272: Software Engineering

Instructor: Tevfik Bultan

Lecture 0: Introduction
Software Engineering

• In 1968 a seminal NATO Conference was held in Germany.
  – Purpose: to look for a solution to software crisis
  – 50 top computer scientists, programmers and industry leaders got together to look for a solution to the difficulties in building large software systems (i.e., software crisis)
  – The term “software engineering” was first used in that conference to indicate a systematic, disciplined, quantifiable approach to the production and maintenance of software

• Three-decades later (1994) an article in Scientific American (Sept. 94, pp. 85-96) by W. Wayt Gibbs was titled:
  – “Software’s Chronic Crisis”
Software’s Chronic Crisis

Large software systems often:
- Do not provide the desired functionality
- Take too long to build
- Cost too much to build
- Require too much resources (time, space) to run
- Cannot evolve to meet changing needs

- For every 6 large software projects that become operational, 2 of them are canceled
- On the average software development projects overshoot their schedule by half
- 3 quarters of the large systems do not provide required functionality
Software Failures

• There is a long list of failed software projects and software failures

• You can find a list of famous software bugs at:
  http://www5.in.tum.de/~huckle/bugse.html

• I will talk about two famous and interesting software bugs
Ariane 5 Failure

- A software bug caused European Space Agency’s Ariane 5 rocket to crash 40 seconds into its first flight (*cost: half billion dollars*)

- The bug was caused because of a software component that was being reused from Ariane 4
- A software exception occurred during execution of a data conversion from 64-bit floating point to 16-bit signed integer value
  - The value was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed and an exception was raised by the program
- When the primary computer system failed due to this problem, the secondary system started running.
  - The secondary system was running the same software, so it failed too!
Ariane 5 Failure

- The programmers for Ariane 4 had decided that this particular velocity figure would never be large enough to raise this exception.
  - Ariane 5 was a faster rocket than Ariane 4!
- The calculation containing the bug actually served no purpose once the rocket was in the air.
  - Engineers chose long ago, in an earlier version of the Ariane rocket, to leave this function running for the first 40 seconds of flight to make it easy to restart the system in the event of a brief hold in the countdown.

- You can read the report of Ariane 5 failure at: http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html
Mars Pathfinder

- A few days into its mission, NASA’s Mars Pathfinder computer system started rebooting itself
  - Cause: Priority inversion during preemptive priority scheduling of threads

- Priority inversion occurs when
  - a thread that has higher priority is waiting for a resource held by thread with a lower priority

- Pathfinder contained a data bus shared among multiple threads and protected by a mutex lock

- Two threads that accessed the data bus were: a high-priority bus management thread and a low-priority meteorological data gathering thread

- Yet another thread with medium-priority was a long running communications thread (which did not access the data bus)
Mars Pathfinder

• The scenario that caused the reboot was:
  – The meteorological data gathering thread accesses the bus and obtains the mutex lock
  – While the meteorological data gathering thread is accessing the bus, an interrupt causes the high-priority bus management thread to be scheduled
  – Bus management thread tries to access the bus and blocks on the mutex lock
  – Scheduler starts running the meteorological thread again
  – Before the meteorological thread finishes its task yet another interrupt occurs and the medium-priority (and long running) communications thread gets scheduled
  – At this point high-priority bus management thread is waiting for the low-priority meteorological data gathering thread, and the low-priority meteorological data gathering thread is waiting for the medium-priority communications thread
  – Since communications thread had long-running tasks, after a while a watchdog timer would go off and notice that the high-priority bus management thread has not been executed for some time and conclude that something was wrong and reboot the system
Software’s Chronic Crisis

• These are not isolated incidents:
  – An IBM survey of 24 companies developing distributed systems:
    • 55% of the projects cost more than expected
    • 68% overran their schedules
    • 88% had to be substantially redesigned
Software’s Chronic Crisis

- Software product size is increasing exponentially
  - faster, smaller, cheaper hardware
- Software is everywhere: from TV sets to cell-phones to watches to cars
- Marc Andreessen: “Software is Eating the World”
- Software is in safety-critical systems
  - cars, airplanes, nuclear-power plants
- We are seeing more of
  - distributed systems
  - embedded systems
  - real-time systems
    - These kinds of systems are harder to build
- Software requirements change
  - software evolves rather than being built
Summary

• Software’s chronic crisis: Development of large software systems is a challenging task
  – Large software systems often: Do not provide the desired functionality; Take too long to build; Cost too much to build; Require too much resources (time, space) to run; Cannot evolve to meet changing needs

• Software engineering focuses on addressing challenges that arise in development of large software systems using a systematic, disciplined, quantifiable approach
No Silver Bullet

- In 1987, in an article titled: “No Silver Bullet: Essence and Accidents of Software Engineering” Frederick P. Brooks made the argument that there is no silver bullet that can kill the werewolf software projects.

