User Personas

Olwen: The System Administrator

Characteristics

- Software Professional
- Uses Arch at work (don't ask)
- Most interested in reliability, stability, fine-grained control
- Needs to be able to automate tasks
- Needs to be able to receive alerts and view logs. Requires record keeping

Gwyn: The Power User

Characteristics

- Prefers doing work on the command line wherever possible.
- Prefers using the keyboard to control applications.
- Some understanding of the mechanics of a linux system.

Interests

- Works in IT, help-desk, linux support
- Programs casually.

Lleu: The Casual User

Characteristics

- Hasn't been using a Linux system for long.
- Primarily uses GUIs wherever possible.
- Not comfortable with the command line
- Prefers to use the mouse to control applications.
- Wants to spend as little time managing the installed applications as possible.

The Power User

Characteristics

- Prefers doing work on the command line wherever possible.
- Prefers using the keyboard to control applications.
- Some understanding of the mechanics of a linux system.

Interests

- Programs casually.

User Scenarios

Scenario 1: Lleu needs a pdf reader.

Lleu has been using their new Arch Linux system for a couple of weeks now; they've had some trouble, but they've managed to solve their problems mostly without help (other than documentation and tutorials) which has given them confidence. A colleague wants to show them a report they are working on, and sends Lleu a file over email. The file in the PDF format. When Lleu tries to read it they realise: their system doesn't have a PDF reader installed! If this were windows (which Lleu is familiar with) they would just go to a web browser, search for pdf reader, and then install the first option available (Adobe Reader). But since this is Linux, it's done differently, and Lleu knows this.

Lleu knows that to install software, you use a special program, the package manager, so they open up the package manager application. They're not sure where to go from here, but they know they want to search for something, so they look for something that looks like the search bars they are familiar with from web applications. Up the top of the window they notice what looks like a text-entry box with a little magnifying glass next to it, they guess this is it, and click on it; the box gets highlighted slightly, and this makes them confident that they application is listening for their input.

Lleu types in "pdf" into the box and presses enter, and a large number of results shows up. They scan through the results looking for something useful; the vast majority of the results are confusing to Lleu, talking about conversion, libraries, or other terms they don't understand, at least in this context. They find the first thing that seems to be a pdf viewer, open up it's information screen, and click install. The system seems to perform some function, eventually indicating that the program is installed. "Right!" thinks Lleu, "now I should be able to open the file!". They go back to the downloaded file in the web browser, and try and open it again. They get the same screen as before, asking them to choose an application to open the file with; this time Lleu notices that the program they just installed is in the list. They select it, and it opens. The reader app is a bit of a mess: a very old looking interface, with no clear controls or menus; thankfully it accepts mouse commands and Lleu can scroll through the document easily enough. They aren't particularly happy with the reader, and decide that when they have time they may try and look through the package list again to try and find a better one.

What can we learn from this scenario?

- Often, simple text search is not enough, and will give much too many irrelevant results. This is a very common situation in Linux package managers. Users could be helped in their search with the use of categories, recommendation or rating systems, or search hints (think like in google and other search engines).

- Having one central place to find applications to install can be tremendously useful, and will encourage users to go to that place when they are looking for a new program to fit some need. This can be clearly seen in Android and IOS through Google Play and the App Store respectively.
- Software varies massively in quality, especially in the open-source ecosystem. Often users will have no way of telling from the outset how mature, polished, or complete a particular application will be until they have already installed and run it; This can cause wasted time and frustration for the user. Again, a review/recommendation system could be particularly helpful here, as well as a more detailed description of the program's functionality.

Scenario 2: Gwyn is installing a C compiler.

Gwyn is starting a hobby programming project, and is trying to decide on what tools and languages to use to develop it in. She decides that she want's to give C another shot, because it would be useful to know for some of her other projects. However in the past she has had very bad experiences with the "standard" compiler, gcc. She decides to look online to see if there are any other compiler projects out there that are mature, modern, and which run on linux (she's aware of the Microsoft compiler system, but she isn't willing to switch back over to Windows to use it).

