
GUI design - science, or psychology?

Harry Powell

16th August 2017

1/28

This is an introduction to designing graphical user interfaces - GUIs, or what is now often what is referred to as "UX" or "user experience" design (also referred to as "UXD"). For most current users, the GUI will be the way that they interact with the scientific applications that they are running; an interface that is designed with the final user in mind rather than the needs of the application programmer is likely to encourage people to use the application.

A thought on psychology...

Google tells me -

psychology

/sʌɪˈkɒlədʒi/ 

noun

1. the scientific study of the human mind and its functions, especially those affecting behaviour in a given context.

synonyms: study of the mind, science of the mind, science of the personality, study of the mental processes
"she has a degree in psychology"

as far as I am concerned, properly conducted psychology is as scientific as any other science

2/28

Over the years, poorly conducted and interpreted psychological studies have given the subject a bad name; however, when properly performed, and when the results are analysed using robust statistical methods, psychology is as valid a science as any other.

Human centred design

Think about your target audience

- They are all human (so they each have a psychology)
- Most will have used other interfaces
- Use familiar idioms, e.g.
 - what does Ctrl-O (⌘-O) normally do?
 - People like things that they are used to (mostly)
- Make frequent actions easy & obvious
- "intuition" is deeply ingrained knowledge based on experience

3/28

Currently, UXD follows the methods of "human centred design". In other words, interfaces are designed with the final user in mind, who is invariably a human (or an entity with a very similar psychology).

Everyone who is encountering scientific software for the first time now will have already used other interfaces to perform other tasks; they will anticipate the function of different interactions in the light of their previous experience, so it makes the transition to new software easier if their actions lead to outcomes similar to what happens with other software. For example, Ctrl-O (for Mac users, ⌘-O) is usually associated with opening a file, so (under most circumstances) it makes little sense to assign this key combination to another function.

Where actions need to be performed frequently, users will curse the designer who has implemented the functions with widgets (buttons, sliders, etc) widely separated on a GUI.

Occasionally, a new way of interacting is developed (for example, pinching to zoom or swiping to change from one application to another on a tablet or smartphone) that is more intuitive than current methods, but these cases are rare, and should probably be avoided by newcomers to UXD.

As noted, intuition is knowledge that has been acquired and used in practice extensively over a long period, so that the user does not need to think about their actions in a conscious way. UX designers can take advantage of this by allowing users to apply their previously obtained experience in a new interface.

People have different levels of expertise



It is plain that people using an application will have different previous experience; in any field, a few people will be novices and may well need some training (but they should be able to use their previous experience to reduce what is necessary). A few users will be real experts who need no instruction - they may be the authors of the application program, or even the original developer of the scientific method.

Most of the users of any application will be intermediate in their skills; for example, I have used Microsoft Word since the early 1990's, and consider myself to be fairly competent - but I couldn't write a VB macro to convert a directory of .docx files to PDF files without googling it first, even though all the functionality has been in MS-Word for decades.

Incidentally, the real expert on the motorcycle and in the racing car is the only person to have won world championships on both forms of transport - John Surtees.

Types of users

- Novices
 - no-one wants to remain one
 - they are intelligent but busy
 - need to know how to operate the product, do *not* need to know how it works
 - will become intermediates if they continue using the product
- Intermediates <-----Design for this group
 - Most of your users
 - You want your novices to become intermediates as soon as possible
- Experts
 - Influence novices disproportionately
 - Are trusted by other users
 - Need access to "rarely used" functions
 - Are not typical

5/28

While some of your potential users will be novices, it makes no sense to design an interface based around their needs. After a very short time, if they continue to use the GUI, they should no longer be novices because they will have gained experience in its use.

Experts can be a very bad influence on novices; they know the shortcuts (as well as the long routes when things don't go as planned), and their training is often so far in the past that they have forgotten just how difficult it can be to make progress. They are also not a good target for most UX designers because they form a small fraction of the user base.

Almost without exception, UX design should be aimed at intermediate level users (which nevertheless, includes people with a wide range of experience and abilities); the main thing to be borne in mind is "most of the users, most of the time". A well-designed interface will allow the novice to make progress but will allow an intermediate user to customise their actions to give a better result with minimal input.

Different mindsets

Users are interested in goals, *e.g.*

- "I want to analyse the active site in this enzyme"
- "Is my DPPE chelating or bridging?"
- "Will my MOF bind tert-butylamine?"

Applications programmers are interested in actions, *e.g.*

- calculate an FFT
- minimise this function *via* maximum likelihood

They *care* about the algorithms *e.g.*

- maximum likelihood is important in refinement
- the FFT indexing in Mosflm/Denzo/DIALS is important

They are experts in the use of the software

- most would be happy to change a few lines of code to get the program to do what they want
- They are experts in the science behind the software

6/28

It should not need to be stated that users and application programmers will often have different views on the software.

Most users are more interested in goals - in the final result from their efforts, rather than the route they took to arrive. Taking a scientist solving a crystallographic problem, for example, they might want to know something about

- a catalytic triad in an enzyme's active site
- the mode of binding of a bidentate ligand
- how large the cavities are in a metal organic framework

but probably don't really care about whether a least-squares or maximum likelihood method has been used in a minimisation, as long as it is reliable.

