Information Transformation in Software Engineering: From Business Intent to a Working System

Gytis Gurklys

2026-08-12

https://gytisgurklys.eu/en/posts/information-transformation-in-software-engineering/

ABSTRACT

This paper examines the software engineering process as a sequential transformation of information: from unspoken business intent to a system running in a production environment. A six-level transformation model is proposed, defining for each level its input, transformation, output and characteristic error type. The first three levels are decomposed into finer steps, a single artifact registry with a dependency network is compiled, a role model is derived from that registry, and general principles of process design are formulated. The work of the business analyst (requirements engineer) at the first transformation level is examined in detail. The method is conceptual deductive analysis: the model is presented as a hypothesis grounded in internal consistency and illustrated with examples from the invoice management domain; the whole chain is presented at the granularity of the model, while the first level is examined in detail. The context of the analysis is a business application with a graphical user interface, operating globally and serving thousands of users.

Keywords: requirements engineering, business analysis, information transformation, software engineering process, artifact traceability, validation.

CONTENTS

1. INTRODUCTION

1.1. Object of study and relevance

Software development is traditionally described as a sequence of activities: analysis, design, programming, testing, deployment. This paper takes a different cut — software development is examined as a process of information transformation, in the course of which natural-language statements about business goals are gradually turned into executable code and, finally, into a working system.

This cut is relevant for several reasons. First, it makes it possible to localise precisely where and what kind of information is lost or distorted, and to design prevention mechanisms accordingly. Second, it explains why certain process steps are necessary rather than conventional: each step performs a defined informational job, and if that job is left undone, someone will produce the missing information later, usually in an uncontrolled way. Third, it provides a unifying basis for analysing artifacts, roles and quality assurance practice.

Requirements, in this paper, are statements describing the goals that software compiled from code and running in a production environment must achieve.

1.2. Aim and objectives

The aim of this paper is to present a coherent model of information transformation spanning the whole path from business intent to a working system, and on that basis to formulate practical guidance for the work of a business analyst. The depth of coverage is deliberately uneven: the whole chain is presented at the granularity of the model, while detailed analysis is devoted to the first level; the internal decomposition of levels four to six is left for further work.

Objectives:

1. Define the transformation levels, specifying for each its input, transformation, output and characteristic error type.
2. Decompose the first three levels into steps according to a uniform criterion of distinction.
3. Compile an artifact registry with unique identifiers and a dependency network.
4. Derive a project role model from the registry.
5. Formulate general principles of process design.
6. Describe in detail the work of the business analyst at the first level, including working techniques, quality criteria and typical failure scenarios.
7. Position the model relative to existing software engineering theories and identify directions for testing it.

1.3. Method and scope

The paper is a conceptual analysis: the model is constructed deductively, from the nature of information transformation, and illustrated with examples from the invoice management domain. The context of the analysis is a business application with a graphical interface operating on a global scale with thousands of users; this context determines the emphasis in particular places (localisation, concurrency, regional differences), but the model itself does not depend on it.

The status of the work is a conceptual hypothesis: the model is justified by internal consistency and deduction, not by empirical testing. The examples are illustrations, not proofs; a seamless walk through the first level is given in Appendix A. At the same time the model formulates testable predictions: for example, that omission-type errors (a missed party, a missed scenario) correlate more strongly with late-detected and expensive defects than distortion-type errors, and that projects without systematic generation procedures (the 1.1 map, the 2.4 questioning) have a higher share of silent

errors. Such predictions can be tested by retrospective defect analysis, without interfering with the process as it runs.

The paper deliberately does not examine: the formation of business strategy (it happens before the first level and outside the boundaries of software engineering), project management methodologies (they organise the process in time but do not change the structure of the information flow) and the internal decomposition of levels four to six (left for further work).

1.4. Structure of the paper

The paper consists of two parts. The first part (Sections 2–8) presents the whole model: theoretical foundation, relation to existing theories, the six-level structure, design principles, the internal decomposition of levels, the artifact registry, the role model and process topology. The second part (Sections 9–14) examines the work of the business analyst in detail: definition of the role, the core working concepts, an analysis of the seven steps, the prevention of silent errors and quality gates. In Appendix A a single requirement is taken through all the steps of the first level; a list of references is given at the end.

PART ONE. THE TRANSFORMATION MODEL

2. THEORETICAL FOUNDATION

2.1. The notion of information transformation

Software development can be described as a sequential condensation of information: at every level uncertainty decreases and formality grows, until natural-language statements turn into executable code and code into a working system. Information moves from answering "why" (goals) through "what" (requirements, behaviour) and "how" (design decisions) to execution (code, a working system).

The term "information" is used here in an engineering sense, not in the sense of information theory: information here is the decisions taken and the uncertainty removed about the system being built, not Shannon's quantitative measure [14]. The claim that a level "produces new information" means that decisions are taken in it that were not present in earlier artifacts, not that the number of bits in a transmitted message grows. This distinction is essential: the chain produces not bits but decisions.

Every transformation is at the same time a risk of loss: ambiguity is possible, a scenario may be missed, an interpretation may be wrong. From this follows the requirement of traceability — the ability to trace any lower-level artifact back to the higher-level artifact that justifies it.

The transformation sequence across the whole chain, in summary: tacit (unspoken) knowledge → explicit and unstructured → structured → formalised → verified → agreed → executed. Each stage adds one property to the information: recorded, structured, checkable, agreed.

2.2. Formalisation as the narrowing of language

Technically, formalisation is the narrowing of language: free natural language is replaced by an increasingly restricted one (templates, mandatory fields, a controlled vocabulary), until a fully determined language — code — is reached.

The essential mechanism of formalisation is not the form of recording but forced choices. Natural language allows uncertainty to survive: phrasings such as "a sensible deadline" or "an appropriate user" look like information but are in fact deferred decisions. A template with a mandatory field does not permit deferral. Most of the work of formalisation is therefore not writing but taking the deferred decisions, or passing them to whoever has the authority to take them.

What formalisation yields:

1. A guarantee of a single interpretation. Different people at different times read a formal statement in the same way, without the author being present; the information becomes self-standing, no longer tied to the person who created it. This is critical, because between levels information changes hands.
2. Checkability. A formal statement has truth conditions, so a test can be derived from it while no code yet exists.
3. Earlier error detection. A contradiction between formal statements is visible mechanically at the specification stage, where the cost of correction is incomparably lower than in a production environment.
4. Mechanical processability. Formal artifacts can be checked with tools, traced, and used to generate other artifacts.

The price of formalisation is the risk of false precision: a template forces a choice of precision even where the available knowledge does not justify it. A specific value, once written down, looks reliable by virtue of its form alone. The instrument for managing this risk is assumption discipline (see

Subsection 4.5). In summary: formalisation does not create knowledge, it only shows where knowledge is missing, and this should be regarded as its most valuable side effect.

2.3. The function of classification

Classification and formalisation do different jobs: classification orders the space between statements, formalisation the space inside each statement.

Collected raw material is naturally ordered by origin (who said it, in which document it was found), but it is used according to purpose (where in the system it will live). Classification is re-indexing from origin to purpose. It confers four properties:

1. Comparability. Only statements assigned to the same class and the same anchor become comparable with one another; duplicates and contradictions are visible only in a shared dimension.
2. The possibility of asking about completeness. The question "do we have all the rules of a given kind" can be asked only once that kind is a defined class; classification turns unknown unknowns into countable empty cells.
3. Early routing. The type of a statement determines where in the future system it will materialise (a data constraint in the validation layer, a state rule in the state machine, an access rule in the authorisation layer); classification is an early architectural decision.
4. The choice of a formalisation language. Each class has its own template; classification is a precondition of formalisation, because a statement cannot be formalised correctly without knowing what it is.

2.4. The irreducibility of the chain

A hypothetical case in which the client formulated their needs directly in program code reveals the nature of the chain. Code is a fully determined language: in it every edge case, every error branch and every concurrency scenario has a defined answer. The information the client holds contains no such answers — the corresponding questions never arose in their work. From this follows an essential conclusion: the transformation chain exists not in order to translate available information into another notation, but in order to create the information that is missing — each level generates decisions that were not present at the previous one. The obstacle is not language but the missing decisions.