After some searching, she finds what she's looking for: the LLVM system, and the clang compiler that is a part of it. She opens up the package manager, and immediately searches for "clang" in the search bar; she's done this before and knows what she's doing. A number of results pop up, but since one is a complete match, it is put at the top of the list and is highlighted. She selects this option with her mouse, which annoys her somewhat, since she much prefers using keyboard shortcuts where possible, but doesn't know the keyboard shortcuts for this particular application, and doesn't know where to look to find them. In her experience they tend to be cumbersome or non-existent on GUI applications, so she hasn't bothered.

The info screen for clang pops up when she selects it. She selects install. Once it is done, she switches over to the terminal, writes a little "Hello World" application in C, and compiles it. It works smoothly, and she gets started with the project.

What can we learn from this scenario?

- If a user knows exactly what they're looking for, the system should ensure they can find it as quickly as possible. If the user knows the exact name of it, prioritize results that match the name exactly; if the user doesn't know exactly, allow them to search using the information they *do* have.
- Allowing for mulitple input systems (mouse or keyboard, text or key-commands, etc.) can allow users to utilize the system they find most effective. Forcing them to use a system that they don't prefer may cause undue frustration. Additionally, if the user isn't aware of the different

input systems (e.g., through documentation or tooltips) they won't be able to use it even if they would prefer it.

Scenario 3: Olwen needs to manage multiple versions of Java.

Olwen is managing their work development machine for a new Java project he has been assigned to. The project requires that it support multiple versions of java for machines that don't (or can't) have the latest versions; as a result, developers need to have multiple versions of Java installed simultaneously and test the software on each version.

Olwen starts up the system's package manager and does a search for "java". There are a *lot* of java related packages, and Olwen would find it very hard to find what he was looking for if the software didn't help out. The package manager understands that "java" is a core package that can have multiple different versions, so it has grouped them all together into a single "meta-package" and placed them at the top of the search results; Olwen clicks on this item and the system brings up the info screen. The screen shows all the available versions of the package, indicating which ones are installed. Olwen can select each version to toggle whether it will be on the system; he clicks on each of the versions that isn't yet installed, and then clicks the "install" button to confirm. The system brings up a window showing the changes, and asks for final confirmation.

Later on, management decided to drop support for a couple of the earlier versions of Java. Thankfully, the package system makes it easy for Olwen to just remove the versions he doesn't need.

What can we learn from this scenario?

- Packages can have multiple versions that don't necessarily conflict. This can be seen in this example (with Java) and with other packages (like Python). Having a system which understands this, and allows the management of multiple versions can be very useful.

Scenario 4: Lleu is trying to clear up disk space.

Lleu has run into a problem: their laptop has run out of disk-space (they knew they should have got the larger SSD), and they need to clear some space immediately. The package manager they're using shows the size of the files it's downloading, so Lleu guesses that there is probably a lot of space being taken up by the packages.

They open up the package manager, it opens to the standard main screen. For some reason, this system doesn't have the typical set of drop-down menus at the top of the screen like most programs they're familiar with (even many linux programs have them) and Lleu is feeling a little bit lost. But then they notice the

bit “Help!” button over in the corner, in fact it is gently pulsating a bit to make itself seen; They click on it. A screen pops up with several bits of information: * An introduction to the program * A link to the full documentation * A series of keyboard shortcuts

Lleu opens up the full documentation and starts browsing. There are many sections, all dealing with some subsystem of the software. They find the section “Managing the package database” and look in there for something useful; after a couple of minutes of searching, they haven’t found it. Back at the base of the documentation is a section “managing the package cache”. “What’s a cache?” they think to themselves, they’ve heard the word in a non-computing context before, referring to some sort of stash of supplies; that sounds somewhat useful, so they read that section.

