Tutorial 1 Answers

Question 1

Complexity
Software in it what is has to do, is often essentially complex. We can think of software which is accidentally complex such as a large scale e-commerce system (simple in concept, complex in implementation) as well as systems which have inherent complexity such as speech or picture recognition systems.

Conformity
Software has to confirm to the context its operating within, so for example a piece of software has for the benefit or tax system has to confirm with the benefit or tax rules and systems. This also includes conformance with other software systems which provide other services such as payment systems, external hardware. Errors with conformance are a common cause of software failure.

Changeability
Requirements for software are notoriously fluid changing all the time. The fact that software can be changed very quickly can cause many problems in terms is poorly tested. As a software system gets larger ensuring that the system doesn't suffer in terms of regression faults becomes harder and harder to test for.

Invisibility
Unlike a building design or a piece of hardware, a lot of the complexity of software is invisible. Since the representation of software in general is 2-D (a listing) but its actual manifestation is more like 3-D since there are links (couplings) possible between any 2 or more elements within the system. When we try and represent software visually often it is very difficult to understand the complexity of the picture produced.

Question 2

Accidental complexity is the result of the design and implementation of a system which in concept at least may not be particularly complex. So if we think of a system which is an online e-commerce site, conceptually it is very simple, but it might have to connect to many different payment systems. It might have to deal with very many customers so have a system of scaling/clustering the database. Inherent complexity is complexity within the problem itself, so for example an encryption algorithm or a vision or AI system. Accidental complexity can often be factored out by re-using an existing system which handles scaling, inherent complexity is a lot more difficult to deal with as it is part of the problem domain being solved.

Question 3

Better HLL languages

These help reduce complexity be allowing expression in easier to understand more natural languages, we can compare assembly language.

```
mov cx, 100  ; set the counter
loop1:

loop loop1
```

Compared with

```
for (int idx=0;idx<100;idx++) {

}
```

We can see that the high level code is easier to read, allows for loops within loops, and makes it simpler to implement loops with different exit conditions, as this is part of the loop notation.

The other great benefit we can see is that higher level languages allow for structure to imposed in code and type checking to a carried out on data types. These types of constraints in general reduce the accidental complexity of any problem but not its inherent complexity.

OO languages

OO developments, provided a step change in ability of the programmer to put a constraint on the accidental complexity of any given problem. The use of abstract hierarchical data types, which allows the programmer/designer to define the data structure as well as permitted sets of operations. OO has been shown to allow for greater re-use of code without changing the base code. This concept of building up from a base class library has been shown to improve reuse by up to 70% over old C style projects. Other features which improve re-use and reduce complexity are:

Use of generic data types (so code doesn't have to be re-written just because the data type changes, for example within a sort algorithm).

Provision of interfaces, this allows interactions between code to be constrained before the actual code body is written, this makes code integration much easier.

Provision of garbage collection (so now the programmer doesn't have to keep a track on data pointers).

Artificial intelligence

Split into 2 types:

The first using computers to solve problems which were normally in the past solved by human intelligence and the second, rule-based programming (these are commonly called expert systems). The problem however is that the first type of AI is only specific to a particular problem domain for example speech recognition, Brook couldn't see how these specific solutions to particular problems could be applied to software development in general.

The other possible application of AI, is the use of experts systems. This is a particular approach to software development that involves the development of a set of rules, an inference engine which operates on these rules and any incoming data. One possible application of this type of expert system could be to system testing. The expert system could examine the code and depending on the structure of the code and the rule base develop a set of automated tests. The other possibility is based on the symptom of a fault, it could possibly develop a set of other tests to determine the location of the fault. The essential idea is that the expert system would be programmed with the expertise of the best programmers and be used to help to novice developers debug their development more quickly. One of the great difficulties in developing such a system is getting developers who can articulate why they take a particular approach and translate this into a simple set of inference rules.

Automatic programming

The idea behind automatic programming, is that the specification is defined in such a way that some automated method can be used to translate this into program (i.e. automatically).

This in general is hard to do since one of the hardest part of any difficult programming problem is to correctly and unambiguously define the problem in the first place. For some very simple applications such as sort routines, the definition can be expressed mathematically. For many applications the specification as such is developed iteratively from a series of prototypes and many stake holders for projects would find it difficult to express their requirements in a manner which is both complete, correct and not open to misinterpretation.

Graphical programming

This is the development of software using some type of graphic tool or graphic user interface. One of the major problems here is that no single graphical notation has been developed which encompasses all the functional elements of a program design. UML which is widely used today breaks up the design into a number of different representations some static and some dynamic which try and define the structure and behaviour of the software. However since this is not 1 notation but many and only provides limited descriptions of different parts of the software it is hard to see how this would solve all aspects of software development. There is one notable exception to this, which is the development of the software GUI, tools are readily available which allow one to design this part of the software graphically and provide hooks into the main code body.

Question 4

For this answer, pick your own favourites, these are some of mine.

Run time error checking and debugging

Many of the ways that software development has been advanced has been the use of environments which control not just the development but the run time context of the software.

