Scripting Practices in Complex Systems Management

Eser Kandogan, Paul P. Maglio, Eben M. Haber John H. Bailey
IBM Almaden Research Center *CA*
{eser, pmaglio, ehaber}@us.ibm.com, john.bailey@ca.com

Abstract

System administrators are end-users too. And as end-users, they develop tools, create web pages, write command-line scripts, use spreadsheets, and repurpose existing tools. In short, they engage in end-user programming activities in support of their systems management work. We examined system administrator practices in software tool development, operations, and maintenance based on ethnographic field studies at service delivery centers and data centers across the United States. Our findings suggest that software practices were mostly informal and collaborative and mixed within formal change processes; tool development and debugging were interleaved with tool use and maintenance as they interacted with live systems; and the complexity of large-scale systems and the risks involved in changing live and critical systems put increased demands on system administrators. We argue that system administrators might benefit from certain software engineering methodologies such as agile software development and software modeling.

1. Introduction

System administrators design, configure, troubleshoot, and maintain complex computer systems (including database management systems, web servers, and application servers), distributed across networks, and built on complex architectures and topologies designed by multiple vendors [1,3]. Yet system administrators are end-users, too [5]. They conduct their work using several vendor-supplied tools or develop their own tools. A recent survey suggests that 37.5% of system administrators earned a bachelor's degree in a relevant field whereas 80% of them claim to have developed much of their system administration skills on-the-job [8]. They are clearly not professional software developers. They develop software tools that they use in their work. In fact, they are at once end-users, builders, and repairers, who rely on their technical, social, and organizational skills to conduct work.

Since 2002, we have been conducting ethnographic field studies of system administrators. The main purpose of our studies was to examine system administration practices to understand underlying cost-factors of labor-intensive IT services delivery work. Profitability was a significant concern among IT service providers and we set out to study how productivity could be improved by changing practices and processes, and by developing technology and automation.

We studied more than six sites, including large corporate service delivery centers, university computing centers, and government labs in a total of 16 visits. In the course of these studies, we found end-user programming to be pervasive throughout system administration work. As end-users, system administrators developed tools, created web pages, wrote command-line scripts, used spreadsheets, and repurposed existing tools in support of systems management tasks, such as system monitoring, backup and recovery, database configuration, storage design, and inventory management. Simply put, they engaged in end-user programming activities to improve their own productivity.

System administrators, like other software end-users, usually lack professional software development training and experience. However, they oftendevelop their own tools using various scripting languages that they have learned on the job with the help of more experienced colleagues and through self-study. System administrators often share their scripts with colleagues, who in turn modify and adapt them to their particular

ACM CHIMIT'09, November 7-8, 2009, Baltimore, Maryland, USA.
Copyright 2009 ACM 1-60558-572-7/09/11…$10.00.

task, thus reinforcing the learning cycle [9]. Unlike most other software end-users, however, they tend to be very technically oriented and have deep knowledge of IT software applications and infrastructures. Although systems administrators become quite skilled at developing scripts, as in other end-user developed software, errors were prevalent[6]. In fact, problems with error messages and handling were particularly acute in both home-grown and vendor tools, with as much as 25% of a system administrator's time spent following blind alleys suggested by poorly constructed and unclear messages, often the result of the sheer complexity of systems managed [10].

Though there has been considerable work on empowering end-users in general to develop software to support their tasks, there are still significant barriers in effective use of languages, libraries, and environments [11]. Researchers are exploring new paradigms, which examine the practice from the end-user software engineering perspective [12]. The fundamental question asked is whether it is possible to bring the benefits of rigorous software engineering methodologies to end users [13].

In this paper, we consider this question of whether and how to bring software engineering methods to system administrators, particularly in the light of the software practices of system administrators based on our field studies. We argue that introducing some software engineering methodologies into the field may help, but that the context of system administration work needs to be taken into account. In particular, given the complex, risky, collaborative, dynamic, and reactive nature of system administration work, a systematic approach to software development may not be ideal or even possible. Yet there is a potential to develop solutions that carefully consider system administrators' relationship to the systems they manage – in environments where design, implementation, testing, and maintenance of software artifacts are tightly integrated and offer incremental, collaborative software development.

