion Principle: invert a dependency (arrow) in a cycle group, and it usually breaks the cycle
- Lift the cyclic dependency into a higher-level class that depends on the elements involved
- Demote the cycle to a lower-level class that handles the communication between the elements.
- Just move certain functionality to break a cycle.

Results: easier to test, understand, maintain, reuse.
Metric: the number of elements in your biggest source file cycle group. Define a threshold of 5, as soon as a cycle group has six or more elements, test fails. This ensures your code won't be a pile of mud.
Stats: 80% of systems with 100K+ LoC are piles of mud.
Usage: too many metrics can annoy your developers, and slow your progress without benefits. Around 5 or 6 metric based rules is the sweet spot. More declines your process.

Metrics to measure entropy
1. Average Component Dependency (ACD): how many elements a randomly selected element would depend on, including itself. Boxes are components (source files).



- Add all the boxes' values = Cumulative Component Dependency (CCD) = 20.
- CCD / num of boxes = Average Component Dependency (ACD). The minimum is 1 (no deps), the maximum is the number of boxes, 6.
- Depend Upon Metric (number in the box) / num of boxes for each node = Fan Out Metric = Propagation Cost Metric (PC) = high PC means high Entropy.
- PC = ACD / num of boxes. 3/6 = 0.5 = every time we touch something, 50% of all components are affected by average.
- PC = CCD / number of boxes (n) squared since ACD = CCD/n and PC = CCD/n^2
- So if the number of components doubles, the CCD needs to grow by a factor of 4 to keep PC the same. Minimize this PC Metric, but if it goes down due to increase in components, not good.
- If your system is small (n < 500), higher PC values are less concern.
- If your system is big (n >= 5000) even 10% is concerning: every change might affect an avg of 500 components (ACD 500).

2. Cyclicity
- Cyclicity: defined as the square of the number of elements in a cycle group. A cycle of 5 elements, has cyclicity of 25
- Relative cyclicity: add all the cyclicity values of all your cycles, take the square root of that, divide that result by n and multiply that result by 100

Metrics to measure size and complexity

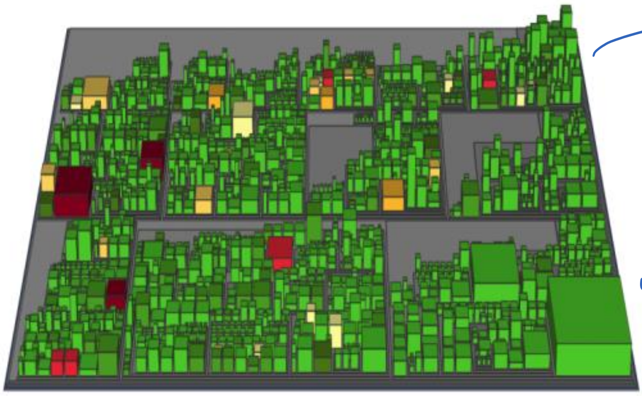
1. Size Metrics
- Lines of Code (LOC): executable lines, skip the empty and comment lines. If your file has 5K LoC, it

is complex. Recommended: size of source files around 800 LOC, something bigger is better to divide into more files.

- Number of Statements: measure the size of functions. Keep your code readable and maintainable. Recommended: threshold of 100 statements per function.
- Cyclomatic Complexity: computes the number of different possible execution paths in a method. This is a floor value for the number of test cases needed for 100% coverage. It is a flow graph and the number of nodes and edges in that graph. Its computation: start with 1 and add 1 for each loop statement or conditional statement or cases in a switch. High CC correlates to high complex and hard-to-read functions. Error rates increase quickly for values above 24. Safe value is 15. Also add 1 per logical && || expressions.
- Identation Debt: measures complexity by counting the maximum code indentation levels in functions. The deeper the indentation, the more complex the method. Spots complex code. Recommended: threshold of 4 maximum indentation level.

