Software Improvement Group

SIG ARCHITECTURE QUALITY MODEL

Guidance for Producers

June 16, 2025

SOFTWAREIMPROVEMENTGROUP.COM

Authors

Dennis Bijlsma
Head of Product
dennis.bijlsma@softwareimprovementgroup.com

Michael Olivari
Senior Product Owner
michael.olivari@softwareimprovementgroup.com

Bugra M. Yildiz
Senior Software Consultant
bugram.yildiz@softwareimprovementgroup.com

Kriti Dhingra
Feature Consultant
kriti.dhingra@softwareimprovementgroup.com

1. INTRODUCTION

The ability for a company's IT solutions to adapt and grow with changing business needs continues to be a point of contention within the software industry. At an increasing rate, we see IT managers being challenged with modernizing and/or evolving large portions of their software landscape due to new business initiatives due to significant events (such as an acquisition of new IP) or production issues due to dependence on legacy software.

These organizations often lack the insights with respect to how their applications communicate with one another and it is unclear where knowledge over the system resides within the development team. Dependencies across the fundamental parts of the system are often hidden from view, resulting in any one change often creating a cascade of complications that need to be addressed with any new development. These complications significantly delay the time to market for a given line of business and costs begin to snowball quickly. As a result of this architectural brittleness, organizations are often hesitant to make foundational changes without insight into these aspects. This is where we come in.

In the past, we utilized the TÜViT Maintainability Model to assess the architecture of a system developed by a single team. Component level metrics were defined to give insights into how a system is structured in its basic parts, how these parts communicate statically, and what anti-patterns in communication may be present. While these software aspects work well in the limited scope of a single system, it does not go far enough in providing insights into how a cluster of systems interact together. Our Architecture Quality Model has been developed to measure the degree to which an organization's implemented software architecture within a line of business, often consisting of a collection of software systems/applications, is designed with evolution in mind.

2. MODEL DEFINITION

SIG defines architecture quality as the degree to which an architecture is flexible and adaptable to change. The basic idea comes down to modularity within an architecture - specifically how well are systems in the landscape defined via components and the degree to which these components can easily be worked on in isolation or swapped out with new functionality. Ideally, components are encapsulated to a high degree and map to a specific business responsibility or technical boundary defined by the line of business.

Currently, there is no standardized approach in the industry to measuring architectural flexibility for this context. However, due to SIG's extensive experience and expert domain knowledge, we have been able to derive 5 sub-characteristics of software architecture that can easily be assessed from specific measures on source code:

Figure 1 Architecture aspects in the SIG Architecture Quality Model

The architecture aspects used in the model use the following definitions:

- **Structure**
  Definition: The arrangement of and relations between the parts or elements of an architecture.
  Unlike modern systems with microservices or service-oriented architectures, legacy systems typically lack clear boundaries between functional and technical areas. The extreme example is a huge monolith with various problem areas in one big box. This makes it difficult to navigate issues, distribute maintenance effort, and extend functionality.

- **Communication**
  Definition: The complexity of imparting or exchanging of data throughout the architecture.
  A common issue for legacy systems is "spaghetti code," where dependencies are too intertwined to analyze and understand the functional logic mapped to code. The result is then that no one dares to touch any piece of code, as any change could lead to issues in production.

- **Data Access**
  Definition: Ease and efficiency of accessing or retrieving data stored within a database or other repository.
  There's usually only one big database for multiple functional areas of a legacy system. When many parts of a system depend on a single database, modifications to the data structure will very likely result in a cascading

effect of modifications across the areas that rely on it, making the database schema for a legacy system too rigid for change.

- **Evolution**
  Definition: Degree to which changes can be made in isolation across the architecture.
  In legacy systems, it's difficult to make a change in one area without affecting another, making it difficult for multiple teams to work independently. This co-evolving behavior should be avoided, as not being able to isolate change is one of the biggest obstacles for development productivity.

- **Knowledge**
  Definition: Degree of technical knowledge distribution among team member within the organization.
  In working with our clients, we see a great deal of large legacy systems highly dependent on just one or two key developers. This means huge risks for business continuity; if these key developers leave or retire, sufficient knowledge of the systems will cease to exist.

