Basic Concepts and Preliminaries

Idaho State University | Computer Science

Isaac Griffith

CS 4423 and CS 5523
Department of Computer Science
Idaho State University

ROAR


software evolution
refactoring
meta-programming
model inconsistency
declarative
supporting
intentional
code
program
model-driven
software
object-oriented
evolution
software engineering lab
support
framework
transformation
graph transformation
umons
consistency maintenance
inconsistency management
model transformation
formal
approach
tool
challenge
design pattern
graph-based
description logic
detection
unanticipated
model refactoring
taxonomy
uml model
dependency
reuse
uml
architectural
future
formalism
study
concept
analyse
high-level
detect
classification
comparison
view
application
state-of-the-art
using
analysis
case
behaviour
implementation
uniform
interlaced
fine-grained
detecting
simulation
generate
software development
trend
solution
large-scale
conflict
language-independent
semantics
through
analysing
system
software life-cycle
engineering
concept analysis

ROAR

Idaho State University
Computer Science

Outcomes

After today's lecture you will:

- Have an understanding of the course syllabus
- Have an understanding of the basic concepts surrounding maintenance and evolution

ROAR

Syllabus Review

CS 4423/5523

ROAR

Introduction to SME

CS 4423/5523

ROAR

Evolution vs. Maintenance

- The terms **evolution** and **maintenance** are used interchangeably.
- However there is a semantic difference.
- According to Lowell Jay Arthur:
  - "**Software maintenance** means to preserve from failure or decline"
  - "**Software evolution** means a continuous change from lesser, simpler, or worst state to a higher or better state."
- According to Keith H. Bennett and Lie Xu:
  - "**maintenance** for all post-delivery support and **evolution** to those driven by changes in requirements."

Evolution vs. Maintenance

- **Maintenance** is considered to be a set of planned activities where as **evolution** concerns whatever happens to a system over time.
- Mehdi Jazayer's view on software evolution:
  - "Over time what evolves is not the software but our knowledge about a particular type of software."

Software Evolution

- In 1965, Mark Halpern used the term **evolution** to define the dynamic growth of software.
- The term evolution in relation to application systems took gradually in the 1970s.
- Lehman and his collaborators are generally credited with pioneering the field of software evolution.
- Lehman formulated a set of observations that he called **laws of evolution**.
  - These laws are the results of studies of the evolution of large-scale proprietary or closed source systems (CSS).
  - The laws concern E-type systems:
    - "Monolithic systems produced by a team within an organization that solve a real world problem and have human users."

Lehman's Laws

1. **Continuing Change** – A system will become progressively less satisfying to its user over time, unless it is continually adapted to meet new needs.
2. **Increasing Complexity** – A system will become progressively more complex, unless work is done to explicitly reduce the complexity.
3. **Self-regulation** – The process of software evolution is self regulating with respect to the distributions of the products and process artifacts that are produced.
4. **Conservation of Organizational Stability** – The average effective global activity rate on an evolving system does not change over time, that is the average amount of work that goes into each release is about the same.

Lehman's Laws

5. **Conservation of Familiarity** – The amount of new content in each successive release of a system tends to stay constant or decrease over time.
6. **Continuing Growth** – The amount of functionality in a system will increase over time, in order to please its users.
7. **Declining Quality** – A system will be perceived as losing quality over time, unless its design is carefully maintained and adapted to new operational constraints.
8. **Feedback System** – Successfully evolving a software system requires recognition that the development process is a multi-loop, multi-agent, multi-level feedback system.

FOSS System Evolution

- Pirzada (1988)
  - Noted differences between Unix evolution vs systems studied by Lehman
  - Academic vs industrial software development implies different evolutionary patterns.
- Godfrey and Tu (2000)
  - Empirical study of FOSS evolution
  - Growth trends of Linux OS are super-linear (1994-1999)
- Robles et al.
  - Lehman's laws, 3, 4, and 5 are not fitted to large scale FOSS systems

"FOSS is made available to the general public with either relaxed or non-existent intellectual property restrictions. The **free** emphasizes the freedom to modify and redistribute under the terms of the original license while **open** emphasizes the accessibility to the source code."

