Wine Hacking
Or how to get your MRs upstreamed

Huw Davies
Upstreaming MRs

It must be simpler than navigating the Minneapolis Skyway!
Why review?
We'd like to avoid

<table>
<thead>
<tr>
<th>COMMENT</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created main loop &amp; timing control</td>
<td>14 hours ago</td>
</tr>
<tr>
<td>Enabled config file parsing</td>
<td>9 hours ago</td>
</tr>
<tr>
<td>Misc bug fixes</td>
<td>5 hours ago</td>
</tr>
<tr>
<td>Code additions/edits</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>More code</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>Here have code</td>
<td>4 hours ago</td>
</tr>
<tr>
<td>AAAAAAAAA</td>
<td>3 hours ago</td>
</tr>
<tr>
<td>ADKFJSLKDFJSDLFJ</td>
<td>3 hours ago</td>
</tr>
<tr>
<td>My hands are typing words</td>
<td>2 hours ago</td>
</tr>
<tr>
<td>HAAAAAAAAANDS</td>
<td>2 hours ago</td>
</tr>
</tbody>
</table>

As a project drags on, my git commit messages get less and less informative.

https://xkcd.com/1296/
Who's who

• Will involve interaction between
  • You, the author
  • At least one reviewer
Your rôle

• Your job is to explain your code to the reviewer
• The easiest way to do this is to keep the code simple!
• You are the world's expert on your piece of code
• Do not assume that the reviewer is also an expert
  The reviewer might not be a maintainer
Reviewer's rôle

• To help!
• Will either
  • Approve MR - yay!
  • Provide constructive feedback
• The aim is to produce a high quality contribution
Possible workflow

do
{
    research_and_code();
sleep();
    if (self_review() != ok) continue;
    submit();
    upstreamed = wait_feedback();
} while (!upstreamed);
Possible workflow

do
{
    research_and_code();
sleep();
    if (self_review() != ok) continue;
    submit();
    upstreamed = wait_feedback();
} while (!upstreamed);
Easy things to get right

Whitespace

- Match formatting to surrounding code
- Usually 4-space indents
- Watch out for tabs and end-of-line whitespace
- Do add spaces either side of binary ops

  \((x \geq y + 1)\) rather than \((x \geq y + 1)\)
More easy things to get right

- Generally we prefer `snake_case` over `CamelCase`
- Likewise `name` over `lpszName`
- https://wiki.winehq.org/Submitting_Patches
Keep each commit small

Each commit should be as small as possible

Diff stats like this don't encourage the reviewer

<table>
<thead>
<tr>
<th>File</th>
<th>Changes</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlls/foo/bar.c</td>
<td>345</td>
<td>++++++++</td>
</tr>
<tr>
<td>dlls/foo/tests/bar.c</td>
<td>456</td>
<td>+++++++++++++++++++</td>
</tr>
</tbody>
</table>
How to keep commits small

- One idea per commit
- The overall feature doesn't need to work in one go
- Refactor first then add new things
- It’s almost always possible to simplify things!
- Use helper functions
Helper functions
Can help to reduce commit size

• Implement as stub and flesh out in a later commit
• Reusing existing code? Move to helper first
• Can help when the control flow looks awkward
• However don't add a helper before calling it (dead code)
Write tests!

Why?

- To show that your implementation is correct
  This in turn helps explain your change
- To prevent future regressions
Write tests!

How?

• Ideally add them at the start of the MR with todo_wine

  Then remove the todo_wine in the implementation's commit

• Otherwise they can go in at the end of the MR

• Try to keep them simple too

• Make sure they pass after each commit!
Commit message 1

• Write in the imperative:

  "foo: Make x do y." rather than "foo: This makes x do y."

• Keep it short

• Avoid things like "Fix blah"

• Generally the word "Also" means you can split the patch
Commit message 2

- A more detailed explanation can follow on subsequent lines
- Include any relevant `Wine-Bug:` tag
- Update if the code has changed
- Can take longer to write a good commit msg than the code itself!
Possible workflow

do
{
    research_and_code();
    sleep();
    if (self_review() != ok) continue;
    submit();
    upstreamed = wait_feedback();
} while (!upstreamed);
Why the `sleep()`?

- To allow you to context switch
- You'll come back with a fresh prospective
- An actual sleep isn't a bad idea!
- Also prevents the reviewer being swamped
  
  "Huw is not a compiler" [1]

Possible workflow

do{
    research_and_code();
sleep();
    if (self_review() != ok) continue;
submit();
    upstreamed = wait_feedback();
} while (!upstreamed);
Self review 1
Global overview

Look at the patch in its entirety

• Does commit msg make sense?
• Does formatting match?
• Check frees / releases
• Can control flow be simplified?
Self review 2
Local overview

Look at each line of code carefully

• Is it doing what you think?
• Is it necessary?
• Can it be split?
Self review 3

• Keep in mind all comments already received
  Even from earlier versions
• If you find it hard, think about the reviewer and simplify!
• Practise reviewing other people's code
Possible workflow

do {
    research_and_code();
    sleep();
    if (self_review() != ok) continue;
    submit();
    upstreamed = wait_feedback();
} while (!upstreamed);
Possible workflow

do{
research_and_code();
sleep();
if (self_review() != ok) continue;
submit();
upstreamed = wait_feedback();
} while (!upstreamed);
Merge Requests

Size

- Keep the number of commits per MR below around five
- It's fine to split your work over several MRs
- This keeps things manageable for the reviewer
- A change in an early commit doesn't require updating loads of commits
Merge Requests
Mechanics

• Wait for the first MR to be merged before sending the next
• To preserve the discussion trail
  • Push updated commits to the same MR - don't create a new one
  • When splitting, mention the original MR in new one
Possible workflow

```c
do
{
    research_and_code();
    sleep();
    if (self_review() != ok) continue;
    submit();
    upstreamed = wait_feedback();
}
while (!upstreamed);
```
How to respond to feedback

• Read and digest all of the comments
• If you don't understand, think
• If you still don't understand, ask
• Ensure that you address all of the comments
• One comment may apply to several similar issues
• It's not a race! Don't send the next version immediately
But my MR didn't get any feedback!

- Your work is likely too complicated or not obviously correct
- Feel free to ask for an update
- You can now assign a reviewer yourself
Possible workflow

do
{
    research_and_code();
sleep();
    if (self_review() != ok) continue;
    submit();
    upstreamed = wait_feedback();
} while (!upstreamed);
finalize();
Success!

- Update any bugs
- Party!
Conclusions

• Keep everything simple!
• Address all feedback
• Take your time!
• Good luck!