2. Ethnography of System Administration

We conducted a series of ethnographic field studies in IT service delivery organizations over the course of five years. Our study methods included naturalistic observations, in-situ interviews, surveys, and diary studies. In 16 site visits, we observed and interviewed more than 30 system administrators and others in large corporate, university, and government service delivery centers and data centers across the United States. We observed the work practices of system administrators, including security, database, web, storage, and operating system administrators and data center operators as they unfolded in their natural settings [1]. Two researchers participated in each observation, which lasted three to five days. One of the researchers followed a system administrator as he or she conducted work in the office, attended meetings, etc. recording interactions with the computer systems and activities with others on video. The other researcher took notes and engaged with the system administrator by conducting an initial background interview, and asking occasional questions during observation. We asked participants to speak aloud while working, to the extent possible without interrupting their work. At the end of each day, we asked questions to clarify what we observed during the day. Additionally, we collected physical and electronic materials and took pictures of some of the artifacts in their work environment.

Field studies offer insights into work that cannot be found in focus groups, lab studies, or surveys alone (see [14]). When work is examined in context, it becomes clear that people work creatively with technology to support their practices flexibly and adaptively—though systems are often designed inflexibly, people make do, naturally working around limitations and built-in constraints [15].

Here, we report a single case study where we examined aspects of the work of database administrators as it related to end-user software development. Though we have space only for one such case, it is representative of many of the practices in the field we observed through our multi-year study, including practices such as planning and rehearsal, incremental and multi-perspective verification, progressive task performance with multiple scripts, and pair-wise collaboration. While data from only one case study is presented here, conclusions are derived from many other case studies.

3. A Case Study: "It's a Rehearsal"

In Fall 2002, we visited a large corporate data center where we spent three days observing database administrators as they worked in teams. The site employed more than 5000 people providing various IT services, including data management.

Christine and Mike worked as database administrators and their responsibilities included installation and configuration of database systems, monitoring system performance and capacity, and performance analysis and tuning, depending on the particular database technology used and on the needs of customers. Likewise computer and software infrastructure requirements were driven by customers and could be unique to a specific customer.

Step	**Who**	**Action (directory /dba/work/tblmove)**	**Start**	**Dura.**	**Day**	**actual**
1	Hillary	Offline backup with SAP down after	6:00	5hr	Sat	
2	Hillary	Create new tables (prefixed with QCM) and insert data **db2 + c -tvf crttables.ddt -z crttables.out**	11:30	3.5hr	Sat	
3	Hillary	Drop indexes and constraints on original tables **db2 -tvf dropindexes.ddl -z dropindexes.out**	15:30	2.5hr	Sat	
…	…					
7	Hillary	Page/call Patrick				
8	Patrick	Test access to the table by listing its contents in transaction SE16	23:00	4hrs	Sat	
9	Patrick	Check whether the table has been accessed			Sat	
10	Patrick/Hillary	Update table statistics **Runstats.sh > runstats.out**			Sat	
11	Patrick	Check consistency of the tables			Sat	
12	Patrick/Hillary	Execute script to recreate views associated with tables **db2 –tvf crtviews.dll –z crtviews.out**			Sat	
…	…					
16	Hillary	Offline backup	4:45	5hr	Sun	
		Finish	10:00			

Figure 1. Part of the one-page summary instructions for the table-move operation: For each of the 16 steps the instructions included information about the responsible system administrator, specifics of the instructions such as commands to run, start times, duration, and day of the operation.

Below we report our observations of Christine and Mike as they rehearsed a "table move" operation using IBM DB2© data management software on AIX for SAP© application tables. They worked side-by-side for hours as they practiced moving large data and index tables from a tablespace that was reaching the maximum space limitation to two new tablespaces. This was a new customer account for Christine while Mike had provided services for this customer in the past and was in the process of bringing Christine up-to-speed and turning the account over to her. Though Christine was an experienced database administrator, this was an important customer, and both wanted the transition to be as smooth as possible. Mike was always there physically in her office or virtually from his office or at home helping Christine throughout the operation.

