

Software Evolution

Level 1	1
1.1 Definition	1
1.2 The Laws	2
1.3 Challenges	2
Level 2	3
2.1 Elements of Software Evolution	3
2.2 Emergent Language Design	4
2.3 Coupled Software Evolution	5
2.4 Software Product Portfolios	5
Level 3	6
3.1 Software Quality Models	6
3.2 Technical Debt	7
3.3 Software Metrics	7
3.4 Code Review	8
Level 4	8
4.1 Legacy Languages	8
4.2 Refactoring	9
4.3 Software renovation	9
Level 5	10
5.1 Code Cloning Causes & Consequences	10
5.2 Types of Clones	10
5.3 Clone Detection	11
Level 6	11
6.1 Mining Software Repositories (MSR)	11
6.2 MSR for Understanding → Questions	12
6.3 MSR for Prediction	12
Level 7	13

7.1 Search-Based Software Evolution (SBSE)	13
7.2 Formal Verification in Software Evolution	14

Level 1

1.1 Definition

Evolution is studying software that already exists.

70% of software is maintenance. 90% of the costs of software are in maintenance. 43% of existing banking systems are built on COBOL. 200B lines of code in COBOL exist. Replacement costs 25\$ per line.

DEF: Maintenance

Modification of a software project after delivery to:

- correct faults (bug fixing)
- improve performance (50%)
- adapt the product to a modified environment
- *prevent future issues*: 4% of all maintenance efforts, because you cannot plan for future issues.

DEF: Legacy Software

A large¹ and complex system that took years to build and even longer to debug. It has to be business critical², old and valuable.

If the software is not business critical, it can be retired or sold to someone else. As software evolves, it becomes harder to maintain.

¹Small software can be rewritten from scratch.

²Bugs cannot be tolerated.

1.2 The Laws

DEF: S-type programs

Specifiable software: once it's implemented and tested it's finished. This type of software does not evolve.

DEF: P-type programs

Problem solving software: it *imperfectly* models a real world task. The acceptance can only be qualitative: "this solution is good enough". They can evolve continuously.

DEF: E-type programs

Embedded software: the solution is a part of the world, acceptance is subjective. Some people may accept it and some may not. They are inherently evolutionary.

P-types are usually solved only once. E-types can change because the world they are situated in can change.

1.2.1 Lehman's Laws

1. **Continuing Change**: an e-system **rots**³ unless it's adapted. This process never stops. This is true for P-programs.
2. **Increasing Complexity**: an e-system becomes more complex as it evolves, unless we explicitly do something.
3. **Self Regulation**: an e-system is a self regulating process (SRP). It obeys certain statistical laws: trends and invariances. The distribution is close to normal
4. **The Work Rate is Invariant**: the activity of developers stay the same (conservation of organizational stability).

³Becomes less useful

5. **Invariant Perceived Complexity**: the number of new features per release is statistically invariant (law of conservation of familiarity). Not overwhelm or bore the user.
6. **Continuing Growth**: the e-system must add new features overtime to keep the users satisfied.
7. **Declining Quality**: the perceived quality of the system declines for both the customers and the developer (unless constantly maintained).
8. **Feedback System**: a developed system evolves through a feedback system. It can be multi-level, multi-loop or multi-agent.

1.3 Challenges

1.3.1 Cyber Physical Systems

(CPS): a view on a software system where you go beyond it just being software. It's a **feedback system** with a lot of agents that don't work on one machine, there are delays. It can be adaptive and predictive. The agents can be intelligent (human or AI). Feedback systems often work in real time, so there is a definition of time, urgency and concern on where data will be at a certain time. **Humans** are also involved. There are ecosystems, communities, individuals. Human behavior is hard to model and to express. The **context** is also in the loop: consider energy consumption, carbon **footprint**, water usage and sustainability driven solutions. The costs must also be considered.

1.3.2 Bit Flips

Especially for software that is running in space.

1.3.3 Consistency

Consistency means having no contradictions.

- Structural consistency: conform to the metamodels
- Behavioral consistency: consistent with expected behavior
- Conditional consistency
- Trace/perfect equivalence

- Inner Consistency: the artefact doesn't contradict itself
- Outer Consistency: the contract between two or more artifacts must not be broken.

