Ένωση Πληροφορικών Ελλάδας

Διαχείριση Έργων Λογισμικού Σύγχρονες Μεθοδολογίες

Χάρης Γεωργίου (MSc, PhD)

Ένωση Πληροφορικών Ελλάδας

Στόχοι:

- Πρώτος "καθολικός" φορέας εκπροσώπησης πτυχιούχων Πληροφορικής.
- Αρμόδιος φορέας εκπροσώπησης επαγγελματιών Πληροφορικής.
- Αρμόδιος επιστημονικός "συμβουλευτικός" φορέας για το Δημόσιο.
- Αρωγός της Εθνικής Ψηφιακής Στρατηγικής & Παιδείας της χώρας.

Τομείς παρέμβασης
Ποιοι είναι οι κύριοι τομείς παρεμβάσεων της ΕΠΕ;

1. Εθνική Ψηφιακή Στρατηγική & Οικονομία
2. Εργασιακά (ΤΠΕ), Δημόσιος & ιδιωτικός τομέας
3. Παιδεία (Α′, Β′, Γ′)
4. Έρευνα & Τεχνολογία
5. Έργα & υπηρεσίες ΤΠΕ
6. Ασφάλεια συστημάτων & δεδομένων
7. Ανοικτά συστήματα & πρότυπα
8. Χρήση ΕΛ/ΛΑΚ
9. Πνευματικά δικαιώματα
10. Κώδικας Δεοντολογίας (ΤΠΕ)
11. Κοινωνική μέριμνα (ICT4D)

Harris Georgiou (MSc, PhD) – https://github.com/xgeorgio/info

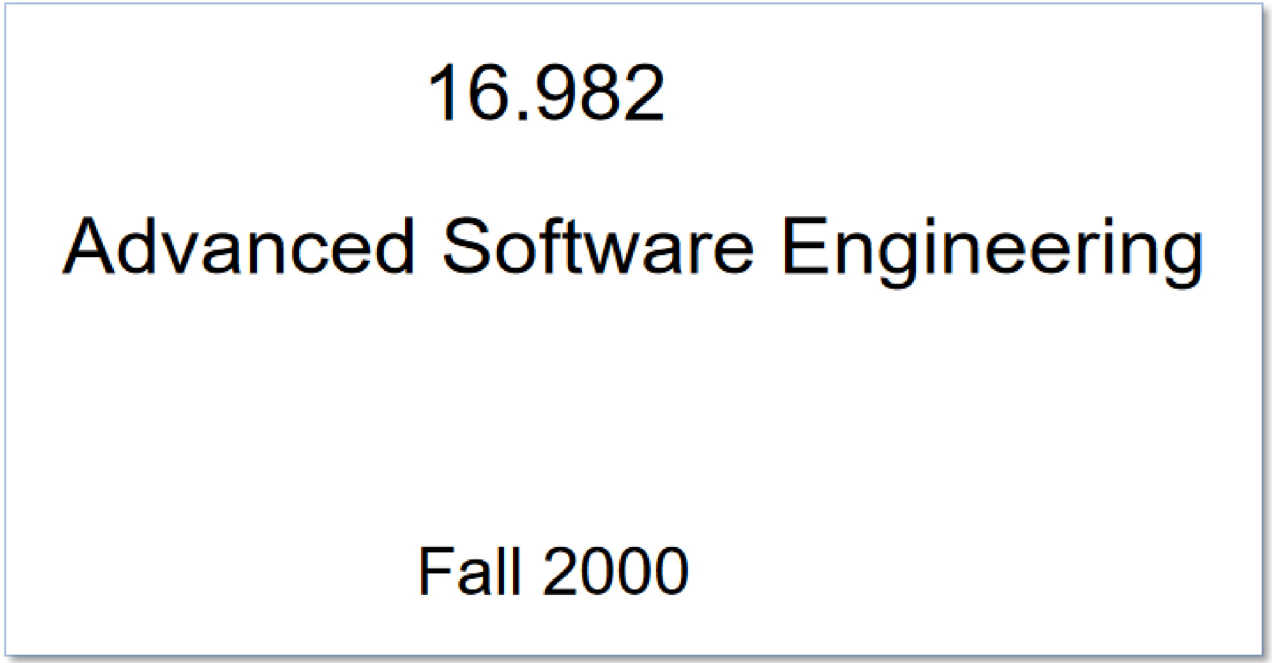
- R&D: Associate post-doc researcher and lecturer with the University Athens (NKUA) and University of Piraeus (UniPi)
- Consultant in Medical Imaging, Machine Learning, Data Analytics, Signal Processing, Process Optimization, Dynamic Systems, Complexity & Emergent A.I., Game Theory
- HRTA member since 2009, LEAR / scientific advisor
- HRTA field operator (USAR, scuba diver)
- Wilderness first aid, paediatric (child/infant)
- Humanitarian aid & disaster relief in Ghana, Lesvos, Piraeus
- Support of unaccomp. minors, teacher in community schools
- Streetwork training, psychological first aid & victim support
- 2+4 books, 170+ scientific papers/articles (and 5 marathons)

Επισκόπηση – Πηγές

- Περιεχόμενα:
  - Τι είναι η «Τεχνολογία Λογισμικού» και η Διαχείριση Έργων Λογισμικού (SPM).
  - Γιατί οι μεθοδολογίες ανάπτυξης λογισμικού είναι απαραίτητες.
  - Κλασικές μεθοδολογίες, πλεονεκτήματα και μειονεκτήματα.
  - Βασικές έννοιες και τεχνικές:
    - Hierarchical design, UML diagrams, modularity, testing methods, ...

- Πηγές:
  - MIT OpenCourseWare (MIT-OCW), Nancy Leveson: 16.355J / ESD.355J Advanced Software Engineering, Fall 2002 – https://dspace.mit.edu/handle/1721.1/35847
  - Stephen Kan, "Metrics and Models in Software Quality Engineering", 2nd ed., Addison-Wesley (2002).
  - Ian Sommerville, "Software Engineering", 8th ed., Addison-Wesley (2008).
  - "2018 Intl. Conf. on Software Engineering: Celebrating its 40th anniversary, and 50 years of Software engineering." ICSE 2018 – Margaret Hamilton – https://www.youtube.com/watch?v=ZbVOF0Uk5lU

Τι είναι η «Τεχνολογία Λογισμικού»;



16.982

Advanced Software Engineering

Fall 2000

Some "Data" (Myths?)

- The development of large applications in excess of 5000 function points (~500,000 LOC) is one of the most risky business undertakings in the modern. world (Capers Jones)
- The risks of cancellation or major delays rise rapidly as the overall application size increases (Capers Jones):
  - 65% of large systems (over 1,000,000 LOC) are cancelled before completion
  - 50% for systems exceeding half million LOC
  - 25 % for those over 100,000 LOC
- Failure or cancellation rate of large software systems is over 20% (Capers Jones)

Copyright © Nancy Leveson, Sept. 2000

More "Data" (Myths?)

- After surveying 8,000 IT projects, Standish Group reported about 30% of all projects were cancelled.

- Average cancelled project in U.S. is about a year behind schedule and has consumed 200% of expected budget (Capers Jones).

- Work on cancelled projects comprises about 15% of total U.S. software efforts, amounting to as much as $14 billion in 1993 dollars (Capers Jones).

Copyright © Nancy Leveson, Sept. 2000

And Yet More

- Of completed projects, 2/3 experience schedule delays and cost overruns (Capers Jones) [bad estimates?]

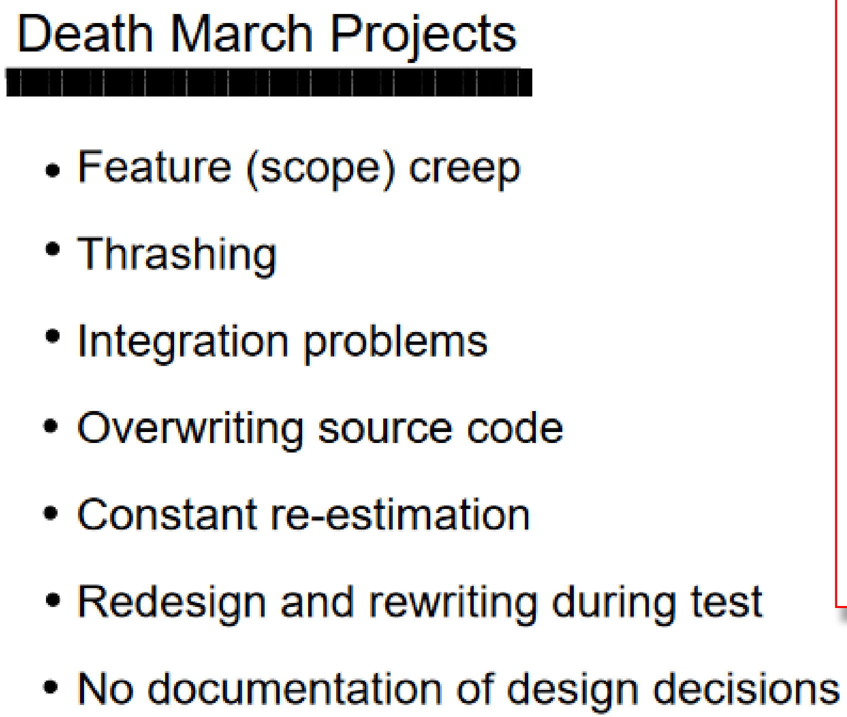
- 2/3 of completed projects experience low reliability and quality problems in first year of deployment (Jones).

