Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 03 (26.10.2015)

The Software Development Process

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

Your Daily Menu

- Models of software development
  - The software development process, and its rôle in safety-critical software development.
  - What kind of development models are there?
  - Which ones are useful for safety-critical software – and why?
  - What do the norms and standards say?
- Basic notions of formal software development
  - What is formal software development?
  - How to specify: properties and hyperproperties
  - Structuring of the development process

Where are we?

- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with SysML
- 07: Detailed Specification with SysML
- 08: Testing
- 09 and 10: Program Analysis
- 11: Model-Checking
- 12: Software Verification (Hoare-Calculus)
- 13: Software Verification (VCG)
- 14: Conclusions

Software Development Models

Software Development Process

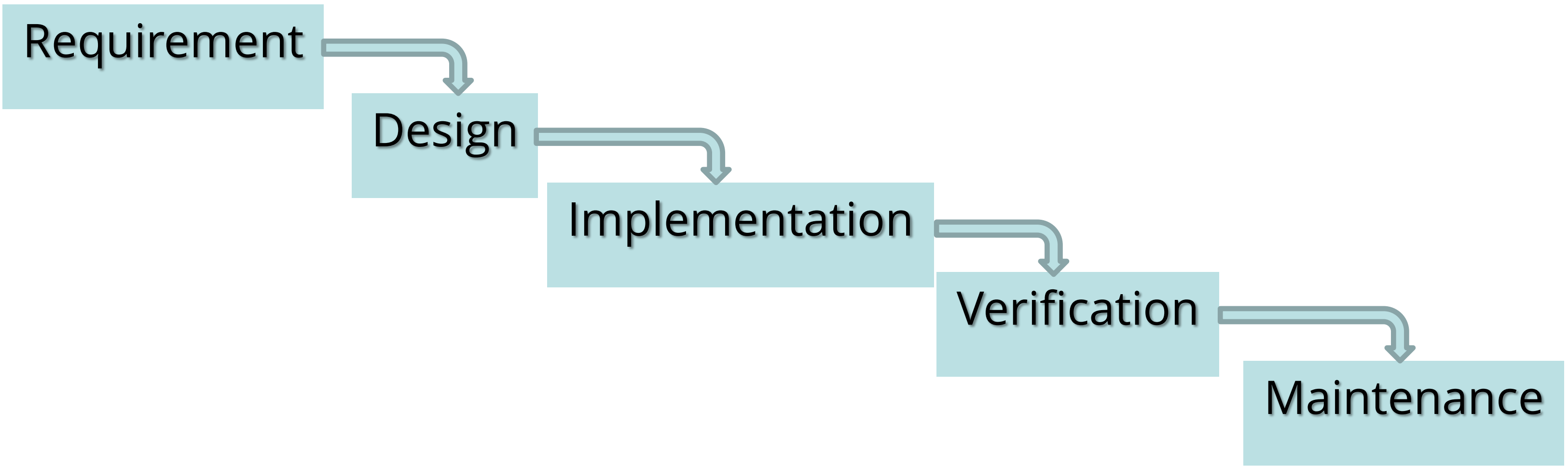
- A software development process is the **structure** imposed on the development of a software product.
- We classify processes according to *models* which specify
  - the artefacts of the development, such as
    - the software product itself, specifications, test documents, reports, reviews, proofs, plans etc
  - the different stages of the development,
  - and the artefacts associated to each stage.
- Different models have a different focus:
  - Correctness, development time, flexibility.
- What does quality mean in this context?
  - What is the *output*? Just the sofware product, or more? (specifications, test runs, documents, proofs...)

Agile Methods

- Prototype-driven development
  - E.g. Rapid Application Development
  - Development as a sequence of prototypes
  - Ever-changing safety and security requirements
- Agile programming
  - E.g. Scrum, extreme programming
  - Development guided by functional requirements
  - Process structured by rules of conduct for developers
  - Less support for non-functional requirements
- Test-driven development
  - Tests as *executable specifications:* write tests first
  - Often used together with the other two

Waterfall Model (Royce 1970)

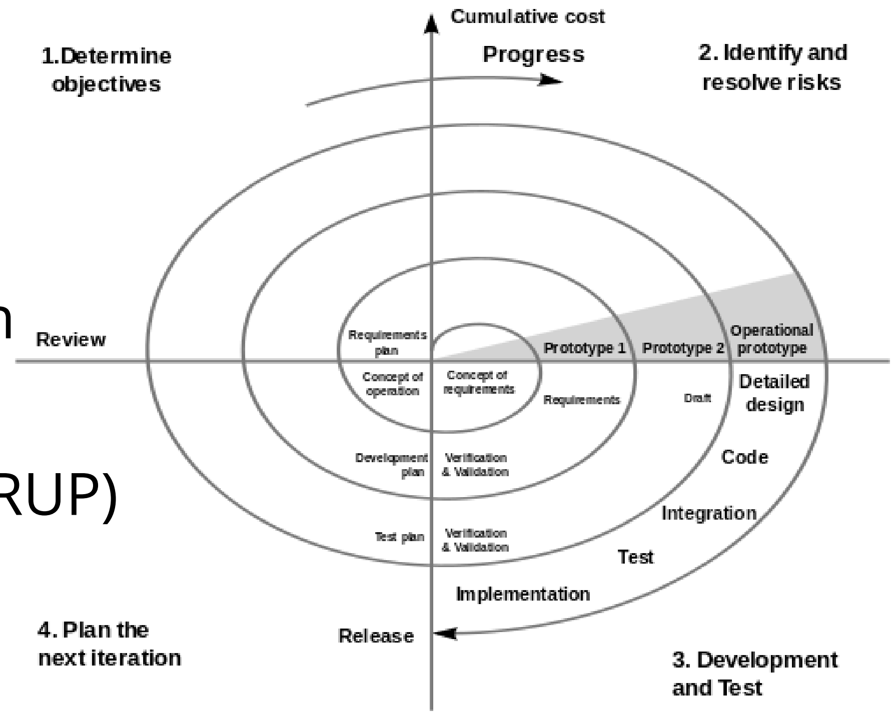
- Classical top-down sequential workflow with strictly separated phases.