Some form of (in)consistency management is required. I.e. compute and sync views on models in model driven software engineering, maintaining GUIs in MVC, syncing data in different formats, updatable views in DBMS, (de)serialization...

1.3.4 Quality

Goal-question-metric. Ask questions to achieve goals. Then on the quantitative level find some piece of data that can be used to find answer.

1.3.5 Adaptability

The ability of a system to adapt itself when it changes. Adaptation is critical for survival of the software and to improve its quality. Types of adaptation:

Parametric tweak parameters over time:

Algorithmic choose the most appropriate algorithm:

Resource reallocate utilized resources.

Adaptable Software Factors:

Extensibility how easy it is to change the *amount* of capabilities;

Flexibility how easy it is to change the *kind* of capabilities;

Tunability how easy is it to *tweak*.

Fixability if we encounter *one* problem, how easy it is to fix it without affecting other things.

Adaptation can be possible **through tolerance** by preserving the original behavior when a new change comes from the environment or **through adjustment** by making unchanged artifacts sync when welcome changes occur.

Level 2

2.1 Elements of Software Evolution

2.1.1 Software Engineering Lifecycle

Phases:

- Requirements
- Design/architecture
- Construction/implementation
- Testing/integration
- Operations/deployment: connecting the stakeholders to the built system.
- Maintenance

No longer phases, due to AGILE development (they happen weekly/biweekly in sprints).

2.1.2 Requirements → Requirements Creep

The most significant risk in SE. Environment changes unexpectedly, so the requirements change. This can make a system useless or allow to pursue new opportunities.

The amount of errors introduced in Requirements & Design grew from 40% in 1984 to 70% in 2002. Rework cost is 300×.

It is common for requirements to change.

Stable/Enduring requirements technical core of business.

Changing requirements

- Mutable: the market has changed, it is possible to improve the system or cutoff unused features.
- Emergent: occur when stakeholders are involved in requirement elicitation.
- Consequential: directly caused by delivery⁴, more features are demanded.

⁴From Lehman's 7th law: the perceived value of the system decreases when no changes are made.

- Adaptive: variation in the tasks, more things to do than expected
- Migration: move from one piece of hardware to another, or from one database to another. Some requirements can be retired after the migration.

2.1.3 Design → Model Management

Models are used to manage complexity→they are used in complex projects→they become complex→they need to change/evolve→they need to be managed.

Types of model management:

- Quality check: slight improvement
- Transform with intent
- Version control: merge changes in one version.

DEF: Model

A model is a simplified representation of a part of the world from a particular view.

A **model change** can be triggered by:

- world changes → the model becomes incorrect;
- when more data becomes available, the model incorrectly represents reality/becomes uncertain.
- the needs for the model change: and it must adapt to fit the new view
- the focus widens, and the model becomes too detailed
- the focus narrows, and the model becomes too abstract
- a bad representation

2.1.4 Construction → Evolution of Code

Refactoring changing the code without changing it's behavior. *Correctness must be guaranteed* by the tool used, testing or proofs.

Restructuring change the behavior, but *maintain some form of structure*.

Reengineering happens on the design level. What design pattern should be used, what should the level of visibility be...?

Rearchitecting can be based on rewriting or reengineering. It means recovering the architectural models while disregarding the implementation.

2.1.5 Testing → Test Code Smells

Badly written tests:

Mystery Guest test is not self contained;

Resource Optimism wrong optimistic assumptions;

Test Run War two test pass independently, but fail when run together;

General Fixture test classes contain broad functionality in their setup;

Eager Test tests multiple things at the same time;

Lazy Test test for things in the same equivalence class;

Assertion Roulette assertion with multiple big conditions;

Indirect Testing mismatch between class and test class;

2.1.6 Operation → Runtime Evolution

- Customization/Personalization of the program while it's running.
- Resource discovery
- Hotswap plugins
- Late binding

2.1.7 Maintenance → Data Migration

Data migration occurs because:

- the database version upgrades
- the existing setup is insufficient
- company policy changes
- cutbacks

2.2 Emergent Language Design

How languages evolve through time.

1. Human computes, machine helps

2. Machine computes, program dictates steps
3. Machine computes, program prescribes tasks
4. Machine runs tasks, program instructs
5. Machine runs tasks, instructions grouped into functions, program calls functions
6. ... functions from a machine, which computes

2.3 Coupled Software Evolution

When two artifacts are consistent, if one of them evolves, the other one **coevolves** (coupled evolution).