- Software errors in fielded systems typically range from 0.5 to 3.0 occurrences per 1000 lines of code (Bell Labs survey).

- Civilian software: at least 100 English words produced for every source code statement.

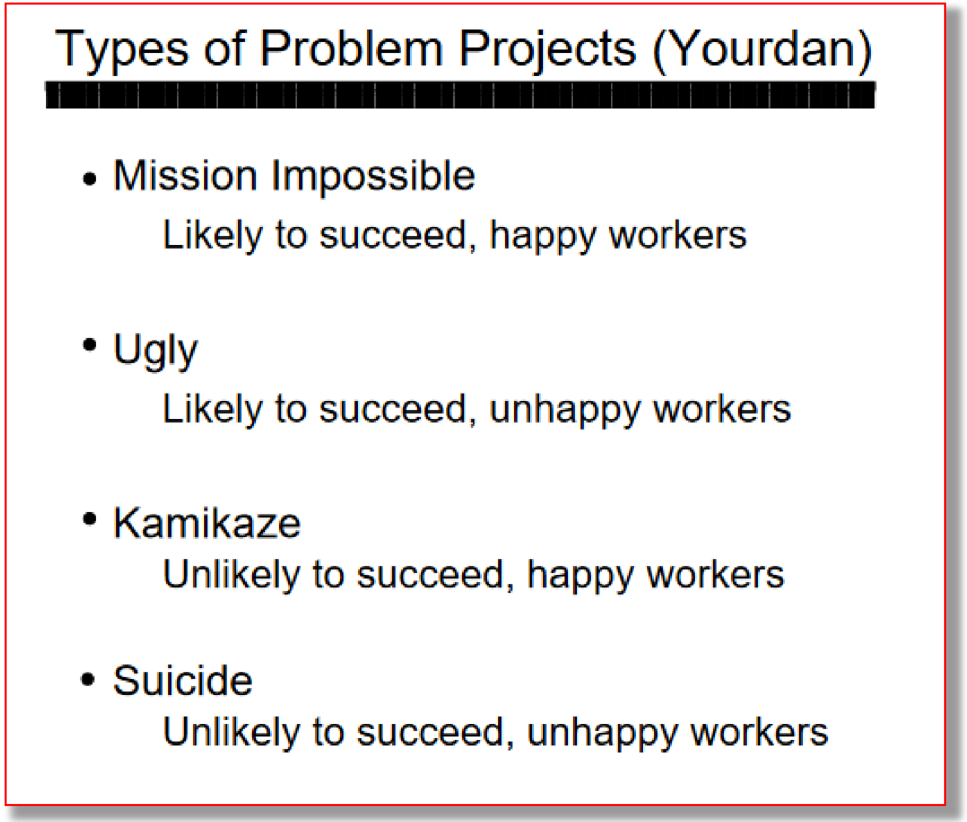
  Military: about 400 words (Capers Jones)

Copyright © Nancy Leveson, Sept. 2000



Death March Projects

- Feature (scope) creep
- Thrashing
- Integration problems
- Overwriting source code
- Constant re-estimation
- Redesign and rewriting during test
- No documentation of design decisions

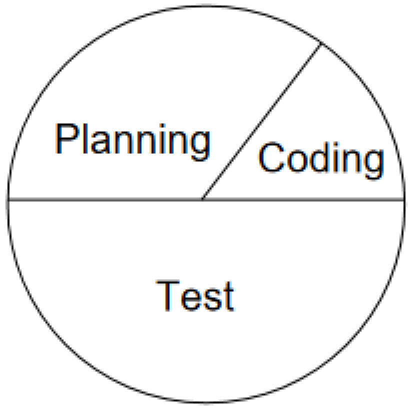


Types of Problem Projects (Yourdan)

- Mission Impossible
  Likely to succeed, happy workers
- Ugly
  Likely to succeed, unhappy workers
- Kamikaze
  Unlikely to succeed, happy workers
- Suicide
  Unlikely to succeed, unhappy workers

Copyright © Nancy Leveson, Sept. 2000

Development Costs

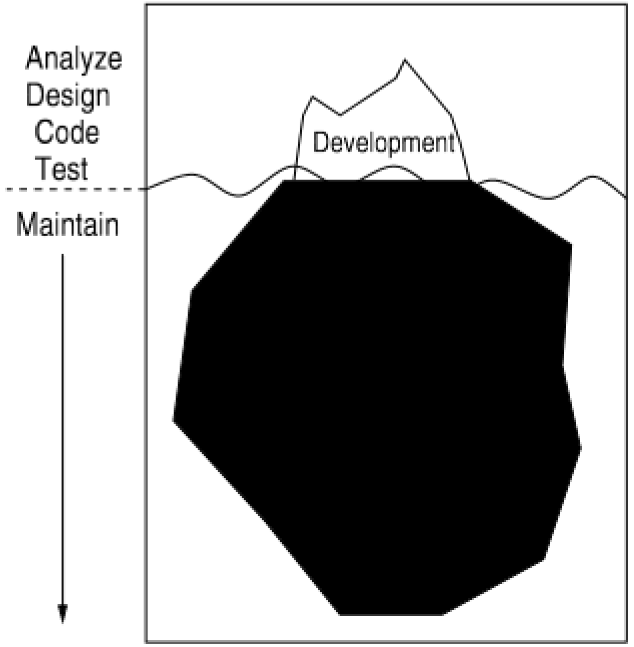


1/3 planning

1/6 coding

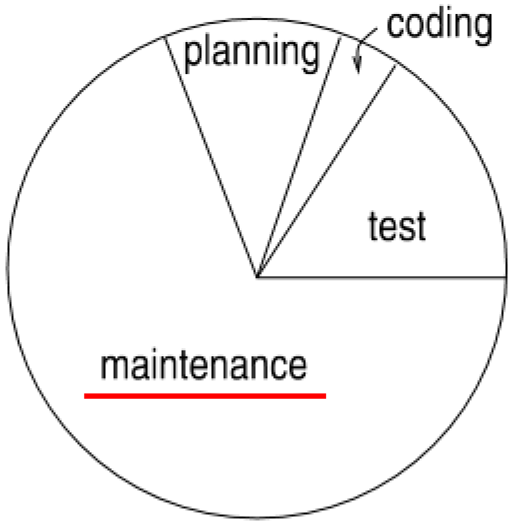
1/4 component test

1/4 system test



Development costs are only the tip of the iceberg.

Copyright © Nancy Leveson, Sept. 2000



Software Maintenance:

20%	error correction
20%	adaptation
60%	enhancements

Most fielded software errors stem from requirements not code

Copyright © Nancy Leveson, Sept. 2000

Are Things Improving?

- Is software improving at a slower rate than hardware?

  "Software expands to fill the available memory" (Parkinson)

  "Software is getting slower more rapidly than hardware becomes faster" (Reiser)

- Expectations are changing

Copyright © Nancy Leveson, Sept. 2000

Why is software engineering hard?

- "Curse of flexibility"
- Organized complexity
- Intangibility
- Lack of historical usage information
- Large discrete state spaces

Copyright © Nancy Leveson, Sept. 2000

The Curse of Flexibility

- "Software is the resting place of afterthoughts."
- No physical constraints
  - To enforce discipline on design, construction and modification
  - To control complexity
- So flexible that start working with it before fully understanding what need to do
- The untrained can get partial success.

  "Scaling up is hard to do"

Copyright © Nancy Leveson, Sept. 2000

What is Complexity?

The underlying factor is intellectual manageability

1. A "simple" system has a small number of unknowns in its interactions within the system and with its environment.

2. A system becomes intellectually unmanageable when the level of interactions reaches the point where they cannot be thoroughly
   - planned
   - understood
   - anticipated
   - guarded against

Copyright © Nancy Leveson, Sept. 2000

Ways to Cope with Complexity

- Analytic Reduction (Descartes)
  - Divide system into distinct parts for analysis purposes.
  - Examine the parts separately.

