Code Reviews

This is a list of thing to look for when code reviewing. Code reviews involve a certain amount of subjective opinion so to some extent this is just some things to be on the lookout for but the reviewer will have to use some judgement.

Coding Standards.

- We should be following certain coding standards, however, a number of the existing standards I'm not overly bothered about. Too many times reviewers see their main job to be enforcing what are, in the grand scheme of things, relatively minor stuff - syntax and certain formatting conventions. Some stuff is more important than others - for myself I will insist on useful indentation and not all sorts of random indents, as that is a massive readability issue which can result in someone misunderstanding some scoping logic. That is not to say that we should ignore coding standards in a code review, but most of the time this is not even close to the important thing.

Tests.

- Make sure the code that has been added (or changed) has been adequately tested, that it appears that all interesting conditional code looks like it is covered, that there are useful asserts that check for important outcomes.
- Where possible require that tests are using mock classes to abstract away the database.
- Ensure the tests are named usefully, in particular don't allow names like 'positiveTest' and 'negativeTest' as generally positive/negative are meaningless to a future reader. Remember that when a test fails I want to see in the test run UI some hint of what was being tested. It also helps the code reviewer and the future developer to understand what has been tested already etc.
- Ensure that only 1 thing is being tested per test method. Many asserts are fine, but if there are unit tests for an invoice being posted there should be separate test methods for invoices being posted, not eligible for posting, or errors etc. Nothing is worse than trying to read down a large test to understand what behaviour is defined. It also helps with test runs as it means 1 fail doesn't stop all tests running to see what else fails. Where this can be relaxed is usually where there is a bulk handling test - the focus of such a test is often to check that different conditions work when many records are passed in. But in that case there should be the single unit tests as well.

Coding Patterns

- Understand the general patterns we want and ensure the developer has followed them to at least some useful degree. E.g. No logic in UI classes, no logic in Batches and other such stuff. DML is done via a unit of work type class and SOQL is done via some selector etc.
- Where the code is in Legacy areas this may be more awkward. We can't always insist on big sweeping changes from old 'bad' code, but we should insist on some improvement that moves us closer. E.g. if the legacy code is in a VF controller then at least insist it be moved into some sort of service or other standalone class with no UI specific stuff in it. If elsewhere then maybe ask them to use a unit of work for any existing DML. Basically we should always be looking for at least some refactor in Legacy code towards something that is better.

Bugs

- Whilst the code reviewer is not the QA person, you must look at any code etc with a QA hat at least partially hanging off your head. Always look at code etc and think how it can go wrong, demand fixes if there is not a good explanation as to why it won't happen. A few obvious ones:
  - Null pointers
  - List checking to avoid index out of bounds
  - SOQL to return a single object (unlike lists can fail at the query)
  - Map keys being case sensitive even though most other stuff isn't.
- Bulkification. a somewhat misused terms at times, but no matter how much someone says the end user only works on 1 record at a time we need to handle many records at once in some areas, e.g. triggers and reusable service code. Do not accept code that assumes 1 record only (e.g. only looks at the first record in a list).
- Static collections. Lists/Sets/Maps that are declared as static can be problematic, as not everyone understands their life cycle and how that interacts with certain SF mechanics. The most likely area they go wrong is code fired from triggers, as that code may be called many times without the developer thinking about it.
- Triggers. Code that is fired in a trigger has to understand that you have no control over how many times it may legitimately be called, as such reject attempts to 'avoid recursion' through simple flags, they do not work. The logic needs to be written to not get itself recursive and not rely on tricks like static flags to say it has already run.

Design

- Whilst code review is not the best time to validate a design approach to a ticket, it is still a better time than later on. If you think there is some design element that is really wrong then raise it - maybe pull in some comments from others if needed.
- Some specific things to look for:
  - Adding fields to objects, we often add fields to object types for the wrong reasoning.
  - Process Builders. We should be migrating away from these asap, I doubt we will have new ones, but even for older legacy ones we should be trying to switch away from them rather than maintain them.
  - Flows. Not as bad as Process builders, but look out for some things. We should not be using new flows to update records, reject them. For legacy flows, ensure we not allow flows that update a field on the same object that triggered a flow, unless it is a 'before flow'.
  - Any 'logic' that is hard coded to a person. Doesn't matter what type of logic, we do not want specific users being hard coded.

Error Handling

- This is not the same as looking for bugs. This is about checking how we handle error conditions - we don't want messages going out to the end user that leave them confused. Too many areas we have poor or even raw technical errors going out to the user, which do not easily hint and what they can do to fix a problem.

- Whilst not an end user issue, crucially we do not (usually) want partial commits. Any where a ‘try … catch … ’ (or finally) exists that catches an exception and swallows it then double check for any DML and whether there are savepoints and rollbacks.

Clean Code (to some extent this is coding standards)

- No code that is unused, if you see code that can’t be executed (often methods that are never called) then demand their removal.
- No commented out code. Some will say that we may want it later, don’t accept that. If code is commented out then it is not needed and just gets in the way - it interferes with easy reading and searches for logic etc. If someone wants something back they can add it then or pull it back from source control (assuming we don’t keep dropping the history).
- Proper method access. Any methods should be checked to see whether they are public/Global when they do not need to be. All methods should be private unless they are actually called from somewhere else.
- As our code is unmanaged we pretty much never need Global - always reject Global.
- Large methods. If a method gets to big request it is split up. Too big is subjective - a method that will require scrolling to read is probably too big. If a method clearly does more than 1 thing it is to big (even if it is fairly small).
- Pointless comments. If someone has copied and pasted, them make sure they haven’t copied an pasted a now wrong comment. If a comment just states the obvious then get it removed ('// now set X' just above a line that show 'something.setX();' is stating the obvious). Comments that explain reasoning are good, explaining the how is usually not useful unless it is some complex process that needs explaining.