Code Review Process with GitHub

Effective Code Review

If you're working on a software project with more than one person, code review is a necessary piece of a healthy workflow.

What

Code review is the process of having another human being read over a diff (new code added since the previous review). It's important to note that code review is about code. Code review doesn't mean an architecture review, a system design review, or anything like that.

Why

Why should you do code review? It's got a few benefits:

- By forcing someone else to have the familiarity to review a piece of code you guarantee that at least two people understand it.
- It ensures readability. By getting someone else to provide feedback based on *reading*, rather than *writing*, the code you verify that the code is readable, and give an opportunity for someone with fresh eyes to suggest improvements.
- It catches bugs. By getting more eyes on a piece of code, you increase the chances that someone will notice a bug before it manifests itself in production. This is in keeping with Eric Raymond's maxim that, "given enough eyeballs, all bugs are shallow".
- It encourages a healthy engineering culture. Feedback is important for engineers to grow in their jobs. By having a culture of "everyone's code gets reviewed" you promote a culture of positive, constructive feedback. In teams without review processes, or where reviews are infrequent, code review tends to be a tool for criticism, rather than learning and growth.

What should a code reviewer be looking for?

- Intent - What change is the author trying to make, is the bug they're fixing really a bug? Is the feature they're adding one we want?
- Architecture - Are they making the change in the right place? Did they change the HTML when really the CSS was busted?
- Implementation - Does the change do what it says? Is it possibly introducing new bugs? Does it have documentation and tests? This is the nitty-gritty of code review.
    - clarity
    - performance
    - complexity
    - impact on other modules/systems
    - duplication
    - deployment issues

- Grammar - The little things. Does this variable need a better name? Should that be a keyword argument?

What will a code reviewer produce?

There are three different types of review elements:

- TODOs: These are things which must be addressed before the commit can be merged; for example a bug in the code
- Questions: These are things which must be addressed, but don't necessarily require any changes; for example, "Doesn't this class already exist in the stdlib?"
- Suggestions for follow up: Sometimes you'll want to suggest a change, but it's big, or not strictly related to the current commit, and can be done separately. You should still mention these as a part of a review in case the author wants to adjust anything as a result.

Generic Checklist for Code Reviews

Structure

❏ Does the code completely and correctly implement the design?
❏ Does the code conform to any pertinent coding standards?
❏ Is the code well-structured, consistent in style, and consistently formatted?
❏ Are there any uncalled or unneeded procedures or any unreachable code?
❏ Are there any leftover stubs or test routines in the code?
❏ Can any code be replaced by calls to external reusable components or library functions?
❏ Are there any blocks of repeated code that could be condensed into a single procedure?
❏ Is storage use efficient?
❏ Are symbolics used rather than “magic number” constants or string constants?
❏ Are any modules excessively complex and should be restructured or split into multiple routines?
❏ Are things that are likely to change (e.g. tax rates, displayed strings etc) modifiable without editing code (i.e. configuration file)? Is the correct config file being used?

Documentation

❏ Is the code clearly and adequately documented with an easy-to-maintain commenting style?
❏ Are all comments consistent with the code?
❏ Are all assumptions & reasons for unusual code commented (e.g undocumented API bug)?

Variables

❏ Are all variables properly defined with meaningful, consistent, and clear names?
❏ Do all assigned variables have proper type consistency or casting?
❏ Are there any redundant or unused variables?

Arithmetic Operations

❏ Does the code avoid comparing floating-point numbers for equality?
❏ Does the code systematically prevent rounding errors?
❏ Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
❏ Are divisors tested for zero or noise?

Loops and Branches

❑ Are all loops, branches, and logic constructs complete, correct, and properly nested?
❑ Are the most common cases tested first in IF- -ELSEIF chains?
❑ Are all cases covered in an IF- -ELSEIF or CASE block, including ELSE or DEFAULT clauses?
❑ Does every case statement have a default?
❑ Are loop termination conditions obvious and invariably achievable?
❑ Are indexes or subscripts properly initialized, just prior to the loop?
❑ Can any statements that are enclosed within loops be placed outside the loops?
❑ Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?

