
# Fundamentals of RE

## Chapter 1

## Setting the Scene



## Setting the scene: outline

- What is Requirements Engineering (RE) ?
  - The problem world & the machine solution
  - The scope of RE: the WHY, WHAT and WHO dimensions
  - Types of statements involved: descriptive *vs.* prescriptive
  - Categories of requirements: functional *vs.* non-functional
  - The requirements lifecycle: actors, processes, products
  - Target qualities and defects to avoid
  - Types of software projects
  - Requirements in the software lifecycle
  - Relationship to other disciplines

## Setting the scene: outline (2)

- Why engineer requirements?
  - The requirements problem: facts, data, citations
  - Role and stakes of Requirements Engineering
- Obstacles to good RE practice
- Agile development and RE




## The problem world and the machine solution

- To make sure a software solution "correctly" solves some real-world problem, we must first fully **understand** and **define** ...
  - **what problem** needs to be solved in the real world
  - the **context** in which the problem arises
- Example: car control
  - **Problem**: manual handbrake release can be inconvenient in certain situations
  - **Context**: car driving, braking, driver 's intent, safety rules, ...

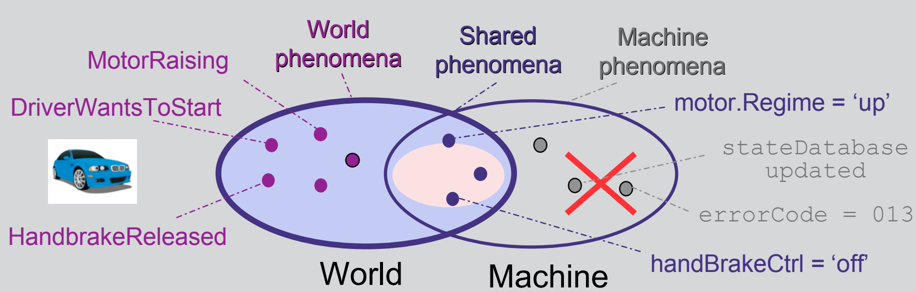
## The problem world and the machine solution (2)

- **World**: problematic part of the real-world, made of
  - human components: organization units, staff, operators, ...
  - physical components: devices, legacy software, mother Nature, ...
- **Machine**: what needs to be installed to solve the problem
  - software to be developed and/or purchased
  - hardware/software implementation platform, associated input/output devices (e.g. sensors & actuators)
- Requirements engineering (RE) is concerned with ...
  - the desired machine's **effect on the problem world**
  - the **assumptions** and **relevant properties** about this world




## The problem world and the machine solution (3)

- The world and the machine have their own phenomena while sharing others
- RE is solely concerned with **world** phenomena, including shared ones [Jackson95]
  - unlike software design, concerned with machine phenomena

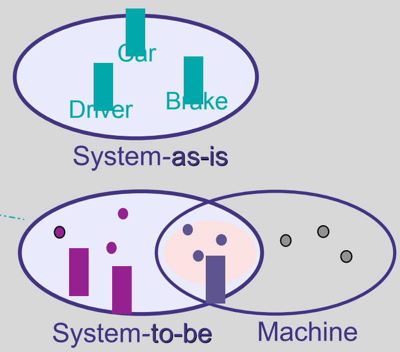


## The problem world involves two system versions

- **System**: set of interacting components structuring the problem world
- System-**as-is**: system as it exists before the machine is built into it
- System-**to-be**: system as it should be when the machine will operate into it

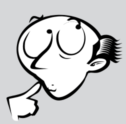
Concepts, phenomena, rules about car handbraking

Concepts, phenomena, rules about *automated* handbraking









## RE: a preliminary definition

**Coordinated** set of **activities** ...

- for **exploring**, **evaluating**, **documenting**, **consolidating**, **revising** and **adapting** the **objectives**, **capabilities**, **qualities**, **constraints** & **assumptions** on a software-intensive **system**
- based on **problems** raised by the system-**as-is** and **opportunities** provided by new technologies

## What others said ...



- Requirements definition must say ...
  - **why** a new system is ne... ...rrent or foresee... ...ditions...
  - **what** sy... ...ntext,
  - **how** the... ...d





- RE is conc... ...real-world **goals** for, **functions** of, **constr**... ...ftware systems; and with their
  - **link** to ...ise **specifications of software behavior**,
  - **evolution** over time and families