3.1 "It is easier to back out if you catch it early on"

Rehearsing critical operations was a common practice among database administrators. They often would test their procedures on multiple servers, such as sandbox, test, and consolidation servers, which increasingly resembled the system configuration and data of the production servers. Only when procedures were tested progressively through all these servers, would they propagate changes to production servers in a highly restricted manner and in the limited time allotted.

The table-move operation consisted of several steps for creating new tables, including dropping old indexes, creating new indexes, renaming new tables, among others. Christine inherited a twenty-page document with detailed instructions and sample scripts from Hillary, who had performed this operation in the past. She also kept a one-page summary of the plan close-by (see Figure 1). This summary not only included tasks to perform but also had start times and duration to help them figure out how long tasks take and whether they would fit in the allotted change windows. While some of the instructions were fairly detailed showing all the parameters of the commands, some were fairly high level descriptions of the tasks. Some were just verification steps without much detail, some were purely coordination steps since often such large operations had tasks performed by several administrators collaboratively.

The twenty-page documentation was fairly detailed. There were specific instructions on how to produce the individual statements; for example, she ran the `db2look` command to find which columns to use in the table-create statement. It also included several notes capturing others' experiences, such as typical execution times, suggestions, explanations, and mandatory to-dos.

In all, Christine wrote seven scripts. Rather than creating one script that did everything, she preferred to have multiple scripts, as it helped her identify and isolate problems more easily. In fact, there were explicit steps where she needed to check status, verify changes, etc. (see Figure 1):

***Christine**: You have to check the output after each step in case there is any error. It is easier to back out if you catch it early on.*

The first script Christine wrote was to create tables. Using the sample script code, she customized it for her specific environment by changing server names and such. Over time, she and others developed common scripting practices that helped them to avoid some problems in advance. Mike explained that inheriting these (sample) scripts essentially passed on these practices:

***Mike**: If the index has special characters then index name has to be in double quotes. If not, it fails. What the heck? Now, it is better to put the index in double-quotes, just for the heck of it. Some of the headaches that we ran into before, are [now] in the procedures for the script.*

As Christine created her scripts, also put her name as the file suffix. She explained that this helped her easily identify her scripts among all others in the script directory:

***Christine**: I have .christine at the end. On the other system, where I created [the scripts], there were tons of scripts. So, that is how I identify scripts that are mine, when I ftp them over here.*

Indeed, the directory `/sapdbawork/dbawork/tbmove/move2002` had all the scripts she created. The file structure was arranged such that all the SAP-related scripts were in `/sapdbawork/dbawork/`, and from there, they created subdirectories for specific tasks, such as the table-move (e.g. `tbmove`), and the specific instances of these tasks (e.g. `move2002`).

Christine had a window of ten-hours to complete the task. She told us that once scripts were validated on consolidation servers, doing it on the production servers would be fairly straightforward:

***Christine**: I will just do a global change from LC0 [consolidation server] to LP0 [production server]. That will be it. That is one of the reasons why we are actually doing it on consolidation before production because 99.99% it is the same script. By the time it gets to production, your scripts are pretty much set in stone.*

Production work was scheduled for the following day, Saturday. During the consolidation work, Christine was also on-call. In fact, she was called frequently in to phone meetings, responded to issues on other accounts, and consulted others – all as part of her on-call duties.

3.1.1 Analysis

In this part of the story, we observed Christine's efforts to prepare for a deployment of a configuration change to database tablespaces. Particularly interesting were the practices around planning and rehearsal. They developed, deployed, and tested code progressively on multiple systems. This was not surprising given the highly risky nature of the work – any mistakes made on the actual production systems would be seen by the customer, and perhaps cost the customer time or money. In this case, rehearsal was practiced very carefully and diligently across the organization, and there was the infrastructure and organizational support. As Christine said, by the time it gets to the production servers, you expect everything to run smoothly. To support this, they developed coding practices where it was easy to promote a script from one system to the other by only changing a single line in the code, where a connection to a database server was established.

Collaboration was important during design and development. We saw Christine using Hillary's documentation extensively. She not only used sample script code from the documentation but also relied on notes describing other experiences, optional steps, warnings, and expected execution times for each step. By using the documentation, Christine leveraged the community to improve the quality of her work. For example, Mike explained that fixes for some of the problems they had in the past were built into the sample scripts in the documentation. When Christine inherited the document, all these community best practices came along with it.

