

Automated data **collection and **integration****
 Rob Hoof, Bruker AXS BV, Delft
 rob.hoof@bruker-axs.nl

© 2004 Bruker AXS. All Rights Reserved.

In this talk I will be jumping between IT background for crystallographers, and real crystallographic coding. I will show no direct crystallographic algorithms, but will focus on how they can be programmed in a future-directed way.



Early History




An instrument manufacturer builds the **instrument...**
...The **crystallographer makes the **software****

© 2004 Bruker AXS. All Rights Reserved.

A long time ago, there was an instrument manufacturer which was basically a mechanics workshop. There was some firmware in the instruments, but all application software was written by the crystallographers that were using the equipment.

1

2




History

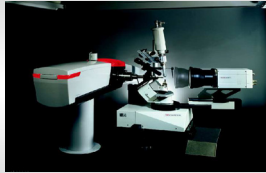
Since then, basic crystallography has progressed from **Art, via **Science**, to **Technology****

© 2004 Bruker AXS. All Rights Reserved.

The instruments as shown in the previous slide were used when crystallography was mainly a goal. More and more crystallography is moving from goal towards a means, a method, to reach other goals like the explanation of a reaction mechanism for an enzyme. It is used as a technology.



Recent History



An instrument manufacturer builds the **instrument...**
...with the **software**

© 2004 Bruker AXS. All Rights Reserved.


More recently, the instruments have become a lot more complex, and a lot more accurate. The added complexity and accuracy requires a lot more instrument-specific programming. Also, because much of the application software is now no longer subject of the research being performed with the instrument, it is supplied by the manufacturer of the instrument.

3

4

What software (for SCD) ?

- Mount and center crystal
- Check whether it diffracts
- Determine a unit cell
- Check whether there is unexplained diffraction
- Make a data collection strategy
- Collect data
- Integrate data
- Correct data
- Solve structure
- Refine structure



More and more aspects of the software are integrated into the standard program tools, and more and more different crystallographic techniques progress from the research stage into something that can be routinely performed. Vertical as well as horizontally the software becomes more and more integrated.

Structure determination of twins and incommensurate structures is now commonplace, integration of quasicrystals not yet.

More Progress

As more progress is made in the field, more and more applications are delivered with the instrument.



Roughly half of the development of a new instrument now is invested in software.

Hardware dependence?

A large fraction of the data collection software can be made independent of the exact instrument



For this to work, the easiest would be if there would be no assumptions about the hardware in the application software.

This is impossible: the software that once was used to integrate point detector data in small molecule SCD could not have foreseen the changeover to area detectors.


Many vendors have made a change from one-axis goniostats to 4-axis goniostats with accompanying changes in the code

But: if the design of the software is well thought out, it is possible to minimize the problems keeping the software running

Motto of industrial software design

You can not predict the future.

**What will a machine look like in 5 years?
What will it be able to do?
Will the software be able to handle those changes?**



If the future would be known, we could perfectly plan the software such that it could accommodate “future” changes. But even if we can not foresee the future it will be possible to structure the software such that it will accommodate changes in instrument design and experiment design with minimal efforts.

It is that structure that I would like to focus on in this class.

Motto of industrial software design

You can not **predict** the future.

Instead you must try to **foresee** all possible different futures



The lack of possibility to predict the future is a nice parallel between software design for crystallography and politics.

A recent paper in Scientific American that describes the importance of future developments in political decision making gave a nice alternative to predicting the future as the basis of a decision: predict a wide range of different futures, and make sure that the decision is performing well in as-wide a range of futures as possible.

This trick is valid as well for crystallographic software development.

Application

What determines our **adaptability**?



So what does determine how adaptable we are towards future developments? To find out, we will have a look at some data from a diffraction machine.

9

10

What are Data?





This is a diffraction image from a CCD instrument.

Is this “data”?

What are Data?