2.1 MAPPING ARCHITECTURE ASPECTS TO SYSTEM PROPERTIES

As the system is developed, understanding how code is structured, coupled, and evolving, gives insights into which areas of the codebase are most flexible and resilient to change versus those that will create the most headache for development when adapting the software. To this effect, SIG has defined 9 system properties that are easily measured from source code and version control data to evaluate each of the 5 sub-characteristics of architecture quality.

Figure 2 Mapping system properties to architecture aspects

With this model, it is possible to assess each sub-characteristic for every component level within a collection of related systems inside an organization's IT landscape, from lowest subcomponent-level upwards to system-level, with an overall valuation of the architecture quality for the collection itself. Effectively, this means that every component will receive a quality score for each sub-characteristic of architecture so that it is easy to pinpoint areas of inflexibility within an architecture simply by drilling down or up through components.

2.2 AGGREGATION

Calculating an overall score for architecture quality is done via aggregation over the quality scores for each sub-characteristic.

For example, suppose a system consists of 2 components, Component A and Component B, we first measure and evaluate the quality profile for each sub-characteristic of Component A and then do the same for Component B. We then measure the sub-characteristics on the parent component, in this case the system itself, to determine its external quality profile.

Averaging the sub-characteristic scores for both Component A and Component B based on the percentage of volume each component makes up for their parent component, we then can calculate an internal risk profile for the system. Together, we simply average the sub-characteristic ratings for the external quality and internal quality profiles to come to an overall quality profile for the parent component. This process is repeated at all levels across the collection of systems to arrive at an overall score for the landscape itself.

3. SYSTEM PROPERTY DEFINITIONS

The following section details the individual metrics utilized in the SIG's Architecture Quality Model. The motivation, approach and specific design decisions are detailed, along with any future work that may be considered in following revisions of the model.

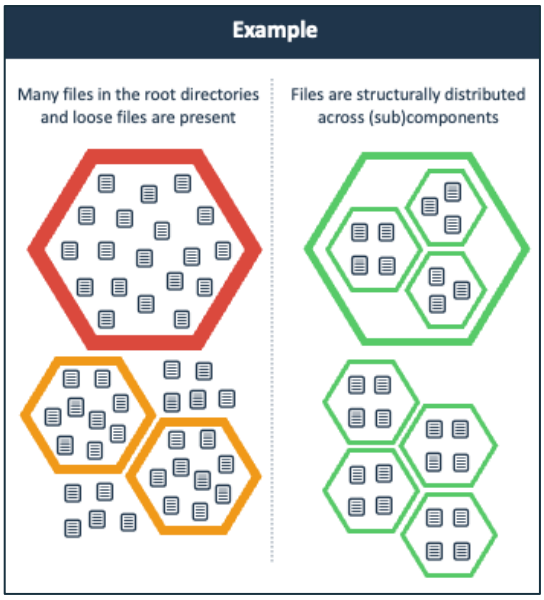
For each system property, the following information is provided:

- A description of the metric.
- A conceptual example.
- The measurement level: The type of system level for which the metric can be applied.
- The unit of measurement: Defines the actual measurement that is performed and describes the unit for the (numerical) measurement result.
- Metric order: Indicates how the measurement results should be interpreted.
- 4- Star Rating threshold: Indicates the metric value needed to obtain an "above market average", corresponding to the observed value of the unit of measurement for a given code artifact / component.
- Some typical architectural questions the metric can help to address.

3.1 CODE BREAKDOWN

The metric Code Breakdown assesses how well the files in a codebase are organized into components and subcomponents.

Best practices involve organizing files into component and subcomponents (directories, packages, etc.) by some logical or technical grouping. In this way, the codebase becomes more maintainable: Understanding the code structure becomes much easier when changes are required, reuse of those components can be simpler. A low score of the Code Breakdown metric points that some re-structuring of the files in the codebase is needed for improvement.



Metric information

- Measurement level: Component
- Unit of measurement: Number of source code files in the component, where those files are not located within one of the component's subcomponents.
- Order: Lower is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should contain 10 files or fewer. Components that contain more than 6 files should introduce sub-components.