- Unpractical as actual workflow (no feedback between phases), but even early papers did not *really* suggest this.

Spiral Model (Böhm, 1986)

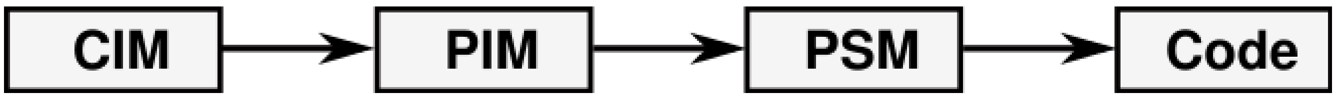
- Incremental development guided by **risk factors**
- Four phases:
  - Determine objectives
  - Analyse risks
  - Development and test
  - Review, plan next iteration



- See e.g.
  - Rational Unified Process (RUP)
- Drawbacks:
  - Risk identification is the key, and can be quite difficult

Model-Driven Development (MDD, MDE)

- Describe problems on abstract level using *a modelling language* (often a *domain-specific language*), and derive implementation by model transformation or run-time interpretation.
- Often used with UML (or its DSLs, eg. SysML)

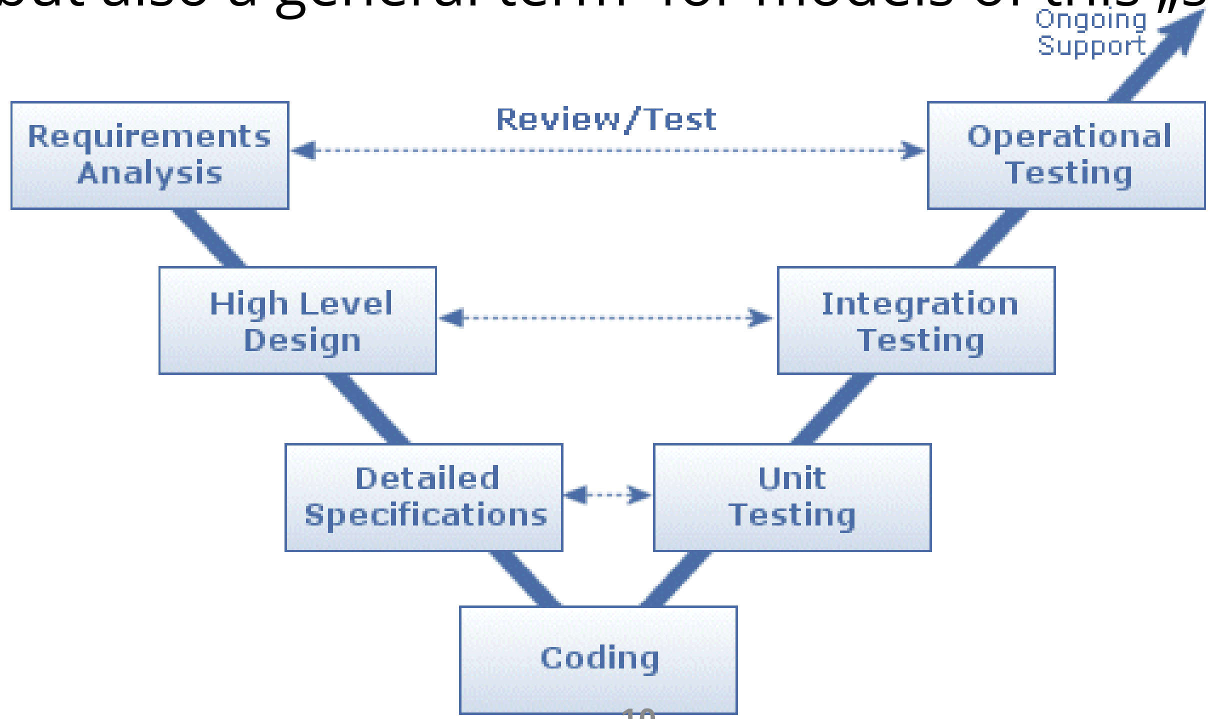


- Variety of tools:
  - Rational tool chain, Enterprise Architect, Rhapsody, Papyrus, Artisan Studio, MetaEdit+, Matlab/Simulink/Stateflow*
  - EMF (Eclipse Modelling Framework)
- Strictly sequential development
- Drawbacks: high initial investment, limited flexibility

