

Week 4,5,6

Mr. Usman Ghani
Lecturer Computer Science



Agenda

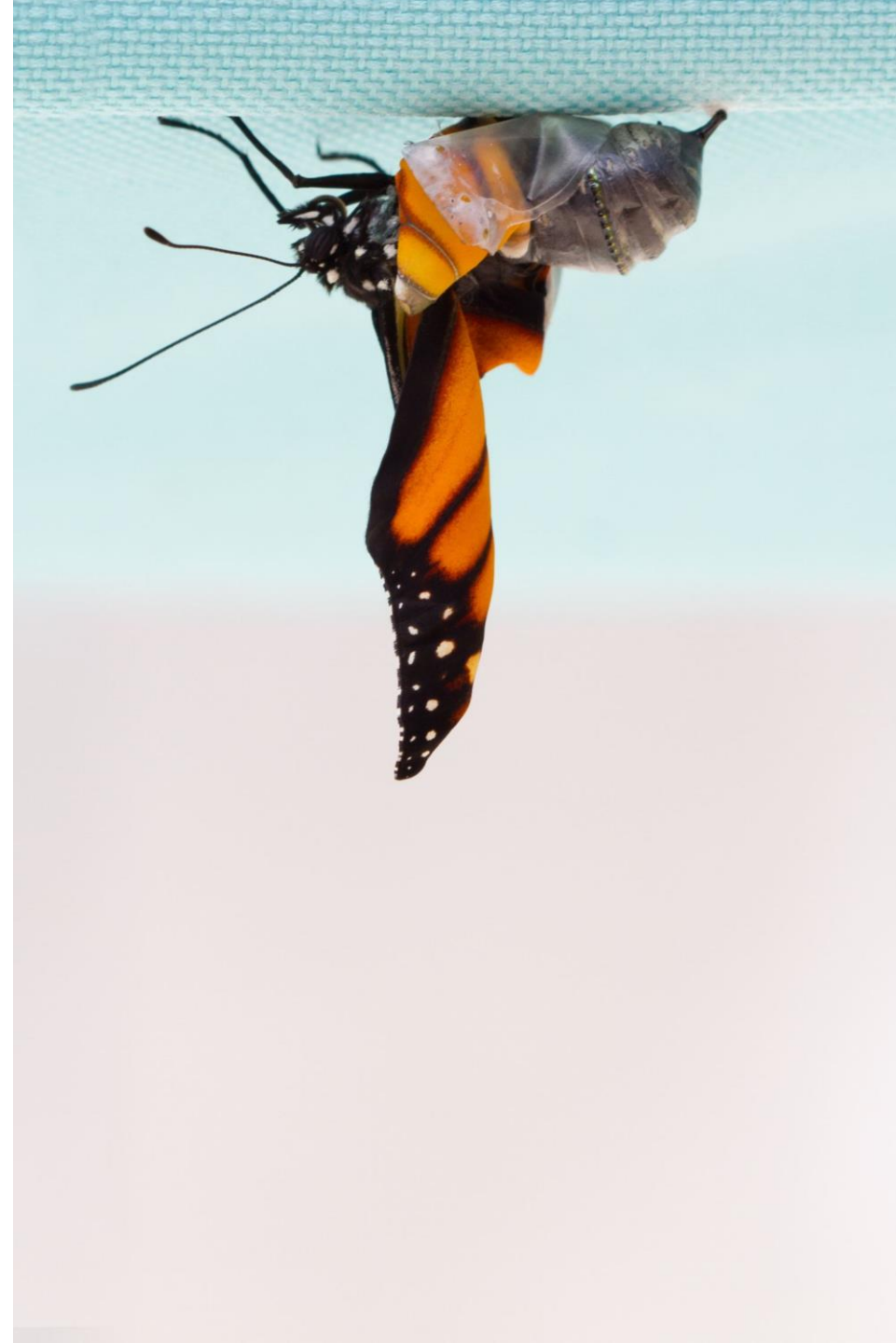
- Software Evolution Process
- Program Types
- Laws of SW evolution
- implications of the laws
- evolution of open-source software
- SW models, top down and bottom-up SW evolution
- Using the dual approach



- Requirement-driven SW evolution
- SW evaluation techniques
- Unified process, Architecture & synthesis process
- Scenario based, design patterns
- Challenges
- Classification of challenges
- Preserving SW quality,
- SW re documentation
- Renovation
- Technologies & architectures
- Measurable benefits

Software Evolution

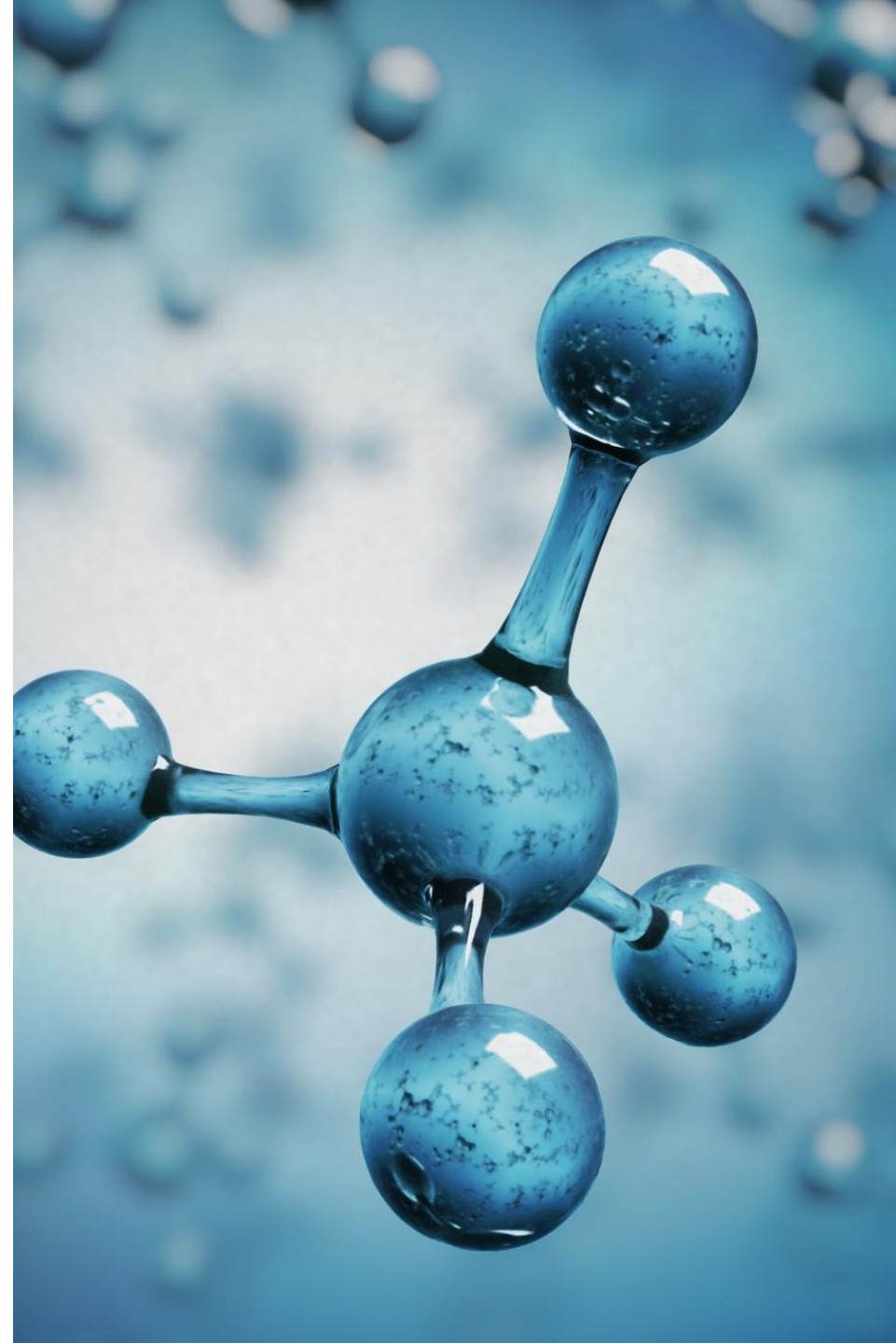
- Evolution is a process of progressive change and cyclic adaptation over time in terms of the attributes, behavioral properties and relational configuration of some material, abstract, natural or artificial entity or system.



