What Do Systems Engineers Need To Know About Software?

Sarah Sheard
October 24, 2016

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Tutorial agenda

Overview
  • Purpose of tutorial
  • Attitudes
  • What are software and hardware? What are systems and software engineering? How are they similar and different?

Vocabulary

Some specific software knowledge for systems engineers

Example concerns: Safety and security

The future: A partnership

Conclusion
Purpose: Systems engineers understand…

How does software differ from hardware?
How does software engineering differ from systems engineering?
Basic software terms and concepts not always clear to systems engineers
Approach for the future: Partnership

Note: SW = Software, SWE = Software engineer(ing),
SysE = Systems engineer(ing)
Attitudes … correct?

SysEs think:
• Software engineers think only in bits and bytes
• Software engineers don’t recognize that anything is important other than software or logic that will become software
• Software engineers are subsystem engineers and need to listen to me, because I’m responsible for the larger system

SWEs think:
• Systems engineers act imperious and take credit for the system that software engineering creates
• Systems engineers don’t understand what we do, insist on functional software, don’t hand over fully complete requirements, …
• “What’s the secret sauce that SysEs have that we don’t?”
• Systems engineers are a pest…keep trying to make us plan our work, but we do Agile, we can’t plan; insist on all requirements up front…
Desired attitude re systems engineering

We want Software Engineers to see Systems Engineers
• As partners in building a useful complex system
  - Value of each is understood by both
• Primary system responsibility during requirements and validation
• Those to look to for breadth
• Responsible for overall system including “hardware”
• Having complementary expertise
What is software?

- Instructions that turn a general purpose computer into a special purpose computer
- Algorithms
- Computer programs
- The framework on which all capability is built
- The complexity remaining when replacing formerly dedicated (long-lead-time) hardware with generic computer hardware
- All the items needed to provide the user with the functionality they require
  - May include computers, training, backup plans, etc.
Facts about software

Every new software product is unique; otherwise an old one would be copied

Evolving system requirement needs drive software requirements, causing continuing change*

Software provides unprecedented capabilities, but is vulnerable to remote attacks
  • Code reuse (bespoke, but also libraries, drivers, vendor products) is desired to save cost, but it is a major source of vulnerabilities

Increasing complexity comes from dynamic program evolution
  • Complexity leads to fragility which yields often non-repeatable failures

Testing
  • Testing is a horribly inefficient way to find problems to be fixed in poorly written code
  • Software testing, including such activities as inspections, are sampling processes

Programming laws

Conservation of Organizational Stability
• Invariant Work Rate

Conservation of Familiarity
• Invariant Perceived Complexity
What does software do?

Makes components function, turns components into a system
Provides the glue among other system components (Communication, data)

Four things

• Turns data into information
  - Data from sensors, which view the physical world, e.g.
• Stores data and information
• Makes decisions
  - Enacts algorithms. E.g. decision algorithms.
• Drives actuators
  - to change something in the physical world
Varieties of software

IT vs Embedded systems

• IT: Software that drives only with computer hardware and user interface hardware
• Embedded systems: Software affects the physical world through sensors and actuators

How is it procured?

• Design and build it ourselves
• COTS or GOTS (commercial, or government, off-the-shelf)
• FOSS (Free and open source software)

Two phases

• Development: the fun stuff
• Maintenance: hard. Fixing someone else’s code
  - Corrective (bugs), Perfective (new requirements), Adaptive (for changes in databases or operating system), Preventive (reliability…)
What is hardware?

Only software engineers are satisfied with calling a generic group of things “hardware”
It means “not software”
Dictionaries: Hardware

Metal goods and utensils (locks, cutlery, tools, etc.)

(Computers) The sum of all physical objects, such as electrical, mechanical, and electronic devices which comprise a computer system…

Major items of equipment (military equipment; electronic and electrical devices of a vehicle or a computer)

(Military) Weapons, transport, and other physical objects used in conducting a war

Physical objects used in carrying out an activity, in contrast to knowledge, skill, or theory

Weapons, especially handguns, carried on the person, as in: Check your hardware at the door before entering.

What about: “Everything in an airplane other than the software?”

