# Software Evaluation: Tutorial-based Assessment

Mike Jackson, Steve Crouch and Rob Baxter

*Tutorial-based evaluation provides a pragmatic evaluation of usability of the software in the form of a reproducible record of experiences. This gives a developer a practical insight into how the software is approached and any potential technical barriers that prevent adoption.*

A tutorial-based evaluation is a reflection of your subjective experiences in learning, building, installing, configuring and using the client's software. The result is a pragmatic and candid report based on your experiences, explaining what you achieved with the software, following its documentation and using the related resources available e.g. web pages, tutorials, e-mail archives etc. It also includes the issues and shortcomings you observed, and can also include suggested workarounds and fixes for these issues. As a result, your experiences, when written up, can serve the clients as a practical guide for getting the software to work.

Your evaluation centres around carrying out typical tasks using the software. The nature of the tasks will depend upon the software and you should ask the client about any tasks they particularly want the evaluation to consider. Ideally, these should be representative of the core functionality of the software and its intended purpose. They should also cover the tasks of getting into a position whereby the software can be used e.g. downloading and installing it for users, or, for developers, setting up a development environment and getting the source code. If you've not already explicitly agreed upon the user classes from whose perspective the evaluation will be undertaken, then the tasks will implicitly determine these:

- User evaluations evaluate software from a user's perspective. This evaluation focuses on the usability of the software as-is without the need for writing code. This can include using a web portal, a command-line tool, or a graphical user interface.
- User-developer evaluations evaluate software from the perspective of a user who is developing against APIs offered by the software and which allow users to develop and use their own components with the software. This includes how easy it is to install, configure and use the software, set up a development environment, and develop and use components.
- Developer evaluations evaluate software from the perspective of someone who wishes to change the software itself e.g. to fix bugs, add features or recode components. An example of such a developer would be a new developer joining the original developers of the software.
- Member evaluations are the same as developer evaluations in that they evaluate software from the perspective of someone who wishes to change the existing software. However, their scope may be beyond that of a developer evaluation and cover certain resources not available to developers. This might apply for example, to a project that releases their code as open source but has many project resources available only to members of their project e.g. test scripts or additional source code.

Your evaluation consists of doing the tasks, following any available documentation in a logical (or suggested) fashion to achieve eventual successful use of the software. Successful use is defined as being able to execute the task(s) successfully. Specific directions about evaluations from each of these perspectives are given below.

You should agree at the outset with the client how you should ask for help, report issues and bugs. Ideally, you'll both agree that you'll use the same mechanisms as offered to actual users so if, for example, users must e-mail issues then do that, if developers use an issue tracker than ask for access to do that. This allows you both to assess how the client handles support but also to ensure the client

becomes aware of problems as they occur and, if, for example bug/issue trackers are used, for the client to have these logged.

## User perspective

The evaluation focuses on ease of download, installation and use of the software and any prerequisites, with no programming being required. Try and keep in mind the intended user and consider the following from their perspective, not yours e.g. you may know what a "heap corruption error" is or "select the canvas widget" means but the appearance of this error in a genome sequencing tool is most inappropriate!

At each stage in the task, consider the following usability guidelines[1]:

- Visibility of system status. Does it give users appropriate feedback within reasonable time?
- Match between system and the real world. Does it speak the user's language and make information appear in a natural and logical order? Are implementation-specific details hidden from the user?
- User control and freedom. Does it provide clearly marked exits, undo and redo?
- Consistency and standards. Is it consistent within the software, with other similar packages and with platform conventions?
- Error prevention. Does it prevent errors in the first place or help users avoid making them?
- Recognition rather than recall. Does it make objects, actions and options visible and reduce the amount of information a user has to remember?
- Flexibility and efficiency of use. Does it offer short-cuts and support macros for frequently-done action sequences?
- Aesthetic and minimalist design. Does it avoid showing irrelevant or rarely-needed information?
- Help users recognize, diagnose, and recover from errors. Does it make errors clear, comprehensible, precise and suggest solutions if possible?
- Help and documentation. Does it provide concise, accurate, clear, easily-searchable task-oriented doc centred around concrete lists of steps?

Explore what happens when instructions are not followed e.g. deliberately type in the wrong command, enter an illegal value etc. This allows assessment of the robustness of the software, its error reporting, how easy it is for users to recover etc. Think of it as your job to try and break the software!