	214	215	216	217	218	219	220	221	222	223	224	225
288	54	66	48	68	59	49	38	36	36	45	56	50
285	38	58	60	56	55	57	62	40	40	45	44	48
284	46	56	55	54	53	76	76	55	40	44	50	58
283	37	52	53	45	106	168	156	109	78	61	69	42
282	38	40	48	123	334	643	497	204	111	74	80	47
281	42	52	32	216	715	1498	1126	341	122	83	56	52
280	47	51	75	198	846	1703	1288	418	150	67	52	47
279	45	55	85	136	462	956	616	303	107	57	83	58
278	45	62	57	88	162	320	286	161	86	55	49	49
277	37	41	56	69	91	96	98	98	64	54	48	46
276	43	42	56	68	51	75	65	59	62	44	46	40

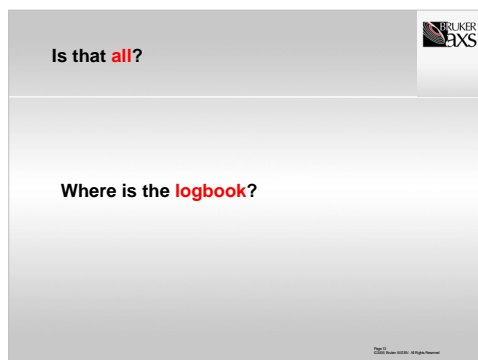


This is a part of the previously shown diffraction image, representing the exact numbers underlying the image. It represents the area around a single reflection.

- Can we integrate this reflection?
- How would it look in a HKL file?
- Which reflection is this?
- Is this “data”?

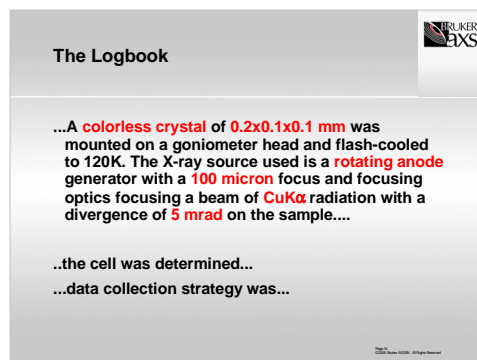
11

12



I indeed think that the answer to the question asked in the previous two images is “yes”: the diffraction image **is** the data. But the next question is “is data all we need”?

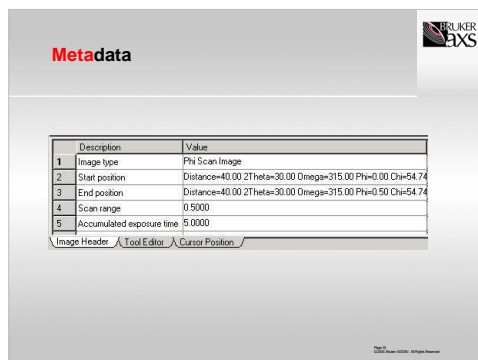
What we are missing is **HOW** this data was obtained. The logbook of the experimenter.



What I mean by the logbook is basically what all crystallographers write or at least used to write in an “experimental” section of a paper about how the structure determination was performed.

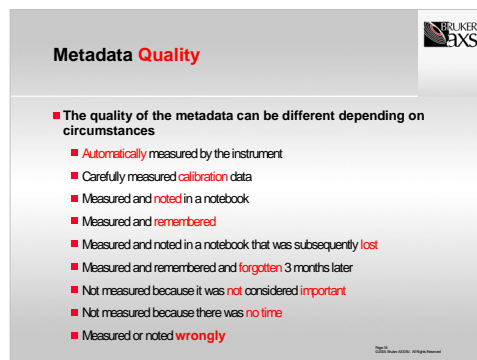
Given here are just a few examples.

This information is essential: not just to be able to repeat the experiment (*i.e.* for scientific reasons), but plain and easy because without additional information we do not know what reflections are visible on each diffraction image, and structure determination is impossible. All of the additional surrounding information should be present in a logbook.