 9

## **System** requirements vs. **software** requirements

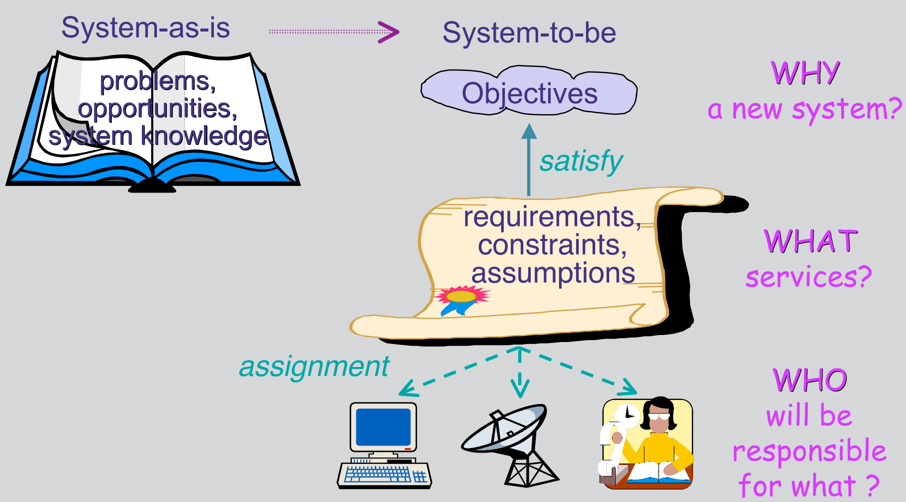
- **Software-to-be**: software to be developed - part of the machine, component of the system-to-be
- **Environment**: all other components of the system-to-be, including people, devices, pre-existing software, etc.
- **System requirements**: what the *system*-to-be should meet; formulated in terms of phenomena in the environment

  "The handbrake shall be released when the driver wants to start."
- **Software requirements**: what the *software*-to-be should meet on its own; formulated in terms of phenomena **shared** by the software and the environment

  "The software output variable *handBrakeCtrl* shall have the value *off* when the software input variable *motorRegime* gets the value *up*."

 10

## The scope of RE: the *WHY, WHAT, WHO* dimensions





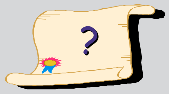


## The WHY dimension

- Identify, analyze, refine the system-to-be's **objectives**
  - to address analyzed deficiencies of the system-as-is
  - in alignment with business objectives
  - taking advantage of technology opportunities
- Example: airport train control

  "Serve more passengers"

  "Reduce transfer time among terminals"
- Difficulties
  - *Acquire domain knowledge*
  - *Evaluate alternative options* (e.g. alternative ways of satisfying the same objective)
  - *Match problems-opportunities*, and evaluate these: implications, associated risks
  - *Handle conflicting objectives*



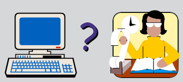

## The WHAT dimension

- Identify & define the system-to-be's **functional services** (software services, associated manual procedures)
  - to satisfy the identified objectives
  - according to quality constraints: security, performance, ...
  - based on realistic assumptions about the environment
- Example: airport train control

  "Computation of safe train accelerations"

  "Display of useful information for passengers inside trains"
- Difficulties
  - Identify the right set of features
  - Specify these precisely for understanding by all parties
  - Ensure backward traceability to system objectives



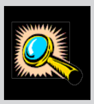

## The WHO dimension

- Assign responsibilities for the objectives, services, constraints among system-to-be components
  - based on their capabilities and on the system's objectives
  - yielding the software-environment boundary
- Example: airport train control
  - "Safe train acceleration" ... under direct responsibility of software-to-be (driverless option) *or* of driver following software indications ?
  - "Accurate estimation of train speed/position" ... under responsibility of tracking system *or* of preceding train ?
- Difficulties
  - Evaluate alternative options to decide on the right degree of automation

## Setting the scene: outline

- What is Requirements Engineering?
  - The problem world & the machine solution
  - The scope of RE: the WHY, WHAT and WHO dimensions
  - Types of statements involved: descriptive *vs.* prescriptive
  - Categories of requirements: functional *vs.* non-functional
  - The requirements lifecycle: actors, processes, products
  - Target qualities and defects to avoid
  - Types of software projects
  - Requirements in the software lifecycle
  - Relationship to other disciplines



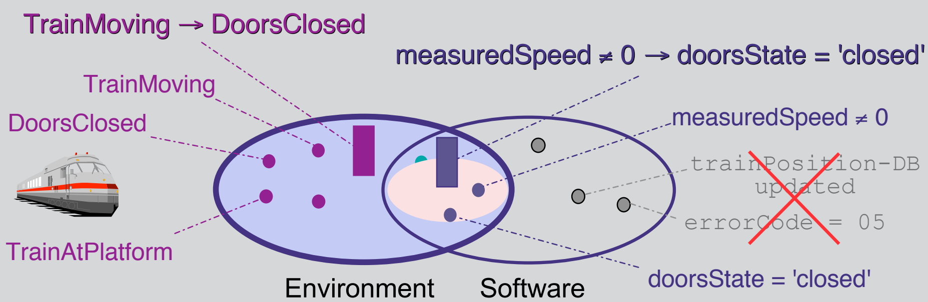



## Statement Types

- **Descriptive** statements state system properties holding regardless of how the system should behave
  - natural law, physical constraint, etc
  - e.g. "If train doors are closed, they are not open"
    "If the train's acceleration is positive, its speed is non-null"
- **Prescriptive** statements state desirable properties holding or not depending on how the system behaves
  e.g. "Doors shall always remain closed when the train is moving"
- Important distinction for RE:
  - prescriptive statements can be negotiated, weakened, replaced by alternatives
  - descriptive statements cannot

## Statements may differ in scope

- A RE statement may refer to phenomena ...
  - owned by the environment
  - or shared between the environment & the software-to-be: one **controls** phenomena **monitored** by the other, and resp.