- Following Brooks, let’s philosophize about software a little bit.
Essence vs. Accident

• Essence vs. accident in software development
  – We can get rid of accidental difficulties in developing software
  – Getting rid of these accidental difficulties will increase productivity

• For example using a high level programming language instead of assembly language programming
  – The difficulty we remove by replacing assembly language with a high-level programming language is not an essential difficulty of software development,
    • It is an accidental difficulty brought by inadequacy of assembly language for programming
Essence vs. Accident

- Essence vs. accident in software development
  - Brooks argues that software development is inherently difficult
    - “The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. … The hard part of building software is the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.”
  - Even if we remove all accidental difficulties which arise during the translation of this conceptual construct (design) to a representation (implementation), still at its essence software development is difficult
Inherent Difficulties in Software

• Software has the following properties in its essence:
  – Complexity
  – Conformity
  – Changeability
  – Invisibility

• Since these properties are not accidental representing software in different forms do not effect them

• The moral of the story:
  – Do not raise your hopes up for a silver bullet, there may never be a single innovation that can transform software development as electronics, transistors, integrated-circuits and VLSI transformed computer hardware
Complexity

- Software systems do not have regular structures, there are no identical parts

- Identical computations or data structures are not repeated in software

- In contrast, there is a lot of regularity in hardware
  - for example, a memory chip repeats the same basic structure millions of times
Complexity

- Software systems have a very high number of discrete states
  - Infinite if the memory is not bounded

- Elements of software interact in a non-linear fashion

- Complexity of the software increases much worse than linearly with its size
Complexity

Consider a plane that is going into a wind-tunnel for aerodynamics tests
- During that test it does not matter what is the fabric used for the seats of the plane, it does not even matter if the plane has seats at all!
- Only thing that matters is the outside shape of the plane
- This is a great abstraction provided by the physical laws and it helps mechanical engineers a great deal when they are designing planes

Such abstractions are available in any engineering discipline that deals with real world entities

Unfortunately, software engineers do not have the luxury of using such abstractions which follow from physical laws
- Software engineers have to develop the abstractions themselves (without any help from the physical laws)
Conformity

• Software has to conform to its environment
  – Software conforms to hardware interfaces not the other way around

• Most of the time software systems have to interface with an existing system

• Even for a new system, the perception is that, it is easier to make software interfaces conform to other parts of the system
Changeability

• Software is easy to change, unlike hardware

• Once an Intel processor goes to the production line, the cost of replacing it is enormous (the Pentium FDIV bug in 90s cost Intel half billion dollars)

• If a Microsoft or Apple product has a bug, the cost of replacing it is negligible.
  – Just ask users to update their software
Changeability is not an Advantage

• Although it sounds like, finally, software has an advantage over hardware, the effect of changeability is that there is more pressure on changing the software.

• Since software is easy to change software gets changed frequently and deviates from the initial design:
  – adding new features
  – supporting new hardware
Changeability

• Conformity and Changeability are two of the reasons why reusability is not very successful in software systems

• Conformity and Changeability make it difficult to develop component based software, components keep changing
Invisibility

• Software is invisible and un-visualizable
• Complete views can be incomprehensible
• Partial views can be misleading
• All views can be helpful

• Geometric abstractions are very useful in other engineering disciplines
  – Floor plan of a building helps both the architect and the client to understand and evaluate a building
• Software does not exist in physical space and, hence, does not have an inherent geometric representation
Invisibility

- Visualization tools for computer aided design are very helpful to computer engineers
  - Software tools that show the layout of the circuit (which has a two-dimensional geometric shape) makes it much easier to design a chip