In information technology, the name for the information surrounding the data, the information that describes **what** data we have is *metadata*.

Not only the logbook of the experimenter, but also the logbook of the instrument itself provides metadata. For a 2D diffraction image, this information is often stored in the *image header*.

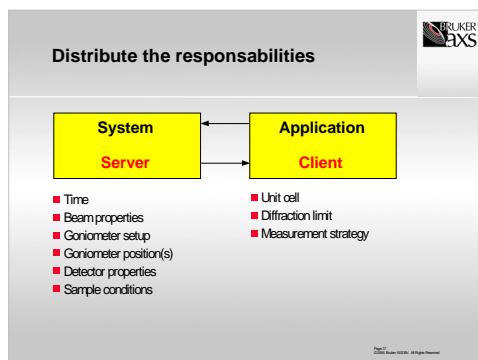


Metadata comes in different qualities. Some metadata is better than others. Lets go through a list of metadata from good through bad.

Even though the later items in this list may come across as humor please be assured that I mean this very seriously!

• Sometimes after performing experiments at a synchrotron people come back to the lab with a data set, but they do not remember the orientation of the phi axis of the goniostat.

• More involved: if the primary beam position is given somewhere in the metadata, this needs *metametadata* to be interpretable: x,y or y,x; mm or pixels, up/left or down/right as positive axes? And the values can be there from a previous determination three days ago...



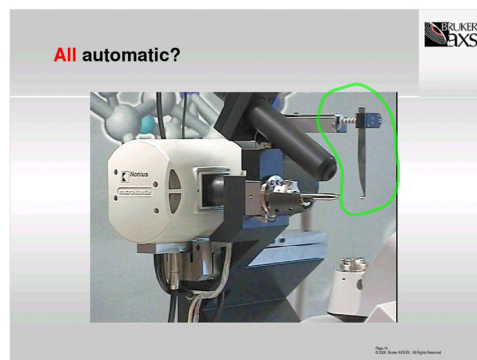
Another way to classify the metadata is to see who should be responsible for its determination. A diffraction experiment can be separated into two parts. In fact this is how we have written software where these two parts are written as a client connecting to a server program.

Responsibilities of the server include everything that is part of the instrument, and responsibility of the client is everything that is part of the experiment.

Not all is so clear:

- where is the crystal shape stored?
- The data-collection strategy depends on the hardware configuration.

17



For a number of years I have been involved in the development of the KappaCCD, an instrument that automated most of the motions and measurements of the system. The metadata could therefore be quite trivially recorded.

But not all of the instrument was wired: the beamstop, indicated here, could be manipulated by the user.

18

Manual beam stop

- Beam stop is **movable** and exchangeable. But: **No sensors**.
- Which beam stop and where it is positioned is entered into the program **by the user**.
- On the server side, the information is used for **collision avoidance**
- On the client side, the information is used to calculate which parts of the detector can not be trusted (**obscuration**).

Does every user always set these parameters?

The beamstop could be exchanged and moved by the user, but there was no sensor (except for a “beamstop present and in place”).

The user is supposed to note the change of beamstop identity and/or position to the program, but this is often forgotten.

If it is, the metadata registered in the image header regarding the beam stop is incorrect, and the unexpected position of the beam stop can cause an annoying collision with the detector.

19

The best quality metadata

The best quality metadata....
....comes from the **most automated and integrated system**

An important conclusion can be drawn from this: The best logbook is one that has been kept by the instrument computer itself!

Metadata that has human involvement is by nature more unreliable than 100% instrument-determined metadata.

Of course, to have instrument-determined metadata one needs the instrument to measure as much as possible by itself. Many, but not all of the parameters stored as metadata are also suited for automation. As the MAR dtb system shows, a beam stop position can be motorized as well, and the size of the beam (the collimator is another thing that is not automated in a KappaCCD) can be determined by motorized slits.