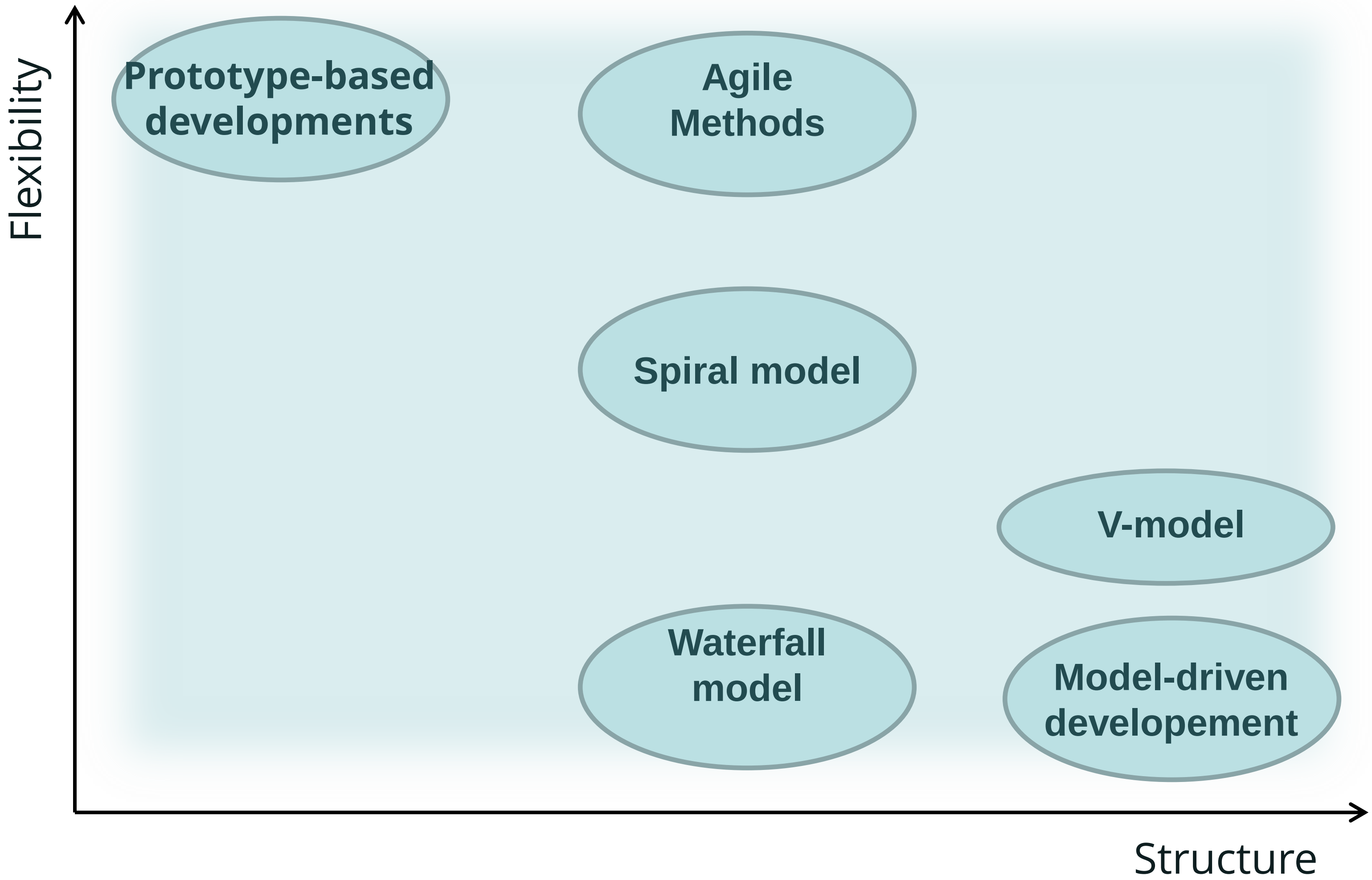
* Proprietary DSL – not related to UML

V-Model

- Evolution of the waterfall model:
  - Each phase is supported by a corresponding testing phase (verification & validation)
  - Feedback between next and previous phase
- Standard model for public projects in Germany
  - … but also a general term for models of this „shape“