Application programmers should care about the actions used to arrive at the answers - they need to make sure that the results are reliable and defensible. Further, if they find that the methods do not work as expected, many would be perfectly happy to edit their own (or other people's) software so that it did work as designed. To a certain extent, they will not care at all about the scientific results as long as their part in the process has performed as well as it can.

It has to be said that there are a very few people who fall into both camps, but statistically speaking, they are outliers.

Interfaces to applications

- Command-line and/or control files
 - so 1980's
 - probably need to read and understand a log file
 - what's a log file?
- Automation
 - user (with luck) never even sees the applications - only the results
- Graphical User Interfaces
 - interaction *via* widgets
 - visual feedback *via*
 - line plots, scatter plots
 - pie charts
 - images and overlays

7/28

Command-line interaction with scientific programs dates back to the days of 80 column punched cards; this is why many commands are limited to 80 characters, and need some kind of continuation marker at the end of the line if they are longer than this. For many applications, they will provide the most precise control over what the program is actually doing, but they are not the obvious way for people to tell programs what to do in the modern world.

Automated processing (usually *via* some kind of pipeline) will make use of a command-line syntax, but it will be hidden from the user. Most importantly, the user will rarely have much control over the course of the processing; pipelines are the epitome of black boxes.

Since the early to mid 1990's, the normal way for people to interact with any computer program has been with some kind of GUI using the WIMPs protocol (Windows, Icons, Mouse, Pointer), so there is a whole generation of users for whom command-lines and control files are alien when they first encounter scientific software. The introduction of tablet computers and smartphones has meant that many people now view the WIMPs interface as a second choice. Recent editions of Windows (in particular) can operate using either a touch screen or mouse *etc.*; the change away from WIMPs is likely to continue.

The other advantage of GUIs is the feedback. While line printer (what's a line printer?) plots were used to interpret results previously, the output is substantially enhanced by the use of visually attractive plots (which are not new inventions - Florence Nightingale used pie charts in the 1850's).

Why have a GUI?

- It's how people access software now!
- Easily interpretable feedback (including error notification)
 - Visualising results
 - Locating problems
 - Identifying pathologies
- Removing the need for learning command-line syntax
- When autoprocessing "fails", e.g.
 - doesn't give a file with $h, k, l, |F|$ (or I or $|F^2|$) & $\text{sig}(|F|)$ (as a minimum)
 - doesn't work well enough for the desired experimental outcome
- Training users (explaining the "black box")
- Avoiding typos (enter values *via* widgets)
- *etc.*

8/28

Most people coming to any program these days will expect to use a GUI - application developers, "experts", "power users", "old fogeys" *etc.* might not, but these make up a tiny minority of normal users.

A *good* GUI will give a good indication that everything has gone well in processing, or will provide feedback that readily identifies any issues; careful design of an interface will also include a means to inform the user of which problems are most serious and how to address them, and yet leave the scientific decisions about the processing in the hands of the user.

Hard-core UX designers will eschew access to any command-line interface, so the user does not need to even know it exists. In practice, this is rarely the case (see the hidden command-line interface in *iMosflm* or the scripting interfaces to *Coot* or *PyMol*), but most users most of the time should not need to know the commands.

Autoprocessing sometimes results in processing that can be improved with small adjustments; in these cases, it is useful to have more control over the underlying applications.

Training is often overlooked as a use for GUIs; when a novice is learning about a method, using a GUI can break down a process into individual steps so their importance can be emphasised.

Not using the keyboard to enter commands is the best way to avoid typographical errors, which can be difficult to identify - we tend to see what we think should be present in text!

Some thoughts on GUIs

A GUI *might* be an afterthought, but it is probably the way most people use the application.

- isn't the the application the important bit?
- A GUI is not purely cosmetic
 - Should enable the user to process more effectively
- Written by the application developers
 - Never in a commercial environment
 - Almost always in non-commercial environments
- What makes a good GUI?
 - reaches a target user-group
 - appropriate controls/feedback
 - minimised excise

It is best to separate the roles of GUI designers & programmers and Application programmers because they address different problems; **however**, GUI designers and application programmers should work together.

9/28

For an application developer in academic software production, a GUI will probably be an afterthought; many (most?) academic software developers are scientists who want to analyse data to answer a question, and the methods used to reach the answer are the important things. They *know* what the input and output should look like, and (in answering the intellectual question) are extremely capable of scanning through voluminous output for a single value hidden amongst many thousands of others.

The GUI is not primarily for the programmer(s), though if it is any good, they will also use it in preference to other methods, because it will give a route to more effective processing.

Unfortunately, in academic environments, the GUI is usually designed and written by people with no expertise in UXD - this is not the case in industry, where the roles of application programmer and interface designer are separated. However, close interaction between the two groups, and an understanding of each others' roles is paramount.

A good interface will be designed with a clear target audience in mind. This audience depends on the function of the application, but in most cases the "perpetual intermediate" users should be remembered.