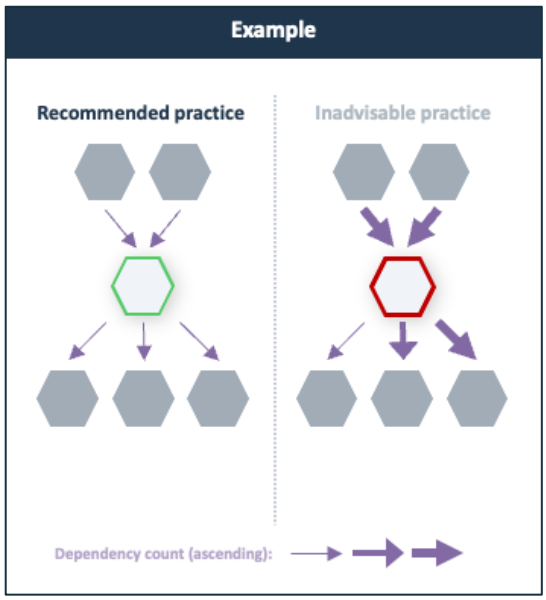
Related questions

- Which components and subcomponents require reorganization of their files?
- Are there loose files in the root directory of a component?
- Are files distributed amongst a component's subcomponents?

3.2 COMPONENT COUPLING

Component Coupling measures the degree to which components are depended on and depend on other components that make up a system. Components that must be modified together due to explicit dependencies are considered coupled components.

A component with a high degree of external dependencies to and from other components is more difficult to maintain in isolation. The ability of this component to evolve is limited as a change in its interface would likely cause a cascading effect of many additional changes across other components that comprise the system or landscape.



Metric information

- Measurement level: Component
- Unit of measurement: The sum of the count of a component's incoming and outgoing dependencies. Internal dependencies between the component's subcomponents are not included.
- Order: Lower is better

4-Star Rating threshold
To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should contain 42 or less dependencies on other components. Components with more dependencies should either try to reduce coupling altogether or introduce sub-components to subdivide the dependencies.

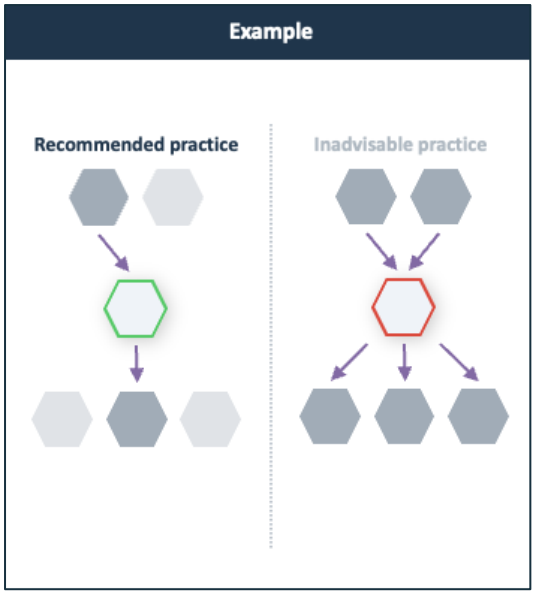
Related questions
- How many dependencies, both to and from a component, are there?
- Are there any components with many dependencies from or to other components?
- Are there obvious cyclic dependencies between components?

3.3 COMPONENT ADJACENCY

Component Adjacency quantifies the number of distinct components that a component directly interacts with (adjacent to).

A lower value of this metric is more desirable, as it indicates that a component has fewer direct inter-component connections. This reduced interconnectivity can lead to several benefits including easier maintenance, improved modularity and decreased risk of ripple effect in case of architectural changes.

Components with lower adjacency are typically more independent and easier to test and modify in isolation. Conversely, a higher value of this metric suggests that a component interacts with too many other components, potentially making it more challenging to maintain and evolve over time.



Metric information
- Measurement level: Component
- Unit of measurement: The number of distinct components that a component directly interacts with (adjacent to).
- Order: Lower is better

4-Star Rating threshold
To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should, on average, interact with 6 or fewer other components.