Defensive Programming

❑ Are constants always on lefthand side of equality operators?
❑ Are indexes, pointers, and subscripts tested against array, record, or file bounds?
❑ Are imported data and input arguments tested for existance, validity and completeness?
❑ Are all output variables assigned?
❑ Are the correct data operated on in each statement?
❑ Is every memory allocation deallocated?
❑ Are timeouts or error traps used for external device accesses?
❑ Are files checked for existence before attempting to access them?
❑ Are all files and devices are left in the correct state upon program termination?

Using pull requests for code reviews

Creating the Pull request in github

You can think of pull requests as a discussion dedicated to a particular branch.For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they'll be able to see the question right next to the relevant commits. A pull request appears in github as an issue.

Pull requests let you tell others about changes you've pushed to a GitHub repository. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.
This guide walks through the process of sending a hypothetical pull request and using the various code review and management tools to take the change to completion.

Pull requests can be sent from any branch or commit but it's recommended that a feature branch be used so that follow-up commits can be pushed to update the pull request if necessary.

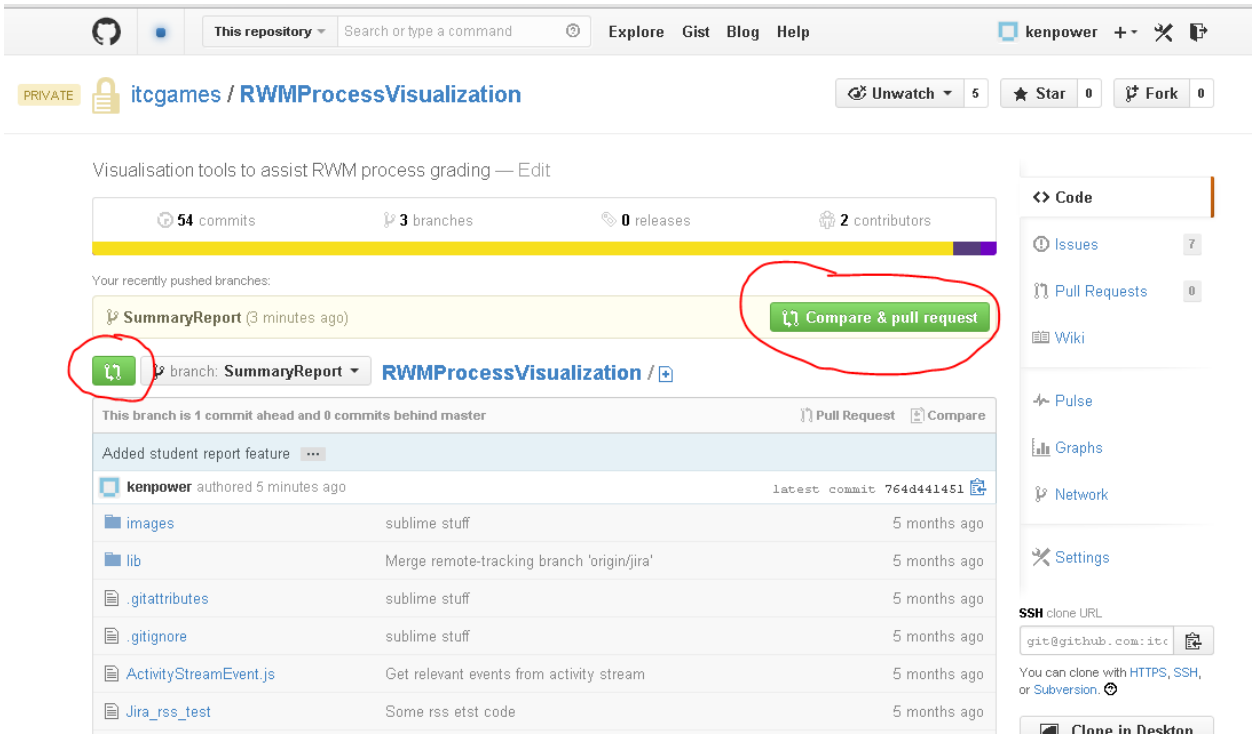
Initiating the pull request

In the following example, kenpower (the "coder") has completed some work on a fork of the itcgames/RWMProcessVisualization repository, pushed a commit to a "SummaryReport" branch, and would like someone to review before merging.