The same thought is expressed by the classical distinction between accidental and essential complexity [1]: notation and tools can be improved without limit, but the essential complexity (deciding precisely what is wanted) does not go anywhere. Precise specification is programming, whatever language is used.

In terms of decisions: if the output contains decisions that were not in the input, the difference must be produced by a person, a process or a guessing mechanism (the sense of the term "information" is defined in Subsection 2.1). Programming in natural language, large language models included, does not abolish the chain but compresses it into dialogue: either clarifying questions resolve the ambiguity (the analogue of requirements elicitation), or guesses fill it in — and then nobody has consciously decided the edge cases of the system's behaviour. This theme is developed in Subsection 2.6.

2.5. Relation to existing theories

The model is not built on empty ground: most of its building blocks are inherited from existing traditions in software engineering. It is honest to name exactly what is borrowed and what is new, because originality is visible only against a background.

Borrowed. The distinction between accidental and essential complexity, on which the irreducibility argument (2.4) rests, is Brooks's [1]. The quality criteria for requirements (atomicity, unambiguity, testability), the distinction between validation and verification, and the separation of functional from

non-functional requirements come from the requirements engineering literature [3, 4, 5, 6]; elicitation techniques (interviews, observation, situational questions) are described in detail by Gause and Weinberg [7]. The goal hierarchy and the relation between a goal and a requirement (steps 1.3–1.5) follow the goal-oriented requirements engineering tradition [15, 17]. Traceability as a problem in its own right was formulated by Gotel and Finkelstein [8], and baselines and change management procedures are standard configuration management material [9]. The form of a decision record (4.4) follows the practice of architecture decision records [10]. A glossary with established term definitions (step 2.1) does the same job as ubiquitous language in domain-driven design [11], while the inventory of interaction units (step 2.2) builds on use case methodology [12]. The MVP learning criterion (8.3) is taken from Lean Startup [16].

New. The claim to originality covers four things. First, error type as the criterion for distinguishing a step (4.1): a step is justified only when one can go wrong in it in a way one cannot go wrong anywhere else; in the literature, steps are usually distinguished by activity or by artifact. Second, deriving the role model and the quality gates from the bottlenecks of the artifact registry (Sections 6–7) rather than postulating them. Third, a reformulation of the purpose of the chain: not to translate available information but to produce the missing decisions; this thought is close to Naur's thesis that programming is theory building [2], but here it is operationalised at the level of artifacts and steps. Fourth, a single fractal transformation sequence (4.8), applied both to the chain as a whole and to the interior of each level.

Differences. The model differs from the waterfall reading in treating levels as logical rather than calendar phases (see Subsections 3.1 and 8.1–8.3). It differs from part of agile practice in not treating baselines and quality gates as bureaucracy: they arise from information changing hands at bottlenecks, and therefore survive in small iterations too, only on a smaller scale.

2.6. The place of large language models in the chain

Large language models change the economics of the chain but not its structure; the model makes it possible to state this precisely.

What is compressed. Language models make the work of formalisation cheaper — that is, the accidental complexity: translation between notations, the generation of drafts, the fifth-level transformation from design to code, and part of the second-level mechanics (writing out scenarios to a template). Wherever the transformation is a translation of information already held, a language model performs it quickly and often well.

What remains. The essential work, by 2.4, is the production of missing decisions, and it does not go anywhere — only the place where it happens changes. A dialogue with a model is a compressed chain: clarifying questions correspond to elicitation, and every question left unasked means the gap was filled by a guess. A guess is an omission-type error: it is invisible in review, because generated text looks finished and convincing. Hence a prediction of the model: the use of language models reduces the share of distortion errors (mechanical translation is accurate) but, without systematic coverage procedures, increases the share of silent omissions, because guesses are produced at machine speed and in convincing form.

What a language model cannot do in principle. Validation (step 1.6) compares an artifact against a benchmark that exists only in the memory of the parties (Subsection 11.6), and therefore cannot be delegated to any tool without access to those parties' intentions. Questions of mandate (4.3) — conflicts, priorities, scope boundaries — are questions of value, not of knowledge, and therefore cannot be delegated to anyone without the authority. The practical conclusion: language models shift the centre of gravity of the analyst's work from formulation (1.5) towards validation (1.6) and alignment (1.7), and assumption discipline (4.5) becomes more important still, because there are now more sources of guesses.

3. THE SIX-LEVEL MODEL

3.1. The structure of the levels

Table 1. Transformation levels and their languages

Level	Transformation	Language
1	Business intent → requirements baseline	Problem
2	Requirements → functional specification	Problem
3	Specification → architecture	Solution
4	Architecture → detailed design	Solution
5	Design → code	Solution
6	Code → working system	Operational

The nature of the information in the chain changes at two jumps: between the second and third levels (from "what the system does" to "what it is made of") and between the fifth and sixth (from text to a running process). Accordingly, the levels group into three languages: the first two speak the language of the problem, the third to fifth the language of the solution, and the sixth the language of operation.

Two remarks on the structure. First, the levels are logical rather than calendar phases: the model describes the structure of the information flow, not the sequence of work in time. The levels are traversed in small segments and repeatedly (Subsections 8.1–8.3), and a baseline holds for the scope of an iteration, not for the whole product in advance (8.4). Second, the two language jumps fix only the boundaries between the three languages; the internal granularity between them (for example, separating the third and fourth levels) is a design choice justified by the step distinction criterion (4.1), but it is not the only one possible.

3.2. Characteristics of the levels

Level one. Business intent → requirements baseline. Input: business goals, the legal context, the needs of interested parties — natural language, often contradictory and incomplete. Transformation: the reduction of uncertainty through elicitation, goal refinement, formulation, validation and alignment. Output: an approved baseline of functional and non-functional requirements together with scope boundaries.

Level two. Requirements → functional specification. Transformation: decomposition into concrete system behaviour. The unified information of the requirements is broken out into four mutually complementary projections: structure (the domain model), dynamics (use cases), constraints (rules) and presentation (the interface). The level generates new information — alternative and error scenarios, edge cases that were not and could not be in the requirements, because requirements speak about the goal of success while a specification must cover failures as well. With this comes a new kind of risk — mutual inconsistency between the projections — and so the level ends with a cross-consistency check.

Level three. Specification → architecture. Transformation: systemic decisions cutting across all components — structure, technologies, the distribution of data, the security model. The rhythm of this level differs from the first two: not "collect and formalise" but "generate alternatives and choose", because architecture creates information that does not yet exist in the world — decisions. Error type: expensive to reverse, and systemic.

Level four. Architecture → detailed design. Transformation: mapping behaviour onto concrete components — interface contracts, the database schema, the interior of modules, the division of work into implementation units with acceptance criteria. Error type: local, and comparatively cheap to fix. The third and fourth levels are separated precisely because of the different cost and reversibility of

error: decisions are taken in them at a different scale, by different competences, and with a different price for being wrong.

Level five. Design → code. Transformation: implementation — formalisation carried to the end, where no freedom of interpretation remains, because the code is executed by a machine. Output: code, automated tests, a deployable artifact.

Level six. Code → working system. Transformation: composition, configuration, environments, data migrations, localisation content, deployment strategy. The error type is distinctive: "the code is correct but the system does not work" — errors of configuration, load and differences between environments, which are impossible to see at earlier levels. The necessity of this level follows from the definition of requirements itself: requirements describe the goals that working software must achieve, and the text of the code by itself achieves no goal at all. The true end point of the chain is behaviour in the production environment and the change that has taken place in the business process.

3.3. The boundaries of the model

Two potential levels are deliberately left out of the model.

A level zero (business strategy → intent) exists, but takes place outside the boundaries of software engineering: its transformations are performed by the market and the organisation, not by processing information about the system being built.

Verification is not a link in the chain, although its output ("a verified system") looks like one. Verification is the companion of every link: requirements are validated, the specification is checked for consistency, the design is reviewed, the code is tested. Including it as a level would make the model assert, wrongly, that checking happens once at the end.