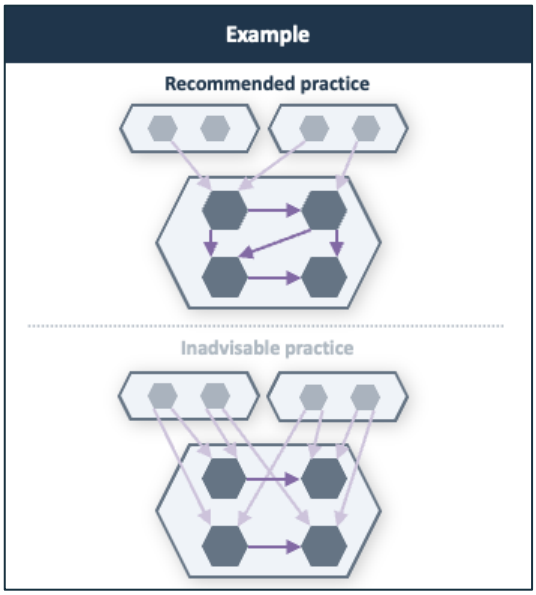
Related questions

- How many other components does a component interact with?
- Are there any components with too many adjacent components?

3.4 COMPONENT COHESION

Component Cohesion measures the degree to which components encapsulate specific business responsibilities / functionality within the system. Components are considered cohesive when they have been designed around a specific business responsibility and only consist of those sub-components and modules that implement the defined requirements of the business responsibility.

The logic within a component should be related so that the component is more reusable and/or replaceable by a new one when the need arises. When a component contains diverse functionality, it is more likely that it is going to be referred to by more external components, which will lead to more dependencies across the system. This functional entanglement often results in increased development effort and longer release cycles when any foundational modifications to the system is needed to be made.



Metric information

- Measurement level: Component
- Unit of measurement: Ratio between the component's internal and external dependencies.
- Order: Higher is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should have a ratio between internal dependencies and external dependencies that is at least 70%. This means that the internal dependencies within a component should significantly outnumber the external dependencies to/from other components. Cohesion can be improved by merging components that are highly coupled, or by introducing a new component consisting of parts that were previously part of other, tightly coupled components.

Related questions

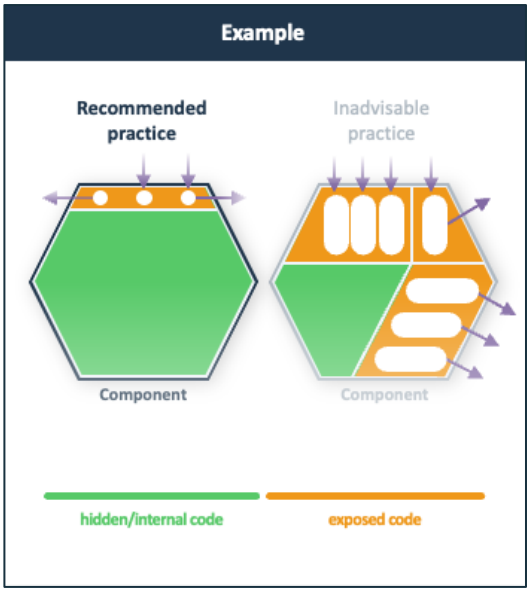
- Do subcomponents communicate only to components outside of the chosen component?

- Is there a complete path between all subcomponents within a component? Do all subcomponents depend on at least one other subcomponent within a component?
- Are there identifiable “islands” / graphs where some subcomponents communicate with some subcomponents but not others?

3.5 COMMUNICATION CENTRALIZATION

This metric measures the extent in which the communication from one code component to another is centralized, specifically the degree to which code within a component is externally accessed by other components and the degree to which code inside the component has direct dependencies on code located in other external components. The less internal code that interfaces with other components in the system, the better encapsulated a component is said to be.

When calls from a component to other components are not centralized, encapsulation is low, resulting in a component that is increasingly more sensitive to changes in the “outside world” and makes it more difficult to update the component if such changes occur. Likewise, when a large volume of code of a component is makes up its interface, i.e. the code that is directly accessed by other components, also indicates low encapsulation of the code within the component. As a result, when the code within the component is updated, the risk that other components are affected (due to changes to the interface) is higher.



Metric information

- Measurement level: Component
- Unit of measurement: Percentage of code within a component that is *not* involved in direct communication with other components.
- Order: Higher is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should have 47% or more of their code categorized as internal code. If needed, the amount of exposed code can be reduced by localizing the code used by other components.