The documentation guides Lleu to a button on the left-hand side of the main screen, a little button with no text, shaped like a series of boxes; they mouse over the button and it comes up with a tooltip: “Manage Package Cache”. Lleu clicks it, and a small section pops out from the left showing some information: the number and size of the packages stored in the package cache. At the bottom of this new section is a couple of buttons, the mouseover tooltips show “Conservatively clean cache” and “aggressively clean cache”. They click on “aggressive” and a message pops up “Fully clearing the cache may prevent you from being able to downgrade packages in the future, are you sure?”. Lleu thinks to themself: “why would it do that? shouldn’t I just be able to download them again?” and clicks on “Yes”. The cache clearing operation goes ahead, and clears up a significant amount of space. Despite the frustration with the interface and documentation earlier, Lleu is happy that the job is done, and they can continue using the computer.

What can we learn from this scenario?

- Important functionality should be easy to find as quickly as possible; a user shouldn’t have to slog through documentation and confusing menus to find the single option they’re trying to change. Drop-down menus *are* somewhat clunky, but they serve a purpose: putting away all those little options into sensible categories so that users can find them. While getting rid of drop-down menus seems tempting, make sure your interface can allow the users to access all the options through a more elegant method, or they’ll just end up lost and frustrated.
- Documentation is hard to get right, but getting it right could be the most important thing you could do for usability; with excellent documentation, a harder system becomes much easier to learn.
- Good icons are also very hard to get right, trying to visually represent a complex action with a small picture can take a team of designers all on it’s own. Good icons have to be clear, obvious as to what they represent, and evocative of the actual action. After the user figures out what icon

corresponds to which task, memorization can do a lot of work.

- Language is important for usability. Novice users may not be familiar with the large amount of jargon terms that are commonly used with computer systems. These terms were made to be evocative and to form analogies to real-world objects or tasks, but the abstract nature of the terms (in the computing context) may not communicate what the designers want them to. Documenting terms can do some of the work, but often it is better to just not use jargon at all, and use more verbose or familiar terms. This all depends on your target users.

Scenario 5: Gwyn broke her computer with an update.

Gwyn is working on her software project, and has run into a problem. After stopping to have a break, turning off the computer, and then turning it back on when she returned, the OS has developed a problem. Gwyn suspects that an update she did earlier in the day has broken something. With this possible cause in mind, she begins investigating by opening up the package manager. Up the top of the window are a number of icons representing the systems core functionality; one of these is an icon that appears to represent a clock. Guessing that this might be what she is looking for, Gwyn clicks on it. A new window opens up, and jackpot, it shows the history of changes made to the packages. Gwyn recognizes some of operations: the compiler she installed a week ago, a cleanup operation she did yesterday, and there it is: the update she did this morning.

When Gwyn clicks on the operation in the list, it opens up to show the individual packages that were updated as part of it. Additionally it has another drop-down menu showing the message log from the operation (what the system would have shown Gwyn when she performed the update). On both the operation and the packages, there is an option that can be selected to undo the process. Gwyn selects this option for the operation; this causes all the packages that were part of that operation to be selected for removal. Gwyn clicks on the "Do Changes" button at the top and is presented with a small screen showing the changes she chose. she click "confirm" and the system performs the changes.

Gwyn then goes about rebooting the computer to see if it worked, since the problem originally only appeared with the first reboot after updating. It works, and then she must go about the process of figuring out what package broke the system; unfortunately, the package manager is unlikely to be much help here.

What can we learn from this scenario?

- Undo functionality is a tremendously useful thing, that's why we see it in text editing so much. It is so much easier to just tell a system: "get me to this previous state" rather than having to undo all the changes yourself, which can take a long time with no guarentee that you will get it right in the first place. Wherever possible, undo functionality should be included

whenever a user makes some change. This included UI changes: the user should be able to get back to seeing what they were just looking at with a click.

Scenario 6: Olwen is trying to replicate an install configuration on multiple devices.

