Lecture 20: Software Evolution

- Basics of Software Evolution
  - Laws of software evolution
  - Requirements Growth
  - Software Aging
- Basics of Change Management
  - Baselines, Change Requests and Configuration Management
- Software Families - The product line approach
- Requirements Traceability
  - Importance of traceability
  - Traceability tools

Program Types

Source: *Adapted from Lehman 1980, pp1061-1063*

→ S-type Programs ("Specifiable")

- problem can be stated formally and completely
- acceptance: Is the program correct according to its specification?
- This software does not evolve.
  - A change to the specification defines a new problem, hence a new program

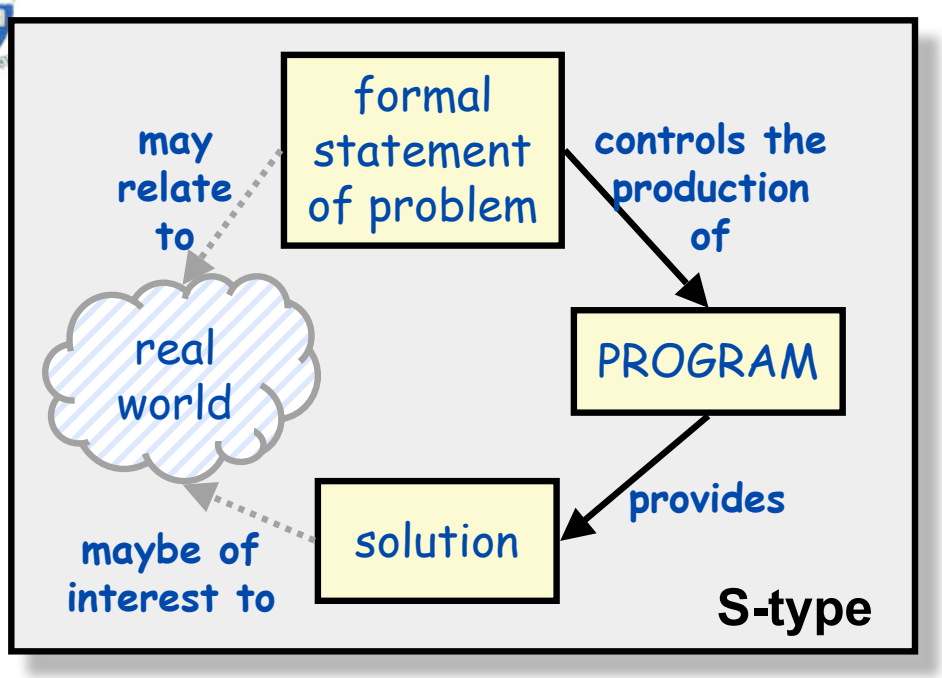
→ P-type Programs ("Problem-solving")