## Types of statements: system requirements, software requirements

- **System requirement**: *prescriptive* statement referring to *environment* phenomena (not necessarily shared)
  - to be enforced by the software-to-be possibly together with other system components
  - formulated in a vocabulary understandable by all parties

TrainMoving → DoorsClosed

- **Software requirement**: *prescriptive* statement referring to *shared* phenomena
  - to be enforced by the software-to-be solely
  - formulated in the vocabulary of software developers

measuredSpeed ≠ 0 → doorsState = 'closed'

(A software req is a system req; the converse is not true)

## Types of statements: domain properties, assumptions, definitions

- **Domain property**: *descriptive* statement about problem world phenomena (holds regardless of any software-to-be)

  trainAcceleration > 0 → trainSpeed ≠ 0

- **Assumption**: statement to be satisfied by the environment of the software-to-be
  - formulated in terms of environment phenomena
  - generally prescriptive (e.g. on sensors or actuators)

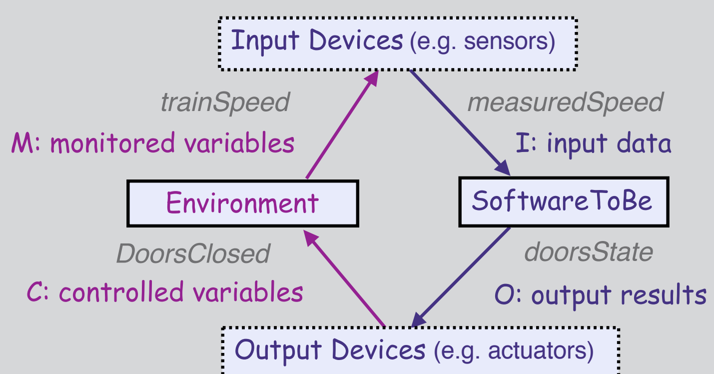
  measuredSpeed ≠ 0 **iff** trainSpeed ≠ 0

- **Definition**: statement providing a precise meaning to system concepts or auxiliary terms
  - no truth value

  "measuredSpeed is the speed estimated by the train's speedometer"




## Relating **software** reqs to **system** reqs: the 4-variable model [Parnas95]




$SysReq \subseteq M \times C$ relation on environment monitored/controlled variables

$SofReq \subseteq I \times O$ relation on software input/output variables

SofReq = *Map* (SysReq, Dom, Asm)

translates SysReq using domain properties and assumptions

## Mapping system reqs to software reqs involves satisfaction arguments

SOFREQ, ASM, DOM |= *SysReq*

"**If** the software requirements in SOFREQ, the assumptions in ASM and the domain properties in DOM are all satisfied and consistent, **then** the system requirements *SysReq* are satisfied"

SofReq: measuredSpeed ≠ 0 → doorsState = 'closed'

ASM: measuredSpeed ≠ 0 **iff** trainSpeed ≠ 0
doorsState = 'closed' **iff** DoorsClosed

Dom: TrainMoving **iff** trainSpeed ≠ 0

---

*SysReq:* TrainMoving → DoorsClosed

*Further to requirements, we need to elicit, evaluate, document, consolidate relevant assumptions & domain properties*



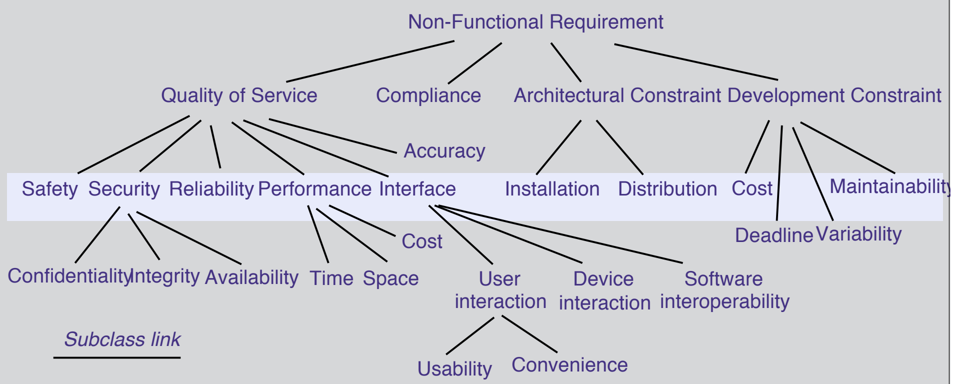

## Categories of requirements

- **Functional requirements:** prescribe what services the software-to-be should provide
  - capture intended software effects on environment, applicability conditions
  - units of functionality resulting from software operations