20

Prepared for the future

How to be prepared?

- We make sure that we have appropriate **metadata**.
- We keep the metadata at the appropriate location

Is that sufficient?

- We also have to make sure that the actual software will survive the years.

OK. So, a necessary condition for making future-proof software is to keep appropriate meta-data, and to keep that meta-data at the appropriate location.

This, however, is not a guarantee for future success of our current software. We need to make sure that the software is written in a future-proof manner itself too.

21

Software design

- **As few assumptions about the environment as possible**
- **Run on different platforms**
- **Use a productive programming language**
 - Performance is less important than you think
 - Do not spend any time optimizing rarely executed code
- **Use the right programming techniques**
 - Readable code
 - Portable code
 - Documented code
 - Object oriented/modular code
 - Group-development: code reviews, coding standards, sharing modules.

Often, software written at a lab for internal usage contains, at least initially, a lot of hidden dependencies on the lab environment, like the location of other software package on the system. For future-proof software this is not a good idea.

Relying on a single hardware platform is also a bad idea. e.g. look at the history of DEC's VAX computers, and the history of bus architectures. Anything that was critically dependent on one of the dead technologies has died with it.

Optimize the time spent programming vs. the computer time saved: assembly language routines may be fast, but programmer time may be more expensive!

These parts of the design are everyone's own responsibility. In the following part, we will discuss some examples of programming techniques.

22

Intermezzo

- **Is hand-coded assembly really the fastest code?**
 - Is an assembly coder likely to use the fastest existing algorithm?
 - Did you ever use bubble-sort?
- **Is python code slow?**
 - Or is 99% of the execution time in a single loop that can be optimized?
 - Did you ever profile your software?

Using existing libraries gains you speed and flexibility that was coded by experts.

Someone writing low-level code is likely to choose algorithms he knows, and spending large amounts of time optimizing the execution speed. Simply using another algorithm unknown to the user may gain an order of magnitude in speed.

23

Example of future-proof coding (1)

"Count the total number of 'a' characters in all descriptions in a dictionary that contain at least one 'z' character."

```
def counta(dictionaryfile):
    totala = 0
    for line in dictionaryfile:
        if line.startswith("description:"):
            if 'z' in line[12:]:
                totala += line[12:].count('a')
    return totala
```

This is an example of a simple programming exercise that has been solved exactly as stated.

The subprogram is written in pseudo code, any resemblance to existing programming languages is purely coincidental.

The hidden assumption of the programmer is that none of the conditions set will ever change.

24

Example of future-proof coding (2)

- **This subroutine needs changes if:**
 - The dictionary is no longer on a file
 - The dictionary is stored in a set of files
 - The dictionary file is no longer in a line-by-line format
 - The record structure of the dictionary changes
 - The character encoding of the dictionary changes from ASCII to UTF16
 - The conditions placed on the counting are changed
 - Something else than counting would take place
 - We would need to count 'c' characters

A lot of imaginary future developments will require changes to this code.

In fact, the differenced can be grouped into five different groups:

- The location of the data
- The format of the data
- Data filtering
- Data analysis
- Parameters of the analysis

Does this suggest how the initial code example can be improved?

We need to split responsibilities

- **Four components:**
 - Data source
 - Data decoder
 - Data filter
 - Data processor with parameters

Yes! The software should be split into independent components that work together to perform the given task. A change in the environment will then require only a change in one of the modules.

Advantages of the modular approach

- **Multiply instead of add up; re-use instead of copy code.**
 - 4 sources, 4 decoders, 4 filters and 4 analyzers make 256 different combinations, not 16 (or 4)
- **Algorithm abstraction**
 - While writing a filter, it is not important to know the source of the data
- **Parallel development**
 - Once the interfaces have been defined, different people can work on the code in parallel without discussing the implementations all the time.
- **Expert developers**
 - Let a database manager write good database access code, and an dictionary specialist write good filters.