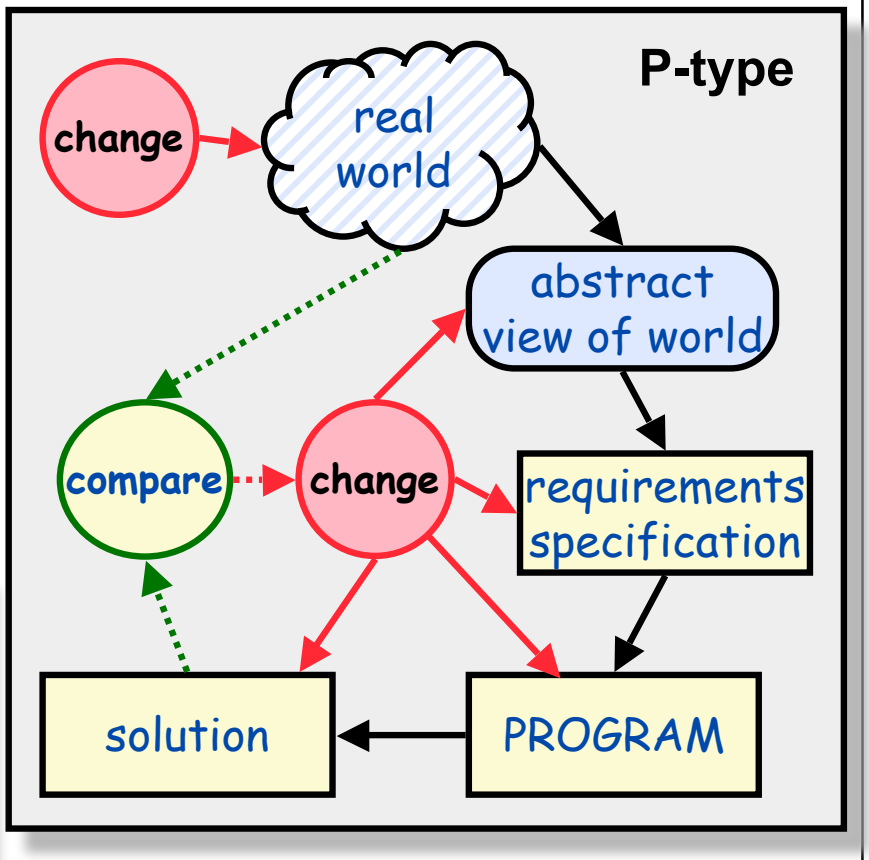
- imprecise statement of a real-world problem
- acceptance: Is the program an acceptable solution to the problem?
- This software is likely to evolve continuously
  - because the solution is never perfect, and can be improved
  - because the real-world changes and hence the problem changes

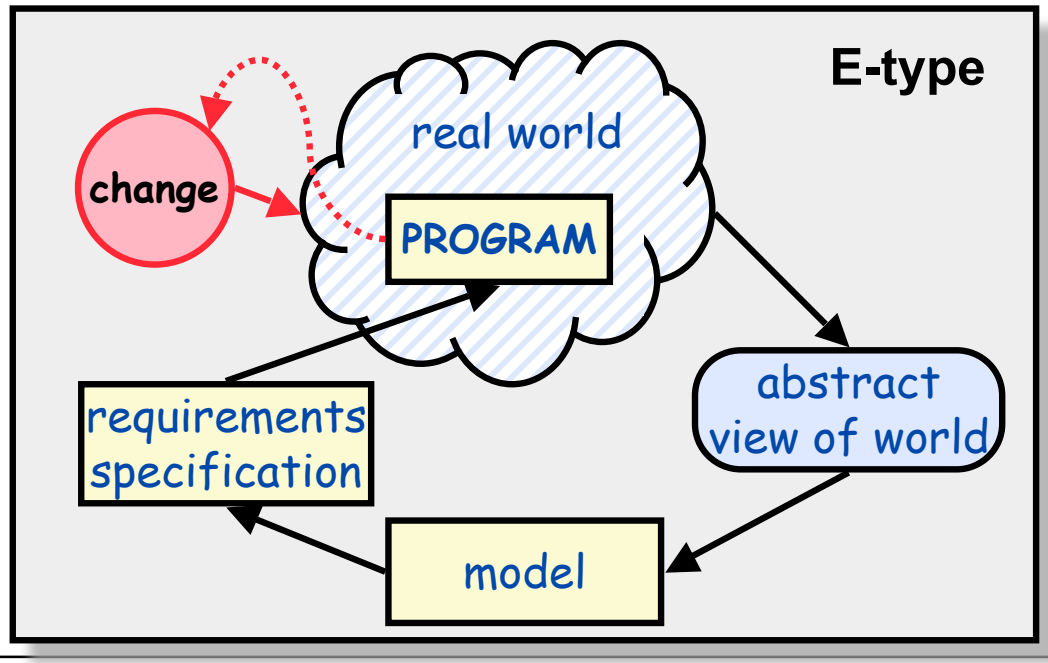
→ E-type Programs ("Embedded")

- A system that becomes part of the world that it models
- acceptance: depends entirely on opinion and judgement
- This software is inherently evolutionary
  - changes in the software and the world affect each other

Source: *Adapted from Lehman 1980, pp1061-1063*







Laws of Program Evolution

Source: *Adapted from Lehman 1980, pp1061-1063*

→ Continuing Change

- Any software that reflects some external reality undergoes continual change or becomes progressively less useful
  - change continues until it is judged more cost effective to replace the system

→ Increasing Complexity

- As software evolves, its complexity increases…
  - …unless steps are taken to control it.