Software Maintenance

- Software defect removal and quality control are not perfect
  - There will always be defects
  - Therefore, software maintenance is needed.
- Four types of software maintenance (Swanson and ISO/IEC 14764), based on developer intent:
  - **Corrective**
  - **Adaptive**
  - **Perfective**
  - **Preventive**

Software Maintenance

Kitchenham - Maintenance modifications hierarchy based on activity:

- **Activities to make corrections**:
  - Activities taken to correct/reduce discrepancies between expected behavior and the actual behavior of a system
- **Activities to make enhancements**:
  - Activities are performed to implement a change to the system
  - Three types
    - enhancements that change existing requirements,
    - enhancements that add new system requirements, and
    - enhancements that change the implementation but not the requirements

Idaho State University
Computer Science

Software Maintenance

Chapin et al. expanded the typology of Swanson into an evidence-based classification of **12** different types of software maintenance:

- Training
- Evaluate
- Updative
- Preventive
- Adaptive
- Corrective
- Consultive
- Reformative
- Groomative
- Peformance
- Reductive
- Enhancive

ROAR

COTS Maintenance

The major difference between component-based software systems (CBS) and custom-built software systems:

- Skills of system maintenance teams
- Infrastructure and organization
- COTS maintenance cost
- Larger user community
- Modernization
- Split maintenance function
- More complex planning

Evolution Models & Process

- Software maintenance has its own software maintenance life cycle (SMLC) model.
- Three popular SMLC models;
  - Staged model of maintenance and evolution
  - Iterative models
  - Change mini-cycle models

Idaho State University
Computer Science

Software Maintenance Standards

- IEEE and ISO have both addressed software maintenance processes
  - IEEE/EIA 1219 and ISO/IEC 14764 as a part of ISO/IEC 12207 (life cycle process)

- **IEEE/EIA 1219** organizes the maintenance process in seven phases:
  1. problem identification
  2. analysis
  3. design
  4. implementation
  5. system test
  6. acceptance test
  7. delivery

- **ISO/IEC 14764** describes software maintenance as an iterative process. It's activites are:
  1. process implementation
  2. problem and modification analysis
  3. modification implementation
  4. maintenance review/acceptance
  5. migration
  6. retirement

Software Configuration Management

- The discipline of managing and controlling change in the evolution of the software system.
  - Ensures released software is not contaminated by uncontrolled or unapproved changes.
- SCM systems has four different elements:
  - Identification of software configurations
  - Control of software configuration
  - Auditing software configurations
  - Accounting software configuration status

Reengineering

- Reengineering is done to transform an existing “lesser or simpler” system into a new “better” system
- Reengineering is the examination, analysis and restructuring of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form.

Definition (Chikofsky and Cross II)

The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Reengineering

$$Reengineering = ReverseEngineering + \Delta + ForwardEngineering$$

1. **Reverse engineering**: the activity of defining a more abstract, and easier to understand, representation of the system
   - The core of reverse engineering is the process of examination
   - Does not involve changing the software under examination.
2. Δ: captures alteration that is a change to the system.
3. **Forward engineering**: the traditional process of moving from high-level abstraction and logical, implementation-independent designs to the physical implementation of the system.

Legacy Systems

- In general:
  - an old system which is valuable for the company which often developed and owns it.
  - It is the phase out stage of the software evolution model of Rajlich and Bennet.
- Some Definitions:
  - “large software systems that we don’t know how to cope with but that are vital to our organization.” – Bennet
  - “any information system that significantly resists modification and evolution to meet new and constantly changing business requirements.” – Brodie and Stonebraker

Legacy Systems

- Options for managing legacy systems:
  - **Freeze**: The organization decides no further work on the legacy system should be performed.
  - **Outsource**: An organization may decide that supporting software is not core to the business, and may outsource it to a specialist organization offering this service.
  - **Carry on Maintenance**: Despite all the problems of support, the organization decides to carry on maintenance for another period.
  - **Discard and redevelop**: Throw all the software away and redevelop the application once again from scratch.

Legacy Systems

- Options for managing legacy systems:
  - **Wrap**: It is a black-box modernization technique – surround the legacy system with a software layer that hides the unwanted complexity of the existing data, individual programs, application systems, and interfaces with the new interfaces.
  - **Migrate**: Legacy system migration basically moves an existing, operational system to a new platform, retaining the legacy system's functionality and causing minimal disruption to the existing operational business environment as possible.

Impact Analysis

- **The task of estimating the parts of the software that can be affected if a proposed change request is made.**
- Two classes of techniques:
  - **Traceability analysis**
  - **Dependency analysis**

Impact Analysis

- **The task of estimating the parts of the software that can be affected if a proposed change request is made.**
- Two classes of techniques:
  - **Traceability analysis**:
    - Artifacts (requirements, design, code and test cases) related to the feature to be changed are identified.
    - A model of inter-artifacts such that each artifact in one level links to other artifacts is constructed.
    - This model helps to locate a piece of design, code, and test cases that need to be maintained.
  - **Dependency analysis**

Impact Analysis

- **The task of estimating the parts of the software that can be affected if a proposed change request is made.**
- Two classes of techniques:
  - **Traceability analysis**
  - **Dependency analysis**
    - Assess the affects of change on semantic dependencies between program entities.
    - Identifies the syntactic dependencies that signal the presence of semantic dependencies.
    - Techniques used: **call graph analysis** and **dependency graph based analysis**.