Several advantages are gained by this modular approach.

Lets apply factoring to crystallography

- goniometer positions
- goniometer motions
- other instrument operations
- low-level communication protocols
- high-level instrument protocol drivers
- crystallographic data format decoders/writers
- data collection strategies
- diffraction calculations and X-ray tracers
- ...

- Goniometer positions: get from and set to hardware, ask whether hardware can reach a position, convert between different types.
- Operations on the hardware: scans, setting generator or cryostat. These can be combined into an experiment through a common application programmers interface.
- Communication protocol at a low level: connection protocol to the hardware, high-level is command interpretation.
- Data file formats: Program interpretation of different reflection file formats or diffraction image formats such that they can be used interchangeably.

The actual implementation of each module is completely hidden behind a common interface. It is therefore possible to use external programs as well as internal implementations in a completely transparent fashion

Apply factoring to diffraction images

- Image transfer protocols
- Image file formats (read/write)
- Image sources
- Calibration filters
- Image processors
- Image display
- Image filters for display
- Image transformations

We will use diffraction images as a more involved example of how we can factor code into different modules that can be made responsible for their own part of the work and to see what kind of gain we will have when we do that.

Listed here are a number of different tasks that can be performed on diffraction images.

Structure

```

graph LR
    Source[Source] --> Sink_Filter_Source[Sink Filter Source]
    Sink_Filter_Source --> Sink[Sink]
  
```

Now, instead of performing all operations that need to be performed on a diffraction image in a program in a simple succession of calls, we can see the program structured as modules that pass copies of the data and metadata around.

We can see three different kinds of modules on an abstract level: sources, filters and sinks. Since a filter can be seen as a both a sink and a source, the protocol that is used for communication of an image from one to the other module (indicated with the arrows) is always the same

Structure

```

graph LR
    Reader[Image file reader] --> Correct[Correct ADC0]
    Correct --> Display[Display]
  
```

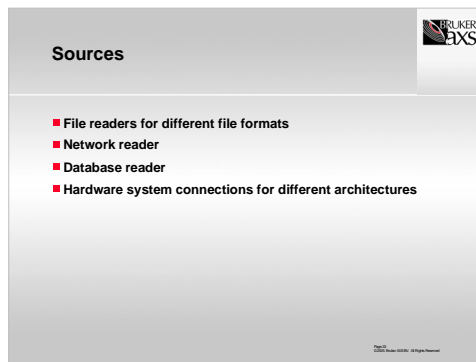
Here is a simple example of such a train of three.

The "ImageSink" interface

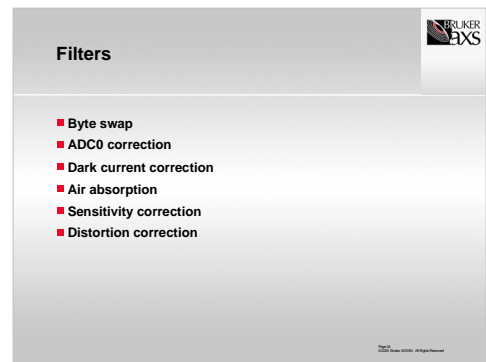
- Begin()
- BeginSeries(metadata)
- BeginImage(metadata)
- Header(metadata)
- Frame(data)
- EndImage()
- EndSeries()
- End()

The communications protocol consists of a begin and an end message with zero or more series of zero or more diffraction images in between.

An *imagesink* is an object that can accept these messages (has these methods). An *imagesource* is an object that can emit these messages (call these methods).



Traditionally, application software always reads data from a diffraction image file. But if we have the flexibility in our software, we can think of several kinds of sources to be implemented. By the modularity magic, any program can use any source!



Many filters are traditionally implemented in a monolithic way. Separating each filter operation into its own module makes it possible to compose filters easily in ways that were not imagined before.