Note: The term “hardware” grates on non-software people
How is software the same as and different from hardware?¹

For both, design is key
• For software, production is just copying
Much can be automated, e.g. code generation from highly precise design; testing
Myth: Software is easy to change than hardware
• Reality: Yes, for toy problems
• Reality: For real-life problems, changed software is at least as hard as new software
• Reality: For complex systems, software change is probably harder than hardware change (less isolable)
Software becomes fragile when complex: can break it by adding to it.
Software fixes often cause more and worse problems; implications hard to see
Working from your own vs. others’ code is different
Aging
• Software, in theory, does not get old: No corrosion, stress fractures etc.
• But: Patches, re-working to work with changing environment. Also lack of support
How is software the same as, and different from, hardware?²

Interfces. Software interfaces are 100% under the control of the software designer and programmer: Mistakes can be made

- In contrast, physical interfaces (structural, electrical, fluid) are to a great deal determined by nature (and by previous designers’ decisions, i.e. of parts)
- Note: for reused/OTS software: to some extent these are also determined by previous designers’ decisions

Hard to keep track of how software is used

- Copied easily
- Aftermarket applications
- IT vendors have to forcefully stop supporting old versions

Much use of patterns and styles in software (at least in theory)
What is engineering?

Make things
Using technology
For human purposes

Engineering

Need
(purpose)
Technology

Engineered
Things

Does this apply to systems engineering? To software?
How are SWE and SysE the same?

Systems engineering
- Interpret high-level, ambiguous requirements
- Ensure subsystem and software designs fit
- Ensure correct external interfaces, interfaces among subsystems and software
- Orchestrate Disciplines
- Customer interface (non-SW)
  - TPMs, “ilities”
  - Acquisition lifecycle, V-model with feedback
  - Define system V&V approach
  - Program management
  - Don’t do: Details

Software engineering
- Identify Needs
- Identify Requirements (Functional and quality)
- Architect system/software
- Design system/software
- Identify stakeholders, viewpoints, views
- Identify interfaces; determine rules and performance
- System-Software tradeoffs
- Customer interface (SW requirements and design)
- Non-functional requirements
  - Agile for DoD; Earned value for Agile
  - Define and execute system and software V&V
  - Manage rapid change

- Develop and build to low-level, detailed, precise requirements
- Formal Methods, Coding
- SW Devel. Processes, Methods, Tools
- Make interfaces among software modules, data, and communications path work
- Reuse/New Tradeoffs
- Customer interface (coding)
- SW quality attributes
  - Agile sprints, releases
  - Execute software V&V
  - SW Project Management
  - Don’t do: “Hardware”

Breadth

Depth (SW)

From: SEI Blog
https://insights.sei.cmu.edu/sei_blog/2014/05/needed-improved-collaboration-between-software-and-systems-engineering.html
Similarities: software and systems engineering

Both call themselves “SE”!
  • Both pursue quality and system usability
  • Both have many specialties: one SWE is not replaceable by another SWE any more than a structural systems engineer is replaceable by a data modeler
  • Standards have been in the process of harmonizing for 15 years

Interplay of engineering, architecture, management

Evolving: Both more complex, Both turning into systems-of-systems

Both are done in projects
  • Hard to estimate
  • Very expensive to develop, but increasingly: to maintain
  • Delays in one hold up the other
How is software engineering different from systems engineering or specialty engineering?1

SW development lifecycle is likely different from physical life cycles
- Don’t need long-lead items (except when they do…)
- Acknowledges likelihood of customer desires changing (agility, close customer contact)
- Build-test-build
- “When software is treated as a minor subset of a system, it is likely to become the largest upset”

SW considers functional and non-functional requirements to be different

SW requires growth capacity
- Most systems have no hardware growth capacity
- Software in any particular instance *must* be changed during life (sustainment=enhancement)
- Any system must have extra processing and memory
How is software engineering different from systems engineering or specialty engineering?①

Verification

• Less variety of testing (i.e. no vibration, thermal); more tests. A hierarchy of component and subsystem tests; integration test, regression test, development test, operational test, inspections
• Vulnerability testing is growing
• Software testing is a sampling process
• Number of possible tests approaches infinity

Configuration management

• All software is intellectual
• SW CM is different from and separate from other CM
• Rigorous, used hourly to daily, and extremely important
  - Identification
  - Change control (i.e. version control)
  - Maintenance (patches, block iterations…)
From literature: “What software managers need to know”

“SW Project Manager is most important role for project success”

Software-specific processes, e.g. code reviews, languages

Strategy development and engagement

Evaluating SWEs

Creating and evaluating effective development teams

Controlling project costs and schedule

Multiple risk assessment and mitigation methods

Computing the confidence level in various cost estimates

Trade studies are imperative

Communication, project planning and scheduling, managing complexity, strategic planning (balanced scorecard, strategy maps, SWOT), Estimating, Cost accounting, Risk management

SWOT = strengths, weaknesses, opportunities, threats

Source: Peters 2015
Tutorial agenda

Overview

Vocabulary
  • General technical
  • Management related
  • Specific technical
  • Software “ilities”