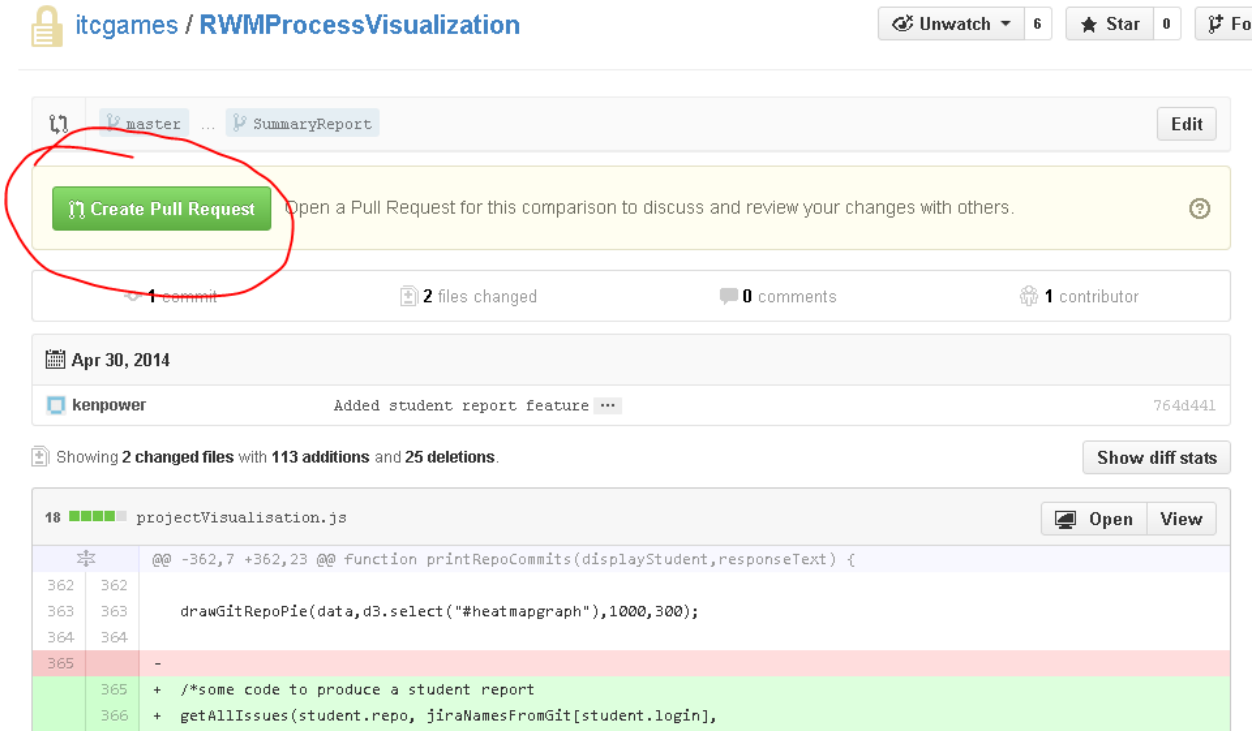
Before creating a pull request, the coder should;

1. check their code against the review checklist
2. make sure his branch has been pushed to the central repository (ie. git push)