So for example we have language such as Java, this runs as an interpreted language which means it can do run time checks on the code and check for errors which cannot be determined at compile time, for example:

Code which tries to write off the end of an array.

Code which tries to incorrectly assign a object reference of 1 type to an object reference of another type. In Java this type of assignment could be the result of upcasting then downcasting and could not be determined at compile time.

Example (somewhat artificial):

```
int GetMaxSpeed(Vehicle vehicle) {

  Car car=(Car)vehicle;  // this may fail at run time, with cast exception

   return(car.getMaxSpeed());

}
```

Interpreted languages high levels of run time exception handling makes the isolation of bugs a lot simpler.

Debug tools

Again this are a lot more powerful due to the nature of interpreted languages, it is simpler for a trace program to break on many and complex conditions due to the fact that the interpreter is a software entity and not the CPU itself.

Code re-factoring tools

Refactoring is an important part of any on-going large scale project. It makes code easier to read can improve its flexibility and structure. Tools which allow re-factoring without great effort make re-factoring more likely, imagine if you had to rename 2000 references to a method name by hand, you in most circumstances wouldn't do it due to lack of time. So re-factoring tools make it likely that re-factoring will be done.

GUI development
This used to be a very labour intensive and error prone part of the development process. This has been improved greatly be the use of the following technologies:

Web convergence
Develop all UI elements using web technologies this makes the porting of code from Windows™ to Unix trivial. However this still leaves the problem of how to accommodate different form factors and how to deliver code which must stand alone (with the support of a network).

Mobile convergence tools
There are many products which try to provide some degree of convergence in the mobile platform field. Examples are PhoneGap and MonoTouch. These types of tool are very important in they allow the low cost porting of code between platforms and reduce the chance of bugs being introduced within the porting process.

Program verification
This is the mathematical proving that a particular program meets its specification.
The problems with this are many:
The specification might be faulty.
The mathematical proof might be faulty.
It is very labour intensive.

In summary program verification may be useful in very particular parts of a code which have a very defined functionality such a secure operating systems kernels, but are do not provide much benefit to software development in general.

Environments and tools
These have made the development of complex systems somewhat easier but the payoff in terms of productivity is somewhat limited. Many of the tools we use today have been around in some form or another for many years (for example programming language specific editors, interpreted debuggers), many of the tools we use today are enhancements of these early tools. They help to tackle accidental complexity of large projects.

Question 5
The main criticisms are as follows:

The original data from which the conclusions were drawn was not published with the report. This makes it impossible to determine how the data was processed to produce the conclusions and if an alternative interpretation of the data can be seen or draw any other conclusions by producing other statistics with relation to the data.

Incompleteness
The classification of the projects was incomplete, there was the following classes defined:

Type 1, or project success. The project is completed on time and on budget, offering all features and functions as initially specified.

Type 2, or project challenged. The project is completed and operational but over

budget and over the time estimate, and offers

fewer features and functions than originally

specified.

Resolution Type 3, or project impaired. The project is cancelled at some point during the development cycle

This is incomplete since "For instance, a project that's within budget and time but that has less functionality doesn't fit any category" this is a common case and would not be described properly by the Chaos report.

Lack of context
Some projects could overrun by 25% and be still considered a success some might fail at an overrun of 5%, imaging a database system for the 2012 Olympics being delivered in September. The definition of success or failure is very dependent on context. Also some projects may be successful even if all functionality is not delivered, it depends often on what is to be considered to be the core requirements.

Question 6
f/a is the forecast over actual ratio for a given project aspect. So f/a for time to complete the project, would have the following meaning, >1 would be an under-run, i.e. project completed before time and <1 would be an over-run. With project functionality f/a <1 indicate the project has undelivered with an f/a>1 means the project has delivered more than was expected.

Question 7
By plotting f/a for many projects one can see two important aspects, the first is the overall forecasting performance of the organisation, values closer to 1, and with a low deviation from the mean indicate better performance. Secondly by looking at the mean value of the f/a values over many projects we can see if there is any bias in the predictions. A long term mean for f/a for time to complete or 2, would indicate on average the organisation over estimates their time to project to complete by a factor of 2.

Question 8

EQF is the estimation quality factor. It indicates how close the estimation is to the actual outcome. So for example, if the total time for a project to complete finally was 20 days, and you estimated, 5, 6 and 7. Your EQF would be 20/average of the deviation from the actual.

So deviations are

20-5 = 15

20 – 6 = 14

20 -7 = 13

The average deviation is 14 (15+14+13/3), so the EQF = 20/14 = 1.4

Values of EQF greater than 8 are considered very good and in practise it is very rare to get values greater than 10. An EQF of 2 means the estimate is out by 50%, EQF of 5 the estimate is out by 20% and 10 means out by 10%.

Question 9

The original chaos report was one-sided because it reported in project failure but not success, so it reported in overruns for time and under runs for functionality but did not report in projects which under ran their time schedule or delivered more functionality than originally specified. For many projects the reason for overrunning is due to specification being added to as the project, so this exclusion would be relevant for many projects.