**Independent Project 04**

**Software Metrics of Collaborative Online judge**

**Ip Chan**

**City University of Seattle**

chanip@cityuniversity.edu

# Introduction

In this paper I will discuss how I would design a collaborative online judge system.Collaborative Online judges are systems designed for the reliable evaluation of algorithm source code submitted by users, which is next compiled and tested in a homogeneous environment. Online judges are becoming popular in various applications. For this week I will develop some metrics for verification and acceptance testing. High cohesion and low coupling are the criteria for judging the quality of this design. This report is mainly about this software metrics section. Software measurements are objective, quantitative assessments of software attributes. Software metrics are standard measurements of software attributes.

# “Good” Design Attributes

A good design software should match the requirements. The detailed requirements of this collaborative online judge system have been explained in detail in IP03; the metrics of these requirements I will discuss in detail in this report which included **functional metrics and non-functional metrics**. Besides, two additional characteristics should be present for a good design: attributes are **consistency across design** and **completeness of the design** which are explained in the following.

## Consistency and completeness

Consistency, defined as the requirement that a series of measurements of the same project carried out by different raters using the same method should produce similar results, is one of the most important aspects to be taken into account in the measurement methods of the software. Completeness means a quality demanded to the set of software requirements and to each requirement itself, in order to ensure that there is no information left aside. Learn more in: Dealing with Completeness in Requirements Engineering.

# Testing Metrics Before Agile

Agile testing enables collaboration and consistent communication between the development and testing teams. The Agile development environment relies on the collaborative effort of cross-functional teams.

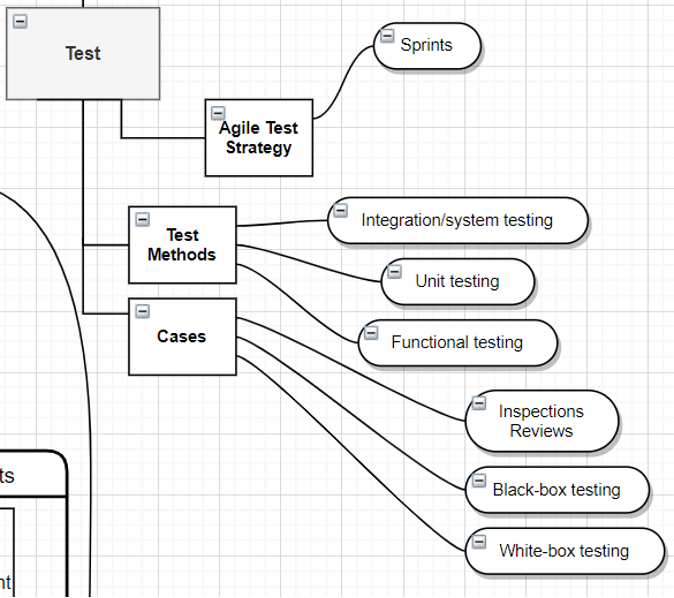
Below are the common agile metrics in detail.

## Sprint Burndown

Agile teams use Sprint Burndown charts to depict a graphical representation of the rate at which teams complete their tasks and how much work remains during a defined sprint period.

The typical burndown chart plots ideal effort hours for completing a task using remaining hours of effort on the y-axis and sprint dates on the x-axis. The Agile team then plots the actual remaining hours for the sprint.

Figure 2.1 shows that the measure completed in the Test part needs to meet the Agile Test, Test Methods, and Cases. An agile test needs to go through Sprint. Test methods have gone through three main tests which included Unit test, functional test, and Integration/system testing. The Cases section performed Inspections and reviews, black-box testing, and white-box testing.