The documentation also contained instructions on how to produce the final scripts with all the necessary contextualization. This approach was preferred over writing the scripts sufficiently abstract to pull in the necessary parameters and context. They opted to hardcode such parameters into the scripts either manually or by running specific commands. We think that this was because of the transient nature of the scripts used, even though the outline of scripts was reused many times.

Hillary's documentation called for creating several scripts for each step of the procedure. The practice of splitting a procedure into multiple scripts was aimed at controlling and containing errors such that problems were monitored by people and corrected before proceeding. First, it was often very difficult to build error checking into scripts. It was particularly difficult in the case of database administration, as the sheer complexity of the tasks and the systems significantly increased the number and type of errors to be handled. Moreover error handling may differ based on the type of error. We saw some errors reported but ignored, some that required restart, some that were easy to fix, and some even expected. It was just difficult to predict and build appropriate error-handling logic into the script code.

Yet the issue here was not that simple error checking cannot be built into scripts easily. It was more than that. As we saw in Christine's instructions, every now and then there were verification steps, such as checking consistency of tables, performing application tests, etc. These checks could only be carried out by people who could judge whether things were proceeding correctly by interpreting output potentially from multiple perspectives within the context of the task. It seems to us that these people-steps were put there intentionally to try to increase reliability.

We did not observe any formal software maintenance activity, such as using a software version control tool. If they did any maintenance it was fairly informal as we saw Christine putting her name as a file suffix so that she could identify her scripts among many others. However, they did use a shared repository of scripts, and the organization of the script repository reflected substantial attention given to this activity, as scripts were carefully categorized by customer, task, year, etc. Maintenance of the documentation was on the other hand substantial in that documentation was used over and over by many people, reflecting experiences, conventions and best practices used in the organization. We believe that maintenance of the documentation was given substantially more attention as opposed to maintenance of the source code due to the expected lifetime of the scripts. Scripts were for the most part very customized, with lots of hard-coded configuration and were intended only for transient use.

In summary, we saw several practices related to maintenance and verification activities:

- Planning and rehearsing with progressive verification of scripts on multiple production-like systems.
- Reliance on verification by people rather than building error checking into code.
- Abstraction of large procedures into multiple verifiable steps.
- Maintenance of documentation rather than maintenance of source code
- Use of documents to communicate and share knowledge of practices, processes, and scripts.

3.2 Sometimes you think too much into it

Having written the scripts, Christine was now ready to begin work on consolidation systems using the instructions in the implementation plan. The first script she needed to execute was for creating the tables. She carefully typed the following command, as documented in the plan:

```
  nohup  db2  +c  -tvf  crttable.ddl.
christine –z  crttables.out.christine
```

In typing this, she diligently copied and pasted the script name by first getting a directory listing. In fact, she practiced copy-and-paste almost religiously, so much that she would not type a table name or script name without getting a list of some sort to copy from.

Just before submitting this command, she held back for a couple of minutes to examine it once again. Looking over Christine's shoulder at screen, Mike suggested that she run the script in the background, and she added an "&" to the end of the line to do that. After another minute or so, Christine hit Enter. And she immediately got an error. Mike was quick to identify the cause:

> ***Mike****: DB2 started, or…? I know you said you stopped SAP but did you start DB2?*

Christine was unaware that the script to bring down SAP (the specific database application she was working with) had also stopped DB2 (the database system underlying the application). So, she started DB2 and reran the script. This time there were no immediate errors. Mike suggested that she open another window to check progress continuously by "tailing" the output from the script, effectively spying on the output that was being written to a file.

Examining the script output, Christine noticed a problem with the script just in the first few lines. The first table creation had failed giving errors, but the second table creation was proceeding normally. Christine opened the script and Mike again was quick to see what was wrong:

> ***Mike****: Your connect [command] needs a semicolon at the end.*
> ***Christine****: But it is working for... that is the only connect I have right?*

In the script, the connect command did not end with a semicolon, which resulted in a syntax error. The interpreter reported the error, but it did not stop running the script; it continued to execute the remaining commands. So, the command to create the second table began executing normally, but because the system had not connected to the appropriate database server, the second table was being created in the wrong place. Christine quickly fixed the script by adding a semicolon. However, she could not stop the script from running because the system was making a massive update. In fact, it continued for more than half an hour because the table was so big.