Evolution Models and Theories

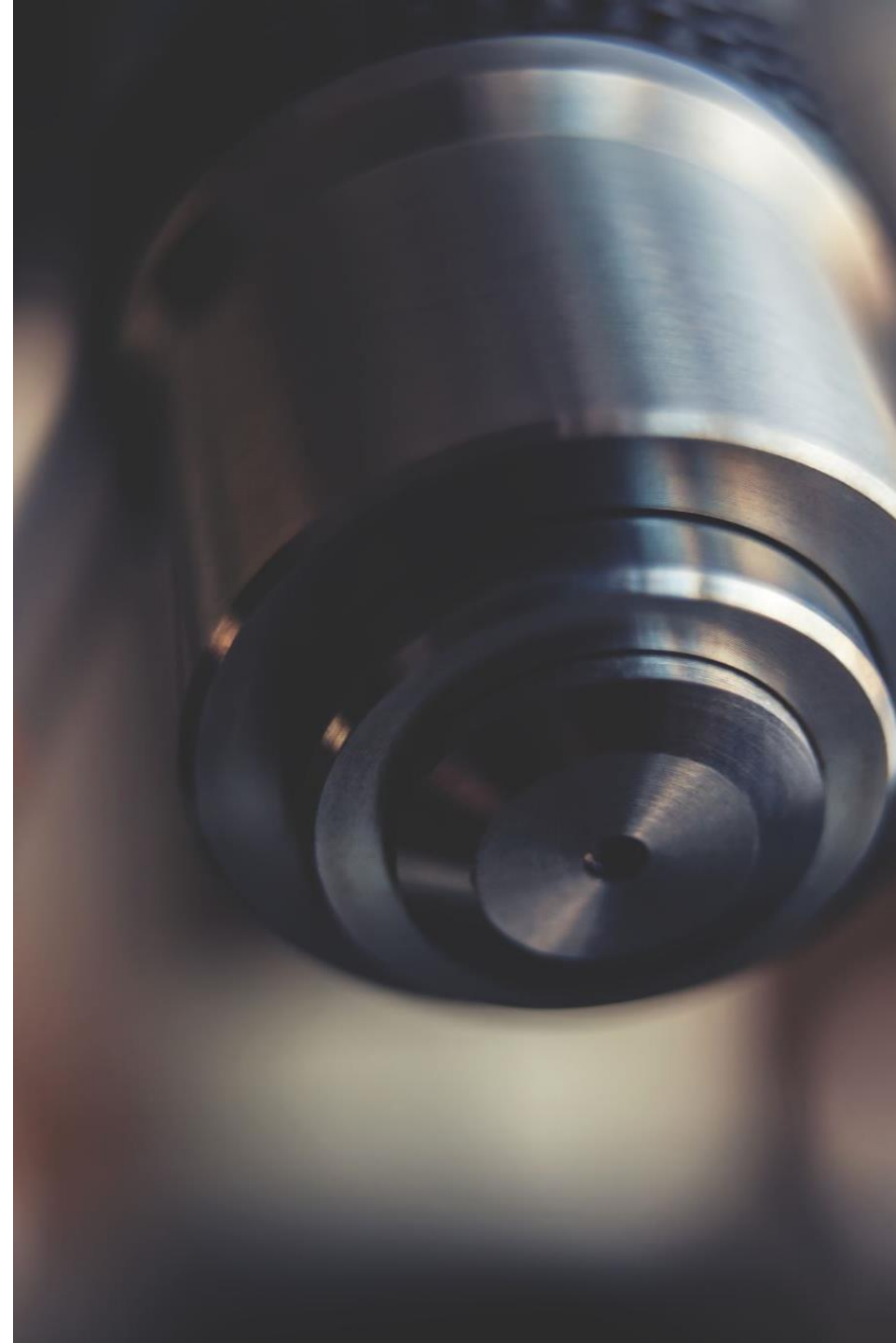
The evolution of technology through technological regimes that depend on:

- Product features
- Development processes
- Infrastructure
- Productive units



Empirical Studies of Software Evolution

- It is necessary to identify and describe what empirical studies of software evolution have been reported.



Studies of the Laws of Software Evolution

The most prominent studies of software evolution have been directed by M.M. Lehman and colleagues over 30 years period.

1. two operating systems—IBM OS 360, ICL VME Kernel
2. one financial system—Logica FW
3. two versions of a large real-time telecommunications system
4. one defense system—Matra BAE Dynamics.



The Software Evolution Process

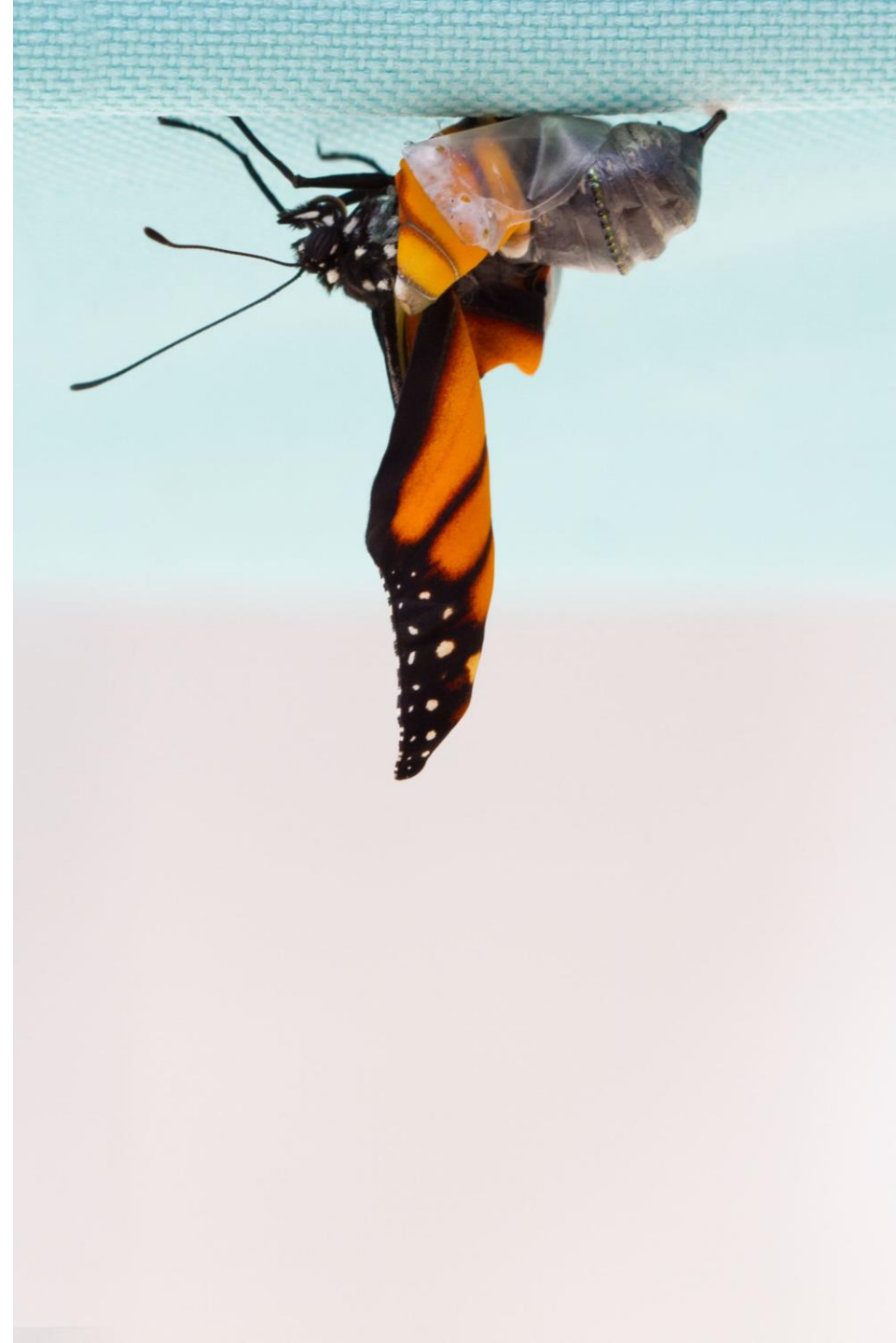


- Software evolution is the initial development process of a software product.
- Software evolution of a software system is a continuous change from a lesser, simpler or worse state to a higher or better state.