- Three important assumptions:

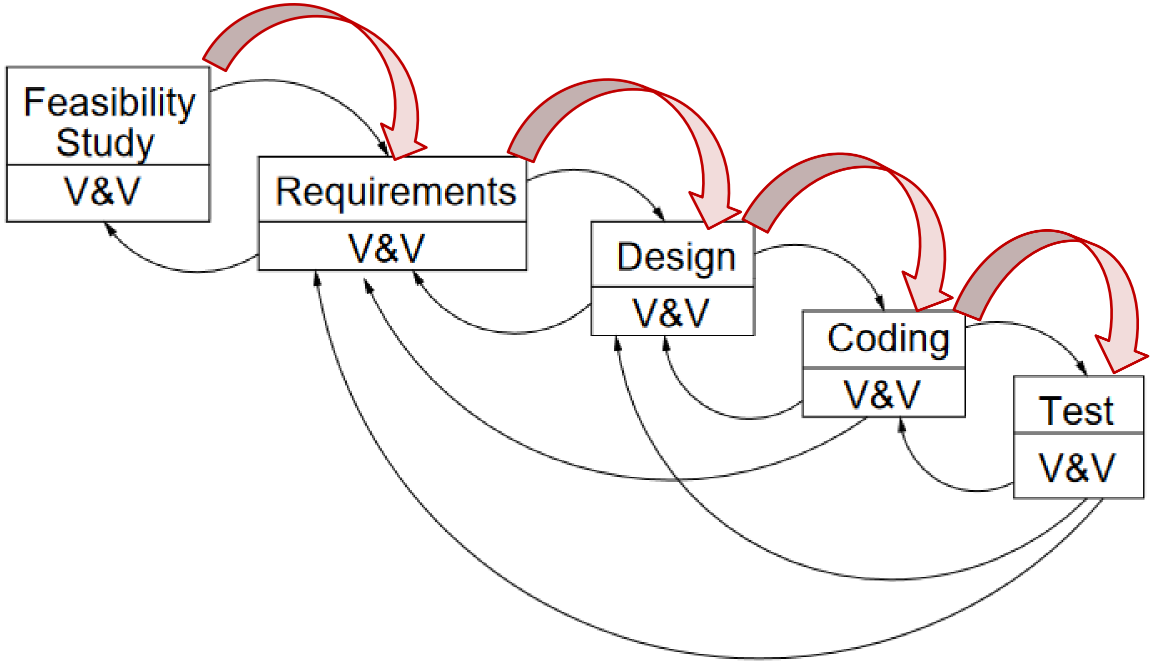
  1. The division into parts will not distort the phenomenon being studied.

  2. Components are the same when examined singly as when playing their part in the whole.

  3. Principles governing the assembling of the components into the whole are themselves straightforward.

Copyright © Nancy Leveson, Sept. 2000

Waterfall Model



- Deliverables – baselines
- Document–driven process
- "Big Bang" testing, "stubs", daily build and smoke test
- "A Rational Design Process and How to Fake It"

Copyright © Nancy Leveson, Sept. 2000

Evolutionary Model

- Prototyping – "Do it twice"
  - to assess feasibility
  - to verify requirements
- May only be a front end or executable specification
  Or develop system with less functionality or quality attributes
- 3 approaches:
  1) Use prototyping as tool for requirements analysis.
     Need proper tools

  2) Use to accomodate design uncertainty.
     Prototype evolves into final product
     Documentation may be sacrificed
     May be less robust
     Quality defects may cause problems later

  3) Use to experiment with different proposed solutions
     before large investments made.

Copyright © Nancy Leveson, Sept. 2000

Evolutionary Models (2)

- Drawbacks:
  - Can be expensive to build
  - Can develop a life of its own − turns out to be product itself
  - Hard to change basic decisions made early
  - Can be an excuse for poor programming practices

- Experimental Evaluation:
  - Boehm: prototyping vs. waterfall

    Waterfall: addressed product and process control risks better
    Resulted in more robust product, easier to maintain
    Fewer problems in debugging and integration due to more thought−out design

    Prototyping: addressed user interfaces better
  - Alavi: prototyping vs. waterfall applied to an information system

    Prototyping: users more positive and more involved
    Waterfall: more robust and efficient data structures

Copyright © Nancy Leveson, Sept. 2000

Incremental Model

- Functionality produced and delivered in small increments.
- Focus attention first on essential features and add functionality only if and when needed
- Systems tend to be leaner -- fights overfunctionality syndrome
- May be hard to add features later
- Variant: Incremental implementation only
  - Follow waterfall down to implementation
  - During requirements analysis and system design
    - Define useful subsets that can be delivered
    - Define interfaces that allow adding later smoothly
  - Different parts implemented, tested, and delivered according to different priorities and at different times.

Copyright © Nancy Leveson, Sept. 2000

Spiral Model

- Includes every other model
- Risk driven (vs. document driven or increment driven)
- Radius of spiral represents cost accumulated so far

Do you need one uniform process over entire project?

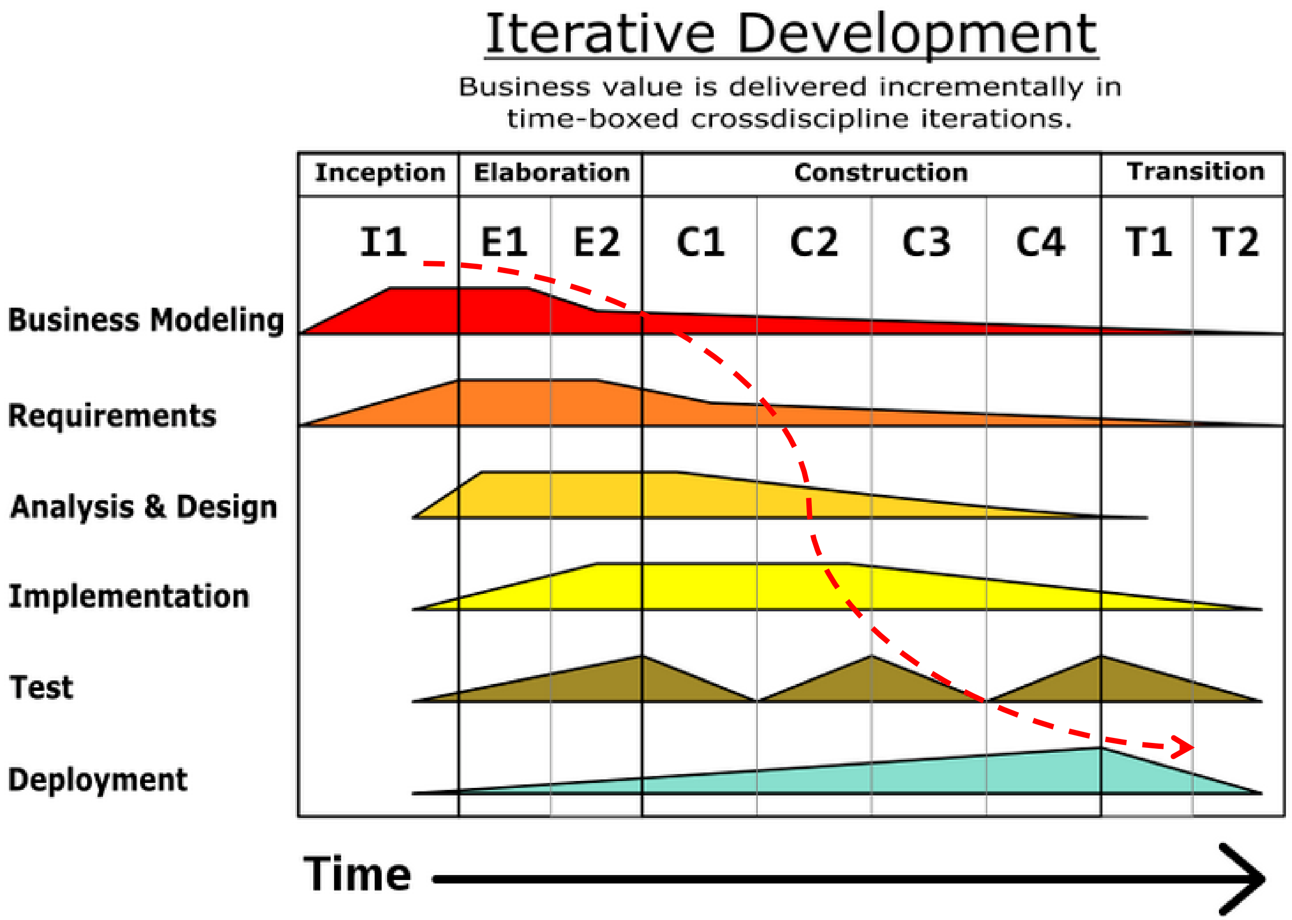
- In requirements analysis, identify aspects that are uncertain

  e.g., library:

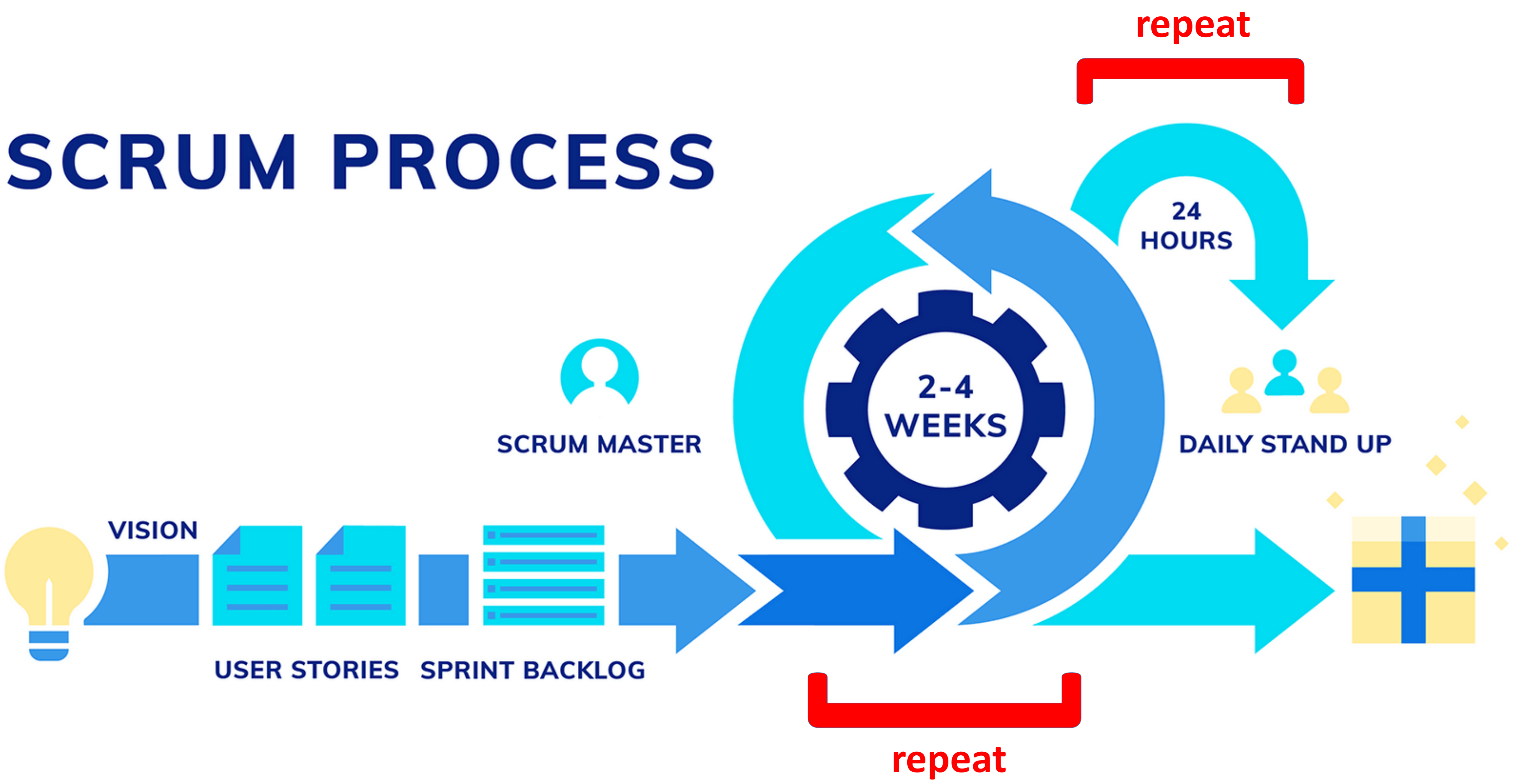
  checkout and checkin (inventory control) − relatively certain
  card catalogue, user search − relatively uncertain

  then have separate processes for the different parts.

Copyright © Nancy Leveson, Sept. 2000

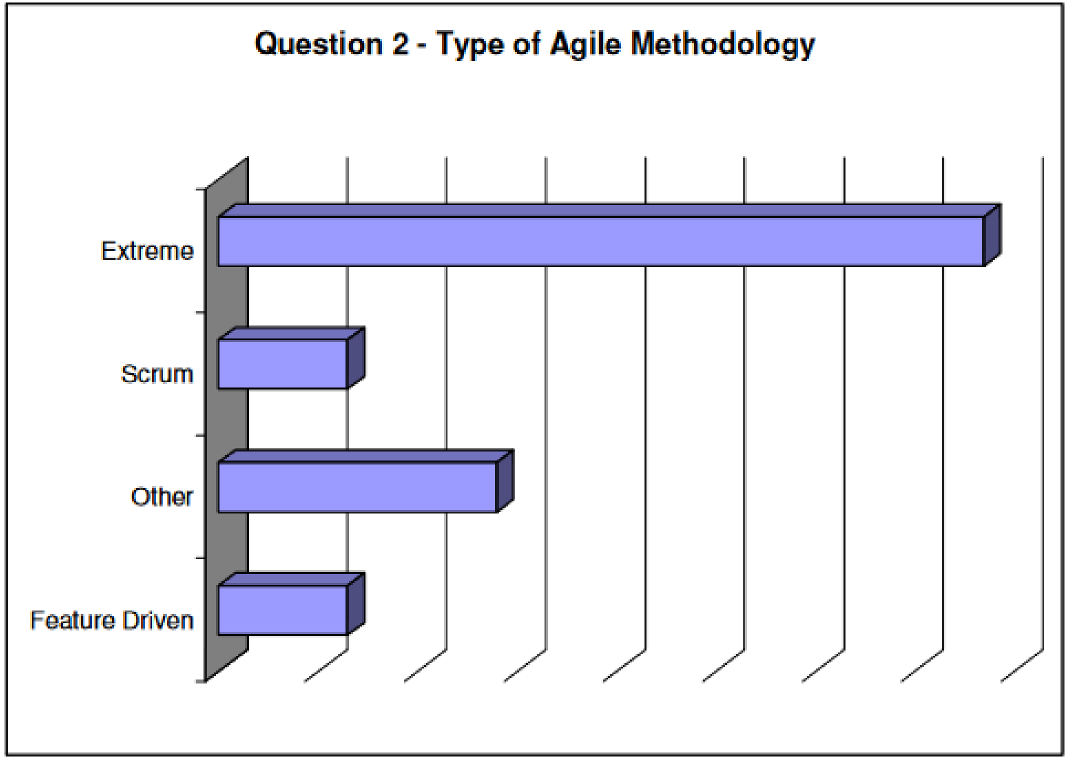


Source: Unified Process Model for Iterative Development (Wikipedia.org)



Source: https://aristeksystems.com/blog/how-not-to-burn-your-budget-with-agile-scrum/

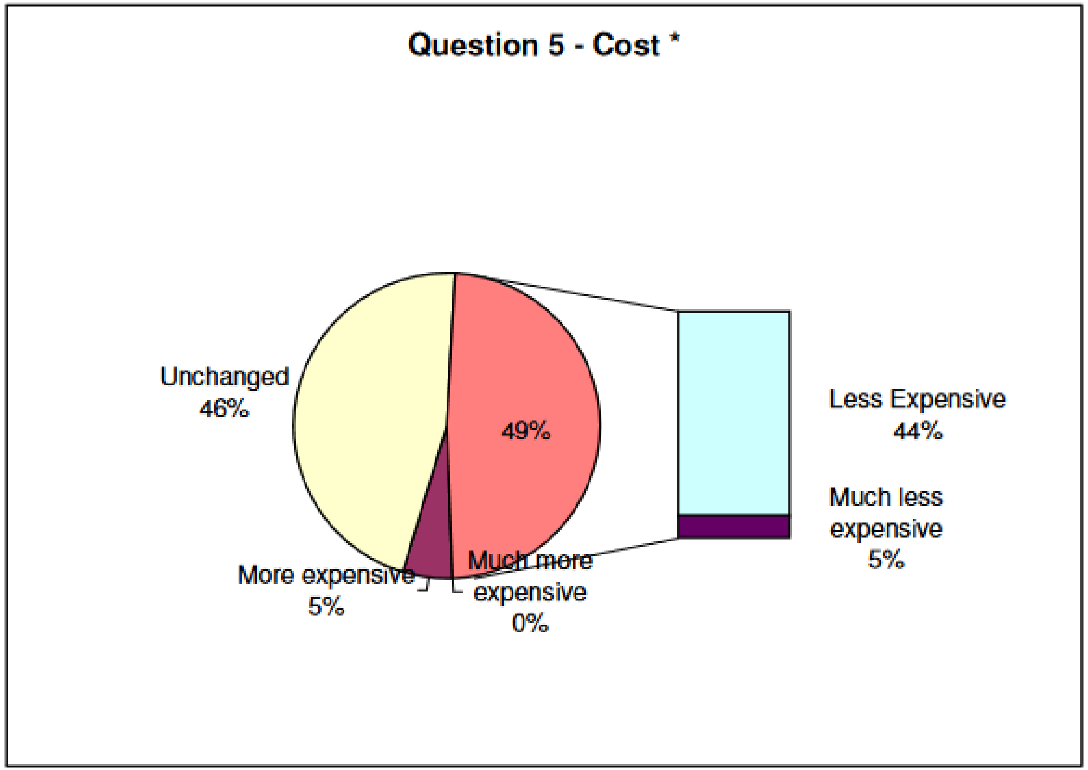
Question 2: What form of Agile processes are you most using at the moment?