Finally, when they were able to interact with the system again, Mike suggested dropping (that is, removing) this incorrectly created table. But the drop-table command also returned an error:

```
SQL0204N   "SAPR3.QCMVBAP" is an
undefined name. SQLSTATE=42704
```

Thinking that the error had to do with incorrect syntax, she tried several alternatives, including putting the table name in quotes, removing the schema name, and more. None fixed it. After a while, Mike realized what was going on:

> ***Mike***: *It backed out the whole thing, cause we never committed it. We don't have to drop anything. That is why the command didn't work because it couldn't find the stupid thing.*
> *[...] Sometimes you think too much into it.*

They were back to where they had been 45 minutes earlier. Christine reran the script, again carefully noting the start time. And this time, she added the –s option so that the system would abort script execution when it returns an error:

```
nohup db2 +c -stvf
  crttable.ddl.christine –z
  crttables.out. christine
```

Another 30 minutes passed and finally the first table in the script was created. Christine noted the execution time of the script on paper:

> ***Christine***: *I will put that in the notebook. So, if someone else does this in six months or a year, I will have some basis to go by.*

While waiting for the second table to be created, she decided to update the documentation. While working on the documentation she noticed that it took Mike about an hour and half last time around, roughly 26GB per hour, and she estimated how long it would take this time. In the documentation, she replaced all occurrences of command options "-tvf" with "-stvf" in the document. She also made a note that DB2 needs to be restarted after the SAP-stop script, which was omitted when she read the document preparing for this task. Throughout the procedure, we observed her edit the document with lessons learned while waiting for scripts to execute.

As Christine executed these scripts, Mike left the office briefly but he continued to observe her progress. In fact, he noticed that something was missing from the "create index" script and so he prepared another script to be run before the next step. He realized that the output from the first script did not contain the usual error message they got when they altered primary keys; that is, normally, there was a specific error message whenever they changed the way an index worked, but up to this point, he had not seen this message, so he concluded that they had missed a critical step. Back in her office, he said that she should run his new "primary key" script:

> ***Mike***: *You see I have never seen the message. When you create the index, you get the message about the primary key. I remember seeing the primary key when you create the index... Basically I created the primarykey.ddl script. That should be two minutes...*

Afterwards, Mike left to go home but promised to be back online as soon as possible. Sure enough, before the "create index" script was done, he was back online. When the primary key script completed, Christine sent Mike an instant message to make sure she was seeing the error message he expected:

> ***Christine****:* `is that the normal message you were talking about? sql0598w`
> ***Mike****:* `that error message is a good message.`

Having verified the error message, Christine recorded the times for this step and moved on to the remaining steps. She was almost ready for the big day, Saturday, the day of production server operations. In an instant message to Mike, she said that given her experience so far, she should have enough time to complete the change on the production systems on Saturday.

3.2.1 Analysis

As Christine and Mike began the configuration process in the consolidation server, they had several errors to deal with. This was okay – after all they purposefully institutionalized the practice of staged deployment by verifying scripts on a series of servers to detect errors as early as possible.

Pair-wise extreme collaboration worked pretty well, too, particularly when steps were missing in the documentation. Mike had experience with these set of instructions and knew expected behavior of the system. So, he was always quick to notice potential pitfalls or warn Christine in advance. When errors did occur he stepped in and quickly put Christine on the right track by recommending certain actions, such as restarting the database after the first step. In fact, throughout the deployment, we saw Christine and Mike sit side by side and perform tasks together or collaborate over instant messaging. There was considerable verification: Christine would type a command and wait for Mike to comment on it. Mike would check up on her from his office and from home, examining script output. Faced with risky and lengthy operations, this style of collaboration ensured reliability by adding a second pair of eyes. We think this practice provides the same sort of benefits as pair programming [16].