2.3.1 Coupled Evolution Examples

Cooperative Editing the changes must sync and be distributed. I.e. shared Google doc.;

Source code and runnables one way consistency: the executable artefact is destroyed and recreated from the source code;

Consistency management in software modelling how to keep the diagrams and models in sync to make them provide a consistent view of the system;

Co-evolution of design & implementation

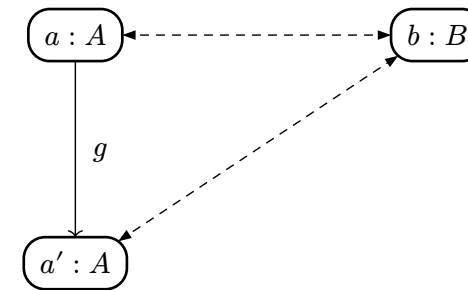
View-update translation only part of a thing is visible for security reasons. It should be possible to propagate the change from the view to the data source.

Structural/projectional editing

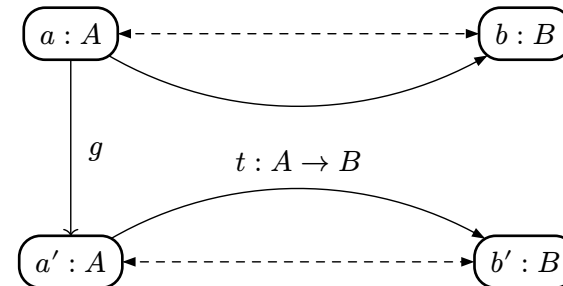
Negotiated program transformations

2.3.2 Patterns

No reconciliation: g is a transformation that doesn't endanger the consistency.



Trivial Reconciliation: b is discarded, b' is generated from scratch.



Symmetric Reconciliation: occurs when a transformation means something conceptually, and the transformation is two functions together.

Asymmetric Reconciliation: a transformation can be reapplied to a newly created diagram.

Evolution is not always triggered from the outside, with coevolution it is triggered within the code.

2.4 Software Product Portfolios

A product family/line is a group of products advertised with the same methods, but have different consumers.

Domain engineering: what are the building blocks that can be used later to put together an application.

Application engineering: make a model out of the requirements, which is conformed to the variability model.

2.4.1 Software Product Line Evolution

1. Specifics:

- Longer Life Span
- Bigger size and complexity
- More interdependencies

2. Assets:

Common present everywhere

Variable configuration dependent

Specific only for a particular product

3. Adding products

2.4.2 Variability Models → Feature Model

DEF: Feature

A feature is a distinguishable characteristic of a concept relevant to some stakeholder(s).

It's usually drawn as a tree with or/xor-groups, requires or excludes.

Domain engineering and application engineering are separated.

Level 3

3.1 Software Quality Models

Code quality can refer to:

Quality in use bugs that manifest;

External quality that can be benchmarked;

Internal quality the code is examined, not ran;

Process quality the process that was followed in order to create the code.
i.e. AGILE.

Process quality models: the way software is handled.

Product quality models: how good a software system is.

3.1.1 SQuaRE (Product)

Software and systems Quality Requirements and Evaluation

Suitability functional requirements: completeness, correctness, appropriateness.

Efficiency time behavior, resource utilization, capacity.

Compatibility co-existence of pieces or systems of software, interoperability.

Usability how easy it is to learn and use the system, how does it look, how accessible is it.

Reliability availability and maturity of the system, it's fault tolerance and recoverability.

Security who is accountable when something bad happens, is the person that is using the software who they say they are

Maintainability the ability to do something with the system.

Portability can you do something at runtime to adapt to changes in the environment.

3.1.2 CMMI (Process)

Capability Maturity Model Integration: level of maturity available in a project or organization.

Level 1 Initial: unpredictable, reactive, poorly controllable.

Level 2 Managed: reactive, project specific.

Level 3 Defined: proactive, organization specific and tailored processes.

Level 4 Measured: measured, controlled, quantitatively managed.

Level 5 Optimizing: improving with experiments.

3.2 Technical Debt

Poor code in the codebase→task based on it takes longer→system doesn't evolve timely→business suffers, company makes less money→less money for developers→developers become overstressed.