Excise will be discussed later.

Interface design is *very* important...



Sir Jonathan Paul "Jo
Chingford, England. I

This is a brief aside to remind application programmers that interface design is extremely important and very marketable. Jony Ive is the highest paid employee of Apple, and this is down (largely) to his design skills; his remuneration is reported to be substantially higher than that of the company's CEO.

He earns substantially more in one day than an Apple application programmer earns in a month.

Different strokes for different folks

An application programmer should know about

- what input and output is important for completing the actions
- how to program complex algorithms - the science

A GUI programmer needs to know

- what input and output is important to achieve the goals
- how to program widgets, canvasses, *etc.*

A GUI designer *needs* to know

- about users
- *absolutely nothing* about the algorithms
- *something* about the science *can* be helpful (but not essential)
- which goals are important
- what i/o methods are possible

11/28

The different roles of interface designers (& programmers) and application programmers mean that they require different expertise. GUI designers and GUI programmers are often the same people, but have subtly different roles.

As noted before, it is reasonable to expect that the application programmer will be an expert in their field - so in crystallography, they should probably be considered (and to consider themselves) as a crystallographer, or at least a scientist with expertise that is directly applicable to crystallography. They should certainly know what input is required to achieve their scientific result and what results should be obtainable, and what those results are likely to mean in the context of the experiment performed. Most of their concern will be with implementing the algorithms that make up the science.

The GUI *programmer needs* to know none of this (though an appreciation of the science may be helpful). They *do* need to know what form the input may take (*e.g.* integers, floating point numbers, output from an ADC), and what range of values are likely for both input and output (and which should trigger a warning).

Similarly, some knowledge of the science might be helpful for the GUI *designer* but is not essential; any knowledge of the scientific algorithms themselves will probably not help them in the design process. They *do* need to know about the psychology of how users interact with interfaces, and how to present output clearly in an "intuitive" way; this intuition depends on the user's previous experience.

Users don't care about algorithms

"Mosflm autoindexes with a method based on applying a Fast Fourier Transform routine to the dot products of one-dimensional projections of reciprocal space scattering vectors and vectors describing a hemisphere in reciprocal space."

"Denzo autoindexes by applying a Fast Fourier Transform to the three-dimensional reciprocal space scattering vectors themselves."

- To a very good approximation*, none of your users care about this - they only care that it works in the program they are using.

* I'd *guess* fewer than 1 in 100

12/28

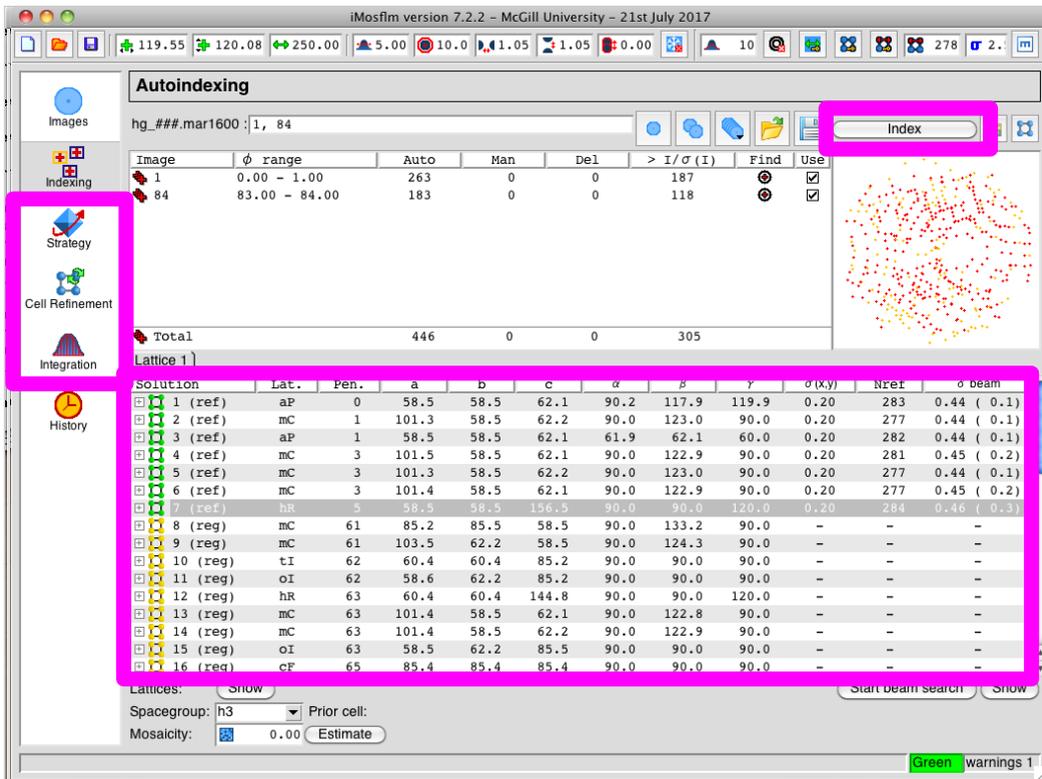
This is to emphasise that, not only do GUI designers not need to know about the algorithms employed in an application, but that users (for the most part) simply do not care.

