Design of Reactive Systems
Summer 2001

Prof. Dr. Stefan Leue
Institute for Computer Science
Albert-Ludwigs-Universität Freiburg

leue@uni-freiburg.de

Copyright © Stefan Leue 2001
Problems of the Software Design Process

Part 1
What is Software?

♦ Software is engineered, not manufactured
  ‣ mostly custom built
  ‣ little component assembly
  ‣ human intense production process

♦ Software doesn’t wear out
  ‣ but it deteriorates due to
    ‐ change to accommodate changing requirements
    ‐ changes in the environment software is executing in (hardware, operating system, etc.)

♦ Software is complex

♦ Software is determining system factor
  ‣ in technical systems, determines up to 80% of development effort
Relevance of Software

♦ Examples: Telephony Switches

- Nortel Networks, DMS-100
  - appr. 25-30 mio lines of code
  - life-cycle: development started in 1981
  - appr. 2000-3000 sw developers concurrently involved

- Lucent Technologies 5ESS
  - appr. 10 mio. lines of code
  - 30% of code is running in background to recover from sw failures
  - Mean Time Between Failure: appr. 7 years (would be in the order of hours without recovery code)

- Lucent PathStar (telephony over IP)
  - a few 100k lines of code
  - call processing and intelligent feature processing: some 10k lines of code

- Ericsson AX 301 (telephony over ATM)
  - 1 mio loc (Erlang) (4 times less code than usage of other language)
  - 0.5 mio loc C++
  - 30 k loc Java
Product Requirements

♦ Correctness and Reliability
  ‣ embedded technical systems, software cast in silicone
    – assume: not safety-critical
    – high degree of replication
    – hw and sw intertwined
      • faulty sw control entails very high recall and repair costs
  ‣ telecommunications systems
    – safety sensitive
      • failure causes immense cost to operator
        * example: AT&T switch failure in NYC
      • unavailability causes customer dissatisfaction
  ‣ safety critical systems
    – failure may entail loss of life
    – example: avionics
    – very low failure probabilities desirable

♦ Common
  ‣ correctness of software
Software Engineering

♦ Engineering
  ‣ ".. profession devoted to designing, constructing, and operating the structures, machines, and other devices of industry and everyday life."

♦ Characteristics of an Engineering Discipline
  ‣ Well-understood technologies
  ‣ Well-defined processes
  ‣ Predicatability of process stage results
  ‣ Repeatability of process steps

♦ Software Engineering
  ‣ Term first coined by a NATO study group in 1967.
  ‣ Observation that there is a software crisis.
  ‣ Conclusion, that an engineering-like approach towards the development of software was the right way to tackle this problem.
  ‣ First NATO Software Engineering Conference, held in Garmisch, Germany, in 1968
Software Engineering

Quotes from


- "... an awareness of the rapidly increasing importance of computer software systems in many activities of society."
- "... For example, the rate of increase of air traffic in Europe is such that there is a pressing need for an automated system of control."
- "There was considerable amount of debate on what some members chose to call the `software crisis' or the `software gap'."
- "... statement of concern were made ... about the tendency for there to be a gap ... between what was hoped for from a complex software system, and what was typically achieved."
- "The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities... There are many areas where there is no such thing as a crisis - sort routines, payroll applications, for example. It is large systems that are encountering great difficulties."
Quotes from


``Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death."

``To-day we tend to go on for years, with tremendous investments, to find that the system, which was not well understood to start with, does not work as anticipated."

``We build [software] systems like the Wright brothers built airplanes - build the whole thing, push it off the cliff, let it crash, and start over again."

"… implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."
Software Engineering

Quotes from


- "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

- "There was general agreement that 'software engineering' is in a very rudimentary stage of development as compared with the established branches of engineering."
Software Crisis?