By far the most popular form of Agile processes used is Extreme Programming, often shortened to XP. Extreme processes are being used by 59% of all respondents.

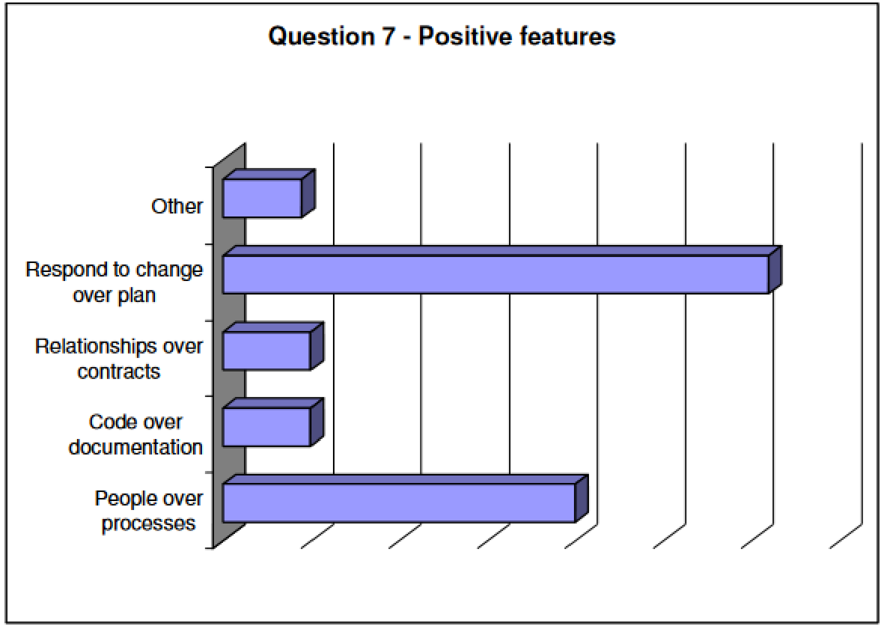
Source: "Agile Methodologies Survey Results“, Shine Technologies, Jan.2003

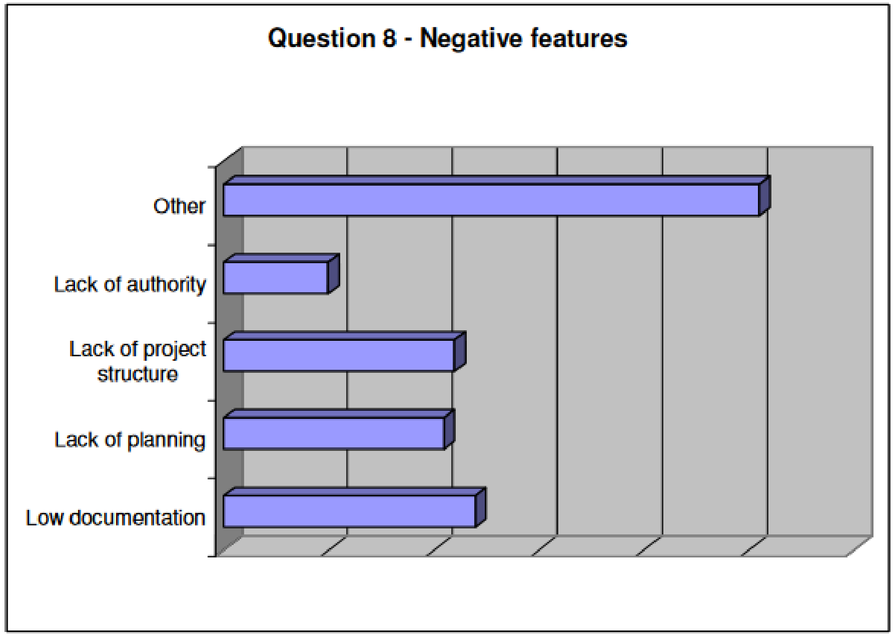
Question 5: Has adoption of Agile processes altered the cost of development?



Across respondents with average knowledge or better, 48.6% believed that development costs were reduced. Including the responses that indicated that costs were unchanged, a whopping 95% believe Agile processes have either no effect or a cost reduction effect.

Source: "Agile Methodologies Survey Results“, Shine Technologies, Jan.2003





Source: "Agile Methodologies Survey Results“, Shine Technologies, Jan.2003

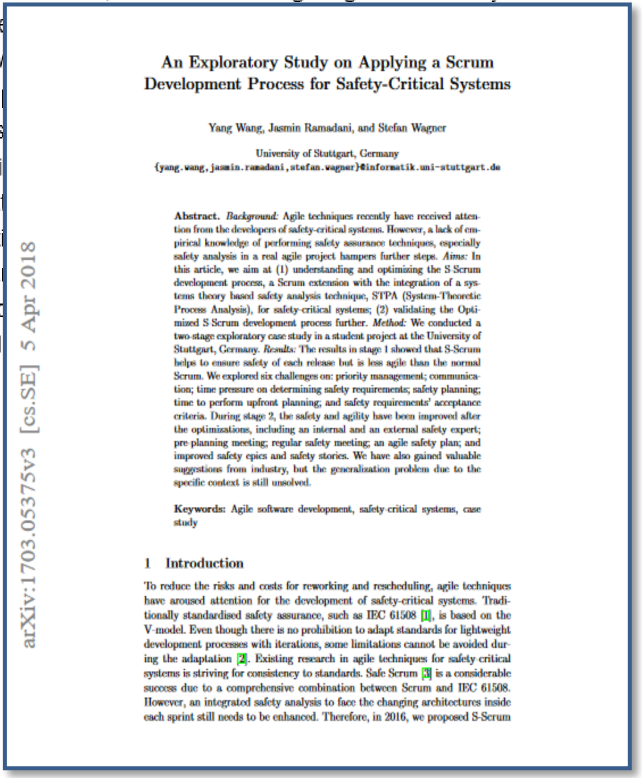
[Submitted on 15 Mar 2017 (v1), last revised 5 Apr 2018 (this version, v3)]

An Exploratory Study of Applying a Scrum Development Process for Safety-Critical Systems

Yang Wang, Jasmin Ramadani, Stefan Wagner

Agile techniques recently have received attention in developing safety-critical systems. However, a lack of empirical knowledge of performing safety assurance
techniques in practice, especially safety analysis into agile development processes prevents further steps. In this article, we aim at investigating the feasibility and the
effects of our S-Scrum development process, and stepwise improving and proposing an Optimized S-Scrum de
environment. We conducted an exploratory case study in a one-year student project "Smart Home" at the Univ
project and collected quantitative and qualitative data from questionnaire, interviews, participant observation,
Furthermore, we evaluated the Optimized S-Scrum in industry by conducting interviews. The first-stage results
Process Analysis) can ensure the safety during each sprint and enhance the safety of delivered products, whi
original Scrum. Six challenges have been explored: Management changes the team's priorities during an iterat
functional requirements are determined too late; Insufficient upfront planning; Insufficient well-defined completi
We investigated further the causalities and optimizations. The second-stage results revealed that the safety a
We have gained a positive assessment and suggestions from industry. The optimized S-Scrum is feasible for
capability to ensure safety and the acceptable agility in a student project. Further attempt is still needed in ind

Comments:	16 pages, 4 figures
Subjects:	**Software Engineering (cs.SE)**
Cite as:	arXiv:1703.05375 [cs.SE]



arXiv:1703.05375v3 [cs.SE] 5 Apr 2018

An Exploratory Study on Applying a Scrum Development Process for Safety-Critical Systems

Yang Wang, Jasmin Ramadani, and Stefan Wagner

University of Stuttgart, Germany
{yang.wang,jasmin.ramadani,stefan.wagner}@informatik.uni-stuttgart.de

Abstract. *Background:* Agile techniques recently have received attention from the developers of safety-critical systems. However, a lack of empirical knowledge of performing safety assurance techniques, especially safety analysis in a real agile project hampers further steps. *Aims:* In this article, we aim at (1) understanding and optimizing the S-Scrum development process, a Scrum extension with the integration of a systems theory based safety analysis technique, STPA (System-Theoretic Process Analysis), for safety-critical systems; (2) validating the Optimized S-Scrum development process further. *Method:* We conducted a two-stage exploratory case study in a student project at the University of Stuttgart, Germany. *Results:* The results in stage 1 showed that S-Scrum helps to ensure safety of each release but is less agile than the normal Scrum. We explored six challenges on: priority management; communication; time pressure on determining safety requirements; safety planning; time to perform upfront planning; and safety requirements' acceptance criteria. During stage 2, the safety and agility have been improved after the optimizations, including an internal and an external safety expert; pre-planning meeting; regular safety meeting; an agile safety plan; and improved safety epics and safety stories. We have also gained valuable suggestions from industry, but the generalization problem due to the specific context is still unsolved.

