Formal Methods in Software Engineering

Engr. Madeha Mushtaq
Department of Computer Science
Iqra National University

From Classic Languages to Formal Methods

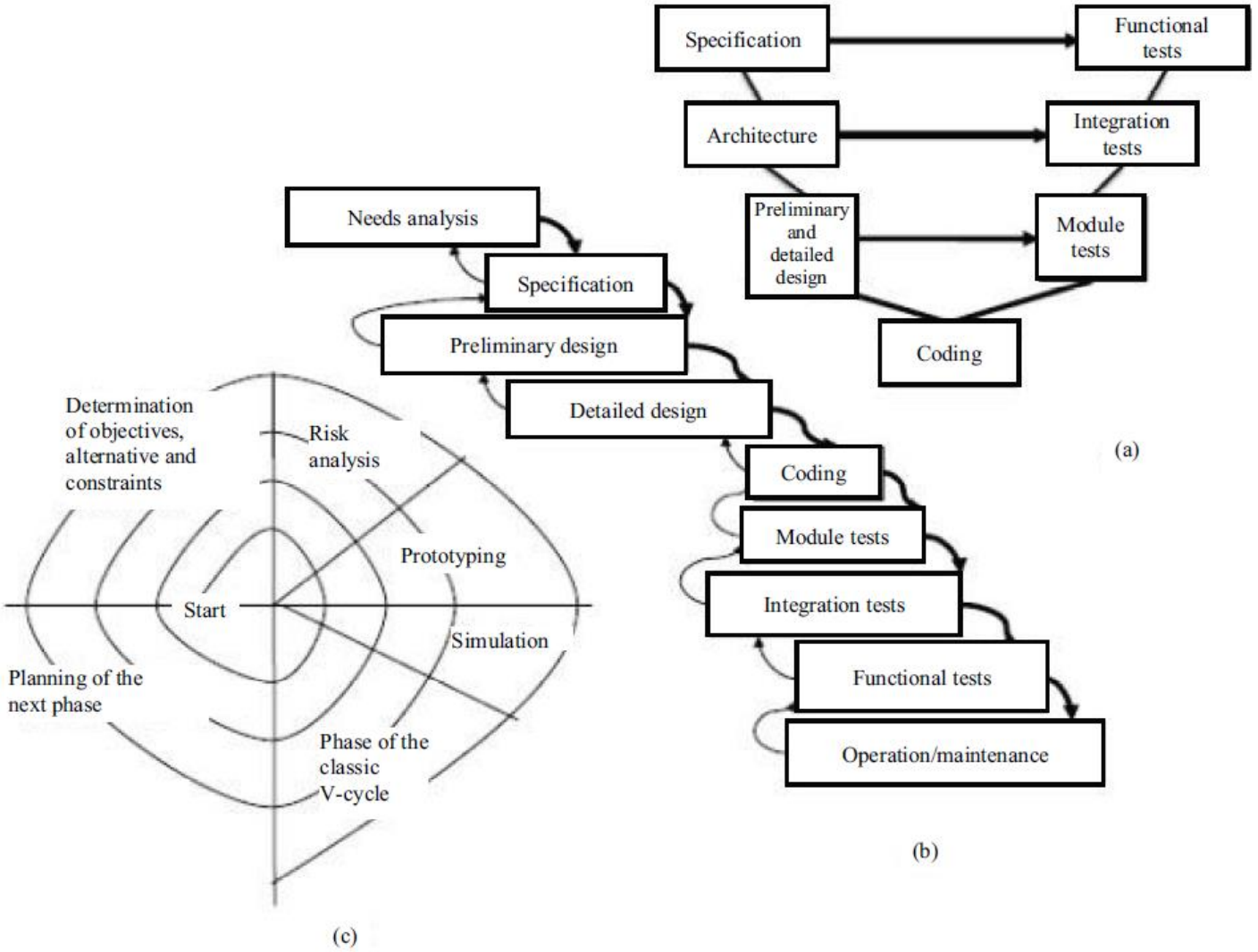
- **Classic development**

- Here we will analyze the classic (meaning non-formal) process used to make a software application.
  - Development process:
    - The creation of a software application is broken down into stages (specification, design, coding, tests, etc.). We refer to it as the lifecycle.

Classic development

- Coding:
  - The classic development process of a software application is based on the use of programming language, for example Ada, C and/or C++.
- Specification and architecture
- Verification and validation (V&V)
- Summary

Classic development



Software Engineering andFM

- Every software engineering methodology is based on a recommended development process proceeding through several phases:
  - Requirements, Specification, Design
  - Coding, Unit Testing
  - Integration and System Testing, Maintenance

- Formal methods can
  - Be a foundation for designing safety critical systems
  - Be a foundation for describing complex systems
  - Provide support for program development

What are Formal Methods?