Ripple Effect & Change Propagation

- Two additional notions related to impact analysis are very common among practitioners: **Ripple Effect** and **Change Propagation**
- **Ripple Effect**
- **Change Propagation**

Ripple Effect & Change Propagation

- Two additional notions related to impact analysis are very common among practitioners: **Ripple Effect** and **Change Propagation**
- **Ripple Effect**
  - Measures how likely it is that a change to a particular module may cause problems in the rest of the program.
  - Can provide knowledge about the system as a whole through its evolution:
    - How much its complexity has increased or decreased since the previous version?
    - How complex individual parts of a system are in relation to other parts of the system?
    - To look at the effect that a new module has on the complexity of a system as a whole when it is added.
- **Change Propagation**

Ripple Effect & Change Propagation

- Two additional notions related to impact analysis are very common among practitioners: **Ripple Effect** and **Change Propagation**
- **Ripple Effect**
- **Change propagation**
  - Activity ensures that a change made in one component is propagated properly throughout the entire system.

Refactoring

- **Refactoring** is the process of making a change to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
  - It must preserve the “observable behavior” of the software system (through regression)
  - To improve the internal structure of a software system (improve maintainability)
- Improve the internal structure of the code:
  - removes duplicate code
  - simplifies
  - making code easier to understand
  - help to find defects
  - adding flexibility

Program Comprehension

- The task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution and re-engineering purposes.
- Cognitive Models
  - Programmer's mental representation of the program
  - Generating and investigating hypotheses
    - Hypotheses are an incremental way to understand code
    - Hypotheses form based on some understanding of the code
    - Hypotheses are verified by reading code
  - By continuously formulating new hypotheses and verifying them, the programmer understands more and more code and in increasing details.

Program Comprehension

- Strategies to generate relevant hypotheses:
  - bottom up (starting from the code)
  - top down (starting from high-level goal)
  - opportunistic combinations of the two
- Strategies guide two mechanisms: **chunking** and **cross-referencing**
  - Both produce higher-level abstraction structures
    - Chunking creates new, higher level abstraction structures from lower level structures.
    - Cross-referencing means being able to link elements of different abstraction levels
  - Both help build mental models at different levels of abstractions

Software Reuse

- Introduced by Dough McIlroy in his 1968 seminal paper
- Early reuse research developments include:
  - **Program families** (David Parnas)
    - sets of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analyzing individual differences.
  - **Domain analysis** (Jim Neighbors)
    - is an activity of identifying objects and operations of a class of similar systems in a particular problem domain.

Software Reuse

- Software reuse is using existing artifacts or software knowledge during the construction of a new software system
- Casper Jones identified four types of reusable artifacts:
  - **Data reuse**: standardizing data formats
  - **Architectural reuse**: standardizing a set of design and programming conventions for the logical organization of software,
  - **Design reuse**: common business applications
  - **Program reuse**: reusing executable code

Software Reuse

- Software reuse of previously written code is a way to increase
  - software development productivity
    - cost savings during maintenance as a consequence of reuse are nearly twice the corresponding savings during development
  - quality of the software
    - **Reusability** - a property of a software asset that indicates the degree to which it can be reused

Benefits of Reuse

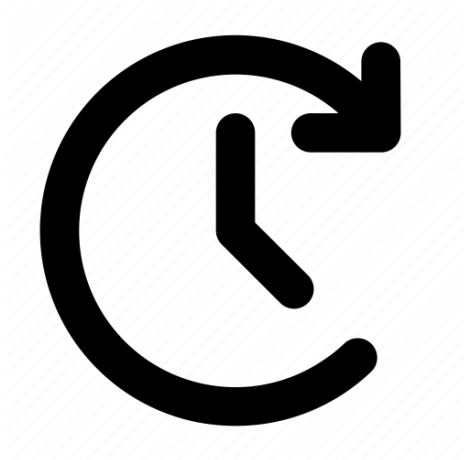
- Increased reliability
- Reduced process risk
- Increased productivity
- Standards compliance
- Accelerated development
- Improved maintainability
- Reduction in maintenance time and effort

Reuse Characteristics

- **Environmental independence** - The components can be reused irrespective of the environment from which they were originally captured.
- **High cohesion** - The components that implement a single operation or set of related operations.
- **Loose coupling** - The components have minimal links to other components.
- **Adaptability** - The components that are adaptable so they can be customized to fit a range of similar situations.
- **Understandability** - The components which are easily understandable so that users can quickly interpret functionality.
- **Reliability** - The components are error-free.
- **Portability** - The components are not restricted in terms of the software or hardware environment they operate in.

For Next Time

- Review EVO Chapter 1
- Review the Syllabus
- Read EVO Chapter 2.1 - 2.3.1
- Watch the Lecture 02



Are there any questions?