- Visualization tools for software are not as successful
  - There is nothing physical to visualize, it is hard to see an abstract concept
  - There is no physical distance among software components that can be used in mapping software to a visual representation
Summary

• According to Brooks, there are essential difficulties in software development which prevents significant improvements in software engineering:
  – Complexity; Conformity; Changeability; Invisibility

• He argues that an order of magnitude improvement in software productivity cannot be achieved using a single technology due to these essential difficulties
Let’s look at an example:

- Sometime ago I asked our IT folks if they can do the following:
  - Every year all the PhD students in our department fill out a progress report that is evaluated by the graduate advisors. We want to make this online.
- After I told this to our IT manager, he said “OK, let’s have a meeting so that you can explain us the functionality you want.”
- We scheduled a meeting and at the meeting we went over
  - The questions that should be in the progress report
  - Type of answers for each question (is it a text field, a date, a number, etc?)
  - What type of users will access this system (students, faculty, staff)?
  - What will be the functionality available to each user?
Requirements Analysis and Specification

• This meeting where we discussed the functionality, input and output formats, types of users, etc. is called requirements analysis
  – During requirements analysis software developers try to figure out the functionality required by the client

• After the requirements analysis all these issues can be clarified as a set of **Requirements specifications**
  – Maybe the IT folks who attended the requirements analysis meeting are not the ones who will develop the software, so the software developers will need a specification of what they are supposed to build.

• Writing precise requirements specifications can be very challenging:
  – Formal (mathematical) specifications are precise, but hard to read and write
  – English is easy to read and write, but ambiguous
After figuring out the requirements specifications, we have to build the software.

In our example, I assume that the IT folks are going to talk about the structure of this application first.

- There will be a backend database, the users will first login using an authorization module, etc.

Deciding on how to modularize the software is part of the **Architectural Design**.

- It is helpful (most of the time necessary, since one may be working in a team) to document the design architecture (i.e., modules and their interfaces) before starting the implementation.

After figuring out the modules, the next step is to figure out how to build those modules.

**Detailed Design** involves writing a detailed description of the processing that will be done in each module before implementing it.

- Generally written in some structured pseudo-code.
Implementation and Testing

• Finally, the IT folks are going to pick an implementation language (PHP, python, Java, etc.) and start writing code.
• This is the **Implementation** phase:
  – Implement the modules defined by the architectural design and the detailed design.

• After the implementation is finished the IT folks will need to check if the software does what it is supposed to do.
• Use a set of inputs to **Test** the program
  – When are they done with testing?
  – Can they test parts of the program in isolation?
Maintenance

• After they finished the implementation, tested it, fixed all the bugs, are they done?

• No, I (client) may say, “I would like to add a new question to the PhD progress report” or “I found a bug when I was using it” or “You know, it would be nice if we can also do the MS progress reports online” etc.
  – The difficulty of changing the program may depend on how we designed and implemented it:
    • Are the module interfaces in the program well defined? Is changing one part of the code effect all the other parts?

• This is called the Maintenance phase where the software is continually modified to adopt to the changing needs of the customer and the environment.
Software Process

• Then there is the question of how to organize the activities we mentioned before (requirements analysis, design, implementation, testing).

• There have been significant research on how to organize these activities
  – Waterfall model, spiral model, agile software development, extreme programming, Scrum, etc.
Summary

• Software development involves multiple activities:
  – Requirements analysis and specification
  – Architectural design, detailed design
  – Implementation
  – Testing
  – Maintenance
  – Software development process