Olwen has been given the task of preparing a new shipment of development PCs for some interns who will be arriving at the offices next Monday. Olwen has been asked to recreate the set of installed packages from one of the already existing development machines.

There are many ways in which Olwen could accomplish this task, but they're hoping the package manager might be able to help them do it quickly. They start the package manger on the existing devlopment machine and immediately open up the documentation. Using the documentation search function, they type in a few keywords associated with what they want to do. The results don't contain anything they deem useful currently, so they switch some of the keywords and try again. A few results down the new results they find something that looks useful "Import and Export Installed Packages", In this section of the documentation is a series of instructions for doing just that. They go back to the package manager and find the commands that were laid out in the documentation; in the end he finds two simple commands: Export and Import. Each command prompts for a file name, the export command writes all the names of the currently installed packages to the file, while the import command retrieves the names from a file and installs them.

This functionality is useful, and Olwen will make use of it, but they were hoping for a way to also export the package *files* so that they wouldn't have to redownload it. They go ahead and export the package names, and then try to solve the second need through combining some other functionality. The package manager has function which will output the path to the package cache, where previously downloaded packages are stored. Then Olwen finds the function which allows one to specify a path from which to install packages. Through both of these functions, Olwen is able to do what they need: finding the package files, copying them and the installed package ledger to a USB drive, and then performing the import operations on the new machines when they arrive. Because the package manager, being a simple GUI application, is non-programmable (unlike it's command line counterpart), Olwen is unable to write a script that performs the functionality; they find this *incredibly* frustrating.

In the end, they get the work done. Olwen decides that in the future, if they are faced with the same problem, they will likely solve it a different way, probably through making an image of the drive and replicating that instead. They decide to do some more research.

What can we learn from this scenario?

- It can be difficult to figure out what functionality the user will need in advance. This problem is the reason for research methods in Interface Design, things like surveys, focus groups, and general data gathering. Having elegant and clever systems that allow the user to perform functionality that was not originally envisaged by the designer is another way to solve this problem.
- Having a useful function in your program isn't worth much if the user can't find it. As a program gets more complicated, it becomes harder to display everything in an easy to find location. While elegant categorization and innovative UI design can certainly help, it can only achieve so much. TODO: USER CATEGORIES By anticipating

Scenario 7: Lleu is trying to learn how to use the package managers advanced functions.

Lleu has found a new game they want to play. This game is available on Linux, so this time, instead of switching over to Windows like they would normally do, they instead decide to give playing it on Linux a shot. They go to the game's website, and find the installation instructions. The game has specific installation instructions for the version of Linux they are using. They learn for the first time here that the package manager has another method of installing programs: rather than choosing them from the list, they can download a special file known as a "PKGBUILD" (Lleu finds the name rather strange, and doesn't quite understand what it means), and the computer will install it for them.

Lleu begins following the instructions from the game's website. As they are part way through the instructions, they start running into problems: the screenshots and instructions don't seem to match what Lleu sees before them on the screen, they seem to show screens, buttons, and describe options that don't exist. It seems as though the instructions haven't kept up to date with the package manager as it has progressed. Lleu spends some time looking around the application, "maybe it's just been moved" they think. After a while with no success, they start getting frustrated and leave to take a break, hoping they will figure it out later.

Later on, Lleu is back and ready to try again. They go back to the instructions and try and pick out the key steps and concepts, and then go to a search engine and search for the name of the package manager and a few of the key terms. After a bit of searching, they find a page in the package manager's documentation which talks about the process of installing packages from an outside source. By following these new instructions, and putting in the relevant values from the games instructions where relevant, they manage to make good progress. However, when it seems like they are almost done, an error occurs. In the process of installing the game, the system runs into an error; but it doesn't go into any

detail at all, just something along the lines of "An error occurred and installation cannot continue". Lleu tries again, receiving the same error. They throw up their hands in disgust, switch over to Windows, and install the game there without issue. In the end they start thinking whether this whole Linux experiment was a mistake on their part.