  "The software shall control the acceleration of all trains"
- **Non-functional requirements:** constrain how such services should be provided
  - **Quality** requirements: safety, security, accuracy, time/space performance, usability, ...
  - Others: compliance, architectural, development reqs
  - To be made precise in system-specific terms

  "Acceleration commands shall be issued every 3 seconds to every train"

## A taxonomy of non-functional requirements

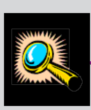



- See definitions and examples in the book
- No clear-cut boundaries, possible overlaps
  - Functional/non-functional: *e.g.* functional reqs for firewall management are security-related
  - Non-functional overlaps: *e.g.* "high frequency of train commands" is related to performance and safety



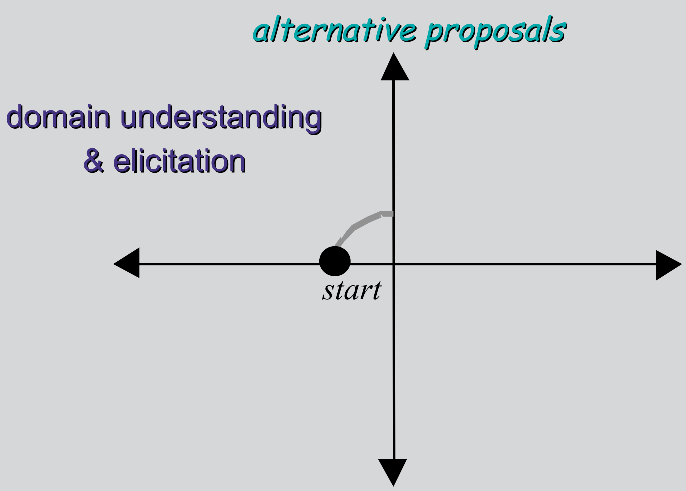

## Setting the scene: outline

- What is Requirements Engineering?
  - The problem world & the machine solution
  - The scope of RE: the WHY, WHAT and WHO dimensions
  - Types of statements involved: descriptive *vs.* prescriptive
  - Categories of requirements: functional *vs.* non-functional



  - **The requirements lifecycle: actors, processes, products**
  - Target qualities and defects to avoid
  - Types of software projects
  - Requirements in the software lifecycle
  - Relationship to other disciplines

## The RE process (1)







## Domain understanding

- Studying the system-as-is
  - Business organization: structure, dependencies, strategic objectives, policies, workflows, operational procedures, ...
  - Application domain: concepts, objectives, tasks, constraints, regulations, ...
  - Strengths & weaknesses of the system-as-is
- Identifying the system **stakeholders**:
  - Groups or individuals affected by the system-to-be, who may influence its elaboration and its acceptance
  - Decision makers, managers, domain experts, users, clients, subcontractors, analysts, developers, ...

**Products**: Initial sections for preliminary draft proposal
Glossary of terms

## Requirements elicitation

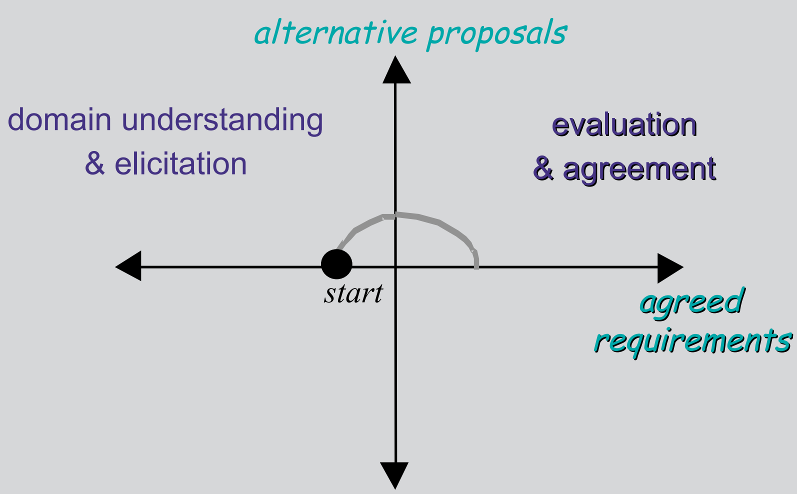
Exploring the problem world ...