4. PRINCIPLES OF PROCESS DESIGN

This section formulates the principles on which the decomposition of the levels rests and which hold for the whole chain.

4.1. The step distinction criterion

A separate step (or level) is justified only when it has its own transformation, its own kind of output and its own error type — that is, when one can go wrong in it in a way one cannot go wrong anywhere else. Decomposition without this criterion creates bureaucracy, not quality.

4.2. The error visibility principle

Designing a process is essentially designing the visibility of errors. The most dangerous errors are silent: a missed interested party, an unnoticed conflict of goals, a missed error scenario, approval without reading. All the mechanisms of the process (registries, findings lists, inventories, cross-checks, forced ranking) serve one task — to make an invisible error visible.

An important corollary of this principle: distortion errors and omission errors require different checking strategies. Distortion (a flow described wrongly) is noticed by the interested parties in review, because they hold the benchmark in memory. Omission (a missing scenario, a missing party) is not caught by review, because the deficiency is invisible — it requires systematic generation and coverage procedures.

The error types used in this paper are summarised in the table: each type is tied to the step where it typically arises and to the strategy that detects it.

Table 2. Error types, where they arise and how they are detected

Error type	Essence	Typical point of origin	Detection strategy
Incompleteness	Not all the information the source held was recorded	1.2 elicitation	Return to the source; a source coverage check against the party map
Distortion	Information written down or translated in a way that changes meaning	1.2 recording, 1.5–1.6 translation, 2.3 scenarios	Review with the parties; the back-translation method
Omission	An element is missing that nobody recorded (a party, a scenario, a factor)	1.1, 2.4, 3.1	Systematic generation and coverage checks; review does not detect it
Ambiguity and false precision	The form is tidy, but the meaning is ambiguous or unfounded	1.5 formulation	Formalisation templates; the assumption registry
Agreement without commitment	Status granted without genuine engagement	1.7 alignment	Proof of engagement instead of a signature

Incompleteness and omission are related (both are a deficiency), but the reference point differs: incompleteness is measured against the information the source holds (the source knew, but it was not recorded), omission against the reality of the domain (nobody even asked, because it was not known there was anything to ask about). Incompleteness can be detected by returning to the source; for omission there is no source, so it is covered only by systematic procedures.

4.3. Separating mandate from competence

Some transformations require knowledge (analysis, formulation, modelling), others authority (resolving conflicts, prioritisation, scope boundaries). A conflict between the goals of two parties

cannot be resolved by analysis, because it is a question of value, not of fact. Questions of competence are settled by specialists, questions of mandate by the person holding the authority.

4.4. Recording a decision with its justification

A decision without a justification is a loss of information: after some time it is impossible to tell "it was decided so for a reason" from "it just turned out that way". What is recorded: the context, the alternatives considered, the reasons for rejection, the consequences consciously accepted. This form (the decision record) holds for architectural and business-level decisions alike; it follows the practice of architecture decision records [10]. This model takes the position that a decision without alternatives considered is treated not as a decision but as a habit, because without alternatives a conscious choice cannot be told apart from inertia.

4.5. Assumption discipline

When formalisation demands a value that nobody knows, the value is not invented but registered as an assumption, with an owner and a plan for confirming it. An assumption not confirmed during validation is moved to the risk registry, not deleted. In this way the illusion of precision is kept apart from knowledge.

4.6. The author blindness principle

Practice shows that an author systematically fails to notice the gaps in their own text: reading it, they see what they meant to say rather than what is written; the effect is well known from proofreading practice, and the model adopts it as a working principle. Certain pairs of roles therefore cannot be combined in one person: the one who formulates cannot lead the validation of their own statements, and the author of a main scenario needs other eyes to generate the alternatives.

4.7. The bottleneck principle

The junctions between levels are narrow: three artifacts pass from the first level to the second, and essentially two from the second to the third. These bottlenecks are natural quality gates, because information changes hands across them, and a change of hands is the greatest point of loss. The boundaries of roles must coincide with the bottlenecks. If a lower level turns out to need the internal artifacts of a higher one, that is a signal that the bottleneck artifact is incomplete.

4.8. Fractal structure

The same pattern repeats inside the levels: prepare the frame of reference → collect or take inventory → generate content → formalise → check → consolidate. It is the same transformation sequence applied to an ever narrower kind of information. The exception is the third level, where the middle part turns from "collect and formalise" into "generate alternatives and choose".

5. THE INTERNAL STRUCTURE OF THE LEVELS

The artifact identification scheme: A<level>.<step>.<sequence number>.

The internal decomposition is given for the first three levels. The decomposition of levels four to six is left for further work; the same principles hold for it: the step distinction criterion (4.1), the fractal pattern (4.8) and the bottleneck principle (4.7).

5.1. The steps of the first level (overview)

The first level is divided into seven steps (a detailed analysis is given in the second part):

Table 3. The steps of the first level and their characteristic error types

Step	Essence	Characteristic error type
1.1 Identifying parties and sources	Who holds information and interests	A missed party, invisible to later checks
1.2 Elicitation	Tacit knowledge → recorded raw material	Incompleteness, distortion in recording
1.3 Goal refinement	Raw material → goals, separated from solutions	A solution recorded in place of a goal
1.4 Structuring and conflicts	A hierarchy, the conflict and assumption registries	An unnoticed conflict
1.5 Formulating requirements	Goals → testable statements about the system	Ambiguity, false precision
1.6 Validation	The question of truth: has it been understood correctly	A distortion in translation not caught
1.7 Alignment and the baseline	The question of will: is there commitment	Agreement without commitment; scope without clear boundaries

The steps group by the property conferred on the information: 1.1–1.2 secure its existence, 1.3–1.4 its structure, 1.5 its form, and 1.6–1.7 its status.

5.2. The steps of the second level

2.1. The domain model and the glossary. Input: A1.7.1. Concepts are extracted from the text of the requirements, synonyms are merged, and for each concept it is determined whether it is an entity, an attribute or a role; relations are drawn with their cardinalities; lifecycles are defined for the significant entities; term definitions are established. Output: A2.1.1 conceptual model, A2.1.2 lifecycles, A2.1.3 role model, A2.1.4 glossary. Error type: a wrong abstraction — the most expensive error of the level, because every later artifact rests on the model.

2.2. The interaction map. Input: A1.7.1, A2.1.3. For each requirement it is determined which actors interact with the system and with what finite goals; interaction units are defined as an "actor + goal" pair with boundaries and initiating events. Output: A2.2.1 use case inventory with traceability to requirements. Error type: a coverage gap — a forgotten interaction unit means forgotten functionality; the later steps elaborate only what is in the inventory.

2.3. Writing out the main scenarios. Input: A2.2.1, A2.1.1, A2.1.4. For each unit the success flow is written out step by step, alternating the user's and the system's perspectives and using only the terms of the glossary. Output: A2.3.1 set of main scenarios. Error type: a distorted flow — comparatively harmless, because it is visible in review.

2.4. Generating alternative and error scenarios. Input: A2.3.1. For every step of a scenario the question "what if not" is put systematically: unsuitable data, interruption, the other party not answering, a concurrent change. This is the most productive point of new information generation in the whole level. Output: A2.4.1 set of alternative and error scenarios, tied to specific steps of the main scenario.

Error type: omission — unlike 2.3, invisible, because the interested parties do not hold failure scenarios in memory. The different visibility of the errors (distortion versus omission) is the reason why 2.3 and 2.4 are two steps and not one.

2.5. Refining the rules. Input: A1.7.1, A2.1.1, A2.1.2, A2.1.3, A2.3.1, A2.4.1. Candidates are collected by scanning the sources for conditional language ("must", "cannot", "only if"), then classified, anchored and formalised to a testable form with defined behaviour in the event of a breach. Output: A2.5.1 rule catalogue. Error type: a rule without an anchor (holding "in general", and therefore nowhere), or rules that contradict one another.