Technical debt is a process that happens when you try to take time from your future self.

With self admitted technical debt you admit to borrowing time from someone else. Studies found no correlation with internal quality.

3.2.1 Backlog

The backlog is composed of things you know should be doing very soon or should have done already.

	Visible	Invisible
Positive	Feature	Architecture
Negative	Bug	Debt

Debt makes actions less efficient, if removed no user will notice, only the developers.

Every corner of this table has a different stakeholder:

Developers want to improve the architecture

Managers want to see more features

User complains about bugs

No one is stakeholder of the technical debt, no one wants to tackle the debt.

	Reckless	Careful
Deliberate	<i>We don't have time for design!</i>	<i>We can ship now and clean up later.</i>
Accidental	There is a lack of knowledge or expertise	<i>Now we know what not to do next time⁵</i>

Table 2: Technical Debt Quadrant

	Evolution	Maintainability
Visible	Improve functionality	Improve quality
Invisible	Architecture	Code smells

Table 3: Technical Debt Landscape

3.3 Software Metrics

Measurement is quantitatively expressed reduction of uncertainty based on one or more observations.

Direct quantification measurement;

Indirect quantification calculation.

Without measurements there would be no requirements. All software would be reliable or user friendly. The component (design, code, test) costs would be fused. There would be no business promises⁶.

Metrics are based on abstractions (models).

3.3.1 Evolution of Measurements

With scale you can compare previous results, quantitative not qualitative comparisons. A scale should be physically defined.

⁵Learn from a previous experience.

⁶i.e. how fast or reliable a system is.

Measurement is a **mapping**. Empirical relations must be preserved under the mapping done by the measurement. If A is hotter than B, B can't measure 50°C and A 10°C.

3.3.2 Scales

Scales define what you are allowed to do with your measurements.

Nominal classification with no order⁷.

Ordinal different classes and rankings, ordering is possible but no other operations.

Interval explicit ordering and differences. Addition and subtraction work, but not multiplication.

Ratio ordering, difference, ratios. Can be measured in intervals.

Absolute number of actual occurrences of something in a real world entity, the observed reality.

3.3.3 Maintainability with mixed paradigms

The paradigms are kept apart and interleaved locally. The focus is not on how much mix there is, but how the mixture is arranged.

A metric is how many times the paradigm changes (switch count). Stretch: how long did the code stay in a certain paradigm.

Mixed paradigm fragmentation can be considered a code smell.

3.4 Code Review

Statistically, the best way to fix bugs.

It forces you to share a mental model with another person. It also helps to:

- Catch bugs
- Improve code quality
- Share knowledge
- Enforce standards
- Improve estimates of how hard it is to implement something

⁷i.e. S, P, E systems

- Mentor new engineers
- Build a better team

3.4.1 Types of Code Review

Tool assisted agents & linters.

Instant pair programming.

Synchronous over the shoulder: work on it first, ask for feedback later.

Asynchronous lightweight.

Team review hour of code.

Formal review artefact evaluation.

3.4.2 Elements

- Clear and shared objectives.
- Defined and followed process.
- Constructive and actionable feedback. Descriptive feedback: share your comprehension process.

Level 4

4.1 Legacy Languages

1GL First generation languages, basically machine code;

2GL Assembly languages, 1-1 mapped to machine code but with names to refer to bytes and registers.

3GL Most languages, from Cobol to C++ and Java.

4GL Focus on efficiency, expressivity. 3GL code was used to patch leaky abstractions.

DSL the opposite of a general purpose language (GPL). Abstractions targeted to a problem domain, the opposite to a library. They are 4GL languages.

4.1.1 Advantages of a DSL

1. Domain specific notations and abstractions, to facilitate reuse in a specific solution domain.

2. Tool support.
3. Conciseness, self-documentation
4. Reliability, portability, testability
5. Designed lifespan
6. Configurable per customer

4.2 Refactoring

DEF: Refactoring

- *I'm refactoring my code because...* I'm collapsing the hierarchy, extracting methods, moving code from one field to another..⁸
- Automated code change supported by your IDE
- Changing the internals without changing the externals (behavior)

A refactoring that breaks code isn't real refactoring.