For GUIs or web portals you should also:

- Assess the extent to which tasks can be achieved without any consultation of user doc or other material. Ideally, it should be clear from the interface alone how the user can achieve what they want to do and what the interface's responses mean in terms of progress through their task.
- Click everything! Every menu should be explored, every button pressed, every field clicked, every widget hovered over to check for tool tips, every window resized to check how it scales etc.

As part of your evaluation, consider supporting resources provided for users since these will form part of a user's experience (especially if they run into problems or have queries). Consider:

[1] These are heuristic evaluation guidelines derived from principles of UI design. They can act as a checklist when using the software as to how usable it is and how its usability can be improved. See http://en.wikipedia.org/wiki/Heuristic_evaluation.

- Release packaging
  - Are the binary releases packaged for immediate use in a suitable archive format, e.g. .tar.gz, .zip or .jar for Linux, .zip, .jar or .exe for Windows?
  - Is it clear how to get the software from the web site? Does it have version numbers? Is it clear from the web site or user doc what other packages are required?
  - Is it clear what the licencing and copyright is on the web site?
- How to get started
  - If the package is a .exe or .jar, is it clear from the web site how to start using it?
  - If the package is a .zip or .tar or .gz is it clear once it's unzipped what to do (e.g. is there a README?).
- User doc
  - Is the user doc accurate? A small typo can lead to big problems for a user (especially new users). Does it partition user, user-developer and developer information or mix it all together?
  - Is the user doc online? Are there any supporting tutorials? Do these list the versions they apply to?
  - Is it task-oriented, structured around helping users achieve their tasks?
- Help and support
  - Is there a list of known bugs and issues, or a bug/issue tracker?
  - Is it clear how to ask for help e.g. where to e-mail or how to enter bugs/issues.
- E-mail lists and forums
  - Are there e-mail list archives or forums?
  - If so, is there evidence of use? This can indicate whether the software is popular or not.
  - Are they searchable?
- Public bug/issue tracker
  - Is there a bug/issue tracker?
  - If so, there evidence of use?
  - Does it seem that bugs and issues are resolved or, at least, looked at?
- Quality of service
  - Is it clear what quality of service a user expect in terms of support e.g. best effort, reasonable effort, reply in 24 hours etc.?
- Contributions
  - Is it clear how to contribute bugs, issues, corrections (e.g. in tutorials or user doc) or ideas?

## User-developer perspective

The evaluation should be focused on development tasks, programming components against published extensibility points or writing clients for services, for example. Whether or not this development task is actually done by you depends upon the languages needed, the complexity of development the software might require, time available and your background. The support for user-developers in carrying out such tasks can still, to an extent, be evaluated regardless as to whether or not you have familiarity in the required programming language.

Evaluate and comment on,

- How easy is it to set up development environment to write code that uses the software or service? This may involve getting the source code of the software but for online services, it might not.
- Is it clear what third-party tools and software you need, which versions you need, where to get these and how to set them up?
- Are there tutorials available for user-developers? Do these list the versions they apply to? Are they accurate, and understandable?
- Is there example code that can be compiled, customised and used?
- How accurate, understandable and complete is the API documentation? Does it provide examples of use?
- For services, is there information about quality of service? e.g. number of requests that can be run in a specific time period. How do user-developers find out when services might be down etc. This can be very important if the user-developer building an application based on the service.
- For services, is there information about copyright and licencing as to how the services can be used? e.g. for non-commercial purposes only, does the project have to be credited etc. Is there information on how any data can be used, who owns it etc.?
- Is there a contributions policy that allows user-developers to contribute their components to the client? How restrictive is it? Who owns the contributions?
- Is the copyright and licencing of the software and third-party dependencies clear and documented so you can understand the implications on extensions you write?

The user perspective areas of release packaging, how to get started, user doc, help and support, e-mail lists and forums, public bug/issue tracker, quality of service, contributions are also relevant to user-developers. But, there may be differences in what is offered to user-developers e.g. a separate e-mail list for queries, or a lower quality of service for support.

## Developer perspective

The evaluation should be focused on development tasks relating to changing the software, to extend it, improve it or fix it. Put yourself in the position of someone joining the project and evaluate the extent to which you can explore, understand, build, extend, fix or recode the software. Again, whether or not this development task is actually done by you depends upon the software, time available and your background. The support for developers in carrying out such tasks can still, to an extent, be evaluated regardless.