The Software Evolution Process (Conti..)

The evolution process includes the fundamental activities of:

- change analysis;
- release planning;
- system implementation;
- releasing a system to customers.



The Software Evolution Process (Conti..)

Software evolution processes depend on

- the type of software being maintained;
- the development processes used;
- the skills and experience of the people involved;
- proposals for change which are the real driver for system evolution.

The Software Evolution Process (Conti..)

Change identification and evolution continue throughout the system lifetime.

- if a serious system fault has to be repaired;
- if changes to the system's environment (e.g., an OS upgrade) have unexpected effects;
- if there are business changes that require a very rapid response (e.g., the immediate release of a competing product)

The Software Evolution Process(Conti..)



Using an emergency repair process can result in the following dangers

- Software becomes inconsistent;
- Changes are not reflected in documentation;
- Software ageing is accelerated by workaround solutions.

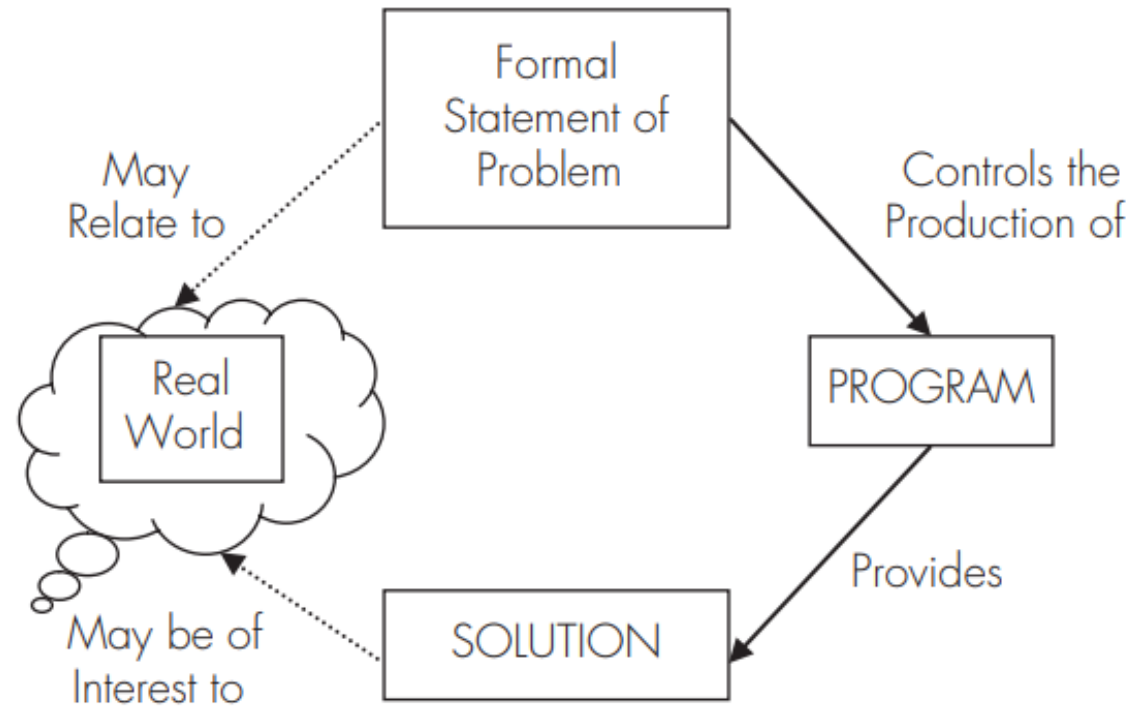
Program Types



- S-type Programs (“Specifiable”)
- E-type Programs (“Embedded”)
- P-type Programs (“Problem-solving”)

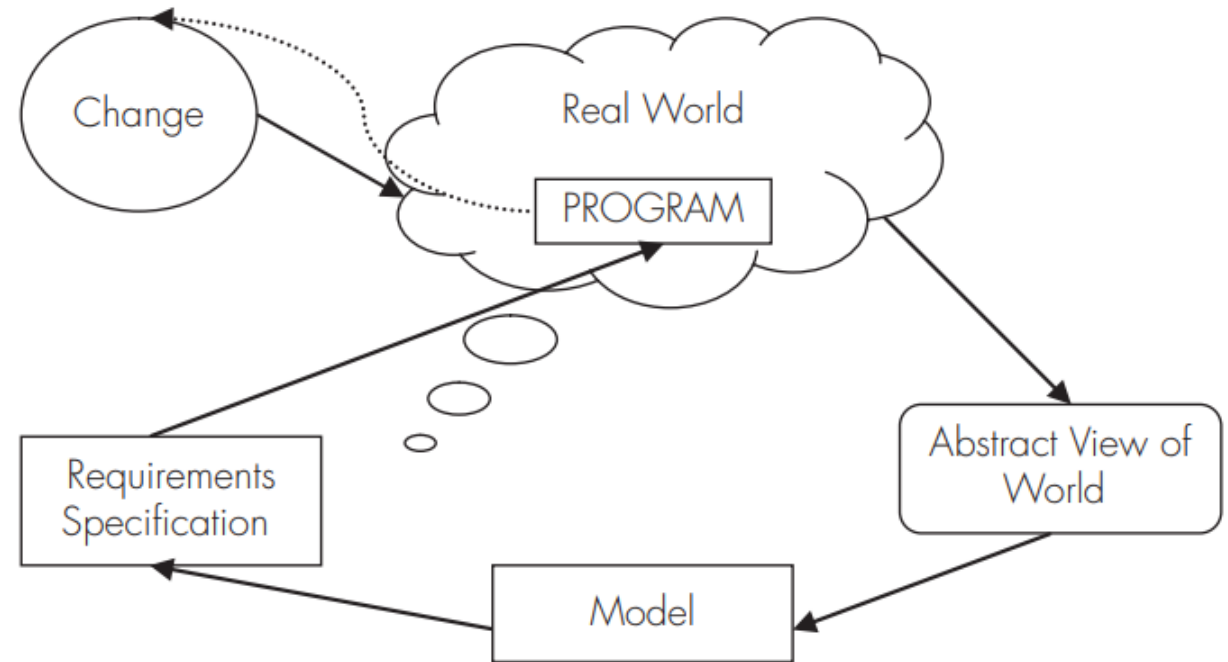
S-type Programs

- This type of software does not evolve.
- A change to the specification defines a new problem hence a new program