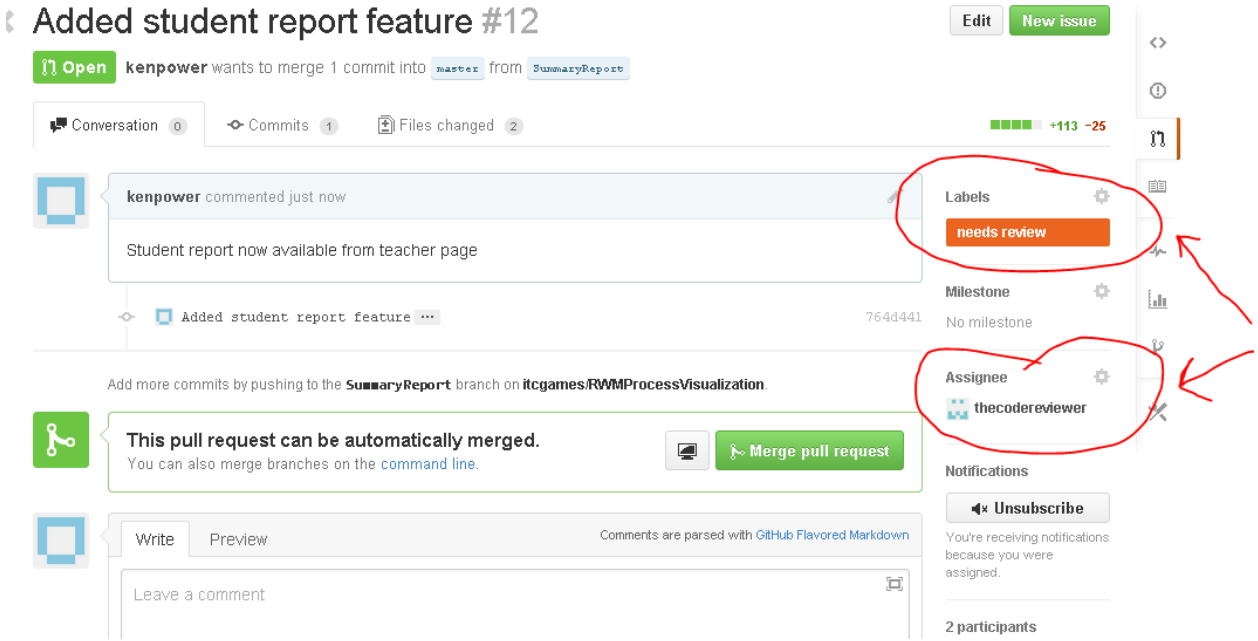
In GitHub the coder will navigate to his repository & branch which needs merging and press the "Pull Request button".



The “compare & pull request” button shows the changes that have been made and present a “create pull request button”, press this to create the pull request:

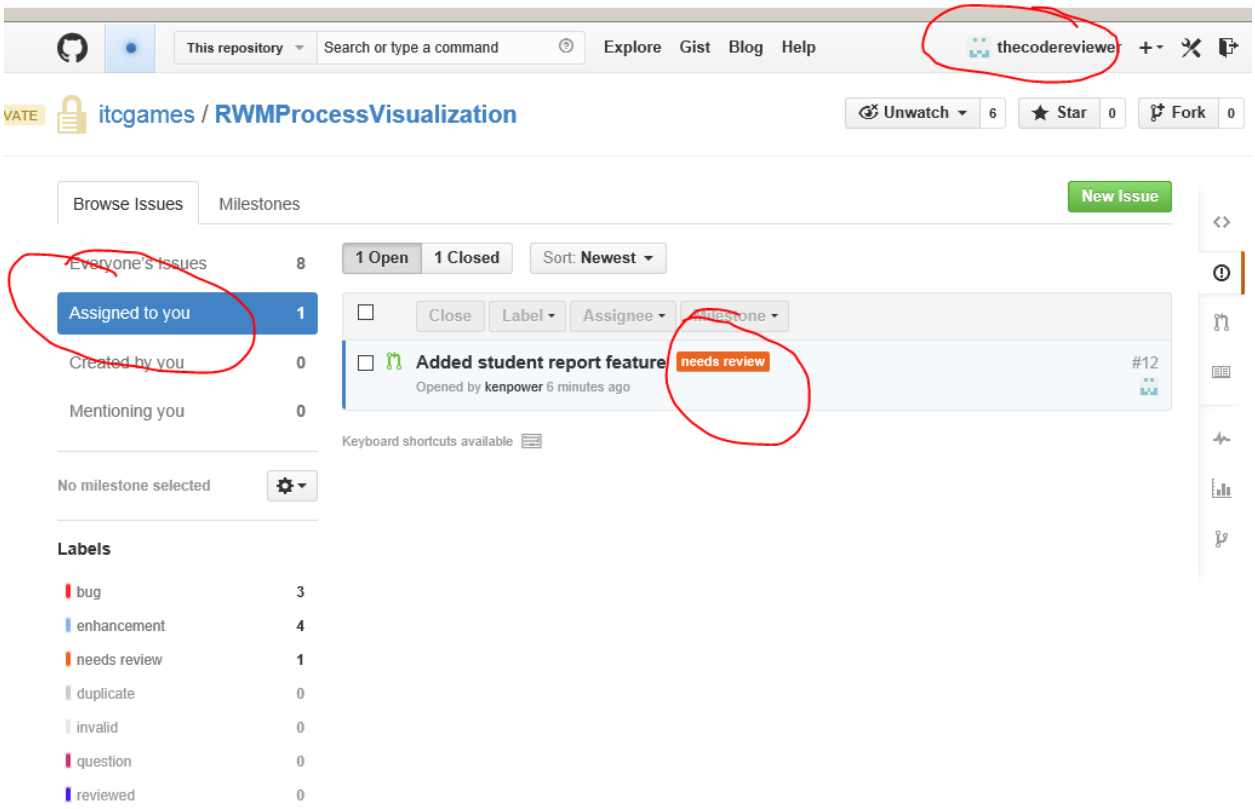


When creating a pull request, you should label it with “needs review” and assign it to the person who is to review it (in this case the reviewer is coincidentally called “thecodereviewer”)

