Proposal for GSoC 2018

Jenkins - Code Coverage Plugin

About Me

BASIC	**English Name:** Share Cheng **Chinese Name:** 郑深语 (Shenyu Zheng)
INFORMATION	**Email:** zsy19980307@gmail.com **Phone**: +86 18738976310 **Github:** cizezsy
EDUCATION	**Henan University,** Kaifeng, China Computer Science and Technology, expected graduation July 2019

ISSUES	JENKINS-43866 Don't round up 99.x% coverage to 100% JENKINS-8958 Code coverage color scheme not suited for color blind users
RELATED EXPERIENCE	● Proficient in Java, familiar with Spring and Spring MVC, fluently use Maven, knowledge and experienced in Android. ● Familiar with Linux Command Line, understand Shell programming. ● Familiar with Git.

Project

Project Name

Code Coverage Plugin

Project Description

Jenkins GSoC 2018 Project Idea - Code Coverage Plugin

Synopsis

1. Config
   we provide a build step configuration which users can specify Ant-Style path and Adapter for each report file(it looks like the build step configuration of warnings-plugin).

We have two methods of providing an Adapter for users to choose. One is to implement AbstractAdapter class and to put a @Extension annotation on the static factory class of the implementation. This factory class is the subclass of Descriptor<AbstractAdapter>. The other is to provide a script file which specifies the details of adapt step. Code Coverage Plugin will parse the script file to an Adapter(optional).

2. Detect
   When Code Coverage Plugin starts running, it will detect code coverage reports according to user's config. Once a report is founded, plugin starts a parse it.
3. Parse
   Since all code coverage reports can be parsed to a tree representation, we write two Parsers for all of the reports. One Parser is for XML format, the other is for CSV format.

   The two parsers have same results - a Tree and a MetaInfo list(it stored in HashMap). The tree contains nodes of the same structure, <Type, Parent, Children, Attributes, MetaInfo>. We recursively parse the report, when we descend, we parse data to Node, and when we ascend, we collect information to create MetaInfo.

   For each **node type**, we have one MetaInfo object and we do not distinguish the level of them. It is said that if we have a Line node in a Class node and a Line node in a Method node, the MetaInfo object of them is the same. The structure of MetaInfo is <Type, Renders, Nodes>. It also has some important methods we will describe in the following content.

   We use MetaInfo to index and render nodes, we use tree to search nodes.

4. Adapt
   We have two approaches to implement an Adapter. One is to write Java code, the other is by writing script config file.

   In the first approach we should implement AbstractAdapter class, and implement a static factory class as a subclass of Descriptor<AbstractAdapter>. In order to detect it automatically, we also should also put an @Extension annotation on factory class. In this step, we mainly do these things.
   Adding renders to MetaInfo. The render is an interface which tells Plugin how to render the content of a node. It contains two methods getContent(Node) and isRestrict(Node). We use the second to restrict the node which should not apply to this render. We only add UI render to MetaInfo renders list, and implement several UI Render subclass as the default implementation. In UI step, Jelly will use the default layout that we provide to show data if the render is default implementation. Also, we provide a custom UI interface to make the rendering step more extensive.
   Adding extra Attribute to nodes, We can add an extra attribute with a name to nodes by MetaInfo. We provide a Render and an attribute name, the MetaInfo will apply this render to its nodes.

Assigning primary key for a node type. We can set an attribute of a node type as a primary key so that we can show several code coverage results on one page. For example, if we use Cobertura and Jacoco, it will show the results of them on one page as the node type and node primary key is the same.
Grouping node with a key to create a new MetaInfo. we can group nodes by key, so that all nodes have the same key will as a child add to a new node(we do not change the structure of node). The children of new node is the nodes which have the same key, the parent is the parent of child nodes of new node. Child nodes save the reference in its attribute, parent node also save this reference. Then, we create a MetaInfo to save these new nodes.