Change History Metrics

1. Change frequency: how often a source file changes in a given time frame. Number of Changes (d) where d is the time frame.
2. Code Churn (d) how many lines were added or removed from a given file in a time frame.
3. Code Churn Rate (Code Churn / no. Of lines = 90) = 2, this file has been rewritten twice in the last 90 days. This pinpoint instabilities.
   Structural erosion shows if a change breaks many unrelated parts. Such problems are introduced in complex files that change frequently.



each building is a source file

grouped by module
↳ area : LoC
↳ height : avg complexity in file
↳ shade : change freq in 90 days.

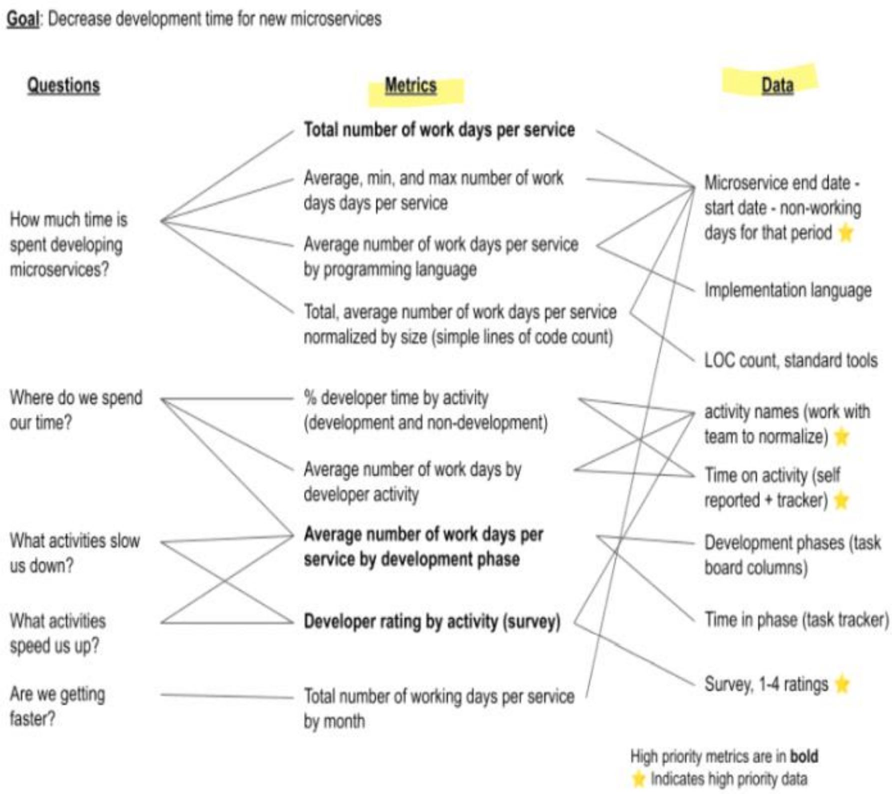
all tall and dark = refactor

Relative Cyclicity: package level 0%, 4% or less for components.
Structural Debt Index, for components with a threshold in the low 100s.

Golden Rules for Better Sofware:

1. Have an enforceable architectural model that defines the different parts of your software and the allowed dependencies between them.
2. Avoid circular dependencies on the namespace/package level.
3. Limit circular dependency on the level of source files/classes. Any cycle group with more than five elements has a good chance to turn into code cancer
4. Limit the size of source files to 800 LoC
5. Limit max indentation to 4 and Modified Cyclomatic Complexity to 15

Goal-Question-Metric Approach



Example: Incident #1: Too Many request to the Service (cloud)

- Is the Foo Service having a problem, or are we having a problem connecting to the Foo Service?
- What is the current Foo Service API usage?
- % timeout request over a 15-minute window
- % authentication error request over a 15-minute window
- Count of total requests
- Count of normal, error, and timeout response
- % error response over a 15-minute window