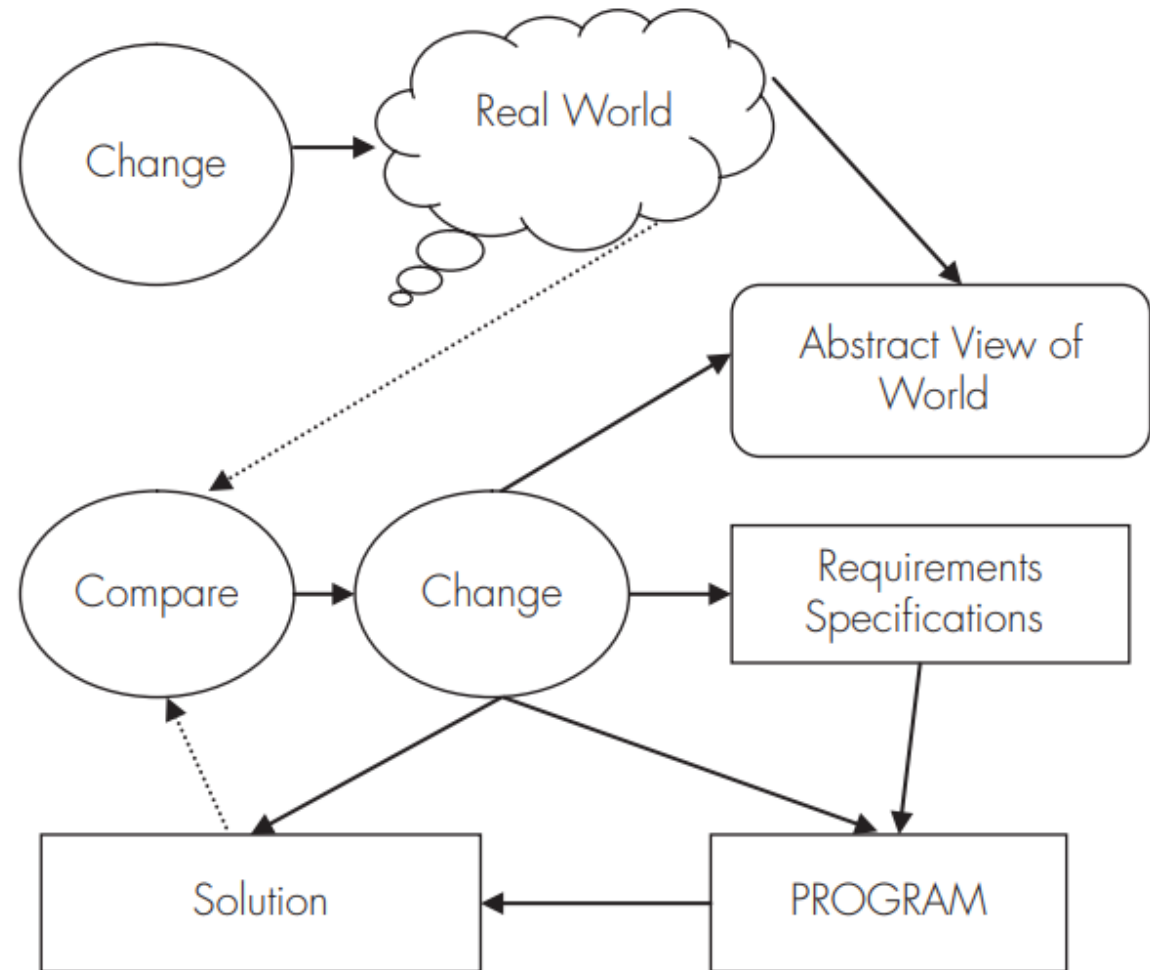
E-type Programs ("Embedded")

- This software is inherently evolutionary.
- Changes in the software and the world affect each other.



P-type Programs

- Imprecise statement of a real world problem.
- This software is likely to evolve continuously because the solution is never perfect.
- Can be improved because the real world changes



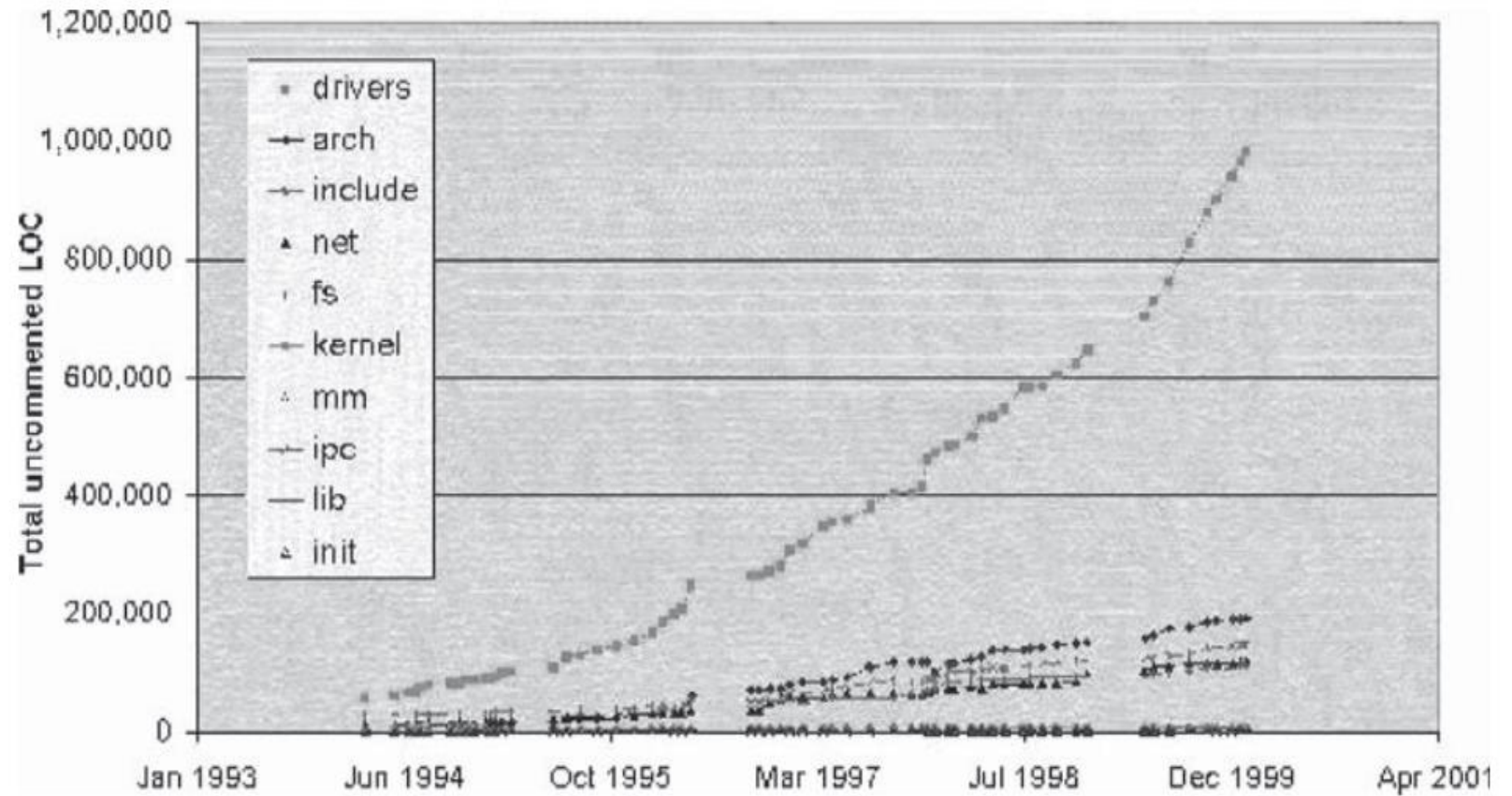


The Laws of Software Evolution

Table 2.1 Current statement of the laws

No.	Brief Name	Law
I 1974	Continuing Change	E-type systems must be continually adapted else they become progressively less satisfactory in use.
II 1974	Increasing Complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it.
III 1974	Self-Regulation	Global E-type system evolution processes are self-regulating.
IV 1978	Conservation of Organisational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime
V 1978	Conservation of Familiarity	In general, the incremental growth and long-term growth rate of E-type systems tend to decline.
VI 1991	Continuing Growth	The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.
VII 1996	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Patterns in Open-Source Software Evolution Studies