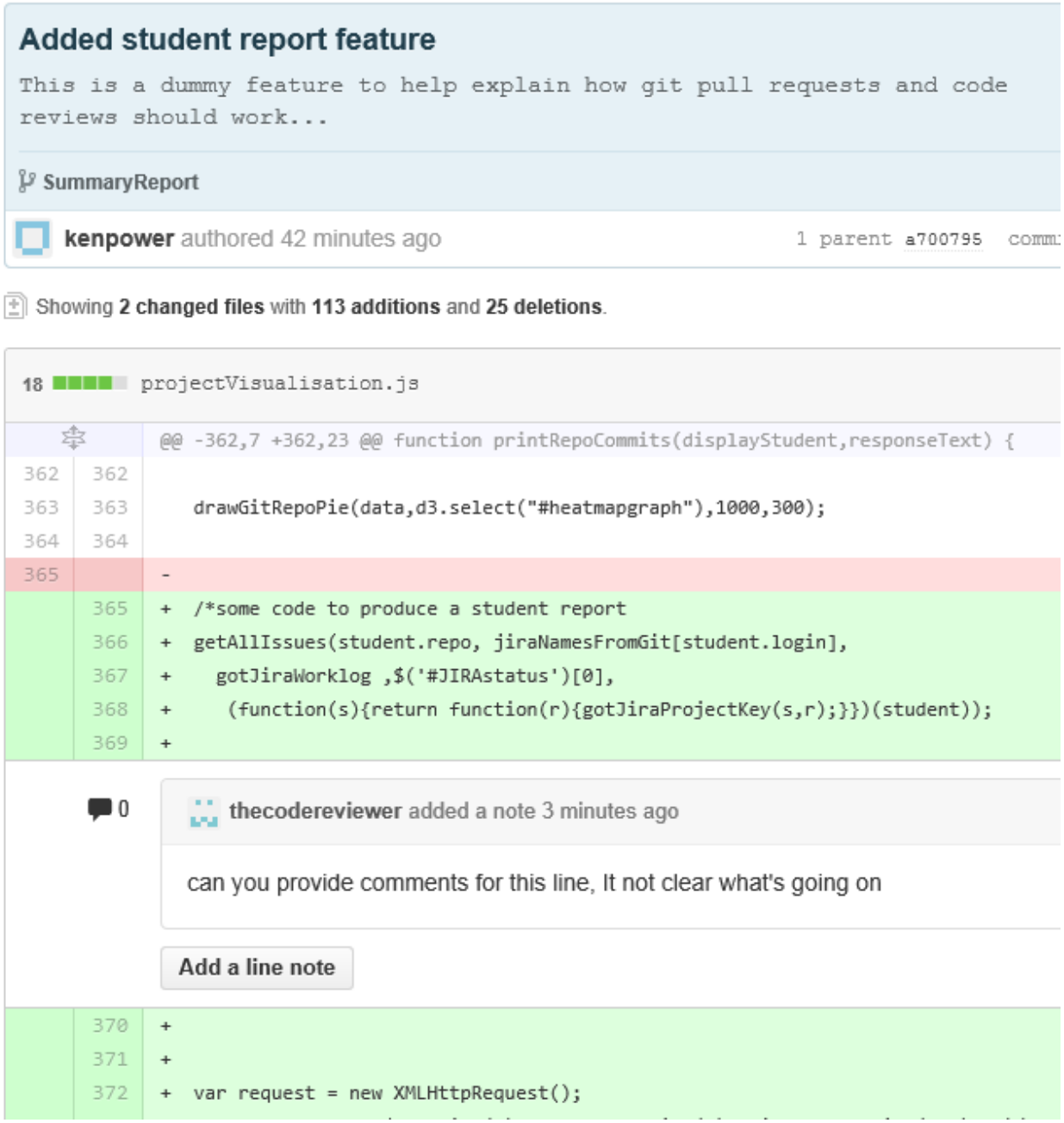


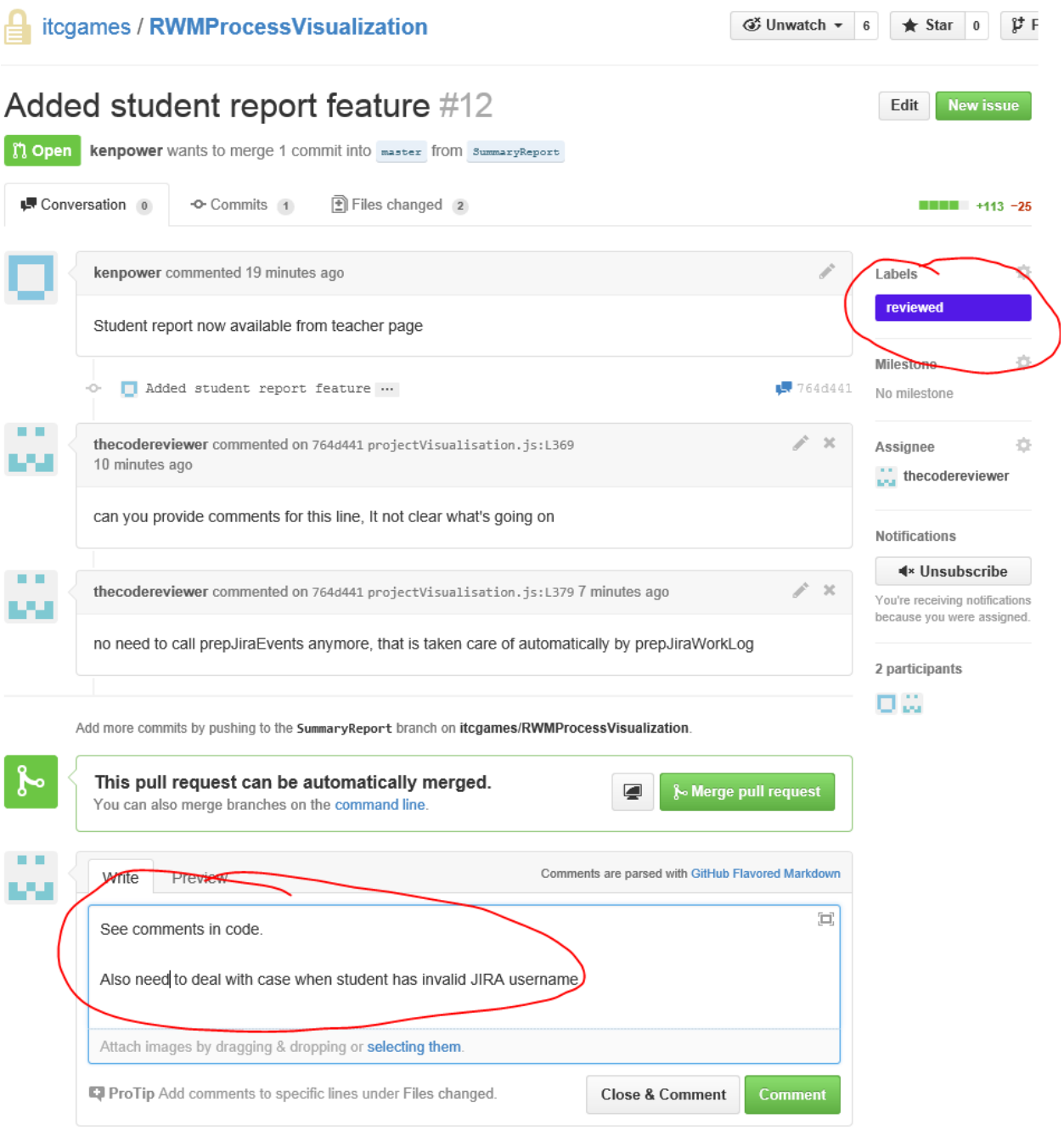
Performing the code review

The code reviewer will see your pull request in his/her issues list with the “needs review” label