• There is active research in all of these areas in the software engineering community
<table>
<thead>
<tr>
<th>Research Area</th>
<th>Submissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software testing</td>
<td>140</td>
</tr>
<tr>
<td>Empirical software engineering</td>
<td>134</td>
</tr>
<tr>
<td>Software evolution and maintenance</td>
<td>117</td>
</tr>
<tr>
<td>Program analysis</td>
<td>115</td>
</tr>
<tr>
<td>Mining software engineering repositories</td>
<td>88</td>
</tr>
<tr>
<td>AI and software engineering</td>
<td>84</td>
</tr>
<tr>
<td>Security, privacy and trust</td>
<td>60</td>
</tr>
<tr>
<td>Tools and environments</td>
<td>58</td>
</tr>
<tr>
<td>Validation and verification</td>
<td>54</td>
</tr>
<tr>
<td>Debugging</td>
<td>45</td>
</tr>
<tr>
<td>Mobile applications</td>
<td>44</td>
</tr>
<tr>
<td>Human and social aspects of software engineering</td>
<td>39</td>
</tr>
<tr>
<td>Program comprehension</td>
<td>35</td>
</tr>
<tr>
<td>Dependability, safety, and reliability</td>
<td>35</td>
</tr>
<tr>
<td>Fault localization</td>
<td>33</td>
</tr>
<tr>
<td>Formal methods</td>
<td>29</td>
</tr>
<tr>
<td>Performance</td>
<td>27</td>
</tr>
<tr>
<td>Search-based software engineering</td>
<td>26</td>
</tr>
<tr>
<td>Software modeling and design</td>
<td>24</td>
</tr>
<tr>
<td>Software architecture</td>
<td>24</td>
</tr>
<tr>
<td>Programming languages</td>
<td>23</td>
</tr>
<tr>
<td>Apps and app store analysis</td>
<td>22</td>
</tr>
<tr>
<td>Agile software development</td>
<td>21</td>
</tr>
<tr>
<td>Middleware, frameworks, and APIs</td>
<td>21</td>
</tr>
<tr>
<td>Parallel, distributed, and concurrent systems</td>
<td>20</td>
</tr>
<tr>
<td>Model-driven engineering</td>
<td>19</td>
</tr>
<tr>
<td>Program repair</td>
<td>19</td>
</tr>
<tr>
<td>Distributed and collaborative software engineering</td>
<td>18</td>
</tr>
<tr>
<td>Software reuse</td>
<td>18</td>
</tr>
<tr>
<td>Specification and modeling languages</td>
<td>17</td>
</tr>
<tr>
<td>Refactoring</td>
<td>17</td>
</tr>
<tr>
<td>Recommendation systems</td>
<td>16</td>
</tr>
<tr>
<td>Requirements engineering</td>
<td>15</td>
</tr>
<tr>
<td>Autonomic and (self-)adaptive systems</td>
<td>14</td>
</tr>
<tr>
<td>Software process</td>
<td>14</td>
</tr>
<tr>
<td>Cloud computing</td>
<td>14</td>
</tr>
<tr>
<td>Software product lines</td>
<td>14</td>
</tr>
<tr>
<td>Program synthesis</td>
<td>14</td>
</tr>
<tr>
<td>Reverse engineering</td>
<td>13</td>
</tr>
<tr>
<td>Software services</td>
<td>12</td>
</tr>
<tr>
<td>Software economics and metrics</td>
<td>12</td>
</tr>
<tr>
<td>Crowd sourced software engineering</td>
<td>11</td>
</tr>
<tr>
<td>Configuration management and deployment</td>
<td>11</td>
</tr>
<tr>
<td>Component-based software engineering</td>
<td>9</td>
</tr>
<tr>
<td>Traceability</td>
<td>9</td>
</tr>
<tr>
<td>Software visualization</td>
<td>8</td>
</tr>
<tr>
<td>Human-computer interaction</td>
<td>8</td>
</tr>
<tr>
<td>Cyber physical systems</td>
<td>7</td>
</tr>
<tr>
<td>Green and sustainable technologies</td>
<td>5</td>
</tr>
<tr>
<td>End-user software engineering</td>
<td>5</td>
</tr>
<tr>
<td>Embedded software</td>
<td>4</td>
</tr>
<tr>
<td>Ubiquitous/pervasive software systems</td>
<td>0</td>
</tr>
</tbody>
</table>
This Course

• Software Engineering has been an active research area since its inception in 1968

• In this course we will have just a sampling of research in various areas of software engineering

• Currently, most significant research results are published in conferences. There are many conferences that publish research results related to software engineering:
  – See http://taoxie.cs.illinois.edu/seconferences.htm
Papers we will discuss

• There are many active research conferences that focus on software engineering research and its sub-areas

• The premier professional organization in software engineering domain is ACM SIGSOFT
  – SIGSOFT sponsors conferences
  – It also gives distinguished paper and impact paper awards

• For this course I selected a set of papers that received awards and also some other impactful papers

• At the end of the class I hope that you will have a good understanding of the software engineering research and some of the major research contributions in this area