Some specific software knowledge for systems engineers

Example concerns: Safety and security

The future: A partnership
Vocabulary \(_1\) (technical)

Required attributes – “ilities” → “quality attributes” (non-functional requirements)…things the system must do that can’t be coded in one specific place

Performance – speed of a computer or processor, or effectiveness such as throughput, response time, or availability (narrower than SysE definition: Any key parameter including such things as battery life or antenna gain)

Exception handling – process of responding to exceptions (during computation)…which are anomalous or undesired conditions that may result in faults if not properly handled (x/0, out of memory…)

Refactoring – complete restructuring of a program, usually because as it has been maintained its structure got awkward, without changing its behavior. Changes nonfunctional attributes

Regression testing – testing that software previously shown to be working still is working after changes (including new interfaces with other software)
Vocabulary_2 (technical)

Object orientation
• Instead of specifying software in terms of actions and logic, software is organized around real-world objects, with attributes and interactions.

Latency – two meanings:
• Latent defects haven’t been discovered, but are suspected.
• Latency in timing = delay.

Entropy (ever-increasing disorder in a system) – two general meanings:
• Software entropy: A computer program that is used will be modified; A modified program will have higher complexity unless specifically addressed; technical debt increases software entropy.
• Information entropy: Expected value of information content of a message.

Prototyping
• In systems engineering: first version.
• In software engineering: partial version.

Atomic Requirements – “Leaf” nodes, most detailed requirements.
Vocabulary$_3$ (management-related)

Stories/User stories

- Aspects of Agile software development comparable to a scenario or to a number of related requirements – written in sentences

Production – phase where software is rolled out to users

Velocity – In Agile software development, the number of work units (e.g. user stories) completed in a given interval. Assists in estimating the next phase

DevOps – a phase where operational software is being continuously improved (developed): used in Agile software development

Technical debt – a financial metaphor for the work that is not done but is known to be needed when implementing a short term solution to a problem. The additional work that must be done long-term
**Vocabulary**

**4 (specific technical)**

**Backplane:** a hardware board that provides connections among main computer circuit boards

**SQL and NOSQL**

- **Structured Query Language:** Programming language developed for managing data in a relational database
- **No (or: Not Only) SQL:** a database management system consisting of objects other than tables in relational databases (can deal more rapidly with a variety of data structures)

**Jira, GitHub etc…** Tools used in SW development (specifically: proprietary issue tracking product, software repository hosting service)
Vocabulary \(5\) software “ilities”

SW architecture is largely driven by **non-functional requirements** (ilities) = “quality attributes”

**Design qualities*** (“conceptual integrity,” maintainability, reusability*)

**Run-time qualities*** (availability, interoperability, manageability,* performance, reliability, scalability,* security)

**System qualities*** (supportability, testability)

**User qualities*** (usability); also modifiability

**Reliability** issues based on bugs, or unintended interactions, not mechanical failures

Redundancy doesn’t solve **vulnerability** problems. If a flaw disables one helicopter, it’ll disable the second one too

*Kaniss, Al. What is a software engineer, CrossTalk May/June 2015*
Tutorial agenda

Overview

Vocabulary

Some specific software knowledge for systems engineers

• Basic software engineering principles
• Evolution
• Paradigms and styles
• Defects
• Lean/Agile/SAFe
• Formal methods

Example concerns: Safety and security

The future: A partnership
Basic software engineering principles*

- Intellectual Control
- Divide and Conquer
- Identify the customers
- Fuzzy into Focus
- Document it
- Essence of software is input-output
- Too much engineering is not good
- Plan for change
- Reuse as possible (not just code!)
- Keep simple: Cyclomatic complexity** <30

Also services, e.g. Cloud, software-as-a-service

Evolution

Requirements for software are always changing

- System requirements changes often end up in software
- Easiest way to implement a new capability
- Threats, operating systems, interfacing software change
- Mental models change (User interface)

Evolving programs cause increasing complexity; causes fragility = unrepeated failures
Paradigms*

Classify languages and programs
E.g., structured, object-oriented, aspect-oriented, agent-oriented, functional, process-oriented
Imperative, functional, logic, object-oriented**
Event-driven, service-oriented, time-driven
Sequential vs parallel computing
Functional vs procedural languages
Applications, such as robotics, machine-learning
Modular vs monolithic, structured vs non-structured (array)