♦ Survey of the US Government Accounting Office - 1979

9 contracts, total $7 mio

- never delivered: 29%
- reworked/abandoned: 19%
- used as delivered: 2%
- used after changes: 3%
- never used: 47%
Software Crisis

♦ Survey by IEEE Software, 1995
  ‣ 30% of all software projects are cancelled
  ‣ 50% of all projects are 150% over budget
  ‣ only 60% of functionality is in the final product

♦ Survey by Standish Group ([http://www.standishgroup.com](http://www.standishgroup.com)), 1994
  ‣ Analysis of 350 companies with over 8000 software projects
    – 31 % of projects were aborted prior to completion
    – in large development companies, only 9% of all projects were completed within projected budget and time limits (16% in small companies)

♦ Survey by PC Week, 1995
  ‣ query of 365 information systems professionals on success of software development projects
    – 16% successful
    – 53% operational, but less than successful
    – 31% cancelled
Software Crisis

♦ Example: Hughes Aircraft of Canada

- Bid on a Transport Canada Project to unify and automate Canada's air traffic control system
- Originally: $377 million fixed price contract
- Increased to $420 Million fixed price contract
- Transport Canada made 5300 change request to requirements without regard to contract stipulations, cost, or schedule impact
- Transport Canada stopped payment because Hughes wasn't meeting milestones
- Hughes threatened to sue government unless payments were resumed and contract was renegotiated: additional $150 million for scaled back system
- Transport Canada threatened to scrap project
Software Crisis

♦ Conclusion: Symptoms of Software Crisis
  ‣ software products are being delivered late
  ‣ software projects exceed budget
  ‣ delivered software often doesn't really do what it is supposed to do
  ‣ software products are defective when delivered
  ‣ large projects get abandoned before delivery of a product

♦ Is it really a software crisis?
  ‣ many functioning software products are being built and delivered, more or less to the customers satisfaction
  ‣ however, one would like to see less of the above happening
  ‣ alternatively: **chronic software affliction**

♦ Consequences
  ‣ attempt to base the software production of an engineering approach, with well-defined inputs, well defined methods, and well-defined results
Why is there a Software Crisis?

♦ Or, why is Software inherently so difficult to produce

  1. No similar system ever built before.
     – Problem never solved before
     – Solution may be unknown
     – Assumptions about the system's environment may be guesses
     – Difficulties to estimate time and number of people needed to complete project
     – Example: baggage handling system at Denver Intl. Airport

  2. Requirements are not well understood

  3. Requirements change during the software life cycle
     – customers typically don't exactly know what capabilities they expect of a new system until they see an initial version or prototype
     • example: spreadsheet
Why is there a Software Crisis?

♦ Or, why is Software inherently so difficult to produce

  4. Complex interactions amongst services and components
     – call forwarding vs. call screening
     – reverse thrust deployment vs. in-flight avoidance
  5. Nature of systems
     – concurrent systems: deadlocks etc.
     – reactive systems: timing
     – embedded systems: hardware interactions
     – information systems: complexity, legacy
  6. Software is easily malleable
     – lends itself to "code-and-fix"
Why is there a Software Crisis?

♦ Or, why is Software inherently so difficult to produce

♦ 7. Software is a discrete artifact

“When software fails, it invariably fails catastrophically. This awful quality is a reflection of the lack of continuity between successive system states in executing software. If the preceeding state is correct, there is no inherent carry over of even partial correctness into the next succeeding state.” (W. Royce, quoted after [Rushby])

“The problem … is that computer hardware and software are discrete systems: their behavior is determined by a succession of discrete state changes. The succession of states need not produce behavior that varies smoothly or continuously, instead it can exhibit discontinuities and abrupt transitions. The ability to select among many alternative courses of action is a source of the power … provided by computer systems, but it is also the reason why their behavior is hard to predict and to validate.” [Rushby]
Why is there a Software Crisis?

♦ Or, why is Software inherently so difficult to produce

7. Software is a discrete artifact
   – consequences
     • correctness cannot be approximated
     • partial tests can only produce correctness claims with limited validity
   – comparison to continuous artifacts
     • suspension bridge cable in which 1% of wires are faulty
     • program in which 1% of statements are faulty
Why is there a Software Crisis?