****

*Figure 2.1 the metrics in the Test section.*

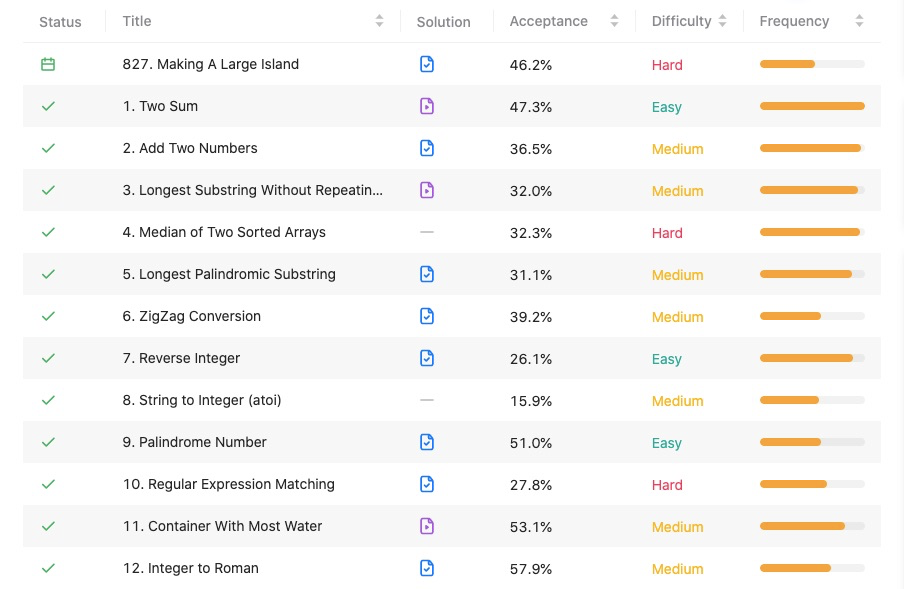
# Metrics Definitions

## Functional Metrics

Now let’s turn our attention to the actual metrics of our service. We will only use two types of metrics in our service, Counter and Gauge. Counter is the total count of some value for a certain metric, whereas the Gauge is the arbitrary value for a metric. Metrics can be divided into functional and non-functional metrics. This section will discuss the functional metrics, in the next section we will discuss non-functional metrics.

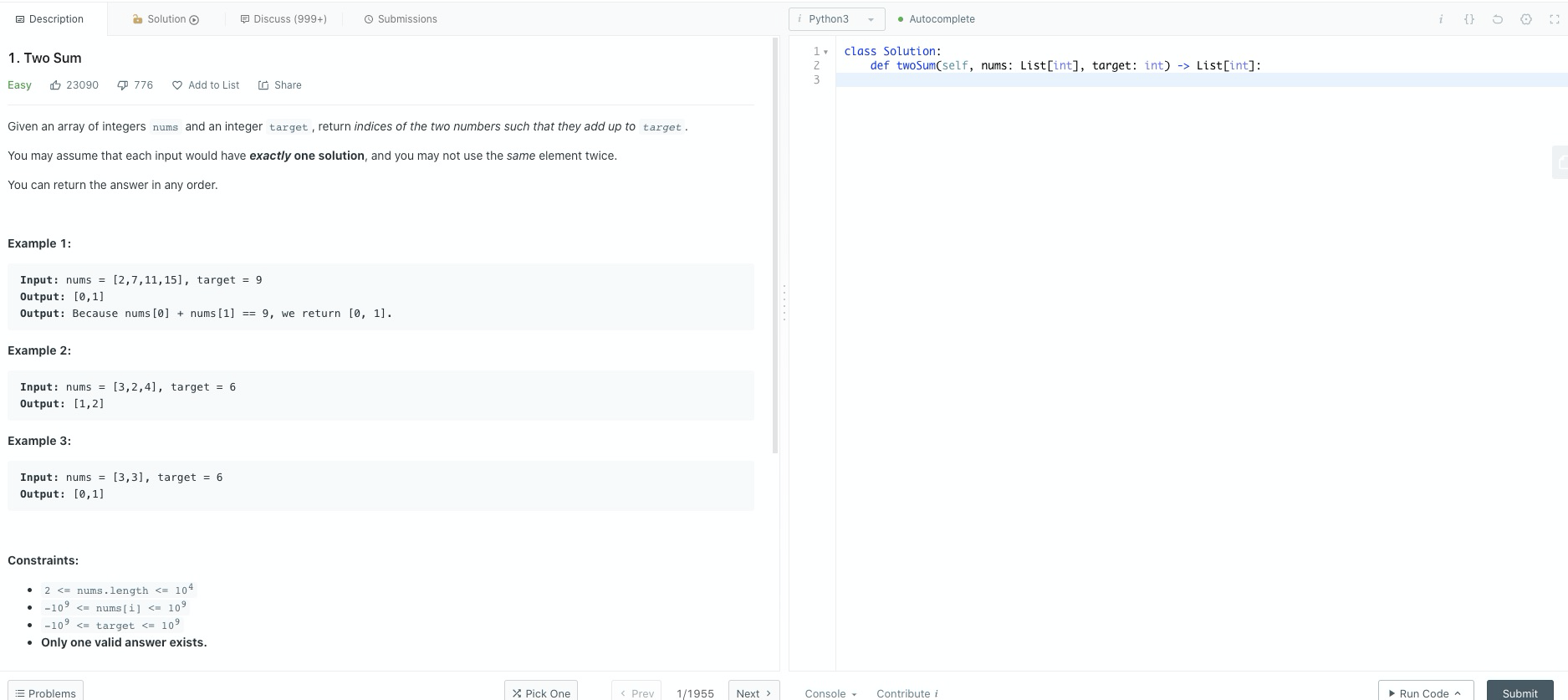
Based on our testing scenarios in previous sections, as well as previous IPs (IP3 in particular), we will need to provide the following functional metrics.