Keywords: Agile software development, safety-critical systems, case study

1 Introduction

To reduce the risks and costs for reworking and rescheduling, agile techniques have aroused attention for the development of safety-critical systems. Traditionally standardised safety assurance, such as IEC 61508 [1], is based on the V-model. Even though there is no prohibition to adapt standards for lightweight development processes with iterations, some limitations cannot be avoided during the adaptation [2]. Existing research in agile techniques for safety critical systems is striving for consistency to standards. Safe Scrum [3] is a considerable success due to a comprehensive combination between Scrum and IEC 61508. However, an integrated safety analysis to face the changing architectures inside each sprint still needs to be enhanced. Therefore, in 2016, we proposed S-Scrum

Requirements specification:

- A structured document that sets out the services the system is expected to provide.
- Should be precise so that it can act as a contract between the system procurer and software developer and thus needs to be understandable by procurers and developers.
- Describes *what* the system will do but not *how* it will do it (objectives but not how objectives will be achieved).

Design specification:

- An abstract description of the software that serves as a basis for (or describes) its detailed design and implementation.
- Describes *how* the requirements will be achieved.
- Primary readers will be software designers and implementers rather than users or management.
- The goals and constraints specified in requirements document should be traceable to the design specification (and from there to the code).

Copyright © Nancy Leveson, Sept. 2000

Types of Specifications

- Informal
  - Free form, natural language
  - Ambiguity and lack of organization can lead to incompleteness, inconsistency, and misunderstandings
- Formatted
  - Standardized syntax (e.g., UML)
  - Basic consistency and completeness checks
  - Imprecise semantics implies other sources of error may still be present.

Copyright © Nancy Leveson, Sept. 2000

Algebraic Specifications

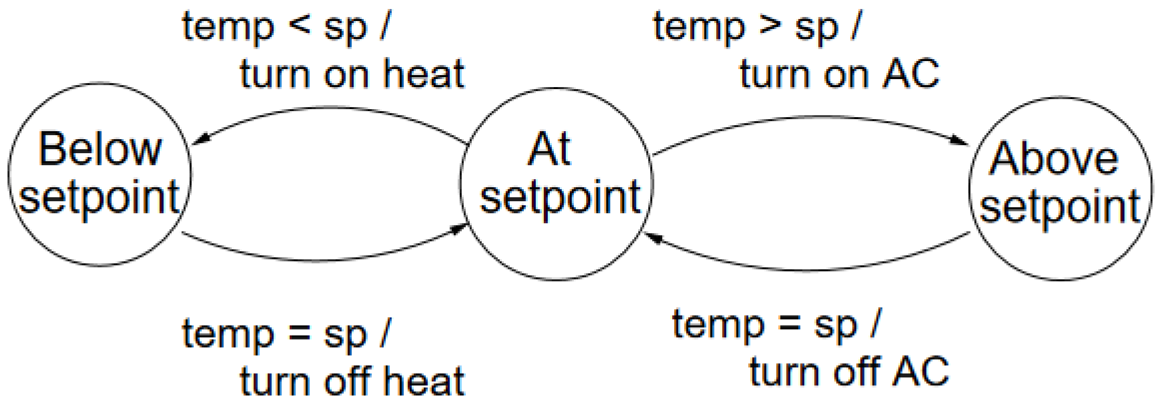
Axioms stated in English:

1. A new stack is empty.
2. A stack is not empty immediately after pushing an item onto it.
3. Attempting to pop a new stack results in an error.
4. There is no top item on a new stack.
5. Pushing an item onto a stack and immediately popping it off leaves the stack unchanged.
6. Pushing an item onto a stack and immediately requesting the top item returns the item just pushed onto the stack.

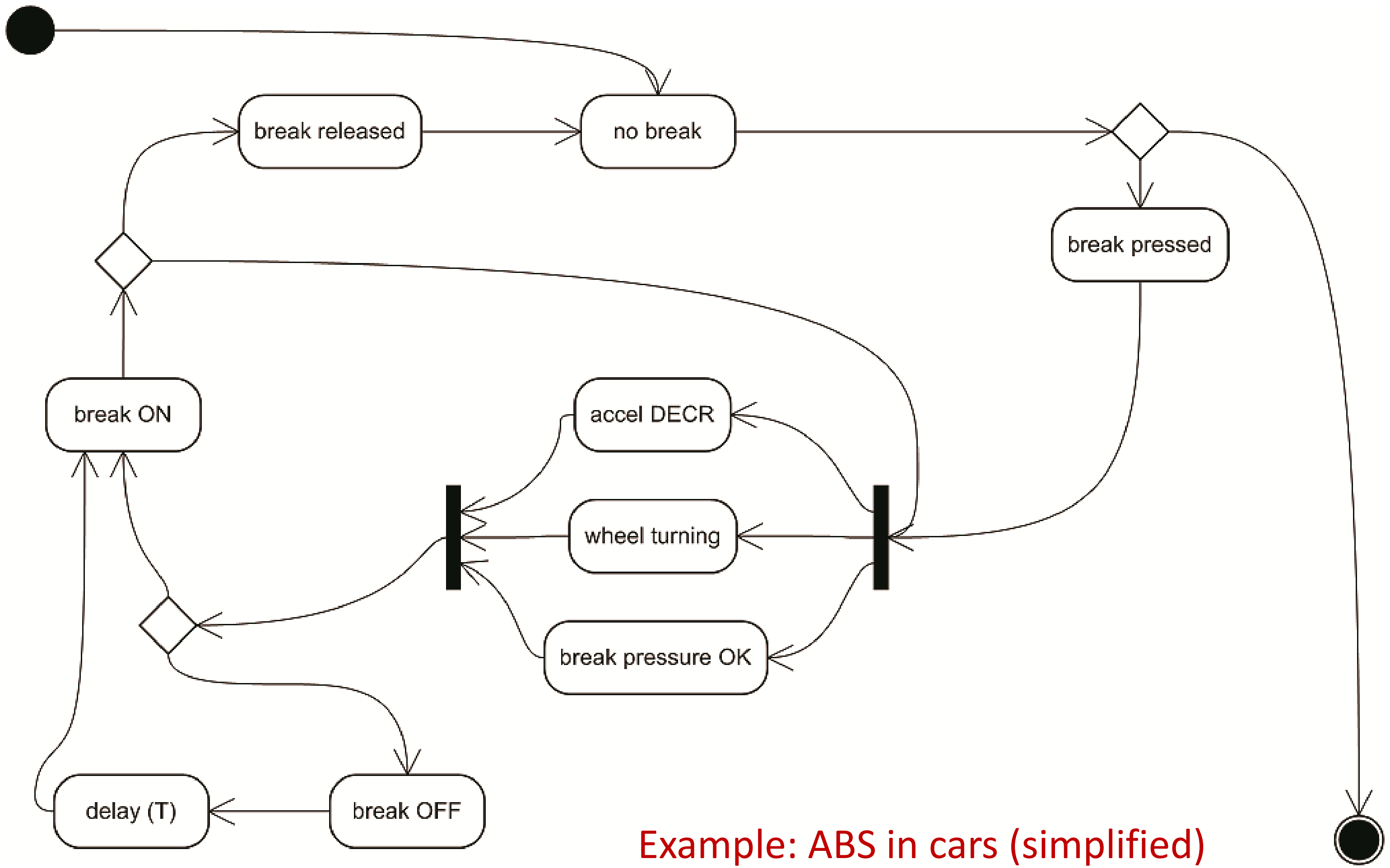
Copyright © Nancy Leveson, Sept. 2000

State Machine Specifications

Define behavior using states and transitions between states



Copyright © Nancy Leveson, Sept. 2000



Example: ABS in cars (simplified)

Structured Programming

- Goal: mastering complexity
- Dijkstra, Hoare, Wirth:
  - Construction of correct programs requires that programs be intellectually manageable
  - Key to intellectual manageability is the structure of the program itself.
  - Disciplined use of a few program building blocks facilitates correctness arguments.