A rule is defined as an invariant — a condition that must hold always, regardless of the path by which the user arrived. A scenario describes a flow, a rule a condition holding across many flows. The essential constraint: a rule can constrain only objects that already exist in the input artifacts. From this constraint exactly four rule types are derived — one for each kind of object existing by step 2.5:

Table 4. Rule types and their anchors

Type	**Constrains**	**Anchor**
Data constraint	The set of values of an attribute	A2.1.1 attributes
Data consistency	A predicate between attributes or entities	A2.1.1 relations
State machine rule	A permitted state transition	A2.1.2 transitions
Access rule	A role–action–entity triple	A2.1.3 roles, A2.3.1/A2.4.1 actions

The typology of rules is not a convention — it reproduces the ontology of the earlier outputs. The limit of the typology follows from the same fact: a rule that requires an entirely new type of object cannot be detected by the typology, because such a rule falls into no type. In this case the typology works not as a detection mechanism but as a signalling one: if formulating a rule requires an object that is not in the inputs, that is a signal of a gap in step 2.1 — the output of the model is fixed first, and only then the catalogue.

2.6. The interface specification. Input: A2.1.1, A2.1.4, A2.3.1, A2.4.1, A2.5.1. Abstract behaviour is mapped onto screens, fields, navigation and states (empty, filled, error, waiting); every field is tied to a model attribute and every validation to a rule; questions of scale are settled (pagination, search, filters, localisable texts). Output: A2.6.1 screen specifications, A2.6.2 navigation map. Error type: an element with no counterpart in the model, or a scenario step with no place in the interface.

2.7. Cross-consistency and consolidation. Input: all second-level artifacts, A1.7.1, A1.7.3. Checking runs in every direction: every scenario step has a screen, every rule a place where it fires, every state is reachable by a scenario, every requirement is covered by at least one use case, and there are no use cases without a requirement (checked against the scope boundaries). Output: A2.7.1 aligned functional specification, A2.7.2 traceability matrix. Error type: projections that are locally correct but not consistent with one another — a kind of error that arises only at this level, because at the first level the information still lived in a single projection.

5.3. The steps of the third level

3.1. Refining the architectural drivers. Input: A1.7.2, A2.7.1. What is architecturally significant is selected: load and growth scenarios, availability, security, integrations, organisational constraints. Quality attributes are turned into measurable scenarios. Output: A3.1.1 prioritised list of drivers. Error type: a driver missed or wrongly prioritised — the architecture is optimised for the wrong problem.

3.2. Generating options and analysing trade-offs. Input: A3.1.1. For every essential question at least two real options are generated and assessed against the drivers; prototypes are built for the riskiest hypotheses. Output: A3.2.1 analysis of options and trade-offs, A3.2.2 prototype results. A3.2.2 is the first empirical artifact in the chain: until now all information had been elicited from people or derived

by reasoning, and here it is obtained from experiment for the first time. Error type: tunnel vision — a single pre-selected option is the only one assessed.

3.3. Taking and recording decisions. Input: A3.2.1, A3.2.2. A choice is made and recorded with a full justification, per principle 4.4. Output: A3.3.1 set of decision records. Error type: a decision without a justification — it can neither be reviewed nor safely changed.

3.4. Defining the structure of the system. Input: A3.3.1, A2.7.1. The system is divided into components with responsibilities and boundaries; the specification is mapped onto the structure. Output: A3.4.1 component model, A3.4.2 responsibility map (a traceability artifact: specification elements → components). Error type: a component boundary drawn wrongly — it shows up slowly, as constant friction in every change.

3.5. Cross-cutting mechanisms. Input: A3.1.1, A3.4.1, A3.3.1. One-off standard answers to the questions that arise in every component: authentication and authorisation, error handling, auditing, localisation and time zones, concurrency control, versioning. Output: A3.5.1 mechanism specifications. Error type: the absence of a mechanism — every component solves the same problem in its own way, and the system becomes heterogeneous where it must be homogeneous.

3.6. Evaluating the architecture. Input: A3.1.1, A3.4.1, A3.4.2, A3.5.1. The architecture is checked against the driver scenarios: peak load, the loss of a region, security coverage; calculations and measurements are carried out. Output: A3.6.1 evaluation report, A3.6.2 risk registry. A3.6.2 is the first artifact to live longer than its own level: it is added to through the fourth, fifth and sixth levels. Error type: a "paper" architecture — internally consistent but never checked against real scenarios; skip this step and the check will happen anyway, only in the production environment.

6. THE ARTIFACT REGISTRY

Table 5. The artifact registry

ID	Artifact	Produced by	Used by
A1.1.1	Party and source map	1.1	1.2, 1.6, 1.7
A1.2.1	Raw material	1.2	1.3
A1.3.1	List of refined goals	1.3	1.4
A1.4.1	Goal hierarchy	1.4	1.5
A1.4.2	Conflict registry	1.4	1.7
A1.4.3	Assumption registry	1.4	1.5, 1.6, 1.7
A1.5.1	Draft functional requirements	1.5	1.6
A1.5.2	Draft non-functional requirements	1.5	1.6
A1.6.1	Validated set of requirements	1.6	1.7
A1.6.2	List of validation findings	1.6	1.6 (repetition criterion)
A1.7.1	Functional requirements baseline	1.7	2.1, 2.2, 2.5, 2.7
A1.7.2	Non-functional requirements baseline	1.7	3.1
A1.7.3	Scope boundaries (now/later/never)	1.7	2.7, change management
A2.1.1	Conceptual model	2.1	2.3, 2.5, 2.6, 2.7
A2.1.2	Lifecycles	2.1	2.5, 2.7
A2.1.3	Role model	2.1	2.2, 2.5, 2.7
A2.1.4	Glossary	2.1	2.3, 2.6, all levels
A2.2.1	Use case inventory	2.2	2.3, 2.7
A2.3.1	Set of main scenarios	2.3	2.4, 2.5, 2.6, 2.7
A2.4.1	Set of alternative and error scenarios	2.4	2.5, 2.6, 2.7
A2.5.1	Rule catalogue	2.5	2.6, 2.7
A2.6.1	Screen specifications	2.6	2.7
A2.6.2	Navigation map	2.6	2.7
A2.7.1	Aligned functional specification	2.7	3.1, 3.4
A2.7.2	Traceability matrix	2.7	change management, all levels
A3.1.1	List of architectural drivers	3.1	3.2, 3.5, 3.6
A3.2.1	Analysis of options and trade-offs	3.2	3.3
A3.2.2	Prototype results	3.2	3.3
A3.3.1	Set of decision records	3.3	3.4, 3.5, level 4
A3.4.1	Component model	3.4	3.5, 3.6, level 4
A3.4.2	Responsibility map	3.4	3.6, level 4
A3.5.1	Cross-cutting mechanism specifications	3.5	3.6, level 4
A3.6.1	Evaluation report	3.6	level 4
A3.6.2	Risk registry	3.6	levels 4, 5, 6 (added to)

Analysing the registry reveals three properties of artifacts that a prose description does not show:

1. Lifespan. The raw material A1.2.1 is consumed by a single step; the baseline A1.7.1 feeds five steps across two levels. The quality of the longest-living artifacts matters most.

2. Certifying artifacts (A2.7.1) do not create new information but confer the status of consistency on a set; the semantics of their dependencies differs from that of generating artifacts.
3. Traceability artifacts (A2.7.2, A3.4.2) are mappings between two worlds and the principal instrument for analysing the impact of change.

7. THE ROLE MODEL

A role is defined as the competence to transform one kind of artifact into another — a translator between two adjacent languages. Roles are derived from the artifact registry rather than postulated.