- Further analysis of problems with system-as-is: symptoms, causes, consequences
- Analysis of technology opportunities, new market conditions
- Identification of ...
  - improvement objectives
  - organizational/technical constraints on system-to-be
  - *alternative* options for satisfying objectives, for assigning responsibilities
  - scenarios of hypothetical software-environment interaction
  - requirements on software, assumptions on environment

**Product**: Additional sections for preliminary draft proposal




## The RE process (2)



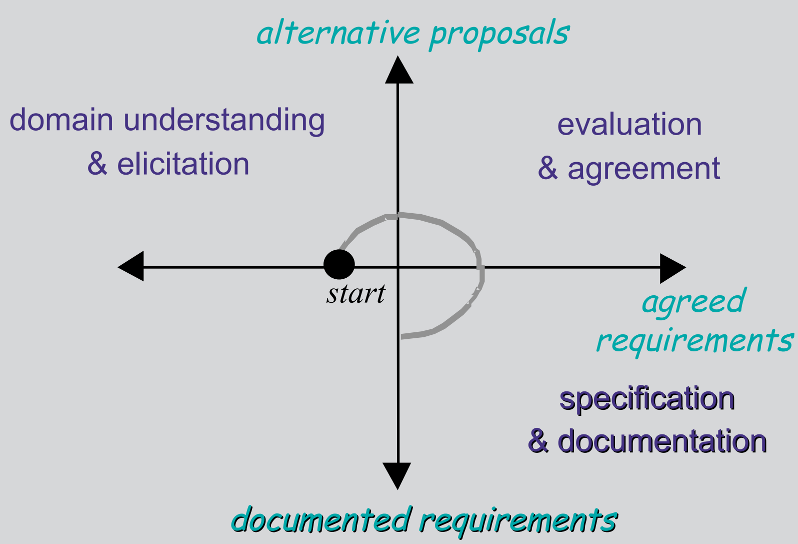
## Evaluation & agreement

- Negotiation-based decision making ...
  - Identification & resolution of **conflicting** concerns
  - Identification & resolution of **risks** with proposed system
  - Comparison of **alternative options** against objectives & risks, and selection of preferred ones
  - Requirements **prioritization**: to resolve conflicts, address cost/schedule constraints, support incremental development

**Product**: Final sections of draft proposal documenting the selected/agreed objectives, requirements, asssumptions (incl. rationale for selected options)




## The RE process (3)



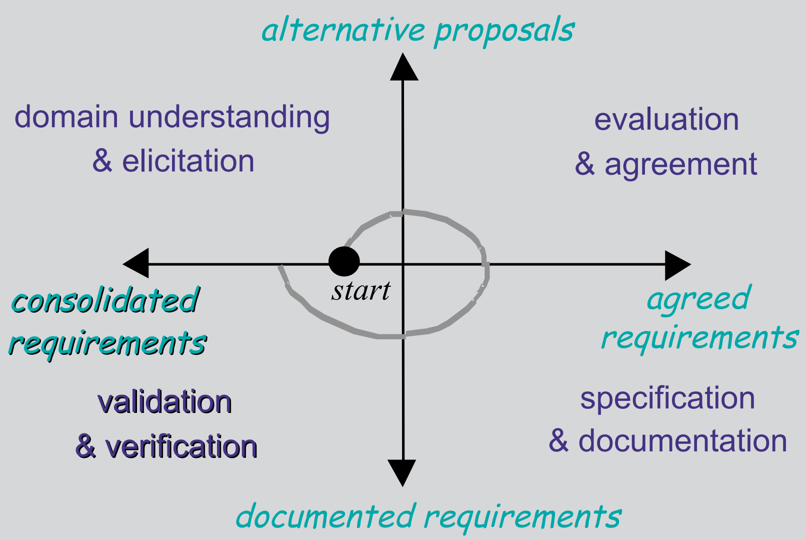
## Specification & documentation

- Precise definition of all features of the agreed system
  - Objectives, concepts, relevant domain properties, system/software requirements, assumptions, responsibilities
  - Satisfaction arguments, rationale for options taken
  - Likely system variants & evolutions
  - Estimated costs
- Organization of these in a coherent structure
- Documentation in a form understandable by all parties

Resulting product: **Requirements Document** (RD)




## The RE process (4)



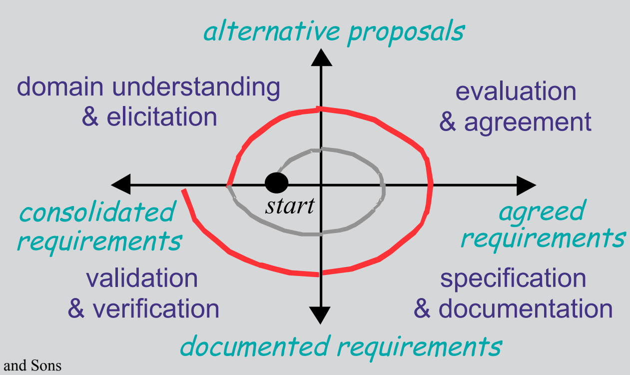
## Requirements consolidation

- Quality assurance activity on RD ...
  - Validation: adequacy of RD items wrt real needs ?
  - Verification: omissions, inconsistencies ?
  - Checks for other target qualities (discussed next)
  - Fixing of errors & flaws
- **Products**: Consolidated RD
  Acceptance test data, prototype
  Development plan
  Project contract