Software Development Models



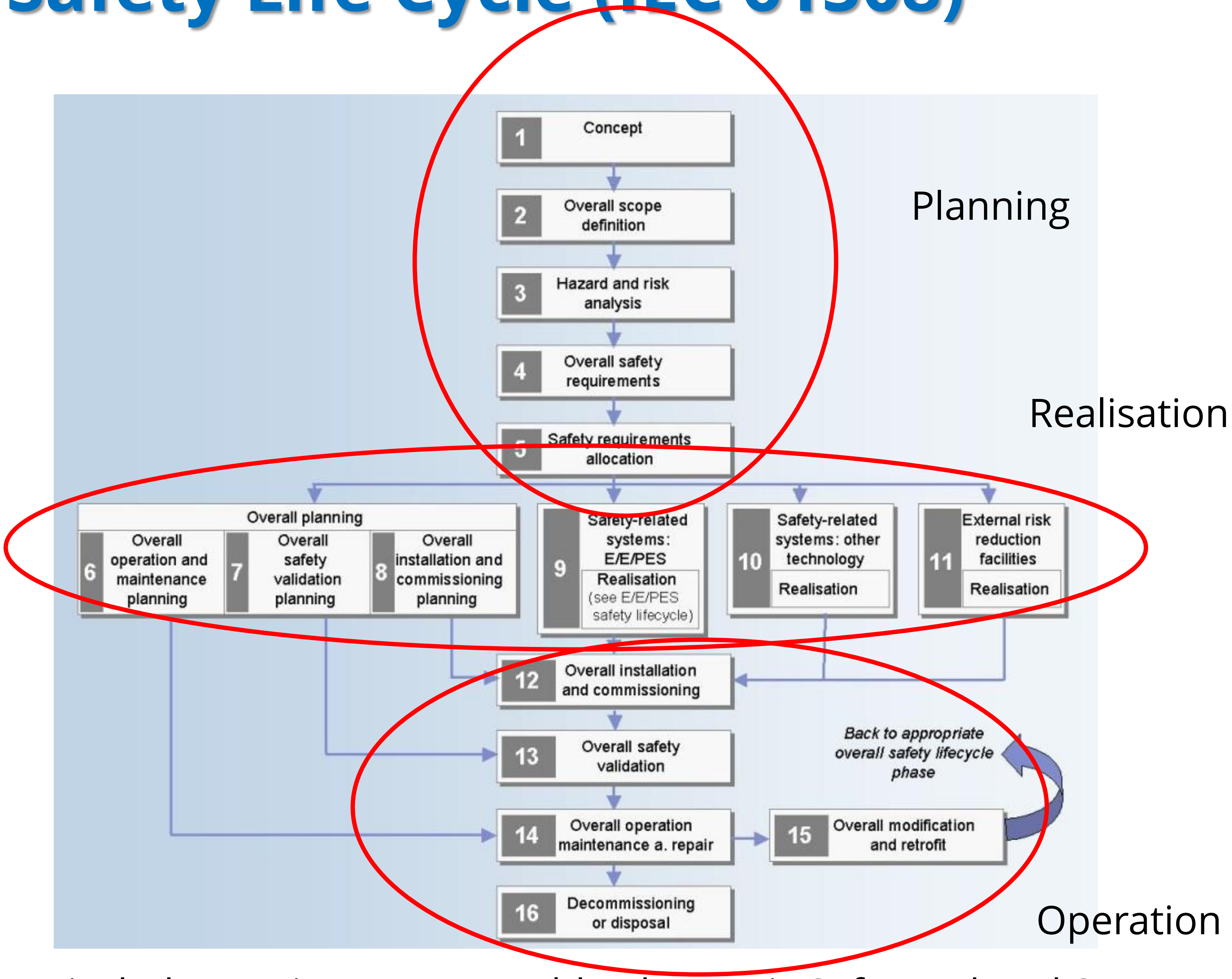
from S. Paulus: Sichere Software

Development Models for Critical Systems

Development Models for Critical Systems

- Ensuring safety/security needs structure.
  - ...but *too much* structure makes developments bureaucratic, which is *in itself* a safety risk.
  - Cautionary tale: Ariane-5
- Standards put emphasis on *process*.
  - Everything needs to be planned and documented.
  - Key issues: auditability, accountability, traceability.
- Best suited development models are variations of the V-model or spiral model.
- A new trend?
  - V-Model for initial developments of a new product
  - Agile models (e.g. SCRUM) for maintenance and product extensions

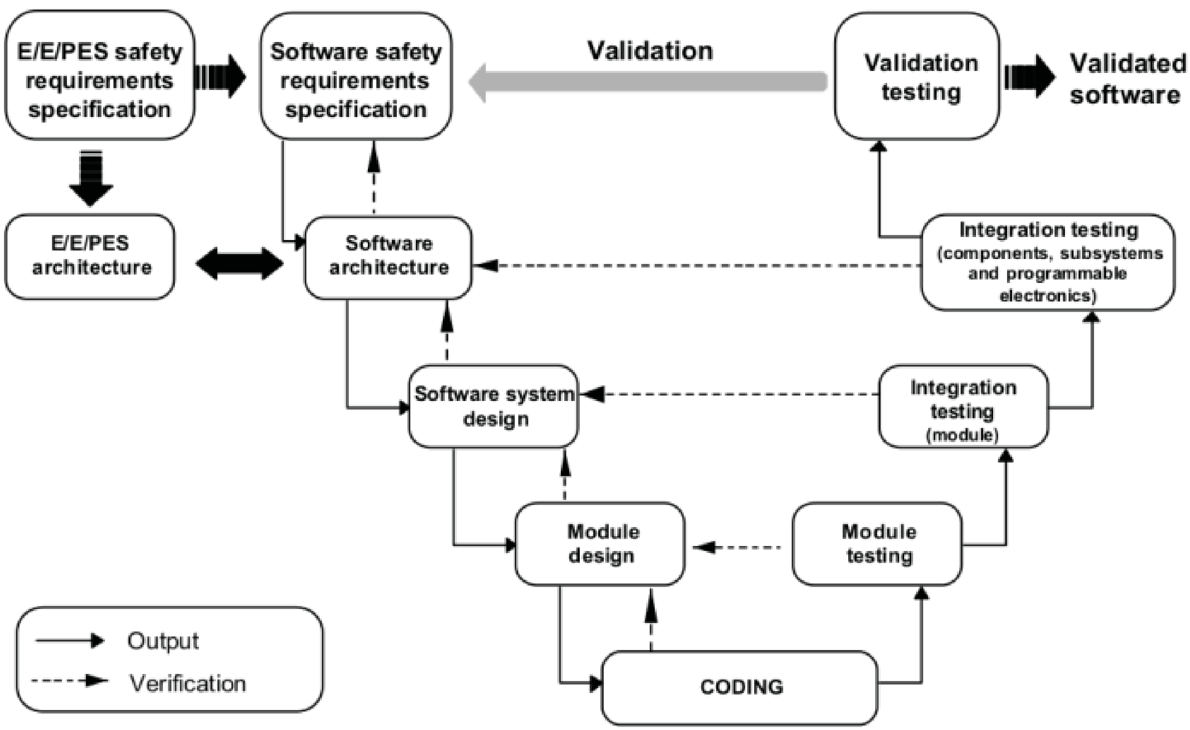
The Safety Life Cycle (IEC 61508)



E/E/PES: Electrical/Electronic/Programmable Electronic Safety-related Systems

Development Model in IEC 61508

- IEC 61508 prescribes certain activities for each phase of the life cycle.
- Development is one part of the life cycle.
- IEC 61508 *recommends* V-model.



Development Model in DO-178B

- DO-178B defines different *processes* in the SW life cycle:
  - Planning process
  - Development process, structured in turn into
    - Requirements process
    - Design process
    - Coding process
    - Integration process
  - Verification process
  - Quality assurance process
  - Configuration management process
  - Certification liaison process
- There is no conspicuous diagram, but the Development Process has sub-processes suggesting the phases found in the V-model as well.
  - Implicit recommendation of the V-model.

Traceability

- The idea of being able to follow requirements (in particular, safety requirements) from requirement spec to the code (and possibly back).

- On the simplest level, an Excel sheet with (manual) links to the program.