7.1. Transforming roles

Table 6. Transforming roles

Role	Level	Language pair	Core competence
Business analyst (requirements engineer)	1	The world of the parties ↔ requirements	Requirements elicitation, separating goals from solutions
Business analyst (requirements engineer, systems analyst)	2	Requirements ↔ the behaviour model	Decomposition, formalisation, systematic generation of error scenarios
Interface (UX) designer (business analyst, systems analyst, requirements engineer)	2.6	The behaviour model ↔ the user experience	Interface, localisation, accessibility
Architect (a security architect where needed)	3	Behaviour and quality attributes ↔ solution structure	Generating alternatives, trade-offs, irreversible decisions
Senior engineer (technical lead)	4	Architecture ↔ contracts and units of work	Translating structure into implementation units
Programmer	5	Design ↔ code	Final formalisation
Deployment and operations (DevOps) engineer	6	Code ↔ a working system	Configuration, migrations, deployment strategy

Two remarks on this model. First, the taxonomy of roles deliberately coincides with the traditional one: the claim to novelty is not new roles but the manner of derivation — the roles here are not postulated from a list of professions but follow from the bottlenecks of the artifact registry. Second, in real organisations roles cut across levels: the UX designer takes part in first-level validation too, and the security perspective begins in the non-functional requirements rather than at the third level. The table shows the centre of gravity of each role, not the boundaries of its activity.

7.2. Mandate and source roles

The product owner is a mandate role: they decide where authority rather than knowledge is required (the 1.7 conflicts, priorities, scope boundaries, later trade-offs). An analyst can formulate a conflict but not resolve it: a decision is an act of responsibility, not of knowledge.

The domain expert is a source role with authority: a source of information in elicitation, a validator in validation, a reviewer in second-level reviews.

7.3. The cross-cutting role

The test engineer is deliberately not assigned to any level, because verification is the companion of every link (Subsection 3.3). Their work begins at the second level: tests are derived from the rule catalogue and the scenarios while no code yet exists, and adversarial thinking is the most valuable contribution to step 2.4. That this role cannot be localised is a deliberate consequence of the decision in Subsection 3.3: treating verification as a companion rather than as a level makes the competence of verification cross-cutting as well. This is a trade-off: the model gains accuracy (checking happens everywhere) but loses the ability to assign this role to a single bottleneck.

The project manager governs not the transformation of information but its execution in time, and therefore falls outside this model.

7.4. Structural conclusions

1. The boundaries of roles coincide with the bottlenecks: handovers between roles happen through consolidated artifacts (A1.7.x, A2.7.1). This is not a convention but a consequence: bottlenecks are quality gates precisely because information changes hands across them.
2. By the author blindness principle (4.6) certain pairs of roles cannot be combined in one person: the one who formulates and the one who leads validation (1.5/1.6), the author of the main scenario and the generator of alternatives (2.3/2.4).
3. In a small team one person inevitably performs several roles; in that case it is essential to state explicitly which role is being performed at a given moment, because each role is looking for a different type of error.

8. PROCESS TOPOLOGY

8.1. Iterativity

The steps are not strictly sequential: formulation reveals gaps that send the work back to elicitation; formulating a rule reveals a missing element of the model. The direction of the information flow stays constant, but movement through the steps is iterative. Going back is normal; what is abnormal is going back unnoticed — changing an earlier artifact without updating what was derived from it.

8.2. Reducing batch size

Large batches are the medium in which silent errors grow: reading a large document attentively exceeds cognitive capacity, whatever the discipline. Batch size is reduced at two levels: the validation batch (confirmed in parts, the riskiest first) and the chain batch (a small set driven through all six levels quickly). Iterative development, seen this way, is a means of limiting the damage of uncommitted-to validation to the size of an iteration.

8.3. The minimum viable product and the feedback loop

The minimum viable product (MVP) is prioritisation with two additional constraints:

1. Coherence. The MVP is not the top of a priority list but a coherent segment: the smallest subset of requirements that together make up a complete value loop from beginning to end. Because dependencies are visible only in second-level artifacts, the boundary of the MVP is settled by iterating between step 1.7 and the second level.
2. The learning criterion. Selection is driven not by the question "what matters most" but by the question "what is the least we need in order to find out whether our hypothesis works" [16]. Because every requirement is a causal hypothesis (see Subsection 10.2), the MVP is a way of driving the riskiest hypothesis through the whole chain to sixth-level measurement at the lowest cost.

The MVP changes the topology of the chain from a line into a loop: the indicator data of the sixth level return to step 1.3 as new raw material, this time empirical; the hypotheses are corrected; the next segment goes through the chain already refined. An essential condition: an MVP without sixth-level measurement is not an MVP but merely a small deployment — the difference lies not in scope but in whether anyone is waiting for the answer to a question that was actually formulated.

8.4. Change management

A baseline freezes information not because the information is perfect but because the later levels cannot work on a moving foundation. The baseline is a contract between levels. Freezing does not mean a ban on change: a change becomes a visible event with a procedure — a request, an impact analysis through the traceability artifacts (A2.7.2, A3.4.2), a decision by the same mandate. The difference is not between "can be changed" and "cannot", but between silent drift and accountable change.

PART TWO. THE WORK OF THE BUSINESS ANALYST

9. DEFINITION OF THE ROLE

The business analyst (requirements engineer) is the transforming role of the first level. Their language pair: the world of the interested parties ↔ the language of requirements. The analyst leads all seven steps of the first level: they perform 1.1–1.5 themselves and facilitate 1.6–1.7, because in those the deciding is done by the interested parties and by the person holding the mandate.

Core competences: requirements elicitation (drawing out tacit knowledge), separating goals from solutions brought in ready-made, formalisation (narrowing language down to testable statements) and facilitation (leading validation and alignment without influencing the content).

The object of the analyst's work is the softest information in the chain: scattered across people's memories and documents, and often unspoken. The essence of the work is to confer four properties on this information, one per group of steps: existence (1.1–1.2), structure (1.3–1.4), form (1.5) and status (1.6–1.7).

10. THE CORE WORKING CONCEPTS

10.1. Goal

Definition. A goal is an indicator of a business state, its desired value, and a time profile. In short: a difference in the business state that is in principle measurable, written down as an indicator, a value and a time.

Analysis of the components:

1. The indicator — a measurable property of the business state with a scale of its own, from binary (compliance: yes/no) to quantitative (the share of late payments as a percentage). A binary scale is a full-fledged measurement scale: in the theory of measurement scales it corresponds to a nominal scale with two values [13]. The business state covers four areas: the flow of processes, their results, commitments and capabilities; a process indicator is the most common but not the only variety.
2. The desired value — one of three forms:
   - a target: the indicator equals a value or falls within an interval;
   - a limit: the indicator does not exceed a maximum or does not fall below a minimum;
   - a direction: the indicator is to be decreased or increased.

   A direction requires an ordered scale, so binary indicators cannot have one — which is why compliance goals are always written as a target. A direction is an admissible temporary form at step 1.3, but it is a debt: the formulation step must turn it into a target or a limit, or register an assumption, because no test can be derived from the direction form (the question "has it been achieved" has no answer). In practice a business rarely seeks to maximise — it seeks a sufficient level, so a limit is the natural final form.
3. The time profile — when the indicator must have the desired value: reached by a moment (measured once), improved continuously (measured periodically), maintained always (measured continuously or audited). Kinds of goal differ not in measurability but in time profile and in the coarseness of the scale.

The three components work as a formalisation template: an unfilled field shows what is still unknown, and is registered in the assumption registry A1.4.3.

Tests for recognising a goal:

- The question "why" must lead to a higher goal; if there is no answer, what has been recorded is a habit.
- The question "how" must have more than one answer; if there is only one, what has been recorded is a solution.

Examples (the invoice management domain):

Table 7. Examples of kinds of goal in the invoice management domain

Kind	**Example**
Achievement	An invoice is sent no later than the next working day after the service is provided
Optimisation	The share of late payments is reduced from 30 to 10 per cent by the end of the year
Maintenance	At any moment it is possible to establish who approved each invoice and when
Compliance	Invoice numbering and content comply with the VAT law (target "yes", profile "always")
Control	Nobody can approve an invoice they created themselves

A note: a control goal is not yet a rule, even though the phrasing is similar. It will become a rule (A2.5.1, anchored to roles and actions) at the second level, when the objects that can be constrained exist; at the first level it is a business intent.

10.2. Requirement

Definition. A requirement is a testable statement about the system: unambiguous, atomic, traceable to a goal.