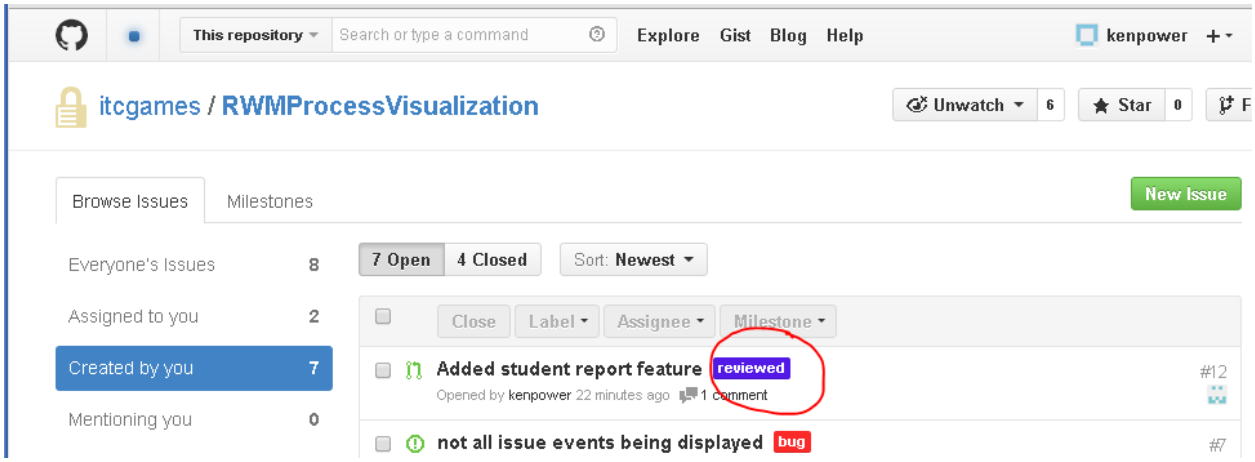
The reviewer can view changes introduced by the commit and comment on individual lines or comment on the commit as a whole





Once the review is complete, the reviewer will label the pull request as “reviewed” and remove the “needs reviewing” label.

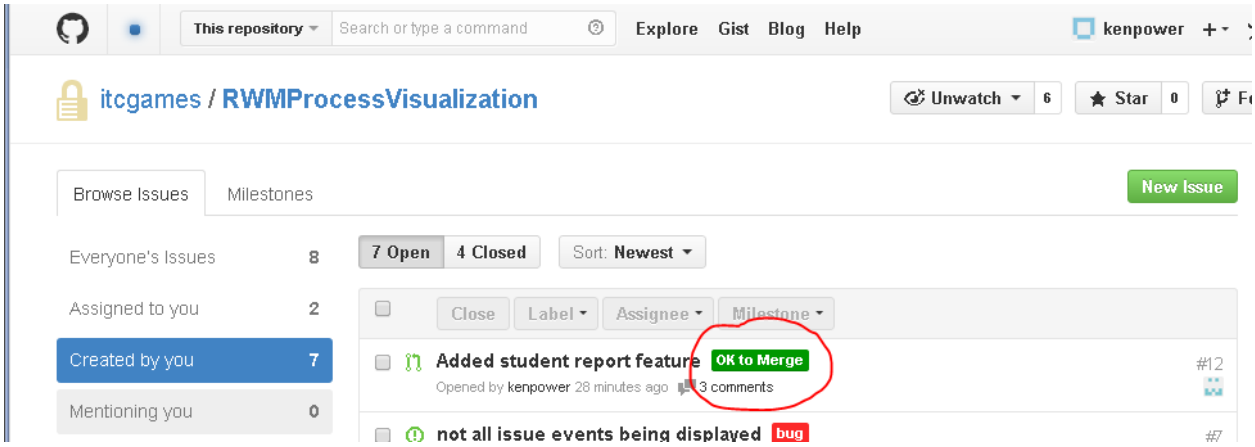
Coder responds to the reviewers comments