What can we learn from this scenario?

- Not everything will work perfectly; things go wrong all the time, and the system needs to be prepared for this, handle it gracefully, and be very thorough about informing the user what happened, how they can fix it, and where to go for more information. There is nothing more frustrating than a program outputting an error code and nothing else (or worse: just saying "error"). How a system goes about reporting problems can determine the difference between the user succeeding at what they are trying to do or them giving up in frustration, and possibly giving up on your system altogether. Additionally, when errors happen, the user shouldn't lose any progress at all: there is nothing worse than an operation failing because of one small option, but the user has to go through the entire process again.
- Your users shouldn't have to rely on third-party sources to learn about features in your program or how to use them. The documentation should be present and complete so that third-party sources can simply point users to your documentation; that way third-party instructions shouldn't go out of date with your system as long as the documentation is also up to date. Making your documentation "digital-native" (as opposed to a book or static webpage) through being linkable, sharable, and searchable, can go a long way to having a functioning ecosystem around your system rather than a set of disjoint, awkward connections.
- Changes to the way an application works, especially in terms of interface, can be a very frustrating experience for users. While expert users can sometimes quickly adapt, most users will face disruptions to their work and a period of relearning where they are significantly less effective. Putting plans in place to transition from one interface and way of doing things to another can really help your users with this process. Extensive documentation showing how to do tasks in the new system should be made, and users should be informed of changes long in advance. However, if it is at all possible, try and allow for mulitple ways of doing things, both the old and the new.

Usability Goals

Speed:

The system should perform its actions very quickly, with almost imperceptible delays for simple actions like searching, querying the database, and accessing help and documentation.

Questions

- Is there only a short or nonexistent delay for most actions?
- How long does a user have to wait for the system to perform operations?

Learnability:

The system should be self-documenting, with help messages, tooltips, and useful information embedded into the interface rather than exclusively in external documentation. When there is a problem (some form of error, or otherwise), the system should provide informative and helpful messages that direct the user towards solutions. In addition to in-system help, there should be documentation that clearly and thoroughly details the whole system, how to use it, and all functionality.

Questions

- Is the documentation high quality (thorough, easy-to-read, well laid out, etc.)
- Are functions documented in-place, in the documentation, or both?
- If a user wants information on a specific operation, how quickly can they find it?
- Does the system have easy to understand concepts, metaphors, etc.?
- Does the system have tutorials?
  - Do they cover a large range of functionality?
  - How much of the system can users learn through tutorials?
  - How easy are the tutorials to follow?

Memorability:

The control buttons should be designed and constructed in such a way as to indicate in which order to use them, and which ones are valid to be used at different times. The operations which are most relevant to the current situation should be front and center in the most obvious position and designed in a way that makes them stand out. Operations that are currently invalid should be faded out, or simply not presented at all.

Keyboard shortcuts should be mnemonic if possible, or have some other way to help with memorization. Users should be able to easily see common keyboard shortcuts at a glance.

Questions

- Can users easily remember how to do certain operations?
- Can users easily remember where to find particular functions?
- Can users easily remember keyboard shortcuts?
- Does the system guide users in a way that encourages memorization?

Utility:

The system should provide the necessary set of functionality to do package management. The basic functions, like adding, removing, and updating packages, and more advanced functions.

Questions

- Does the system provide the necessary functionality to manage the software installed on the user's device.

Safety:

The UI should be designed in such a way that it makes it hard for users to do actions that permanently and irreversably lose work. One way to do this is with an "Undo" function, others are through thoughtful placement of commands, confirmation prompts, and making the system informative enough for the users to be sure that the commands they are issuing do what they expect it to.

The physical aspect of safety (saftey from bodily harm) is not revalent to this system.

Questions

- Does the UI help prevent users from making mistakes?
- Can any changes the user makes be undone?
- Does the system inform the users of what each function does?
  - Do the users understand this information?