- More sophisticated tools include DOORS.
  - Decompose requirements, hierarchical requirements
  - Two-way traceability: from code, test cases, test procedures, and test results back to requirements
  - Eg. DO-178B requires all code derives from requirements

Artefacts in the Development Process

Planning:

- Document plan
- V&V plan
- QM plan
- Test plan
- Project manual

Specifications:

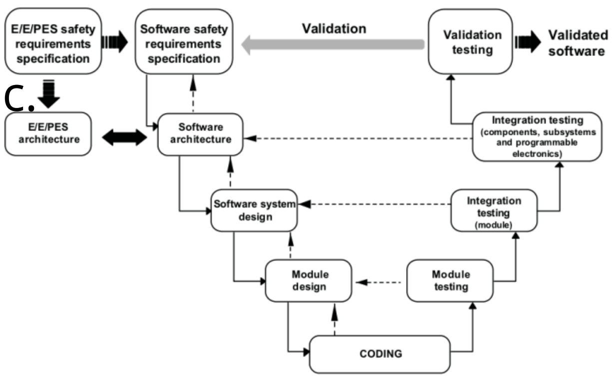
- Safety requirement spec.
- System specification
- Detail specification
- User document (safety reference manual)

Implementation:

- Code

Verification & validation:

- Code review protocols
- Test cases, procedures, and test results,
- Proofs



Possible formats:

- Word documents
- Excel sheets
- Wiki text
- Database (Doors)

- UML/SysML diagrams
- Formal languages:
  - Z, HOL, etc.
  - Statecharts or similar diagrams

- Source code

Documents must be *identified* and *reconstructable*.

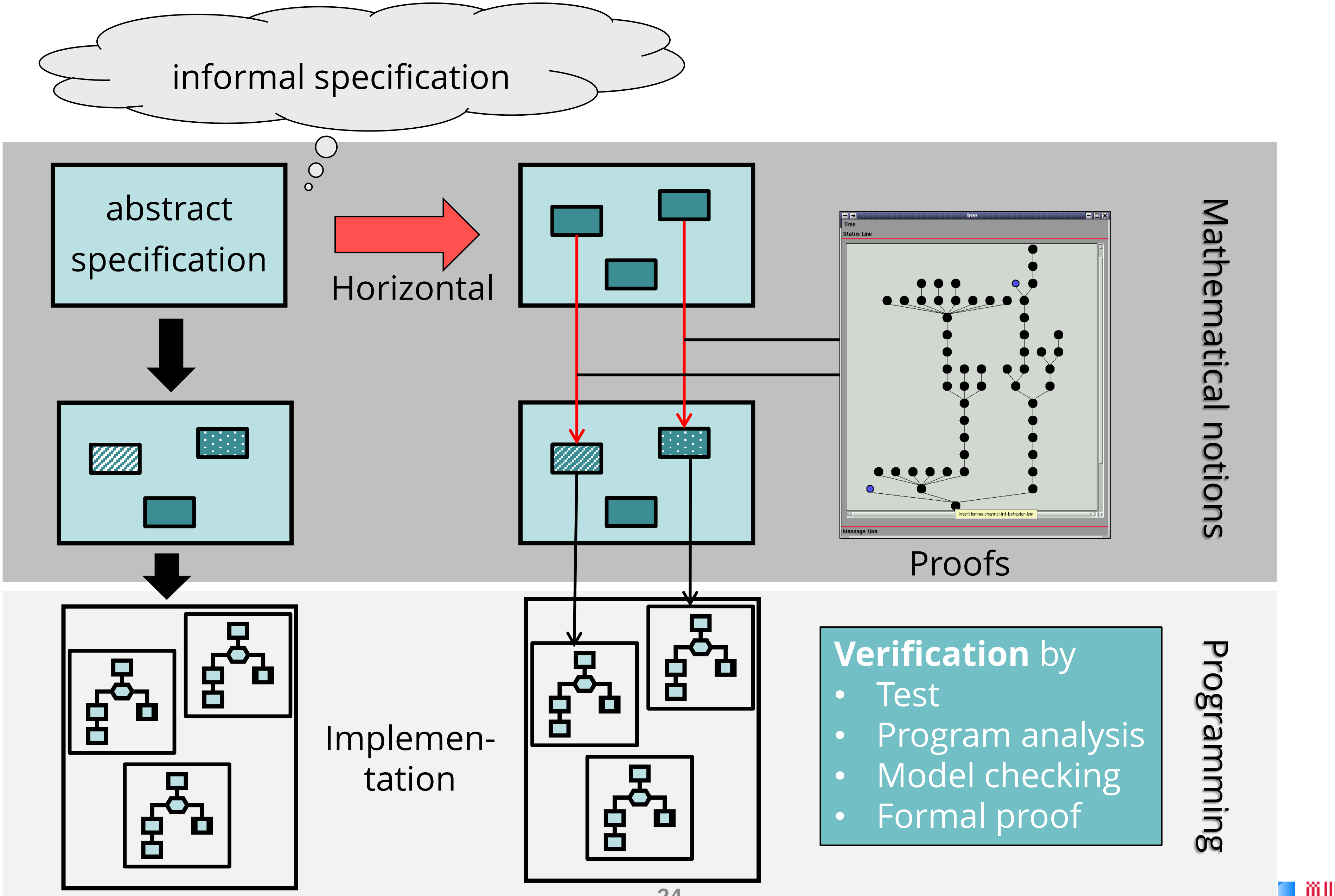
- Revision control and configuration management *mandatory*.