Related questions

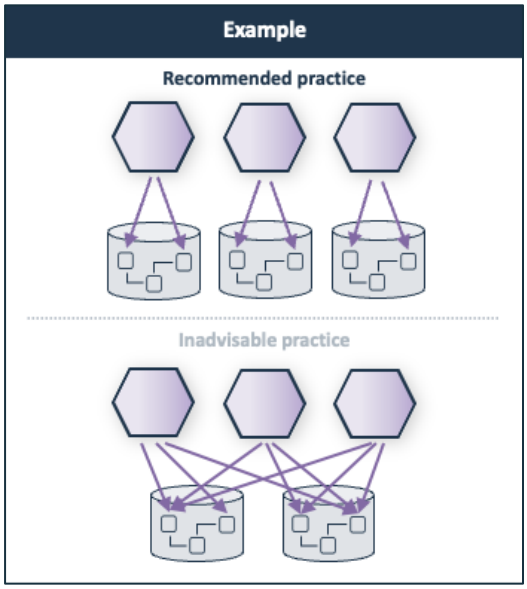
- What other components does this component communicate with?
- Where in the code are these dependencies implemented?
- Is the part of the component that communicates scattered across every file, or is it nicely centralized in certain files or subcomponents?

3.6 DATA COUPLING

Data coupling measures the degree to which components are dependent on individual data entities within a system's datastore(s).

When many components depend on a single data entity (such as, a table) in a data store, modification to that data entity is likely to cause a large effect across the components that rely on it. This happens particularly when a data entity does not have well-defined responsibility boundaries or the dependency structure to such an entity is not centralized in the codebase.

The recommended practice is to limit the number of components accessing individual data entities by centralizing the access and defining data entities based on well-defined responsibilities.



Metric information

- Measurement level: Data store collection
- Unit of measurement: The average number of components accessing distinct data entities within the data store.
- Order: Lower is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, data stores in the system should interact with at most 1 component. If multiple components access the same data store, a data access layer or "owner" service component should be introduced, which shields the data store from direct access by too many components.

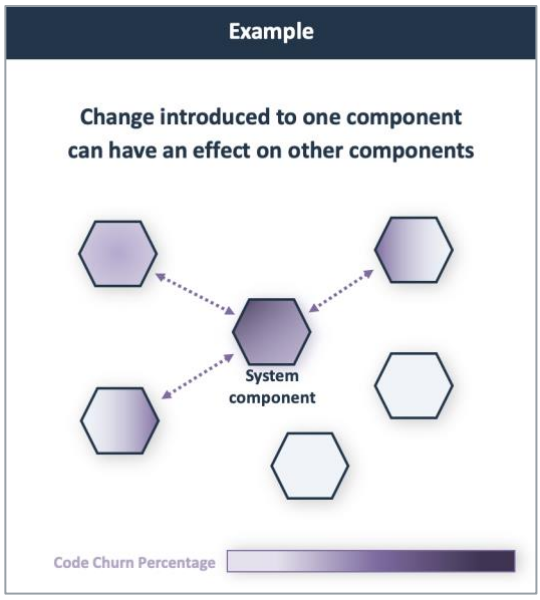
Related questions

- What components need to access the data?
- Are there any data entities which are accessed by too many components?
- Which component has ownership over this data?
- Should this component provide an interface to allow other components to access its data?

3.7 BOUNDED EVOLUTION

Measures the degree of co-evolution of components within a system based on the frequency of coupled code modifications over time. Co-evolving components are defined as components modified together at a regular frequency and indicate implicit functional relationships within a codebase.

The frequency at which components are updated should be limited and respective to a specific context / business function. Implicit code dependencies are often found across components that evolve together at a high frequency, with distant co-evolving components an indication of an anti-pattern of architecture due to a lack of contextual boundaries within a given system. Ideally components that undergo change together are logically grouped together and/or consolidated to reduce risk of an ever increasing maintenance scope.



Metric information

- Measurement level: Component
- Unit of measurement: Percentage of changes made to the component involved in co-evolution with other component(s).
- Order: Lower is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should have at most 16% of commits that include changes with other components.