Copyright © Nancy Leveson, Sept. 2000

Structured Programming (2)

- Restricted control structures
- Levels of abstraction
- Stepwise refinement
- Program families
- Abstract data types
- System structure:
  - Programming-in-the-large vs. programming-in-the-small
  - Modularization
  - Minimizing connectivity

Copyright © Nancy Leveson, Sept. 2000

Stepwise Refinement

Wirth (1971): "Divide and conquer"

- A top-down technique for decomposing a system from preliminary design specification of functionality into more elementary levels.
- Program construction consists of sequence of refinement steps.
- Use a notation natural to problem as long as possible.
- Refine function and data in parallel.
- Each refinement step implies design decisions. Should be made explicit.

Copyright © Nancy Leveson, Sept. 2000

Prime Number Program

```
begin var table p;
    fill table p with first 1000 prime numbers
    print table p
end
```

- Assumes type "table" and two operators
- Design decisions made:
  - All primes developed before any printed
  - Always want first 1000 primes
- Decisions not made:
  - Representation of table
  - Method of calculating primes
  - Print format

Copyright © Nancy Leveson, Sept. 2000

Four Primary Design Principles

1. Separation of concerns
   - Deal with separate aspects of a problem separate.
2. Abstraction
   - Identify important aspects of a phenomenon and ignore details that are irrelevant at this stage.
   - Hierarchical abstraction: build hierarchical layers of abstraction
     - Procedural (functional) abstraction
     - Data abstraction
     - Control abstraction (abstract from precise sequence of events handled, e.g., nondeterminacy)

Copyright © Nancy Leveson, Sept. 2000

Four Primary Design Principles (2)

3. Simplicity
   - Emphasis on software that is clear, simple, and therefore easy to check, understand, and modify.

4. Restricted visibility
   - Locality of information

Copyright © Nancy Leveson, Sept. 2000

General Software Design Concepts

Implementations of the general principles

- Decomposition
  - Can decompose with respect to time order, data flow, logical groupings, access to a common resource, control flow, or some other criterion.
  - Functional decomposition seems to be a natural way for people to solve problems as evidenced by its wide use.
  - Top-down decomposition: start at high levels of abstraction and progress to levels of greater and greater detail.
  - Bottom-up: form and layer groups of instruction sequences until work way up to a complete solution.

Copyright © Nancy Leveson, Sept. 2000

General Software Design Concepts (2)

- Decomposition (con't.)
  - Iterative decision making process:
    - List difficult decisions and decisions likely to change
    - Design a module specification to hide each such decision
    - Break module into further design decisions.
    - Continue refining until all design decisions hidden in a module
- Program Families: design for flexibility, not generality

Copyright © Nancy Leveson, Sept. 2000

General Software Design Concepts (3)

- Virtual Machines
  - A module provides a virtual machine: a set of operations that can be invoked in a variety of ways and orders to accomplish a variety of tasks.
  - Don’t think of systems in terms of components that correspond to steps in processing.
  - Do provide a set of virtual machines that are useful for writing many programs.
- Information Hiding
  - Each design unit hides internal details of processing activities.
  - Design units communicate only through well-defined interfaces.
  - Each design unit specified by as little information as possible
  - If internal details change, client units should need no change

Copyright © Nancy Leveson, Sept. 2000

General Software Design Concepts (4)

- Modularity
  - Separation of concerns:
    1. Deal with details of each module in isolation (ignoring details of other modules)
    2. Deal with overall characteristics of all modules and their relationships in order to integrate them into a coherent system.
  - Base on hierarchy and abstraction:
    - Abstraction handled through information hiding
    - Hierarchy by defining uses and is-composed-of relations
  - Minimize connectivity

Copyright © Nancy Leveson, Sept. 2000

General Software Design Concepts (5)

- Modularity (con't.)
  - Sample things to modularize and encapsulate:
    - abstract data types
    - algorithms (e.g., sort)
    - input and output formats
    - processing sequence
    - machine dependencies (e.g., character codes)
    - policies (e.g., when and how to do garbage collection)
    - external interfaces (hardware and software)
  - Benefits:
    - Allows understanding each part of a system separately
    - Aids in modifying system
    - May confine search for a malfunction to a single module.

Copyright © Nancy Leveson, Sept. 2000

- 60s and 70s: people recognized that a systematic approach to development needed to cope with large-scale projects. Needed a way to promulgate and encourage the adoption of desirable practices.

  A procedural form (do this, then do this, then this ...) lent itself to this role.

  Also easily conveyed through books and courses, easy to teach, easy to write exam questions.

  Yourdan, Michael Jackson, etc.

  Met some real needs.

- By late 70s, use of procedural form was entrenched.

Copyright © Nancy Leveson, Sept. 2000

- But some good practices that did not lend themselves to such a form, e.g., information hiding (for which no satisfactory form of procedural development practice has yet been devised).

- Reaction in 80s to shortcomings was to "pile more on"
  - More diagrammatical forms
  - More models
  - More complexity

    "Arguably, much of this complexity stems from the paradox of object orientation, which seems to provide excellent paradigms for analysis and implementation, but present major difficulties for the designer."

- In 90s, attempts to develop other paradigms for transferring design knowledge, e.g., patterns and architectures.

Copyright © Nancy Leveson, Sept. 2000

Basic Testing Guidelines

- A test case has two parts:
  1. Description of input data
  2. Precise description of correct output for that input
- A programmer should avoid testing his or her programs.
- A programming organization should not test its own programs.
- The results of each test should be thoroughly inspected (lots of errors are missed).
- Test cases must be written for invalid and unexpected as well as valid and expected input conditions.

Copyright © Nancy Leveson, Sept. 2000

Black Box Testing

Test data derived solely from specification (i.e., without knowledge of internal structure of program).

- Need to test every possible input

```
x := y * 2
if x = 5 then y := 3
```

(since black box, only way to be sure to detect this is to try every input condition)

  - Valid inputs up to max size of machine (not astronomical)
  - Also all invalid input (e.g., testing Ada compiler requires all valid and invalid programs)
  - If program has "memory", need to test all possible unique valid and invalid sequences.

- So for most programs, exhaustive input testing is impractical.

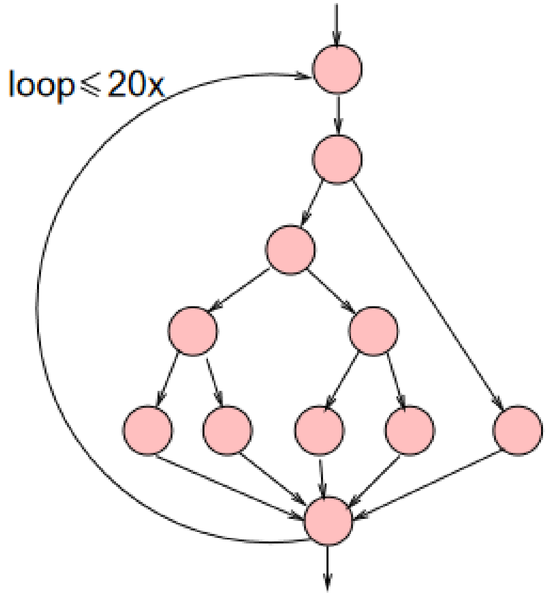
Copyright © Nancy Leveson, Sept. 2000

White Box Testing

Derive test data by examining program's logic.

Exhaustic path testing: Two flaws

1) Number of unique paths through program is astronomical.



(control-flow graph)

$$5^{20} + 5^{19} + 5^{18} + \ldots + 5 = 10^{14}$$

$$= 100 \text{ trillion}$$

If could develop/execute/verify one test case every five minutes = 1 billion years

If had magic test processor that could develop/execute/evaluate one test per msec = 3170 years.

Copyright © Nancy Leveson, Sept. 2000

Static Analysis

- Syntax checks
- Look for error-prone constructions
  (enforce standards)
- Program structure checks

  Generate graphs and look for structural flaws
- Module interface checks

  Detect inconsistencies in declarations of data structures
  and improper linkages between modules
- Human Reviews

  Checklists (inspections)

  Walkthroughs (reviews)

Copyright © Nancy Leveson, Sept. 2000

Software Inspections

- Started by IBM in 1972 (Fagan)
- Process driven by a checklist of likely errors
  - Build checklists through experience and feedback.
  - Some companies consider checklists proprietary.
- Performed after design complete and after coding complete.
- Last about 2 hours, cover about 100 statements per hour.

Evaluation of walkthroughs and inspections:

- Find about 70-80% of errors.
- Most errors found before unit testing.

Copyright © Nancy Leveson, Sept. 2000

Some Decision Factors

- Features of application:
  - Hard real time?
    - Not just efficiency
    - Predictability (need to guarantee deadlines will be met)
  - High assurance?
  - Portability?
  - Maintainability?
  - Others?
- Features of development environment:
  - Availability of programmers, compilers, development tools?
  - Schedule constraints?
  - Others?