* Problem-list
  + The monitoring system should collect metrics for each request the user makes. One of the metrics should be the total count of the existing problems, which should be equal to the number of problems in the database. Everytime the metric should be true if the count of the problems in the problem list returned to the user is the same as the count of problems in the database, false otherwise. This metric itself can be defined as the Gauge. There can also be a rollup metric which is a Counter, and it counts the number of false for the Gauge metric. If the Counter metric increasing rate is above the pre-defined threshold, an alarm will be triggered, and the oncall operator should be engaged to perform mitigation, and root cause analysis.
  + There should be a Gauge metric that collects the response status code for the problem-list HTTP requests. Once again, a rollup Counter can be defined to count the total number of 4\*\* and 5\*\* codes. If the Counter metric increases faster than a predefined threshold, alarm should be triggered and an oncall operator should be engaged.
  + There should be other business logic metrics defined along the way the service is running. Every time there is a major event that causes customer impacts, engineers and operators should sit together and discuss the lessons learned. They should provide certain action items in order to avoid future similar events from happening again. These action items might include additional metrics.



*Figure 3.1 The problem list*

* Problem-details
  + Each problem detail HTTP request user makes, the monitoring system should collect metrics such as the description, the difficulty level, etc. to be consistent with the values in the database. If any of these is not consistent, a false value for the metrics should be collected and reported. Similarly, rollup metrics with Counter type can be used to trigger the runtime alarms.
  + Similar to the Problem-list metrics, response code metrics can be collected and reported.
  + Similar to Problem-list metrics, additional business logic metrics might need to be reported.