Basic Notions of Formal Software Development

Formal Software Development

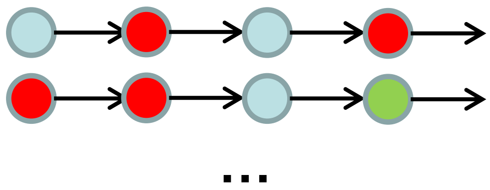
- In **formal** development, properties are stated in a rigorous way with a precise mathematical semantics.
- These formal specifications can be **proven**.
- Advantages:
  - Errors can be found **early** in the development process, saving time and effort and hence costs.
  - There is a higher degree of trust in the system.
  - Hence, standards recommend use of formal methods for high SILs/EALs.
- Drawback:
  - Higher effort
  - Requires **qualified** personnel (that would be *you*).
- There are tools which can help us by
  - **finding** (simple) proofs for us, or
  - **checking** our (more complicated) proofs.

Formal Software Development



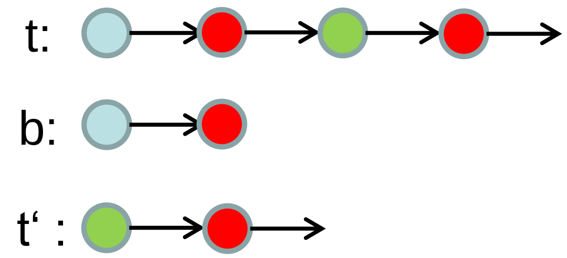
A General Notion of Properties

- **Defn:** a **property** is a set of infinite execution traces (i.e. infinite sequences of states)



- Trace t satisfies property P, written $t \vDash P$, iff $t \in P$

- b ≤ t iff $\exists t'. t = b \cdot t'$
  - i.e. b is a *finite* prefix of t



Safety and Liveness Properties

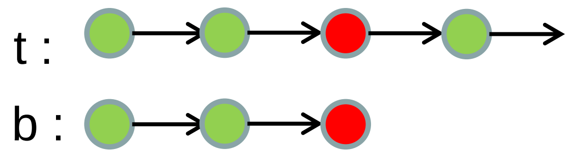
Alpen & Schneider (1985, 1987)

- **Safety** properties
  - *Nothing bad happens*
  - partial correctness, program safety, access control
- **Liveness** properties
  - *Something good happens*
  - Termination, guaranteed service, availability

- **Theorem**: $\forall P \,.\; P = Safe_P \cap Live_P$
  - Each property can be represented as a combination of safety and liveness properties.

Safety Properties

- Safety property S: „Nothing bad happens"
- A bad thing is *finitely* observable and *irremediable*
- S is a safety property iff
  - $\forall t.\, t \notin S \rightarrow (\exists b.\, \text{finite } b \wedge b \leq t \rightarrow \forall u.\, b \leq u \rightarrow u \notin S)$



  - a finite prefix b always causes the bad thing

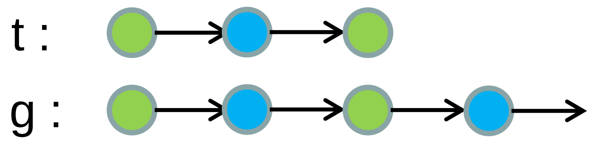
- **Safety is typically proven by induction.**
  - Safety properties may be enforced by run-time monitors.
  - Safety is testable (i.e. we can test for non-safety)

Liveness Properties

- Liveness property L: „Good things will happen“

- A good thing is always possible and possibly infinite:

- L is a liveness property iff
  - $\forall\, t.\ \text{finite}\ t\ \rightarrow \exists g.\, t \leq g \wedge g \in L$
  - i.e. all finite traces t can be extended to a trace g in L.



- **Liveness is typically proven by well-foundedness.**

Underspecification and Nondeterminism

- A *system* S is characterised by a *set of traces*, $[\![S]\!]$
- A system S *satisfies* a property P, written

$$S \models P \text{ iff } [\![S]\!] \subseteq P$$

- Why more than one trace? Difference between:
  - *Underspecification* or *loose specification* –
    we specify several *possible* implementations, but each implementation should be deterministic.
  - Non-determinism – different program runs might result in different traces.
- Example: a simple can vending machine.
  - Insert coin, chose brand, dispense drink.
  - Non-determinisim due to *internal* or *external* choice.

Security Policies

Many security policies are not properties!

- Examples:
  - Non-Interference (Goguen & Meseguer 1982)
    - Commands of high users have no effect on observations of low users
  - Average response time is lower than k.

- Security policies are examples of hyperproperties.
- A **hyperproperty** H is a set of properties
  - i.e. a set of set of traces.
  - System S satisfies H, $S \vDash H$, iff $[\![S]\!] \in H$.

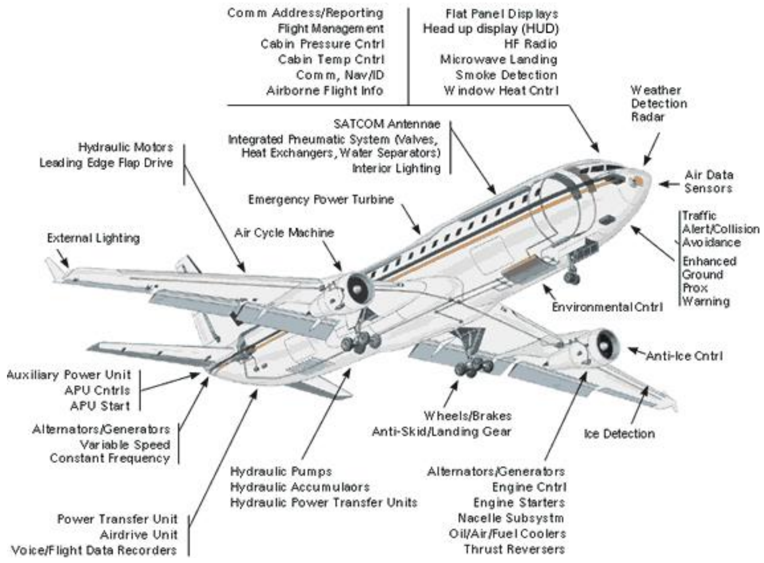
Structuring the Development

Structure in the Development

- Horizontal structuring
  - Modularization into components
  - Composition and Decomposition
  - Aggregation
- Vertical structuring
  - Abstraction and refinement
    from design specification to implementation
  - Declarative vs. imparative specification
  - Inheritence
- Layers / Views
  - Adresses multiple aspects of a system
  - Behavioral model, performance model, structural model, analysis model(e.g. UML, SysML)