→ Fundamental Law of Program Evolution

- Software evolution is self-regulating
  - …with statistically determinable trends and invariants

→ Conservation of Organizational Stability

- During the active life of a software system, the work output of a development project is roughly constant (regardless of resources!)

→ Conservation of Familiarity

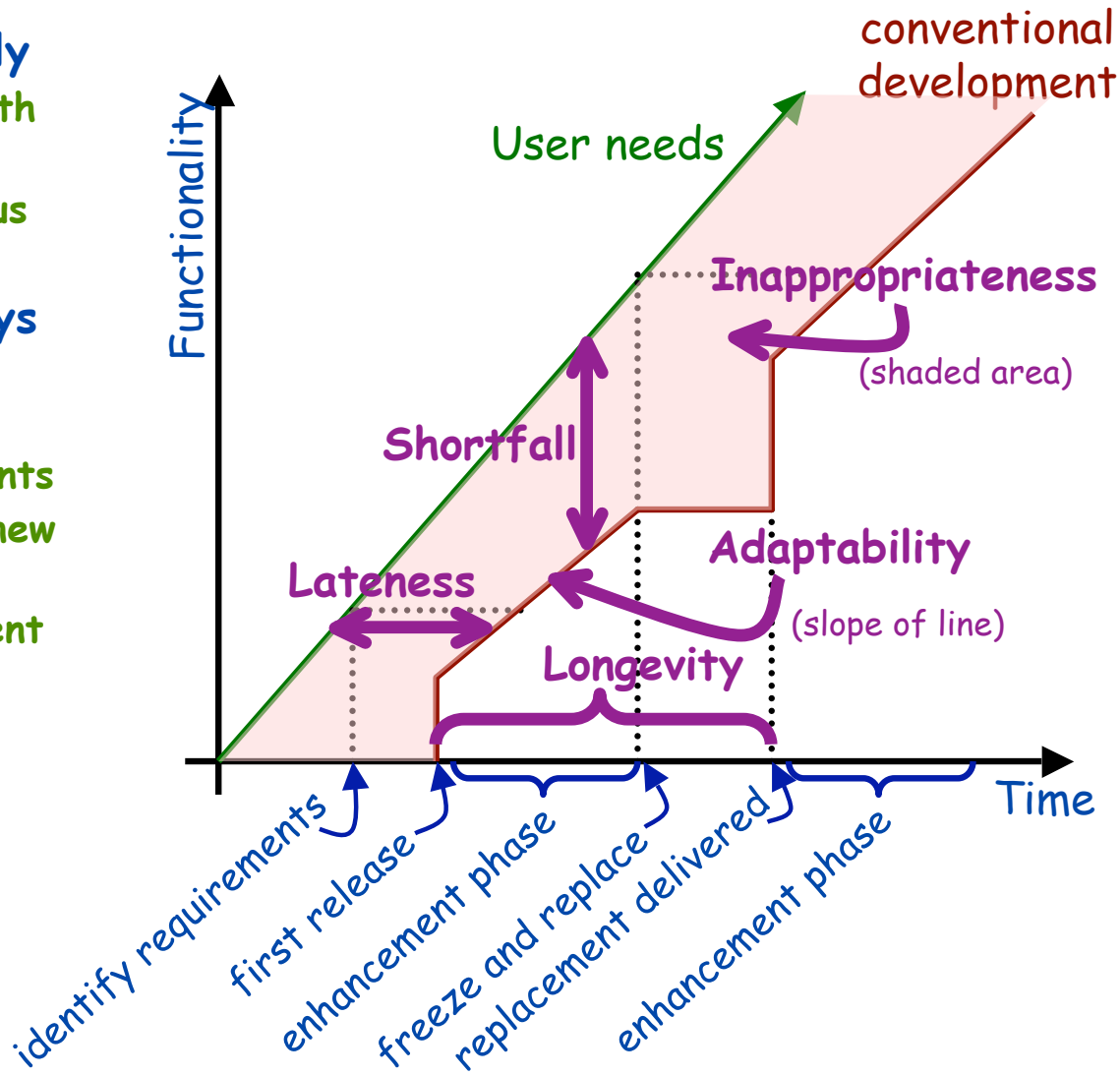
- The amount of change in successive releases is roughly constant