Evaluate and comment on,

- How easy is it to set up development environment to change the software?
- How easy is it to access to up-to-date versions of the source code that reflect changes made since the last release? i.e. access to the source code repository.
- How easy is it to understand the structure of the source code repository? Is there information that relates the structure of the source code to the software's architecture?
- Is it clear what third-party tools and software you need, which versions you need, where to get these and how to set them up?
- How easy is it to compile the code?
- How easy is it to build a release bundle or deploy a service?
- How easy is it to validate changes you've made? This includes building the software, getting, building and running tests.
- Is there design documentation available? How accurate and understandable is it?

- Are there tutorials available for developers? Are they accurate, and understandable?
- How readable is the source code? Well-laid out with good use of white-space and indentation?
- How accurate or comprehensive is the source code commenting? Does it focus on why the code is as it is?
- Is there a contributions policy that allows developers to contribute their changes to the client? How restrictive is it? Who owns the contributions?
- Is the copyright and licencing of the software and third-party dependencies clear and documented so you can understand the implications on changes you develop?
- For open source projects, is it clear how you could become a member?

The user and user-developer perspective areas of release packaging, how to get started, user doc, help and support, e-mail lists and forums, public bug/issue tracker, quality of service, contributions are also relevant to user-developers. But, there may be differences in what is offered to developers e.g. a separate e-mail list for queries or a lower quality of service for support.

## Member perspective

The evaluation is similar to that of developers. Important differences include the fact that members might be able to change any infrastructure used by the project that is not publicly available. Members also need to understand how the project is run and which standards and processes need to be complied with.

Evaluate and comment on,

- Is it clear what are the processes and standards governing the project? e.g.
  - How is the introduction or updating of prerequisites managed?
  - How are release schedules determined?
  - Who manages releases?
  - What are the project coding and doc standards?
  - Who needs to be notified of changes in APIs or configuration file formats? How are these managed?
  - Who determines the direction of the project and how?
  - How is support managed and who is responsible for what?
- Who are members of the project and what are their roles and responsibilities?
- How members communicate?
- Is the copyright and licencing of the software and third-party dependencies clear and documented so you can understand the implications on changes you develop?

## Tutorial-based evaluation results

Your experiences should then be summarised. Explain what the problems were that you found, why these are problems (e.g. why is inconsistent highlighting of hyperlinks an issue, why should shortcuts be provided) and suggest how they can be sorted. It can help if you link out to our SSI guides[2] or other online resources if relevant. Together, these help contribute to SSI's goal of disseminating good software development practice.

Be specific and cite the version(s) of the software you used, any third-party software you downloaded and used with it, the platform(s) upon which you ran the software etc. For web sites or

[2] http://www.software.ac.uk/resources/guides

portals state the browser(s) you used. Follow the advice we give to our clients when writing user doc about being specific, citing error messages exactly as they appear etc.

There are a number of ways you can present your findings. The presentation adopted is up to you but you may wish to ask the client if they have any preferences.

- **Experience-oriented**. Structured according to how you did your tasks, this is step-by-step description of your experience with a task, what you tried, what happened and the problems you encountered. For example, here are some of the section headings from an SSI review of *VRIC*[3]:
    - Approaching and Obtaining VRIC
        - Web Presence
        - Documentation
        - VRIC Server Subversion Repository
    - Building VRIC Server from Source
        - Setting up the Development Environment
        - Building the Software
    - Installing and Configuring the VRIC Server
        - Installation
        - Configuration
- **Task-oriented**. Structured around specific sub-tasks. This groups issues around sub-tasks that the types of user might do e.g. running the software, setting up the development environment, building the software. This can be similar to the experience-oriented approach depending on your preferences.
- **Checklist-oriented**. Structured as a set of instructions which can act as a checklist for clients. For example, here are some of the section headings from an SSI review of *DICOM Confidential*[4]:
    - How to improve GUI usability
        - Provide feedback where the user expects it
        - If using the console for logging then let the user know
        - Avoid Java class names and exceptions in the GUI
        - Prevent errors
        - Try to detect errors sooner
        - Prompt if there are unsaved changes
    - How to improve batch file usability
        - Allow users to run batch files from any directory
        - Avoid the need for users to edit scripts
        - Prevent errors by trimming trailing spaces from properties

If you've done an evaluation from the perspective of multiple user classes then you can sub-group the above according to user class.

[3] http://www.jisc.ac.uk/whatwedo/programmes/vre/vric.aspx

[4] http://sourceforge.net/projects/privacyguard