♦ Myths related to software development
  ‣ Management myths (when projects go awry)
    – We already have books and standards, that give the programmers all they need to know
    – My people to have state-of-the-art development tools, after all, we buy the always the latest computer hardware
    – If we get behind schedule, we just add staff and will be able to catch up easily
      • but how much effort will you spend in getting your new staff up to speed?
  ‣ Customer myths
    – A general statement of objectives is sufficient to start programming
    – Project requirements continually change, but change can easily be accommodated since software is so flexible and easy to change
  ‣ Practitioner's myths
    – Once we write the program and get it to work, our job is done
    – Until I get the program running, I have no way of assessing and assuring its quality
    – The only deliverable for a successful project is the working program
Software Engineering

♦ Consequence
  ▸ attempt to base software development on a rigorous, engineering approach
  ▸ introduction of life-cycle and process models
    – goal: well-defined and reproducible transformation on every stage of the development process

♦ Advantages
  ▸ clear definition of separate tasks (“separation of concerns”)
    – enables team work
    – enhances reproducibility
    – enables dividing complex task into manageable sub-tasks
      • e.g., specification, design, implementation, testing
  ▸ specification and documentation
  ▸ possibility to introduce verification, validation, prototyping and evolutionary development models into production process
Build-and-Fix

- **Build-and-Fix Model** is the *simplest* process model. The product is constructed *without any design or requirement specifications*.

- Main Points:
  - Specifications and Design are *not considered* as initial steps in the life-cycle.
  - The developers simply *build* a product which is *reworked* as many times as necessary until it satisfies the client.
  - It works for *small* programming exercises (100-200 lines long) but fails for products of any reasonable size.
  - Does not produce any specification or design documents → the overall *maintenance* cost is considerably higher.
  - Alternative: Instead of the build-and-fix approach, the initial time is better to be spent to chose an overall process model that will specify the requirements, specification, planning, design, implementation, integration, and maintenance phases.
Life-cycle Model: Waterfall
Life-cycle Model: Waterfall

System Design
- overall design
- HW/SW split
- informal, with customer
Life-cycle Model: Waterfall

**SW Requirements**
- “what” the system has to do
- functional/non-functional
- observable behaviour
- SRS document

**System design**

**Requirements**

**SRS**
Life-cycle Model: Waterfall

- System design
- Requirements
- SRS
- Design

Design
- “how” the system is doing it
- architectural design
- detailed design
- design document
Life-cycle Model: Waterfall

Module Implementation and Test
- implement module structure
- test modules in isolation
Life-cycle Model: Waterfall

- System design
- Requirements
  - SRS
  - Design
  - Implementation
- Integration
  - integrate modules
  - integration testing
  - customer acceptance tests
Life-cycle Model: Waterfall

System design → Requirements → Design → Implementation → Integration → Maintenance

Maintenance
- deploy product
- corrective, adaptive, perfective maintenance
Analysis of Waterfall Model

- Advantages:
  1. Enforces a **disciplined** approach (i.e. take away the fun of programming)
  2. **Testing** and **verification** is enforced at every phase of the life-cycle
  3. **Documentation** produced can be used to **reduce maintenance costs**

- Disadvantages:
  1. The first time the client sees a working artifact is only after the entire product has been coded. This is the syndrome of “*I know this is what I asked for, but it is not really what I wanted*” client response
  2. The model depends heavily on **written specifications**, requirements, design documents etc, the process therefore tends to be **bureaucratic**. Only documentation can not describe what the product will look like how it will really performs, and it does meet the client’s real needs
**Analysis of Waterfall Model**

**Problems** with waterfall model:

1. Difficult to estimate resources at early stages.
2. Difficult to assess whether final product meets expectations.
3. Users don’t know requirements beforehand, at early stages.
4. No emphasis on anticipation for change.
5. Process is “document-driven” → bureaucratic.
6. Not adequate to represent evolving software production process.
7. Nothing is functional and delivered to the user until the end of the development process.
Modifiziertes Wasserfallmodell
The Spiral Model