Requirements Growth

Source: *Adapted from Davis 1988, pp1453-1455*

→Davis's model:

- User needs evolve continuously
  - Imagine a graph showing growth of needs over time
  - May not be linear or continuous (hence no scale shown)
- Traditional development always lags behind needs growth
  - first release implements only part of the original requirements
  - functional enhancement adds new functionality
  - eventually, further enhancement becomes too costly, and a replacement is planned
  - the replacement also only implements part of its requirements,
  - and so on...



Software Aging

Source: *Adapted from Parnas, 1994*

→ Causes of Software Aging

- Failure to update the software to meet changing needs
  - Customers switch to a new product if benefits outweigh switching costs
- Changes to software tend to reduce its coherence

→ Costs of Software Aging

- Owners of aging software find it hard to keep up with the marketplace
- Deterioration in space/time performance due to deteriorating structure
- Aging software gets more buggy
  - Each "bug fix" introduces more errors than it fixes

→ Ways of Increasing Longevity

- Design for change
- Document the software carefully
- Requirements and designs should be reviewed by those responsible for its maintenance
- Software Rejuvenation…

Software "maintenance"

Source: *Adapted from Blum, 1992, p492-495*

→ Maintenance philosophies

- "throw-it-over-the-wall" - someone else is responsible for maintenance
  - investment in knowledge and experience is lost
  - maintenance becomes a reverse engineering challenge
- "mission orientation" - development team make a long term commitment to maintaining/enhancing the software

→ Basili's maintenance process models:

- Quick-fix model
  - changes made at the code level, as easily as possible
  - rapidly degrades the structure of the software
- Iterative enhancement model
  - Changes made based on an analysis of the existing system
  - attempts to control complexity and maintain good design
- Full-reuse model
  - Starts with requirements for the new system, reusing as much as possible
  - Needs a mature reuse culture to be successful

Managing Requirements Change

- Managers need to respond to requirements change
  - Add new requirements during development
    - But not succumbing to feature creep
  - Modify requirements during development
    - Because development is a learning process
  - Remove requirements during development
    - requirements "scrub" for handling cost/schedule slippage
- Key techniques
  - Change Management Process
  - Release Planning
  - Requirements Prioritization (previous lecture!)
  - Requirements Traceability
  - Architectural Stability (next week's lecture)

Change Management

→ Configuration Management

- Each distinct product is a Configuration Item (CI)
- Each configuration item is placed under version control
- Control which version of each CI belongs in which build of the system

→ Baselines

- A baseline is a stable version of a document or system
  - Safe to share among the team
- Formal approval process for changes to be incorporated into the next baseline

→ Change Management Process

- All proposed changes are submitted formally as change requests
- A review board reviews these periodically and decides which to accept
  - Review board also considers interaction between change requests

Towards Software Families

→ Libraries of Reusable Components

- domain specific libraries (e.g. Math libraries)
- program development libraries (e.g. Java AWT, C libraries)

→ Domain Engineering

- Divides software development into two parts:
  - domain analysis - identifies generic reusable components for a problem domain
  - application development - uses the domain components for specific applications.