## RE: an iterative process

- RE phases are ordered by data dependencies
- No strict sequencing: intertwining, overlap, backtracking
- Iterated cycles due to error corrections & **evolving needs**
  - during RE, during software development, after deployment



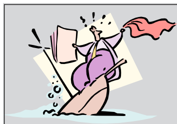
## Setting the scene: outline

- What is Requirements Engineering?
  - The problem world & the machine solution
  - The scope of RE: the WHY, WHAT and WHO dimensions
  - Types of statements involved: descriptive *vs.* prescriptive
  - Categories of requirements: functional *vs.* non-functional
  - The requirements lifecycle: actors, processes, products
  - **Target qualities and defects to avoid**
  - Types of software projects
  - Requirements in the software lifecycle
  - Relationship to other disciplines




## Target qualities for RE process

- **Completeness** of objectives, requirements, assumptions
- **Consistency** of RD items
- **Adequacy** of requirements, assumptions, domain props
- **Unambiguity** of RD items
- **Measurability** of requirements, assumptions
- **Pertinence** of requirements, assumptions
- **Feasibility** of requirements
- **Comprehensibility** of RD items
- **Good structuring** of the RD
- **Modifiability** of RD items
- **Traceability** of RD items



## Errors in a requirements document (RD)

- **Omission**: problem world feature not stated by any RD item
  e.g. no req about state of train doors in case of emergency stop
- **Contradiction**: RD items stating a problem world feature in an incompatible way
  "Doors must always be kept closed between platforms"
  *and* "Doors must be opened in case of emergency stop"
- **Inadequacy**: RD item not adequately stating a problem world feature
  "Panels inside trains shall display all flights served at next stop"
- **Ambiguity**: RD item allowing a problem world feature to be interpreted in different ways
  "Doors shall be open as soon as the train is stopped at platform"
- **Unmeasurability**: RD item stating a problem world feature in a way precluding option comparison or solution testing
  "Panels inside trains shall be user-friendly"

 37

## Flaws in a requirements document (RD)

- **Noise**: RD item yielding no information on any problem world feature (Variant: uncontrolled redundancy)
  "Non-smoking signs shall be posted on train windows"
- **Overspecification**: RD item stating a feature not in the problem world, but in the machine solution
  "The *setAlarm* method shall be invoked on receipt of an *Alarm* message"
- **Unfeasibility**: RD item not implementable within budget/schedule
  "In-train panels shall display all delayed flights at next stop"
- **Unintelligibility**: RD item incomprehensible to those needing to use it
  A requirement statement containing 5 acronyms
- **Poor structuring**: RD item not organized according to any sensible & visible structuring rule
  Intertwining of acceleration control and train tracking issues

 38

## Flaws in a requirements document (2)

- **Forward reference**: RD item making use of problem world features not defined yet

  Multiple uses of the concept of *worst-case stopping distance* before its definition appears several pages after in the RD
- **Remorse**: RD item stating a problem world feature lately or incidentally

  After multiple uses of the undefined concept of *worst-case stopping distance*, the last one directly followed by an incidental definition between parentheses
- **Poor modifiability**: RD items whose changes must be propagated throughout the RD

  Use of fixed numerical values for quantities subject to change
- **Opacity**: RD item whose rationale, authoring or dependencies are invisible

  "The commanded train speed must always be at least 7 mph above physical speed" *without* any explanation of rationale for this




## The RE process may vary according to project type

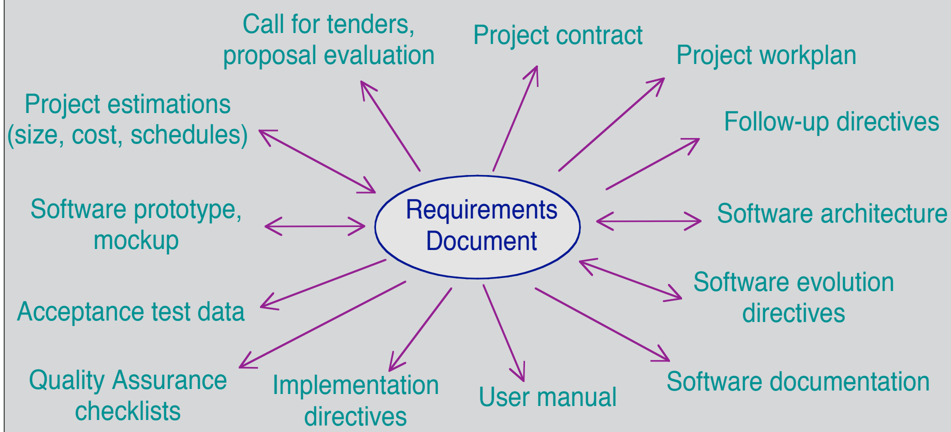
- **Greenfield** *vs.* **brownfield** projects
- **Customer-driven** *vs.* **market-driven** projects
- **In-house** *vs.* **outsourced** projects
- **Single-product** *vs.* **product-line** projects

Variation factors ...