*Figure 3.2 The problem details*

* New-problem
  + Every time a new problem is created, the return code from the database should be reported as a Gauge metric. A rollup metric can be used to count the total number of failure response codes from the Gauge metric.
  + Similar to Problem-detail and Problem-list, HTTP response code metrics and other business logic might need to be defined and reported.
* Editor
  + The monitoring system should collect and report business logic metrics such as the type of programming supported for each particular coding problem.
  + It also should report a metric that collects the response code for the “Submit” button for each coding problem. If the submission failed, a rollup Counter metric should be defined and reported during runtime to trigger the alarms.
  + Similar to Problem-detail and Problem-list, HTTP response code metrics and other business logic might need to be defined and reported.
* User Profile
  + Every time a user makes a HTTP request for her profile, the system should check if the data the service returned is consistent with data in the database.
  + Similar to Problem-detail and Problem-list, HTTP response code metrics and other business logic might need to be defined and reported.
* Backend
  + There should be a metric that shows the effectiveness of the collaborations between multiple users for a particular coding problem. This metric can be Gauge metric showing the response code every time the Websocket-based communication occurs from the backend service.
  + There should be a metric that shows the interaction between the backend service and the upstream Online judger agents. Every time a job is sent from the backend to the Online Judger, a Gauge metric showing the response code should be collected and reported.
  + Similar to Problem-detail and Problem-list, HTTP response code metrics and other business logic might need to be defined and reported.
* Online Judger
  + There should be a metric that reports the effectiveness of the Online Judgers. This metric should report the final execution result for each job that is run inside the judger. The metric should be a Gauge, which reports the actual result of the judgements. A rollup Counter can be used to report the total number of failures (there might be more than one types of failures) from the Online Judgers.
  + Similar to Problem-detail and Problem-list, network protocol response code metrics and other business logic might need to be defined and reported.

## Non-functional Metrics

In addition to the functional metrics discussed above, there should also be various non-functional metrics. These metrics are more generic and can be applied to different services. Many well-designed existing web-services have a comprehensive set of non-functional metrics. Non-functional metrics can be further divided into three broad categories: Security, Availability, and Performance. Below is a list of such metrics, but it is not the comprehensive list, which is beyond the scope of this individual report.

* Security
  + There should be some metrics that monitor the potential DDoS attacks, which is very common in today’s distributed web services. These metrics can be, for example, the source IP addresses from the requests, the ASN with those source IP addresses and so forth. Note that there is no easy way to accurately predict the DDoS attacks, and metrics are one of the mechanisms to proactively protect our services from the attacks.
  + There can be some metrics to collect and emit the HTTP vs HTTPS requests made by the users (assuming our web-services also supports HTTP requests).
  + There can be some metrics to report the TLS protocol versions (in particular, TLS1.2 or TLS1.3) for SSL Handshakes. For that matter, we can also collect the cipher suites used, whether or not the TLS session resumption is used, Session ticket or Session cache is used, and so forth.
  + Security is often the most important characteristic for today’s web services; more important than Availability and Performance. Therefore, additional metrics might need to be defined along the way our service is running and evolving.
* Availability
  + One metric should be used to collect the 4\*\* and 5\*\* response code returned by any of the sub-services, which is discussed above. Rollup Counter metric can also be used to trigger the alarms.
  + A metric should be used to collect the Service uptime and downtime. Typically there is an SLA (Service Level Agreement) that specifies the total downtime for a year, a month, a day, and so forth. For example, a SLA of 99.999% uptime means the whole web service should not crash longer than 5 minutes 15 seconds in an entire year. For that purpose, this metric is very important to monitor the overall health of the system.
  + Health check metrics to check the health of each individual sub-services. Depending on the protocol used, and the implementation and design of each sub-service, the health check might not be the same. For example, HTTP based services such as the the frontend and the backend can report the total of 200 response code to the health check related metrics, whereas the Websocket protocol and Redis, Online Judger, etc. have their own communication protocols, which requires the monitoring system to define specific health check mechanism and metrics.
* Performace
  + RPS and BPS of the entire web-service
  + Time of first byte for each individual component of the service
  + Time of last byte for each individual component of the service
  + Hardware related metrics, such as CPU utilizations for the servers, memory usage, TCP related metrics, and so forth.
  + Others such as DNS lookup time, cache hit rate (e.g. Redis), and so forth.