→ Software Families

- Many companies offer a range of related software systems
  - Choose a stable architecture for the software family
  - identify variations for different members of the family
- Represents a strategic business decision about what software to develop
- Vertical families
  - e.g. 'basic', 'deluxe' and 'pro' versions of a system
- Horizontal families
  - similar systems used in related domains

Requirements Traceability

- From IEEE-STD-830:
  - Backward traceability
    - i.e. to previous stages of development.
    - the origin of each requirement should be clear
  - Forward traceability
    - i.e., to all documents spawned by the SRS.
    - Facilitation of referencing of each requirement in future documentation
    - depends upon each requirement having a unique name or reference number.
- From DOD-STD-2167A:
  - *A requirements specification is traceable if:*
    - (1) it contains or implements all applicable stipulations in predecessor document
    - (2) a given term, acronym, or abbreviation means the same thing in all documents
    - (3) a given item or concept is referred to by the same name in the documents
    - (4) all material in the successor document has its basis in the predecessor document, that is, no untraceable material has been introduced
    - (5) the two documents do not contradict one another

Importance of Traceability

Source: *Adapted from Palmer, 1996, p365*

- **Verification and Validation**
  - assessing adequacy of test suite
  - assessing conformance to requirements
  - assessing completeness, consistency, impact analysis
  - assessing over- and under-design
  - investigating high level behavior impact on detailed specifications
  - detecting requirements conflicts
  - checking consistency of decision making across the lifecycle
- **Maintenance**
  - Assessing change requests
  - Tracing design rationale
- **Document access**
  - ability to find information quickly in large documents
- **Process visibility**
  - ability to see how the software was developed
  - provides an audit trail
- **Management**
  - change management
  - risk management
  - control of the development process

Traceability Difficulties

Source: *Adapted from Palmer, 1996, p365-6*

- **Cost**
  - very little automated support
  - full traceability is very expensive and time-consuming
- **Delayed gratification**
  - the people defining traceability links are not the people who benefit from it
    - development vs. V&V
  - much of the benefit comes late in the lifecycle
    - testing, integration, maintenance
- **Size and diversity**
  - Huge range of different document types, tools, decisions, responsibilities,…
  - No common schema exists for classifying and cataloging these
  - In practice, traceability concentrates only on baselined requirements

Current Practice

Source: *Adapted from Palmer, 1996, p367-8*

- **Coverage:**
  - links from requirements forward to designs, code, test cases,
  - links back from designs, code, test cases to requirements
  - links between requirements at different levels
- **Traceability process**
  - Assign each sentence or paragraph a unique id number
  - Manually identify linkages
  - Use manual tables to record linkages in a document
  - Use a traceability tool (database) for project wide traceability
  - Tool then offers ability to
    - follow links
    - find missing links
    - measure overall traceability

Limitations of Current Tools

Source: *Adapted from Gotel & Finkelstein, 1993, p100*

→ Informational Problems

- Tools fail to track *useful* traceability information
  - e.g cannot answer queries such as "who is responsible for this piece of information?"
- inadequate pre-requirements traceability
  - "where did this requirement come from?"

→ Lack of agreement…

- …over the quantity and type of information to trace

→ Informal Communication

- People attach great importance to personal contact and informal communication
  - These always supplement what is recorded in a traceability database
- But then the traceability database only tells part of the story!
  - Even so, finding the appropriate people is a significant problem

Problematic Questions

Source: *Adapted from Gotel & Finkelstein, 1997, p100*

→ Involvement

- Who has been involved in the production of this requirement and how?

→ Responsibility & Remit

- Who is responsible for this requirement?
- What group has authority to make decisions about this requirement?

→ Change

- What changes are relevant to this requirement?
  - Stakeholders' changed jobs? changed development process?
  - When has responsibility for the requirement changed hands?

→ Notification

- Who needs to be involved in, or informed of, any changes proposed to this requirement?

→ Loss of knowledge

- What loss of project knowledge is likely if a specific individual leaves?

Summary

- Software Evolution is inevitable
  - Software must evolve or become progressively less useful
  - Software becomes more complex as it evolves
  - Software evolutions follows regular patterns
- Good practice plans for evolution
  - Release management
  - Controlled requirements change process
- Traceability needed to recover knowledge
  - Backwards to originating stakeholders
  - Forwards into design and implementation
  - Still many questions traceability won't answer