Some of the errors Christine faced were simple typos that accrued during the script development. Despite meticulous attention paid during development, errors were nevertheless inevitable. A simple missing semicolon led to unexpected behavior in which the script continued to execute, confusing Christine and Mike about the state of the system for some time as they tried to delete a non-existent table. They misinterpreted the errors, attributing them to possible syntactic issues rather than the real semantic issue: that the database server did not commit the changes. Clearly, some of the error messages did not convey the system state very well. Part of the problem was that their systems were really complicated, and so any individual message lacked the overall context of what they were trying to do.

Part of the problem was that cryptic server, database, and table names were confusing and led to errors throughout. To lessen the chance of introducing typos, Christine used copy-and-paste rigorously as she wrote and executed scripts and passed parameters to commands.

Not all errors they dealt with were bad either. Some were “good errors,” as Mike called them, in that lack of one indicated a missed step. He not only told her of the missing error message, but he also wrote a script fix the problem, and asked her to run it.

Beyond script reliability, rehearsal also had significant operational importance in establishing how much time would be needed for particular operations. Christine frequently checked the time to make sure that work would fit in the window allotted for the production changes. Documentation was an essential resource for her not only because it contained information about expected running times but also it contained traces of the deliberate work of past administrators, their experiences, suggestions, and warnings, effectively creating collective know-how within the organization. And Christine did her part by contributing back to the organization reporting her own experience, for instance, documenting –s option to force a stop on error, and recording her own execution times.

In summary, we observed several practices related to test, development and execution activities:

- Pair-wise extreme collaboration when testing and executing scripts.
- Iterative collaboration during script and procedure development by working through documentation.

3.3 I always give myself more time…

While the table move operation was going on Christine was asked to perform a database backup for a particular customer database during a teleconference as part of her on-call duties. She quickly wrote down the specifics on a piece of paper: an online backup for today and an offline backup with tape archive for Saturday. An online backup archived data from a production server while the applications continued to execute. An offline backup would take down the server, stopping all other applications. Thus, it was undesirable to perform an offline backup during regular hours, as that limited application availability.

As she was setting up an online backup, Christine needed to get in touch with Larry about the offline tape backup for the next day. Mike said that he would to remind Larry about the offline backup. To configure the online backup, Christine began editing the crontab file. Cron was a time-based job scheduling service to automatically execute recurring commands, specified in the crontab file on separate lines. Upon saving the crontab file the cron service would automatically schedule these jobs for execution at the specified times.

As they typically used crontab for performing periodic backups, among other things, the crontab file already contained correctly formatted entries for backup jobs. She just needed to find the right entry for an online backup and set the start time appropriately.

Because the crontab file was really long, she searched for the particular backup entry by entering `/ofl using the vi editor`. She uncommented the entry on that line and changed the schedule,

```
  00   16   0   11   *   /dba/lib/db.control
ofltape.cntl > /dba/logs/db.out 2>&1
```

which would execute the backup script at 16:00, just about a minute later. To be sure, she checked the date a few times using the `!date` command without existing the vi editor. Then she typed `!wq` to save the crontab file but waited for about three seconds before finally hitting "Enter", making sure things were correct. Once saved, only a few seconds later, Christine reacted rather nervously:

> ***Christine**: Oh, shoot!!*
> ***Mike**: What??*
> ***Christine**: I think I got the wrong one.*
> ***Mike**: No!!*
> ***Christine**: Oh, no.*
> ***Mike**: No . It was online.*
> ***Christine**: I think, I did*
> ***Mike**: That is okay…*

She immediately tried to revert the crontab file back to its original state. While doing that she was visibly in a kind of panic. She had trouble saving the file, had several typos, executed numerous invalid commands, one after another. Finally, having reverted the file, she checked the process list to see if the backup process was listed, meaning that it had already started and took down the applications. Seeing that there were no backup processes in the list, she was relieved:

> ***Christine**: I always give myself more time.*
> ***Mike**: Just page, do change directory, dba/backup*

Following Mike's advice, she changed the directory to see if the backup process made any new entries in the log file, indicating that the backup had actually started. Fortunately, the last modified date of the backup log file was old.

This time, Christine was lucky. The offline backup process had not started. She told us that she usually gave herself more than one minute before issuing such commands to allow time to catch such errors.