*Good source: http://cs.lmu.edu/~ray/notes/paradigms/
## Architecture Styles

<table>
<thead>
<tr>
<th>Architecture style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client/Server</strong></td>
<td>Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.</td>
</tr>
<tr>
<td><strong>Component-Based Architecture</strong></td>
<td>Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.</td>
</tr>
<tr>
<td><strong>Domain Driven Design</strong></td>
<td>An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain.</td>
</tr>
<tr>
<td><strong>Layered Architecture</strong></td>
<td>Partitions the concerns of the application into stacked groups (layers).</td>
</tr>
<tr>
<td><strong>Message Bus</strong></td>
<td>An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.</td>
</tr>
<tr>
<td><strong>N-Tier / 3-Tier</strong></td>
<td>Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.</td>
</tr>
<tr>
<td><strong>Object-Oriented</strong></td>
<td>A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.</td>
</tr>
<tr>
<td><strong>Service-Oriented Architecture (SOA)</strong></td>
<td>Refers to applications that expose and consume functionality as a service using contracts and messages.</td>
</tr>
</tbody>
</table>

Layering (architectural pattern or style)

The organization of programming into separate functional components that interact in some sequential and hierarchical way, with each layer usually having an interface only to the layer above it and layer below it.

OSI (Open System Interconnection) layered protocols (protocol stack)

TCP/IP Two-layer set of programs: transport and network address functions

Presentation (UI) layer
application layer
business (domain) layer
data access layer

With COTS/FOSS, developing SW = assembling products; maintenance of COTS/FOSS can break system.

Source: whatis.techtarget.com/definition/layering
Graphic source: Wikipedia: Abstraction layer

TCP/IP Tranmission control protocol/Internet Protocol
COTS/FOSS: Commercial off the shelf, Free & Open Source Software
Defects: Why and what

Major indicator of software quality, software process quality, problem reports, software test, readiness...

Qualities: ID number, description, where and how found (date, module, line..., and steps to take to reproduce), version where found, references, person who identified, severity, priority, difficulty of closing, status, who fixed, when closed

Reason: Much less costly to find in phase created

Preventing, counting, reducing, managing

Software engineers think in terms of defects: some have a hard time imagining systems engineers NOT managing via defects

Entire Agile SW development can be organized around working off defects (Backlog)
Defects₂: Relationship to rest of software

How many defects did we find during what phase (usually: what test)? (discovery)

What kind were they? (categorization) Origin, severity, priority, etc.

New programming practices are adopted that reduce creation of defects that escape the “coding” phase into test (e.g. via “strict typing”)

Assigning, scheduling, and fixing defects (resolution) = rework

Testing (verification) esp. regression testing

Closure and reporting
Defects$_3$: Metrics

Number of defects

- Per module, per phase…

Defect rejection ratio: Things originally categorized as defects, but actually were not

Defect leakage (or escape) ratio: fraction of known defects not detected in a phase (often: found later; sometimes using a curve fit to the defect-finding data). “Escape” usually means leaked to next phase without being found
Defects: Causes

Miscommunication of requirements
  • And last-minute changes

Compressed development schedule

Inexperience: Designer, coder, tester

Human error

Version control failure (or lack)

Buggy tools

Defects5: Types (for example)

Defects: Causal analysis and prevention

Causal analysis: What caused this?

Defect prevention
- Understand causes
- Cost (implications) of having defects
- Cost of implementing prevention (training, tools, procedures, processes)
- Impact on quality

Periodic reviews
- Team, maybe larger groups

Categories:
- Planning
- Requirements, Features ~50%
- Functionality as implemented
- Structural bugs
- Data
- Implementation
- Integration
- Real Time and Operating System
- Test definition/ execution bugs
Lean (2003+)
From lean manufacturing
Mary, Tom Poppendieck
Eliminate waste and improve processes (e.g., Kanban)

Principles
• Eliminate waste
• Amplify learning
• Defer commitment
• Deliver fast
• Empower the team
• Build integrity in
• See the whole

Agile (2001+)
Agile Manifesto (Software community)
Work and deliver incrementally (e.g., Scrum, Extreme Programming, TDD…)

Values
- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Similarities and Differences:

- Lean was conceptualized much before Agile
- Lean has a wider scope and can be applied to any industry/domain. Agile was conceptualized specifically for software development
- Agile uses quite a few techniques/toolkits prescribed by Lean


TDD = test-driven development
Redrew charts in Microsoft article intended to verify practice adoption rates.