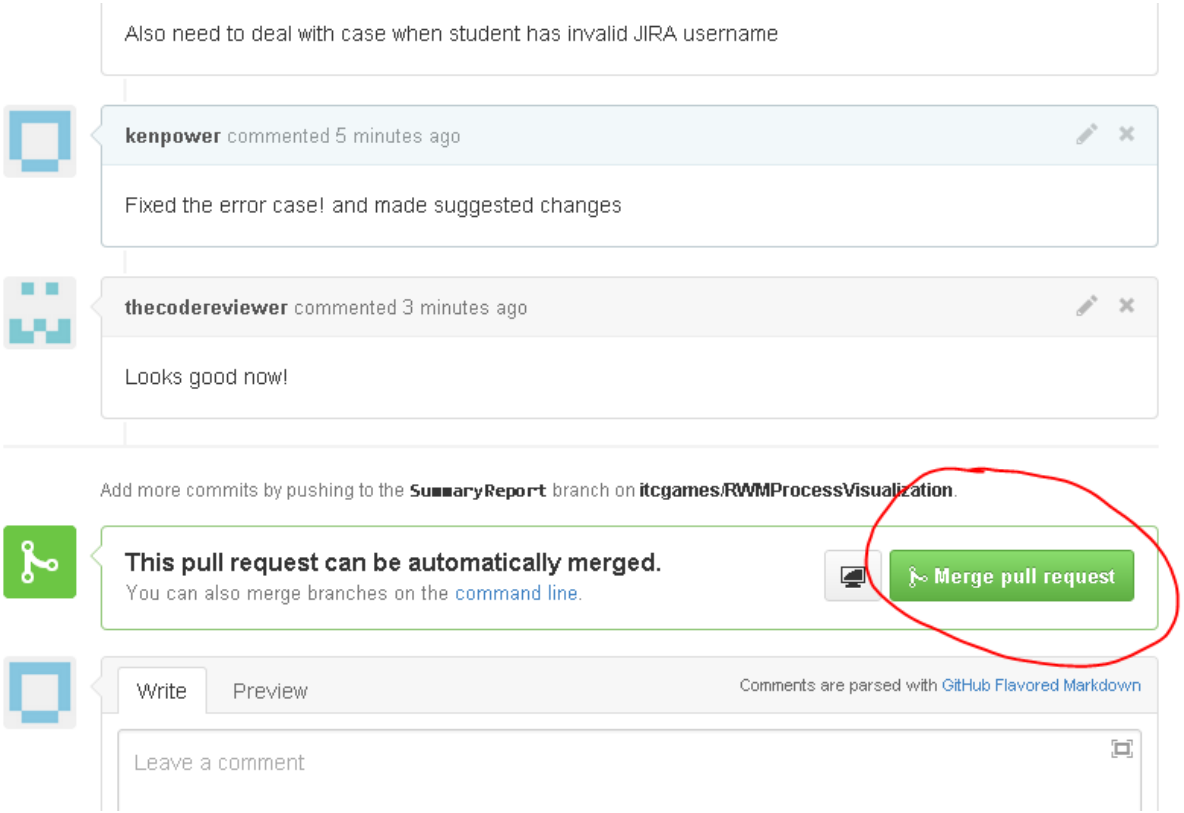
The original committer will see the request as “reviewed”, at that point he can examine the comments and make the suggested changes and commit the changes to the same branch. He will then re-label the original pull request as “needs review” (again).

The reviewer will see the amended pull request and review to see if he is happy with the new changes, if not or if he see something else to change, he’ll add more comments and sent it back to the coder by labeling it “reviewed”.

The request goes back and forth between the coder and the reviewer until the reviewer is satisfied. When the reviewer thinks the branch is ready to merge into the main branch, the reviewer will label it “ok to merge”.



When the coder sees that the pull request is “OK to merge”, he will merge it. Once a pull request is accepted, the actual act of publishing a feature is much the same as in the normal git workflow. First, you need to make sure your local master is synchronized with the upstream master. Then, you merge the feature branch into master and push the updated master back to the central repository.
if the merge can be done automatically by github, no problem. If there are merge conflicts then the coder and reviewer should sit down together to manually resolve the conflicts



Also need to deal with case when student has invalid JIRA username

kenpower commented 6 minutes ago

Fixed the error case! and made suggested changes

thecodereviewer commented 4 minutes ago

Looks good now!

kenpower merged commit **bbacc16** into `master` from `SummaryReport` just now

kenpower closed this just now

Pull request successfully merged and closed

You're all set—the `SummaryReport` branch can be safely deleted.

Delete branch

Write Preview

Comments are parsed with GitHub Flavored Markdown