3.3.1 Analysis

Errors occur even with the best of intentions. In this case, Christine used crontab precisely to avoid errors – yet she created one when she picked the wrong line to execute. Nevertheless, Christine did her best to avoid errors in the first place. We saw her holding off hitting "Enter" to submit a command, reviewing the commands several times. Likewise, her use of crontab to submit commands with additional delay gave her one more chance to abort a potentially erroneous command, as she wound up doing in this case. Repurposing crontab beyond recurring tasks also worked as a cheat sheet so that she did not have to remember the syntax of the commands each time she needed to execute a similar task. Because most commands were already listed in the crontab file she only needed to change the date and time. The last working example was always in the crontab file. In this case we saw her check the date several times and wait a few more seconds to try to make sure her changes were correct. Though there was still an error, she did her job appropriately nonetheless, taking time to go over commands again and again.

When things did go wrong, she verified the state of the backup script in multiple ways. First, she corrected the crontab immediately. Then she verified whether the backup process was running by examining the process listing. Finally, upon Mike's suggestion she also checked to see if any log entries had been produced. This practice of verifying system state from multiple perspectives was common, particularly as tasks got more complex, risky, and long-running (also see 17]).

We also see pair-wise collaboration here providing a level of psychological comfort. When Christine became anxious after realizing she had started the wrong backup, Mike was there to try to calm her down.

In summary, we observed these script execution and verification practices:

- Using a job scheduling service list to (re)execute common scripts to avoid re-parameterization.
- Delayed execution of common scripts via a job scheduler for verification purposes.
- Multi-perspective verification of script execution and system state.

4. Discussion

Script development and use practices of system administrators were informal and collaborative, and combined effectively with formal processes of system administration (see also [4]). Development, debugging, use, and maintenance of scripts, tools, and processes were interleaved as system administrators worked with systems on multiple servers. The size and complexity of the systems themselves, combined with the risks associated with failing to make changes on time or making mistakes that brought live customer systems down, put substantial demands on system administrators, affecting error handling and verification, among other things.

In particular, we observed a number of practices that were developed to support scripting and related activities:

(1) Pair-wise extreme collaboration when testing and executing scripts.
(2) Iterative collaboration in script and procedure development
(3) Planning and rehearsing with progressive verification of scripts on multiple increasingly production-like systems.
(4) Reliance on verification by people rather than building error checking into code.
(5) Abstraction of large procedures into multiple verifiable steps.
(6) Use of documents to communicate and share knowledge on practices, processes, and scripts.
(7) Multi-perspective verification of script execution and system state.

Regarding the question of whether rigorous software engineering practice would help end-user programmers like system administrators, the answer is likely – if the context of the work of the end-user is appropriately taken into account. Among several techniques in software engineering practice, such as software modeling, formal specification, and formal verification, most emphasize early stages of software development. First off, one needs to assess whether upfront design time would pay off when in practice, the focus is on error recovery rather than on error prevention. In fact, even in error recovery, the goal is to bring the systems back in some form, even at the expense of pulling back the changes. Delays often increase costs, and reduce productivity in businesses where the margins are already low.

In IT service delivery, there is no clear release cycle for system configuration and maintenance work. A large part of the work is reactive, done in response to an emerging issue, such as external workload demands or internal system errors. Design time is clearly limited. In this context, rigorous software engineering techniques that require precise understanding of system model and behavior would put a significant burden on the system administrators, as systems are significantly complex and arguably no-one person has full understanding of the complete system behavior [4]. Furthermore, the required behavior is often hard to capture and communicate. And so it was not surprising to see high-level descriptions of system behavior in the form of documents being used heavily in practice. Though the descriptions are vague, they rely on experts to understand and provide the necessary implementation details.

Rigorous software engineering does not only mean precise specifications but also abstract specifications. And this may offer some opportunities. Abstractions focus on the essential aspects of system behavior without being bogged down with the irrelevant details. Given the complexity of systems techniques, precise descriptions is often out of the question. High-level understanding of system model is what is needed at first, followed by quickly narrowing the problem and finally a deep-dive into specifics, to troubleshoot systems effectively. We believe UML [7] descriptions of systems that break down systems into conceptual components, if made available by vendors, would benefit system administrators significantly, in script testing and development. Likewise, if standard UML-like descriptions are adopted by the system administrator community, it may potentially increase collaborative script development.