In nearly 20 years of working with *Mosflm*, and in particular with the autoindexing method that it uses, I cannot remember a single time when a normal user has asked about the mathematics behind the method (although I have told people many times in various lectures I have given!). They don't care as long as

- (a) it works
- (b) if it fails they can change the input so that it does work
- (c) changing the input is straightforward if they need to do so

The fact that *Mosflm* and *Denzo* both use methods based on FFTs, but one is one-dimensional with many FFTs and the other is three-dimensional with a single FFT is a matter of total irrelevance to most users.

Most users are also not bothered about what a scattering vector might be.



This illustrates the kind of thing that most users most of the time want from an integration program when it does the indexing.

They want to go into an indexing task of some kind, tell the program they want to index (they might even think they want to index the diffraction pattern on a selection of images), and they want the indexing to take place. Then, they want an indication of what they can do next.

Here, pressing the "Index" button in the *iMosflm* Autoindexing task will index from a selection of spots on the images selected (which have been selected automatically, and the spots found automatically), the results displayed in an obvious way; between them, *Mosflm* (the data processing program) and *iMosflm* (the GUI) indicate which is the likely best solution, and the three task buttons on the left (Strategy, Refinement, Integration) which were previously greyed-out and inactive become available - because it now makes sense to allow them.

Before you start work on a new GUI

Talk to expert users - watch them working -

- what do they do?
- what information do they *actually* use most of the time?

Is there a current GUI for this application?

- Observe normal users using it
- what do *they* do?
- what information do *they actually* use?

Observe normal users using

- other software (both GUIs and CLI) to achieve the same goal
 - is there information they use that isn't available in your application?

14/28

In spite of the ideal being to aim your new interface at intermediates, it is worthwhile talking to and observing real experts using current interfaces (including the command-line); what actions do they perform most frequently, and what output do they make use of most often? These are likely to be the most important things to implement in any new interface.

If there is a currently existing GUI for the application, you should see how normal users use it;

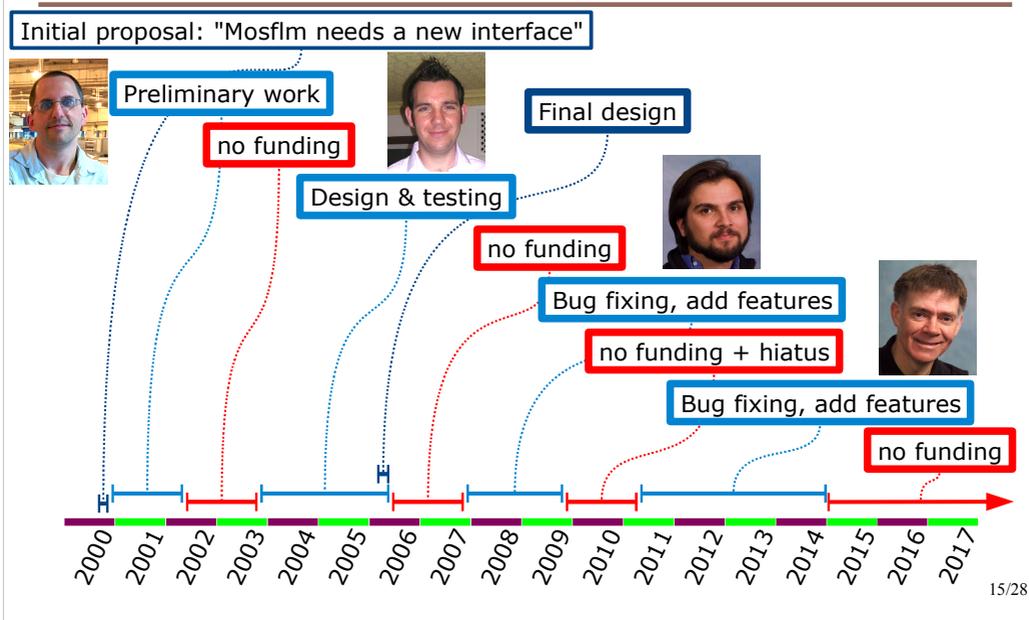
(a) do they use all the functionality?

(b) what do they do most commonly?

(c) do they need to use a command-line interface as well in order to attain their goals?

What about other software that does (broadly) the same thing? Ask the same questions.

Developing a good GUI takes time...



Designing and implementing a good interface takes time and a lot of effort.

The single major mistake that occurs is that the design is picked too early in the process.

In the case of *iMosflm*, Graeme Winter was employed to produce an interface with the look and feel of *ccp4i*, using the same technology. After he left, he developed the automated pipeline *xia2* and is now a senior developer in the *DIALS* project. Subsequently, Geoff Battye spent three years developing the design that is now extant. After throwing away Graeme's design *in its entirety*, he designed an interface based on observation of users, discussion with the application programmers and proper interface design principles.