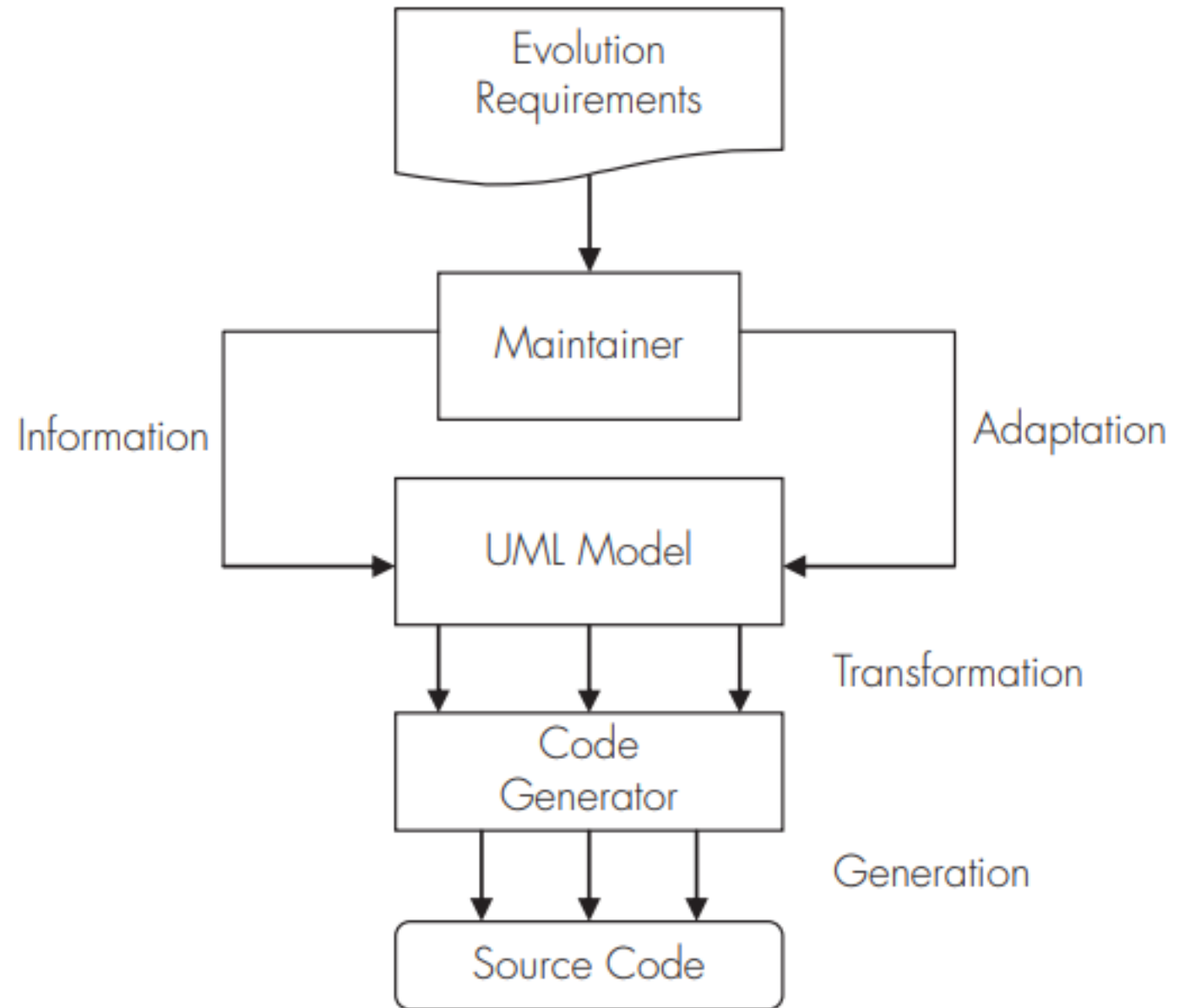
Source: Godfrey and Tu (2000)

Software Models

- Earlier, models were made up of tree diagrams and flow diagrams.
- Later, models consisted of entity/ relationship diagrams, sequence diagrams and state transition diagrams.
- The Unified Modeling Language (UML) attempts to unify all the previous diagram types plus some new ones into a common, all-encompassing set of diagrams intended to describe a software system.
- Diagrams or pictures are not the only way to describe a system. One can argue that the program code is also a description of the system.

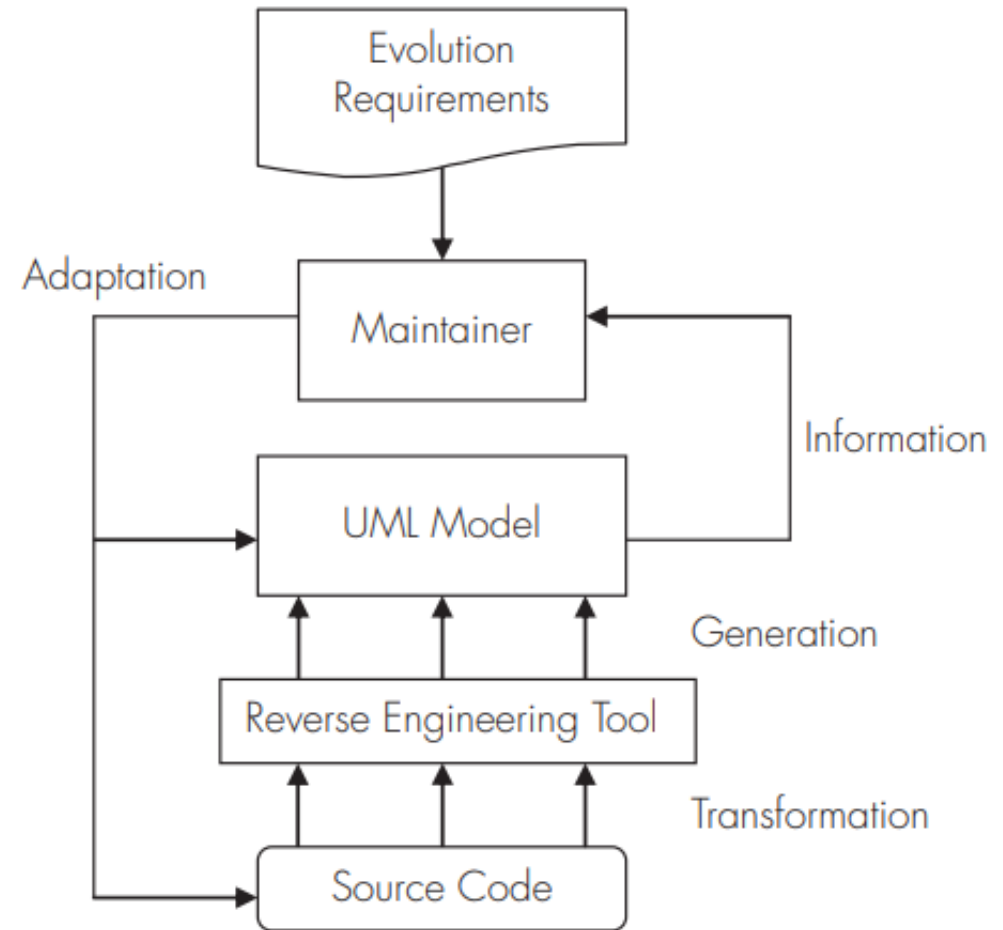
Top-down Software Evolution

- The problem with software evolution is how to keep the description of the system synchronized with the system itself.
- model is changed and that the changes are automatically propagated to the real code.