**Article conclusion:** No clear trends in practice adoption

**My conclusion:** Clear trend that only some practices are performed more often in Agile projects

Scaled Agile Framework SAFe

“Knowledge base for Lean Software and Systems Engineering”
• Proven, codified, and publicly-facing
• Purpose: Scale Agile and lean in larger software enterprises

Principles
• Take an economic view
• Apply systems thinking
• Assume variability; preserve options
• Build incrementally with fast, integrated learning cycles
• Base milestones on objective evaluation of working systems
• Visualize and limit work-in-progress, reduce batch sizes and manage queue lengths
• Apply cadence, synchronize with cross-domain planning
• Unlock the intrinsic motivation of knowledge workers
• Decentralize decision-making

Source: https://www.ivarjacobson.com/scaled-agile-framework
Formal Methods

System design techniques

Rigorously specified mathematical models of software and hardware systems

• Clarify thinking
• Allow early determination of things that won’t work
• Assertions can be proven through *mathematical* proofs of the system design (“regular” test techniques are sampling processes)

Applications

• Formal specification (modeling languages)
• Verification (proving of “theorems” represented by specification; some testing is automated; also model checking)
• Implementation (some can be auto-converted into code)

Concerns: Difficulty (expense), misapplication

Source: https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/
Formal Methods

Varied examples of formal methods

• Abstract state machines (pseudo code that generates Finite State Machines); model algorithms

• AADL (architecture analysis & design language)

• Knowledge-based software assistant (KBSA; Air Force; 80s)

• Petri nets (modeling language describing state transitions of distributed systems; graphical notation for processes with exact mathematical representation)

• UML (unified modeling language) and SysML (systems modeling language)

• Related items

• MALPAS software static analysis toolset: model checker for formal proof of safety critical systems

Sourc: https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/
Tutorial agenda

Overview

Vocabulary

Some specific software knowledge for systems engineers

Example concerns
  • Safety (automotive, flight incident)
  • Security

The future: A partnership
Future cars will likely be able to communicate with each other, sharing information about traffic hotspots and accidents, and use crash avoidance systems to prevent accidents.
Toyota unintended acceleration

Toyota paid $1.6B settlement plus $1.2B for concealment re software problems causing cars to speed up uncontrollably

“efforts to conceal the problem and protect its corporate image led to a series of [89] fatalities”

• People? (driver error)
• Hardware? (floor mats? sticky gas pedal?)
• Software? (no error detection/correction code; cascading memory corruptions)

Note: Federal motor vehicle safety standards not written for software

Cyclomatic complexity of “throttle angle function” was 146

Code has 11,000+ global variables (desired: 0) should be < 30

Sources: http://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf and https://www.washingtonpost.com/business/economy/toyota-reaches-12-billion-settlement-to-end-criminal-probe/2014/03/19/5738a3c4-af69-11e3-9627-c65021d6d572_story.html
In-flight upset event, Boeing 777-200, Aug 1, 2005

Simulated cockpit video:  https://www.youtube.com/watch?v=1XNnEzFF5fg
Security Considerations

These are so critical to system performance, robustness and resilience that we make these particular points separately!

Security

• Software assurance means “free from defects” and “free from vulnerabilities”
• Security processes involve both systems and software
• Software must be free from vulnerabilities that can be exploited by threats that won’t be invented for 5-10 years
• Just about any software can be accessed from just about anywhere on the planet
• Thus, SysE and SwE must evaluate for cybersecurity, from compliances with secure coding practices to actively trying to attack a system as developed, before it is released

Easy to identify and fix with a small patch. Much harder to prove the patched software, under all implementations, remains robust and not vulnerable
Tutorial agenda

Overview

Vocabulary

Some specific software knowledge for systems engineers

Example concerns:

The future: A partnership

• Early involvement
• Trades
• Meetings and estimating
• Communication
• Other
• Help software engineers
Early involvement

Bring in software, especially software architects, before finalizing system architecture
• Explain goal of this phase is not coding, but structuring the software design, giving feedback as to the software issues with the evolving system design, and understanding legacy code usage with its interfacing requirements to new software and security

Up-front software engineering participation means fewer ungrounded assumptions and less re-direction (→rework, complexity)

During planning, focus on identifying the decisions that need to be made for the project to move ahead
• Then determine SysE and SWE roles in determining information for those decisions
Trades

Identify requirements from various stakeholders as imperatives or tradeables

- De-conflict and implement imperatives; track and trade off tradeables

SysEs do larger-system trade studies, e.g., size weight and power (SWAP); software trades to be planned and done; security is shared

- Software architects must estimate impacts to software of various system decisions
- Share threat models and evaluate attack surfaces jointly

Both SysEs and SWEs are responsible for non-functional (=ilities=quality attributes) and functional requirements. Work together to trade off tradeables

Include software-related requirements (or potential requirements if too early for requirements) and software-related design attributes in system-level trade studies SWE functions more like SysEs of the software (software architects) than programmers
Meetings and estimating

Meetings

• Schedule meetings so that SWEs have blocks of time for concentration (e.g. meetings in 4 half days a week, so they can concentrate in 4-hour blocks and one 8-hour day)
  - Systems engineers almost always multitask, and tend to multitask well. Don’t expect SWEs to do it or like it
  - Encourage them to set office hours, then respect them

Estimating

• Remind SWEs that bugs happen and to plan for those. Also, plan for learning, to help re: tendency to underestimate
• Ask, “what might happen that would turn this into a cost and schedule nightmare?” Then estimate and mitigate together
Communication

Keep software informed as to what aspects are known, suspected, or totally guessed, and what is known to be likely to change

• Good SWE encapsulates likely changes so that they won’t disrupt the rest of the software being written, or the system

Help them see the context of where the software will run. What is known about it?