In common with good industrial designers, Geoff did not settle on a single design until just before he finished the project. This flexibility meant that he could pick the best features of his various prototypes and include them in what has been a successful design, while rejecting those features that did not work well.

Following Geoff, Luke Kontogiannis and Owen Johnson spent years making the underlying code robust and adding new features so that *iMosflm* became a reliable interface that people wanted to use.

Mixed in with this, there were periods adding up to ~7 or 8 years where the project was not funded; it is vital to remember the financial support required for good design work and implementation. During these periods, there was little development but a good deal of bug fixing.

Application programs

A good modern application will

- analyse its own progress
- make good automatic choices based on input and results so far

The application will probably have to change to fit with the GUI

- accept new kinds of input (*e.g.* via sockets)
- produce different output (*e.g.* marked up with XML, json...)
- change flow

16/28

A major advance in application programming over the last few decades is that the programmes will often analyse their own output and optimise the processing themselves; previously, they required the user to make this analysis and re-run the application. This has been enhanced in the various pipelines with the addition of expertise from experienced users who are also programmers.

However, in many cases the pipelines use searches through log files for particular text strings so that they can find numeric values, and this means that if the output from the application changes, the pipeline can be broken. Any GUI that relies on the same methods for obtaining its results is somewhat fragile for the same reason.

It makes good sense for the application programmer and the GUI programmer to work together. If the GUI and application can communicate by some means other than "scraping" logfiles, *e.g.* with TCP/IP sockets, they can be tied more closely and exchange the true values of variables. Similarly, if the application produces formatted output using something like XML or json, the appearance of the output is less important (as long as it can be parsed).

Occasionally, it is necessary to change the flow of an application to allow it to fit in with the limitations imposed by logic or by the GUI toolkits; close collaboration between the developers is very important here.

Some tips on interface design

Finally, here are a few tips on some basic features of interfaces that you might find interesting or even useful.

Work for the user

Good application programs work for the user, e.g.

- read the metadata (e.g. image file header information) and use it
- read stored user profiles
 - save them automatically on exit

GUIs should present information & results using easily interpreted idioms, e.g.

- line graphs
- histograms
- pie charts
- scatter plots

Use shape and/or text to distinguish function - don't rely on colour (but it can be used to *enhance* the distinction)



18/28

In both applications and user interfaces, do what you can to make the user's life easier.

For example, if you are writing an X-ray data processing program (like *Mosflm*, for example) then read and interpret the "metadata" that exists in all modern diffraction image files - things like the wavelength used, the crystal to detector distance, the start and end rotation angles for the images. This is all useful in the data processing, and there is no good reason to simply ignore it (if it's wrong, it's a good indication that the beamline staff are not doing their job).

Information about processing should be saved automatically on exiting the program (this is exemplified in tablet-based word processing programs, which typically have no "Save" option. The user should have a "profile" created automatically the first time they run the program, and this should be updated to reflect how they use it - again, automatically.

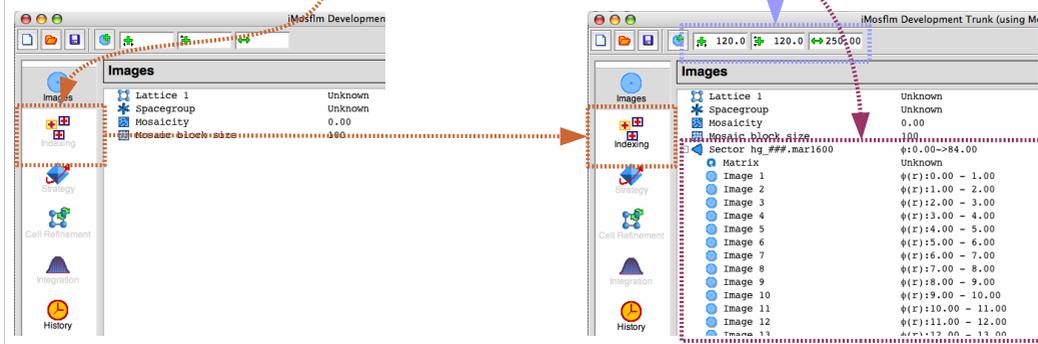
Familiar idioms for presentation of results are easier to interpret - people don't have to wonder what is being presented. For example, a bar chart of a frequency distribution immediately shows if it is Gaussian, or bimodal, or whatever.

The human eye is particularly good at picking out shapes before colours; so don't rely on colours to distinguish output. Remember that ~7% of men globally are red-green colour blind, and occasionally monitors develop faults in one or more colour channels.

Let the computer do the work

e.g. *iMosflm*, the user chooses the images to process, then the program

- reads & interprets all the image headers
- sets processing parameters accordingly
- presents values/output clearly
- updates available actions



iMosflm is a particularly good example of letting the computer do the work. The application *Mosflm* reads the metadata in the image files, and does this for every image in the the dataset. Then, *iMosflm* displays both the common information (beam centre, distance) and the information unique to each image (rotation range). Other information is also read and stored by *iMosflm* but not displayed in the main window, because it is not of immediate primary interest (e.g. although crucially important, it is not necessary to display the wavelength at all stages in the processing).