Software development methodologies such as Extreme Programming [16] and Agile Software Development [2] may also be effectively applied in the service delivery context, given their focus on short development cycles and iterative development with requirements evolving rapidly in collaboration within the organization, and their ability to respond quickly to changes and potential to improve productivity. These methodologies seem to fit well with system administration, with its collaborative, reactive working conditions and where productivity is primary concern.

5. Conclusion

For system administrators – like other end-users – a traditional software development methods are neither ideal nor possible. In system administration, system size and complexity and the risk of errors and downtime set the context and influence script development and related activities significantly. System administrators write small programs under severe technical and business constraints, and they need to produce reliable programs that interact with live systems. Yet we believe there are practices from software engineering that can be borrowed, particularly, agile software development that support collaborative and incremental software development, and software modeling approaches that describe multiple perspectives of the system.

6. References

[1] Barrett, R., Kandogan, E., Maglio, P. P., Haber, E. M., Takayama, L. A., and Prabaker, M. 2004. Field studies of computer system administrators: analysis of system management tools and practices. In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work

[2] Cockburn, A., Agile software development, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002

[3] Lentz, J. L. and Bleizeffer, T. M. 2007. IT ecosystems: evolved complexity and unintelligent design. In Proceedings of the 2007 Symposium on Computer Human interaction for the Management of information Technology (Cambridge, Massachusetts, March 30 - 31, 2007). CHIMIT '07. ACM, New York, NY, 6.

[4] Bailey, J., Kandogan, E., Haber, E., and Maglio, P. P. 2007. Activity-based management of IT service delivery. In Proceedings of the 2007 Symposium on Computer Human interaction For the Management of information Technology (Cambridge, Massachusetts, March 30 - 31, 2007). CHIMIT '07. ACM, New York, NY, 5

[5] Barrett, R., Chen, M., & Maglio, P. P. System Administrators are Users, Too: Designing Workspaces for Managing Internet-scale Systems, CHI 2003 Workshop.

[6] Brown, A. B. 2004. Oops! Coping with human error in IT systems. Queue 2, 8 (Nov. 2004), 34-41.

[7] Jacobson, I. Booch, G., Rumbaugh, J. (1998). The Unified Software Development Process. Addison Wesley Longman

[8] Kolstad, R., SAGE Salary Survey, 2004--2005, http://www.sage.org/salsurv.

[9] Nardi, B. A. 1993. A Small Matter of Programming: Perspectives on End User Computing. MIT Press, Cambridge, MA.

[10] Maglio, P. P. and Kandogan, E. 2004. Error messages: what's the problem?. Queue 2, 8 (Nov. 2004), 50-55.

[11] Ko, A. J. and Myers, B. A. 2005. Human factors affecting dependability in end-user programming. In Proceedings of the First Workshop on End-User Software Engineering (St. Louis, Missouri, May 21 - 21, 2005). WEUSE I. ACM, New York, NY, 1-4.

[12] First workshop on End-user software engineering, International Conference on Software Engineering, 2005.

[13] Myers, B. A. and Burnett, M. 2004. End users creating effective software. In CHI '04 Extended Abstracts on Human Factors in Computing Systems (Vienna, Austria, April 24 - 29, 2004). CHI '04. ACM, New York, NY, 1592-1593.

[14] Luff, P., Hindmarsh, J., Heath, C. (1999). Workplace Studies: Recovering Work Practice and Information System Design. Cambridge, MA: Cambridge University Press.

[15] Suchman, L. (1987). Plans and Situated Actions: The Problem of Human-Machine Communication. Cambridge: Cambridge University Press.

[16] Beck, K. 2000 Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Pub. Co., Inc.

[17] Velasquez, N. F. and Durcikova, A. 2008. Sysadmins and the need for verification information. In Proceedings of the 2nd ACM Symposium on Computer Human interaction For Management of information Technology (San Diego, California, November 14 - 15, 2008). CHIMIT '08. ACM, New York, NY, 1-8.