Do as much systems engineering as possible using models, e.g. SysML. Keep in mind, there are differences compared to SW and software models. Also, “Model based systems engineering” is not all of systems engineering! for example, coordination

Computer science teaches people how to do small programs. Help SWEs learn what goes into large programs
Other

Use case development
• SysEs must manage use case development (for SW requirements)
• Include abuse and misuse cases (what system must not do)

Testing
• Coordinate system and software tests; durations, purpose, what they determine

“Illities” roles, especially Security roles
• Together work reliability, availability, maintainability, attributes
• Partner regarding security: what SW vulnerabilities could be leveraged by adversaries to compromise systems (both IT and embedded systems). What priorities on vulnerabilities do system decisions imply?

Appreciation
• Express appreciation no coders/programmers/what SWEs call “Engineers”=builders. Thank them when they turn in code (before the bug reports come in)
• SWEs should appreciate work of SysEs too
Perspective

SWEs recognize they need to know about the domain,

- Generally SysEs have more opportunity to learn a domain, staying with the same program for years and interacting with customers/operators more
- Help SWEs understand, recognizing they are busy learning new languages and techniques, preventing, finding, and resolving bugs, and trying to think creatively through interruptions

Systems engineers are multi-disciplinary; Software engineers may consider themselves multidisciplinary as well: compare domains

SysE and SWE approaches should complement each other for system and program success
Help Software Engineers

Clarify roles

Liaison to other disciplines

Language translation: SW to other and back

Understanding Agile, compare to Lean and other project management

• Example Scaled Agile Framework or SAFe
Tutorial agenda

Overview

Vocabulary

Some specific software knowledge for systems engineers

Example concerns:

The future: A partnership
Conclusion

Systems engineering is responsible for delivering systems that do what they are intended to which are delivered on schedule and within budget

• Ensures the pieces come together into a system

Software engineering allows the delivered systems to do what they are intended to, including interfacing with other systems being Systems engineered

• Ensures the pieces do assigned roles, alone and together

Systems engineering and software engineering are inextricably linked, a relationship that must:

• Begin at the beginning, and
• Continue throughout the life cycle of the program
P. S. What not to do

Say “I used to code”

Say “There is no software at the system level”

Interrupt programmers, or any kind of software engineer, frequently

Think you can “power through” some function better or faster than a computer (such as looking through a data dump for a bad data element)

Think “software reliability is 1”

Think you can get software faster if you throw more people on the project

Say (or think) “software is much easier to change than hardware”
References


What Do Systems Engineers Need To Know About Software?

Backup slides
Programming Languages issues/questions

Choosing a language
Typing (OO, Functional, Declarative, Procedural)
Encapsulation for data abstractions
Run-time checking
Program redundancy checking
Assertions
Problems
Macros
Libraries
Error-prone constructions
Rapid Prototyping

- Procedural = C, Cobol, Fortran
- OO = Smalltalk, Java, C++
- Functional = Erlang, Clojure, F#
- Declarative = SQL, XSLT
- Multi-paradigm = Python, Java, C++, C#
Software Development Knowledge

Formal Methods
• Specification Languages
• Logic
• Sorting
• Controversy: Formal Methods

Design Notations
Support tools
Object Orientation
Java
DFDs
Testing
• Process
• Coverage testing
• Coverage criteria
• Structural test adequacy

Programming Languages
• Typing
• Encapsulation for data abstractions
• Run-time checking
• Program Redundancy Checking
• Assertions
• Problems
• Macros
• Libraries
• Error-prone constructions
• Choosing a language
• Rapid prototyping

From “Top 12 Things Every SW Engineer Should Know”

1. Fundamentals of emotional intelligence
2. Algorithms and big-O-notation: libraries, individual solutions; analyze others’ code
3. Basics of project management: don’t just estimate code implementation. Recall documentation, security concept, data protection issues, “alignment with worker’s councils, reviews, project management efforts, deployment etc.
4. Mainstream development paradigms
5. Basics about software security: becoming more important
6. Know your development tools and Know the key concepts and underlying technologies of “disciplines”: requirements management, SW & database (DB) design, SW CM, build & deploy, continuous integration, development, debugging, profiling, code analysis, testing. Also infrastructure toolboxes: network monitoring, analysis, pen testing, log file analysis, DB performance tuning