Having read the image files, the display in *iMosflm* is updated to reflect the information that the program now has. Also, the choice of sensible actions will change, and this is also updated.

At all stages, an effort has been made to present useful information as clearly as possible.

Use the common tools

- Most GUIs use a small array of tools that people recognise -
 - windows
 - widgets
 - buttons
 - drop-downs (hamburgers)
 - entry boxes
 - scrollbars
 - pop-ups
 - tooltips
 - pinch to zoom
 - swipe left/right

20/28

When first starting on a project, it may be tempting to invent a new way of presenting information. Until you have gained experience, it akes good sense to use those methods that have stood the test of time in the hands of other developers.

This is a list of the common tools that appear in many interfaces; the last two are rarely found in scientific interfaces yet - but they will be.

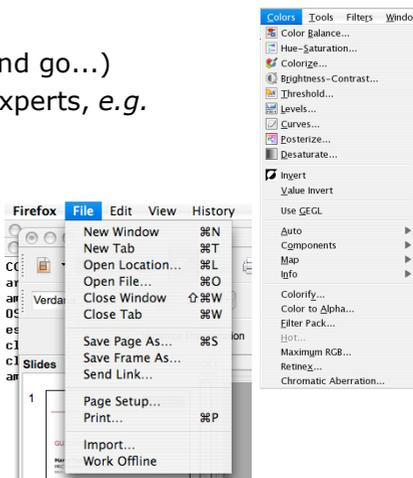
Main windows

- Should be the first window to appear
- Because they are persistently visible, their appearance should not be jarring
- Navigation between the individual tasks should be straightforward
- Contain the normal range of major actions
- Main windows have "sovereign posture"
 - monopolizes user attention for extended periods
 - remember that typical user is an intermediate
 - ∴ don't have the expert options visible

When your interface starts, you will have a main window - since it is likely to be present all the time the GUI is running, it needs to be relatively neutral in appearance, so bright primary colours are best avoided. There should not be too many (how many is too many? Only you can decide) choices at this stage, and it should be reasonably obvious which paths are the most common ones to follow.

Pop-ups

- Grab attention more
- "Transient posture" (*i.e.* they come and go...)
 - for advanced intermediates and experts, *e.g.*
 - more complex options
 - rarely used options
 - support sovereign applications
 - enhance functionality
 - can put complexity here
 - *e.g.* File menu, Gimp color menu



22/28

Pop-up windows, on the other hand, are normally hidden, so they can contain much more functionality. For many users, the functions that they provide will rarely be used - so it can be a worthwhile exercise to customise their appearance and options available according to the preferences and experience of the user. A good option can be to customise them further according to the current state of analysis, but this can be very difficult.

Tooltips - can be useful, but.

PHENIX preferences - Graphics

If checked, PHENIX will show the current model and maps during refinement, updated continuously after each major step.

PHENIX preferences - General

Some changes may not take effect until you restart

User level: Basic

Job list

Job/File

115 REFMACS 3.3

Job 115 Refinement - REFMACS REFMACS 3.3 R=0.16 RFree=0.26 Finished Right mouse click for options

CCP4-7.0.043

Clone job Run Run on server Clone job

66 REFMACS

65 Manual COOT

64 BUCC Job 65 Manual model building - COOT

63 BUCC Finished

62 REFMACS Right mouse click for options

useful extra information

developer got bored...

extra words, no more information

23/28

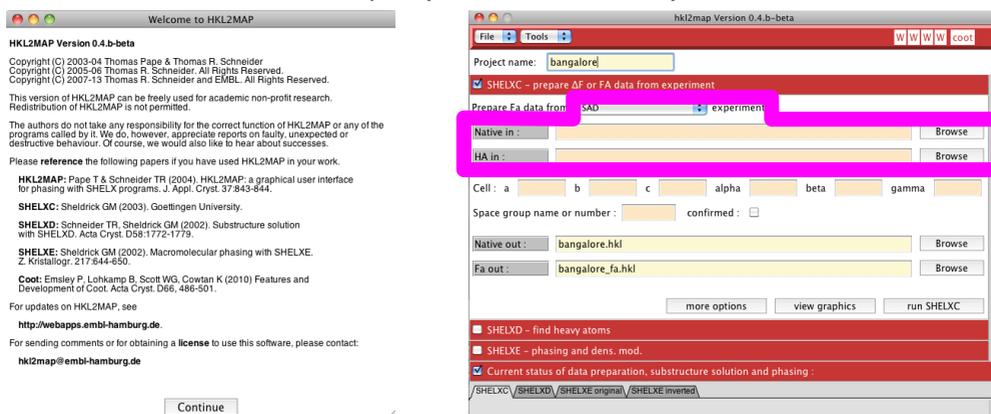
Tooltips are small pop-up windows that give extra information about the function of a widget when the mouse cursor hovers over the widget for a short time.

The examples here (in *ccp4i2* and the *phenix* GUI) are good (on the left), bad (the middle examples - these give no extra information than the button label gives so they are completely useless) and indifferent (those on the right - using different words they tell you what is already obvious and add little extra information).