One example here is the way a sensitivity correction is measured for a CCD detector. This is normally done with a flood-field image. Obviously, the calibration image should not be corrected for detector sensitivity, but it should be corrected for other effects like the dark current of the detector.

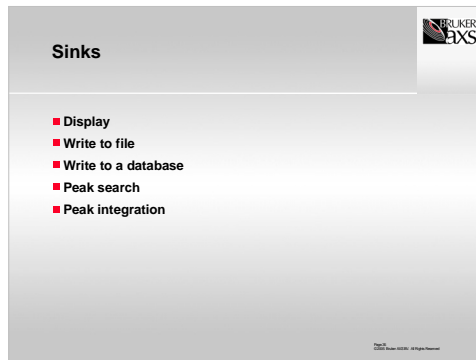


Image sinks are normally the operations that an application is written for.

In the modular case the actual procedure will use object or module composition to build an application.

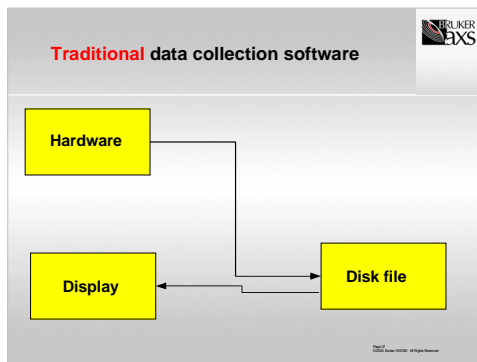
Note how easy it is now in theory to make a program that converts between any two different image file formats! One just needs to plug an image file reader source to an image file writer sink....

Different applications will use different compositions of sources and sinks. There is no forced code duplication anywhere.

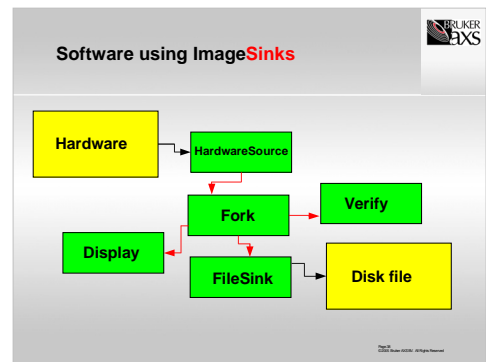


For house-keeping, we can make some special sources and sinks that make the composition of these objects even more flexible.

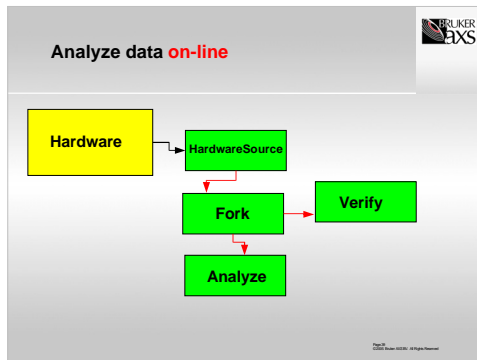
Protocol logging and verification sinks make it possible to make debugging the communication between sources and sinks a breeze.



What do we gain by the source/sink approach?
 This is a simple diagram that shows how traditional data collection software works. There are two separate program threads: one is connecting to the hardware and is writing all the measured data and metadata to a disk file, and a second one is reading the disk file to display it on the fly.



The same software written with the image sink paradigm could look like this. Red arrows are the image sink protocol. Fork is a house-keeping module, Verify is a debugging module.



However, thanks to the imagesink protocol, it is now also possible to compose a completely different application that analyses data on-line without ever storing the information on disc. Gluing the objects together in an object-oriented programming language is as simple as drawing the arrows in a powerpoint slide!

```

On-line analysis in pseudo code

hw = hardware.connect()
analyzer = AnalyzerSink()
verify = VerifySink()
fork = SinkFork([verify, analyzer])
source = hw.Source(fork)
hw.go()
or
hw = hardware.connect()
hw.Source(SinkFork([VerifySink(), AnalyzerSink()]))
hw.go()
  
```

This is the diagram of the previous slide written in pseudo-code, both in a way that each of the objects stays available to the program, and in a more compact way that hides more of the internal structure of the sinks.