Source: www.sw-engineering-candies.com/blog
From “Top 12 Things Every SW Engineer Should Know”

7. Don’t trust code without adequate test. Even the best programmer needs unit, integration, sys tests; performance and memory tests with real data, static code analysis, code coverage measurement, load & stress tests, peer review

8. Key metrics of SW development…at least, lessons from Agile projects

9. Root cause of the last defect…understand what caused it and how to avoid it

10. Understand the business of your customer. This is one of the systems engineering roles. “If you don’t understand the what, you can’t decide about the how” - Sprunck

11. Understand the infrastructure ITIL language/terminology. Sometimes a developer needs to talk to 5 different infrastructure folks for one question. “The ITIL stuff is the glue among people in infrastructure”

Periodic Table of SW Engineering

Things to know (one chart each follows)

- Requirements
- Design
- Lean IT (Agile)
- Maintenance
- Infrastructure
- Basics
- Implementation
- Code Analysis
- Testing
- Usability
- Tools
- Management
- Soft skills

Requirements

Requirements elicitation
Requirements analysis
Atomic requirements
Requirements attributes
Requirements reviews?
Traceability management
Management of requirements portfolio

Design

Component design
Database design
Design patterns
Architecture pattern
Large-scale system design
Design notation

Lean IT

Scrum, Kanban

Agile methods:
• Agile planning
• Pair Programming
• Test-driven development
• Definition of Done
• Continuous integration
• Continuous Delivery
• User stories
• Backlog management

Agile methods, continued
• Stand-up meeting
• Spike solutions
• Planning game
• No Overtime
• Collect code ownership
• Travel Light
• System metaphor

Note: when backed into a corner, Agilists will insist that the method or practice is *not the point*, it’s the view, or how it’s done

Maintenance and reuse

Re-engineering
Reverse engineering
Program comprehension
Maintenance planning
IT change management

Reuse
• Designing for reuse
• Designing with reuse

Infrastructure

Basics of ITIL
DevOps
Monitoring
Build Management
Automated deployment
Test data management

Basics

Big-O notation
Algorithm design
OO languages
Software security basics
Scientific computing
Parallel computing
Numerical methods
Data structures
Functional languages
Encryption basics
Database systems
Game theory

Robotic basics
Aspect-oriented programming
Declarative languages
Network protocols
Distributed computing
State machines
Distributed computing
Procedural languages
Web app security
Machine learning
Artificial intelligence
SW development process

# Implementation & code analysis

<table>
<thead>
<tr>
<th>Basic coding skills</th>
<th>Static code analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code refactoring</td>
<td>Dynamic code analysis</td>
</tr>
<tr>
<td>Code peer reviews</td>
<td>Volume metrics</td>
</tr>
<tr>
<td>Code comments</td>
<td>Complexity metrics</td>
</tr>
<tr>
<td>Code format standards</td>
<td>Code coverage</td>
</tr>
<tr>
<td>Code reuse</td>
<td>Dependency analysis</td>
</tr>
</tbody>
</table>

Testing

Unit testing
Defect root cause analysis
Integration testing
Service testing
Performance testing
Stress testing
Exploratory testing
Etc.

Usability

User interface design
User acceptance
Usability labs

Tools

Code analysis tools
Continuous integration tools
Requirements management tools
Integrated development environment tools
Test automation tools
Profiling tools
Modeling tools
Version control systems

Management

Risk analysis
Expectation management
Task management
Project management basics
Estimations
Measurement of activities
Project Controlling

### Soft skills

<table>
<thead>
<tr>
<th>Presentation skills</th>
<th>Creativity techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training skills</td>
<td>Marketing basics</td>
</tr>
<tr>
<td>Empathy</td>
<td>Leadership basics</td>
</tr>
<tr>
<td>Creation of relations</td>
<td>Good manners</td>
</tr>
<tr>
<td>Conflict management</td>
<td>Intrinsic motivation</td>
</tr>
<tr>
<td>Negotiation skills</td>
<td>Physical fitness</td>
</tr>
<tr>
<td>Rhetoric</td>
<td>Stop talking</td>
</tr>
<tr>
<td>Intercultural skills</td>
<td></td>
</tr>
</tbody>
</table>

Additional information

Interfaces
Conventions and templates
Layering
Algorithmic complexity
Hashing
Caching
Concurrency
Cloud computing
Security (again)
Relational databases