The message here is simple. If you are going to use a tool like this, make sure you are really adding value.

Think about what *needs* to be there

e.g. *hkl2map* is a really nice, easy to use interface that gives useful feedback but extensive output (from SHELXC/D/E) is also available



Customise the display for the current task

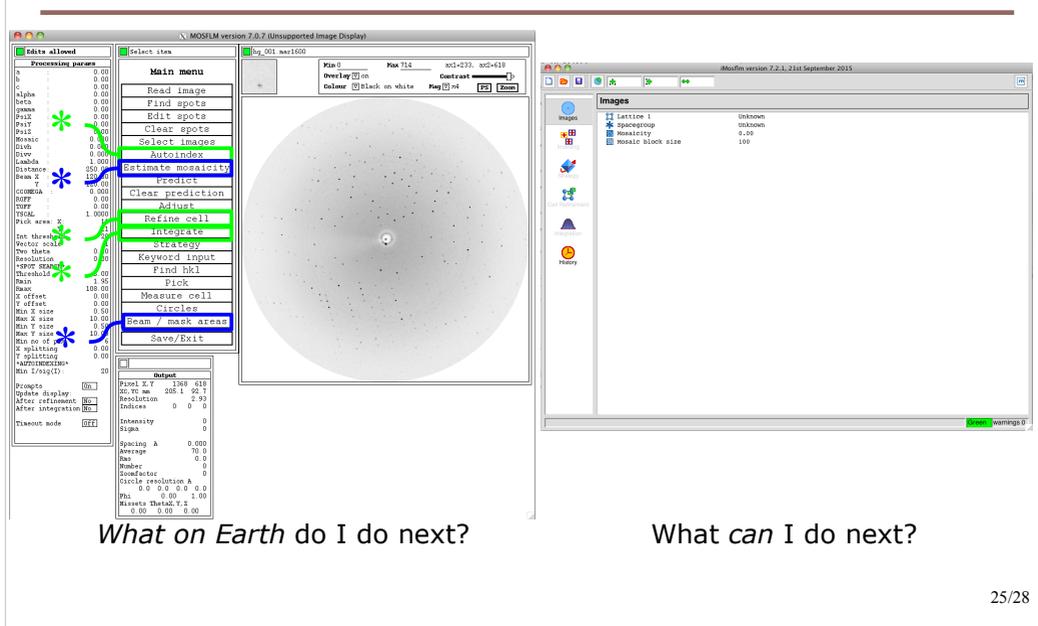
24/28

Something that no user of any application welcomes is the copyright notice with references that appears every time you start a program and that needs to be dismissed. In this example, I start the interface and dismiss the copyright notice window without ever reading it. Not once.

As the slide says, *hkl2map* is a really nice interface to the *SHELXC/D/E* pipeline that also starts *Coot* for you, but it does have flaws. A little extra customisation would have removed the first entry box for a reflection file (labelled "Native in") because it isn't used for SAD phasing. While it is there, it causes confusion -

- do I need to fill in both entry boxes? (answer is "no")
- can I fill in both entry boxes? (answer is "yes, but whatever is in 'Native in' will be ignored")
- can I fill in just "Native in"? (answer is "no")
- can I fill in just "HA in"? (answer is "yes, in fact you need to fill this one in to proceed")

Make life easy for users



Think about making life easy for your users. On the left is the *Mosflm* interface from the early 1990's; there is really little clue about what you can do and what you need to do to proceed. Obviously (is it obvious?) you don't need to read in an image, because one is displayed already, but where do you go from here?

There are 20 action buttons, each of which is functional but most of which will lead to an error warning until after indexing has been successful.

There are 39 fields on the left that can have numbers entered into them, and five buttons that control the display.

In reality, pressing the three buttons outline in green will process a dataset (after you answer some questions in subsequent pop-ups), and using the two buttons outline in blue will customise the processing to give "pretty good" processing.

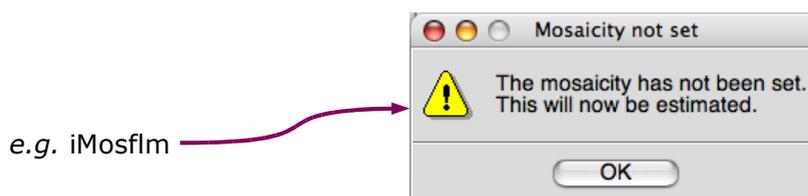
iMosflm, on the right gives you few choices; since there is no image displayed, it is not an intellectual leap to decide that you should add an image (or series of images) to process, at which point the display changes to give further choices.

Avoid adding excise

Actions which are necessary to complete a task but do not contribute directly to its progress e.g.