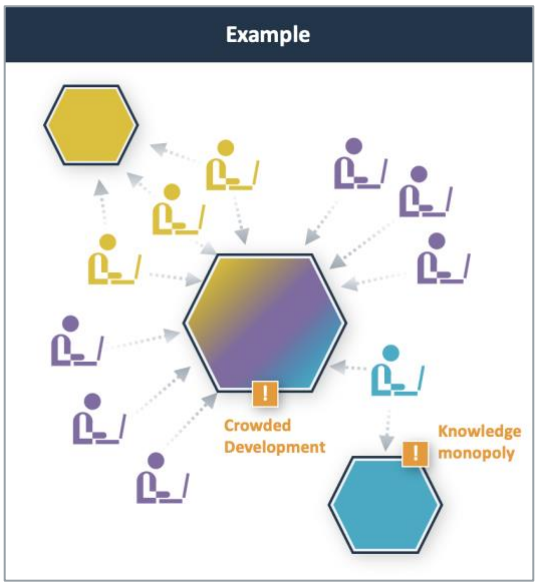
Related questions

- What parts of code are changing together?
- Are those parts spread across the architecture, or located within a component?
- When the component needs changes in other components, is it always the same parts that are touched?

3.8 KNOWLEDGE DISTRIBUTION

Measures the degree to which development can grow and retain knowledge over a given system. Knowledge is determined in part by active development across the components that compose the system and the relative distribution of the development team that is able to efficiently contribute to these components further.

Knowledge should be uniformly distributed over a system to ensure continued development can be accomplished with ease as the development team undergoes change over the lifetime of a system. Too few developers working on a large component, let alone an entire system, should be avoided as there is a risk of losing the crucial knowledge if the team is further reduced in size.



Metric information

- Measurement level: Component
- Unit of measurement: Number of authors with significant contributions to a component per KLOC.
- Order: Higher is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should be maintained by at least 0.26 authors per KLOC. For example, a component containing 10,000 lines of code should be maintained by at least 2.6 authors.

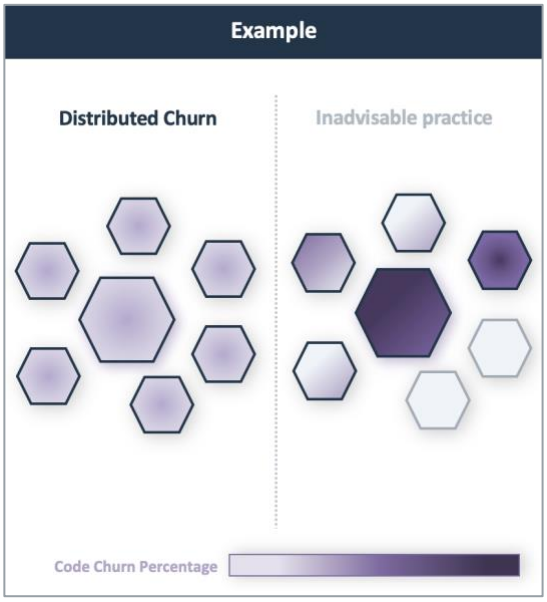
Related questions

- How many developers have contributed functional code to a given component?
- Are there any components that have not had an author within the past (sprint/month/year/decade)?
- Do authors regularly contribute roughly the same amount of churn on components that they are authors of? Does commit volume vary significantly?

3.9 COMPONENT FRESHNESS

Measures the degree to which components are actively being kept up to date and maintained. Components that are “fresh”, or actively maintained, are easier to further maintain, and their knowledge is retained and readily available across the system.

Fundamentally, active maintenance over a component suggests that knowledge over that component is top of mind for a development team. This further suggests that maintenance and functional growth within that component can be completed with a higher degree of efficiency as opposed to components that have not undergone any recent change. Too little activity suggests a component is outdated and knowledge is slowly being lost within the development team. Furthermore, regular maintenance should ideally be uniform to the system and distributed over its multiple components and not centralized within a single component.



Metric information

- Measurement level: Component
- Unit of measurement: Relative weekly churn, calculated by dividing the average weekly churn by the component's volume.
- Order: Higher is better

4-Star Rating threshold

To be eligible for a ★★★★☆ rating in the SIG Architecture Quality Model, components in the system should (on average) change at least 26% of their code per year. For example, a component containing 10,000 lines of code should have at least 2,600 lines of code changed on a yearly basis.

Related questions

- Looking at churn over time, do we see that most modifications happen in just a few components?
- Which components were most recently touched?
- Which components have not been touched for the longest time?
- What percentage of volume of components has been churned since the past (sprint/month/year)?
- Is there a clear pattern of only some components being churned while others remain unchanged?