Refactoring Hudson God Class

programming help sites

JCG partners, recently shared their experience in an attempt to reorganize Hudson, the main class of the famous Continuous Integration server (CI), which is now renamed Jenkins.Let's see what he will tell about this refactoring experience.

We tried to hold Hudson.java refactoring, but unsuccessfully; Only later I was able to successfully spend a refactoring, thanks to the experience with the first attempt and more time. In any case, it was an excellent opportunity for learning. The lessons are learned

The two most important things we learned:
Never underestimate outdated code. It is more complicated and intertwined than you expect, and in its sleeves more unpleasant surprises than you can imagine.
Never underestimate outdated code.

And one more important point: when you are tired and fell into depression, you enjoy reading the "best comments in the history" in stackoverflow :-). Seeing someone else's suffering, it seems that his own becomes less.

I also started thinking that the refactoring process should be more stringent to protect you from excessive deviation from its initial goal and from the loss in the permanent cycle of the correction of anything <-> the detection of new problems. People tend to make refactoring changes in the depth, which can easily knock them down from the way, far from what they really need; It is important to periodically stop and look at where we are where we are trying to get and whether we are not lost, and you do not need to simply reduce the current "branch" of refactorings and come back to some earlier point and try, perhaps a completely different solution. I believe that one of the key advantages of the Mikado method is that it provides you with this global review - which is easily lost when it is only in your head - and with kickback points. Evil Heritage Code

For God's sake, use the Dependency Implementation System! Singletones and their manual search really complicate testing and affect the flexibility of the code.

Do not use open fields. They really make it difficult to replace the class interface.

The reflection and multithreading are made quite difficult, if not impossible, identifying the dependences of a particular code fragment and, therefore, the consequences of its change. I would hardly recognize all the places where Hudson.getInstance is called while its designer works. Our way to failure and success

There are many refactoring methods that can be performed using Hudson.java, since this is a

typical class of God, which additionally spreads its tentacles throughout the base of the code through its evil one-element instance, which is used by anyone for a wide variety of purposes. Goyko describes some problems that it is worth eliminating. Failure

We tried to start with small and "normalize" the initialization of Singlton, which is not done in the factory method, but in the designer itself. I didn't choose the goal very well, as it does not bring much benefit. The idea was to make it possible to have potentially and other Hudson implementations - for example, MockHudson - but in relation to the code of the code it was not really feasible, and even if it were so simple hudson.setinstance, it would probably be enough. In any case, we tried to create a factory method and transfer the initialization of the Singlton instance there, but at the end we lost in the problems of the parallelism: there were either several Hudson instances, or the application itself was blocked. We tried to move the fragments of the code, but dependences would not allow us to do it.

Success

Reflecting on our mistake, I came to the conclusion that the problem was that Hudson.getInstance () is called (many times) during the execution of the Hudson designer with objects that are used there, and flows start from there. This, of course, disgusting practice is to access the pollen copy to its complete initialization. The solution then simple: to be able to initialize the one-element field outside the constructor, we must delete all getInstance calls from its context.

Steps are clearly visible from the corresponding Commits of GitHub.Summary:

1. I used a refactoring "Introducing the factory" on the constructor
2. I changed ProxyConfiguration to not use GetInstance, and expect that the root directory will be installed before its first use.
3. I moved the code that did not need to run out of the designer, in a new factory method - it led to some, I hope the minor, codes reordering
4. Finally, I also transferred the initialization of the instance to the factory method

I can't be 100% sure that the resulting code has such a semantics, as it is important, because I had to make several changes outside the safe automated refactoring, and there were no useful tests, except for attempts to start the application (and, how It often happens with inherited applications, it was impossible to create them in advance).

The reorganized code does not yet give a large value added, but this is a good start for further refactorings (which I will not have time to try :-(), he got rid of offensive use of the instance during his work. Created and the designer code is easier and better. Exercise occupied I have about four pomodoros, that is, a little less than two hours.

If I had time, I would continue to remove the interface from Hudson, transferring its unrelated duties to its own classes (possibly saving methods in Hudson for backward compatibility and delegating these objects), and I could even use some AOP magic to get more Pure code while maintaining

binary compatibility (as Hudson / Jenkins actually do).

Try yourself!

Tune

to get the code

Get code as .zip or via Git:

one

2.

3.

Git@github.com: iterate / coding-dojo.git # 50MB => Takes a while

CD Coding-dojo

Git Checkout -B Mybranch Initial

Compile the code

As described in README dojo.

Run Jenkins / Hudson

one

2.

3.

CD Coding-Dojo / 2011-04-26-refactoring_hudson /

CD Maven-plugin; MVN Install; CD .. # anessary dependency

CD Hudson / War; MVN Hudson-Dev: Run

And go to http: // Localhost: 8080 / (Jetty must automatically select changes for class files).

Further refactorings

If you are an adventure lover, you can try to improve the code, delivering the individual duties of the gods. I would do this:
Remove the interface from Hudson and use it everywhere where it is possible.
Move the associated methods and fields into your own (nested) classes, the original Hudson methods are simply delegated to them (the refactoring of the move method must be useful); eg:

- Management of extensions and descriptors
- Authentication and authorization
- Cluster Management
- application level functionality (control methods, such as restarting, updating configurations, control of socket listeners)
-Ui controller (for this you need to reconfigure the stapler)
Convert nested classes to top-level classes
Provide a method for obtaining instances of classes without Hudson, for example, in the form of singletons
If possible, use individual classes instead of Hudson so that other classes depend only on the functionality that they really need, and not from all Hudson to learn about Jenkins / Hudson

If you want to understand the mode about what Hudson does and how it works, you can check out:
Hudzon architecture and may continue with
Hudson Remote Access API
Expansion points
Providing data remote API
Gudson building
Introduction to the Stapler user interface (its key feature is that it cleverly displays the URLs in the object hierarchy [and browsing files and action methods]), you may also check the link to Stapler
Sidenote: Hudson against Jenkins

Once there was a continuous integration server called Hudson, but after the death of his patron Sun, he found himself in the hands of a person named Oracle. He was not very good in communicating, and no one knew that he was conceived, so when he began to behave a little strange - or at least so understood Hadson's friends - those who worried about the future of Hudson (including most people Originally working in the project) made his clone and called it Jenkins, which is another popular name for the butler. So now we have Hudson with the support of Oracle and the guys from Sonatype and Jenkins with the support of the bright community. This exercise is based on the source code of Jenkins, but to maintain a low level of confusion, I often call it Hudson, because the package is so called the main class. Output

Refactoring outdated code always turns out to be more complex and time consuming than you expect. It is important to follow a method, for example, the Mikado method, which helps you lead a global review where you want to go and where you are, and regularly think about what and why you do not get lost to the series to correct the problem - new problems are detected Steps. It is important to understand when you need to surrender and try another approach. It is also very difficult or impossible to write tests for change, so you must be very careful (maximize the use of safe,

automated refactorings and act in small steps), but fear should not prevent you from trying to save the code from the decay.

Link: What I learned from (almost) failures in Hudson refactoring from our JCG partner in Holy Java blog.

Articles on the topic:
Java Code Geeks Andygene Web Archetype
Why automated tests accelerate your development
The quality of the code matters for customers. Many.
Using FindBugs to create a significantly smaller number of erroneous code
Recommendations for the development of flexible software for users and new users