4.2.1 Refactoring Best Practices

1. Remove dead code (code that is never ran)
2. Extract function/method
3. Combine functions into class
4. Replace loop with pipeline (streams, maps)
5. Collapse hierarchy
6. Inline variable
7. Move a field (inappropriate intimacy)
8. Before refactoring, ensure you have a solid suite of tests
9. Since refactoring changes things in small steps, its easy to find where the bug is.
10. First you refactor the program to make it easy to add the feature, then add the feature
11. If code breaks when doing refactoring, then you're not doing refactoring

⁸Stick to one type of refactoring.

12. Refactoring is not limited to code, can also be applied to grammars, data...

Not always IDE related, sometimes language dependent.

4.3 Software renovation

4.3.1 Realities of working with legacy software

- Source code may be lost, or only bytecode is available
- Documentation is not available or outdated
- Old technology was extended *ad hoc*
- Don't know why it works
- Language or notation mixture and interoperability
- The design has been decided beforehand, no choice but to implement it like this
- Painful choices

4.3.2 Legacy System Renovation Paths

Path 1: Rewrite - *Why don't we just rewrite it?*

Question If something took 30 years to build, how long would it take to rebuild it?

Pro Software engineering is more efficient now than 30 years algorithm

Cons you are still biased to the technology you're using, you need to invest in reverse engineering first. Tools to verify the same functionality are required.

Thus It's possible if flexibility is tolerable or in specific project parts.

Path 2: Refactor

Massive refactoring is possible: GO TO elimination, objectification, wrapping.

Pro possible to automate, can use transformation languages

Con you are still using the same tech

Thus often feasible, sometimes satisfactory.

Path 3: Dethrone

Generate a 3GL from a 4GL for the last time, then you refactor it and improve it. This way you remove the 4GL.

Pros full automation, retires a language, represents semantics

Cons per language approach, hard to generalize across multiple 4GLs.

Path 4: Migration

Restructure the original program so that it's ready for a syntax swap. Only then you restructure the target program to use it's features. You change the language while keeping the logic.

Pro sounds perfect

Cons must restructure the original and target, syntax swap must be feasible, native constructs map to simulated ones.

Thus it's feasible under very specific circumstances.

Path 5: Upgrade

Keep the language, but you change the platform/ecosystem where it operates.

Pros no change to the code, executes in a new environment, empowers later change

Cons technological dependencies remain, it's harder to develop solutions.

Thus can build a business.

Path 6: Reinvent

Take the language and re-develop the compiler (cover a language's subset).

Pros code mostly stays the same, get modern development and execution environments, full control over the target environment.

Cons documentation may not exist, substantial effort

Thus perfect for a combination of rich client and expert migrator company

Level 5

5.1 Code Cloning Causes & Consequences

What makes people create duplicate pieces of code?

Usually you understand, adjust and reuse a certain functionality. When you clone a piece of software, you lose the understanding and make the codebase larger. More code $\Rightarrow$ higher maintenance costs. Removing clones involves refactoring efforts. Usually there is 10-20% of code duplication.

Cloning also leads to dead code. They can compete for internal usage and become disconnected. Removal is not always the best option.

Clones must coevolve. If they have bugs, they must be fixed everywhere (and in the same way).

Templated clones are ok. They can be achieved with polymorphism. "Clone and own": copy and then take care of that clone. Good clones, like code patterns (micro patterns), are called idioms.

Customized clones are inevitable, for example when forking source code to implement a bug fix. Specialized replications are used when you have a working solution that needs to be repurposed for something else. Sometimes finding an abstraction can be hard so cloning is necessary.

Forks are a "less evil version of cloning". Porting something to an architecture to another is easier to make something portable. Experimental variation is used to test differences and performance.

Clones lead to software aging.

5.2 Types of Clones

Type 1 - Exact clones No variance in system behavior

Type 2 - Parametrized clones same execution over different variables.

Type 3 - Near miss clones similar code that shares significant structural or functional similarities, making their detection more complex.

Type 4 - Semantic clones They do the same thing but with different representations.

Structural Clones Similarities on an architectural level (higher level analysis).

Artefact Clones comparisons between files, methods and paragraphs.

Model Clones

Contextual Clones similar things called differently. They aren't clones if they resolve to different things. They might appear similar or different based on the level they are observed from.

5.3 Clone Detection

DEF: Clone

Fragment of code that is duplicated somewhere else.