Bottom-up model driven approach

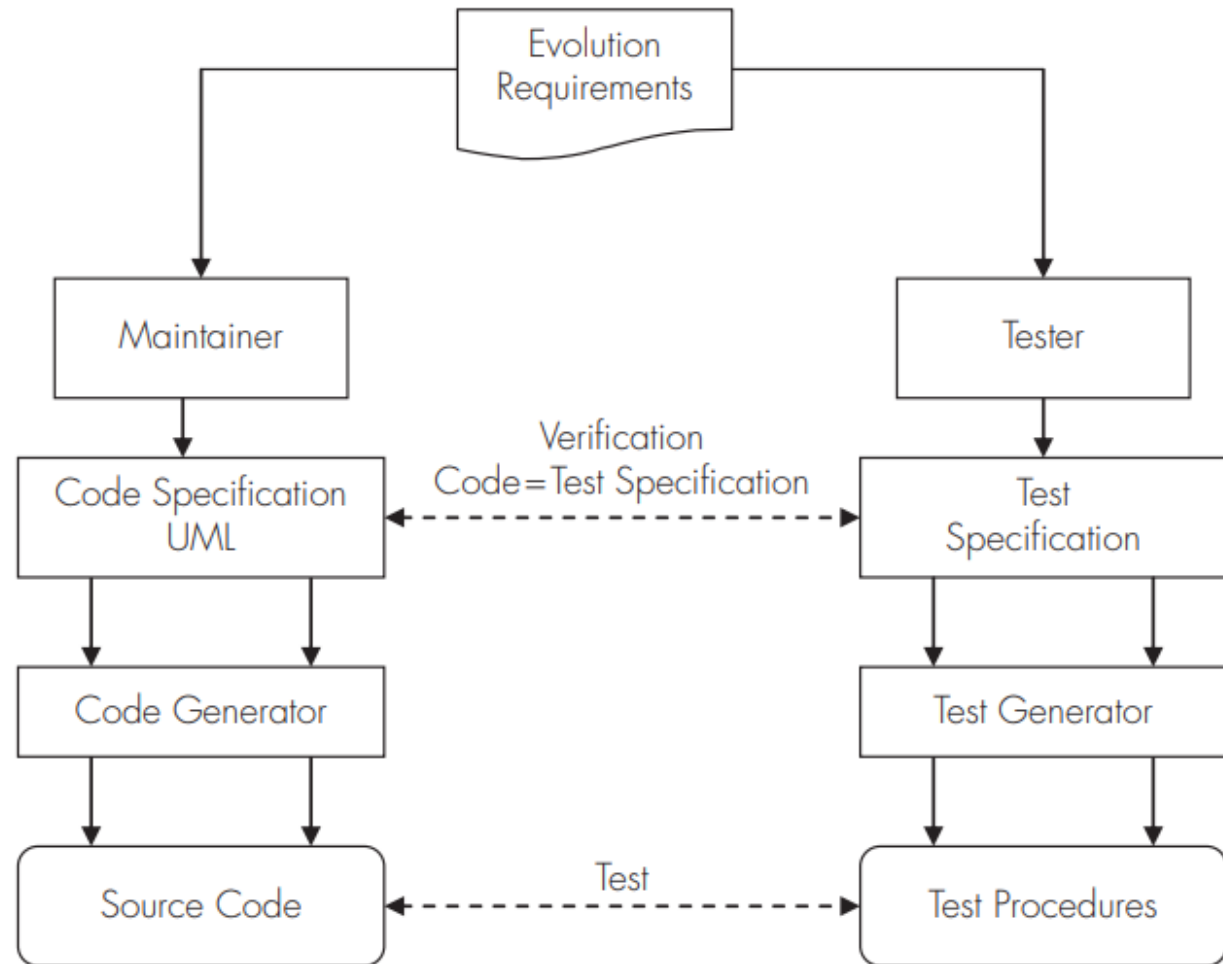
- The bottom-up approach is more contrary. The changes are made to the low-level description of the system i.e., to the code itself, and are then propagated by means of reverse engineering techniques to the upper-level description.



The Need for a Dual Approach

- Both approaches are based on a single description of the software system.
- To verify any given system i.e., to prove that a system is true, one needs at least two descriptions of that system, which are independent of one another.
- Testing implies comparing. A system is tested by testing the actual behavior with the specified behavior.
- If the code is derived from the specification, the code is only that same specification in another notation.
- The test of the system is then, in fact, only a test of the transformation process

Dual Approach



Using the Requirements as a Model

- a table of user interfaces
- a table of system interfaces
- a table of business objects
- a table of business rules
- a table of business processes
- a table of system actors
- a table of system resources and
- a table of use cases.

Requirement-driven Software Evolution

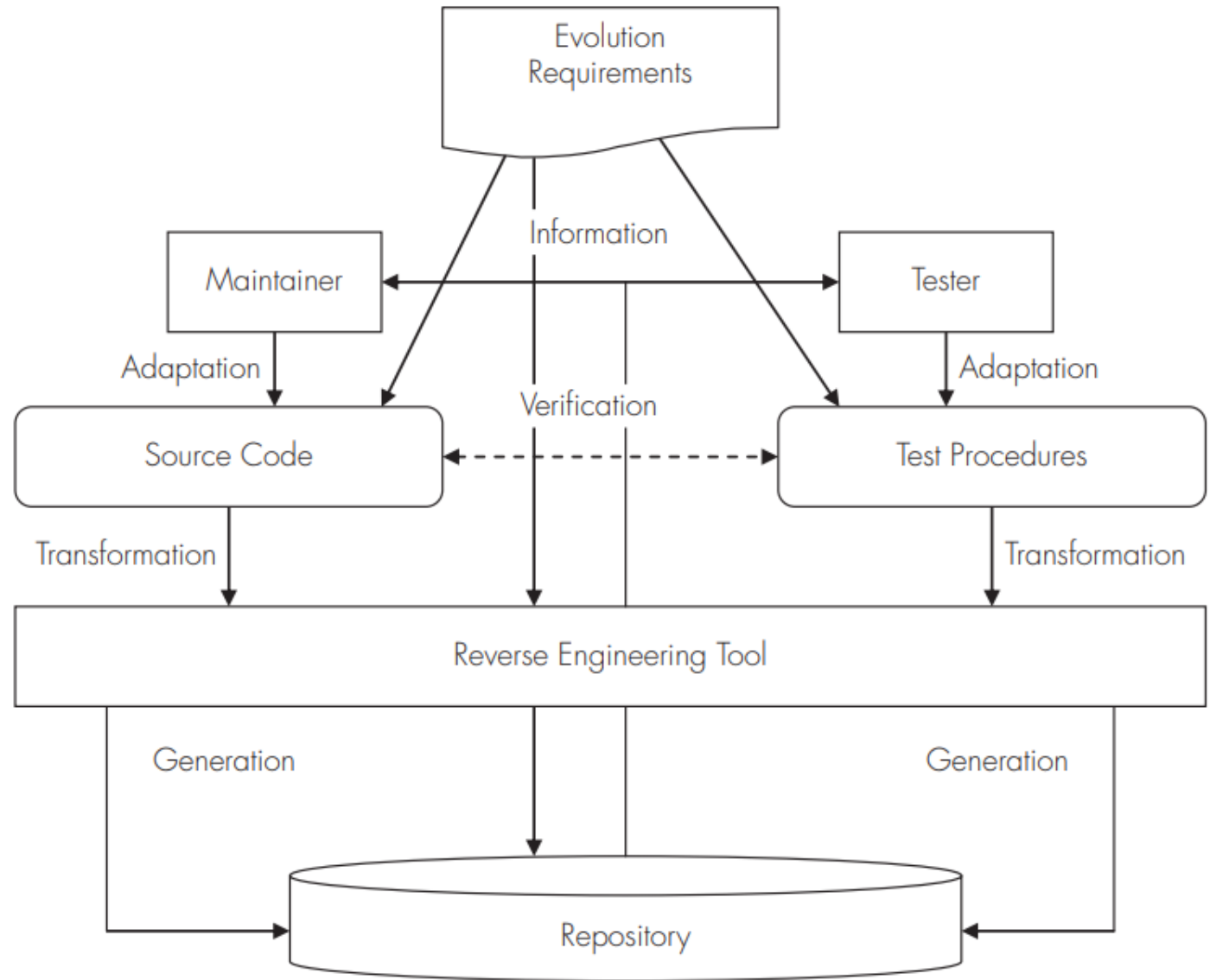
In the requirement model, three descriptions of the system exist, namely:

1. The requirements document.
2. The source code.
3. The test procedures.

Requirement-driven Software Evolution

- For every change or enhancement
- both the source code and the test procedures must be adapted.
- two different people with two different perspectives
 1. The Programmer
 2. The Tester