- Respective weights of elicitation, evaluation, documentation, consolidation, evolution
- Intertwining RE/design
- Respective weights of functional vs. non-functional reqs
- Types of stakeholder & developer involved
- Specific uses of the RD
- Use of specific techniques

## Requirements in the software lifecycle

- The RD impacts on many software artifacts




- RE, system design & software architecture design are inevitably intertwined



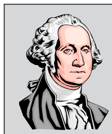

## RE has multiple connections with other disciplines

- Primarily with Software Engineering (SE)
- Other connections:
  - **Domain understanding & requirements elicitation**: system engineering, control theory, management science, organization theory, behavioral psychology, anthropology, AI knowledge acquisition
  - **Requirements evaluation & agreement**: multicriteria analysis, risk management, conflict management, negotiation theory
  - **Requirements specification, documentation & consolidation**: software specification, formal methods in SE
  - **Requirements evolution**: change management, configuration management in SE
  - **System modeling**: conceptual models in DB & MIS; task models in HCI; knowledge representation in AI

## Setting the scene: outline (2)

- **Why engineer requirements?**
  - **The requirements problem: facts, data, citations**
  - **Role and stakes of Requirements Engineering**
- Obstacles to good RE practice
- Agile development and RE



## The requirements problem: facts, data, citations

- Poor requirements are ubiquitous ...

  "Requirements need to be engineered and have continuing review and revision"

  Bell&Thayer '76

- Prohibitive cost of late correction ...

  "Up to 200 x cost of early correction"

  Boehm '81

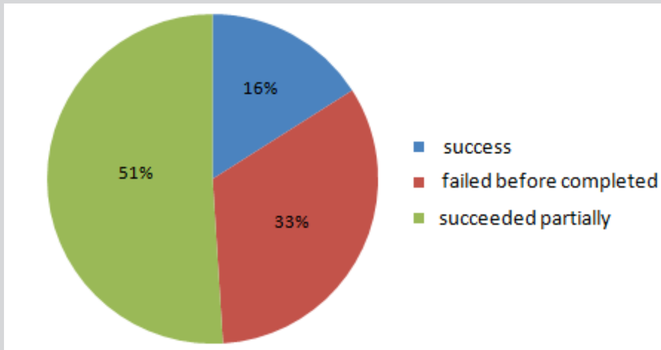
- RE is hard & critical ...

  "Hardest, most important function of SE is the iterative *extraction* & *refinement* of requirements"

  Brooks '87

## The requirements problem: Standish report, 1995

Survey of 350 US companies, **8000 projects**



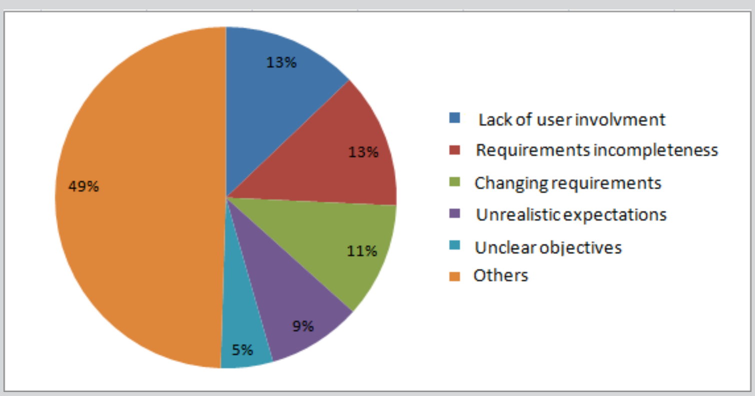

(partial success = partial functionalities, excessive costs, big delays)

Major source of failure:
poor requirements engineering ≅ 50% responses




## The requirements problem: Standish report, 1995 (2)

Major source of failure:
poor requirements engineering ≅ 50% responses:




www.standishgroup.com/chaos.html

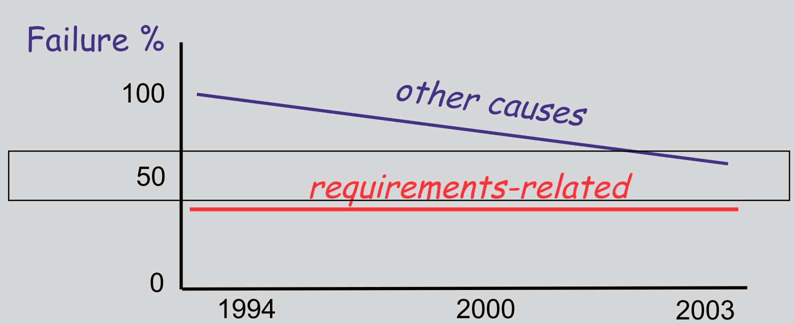
## The requirements problem: European survey, 1996