Figure 7.4 The spiral model. (From Boehm 1988, ©1988 IEEE, by permission of IEEE.)
Rational Unified Process
Effort per Life-Cycle Stage

TRW - 63 projects
Relative Duration per Stage for large projects

Integration & System Testing: 26%
Requirements: 12%
Preliminary Design: 17%
Det. Design, coding, unit testing: 45%
TRW - 63 projects
Relative Cost per Stage for large projects

Integration & System Testing
23%

Requirements
6%

Preliminary Design
15%

Det. Design, coding, unit testing
56%
Effort per Life-Cycle Stage, Including Maintenance

- Maintenance: 60%
- Testing: 16%
- Requirements & Design: 16%
- Implementation: 8%
Relative Importance of Life-Cycle Stages

♦ Survey by Standish Group, 1995
  ‣ Reasons for failures of software projects
    – incomplete requirements: 13.1%
    – lack of user involvement: 12.4%
    – lack of resources: 10.6%
    – unrealistic expectations: 9.9%
    – lack of executive support: 9.3%
    – changing requirements and specifications: 8.7%
    – lack of planning: 8.1%
    – system no longer needed: 7.5%
  ‣ Conclusion
    – elicitation, understanding, documentation and specification of requirements are central contributing factors to software project failures
Relative Importance of Life-Cycle Stages

- **Boehm and Papaccio, 1988**
  - “there are usually three design faults for every two coding faults”

- **Jones and Thayer, 70ies**
  - for projects with 30-35000 delivered source code instructions, 35% of failures were due to design errors
  - for projects with 40-80000 delivered source code instructions, 10% of failures are due to requirements errors, and 55% are due to design errors

- **Basili and Perricone, 1984**
  - 48% of faults observed in a medium-scale software project were attributed to incorrect or misinterpreted functional specifications or requirements

- **Beizer, 1990**
  - “Requirements … are a major source of expensive bugs. The range is from a few percent to more than 50% … What hurts most … is that they’re the earliest to invade the system and the last to leave. It’s not unusual for a faulty requirement … only to be caught after hundreds of sites have been installed.”
Relative Importance of Life-Cycle Stages

- Davis’ Hypotheses regarding the Importance of Requirements Specifications ([Davis])
  - **Hypothesis 1**: The later in the life-cycle that a fault will be discovered, the more expensive it will be to fix it

![Graph showing the unit cost across different stages of software development lifecycle](image-url)
Relative Importance of Life-Cycle Stages

- Possible explanation: errors are discovered only much later in the development process after they were made.
- Additional cost due to late discovery: all implied errors need to be corrected as well.
- Mizuno's observation:

  - Specification:
    - Correct specification
    - Erroneous specification

  - Design:
    - Correct design
    - Erroneous design
    - Design based on erroneous specifications

  - Implementation:
    - Correct programme
    - Programming error
    - Programme based on erroneous design
    - Programme based on erroneous specification

  - Testing:
    - Correct functionality
    - Correctable errors
    - Non correctable errors
    - Hidden errors
Relative Importance of Life-Cycle Stages

- **Hypothesis 2**: Many errors remain latent and are not detected until well after the stage at which they are made.
  - Boehm:
    - 54% of all errors ever detected at TRW were detected after the coding and unit testing stage.
      - 45% attributable to requirements and design stages
      - 9% attributable to coding stage

- **Hypothese 3**: Many requirements errors are being made
  - DeMarco: 56% of all software faults detected can be traced to requirements errors
  - Examples showing that automatic analysis of previously hand-reviewed specifications discover substantial numbers of additional faults
  - earlier numbers
Relative Importance of Life-Cycle Stages

- **Hypothese 4**: Errors in requirements specifications are typically incorrect facts, omissions, inconsistencies, and ambiguities
  - Analysis of the Navy A-7E specification, 77% of all detected errors are not of clerical nature
    - 49% incorrect facts
    - 31% omissions
    - 13% inconsistencies
    - 5% ambiguities