Requirement-driven Software Evolution





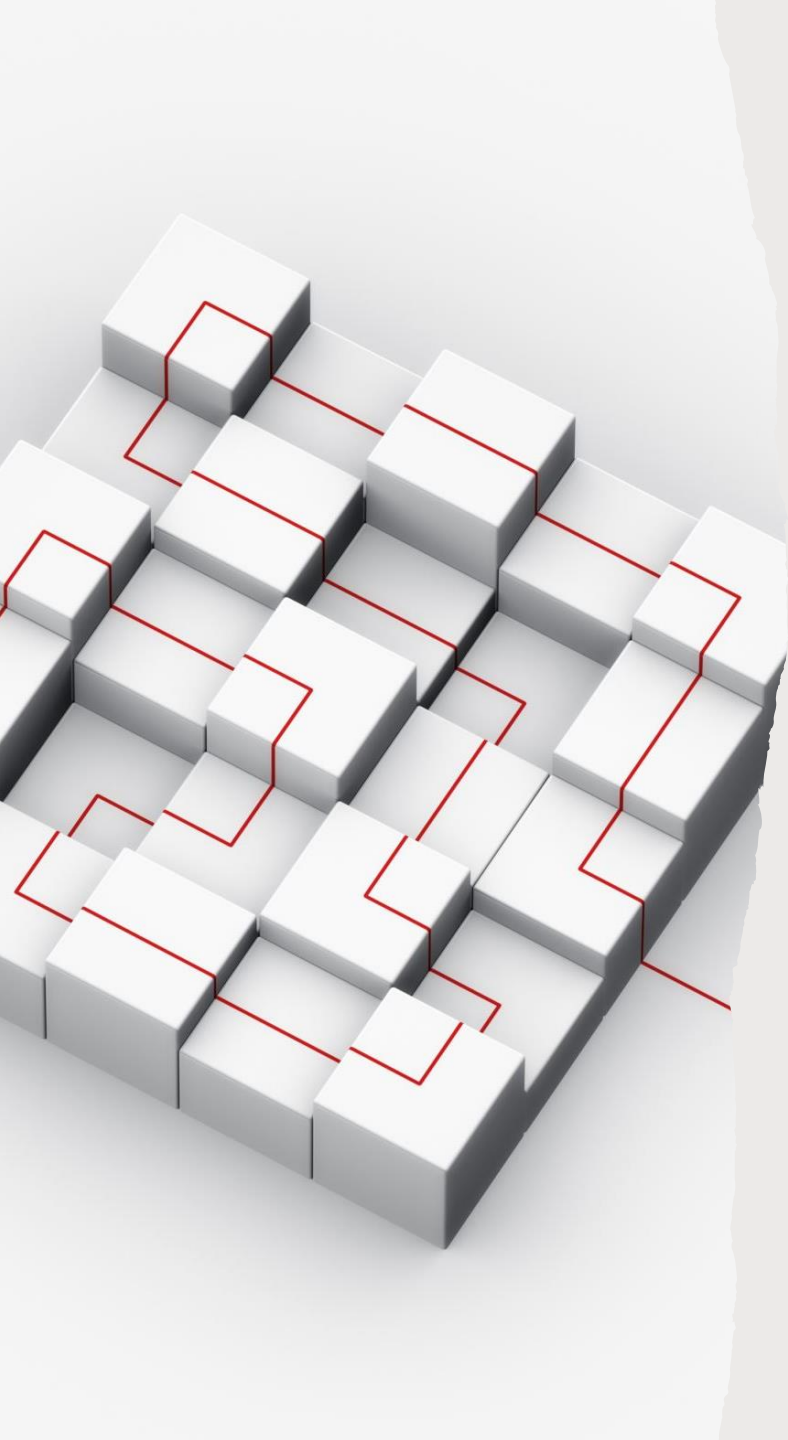
The Future of Software Evolution

- The future of research in software evolution must include the technological regime of F/OSS as a major element. This will be an increasingly practical choice for the empirical study of:
- Individual systems
- Groups of systems of a common type
- Larger regional or global populations of systems



Software Evolution in Software Design and Software Evaluation Techniques

- Understand well-known design and evaluation techniques relating to software evolution
- Understand the challenges in software evolution
- Classify the software evolution challenges
- Enumerate the software evolution challenges




1. The Unified Process

- The Unified Process is a use-case driven, iterative and architecture centric software design process.
- The life of a system is composed of cycles and each cycle concludes with a product.
- Each cycle is divided into four distinct phases.
 1. The inception phase
 2. Elaboration phase
 3. Construction
 4. Transition



2. Software Architecture and Synthesis Process

The Software Architecture and Synthesis process (Synbad) is an analysis and a synthesis process, which is widely used in problem solving in many different engineering disciplines. The process includes explicit steps that involve searching solutions for technical problems in solution domains. These domains contain solutions to previously solved, well established, similar problems. The selection of which solution to use from the solution domain is done by evaluating each solution according to a quality criteria. The method consists of two parts, which are solution definition and solution control.

The image shows several sheets of architectural blueprints, likely for a building, with various dimensions and structural lines. The blueprints are rolled up and layered, with some sheets showing floor plans and others showing cross-sections or detailed views of specific parts of the building. The dimensions are written in black ink, and the lines are thin and precise. The background is a light, neutral color.

3. Scenario-based Evaluation Techniques

- There are many scenario-based techniques that evaluate software architectures with respect to certain quality attributes. Scenario-based Architecture Analysis Method (SAAM), is a method for understanding the properties of a system's architecture other than its functional requirements. The inputs to SAAM are the requirements, the problem description and the architecture description of a system. The first step of SAAM is scenario creation and software architecture description.

4. Design Pattern and Styles

- Some design patterns, make it easier to add new behavior to the system. In their study of comparing design patterns to find simpler solution for maintenance.
- Due to new requirements design patterns should be used, unless there is an important reason to choose the simpler solution, because of the flexibility they provide.
- Identifying these contexts and then selecting the mechanism to use may greatly ease the procedure for the evolution of the software.

Challenges in Software Evolution

- IT society increasingly relies on software at all levels.
- All sectors of society, including government, industry, transportation, commerce, manufacturing and the private sector.
- The productivity of software organizations and software quality generally continue to fall short of expectations, and software systems continue to suffer from symptoms of aging as they are adapted to changing requirements.
- One major reason for this problem is that software maintenance and adaptation is still undervalued in traditional software development processes.

Challenges in Software Evolution(conti..)

- The only way to either overcome or avoid the negative effects of software aging is to place change in the center of the software development process.
- Without explicit and immediate support for change and evolution, software systems become unnecessarily complex and unreliable.
- The negative influence of this situation is rapidly increasing due to technological and business innovations, changes in legislation and continuing internationalization.

Challenges in Software Evolution(conti..)

Beyond a restricted focus on software development and provide better and more support for software adaptation and evolution. Such support must be addressed at multiple levels of research and development. It requires

- Basic research on formalisms and theories to analyze, understand, manage and control software change.
- The development of models, languages, tools, methods, techniques and heuristics to provide explicit support for software change.
- More real-world validation and case studies on large, long-lived, and highly complex industrial software systems.