- Coverage: 3800 EUR organizations, 17 countries
- Main software problems perceived to be in...
  - requirements specification
    > 50% responses
  - requirements evolution management
    50% responses

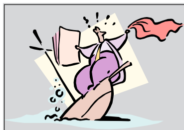
[European Software Institute, 1996]




## The requirements problem is perceived to persist in spite of progress in software technology




[J. Maresco, IBM developersWork, 2007]



## Requirements-related errors are ...

- the most **numerous**
  - ± 40% of software errors
- the most **persistent**
  - found very late, often after software delivery
- the most **expensive**
  - cost ... 5x more if fixed during design
    10x more if fixed during implementation
    20x more if fixed during integration testing
    200x more if fixed after delivery
  - account for 66% of software error costs

[Boehm, Jones, Lutz, Hooks & Farry, ...]




## Requirements-related errors can be dangerous

- US Aegis/Vincennes (1988): shooting of IranAir airbus
  - Missing timing between 2 threat events in requirements on alarm software
- Patriot anti-missile system (1st Gulf war)
  - Hidden assumption on maximum usage time
- London Ambulance System (1993): fatal delays
  - Wrong assumptions on crew behavior, ambulance localization system, radio communication, ...
- Boeing 757 crash, Cali (1995)
  - Autopilot 's wrong timing/localization assumption on flap extension point
- Cf. ACM RISKS Digest Forum website

## Example: inadequate domain property in A320 braking logic

SofReq: reverse = ‘on’ **iff** WheelPulses = ‘on’

ASM: reverse = ‘on’ **iff** ReverseThrustEnabled
WheelPulses = ‘on’ **iff** WheelsTurning

Dom: ~~MovingOnRunway **iff** WheelsTurning~~

--------------------------------------------------------------------------------

*SysReq:* ReverseThrustEnabled **iff** MovingOnRunway

*Warsaw crash: plane moving on waterlogged runway with no wheels turning (aquaplaning)*



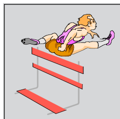

## Role and stakes of RE

- **Technical** impact
  - on many software-related artifacts (as seen before)
- **Managerial** impact
  - basis for communication among parties and for project management
- **Legal** impact
  - contractual commitment client-provider-subcontractors
- Impact on **certification**
  - Mastered RE process required by many quality standards & certification authorities

## Role and stakes of RE (2)

- Impact on **economy**, security, and safety
  - **Cost** and **consequences** of errors in requirements on the software-to-be, assumptions about its environment
- **Social** impact
  - *from* user satisfaction
    *to* degradation of working conditions
    *to* system rejection



## Obstacles to good RE practice

- RE efforts often spent without guarantee of project contract being concluded
- Pressure on tight schedules, short-term costs, catching up on technology
- Too little work available on RE economics
  - Lack of quantitative data on RE benefits & cost savings
  - Progress in RE process is harder to measure than in design, implementation
- RDs are sometimes felt ...
  - big, complex, to be quickly outdated
  - too far away from the executable product customers are paying for

## Agile development and RE

- More agile development may overcome some obstacles
  - early & continuous provision of functionality of value to customer
  - by reducing the req-to-code distance
- Short RE cycles in spiral RE process, each directly followed by short implementation cycle
  - Useful functional increment is elicited directly from the user
  - Evaluation/spec/consolidation phases often shortcut (e.g. spec = test case on the implementation)
  - Increment is implemented/tested by small team at same location, close to the user for instant feedback, using strict rules




## Strong assumptions for agility to be successful

- All stakeholder roles are reducible to one single role
- Project sufficiently small to be assignable to single, small, single-location team (programmers/testers/maintainers)
- "User" can interact promptly & effectively
- Functionality can be provided quickly, consistently, incrementally from essential to less important (no prioritization required)
- Non-functional aspects, environment assumptions, objectives, alternative options, risks may receive little attention
- Little documentation required for work coordination & product maintenance; requirements precision not required; verification before coding is less important than early release
- Requirements changes are not likely to require major code refactoring

*__More/less__ agility is achievable by __less/more__ weight in elicitation, evaluation, documentation, consolidation phases of RE cycles*

## Setting the scene: summary

- What is Requirements Engineering?
  - RE is concerned with the problem world only
  - Scope: WHY, WHAT, WHO issues
  - Statement types: *descriptive* vs. *prescriptive*; requirements, assumptions, domain properties, defs; satisfaction arguments
  - Categories of requirements: functional, non-functional
  - RE is a spiral process; elicit-evaluate-specify-consolidate cycles driven by corrections & evolving needs
  - Multiple target qualities, defects to avoid --some are critical !
  - Weight on each RE phase may depend on project type
  - Requirements impact on many software artefacts
- Why engineer requirements?
  - Requirements-related errors are the most numerous, persistent, expensive, dangerous
  - Technical, managerial, legal, economical, social impact of RE
- Obstacles to good RE practice; agility in spiral RE