U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Software Engineering

T1 - History of Software Engineering

3MIEIC04_B

(7th October 2017)

Bárbara Silva	**up201505628**@fe.up.pt
Julieta Frade	**up201506530**@fe.up.pt
Ventura Pereira	**up201404690**@fe.up.pt

Índice

Caesars Entertainment in New Jersey.

Caesars Entertainment in New Jersey emailed **promotions to 250 people who self-identify as compulsive online gamblers** and are trying to avoid exactly this type of temptation by adding themselves to a do-not-send list.

Caesars Entertainment says a "***back-end software issue***" caused it to wrongly e-mail promotional gambling material to more than 250 "compulsive" online gamblers.

"*The issue that caused our system to inadvertently target their patrons has been fixed and we have had no incidents since,*" Palansky said in a statement Wednesday. "*We can assure the public that this lapse on our part was not an intentional targeting of their patrons, but simply a back-end software issue that failed to properly scrub our database before certain mailings*."

The promotional materials were emailed to more than 250 internet self-excluded gamblers between Feb. 16 and May 28 as well as 19 individuals on the self-exclusion list during that same time frame.

Even though online gambling is legal, caesars entertainment corp has been hit with **a $10,000 civil penalty**.

It's not the first time Caesars has been dinged in connection to compulsive gambling. In May, **it was penalized $3,000 because it did not include in a legible manner the compulsive gambling "1-800-GAMBLER" phone hotline** on billboards hawking online gambling.

References:

https://arstechnica.com/tech-policy/2014/11/software-issue-caused-casino-to-e-mail-promotions-to-compulsive-gamblers/?comments=1&post=27940201
https://www.law360.com/articles/593831/caesars-fined-in-nj-for-marketing-to-excluded-gamblers
http://www.slate.com/blogs/future_tense/2014/11/10/caesars_entertainment_in_new_jersey_accidentally_advertised_to_people_with.html

Software Glitch Accidentally Releases Prisoners

More than 3,200 US prisoners have been released early because of a software glitch. **The bug miscalculated the sentence reductions prisoners in Washington state had received for good behaviour**. It was introduced in 2002 as part of an update that followed a court ruling about applying good behaviour credits. State officials say the early releases **have been happening by accident for more than 13 years**.

"Approximately 3 percent of all released inmates since 2002 were released earlier than allowed by law," said Nick Brown, the governor's general counsel. He said the problem was first flagged when a crime victim's family was notified the perpetrator was about to get out — early. *"The family did its own calculation, determined that the offender was getting out earlier than the court had ordered, and contacted the department to ask why this was happening,"* Brown said.

Washington state officials are now in full damage-control mode. Until the software is fixed, they say no one will be released without a *"hand-calculatio*n" of the release date. State officials said that many early-release prisoners would have to return to jail to finish their sentences. Analysis of the errors showed that, on average, **prisoners whose sentences were wrongly calculated got out 49 days early. One prisoner had his sentence cut by 600 days.** In a conference cal, Dan Pacholke, the state's secretary of corrections, said the state is still digging into what crimes may have been committed by ex-cons in the period of time they should have still been in prison. Prematurely released prisoners are charged with causing two deaths, one a DUI vehicular homicide. Local police are now helping to round up those who still need to spend time in jail. So far, 31 of the early released inmates have been taken back into custody. Most of those who've been taken back into custody have not been accused of committing new crimes while they were on the outside.

References:

http://www.bbc.com/news/technology-35167191

http://wacoalitionforparole.org/early-release-of-wa-prisoners-due-to-doc-computer-glitch/

http://www.npr.org/2016/01/01/461700642/computer-glitch-leads-to-mistaken-early-release-of-prisoners-in-washington

AT&T Lines Go Dead (1990)

At 2:25pm on Monday, January 15th, network managers at AT&T's Network Operations Center in Bedminster, N.J. began noticing an alarming number of red warning signals from various parts of their world-wide network. **Within seconds, the giant 72 screen video array that graphically represented the network was crisscrossed with a tangle of red lines as a rapidly spreading malfunction leapfrogged from one computer-operated switching center to another.** The standard procedures the managers tried first **failed** to bring the network back up to speed and **for nine hours**, while engineers raced to stabilize the network, almost 50% of the calls placed through AT&T failed to go through. Until 11:30pm, when network loads were low enough to allow the system to stabilize, AT&T alone **lost more than $60 million in unconnected calls**. Still unknown is the amount of business lost by airline reservations systems, hotels, rental car agencies and other businesses that relied on the telephone network. This wasn't supposed to happen. AT&T had built a reputation and a huge advertising campaign base on its reliability and security.

It is known that **75 million phone calls were missed and 200 thousand airline reservations were lost**. Working backwards through the data, a team of 100 frantically searching telephone technicians identified the problem, which **began in New York City**. The New York switch had performed a routine self-test that indicated it was nearing its load limits.