- confirmation boxes
- error messages that have to be dismissed
- navigation through menus

These constitute "excise", and interrupt the flow of achieving the goal



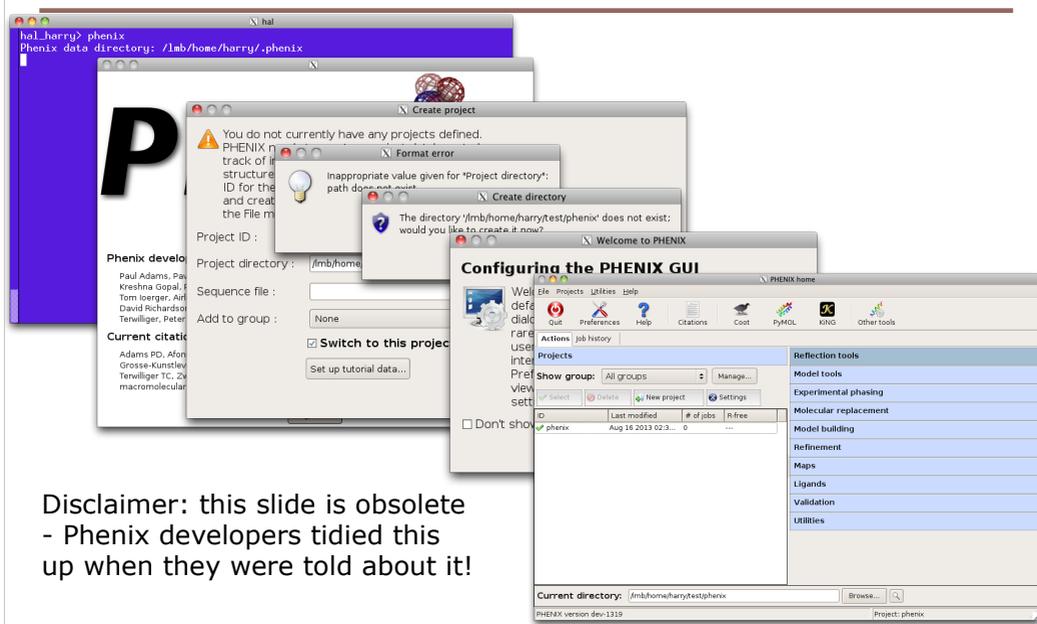
26/28

Excise is the term given to those actions that interfaces require you to perform but which do not actually result in making any real progress.

This is a good (bad?) example from an old version of *iMosflm*. After you have indexed, if you go straight to the "Refinement" or "Integration" task without having set the mosaicity in the entry box in the Autoindexing task or having run the "estimate mosaicity" option, this box pops up. If you press "Okay", the mosaicity estimation runs. There are no other options, apart from closing the pop-up with the red "close" button; if you do this, the Refinement or Integration is likely to fail because (a) *Mosflm* needs a non-zero value of the mosaicity to perform either of these tasks and (b) the automatic image selection for both tasks depends crucially on the value of the mosaicity.

The answer was to perform the mosaicity estimation automatically on moving from Autoindexing to either of the subsequent data processing tasks if it had not been done or if the value was 0°.

An example of excise...



Disclaimer: this slide is obsolete
 - Phenix developers tidied this
 up when they were told about it!

This is a particularly nasty example of excise that used to exist in the *Phenix* GUI that has been fixed long ago, but it makes my point particularly clearly.

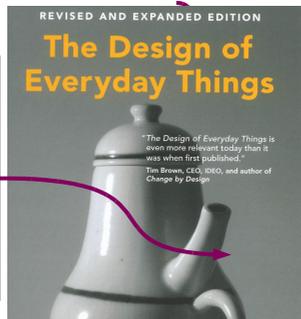
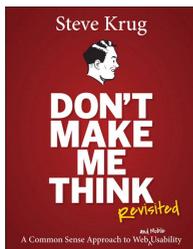
- 1) First off, the copyright notice appeared and had to be dismissed. Then
- 2) I had fill in details about my project (I hadn't decided yet, and I didn't need to fill this in at this point),
- 3) then a warning pop-up told me that I hadn't filled it in - I had to dismiss this
- 4) then it told me that my *Phenix* working directory didn't exist and asked if I wanted to create it - choice of "Cancel" or "Okay" (presumably having got this far I do want to proceed)
- 5) then it tells me it's configuring the GUI for me (though I can still cancel if I want to)
- 6) Finally, with the sixth window I get to the main window that allows me to start work.

Serious about GUI design?



- Caution -
- Contain *very* strong views
 - *Essential* reading for the serious interface designer
 - Will tell you that your current interface is probably all wrong...

ISBN 978-1118766576
currently ~€35
More on *design*



978-0262525671
currently ~€20
on the *psychology of design*
sp. Chapter 6 - *Design Thinking*

978-0-321-96551-6
currently ~€22
read, aimed at web design but useful for GUIs

28/28

Here are a few books that are widely cited in discussions about interface design.

"About Face - the Essentials of Interface Design" is now in its fourth edition, and is as close to a handbook on the subject that you are likely to find.

"The Design of Everyday Things" is more about the psychology of users rather than the actual design process; it puts the responsibility for bad design and user problems arising from this squarely on the shoulders of the designers.

"Don't Make Me Think" is specifically about design for web interfaces, but the messages it contains are transferrable to GUI design in general.