Horizontal Structuring (informal)

- Composition of components
  - Dependent on the individual layer of abstraction
  - E.g. modules, procedures, functions,...
- Example:



Horizontal Structuring: Composition

- Given two systems S_1, S_2, their *sequential composition* is defined as

$$S_1; S_2 = \{s \cdot t \mid s \in [\![S_1]\!], t \in [\![S_2]\!]\}$$

  - All traces from S_1, followed by all traces from S_2.

- Given two traces s, t, their *interleaving* is defined (recursively) as
  $<> \parallel t = t$
  $s \parallel <> = s$
  $a \cdot s \parallel b \cdot t = \{ a \cdot u \mid u \in s \parallel b \cdot t \} \cup \{ b \cdot u \mid u \in a \cdot s \parallel t\}$

- Given two systems S_1, S_2, their *parallel composition* is defined as

$$S_1 \parallel S_2 = \{ s \parallel t \mid s \in [\![S_1]\!], t \in [\![S_2]\!]\}$$

  - Traces from S_1 interleaved with traces from S_2.

Vertical Structure - Refinement

- Data refinement
  - Abstract datatype is „implemented“ in terms of the more concrete datatype
  - Simple example: define stack with lists
- Process refinement
  - Process is refined by excluding certain runs
  - Refinement as a reduction of underspecification by eliminating possible behaviours
- Action refinement
  - Action is refined by a sequence of actions
  - E.g. a stub for a procedure is refined to an executable procedure

Refinement and Properties

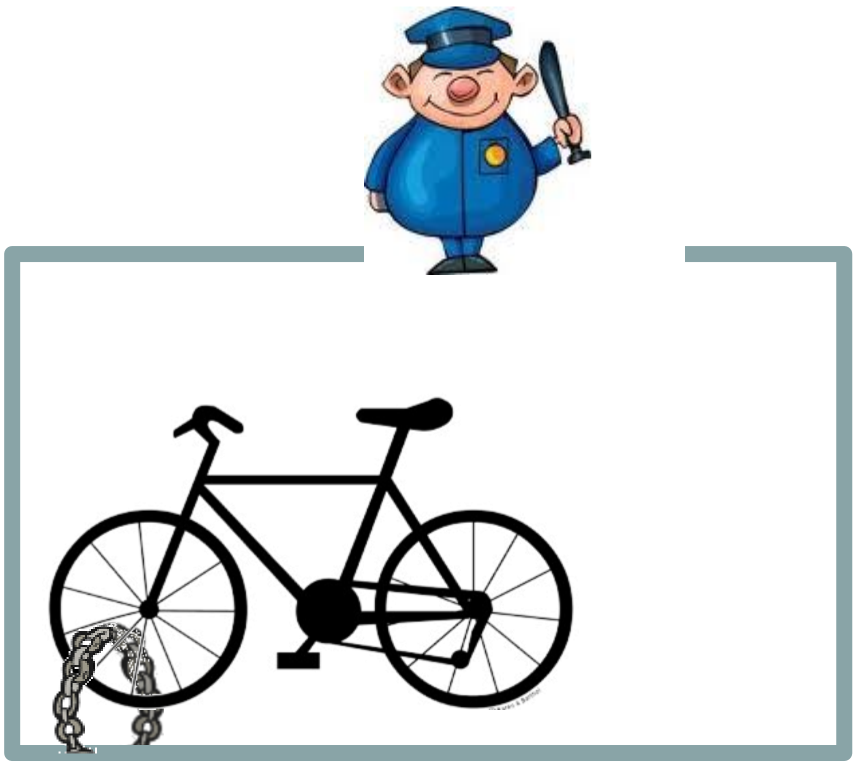
- Refinement typically preserves safety properties.
  - This means if we start with an abstract specification which we can show satisfies the desired properties, and refine it until we arrive at an implementation, we have a system for the properties hold *by construction*:

$$SP \rightsquigarrow SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow Imp$$

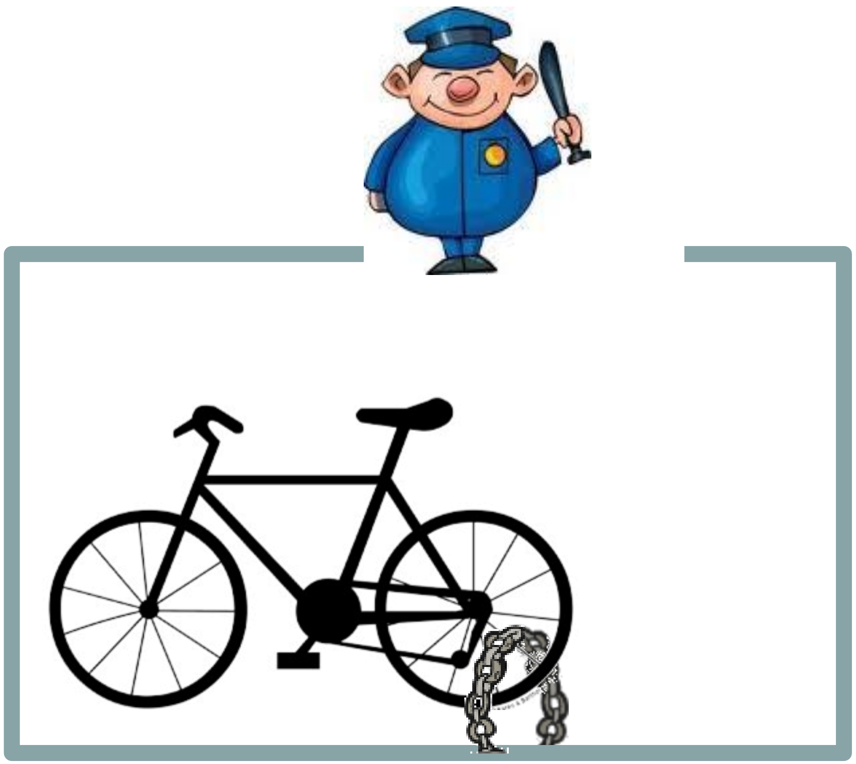
- However, **security** is typically **not** preserved by refinement nor by composition!