Three essential properties:

1. A change of subject. A goal speaks about the business ("delays under 10 per cent"), a requirement about the system ("the system sends a reminder 3 days before the due date"). Up to the formulation step the information in the chain speaks about the world; from that step on, about the system.
2. The nature of a hypothesis. A requirement is not derived from a goal — it is a bet on the goal [15]. From the statement "delays under 10 per cent" it does not deductively follow that reminders should be sent: that is a causal hypothesis, that reminders will encourage payment on time. A requirement may be implemented perfectly and the goal still not reached. One goal usually splits into several hypotheses. The traceability link records which hypothesis a requirement implements, so that at the sixth level, when the indicator fails to move, it is visible which hypotheses failed and what to change. Formulation can be described as translating from the language of wishes into the language of commitments.
3. Separating functional from non-functional requirements. A functional requirement states what the system does, a non-functional one how well. They are held in separate artifacts because they travel by different channels: the functional ones feed the second level (the behaviour model), the non-functional ones the third (the architectural drivers). Joined together, they would clog the bottleneck.

10.3. Distinguishing the concepts

Table 8. Telling goal, requirement and solution apart

Concept	Speaks about	Example
Goal	The business (a desired state)	Management has a monthly sales summary by customer
Requirement	The system (behaviour)	The system allows a summary to be produced by customer and by period
Solution	The implementation (a mechanism)	A spreadsheet export

A typical error: a solution brought in ready-made by an interested party is recorded as a goal, and the solution space is thereby locked for every level that follows.

11. ANALYSIS OF THE STEPS

11.1. Identifying parties and sources (1.1)

Purpose: to establish who holds information and interests at all, before starting to collect information.

– Input: the initial initiative (external to the chain).

– Transformation: roles, affected groups, documents, legal acts and existing systems are identified; for each party it is recorded what information they hold, what their interest is, and what part they play in validation.

– Output: A1.1.1 party and source map.

– Error type: a missed party — the most expensive error of the level: the information of a missed party will silently not exist at every later level, and no later check will show this, because only recorded information can be checked.

Quality criteria: the map contains not only the parties who want something but also the affected ones (who asked for nothing, but whose work the system will change); it includes not only people but documentary sources as well (legal acts, the behaviour of existing systems); where the scope is global, the different regions whose practices may differ are represented.

11.2. Elicitation (1.2)

Purpose: to turn tacit and scattered knowledge into recorded knowledge.

– Input: A1.1.1.

– Transformation: interviews, workshops, document analysis, observation of the work as it is done today [7]; information is not assessed or filtered — priority goes to completeness.

– Output: A1.2.1 raw material (notes, quotations, process descriptions).

– Error type: incompleteness and distortion in recording — what is written down is not what was said, or what the interlocutor will not volunteer is never asked about.

Working techniques:

– Ask about situations, not about wishes: a request to describe the last month-end close yields more information than a question about what one would like from the system.

– Observation at the actual workplace reveals what an interview does not: people do not talk about what is self-evident to them.

– Quotations are recorded verbatim; interpretation happens at the next step.

11.3. Goal refinement (1.3)

Purpose: to extract goals from the raw material, separating them from solutions brought in ready-made.

– Input: A1.2.1.

– Transformation: the question "why" is used to climb until a statement stops being a mechanism and becomes a desired business state; variants of the same goal from different parties are merged; each goal is written to the template indicator + value + time profile; unfilled fields are registered in A1.4.3.

– Output: A1.3.1 list of refined goals.

– Error type: a solution recorded in place of a goal.

An example of the transformation. Recorded in the raw material: the accountant would like a spreadsheet export. The question "why" reveals: the reports for management are prepared in a

spreadsheet. The question "why" again: management needs a monthly sales summary by customer. The goal: "Management has a monthly sales summary by customer." The spreadsheet export remains one possible solution, without locking out the alternatives (an integrated report, automatic sending).

11.4. Structuring goals and identifying conflicts (1.4)

Purpose: to arrange the goals into a structure and make the conflicts visible.

- Input: A1.3.1.
- Transformation: a hierarchy by "why" relations (which goal serves which), dependencies, and conflicts recorded where the goals of the parties meet. Conflicts are only identified: their resolution is deferred to step 1.7, because it requires mandate rather than competence.
- Output: A1.4.1 goal hierarchy, A1.4.2 conflict registry, A1.4.3 assumption registry.
- Error type: an unnoticed conflict — two parties live with incompatible expectations right up to deployment.

Quality criteria: every goal has a place in the hierarchy (a goal with no higher goal is either strategic or questionable); conflicts are written down neutrally, naming both parties and both goals — the registry does not settle who is right; where the scope is global, the compatibility of the goals of different regions is checked systematically, because that is the most common origin of conflicts.

11.5. Formulating requirements (1.5)

Purpose: to turn goals into testable statements about the system.

- Input: A1.4.1, A1.4.3.
- Transformation: three jobs happen at once:
    3. A change of subject — from statements about the business to statements about the behaviour of the system.
    3. Formulating hypotheses — every requirement is a causal bet that a particular system behaviour will push the goal's indicator; the requirement → goal link is recorded as a matter of obligation.
    3. Narrowing the language, by three operations:
        - Unambiguity. Every undefined word ("on time", "quickly", "large") is replaced by a concrete value; a value that is not known is not invented but registered among the assumptions, with an owner and a plan for confirming it. This is the difference between precision and false precision: a value is written down in both cases, but in the case of an assumption it is marked as unfounded.
        - Atomicity. One statement, one requirement. The test: can the statement be rejected or deferred without touching anything else; if not, it is split. A non-atomic requirement cannot be addressed: it cannot have a single identifier, status, priority and test.
        - Closing the directions. A goal in the form "reduce X" is turned into a target or a limit; a limit that is not known is registered among the assumptions. At the level of a requirement the direction form is not admissible, because no test can be derived from it.
- Output: A1.5.1 draft functional requirements, A1.5.2 draft non-functional requirements.
- Error type: ambiguity and false precision — the form is tidy, but the meaning is ambiguous or invented.

Quality criteria: every requirement has an identifier and traceability and is testable (a test can be written while no code exists); functional and non-functional requirements arising from the same goal are separated (for example, "send a reminder 3 days before" is functional, while "the reminder is sent

no later than one hour after the event, peak load included" is non-functional); there are no requirements without a goal — such requirements are returned for investigation or rejected.

11.6. Validation (1.6)

Purpose: to answer the epistemological question of whether the statements formulated correspond to the intentions of the parties.

– Input: A1.5.1, A1.5.2, A1.1.1 (who validates), A1.4.3 (assumptions for confirmation).

– Transformation: three jobs:

    3. A coverage check against A1.1.1 — every requirement is validated by the party whose goal it serves and by the parties it affects. If a party was missed at step 1.1, the error passes silently through validation as well.

    3. Review by the back-translation method (see below).

    3. Confirming the assumptions — every assumption becomes either knowledge or a consciously accepted risk; an assumption not confirmed travels on as an unresolved risk.

– Output: A1.6.1 validated set of requirements, A1.6.2 list of findings.

– Error type: a distortion in translation not caught.

What is specific about validation. In informational terms this is a peculiar step: an artifact is compared against a benchmark that is written down nowhere — the intentions exist only in the memory of the parties. Validation therefore cannot be automated, nor performed without the very people the chain began with.

The paradox of formalisation. At the formulation step the language was narrowed so that the statements would become precise, but in the process they became hard to recognise for the parties whose intentions they express. The better the formulation, the harder the validation: precision and intelligibility to the parties pull in opposite directions.

The back-translation method. From the paradox follows the main rule: the parties must not simply be handed a document to read — formal statements are translated back into their language and, better still, into consequences. Instead of asking whether a requirement is correct, a situation is put to them: the invoice has been sent, the customer has paid half the amount, three days remain until the due date — the system will send a reminder about the balance; is that how it should be. People assent to text but react to situations: a party notices that the partial payment case is wrong only when faced with a concrete consequence. Validation can be described as an experiment: the hypothesis is "we understood correctly", and the party's surprise is the falsifying result.

Techniques (all of them work by the principle of consequences): concrete examples with real data, walking through a scenario out loud, paper mock-ups and prototypes, test cases reformulated as questions, a request to retell in one's own words.