Efficiency:

How well does the system support the user in performing their tasks. Do simple actions require multiple steps, or very few steps?

Questions

- Can a user perform the basic actions of the system in a very small number of steps.

User Experience Goals

Positive: Ones We Want to Create

- Helpful: When using the system, how much help do users receive from it in completing their tasks?
-

Negative: Ones We Want to Prevent

- Frustrating: Does the user have to struggle with the system to get what they need done? Does it get in their way and limit them?
- Confusing: Does the user understand how to use the system's functionality? Does it make sense? Do user's often find themselves confused about what a function does, what an error message means, or where to find what they need?

Use Case Analysis

1: Installing a Package

1. Find the package
2. Open the packages information page
3. Select "install" on the packages info page
4. The user confirms the operation
5. The system performs the installation and displays the result.

2: Searching for a Package

1. The user selects the search function.
2. The user enters the search term.
3. The system performs the search and displays the result

3: Browsing Packages

1. The user selects "browse" or similar
2. The system displays the list of available packages.

4: Updating the System

1. The user selects the "Update System" option.
2. The system shows the changes involved and asks for confirmation
3. The user confirms the update.
4. The system performs the update and displays the results.

5: Installing a Package From an External Source

1. The user enters a URL or file path
2. The system fetches and analyses the files for validity
3. The system displays the changes involved in installation and asks for confirmation
4. The user confirms the installation
5. The system performs the installation and displays the results.

6: Removing a Package

1. The user finds and selects a package, and opens up its info page.
2. The user selects the "remove" option on that page
3. The system prompts the user for extra options, displays the changes involved, and asks fo
4. The user selects extra options and confirms the changes.
5. The system performs the changes and displays the results.

7: File Search

1. The user selects the "file search" option (or similar).

2. The user enters a search string.
3. The system performs the search and displays the packages containing files matching the se

8: Clearing the Cache

1. The user selects the "manage the package cache" option.
2. The system displays the state of the cache (number of packages, size of cache, time since
3. The user selects either "conservative clean" or "aggressive clean"
4. The system asks for confirmation
5. The user confirms the operation
6. The system performs the requested clean and displays the cache state again.

9: Managing Multiple Versions of a Package

1. The user finds a package and opens up its information page.
2. Where a package has multiple versions available, it will display the available versions.
3. The user selects a number of versions to install or remove, and selects "install" (or sim
4. The system calculates and displays the changes and asks for confirmation.
5. The user confirms the operation
6. The system performs the operation and displays the results.

10: Looking at the Install History

1. The user selects the "display install history" option.
2. The system gathers and displays the history of changes to the package database.

11: Undoing Changes

1. The user is in the "Install History" page.
2. The user selects one or many changes in the history and selects "undo changes".
3. The system calculates and displays the changes involved in undoing the history and asks f
4. The user confirms the operation.
5. The system performs the changes and displays the results.

12: Exporting Installed Packages

1. The user selects the "Export Installed Package List" option.
2. The system prompts the user for a location to save the list.
3. The user enters the path and confirms.
4. The system saves the package history to the specified path and displays the result.

13: Installing Packages From a Package List

1. The user selects the "Install from package list" option.
2. The system prompts for the path to a package list file.
3. The user enters the path.

4. The system processes the file, calculates and displays the changes involved, and prompts
5. The user confirms the operation.
6. The system performs the operation and displays the results.

14: Viewing the Documentation

1. The user selects the "Show Documentation" option.
2. The system displays the main page of the documentation.
3. The user browses the documentation by selecting sections they wish to view.
4. The system displays the selected sections.
5. When the user is done with the documentation, they close the documentation screen.

15: Searching the Documentation

1. The user is on the documentation page.
2. The user selects the "seach documentation" option.
3. The system prompts the user for a search string.
4. The user enters a search string and confirms.
5. The system performs the search and displays the results.
6. The user selects an option they are interested in and browses the documentation.