Security and Composition

Only complete bicycles are allowed to pass the gate.



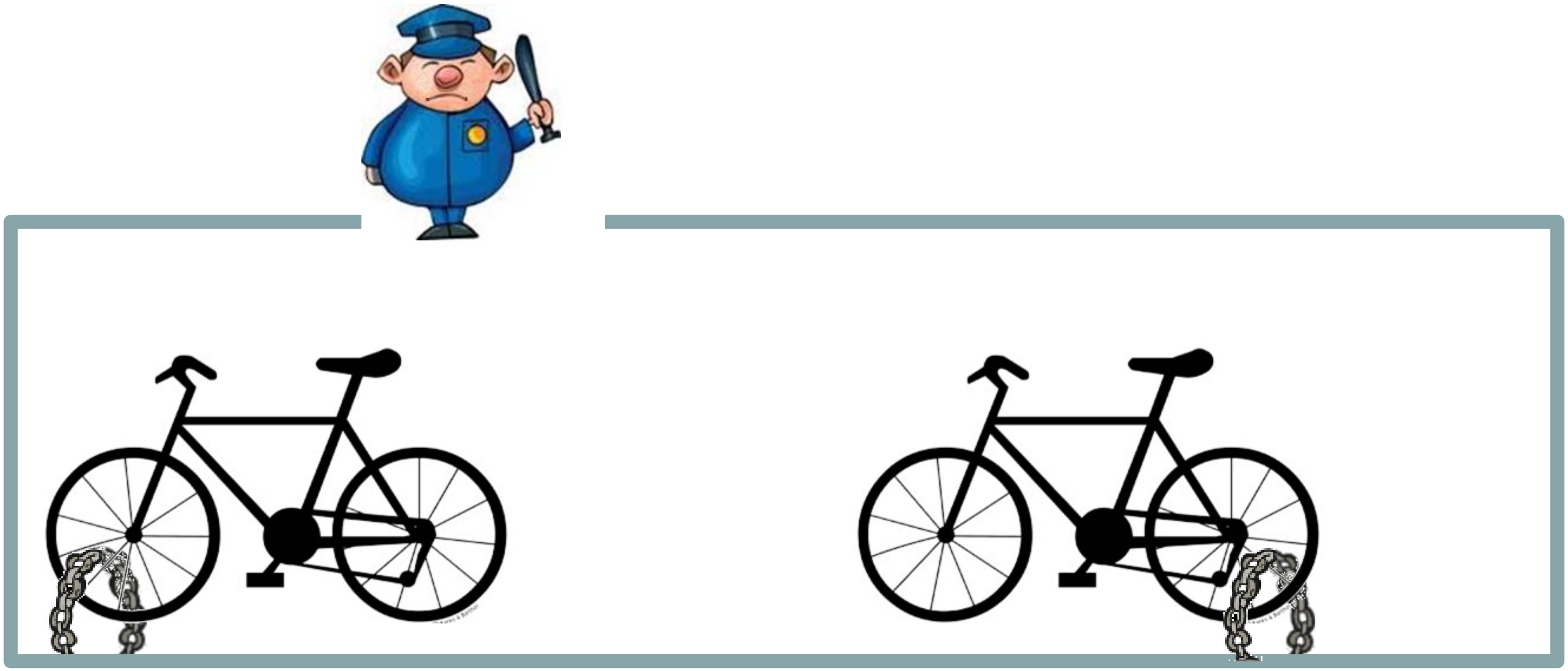
Secure !



Secure !

Security and Composition

Only complete bicycles are allowed to pass the gate.



Insecure !

A Formal Treatment of Refinement

- **Def**: T is a refinement of S if $S \sqsubseteq T \Leftrightarrow [\![T]\!] \subseteq [\![S]\!]$
  - Remark: a bit too general, but will do here.

- **Theorem:** Refinement preservers properties:

  If $S \vDash P$ and $S \sqsubseteq T$, then $T \vDash P$.

  - Proof: Recall $S \vDash P \Leftrightarrow [\![S]\!] \subseteq \mathrm{P}$, and $S \sqsubseteq T \Leftrightarrow [\![T]\!] \subseteq [\![S]\!]$,

    hence $[\![T]\!] \subseteq P \Leftrightarrow T \vDash P$.

- However, refinement does **not** preserve hyperproperties.
  - Why? $S \vDash H \Leftrightarrow [\![S]\!] \in H$, but H **not** closed under subsets.

Conclusion & Summary

- Software development models: structure vs. flexibility
- Safety standards such as IEC 61508, DO-178B suggest development according to V-model.
  - Specification and implementation linked by verification and validation.
  - Variety of artefacts produced at each stage, which have to be subjected to external review.
- Properties: sets of traces

  hyperproperties: sets of properties
- Structuring of the development:
  - Horizontal – e.g. composition
  - Vertical – refinement (data, process and action ref.)
  - Refinement preserves properties (safety), but not hyperproperties (security).