- Techniques and tools based on mathematics and formal logic
- Approach to develop high-quality systems
- Develop a clear, formal, unambiguous, abstract model
- Ensure model actually reflects requirements.

What are Formal Methods?

- Use formal reasoning to ensure the model has required properties
  - Safety properties: State X never reached
  - Lifeness properties: State X eventually reached
  - Fairness properties: In infinite time, X is reached infinitely often
  - Partial correctness: If system terminates on input X, output is f (X)
- Implement model
  - Use proof/synthesis to ensure correct translation
  - Ensure abstraction does not break (i.e. integer numbers vs. machine words, unlimited memory, exact floating point arithmetic).

FM Vs. Classical Process Steps

- In the Classical approach, **Requirements analysis** may fail to identify all requirements.
  - Formal Methods cannot help much here...

- **Specifications** may be ambiguous and/or not reflect requirements.
  - Formal specifications can be unambiguous
  - Formal specifications can be checked for consistency
  - Formal specifications can be used to deduce (unexpected?) properties.

FM Vs. Classical Process Steps

- Bad **design and Architecture**
  - Mostly a matter of taste and experience. . .

- **Coding** in Classical Approach may fail to implement the specifications
  - Program verification (large subfield of FM) can help.
  - Program synthesis/transformation may generate code directly from specifications.

FM Vs. Classical Process Steps

- **Testing** in Classical process cannot (well, rarely) establish absence of bugs!
  - Only existence of bugs can be detected
  - Again, verification can guarantee the absence of bugs (or at least some classes of bugs).
- **Software maintenance** in Classical Approach often involves large-scale systematic changes
  - Formal methods can assure equivalence of changed code
  - Formal methods can automate changes.

Example: Ambiguous Requirements Analysis/Specification

- Spacecraft developed by two teams of engineers at NASA JPL and Lockheed Martin
- NASA uses metric unit system
- Lockheed Martin uses imperial unit system
- Nobody noticed the discrepancy!

- Result: Mars Climate Orbiter crashed into Mars on September 23rd, 1999
- 125 million US$ lost
- 10 month journey to Mars wasted
- Mission partially compromised (back-up space probes to the rescue...)

Example: Maintenance/Porting Error

- ESA had a successful space program.
- Ariane IV one of the most successful commercial satellite launch platforms.
- Ariane V should be bigger, better, faster version.
- To safe development costs, some code from Ariane IV was reused.
- Development took 10 years and EUR 8 billion anyways.

Example: Maintenance/Porting Error

- In the code, 64 bit floating point representation of speed was cast into 16 bit integer value
- Ok for Ariane IV
- Ariane V's higher speed causes overflow condition
- Correctly trapped by software, but no trap handler (efficiency reasons, "it never happens anyways")
- Hence software crash
- Result: First Ariane V rocket out of control, destroyed by safety mechanism (June 4th, 1996).

Example: Design/Coding Error

- Therac-25 was a combined electron/X-ray medical radiation device (1985-1988)
- Able to produce high energy electron beams, using a beam spreader to control intensity
- Also able to produce 25MeV X-rays (by placing a target and a beam spreader into a much amplified electron beam)
- Used in radiation therapy for cancer patients.

Example: Design/Coding Error

- Software was expected to assure that electron beam intensity, target, and beam spreader can only work in safe conditions.
- However, race conditions in the code allowed unsafe states to be reached:
- High-energy, high intensity electron beam without target/beam spreader
- Race condition never detected in testing!
- Result:
- Several people died of radiation burns, several more injured
- Lawsuits settled out of court.

Limitations of using FM

- Basic software development is cheap
  - Minimal investment: One PC, one copy of Linux/gcc
  - Production cost for a new (corrected?) version is negligible (a few minutes of compilation)
  - Distribution cost is cheap (ship a CD/free download).
- Formal methods add massive overhead.
- Needs very complex Hardware and Software.
- Training for formal methods is expensive/time consuming.
- Reasoning about software is harder, but: Debugging software is easier.

Benefits of Formal Specifications

- Higher level of rigor leads to better problem understanding
- Defects are uncovered that would be missed using traditional specification methods
- Allows earlier defect identification
- Formal specification language semantics allow checks for self-consistency.
- Enables the use of formal proofs to establish fundamental system properties and invariants.

End of Slides