The inverted success criterion. A validation session that found not a single discrepancy is to be treated not as a success but as a warning: healthy validation always finds something. An empty findings list A1.6.2 means that validation did not take place, and it is repeated by another method. This criterion makes non-engagement visible.

Typical failure scenarios and the countermeasures:

Table 9. Typical validation failure scenarios and countermeasures

Failure scenario	**Countermeasure**
Polite assent (approving because it is awkward to admit not understanding)	A request to retell; a question about what will happen in a concrete situation
Fatigue validation (a large scope in a short session)	Validate in parts, the riskiest first

Failure scenario	Countermeasure
Leading questions	The session is led by someone other than the author; questions are put openly

11.7. Alignment and the baseline (1.7)

Purpose: to answer the question of will, whether there is commitment to the set; the information is legitimised — it acquires the status of agreement.

– Input: A1.6.1, A1.4.2, A1.4.3 (the remaining risks), A1.1.1 (party coverage).

– Output: A1.7.1 functional requirements baseline, A1.7.2 non-functional requirements baseline, A1.7.3 scope boundaries.

– Roles: the product owner decides (mandate), the analyst facilitates and records.

– Error type: formal agreement without real commitment, or scope without clear negative boundaries.

Resolving conflicts. A conflict between the goals of parties is a question of value, not of fact, and is therefore settled by mandate. Every decision is recorded in the form of a decision record: context, alternatives, reasons for the choice, consequences accepted. Without the record the discussion will start again from the beginning after some time; with it, from the question of whether the circumstances have changed.

Prioritisation. Two principles matter more than any particular method. First, a priority is a decision, not a property: it is assigned by mandate, not by analysis. Second, priorities must be relative: a list in which most items are critical is informationally empty, because the purpose of prioritisation is an order of abandonment — an answer to the question of what is sacrificed first when resources run short. An effective technique is forced ranking: not marking the important ones, but ranking them all; ranking forces comparison, and comparison reveals the true values.

Scope boundaries. A1.7.3 has three values: now / later / never. A clear "no" is a commitment equal to a "yes", and protects against silent expectations. The list of goals not included is cross-checked against the party map: every party must see the fate of its goals, rejected ones included, now rather than at deployment. The boundary between "now" and "later" corresponds to the MVP decision (Subsection 8.3) and is recorded with its justification: which hypothesis is being tested first, and why.

The act of the baseline. Freezing is a contract between levels: the first level commits to the stability of the statements, the second to working from exactly those statements. After the baseline, a change becomes a visible event with a procedure (Subsection 8.4).

12. PREVENTING SILENT ERRORS

The most dangerous error of the first level is a baseline formally approved but never read attentively: it silently poisons every level that follows and shows up only at the sixth, when the indicators fail to move. By comparison, the opposite error (validated requirements that are not committed to) stops the project loudly and early, and is therefore cheaper. Three process levers against the silent variant:

1. Changing the content of approval. From a passive act (a signature) to proof of engagement: answers to situational questions, retelling, walking through a scenario. The findings list A1.6.2 with its inverted success criterion makes non-engagement visible.
2. Reducing batch size. Reading a large document attentively exceeds cognitive capacity whatever the discipline, so validation and confirmation happen in parts, the riskiest first.
3. Moving concreteness forward. People react to the concrete, and concreteness in the chain naturally grows level by level. Every bottleneck is therefore a re-validation gate: screen mock-ups (A2.6.1) are shown to the same parties, and fifth-level demonstrations are given to the business representatives rather than to the team. A more radical form is a prototype built before the baseline, specifically for the purpose of validation: in informational terms this is the production of a throwaway artifact, but it is the cheapest way of forcing silent ignorance to show itself.

An honest limitation: a process can force a demonstration of understanding, but it cannot force sincere commitment — that is a problem of the organisation, not of information transformation. What the process can do is limit the damage: doubts about commitment are recorded in the risk registry with owners, so that it cannot later be claimed that nobody could have known.

13. THE ANALYST'S WORK BEYOND THE FIRST LEVEL

The analyst's work does not end with handing over the baseline:

– Second-level reviews: checking whether the specification has distorted the meaning of the requirements; taking part in the 2.7 coverage check.

– Re-validation gates: organising the showing of mock-ups and demonstrations to the interested parties.

– Change management: change requests are assessed against the scope boundaries, the impact analysis is led through the traceability artifacts, and the decision is put to the mandate.

– The sixth-level loop: after deployment the goal indicators are watched; their data return to goal refinement as empirical raw material; failed hypotheses are identified through traceability and returned to formulation.

14. QUALITY GATES: THE HANDOVER TO THE SECOND LEVEL

A baseline is held to be a reliable bottleneck if all the criteria are satisfied.

Completeness of content:

1. Every goal is written to the full template (indicator, value, time profile), or the gap is registered among the assumptions.
2. Every requirement is atomic, unambiguous and testable, with an identifier and traceability to a goal.
3. There are no requirements without a goal and no goals without a party.
4. Functional and non-functional requirements are separated.

Completeness of decisions:

1. Every conflict in A1.4.2 has a resolution with a justification.
2. Every assumption in A1.4.3 is confirmed or moved to the risks with an owner.
3. Priorities are relative and ranked.
4. The scope has three values, and the boundary between "now" and "later" is justified.

Completeness of process:

1. Every requirement has been validated by the responsible party per A1.1.1.
2. The findings list A1.6.2 is not empty.
3. Every party has seen the fate of its goals, rejected ones included.

If all the criteria are satisfied, A1.7.1–A1.7.3 is a reliable foundation on which the second level can build without repeated returns. If not, the baseline is merely a document with signatures, and the difference will show up at the sixth level. Errors of the first level cannot be repaired at the second, only inherited, and these gates are therefore mandatory rather than advisory.

15. CONCLUSIONS

1. It is productive to examine software development as a chain of information transformation, in which each level is defined by its input, transformation, output and its own error type. The chain consists of six levels with two jumps of language: from the language of the problem to that of the solution (between specification and architecture) and from text to a running process (between code and a working system).
2. The chain is irreducible, because its purpose is not to translate the information available but to produce what is missing: each level takes decisions that were not present at the previous one. Instruments that improve notation reduce only accidental complexity; the essential complexity (deciding what is wanted) remains. This thesis holds for development with large language models too: they compress the work of formalisation but move the production of missing decisions into dialogue, and turn questions left unasked into silent guesses (Subsection 2.6).
3. The interior of the levels shows a recurring structure: prepare the frame of reference → collect → generate → formalise → check → consolidate. Steps are distinguished by a uniform criterion — their own error type; the distinction between distortion errors (visible in review) and omission errors (requiring systematic generation and coverage procedures) is particularly important.
4. An artifact registry with identifiers and dependencies reveals structural properties that a prose description does not show: the lifespan of artifacts, the bottlenecks between levels, and the differing nature of artifacts (generating, certifying, traceability, empirical). The role model is derived from the registry: a role is a translator between adjacent languages, and the boundaries of roles coincide with the bottlenecks.
5. The core of the business analyst's work is four operations: turning tacit knowledge into recorded knowledge, separating goals from solutions brought in ready-made, narrowing language down to testable statements, and changing the status of information (validation and alignment). Defining a goal through an indicator, a value and a time profile provides a formalisation template, while the conception of a requirement as a causal hypothesis ties the first level to sixth-level measurement in a feedback loop.
6. The most dangerous errors of the process are the silent ones, and the most expensive of them is a baseline formally approved but never internalised. The levers of prevention (proof of engagement instead of a signature, reducing batch size, moving concreteness forward) are variants of a single principle: make the invisible error visible.
7. The status of the work is a conceptual hypothesis (Subsection 1.3), but some of the instruments are applicable on their own, without waiting for the whole model to be tested: the goal template (indicator, desired value, time profile) together with the assumption registry, the table of error types (4.2), and the inverted success criterion for validation (11.6). Directions for further work: the internal decomposition of levels four to six, and a retrospective test of the model's predictions against defect data from real projects.