As standard procedure, the switch performed a 4 second maintenance reset and sent a message over the signalling network that it would take no more calls until further notice. After reset, the New York switch began to distribute the signals that had backed up during the time it was off-line. Across the country, another switch received a message that a call from New York was on its way, and began to update its records to show the New York switch back on line. A second message from the New York switch then arrived, less than ten milliseconds after the first. **Because the first message had not yet been handled, the second message should have been saved until later. A software defect then caused the second message to be written over crucial communications information.** Software in the receiving switch detected the overwrite and immediately activated a backup link while it reset itself, **but another pair of closely timed messages triggered the same response in the backup processor, causing it to shut down also.** When the second switch recovered, it began to route its backlogged calls, and propagated the cycle of close-timed messages and shut-downs throughout the network. The problem repeated iteratively throughout the 114 switches in the network, blocking

In pseudocode, the program read as follows:

```
while (ring receive buffer not empty
       and side buffer not empty) DO

  Initialize pointer to first message in side buffer
  or ring receive buffer

  get copy of buffer

  switch (message)

     case (incoming_message):

           if (sending switch is out of service) DO

              if (ring write buffer is empty) DO

                  send "in service" to status map

              else

                  break

              END IF

          process incoming message, set up pointers to
          optional parameters

          break
    END SWITCH


  do optional parameter work
```

over 50 million calls in the nine hours it took to stabilize the system.

The cause of the problem had come months before. In early December, technicians had upgraded the software to speed processing of certain types of messages. Although the upgraded code had been rigorously tested, **a one-line bug was inadvertently added to the recovery software of each of the 114 switches in the network**. The defect was a **C program** that featured a break statement located within an if clause, that was nested within a switch clause.

References:

http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html
http://www.phworld.org/history/attcrash.htm
https://www.slideshare.net/ItrisAutomationSquare/risk-management-and-business-protection-with-coding-standardization-static-analyzer

Reading Notes: No Silver Bullet

The familiar software project has something of this character usually innocent and straightforward, but capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet, something to make software costs drop as rapidly as computer hardware costs do.

Essential Difficulties

Complexity: Software entities are more complex for their size than any other human construct because no two parts are alike. Digital computers are more complex due to their very large number of states. It is necessary an increase in the number of different elements when it comes to scaling-up a software. Its complexity increases much more than linearly and is in essential property, not an accidental one. The difficulty of communication among team members leads to product flaws, cost overruns and schedule delays. Not only technical problems but management problems come from the complexity, and it makes overview hard.

Conformity: The software must confirm because it has recently come to the scene, in others, it must conform because it is perceived as the most conformable. Anyhow, in all cases, much complexity comes from conformation to other interfaces; this cannot be simplified out by any redesign of the software alone.

Changeability: The software entity is constantly subject to pressures for change and usually superseded by later models. All the successful ones get changed as people try in new cases at the edge of, or beyond, the original domain.

The software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.

Invisibility: Software is invisible. Geometric abstractions are powerful tools, and geometric reality is captured by it. The reality of software is not inherently embedded in space.

Past Breakthroughs Solved Accidental Difficulties

High-level Languages: the most powerful stroke for software productivity, reliability and simplicity. It frees a program from much of its accidental complexity. On the other hand, at some point the elaboration of a high-level language becomes a burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs.

Time-sharing: It preserves immediacy, and hence enables us to maintain an overview of complexity. The most serious effect may well be the decay of grasp of all that is going on in a complex system. Meanwhile, the main effect is to shorten system response time.

Unified Programming Environments: *Unix* and *Interlisp* attack the accidental difficulties of using programs together, by providing integrated libraries, unified file formats, piles and filters.

Potential Silver Bullets

Ada and other high-level language advances: Ada is a general-purpose, high-level language of the 1980s. Its philosophy is more of an advance than the Ada language, the philosophy of modularization, of abstract data types, of hierarchical structuring but it is just another high-level language.

Object-oriented programming: An order-of-magnitude gain can be made only if the unnecessary underbrush of type specification remaining today in our programming language is itself responsible for nine-tenths of the work involved in designing a program product.

Artificial intelligence: It's hard to imagine how image recognition or speech recognition, for example, will make any appreciable difference in programming practice.

Expert systems: Suggesting interface rules, advising on testing strategies, remembering but-type frequencies, offering optimization hints, etc. It's difficult finding articulate, self-analytical experts who know why they do things; and developing efficient techniques for extracting what they know and distilling it into rule bases.

"Automatic" programming: It is the generation of a program for solving a problem from a statement of the problem specifications. The system assessed the parameters, chose from a library of methods of solution, and generated the programs. It is hard to imagine how this breakthrough in generalization could conceivably occur.

Graphical programming: It has proved to be essentially useless as a design-tool programmers draw flow charts after, not before, writing the programs they describe. No matter what we diagram, we feel only one dimension of the intricately interlocked software elephant.

Program verification: Program verification does not mean error-proof programs, mathematical proofs also can be faulty. So, while verification might reduce the program-testing load, it cannot eliminate it.

Workstations: The composition and editing of programs and documents is fully supported by today's speeds. Compiling could stand a boost, but a factor of 10 in machine speed would surely leave think-time the dominant activity in the programmer's day.

Promising attacks on the conceptual Essence

Buy versus build: The most radical possible solution for constructing software is not to construct it at all, attending the mass market. Even software tolls and environments can be bought off-the-shelf. Such product not only is cheaper to buy than to build afresh but also tend to be much better documented and somewhat better maintained than homegrown software. Another way of looking at it is that the use of n copies of a software system effectively multiplies the productivity of its developers by n.

Requirements refinement and rapid prototyping: The hardest single part of building a software system is deciding precisely what to build, therefore the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. This problem exists, specially because the clients do not know what they want to make it hard to know what questions must be answered. So, in planning any software activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition.

Incremental development: grow, not build, software.

Great designers: The central question of how to improve the software art centers, as it always, on people. Great designs come from great designers and software construction is a creative process. A little retrospection shows that although many fine, useful software systems have been designed by committees and built by multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers. Consider Unix, APL and Pascal; and contrast with Cobol, PL/I.