Classification of Challenges



Time horizon



Research target



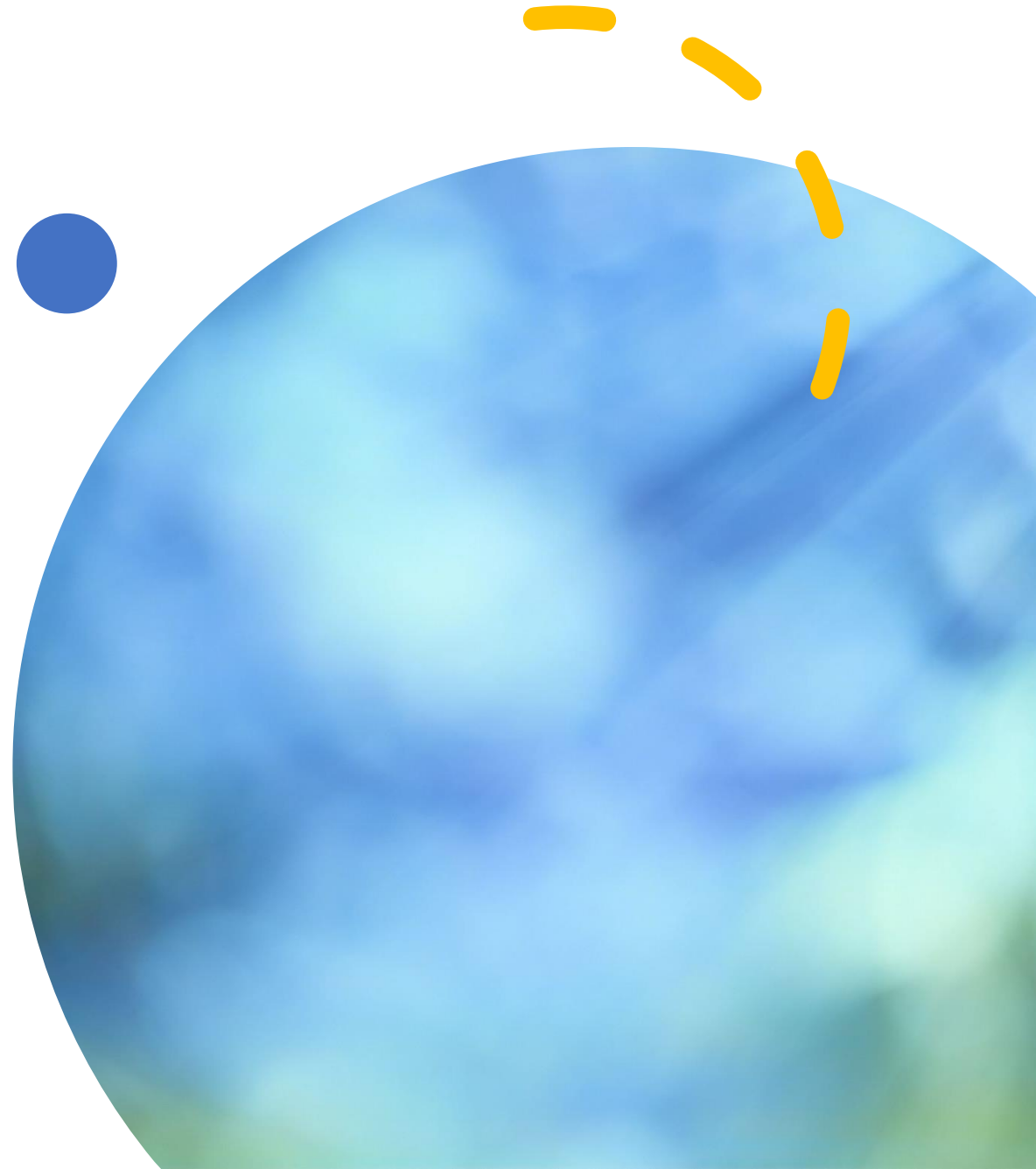
Stakeholders



Type of artifact under study







Type of support needed





Preserving and Improving Software Quality



The phenomenon of software aging, coined by Dave Parnas (1994), and the laws of software evolution postulated by Manny Lehman agree that, without active countermeasures, the quality of a software system gradually degrades as the system evolves. In practice, the reason for this gradual decrease in quality (such as reliability, availability and performance of software systems) is, for a large part, caused by external factors such as economic pressure. The negative effects of software aging can and will have a significant economic and social impact in all sectors of industry. Therefore, it is crucial to develop tools and techniques to reverse or avoid the intrinsic problems of software aging. Hence, the challenge is to provide tools and techniques that preserve or even improve the quality characteristics of a software system, whatever its size and complexity.

The Software Maintenance Process

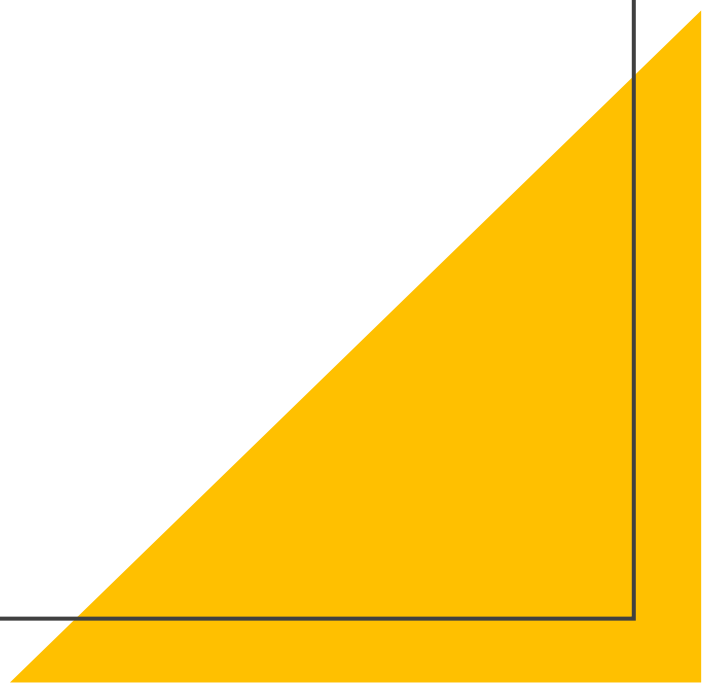
To implement a software maintenance process requires that two key activities be fully mastered regarding software engineering:

1. Software configurations management
2. Change management.

To stress another activity which is relatively little used:
measurement.

A large yellow right-angled triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

The Software Maintenance Process

- Apart from elementary indicators such as days/man-day, or number of Lines of Code (LoC) modified per person, it is important to measure:
 - the technical quality of each component
 - the economical value of each component
 - the evolution of metrics over a given period
- 
- A large yellow triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

Software Redocumentation

- Weak link in the set of deliverables.
- 95% of software documentation is obsolete and or not complete.
- 89% of the cases, developers are obliged to use the source code as the main source for documentation.
- The greater part of the maintenance, trying to understand it rather than modifying the code.

Software Redocumentation

It includes all the information required for the maintenance of the application, notably:

1. How the application is organized (who is the owner, what is the functional or structural decomposition)
2. Exactly, which components are involved (JCL, programs, files and data bases, transactions and so on)
3. What are the relations within a component and between components
4. The Data Dictionary

Software Redocumentation

This documentation should also be up to date, available to all, and easy to use, which implies:

- Processes for automatic update and shared access
- Adaptiveness to the user's cognitive model
- Easy navigation within a component and between components
- A familiar usage metaphor
- A dynamic capacity for impact analysis intra- and inter-components

Software Redocumentation

The redocumentation process generally consists of three main phases which include seven activities. The phases can, thus, be highlighted as:


1. Preparation phase: Analyze the state of the software and its documentation.
2. Planning phase: Decide which parts of the system should be redocumented first and which the general approach.
3. Redocumentation phase: Recreate the various documents that constitute the core of the redocumentation process.





1. Preparation phase


There are two activities in the preparation phase:

1. **System inventory:** The goal of the first activity is to get an idea of the size of the problem and provide the basic information needed in the following activities. It answers questions like:
 - What exactly make up the system?
 - What exactly do we know about that system?
 - Where exactly can we find this information?
- 




1. Preparation phase

2. System assessment: The second step consists in assessing the level of confidence one can have in the code, the documentation and the other sources of information. This is useful to plan the redocumentation in itself and the maintenance in general





2. Planning phase

- **Redocumentation planning:** This activity is the prelude to the redocumentation work. It consists in defining how the redocumentation will be performed and what the priorities are. The planning will be based on results of the preceding phase. Other important points to consider are the maintenance load expectancy and the strategic evaluation of the importance of each part of the system
- 



3. Redocumentation Phase

There are four activities in the redocumentation phase:

1. High level view definition
2. Cross references extraction
3. Subsystems definition
4. Low level documentation



Software Renovation

When an application becomes too complex to maintain, the management faces two options:

1. Re-write the application
2. Replace that application with another or with a software package

Renovation covers redocumentation, restructuration, restoration and reengineering techniques.



Technologies and Architectures

Redocumentation and renovation rest on various technologies:

- Analysis of source code
- Manipulation of source code
- Information diffusion and storage details.

Measurable Benefits

The solutions which have been recommended in this section have been implemented in real cases that have provided valuable information

- Compared to a traditional documentation approach, using an automatic hypertext redocumentation and navigation tool can reduce research time from 5 hours to 1 hour 30 minutes
- The improvement of the maintenance and of the documentation processes, and the restructuration of the application codes can be allowed to process 9% more of requests in 18% less of man-days.

A close-up photograph of a chessboard with several dark wooden pieces standing upright. In the foreground, a light-colored piece lies on its side. The text "The End" is centered over the board.

The End