And now....

Now that we have all this beautiful **code**, we make it accessible through the **buttons** of a GUI

...Or...

41

When is a GUI really handy? (1)

A GUI is really **handy** for operations that you perform manually **once a month**

An operation you perform three or **thirty times** in a row screams for a programming interface.

An operation you need to perform **automatically** can not have a GUI

42

When is a GUI really handy? (2)

A GUI is really **handy** if it can help you perform basic operations

A GUI that supports **every possible** operation that might be useful to someone someday is a **nightmare** for casual users

Typical examples of GUIs: word processor or spreadsheet.

Windows itself, with 500-page books with “Everything you ever wanted to know”, and magazines full of “tips and tricks for using Windows” is a good example of why it is not advisable to use a GUI for every operation.

Scripting is good!

43

When is a GUI really handy? (3)

A GUI is really **handy** if the control flow is in the hands of the human controller

A GUI can be **annoying** in cases where the flow is decided by the program, such as in case of **automation**

If scripting does the work, the demand for a GUI becomes minimal. One will still need to look at the data, but that is not really a GUI.

The real power of a GUI program is unleashed when all operations of the program can be performed in any order, depending on what the user wants to do next.

Use intelligent defaults: Make sure that if the program has a good idea on what the user would want to do, that the default parameters are properly filled in!

44



Extension language?

- We do not want a GUI that supports every possible operation.
- We do want to be able to perform the same operation many times in a row in an automated fashion.
- We want to be able to automate decisions and control flow.


Is the solution an **extension language**?

Will we **write our own** extension language?

In an earlier stage we have seen that we need to be more careful with programmer time than with execution time.

Making an extension language has been very popular for a while (e.g. in the Enraf-Nonius CAD4 software) but it is not very efficient, and it is very difficult to write a good and complete extension language.

45



A programming **library**!


- We choose a programming language
- We make a crystallographic library accessible from that language
- We write command line tools that use the library
- We write GUI's that use the library

Using a GUI designer from the beginning is a **dead end!**

There are very nice developer tools available now, where the development consists of

- Designing a GUI
 - Adding subprograms under the buttons
- Programs written in such a way are terminally attached to the GUI: it will never be possible to run the programs from a script, and extremely difficult to change the GUI toolkit at a later stage.
- If a GUI developer is used, it should be made as a very thin layer over the top of existing code.

46




A **users** point of view

- For normal **weekly** usage:
 - GUI access
- For routine daily or **hourly** usage:
 - Command line tools
- For advanced **special-purpose** usage:
 - Command line tools
- For **very advanced** usage:
 - Write a special application

Special applications can be used for routine operations as well if required.

Special applications could be either command line, or, if really needed, a GUI.

47





Questions the **day after**....

- I know I changed something yesterday, what was it again?
- Why does it ask me to save? I didn't change anything?
- How come I can't reproduce these numbers now?
- Did I integrate all the data?
- Who the @#% did touch that parameter?

48



And their **answers...**

- Logging
- Access control
- Auditing



Formally

- Good Laboratory Practice (GLP, GxP)
 - Keep a proper action log
- A step further: CFR21/11
 - Requires logging in to the control software
 - Requires an unmodifiable audit trail
 - Requires digital signatures on all data
 - Requires that programmers know about CFR21/11





In its formality CFR21/11 adds annoyance to the process.

Pharmaceutical companies are bound to CFR21/11, especially in their process control, and a little less in their R&D.

Conclusions

- Try to **predict** the future
- Describe all things that **could** change in metadata
- Avoid hidden dependencies and assumptions
- Modularize, and **hide** implementations
- Do **not** make **only** a GUI



Hidden assumptions are the goal of a programming exercise in the workshop.