APPENDIX A. A WALK THROUGH THE FIRST LEVEL: THE REMINDER REQUIREMENT

The purpose of the appendix is to show the whole first-level chain working on a single small but complete case. The domain is the same as in the examples in the main text: invoice management. The initial initiative: the company's management is unhappy that customers are late in paying invoices.

A.1. Identifying parties and sources (1.1)

A fragment of the A1.1.1 map:

Table 10. A fragment of the party and source map

Party / source	Information held	Interest	Part in validation
Accountant	The actual flow of payment tracking, the current manual process	Less manual work	Validates the flow of the process
Sales manager	Relations with customers	Not to damage relations with customers	Validates communication to customers
Management (via the product owner)	Business priorities	Working capital flow	Mandate: conflicts, scope
Customers	How incoming invoices are handled on their side	Asked for nothing; an affected party	Indirectly (via the sales manager)
The VAT law	Requirements on invoice content and numbering	A documentary source	The benchmark for compliance
The existing accounting system	The actual data structure of invoices and payments	A documentary source	The benchmark for behaviour

A note on the error type: customers are a typical missed party — they ask for nothing, but reminders will change their experience. If they are not missed here, a validation question about the tone of communication will arise at step 1.6 that would not otherwise have come up.

A.2. Elicitation (1.2)

A fragment of the A1.2.1 raw material (quotations verbatim):

– Accountant: "Every other week I sit down and go through by hand which invoices are paid and which are not. The ones that are late, I phone or write to. I would like the system to send the reminders itself."

– Sales manager: "Just don't start bombarding customers with letters. The big customers don't like it, we arrange things with them individually."

– Observation at the workplace: the accountant marks late invoices in a separate spreadsheet; cases of partial payment she settles differently each time.

– From the existing system: the payment due date is stored with the invoice; a partial payment state exists.

Everything is recorded, including the solution brought in by the accountant ("the system to send the reminders itself") — assessing it happens at the next step.

A.3. Goal refinement (1.3)

The question "why" put to the accountant's wish: why reminders? So that customers pay on time. Why does on time matter? Management's answer: late payments hold up working capital. The statement has stopped being a mechanism and become a desired business state.

The goal to the template (indicator + value + time profile): "The share of late payments is reduced from 30 to 10 per cent by the end of the year" (indicator: the share of late payments; value: a limit of 10 per cent; profile: reached by a moment, maintained thereafter). Sending reminders automatically remains one possible solution, without locking out the alternatives (invoicing in advance, changing the payment terms).

An assumption is registered in A1.4.3: the current figure of 30 per cent is taken from the accountant's verbal estimate and needs confirming from the accounting system's data (owner: the analyst; plan: a report from the existing system).

A.4. Structuring and conflicts (1.4)

A1.4.1: the goal is placed in the hierarchy beneath the higher goal "improve working capital flow" (strategic, outside the chain).

A1.4.2, a conflict entry (neutral, both parties and both goals): the accountant's goal "payments on time, less manual work" and the sales manager's goal "not to damage relations with the big customers" collide on the question of the frequency and reach of the reminders. The registry does not settle who is right; the resolution is deferred to 1.7.

A second assumption is added to A1.4.3: an emailed reminder reaches the person responsible for payment on the customer's side (owner: the sales manager; plan: check with the five largest customers).

A.5. Formulating requirements (1.5)

The change of subject and the narrowing of language. From the goal a causal hypothesis is formulated: a reminder before the due date will encourage payment on time. The undefined words are closed: "before the due date" becomes "3 days before" (the value suggested by the accountant on the basis of the typical bank transfer cycle; registered as an assumption until confirmed by data on customer behaviour).

– A fragment of A1.5.1 (functional): "FR-14. The system sends a reminder to the invoice contact address 3 days before the payment due date, if the invoice is unpaid or partly paid. Traceability: goal G-03."
– A fragment of A1.5.2 (non-functional, separated from the functional one): "NFR-06. The reminder is sent no later than one hour after the scheduled moment, peak load included. Traceability: G-03."

The atomicity test: FR-14 can be rejected or deferred without touching the other requirements; the statement "the system sends reminders and shows a report of late invoices" would be split into two.

A.6. Validation (1.6)

The back-translation method. A situation is put to the accountant: "The invoice has been sent, the customer has paid half the amount, three days remain until the due date — the system will send a reminder about the balance. Is that how it should be?" The accountant is surprised: a partial payment usually means an agreed payment schedule, and in that case a reminder is unwanted. She had assented to the text; she reacted to the situation.

A fragment of the A1.6.2 findings list: "R-01. FR-14 is inaccurate in the case of partial payment: if a partial payment has an agreed schedule, no reminder is sent. The requirement is returned to 1.5 for refinement." The list is not empty, so by the inverted success criterion validation did take place.

Confirming the assumptions: the figure of 30 per cent is confirmed from the accounting data (the assumption becomes knowledge); the assumption about reaching the right addressee on the customer's side is not fully confirmed and is moved to the risks with an owner.

A.7. Alignment and the baseline (1.7)

Resolving the conflict by mandate. The product owner, with both parties present, decides: one reminder is sent before the due date; for big customers marked as "individually managed", no reminders are sent. The decision record: context (the conflict between the accountant's and sales's goals), alternatives (reminders for everyone; no reminders at all; a segmented scheme), the reasons for the choice, the consequence accepted (part of the delays will remain uncovered by reminders).

Scope boundaries A1.7.3: now — the reminder before the due date and the partial payment exception; later — escalation reminders after the due date; never — automated calls to customers. Every party sees the fate of its goals: the accountant sees that escalation has been deferred, and that this is recorded rather than merely assumed.

The baseline A1.7.1 with the refined FR-14 is approved; the boundary between "now" and "later" is justified by MVP logic: the first hypothesis to be tested is that a reminder before the due date will move the indicator. At the sixth level the share of late payments will be measured, and if the indicator does not move, traceability will show which hypothesis failed.

In summary. Every step produced information that was not in its input: the map (who knows anything at all), the raw material (what was said), the goals (why), the hierarchy and the conflicts (how the goals interact), the requirements (what the system will do), the findings (where the understanding differed), the baseline (what is committed to). Had any step been skipped, the corresponding information would have been produced by someone later — most often by a programmer, silently, in the shape of an edge case.

REFERENCES

1. Brooks, F. P. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, 20(4), 1987.
2. Naur, P. Programming as Theory Building. Microprocessing and Microprogramming, 15(5), 1985.
3. Wiegers, K., Beatty, J. Software Requirements. 3rd ed. Microsoft Press, 2013.
4. Sommerville, I. Software Engineering. 10th ed. Pearson, 2016.
5. Robertson, S., Robertson, J. Mastering the Requirements Process: Getting Requirements Right. 3rd ed. Addison-Wesley, 2012.
6. ISO/IEC/IEEE 29148:2018. Systems and software engineering – Life cycle processes – Requirements engineering.
7. Gause, D. C., Weinberg, G. M. Exploring Requirements: Quality Before Design. Dorset House, 1989.
8. Gotel, O., Finkelstein, A. An Analysis of the Requirements Traceability Problem. Proceedings of the First International Conference on Requirements Engineering, 1994.
9. IEEE Computer Society. SWEBOK Guide: Guide to the Software Engineering Body of Knowledge. v4.0, 2024 (the configuration management and baseline sections).
10. Nygard, M. Documenting Architecture Decisions. 2011. Available at: cognitect.com/blog/2011/11/15/documenting-architecture-decisions.
11. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.
12. Cockburn, A. Writing Effective Use Cases. Addison-Wesley, 2001.
13. Stevens, S. S. On the Theory of Scales of Measurement. Science, 103(2684), 1946.
14. Shannon, C. E. A Mathematical Theory of Communication. Bell System Technical Journal, 27, 1948.
15. Zave, P., Jackson, M. Four Dark Corners of Requirements Engineering. ACM Transactions on Software Engineering and Methodology, 6(1), 1997.
16. Ries, E. The Lean Startup. Crown Business, 2011.
17. van Lamsweerde, A. Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, 2009.