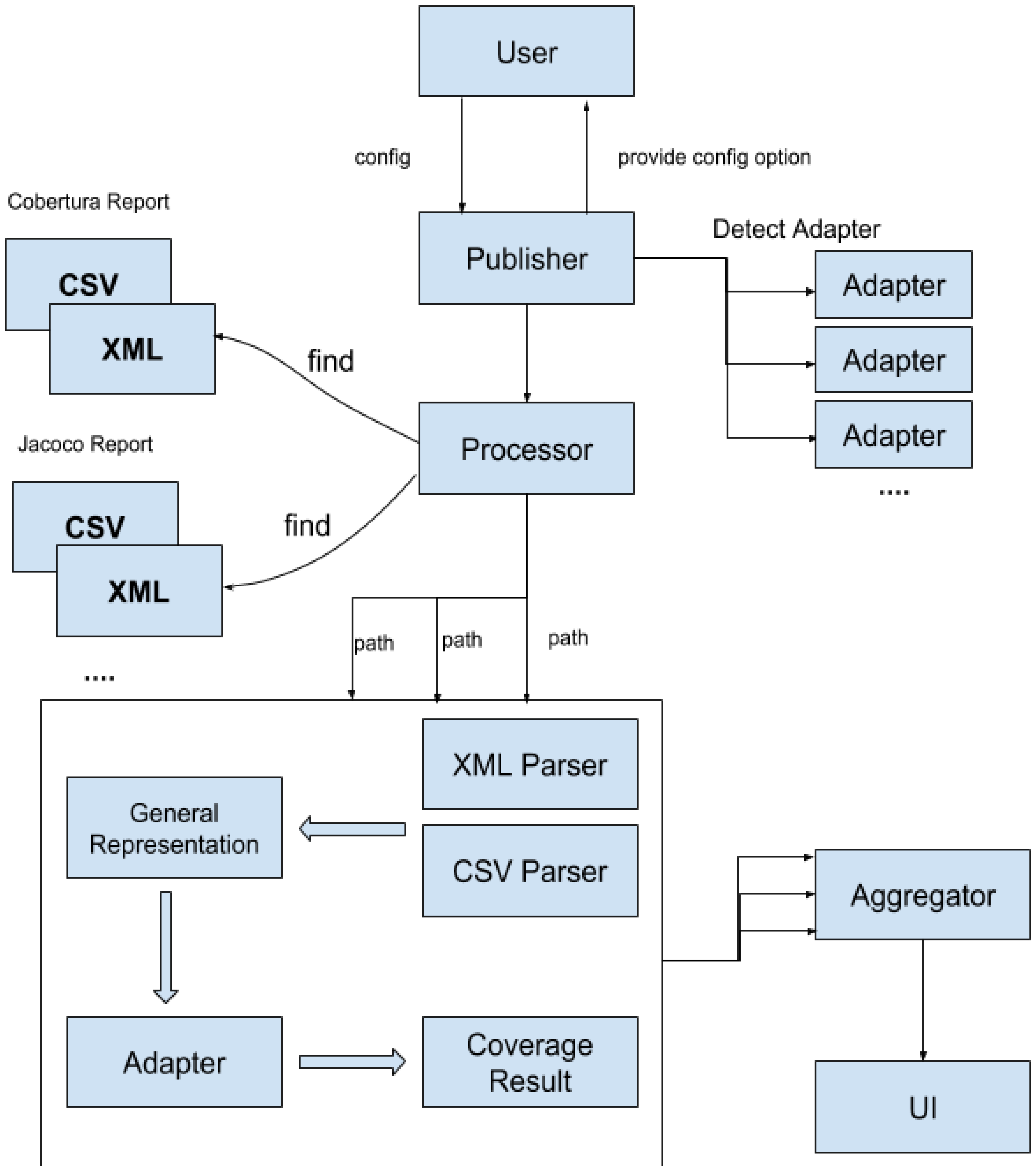
The second is script approach. This is not suitable for the complicated situation and is optional in this proposal(remain to discuss with mentors in the future).