If something is duplicated, any instance of it is considered a clone.

DEF: Clone Pair

Two code fragments that are duplicates of each other

DEF: Clone Class

Any number of code fragments that are all duplicates of one another.

Clone detection can happen on different levels.

Strings Line by line comparison, uses hashing to speed up. Matched sections can be expanded to check further characters.

Tokens Every word has a type. Tokens can be normalized. Algorithms can be used to efficiently identify clones. With lexemes you cannot normalize types, with regular matching you can attempt to match.

Abstract Syntax Trees It is possible to find identical subtrees after assigning the types. It is possible to ignore the leaves to detect type 2 clones. Language specific matching will allow to identify type 3 clones.

Diagram/Model Usually program dependence graph. First you construct the control flow graph, then the data flow. Using approximations it is possible to find subgraphs with the same shape (type 3 clones).

Metrics instead of comparing representations, you split the code in fragments and hash them. You then get a list of metrics for each fragment (size, complexity, number of calls,...). Based on that you can infer the similarity between pieces of code (type 3 & 4 clones).

Level 6

6.1 Mining Software Repositories (MSR)

Empirical software evolution: software is considered an observable phenomenon (like weather). You look at the data that you have and try to formulate hypothesis on it (bottom-up). You then test those hypothesis (top-down).

6.1.1 Software Mining Goals

Identifying developer effort (was it just one person or a team? What did they focus on? Who did what?)

Developer social network how do the developers communicate? How does knowledge propagate?

Identifying the impact of changes what things need to change in unison.

Hotspot analysis identifying what changes frequently, is it because of poorly structured code?

Defect prediction identifying potential future bugs

6.1.2 Mining Phases

1. Data retrieval: look at the available sources and what can be extracted from them (git repository, issue tracker, documentation,...)
2. Data modelling: after the data is retrieved and **structured**, what should be the schema of that data, how will it be saved? (triples, database,...)
3. Data analysis: visualize data in a manner that helps the developer to build tools/panels/plugins.

6.1.3 Questions answered by MSR

- What kind of projects live longer?
- Does copy&paste programming lead to license violations?
- Do users leave projects when bugs don't get fixed fast?
- Is software just code with some extras?

6.2 MSR for Understanding → Questions

Mining software repositories for the sake of understanding what is in them.

6.2.1 Developer Effort

- How hard was/is to develop a particular feature (and how hard it's going to be for me to implement it)?
- What is the maintenance rate of the project (how often are bugs fixed)? Usually it's tied to contractual obligations.
- What is the distribution of effort in the team?

6.2.2 Change Coupling

Two or more artifacts can evolve together, and are co-changing.

- Which components change together often?
- What are the modules with the strongest coupling?
- Which files/folders are responsible for the coupling?

6.2.3 Hotspots

A hotspot is an outlier component, an evolutionary hotspot is an evolutionary outlier, something that changes in a different way compared to the rest of the software.

6.2.4 Unstructured Data

Even code, which is the most structured data, can be unorganized. How easy is it to identify a structure? Is it derived from an *unstructured* document?

Comments are linguistic data, an unstructured part of code. Names/naming conventions are also types of linguistic data.

Bug databases/issue trackers are another type of unstructured data. A structured issue would have an explicit link between a problem and a solution with timestamps and documentation.

Archives are another type of unstructured information. They include mailing lists, chat logs, requirement documents,... All require natural language processing, tokenization, topic discovery and concept matching.

6.3 MSR for Prediction

Why When do we stop testing? When do we know the quality of the software is enough? *What is the most efficient quality assurance resource allocation?*

Some modules are more error-prone than others. Issue trackers (for that project or similar) help.

6.3.1 Error Prone modules

What make a module error prone?

Complexity number of classes, level of interconnection.

Integration (external complexity) how reliant is this module on others.

Problem Domain some problem domains are more error prone than others.

Requirements Creep some module might need to be changed all the time because you didn't have all the requirements at the start.

Process what process was used to develop the module. Code review and testing make modules less error prone. Errors would be caught earlier.

6.3.2 Experimental Setup

To build a predictive model you need to **split** your information. One part is used to infer the predictive model, and the other part to validate it.

Precision measures how many components are truly predicted as defect prone ($\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$). High precision means that the information is usually predicted correctly.