Copyright © Nancy Leveson, Sept. 2000

Relationship between PL and Correctness

"PL" = Programming Language

- Error Proneness
  - Language design should prevent errors.

    Should be difficult or impossible to write an incorrect program.
  - If not possible, then allow their detection (as early as possible)
  - Need for general principles and hypotheses so can predict error-prone features and improve language design.
  - Some hypotheses and data about:

    Go to
    Global variables
    Pointers
    Selection by position (long parameter lists)
    Defaults and implicit type conversion
    Attempts to interpret intentions or fix errors
  - Meaning of features should be precisely defined (not dependent on compiler.

Copyright © Nancy Leveson, Sept. 2000

Relationship between PL and Correctness

"PL" = Programming Language

- Understandability
  - "The primary goal of a programming language is accurate communication among humans."
  - Readability more important than writeability.
    - Well "punctuated" (easy to directly determine statement types and major subunits without intermediate inferences)
    - Use of distinct structural words (keywords, reserved words) for distinct concepts (no overloading, e.g., = for equal, assignment)
    - Avoidance of multiple use of symbols unless serve completely analogous functions (e.g., commas as separators, parentheses for grouping).
  - Necessary to be able to see what is being accomplished at a higher level of abstraction.
    - Permit programmers to state their "intentions" along with instructions necessary to carry them out.

Copyright © Nancy Leveson, Sept. 2000

Relationship between PL and Correctness

"PL" = Programming Language

- Maintainability
  - Locality -- possible to isolate changes.
  - Self-documenting
    - Programming decisions should be recorded in program, independent of external documentation.
    - Good comment convention, freedom to choose meaningful variable names, etc.
    - User-defined types and named constants

      e.g., type direction=(north, south, east, west)
  - Explicit interfaces
    - Should cater to construction of hierarchies of modules

Copyright © Nancy Leveson, Sept. 2000

Can programming language influence correctness?

- Languages affect the way we think about problems:

  "The tools we use have a profound (and devious) influence on our thinking habits, and, therefore on our thinking abilities?"

  Dijkstra, 1982

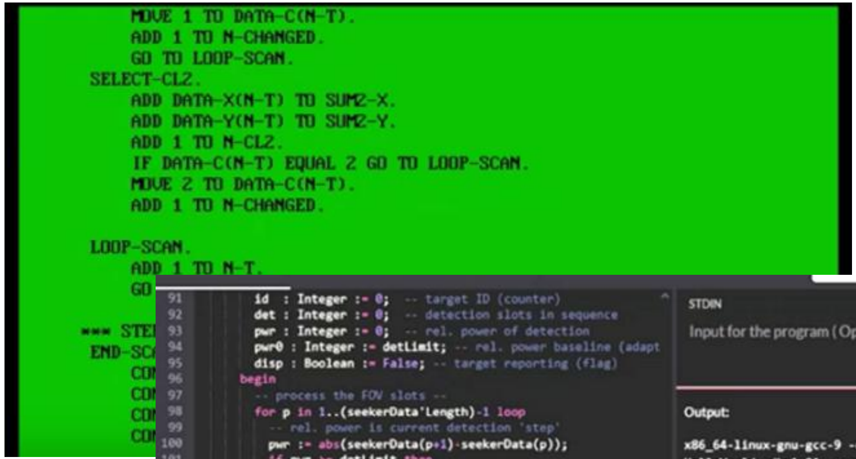
- Additional experimental evidence:
  - C130J software written in a variety of languages by a variety of vendors.
  - All certified to DO−178B standards (FAA).
  - Then subjected to a major IV&V exercise by the MoD
    - Significant, safety−related errors found in Level A certified software
    - Residual error rate of Ada code on aircraft was one tenth that of code written in C.
    - Residual error rate of SPARK code (Ada subset) one tenth that of the Ada code.

Copyright © Nancy Leveson, Sept. 2000

Σύνοψη

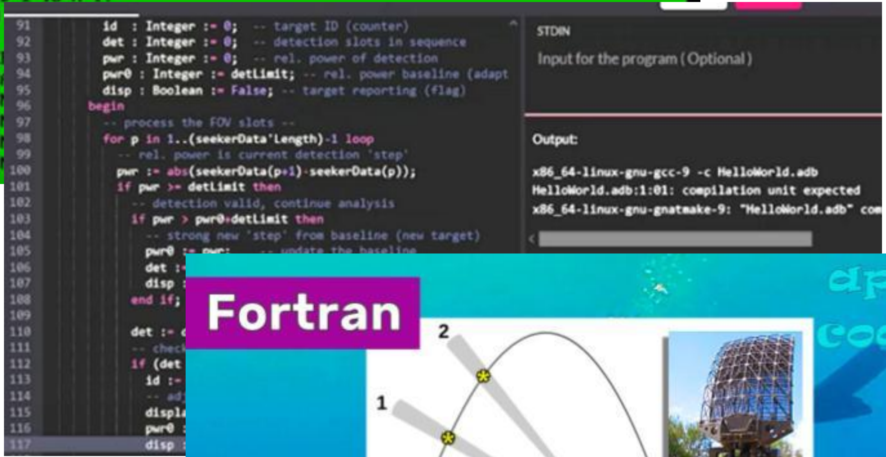
- Περιεχόμενα:
  - Τι είναι η «Τεχνολογία Λογισμικού» και η Διαχείριση Έργων Λογισμικού (SPM).
  - Γιατί οι μεθοδολογίες ανάπτυξης λογισμικού είναι απαραίτητες.
  - Κλασικές μεθοδολογίες, πλεονεκτήματα και μειονεκτήματα.
  - Βασικές έννοιες και τεχνικές:
    - Hierarchical design, UML diagrams, modularity, testing methods, ...

- Πηγές:
  - MIT OpenCourseWare (MIT-OCW), Nancy Leveson: 16.355J / ESD.355J Advanced Software Engineering, Fall 2002 – https://dspace.mit.edu/handle/1721.1/35847
  - Stephen Kan, "Metrics and Models in Software Quality Engineering", 2nd ed., Addison-Wesley (2002).
  - Ian Sommerville, "Software Engineering", 8th ed., Addison-Wesley (2008).
  - "2018 Intl. Conf. on Software Engineering: Celebrating its 40th anniversary, and 50 years of Software engineering." ICSE 2018 – Margaret Hamilton – https://www.youtube.com/watch?v=ZbVOF0Uk5lU



```
        MOVE 1 TO DATA-C(N-T).
        ADD 1 TO N-CHANGED.
        GO TO LOOP-SCAN.
    SELECT-CL2.
        ADD DATA-X(N-T) TO SUM2-X.
        ADD DATA-Y(N-T) TO SUM2-Y.
        ADD 1 TO N-CL2.
        IF DATA-C(N-T) EQUAL 2 GO TO LOOP-SCAN.
        MOVE 2 TO DATA-C(N-T).
        ADD 1 TO N-CHANGED.

    LOOP-SCAN.
        ADD 1 TO N-T.
        GO
  *** STE
    END-SC
        CO
        CO
        CO
        CO
```



```
      id  : Integer := 0;   -- target ID (counter)
      det : Integer := 0;   -- detection slots in sequence
      pwr : Integer := 0;   -- rel. power of detection
      pwr0 : Integer := detLimit; -- rel. power baseline (adapt
      disp : Boolean := False; -- target reporting (flag)
    begin
      -- process the FOV slots --
      for p in 1..(seekerData'Length)-1 loop
        -- rel. power is current detection 'step'
        pwr := abs(seekerData(p+1)-seekerData(p));
        if pwr >= detLimit then
          -- detection valid, continue analysis
          if pwr > pwr0+detLimit then
            -- strong new 'step' from baseline (new target)
            pwr0 := pwr;     -- update the baseline
            det :
            disp :
          end if;

          det :=
          -- check
          if (det
            id :=
            -- adj
            displa
            pwr0 :
            disp :
```

STDIN

Input for the program (Optional)

Output:

```
x86_64-linux-gnu-gcc-9 -c HelloWorld.adb
HelloWorld.adb:1:01: compilation unit expected
x86_64-linux-gnu-gnatmake-9: "HelloWorld.adb" com
```

- Hamming (7,4) error correction codes in R
- Kmeans clustering in COBOL
- Bi-directional Associative Memory (BAM) in Arduino/C
- Linear Regression in SQL, Matlab
- k-nearest-neighbor Classifier in SQL
- ...

YouTube:

@ApneaCoding

https://www.youtube.com/@apneacoding

Github:

@xgeorgio

https://github.com/xgeorgio

Ερωτήσεις

Χάρης Γεωργίου (MSc,PhD)
https://www.linkedin.com/in/xgeorgio/
https://twitter.com/xgeorgio_gr