5. Aggregate
   In this step, we merge results which come from the adapters. We use an Aggregator store all trees and meta info from the previous steps by wrapping them in a CoverageResult object, and we can show several code coverage tool results in the same page by using primary key and level we have set in adapt step.

6. UI
   We use MetaInfo to index node and we only show node which have renders. Jelly will iterate the renders of node MetaInfo. If it is default UI render, Jelly will show it in correspond view component. If it is custom UI render, Jelly will show its render value directly.

   The default UI renders contains AggregateRender, SourceCodeRender and ChartRender etc.

Simple Diagram for Synopsis



Benefits to Community

This Plugin will provide an easy-to-use interface which others can implement a code coverage tool by simply implementing an Adapter. The most repeated work like parse and render will all be done by this plugin, so the other people only need to care about two things -

what data should we show and how do we show - to in order to implement a coverage tool. For example, if we want to add Cobertura to this plugin, we only do these things as follows:

1. Group class nodes by filename to create a MetaInfo object, set file as its type and add this MetaInfo to MetaInfo list.
2. Set “name” attribute as primary key in MetaInfo of method, class, file, package.
3. Add an “aggregate” attribute like aggregate results of cobertura plugin to methods, classes, files, packages and report.
4. Assign an aggregate render to MetaInfo of method, class, file, package and report.
5. Add source code render to file MetaInfo.

If we properly design default renders and MetaInfo, we can implement a code coverage tool simple like above.

Deliverables

- An code coverage plugin for Jenkins
- Two Parsers, one parse XML file and the other parse CSV file. They both produce the same results.
- An generic Adapter interface and a component which detects Adapters.
- A Processor that controls Parser, Adapter, and Aggregator.
- Several default UI Renders.
- An Aggregator which can aggregate all results from adaper.
- A build step config page.
- Support multiple invocations.
- A health threshold for each code coverage tool.
- A health threahold for programming language (optional).
- Support to use script to implement Adapter(optional)
- Multi-Thread mode(optional)

Timeline

Date	Work
Prior - May 14	• Discuss with mentor to work out more details about this plan. • Familiarize with the code and the community, the version control system, the documentation and test system used.
May 14 - 30	• Write a parser for XML file and a parser for CSV file. • Design the interfaces of MetaInfo.
May 30 - June 14	• Design the interfaces of Adapter. • Write a class or function to find all Adapter implementation. • Implement Publish and Processor which is to decide the process of whole Plugin.
June 15 - July 12	• Design the interfaces of Render. • Write default UI renders and a custom UI render interface.

	• Write the adapter for Cobertura. • Implement Aggregator.
July 13 - August 6	• Integrate whole previous work to show the result of Cobertura report. • Discuss with mentor and implement more code coverage tool adapter.(Priority: Jacoco, OpenClover, Coverage.py, PHPUnit…) • Write Unit tests. • Fix Bugs
August 6 - 14	• Write user documentation. • Write developer documentation. Add some optional sugar feature by discussing with mentor.

Extra Information

Working Time

I will stay in Kaifeng, Henan, China during summer. So the timezone for me is GMT+8. Although the code period begins at May 14, I don't have much class to go to this year, so I can start to work earlier.

Reason for Participation

1. I want to be a good programmer, and this is a good opportunity for me
   In my school, I don't have many chances to take participate in a large project or formal development. This is a really good chance for me to experience modern and formal development. Also, by writing this plugin, I can learn many things which are interesting and useful.
2. I want to contribute to open source community, to world
   I have read the story of Linus Torvalds, the legend of Richard Stallman and the tales of many super genii in computer science. I can never be smart like them, but I can do things like them. I want to contribute to open source community. I can't imagine how excited I'm going to be if other people using the code I write, which must be the greatest thing in my life.
3. I want to improve my English skill
   My English is not very good, this is also a chance for me to improve my English communication skill.

I am looking forward to working on the project with Jenkins.