Recall ($\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$) measures how many defect prone components are actually predicted as such.

F-score harmonic mean that combines precision and recall. Avoids cases of unbalanced precision & recall.

Correlation measures the strength and direction of the relationship between predicted ranking and observed ranking.

Co-occurrence if somethings co-occur many times then they may be somehow related.

6.3.3 Main Approaches

Changelog recently/often changed files are the source of defects.

Single Version the current behavior has bugs and will lead to defects in the future (draw conclusions from the current version).

Dependencies if you have a lot of dependencies, thats where vulnerabilities can come from.

Don't focus on the "then", focus on the "now".

Level 7

7.1 Search-Based Software Evolution (SBSE)

Software engineering is a problem with a lot of (correlate, clear or vague) constraints that must all be satisfied. Changing just one of them has a big impact on the solution.

Instead of looking for the perfect solution that satisfies all these constraints, we can use an algorithm that helps us find a **near optimal** solution.

This type of problem solving is called Search-Based Software Engineering.

7.1.1 Requirements for SBSE

A Problem and it's representation

Encoding we don't want to work with real world elements, only with the aspects that can be modelled.

Fitness Functions a quantitative evaluation of the problem solution.

Operators that can be used to produce new or variated solutions.

These solutions' *fitness* is tested on how the *problem* is being tackled. Search based techniques are used to find the optimal solution.

7.1.2 Genetic Terminology in SE

Chromosome problem representation

Genotype decision/solution

Phenotype objective

7.1.3 Search Based Algorithms

Hill Climbing start at a random point on a solution space (hill), you move towards the direction that gives immediate benefits, until you are in a better condition than all of your neighbors (not the whole space). There should be balance between cohesion and coupling.

Simulated Annealing you start somewhere and explore your neighbors. If you see that your neighbors are worse than you, you can still explore

them. This can be done to the extent of the permitted temperature. With time the system can be cooled to expand the domain of valid solutions. This grants the ability to explore things that seemed worse at first glance.

Particle Swarm Optimization evaluates a swarm of candidate solutions. Each candidate independently explores different combinations of code changes, but the entire swarm shares a collective memory of the best results. By communicating the highest quality solutions they discover, the swarm converges on the best one.

Genetic Algorithms survival of the fittest, using the fitness functions. We try to combine the chromosomes/genotypes to form a new genotype that leads to a better Phenotype that can be tested with the fitness function.

7.1.4 Other SBSE-Solvable Problems

Complexity Reduction graphs (to represent complexity), metrics to assess it, refactoring to reduce it.

Automatic Program Repair to patch a bug shown from a test. How to find that one line to fix the bug? The test suite acts as the fitness function. We do not want to go towards tests that fail. Solutions are applied through code mutations.

Program Optimization Analysis on the program at runtime (model, logs,...). Metrics are used to identify what you want to improve (security, testability, usability, complexity,...). Program transformations (refactorings or behavior changes) are used to apply the optimizations. These types of optimizations are very computationally heavy, but cheaper than hiring a developer.

7.2 Formal Verification in Software Evolution

DEF: Formal Verification

A technique used to prove existence consistency. It proves if a system satisfies the requirements.

To do this, we need:

- A formal description of the system
- A formal specification of the requirements (usually text, which can be left to interpretation).
- Rigorous formal rules to reason how the system conforms to the specification/requirements⁹.

7.2.1 Regression Verification (RV)

Writing specifications is hard, so we trust the existence of the system (specification) to define a spec that the current system needs to satisfy.

RV can be used to prove that the values used in regression testing are representative of the correct equivalence classes. In order to do this, an equivalence relation definition is required.

7.2.2 Generalized Test Tables

Since writing specs is hard, is there a simpler way to write them?

We put them in a table that links the inputs to the outputs. We can replace the concrete values in the table with constraints. This also makes RV easier by generating a random value from the equivalence classes. Generalized test tables enable thorough investigations, weaker/stricter conformance can be checked.

7.2.3 Interdisciplinary models

Since writing specs is hard, is there a way to infer them?

We can take a (UML) model of the system, composed of elements. We can infer from it a smaller model, tailored to a specific tool (i.e. a theorem prover).

⁹We assume that this formal verification is automated.

Always using formal methods is not always feasible, but should be used to cover gaps/critical sections.