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere
Interfaces

Most important concept in software
Any good software is a model of a real or imagined system
Modeling requires identification of parts and interfaces among them.
Good modeling has correct and simple interfaces
Book on interfaces: Agile programming by Dr. Robert Martin

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere
Conventions and templates

Naming conventions: indicating function or relationship of a piece of software in its name

Templates help build components

• Template files contain variables
• Allow binding of objects, resolution, and rendering the result for the client

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere
Layering

Layering*

• Subroutines that only call libraries or system are first layer
• Second layer calls first level or libraries or system
• Etc
• Main() sits at top level

Complexity of a component: How many other components does it rely on?

A good software system is pyramid shaped

• Fewer more-complex components, more less-complex

Precursor to refactoring (continuously sculpting software to ensure it is structurally sound and flexible)

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere

* John Lakos: Large scale C++ software design (1997)
Algorithmic complexity

Big O notation: Order of size $O(n)$ means linear with the size of data ($n$) $O(n^2)$ is quadratic with size of data.

Search through a list is $O(n)$, binary search (through a sorted list is $\log(n)$). Sorting of $n$ items takes $n\log(n)$ time

Code should not have multiple nested loops. Instead use hash tables, single lists, and singly nested loops

Elegant algorithms improve performance, but we downplay the importance nowadays due to excellent libraries

Having clean and simple algorithms requires compact and readable code

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere
Hashing

To have fast access to data
Reduces time to find an item as tables get larger by spreading data evenly throughout a table
Uniform hash evenly allocates computers in a cloud database
Google indexes each URL to a particular computer

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere
Caching

In-memory store of a subset of information typically stored in a database

Example: books that were popular last week: generate once and place into cache. Costly calculations done once

Cost: need to prune, usually by least-recently-used

Modern web applications often used a distributed caching system called Memcached

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, as augmented by personal knowledge and knowledge from elsewhere
Concurrency

Parallelism, inside an application

In Java, threads run concurrently

• Producer/consumer: producer generates data or tasks; queue up for worker threads to consume and execute
• But: threads work on common data
• Core Java contains concurrency libraries

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, augmented by personal knowledge and knowledge from elsewhere
Cloud computing

Grew out of parallel computing: running computations in parallel
Then grid computing (runs parallel computations on idle desktops)
Then application server virtualization
Today: host calculations and databases remotely, including indexing
Enables large-scale computing

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, augmented by personal knowledge and knowledge from elsewhere
Security (again)

Information authentication, authorization, information transmission (among others)

Authorization:
- Verify user identity (password) and keep it secure e.g. by transmitting encrypted (SSL=secure socket layer)
- Define permissions

Network protection
- Identify vulnerabilities
- Configure and monitor to thwart hackers
- Continuous patches as new threats/vulnerabilities are defined

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should Know, as augmented by personal knowledge and knowledge from elsewhere
Relational databases

Fundamental achievement in computing

Good for:
- Order management systems
- Corporate databases
- Profit and loss data

Represents each piece of information, added to a table that defines the type of information

Search using query language such as SQL

Data normalization technique partitions data among tables to reduce data redundancy and maximize retrieval speed

Does not scale well to massive web services

Source: http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know, augmented by personal knowledge and knowledge from elsewhere
Software heuristics from Sprunck

- 'Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.' [Boehm01]
- 'Current software projects spend about 40 to 50 percent of their effort on avoidable rework.' [Boehm01]
- 'About 80 percent of avoidable rework comes from 20 percent of the defects.' [Boehm01]
- 'About 40 to 50 percent of user programs contain nontrivial defects.' [Boehm01]
- 'About 90 percent of the downtime comes from, at most, 10 percent of the defects.' [Boehm]
- 'Only 60% of the features in a system are actually used in production.' [Bernstein05, p. 249]
- 'About 80 percent of the defects come from 20 percent of the modules, and about half the modules are defect free.' [Boehm01] (aka Pareto principle)
- 'More than 60% of the errors in a software product are committed during the design and less than 40% during coding.' [Bernstein05, page 47]
- 'Peer reviews catch 60 percent of the defects.' [Boehm01]
- 'Perspective-based reviews catch 35 percent more defects than nondirected reviews.' [Boehm01]
- 'Disciplined personal practices can reduce defect introduction rates by up to 75 percent.' [Boehm01]
- 'Testing efforts consume 50% of development time and 20% of project cost. Regression testing halves the time and cost.' [Bernstein05, page 386]
- 'When lines of comments exceeded lines of code, code is hard to read.' [Bernstein05, page 363]