- **Hypothese 5**: Requirements errors can be detected
  - Effectivity of inspections (manual or machine-supported)
  - Numerous case studies using automatic analysis tools (REVS, SPIN, …)
Relative Importance of Life-Cycle Stages

♦ **Consequences**
  ‣ H3 and H4: Many requirements errors are being made
  ‣ H2: Many of these errors are not being detected early
  ‣ H5: Many of these errors can be detected early
  ‣ H1: Not detecting these errors early contributes to skyrocketing software costs

♦ **Impact of requirements errors**
  ‣ The resulting software may not satisfy user's real needs
  ‣ Multiple interpretations of requirements may cause disagreements between customers and developers, wasting time and dollars and perhaps resulting in lawsuits
  ‣ It may be impossible to thoroughly test that the software meets its intended requirements
  ‣ Both time and money may be wasted building the wrong system
Relative Importance of Life-Cycle Stages

Delegate

Consequence

- apply software engineering methods as *early* on in the life cycle as possible
- in particular
  - careful requirements elicitation and specification
  - careful design
Activities During the Requirements Stage

♦ Starting Point
  ‣ system specification document (hw and sw)
  ‣ customer requirements (abstract)

♦ Activities (after [Kotonya and Sommerville])
  ‣ requirements elicitation
    – interviews, scenarios, market observation, etc.
  ‣ requirements elicitation and negotiation
    – determination, which of the possibly contradictory requirements are important
  ‣ requirements documentation and specification
    – generally comprehensible requirements document
    – non-formal or formal specification
  ‣ requirements validation
    – consistency
    – completeness
    – correspondence of documented requirements and abstract customer or user requirements
Requirements Stage Process Model

customer or user requirements (abstract)

requirements elicitation

requirements analysis and negotiation

requirements documentation and specification

negotiated and validated requirements

validation
Requirements Specification

  ‣ Requirement: “… (2) A condition or capability that must be met or professed by a system component to satisfy a contract, standard, specification or other formally imposed document.”
  ‣ Requirement Specification: “A specification that sets forth the requirements for a system of system component; … typically included are functional requirements, performance requirements, interface requirements, design requirements and development standards.”

♦ According to [Davis]
  ‣ “A software requirements specification is a document containing a complete specification of what the system will do without saying how it will do that.”
Requirements Specification

♦ Complete Description of the Externally Visible Behaviour:
The What/How Dilemma

- Code
- Algorithms
- Module specifications
- Architecture/data flow
- Actual product's behavior
- Product space (set of all legal behaviors)

Copyright © Prentice-Hall Inc., 1993 1990
Requirements Specification Document

♦ Possible structure (c.f. [Sommerville])
  ‣ Introduction
    – purpose, functionality and context of system
  ‣ Glossary
    – definition of technical terms used in document
  ‣ System Models
    – models showing relationships between system components, and relationships between system and environment
  ‣ Functional requirements
    – services provided by the system
  ‣ Non-functional requirements
    – constraints on system and implementation that are quantitative in nature, e.g., performance, reliability, etc.
  ‣ System evolutions
    – anticipated changes
  ‣ Requirements specifications
    – functional requirements in more detail
    – if possible formally specified

♦ Standards for the structure of a requirements document exist, e.g.
  ‣ IEEE 830-1998
  ‣ DIN 66270 ("Lastenheft/Pflichtenheft" in German)
Requirements Specification Languages

Natural Language vs. Formal Notations

- natural language
  
  "when the receiver is lifted, then a dial-tone will eventually be heard"
  
  + very expressive
  
  + will be understood by all persons involved in the development process
  
  - ambiguities

- formal notations and languages (possessing a mathematical semantics)
  
  (∀t₁, t₂ | t₁ ≤ t₂) (ringtone(t₁) ⇒ dialtone(t₂))
  
  + unambiguous
  
  + machine-analyzeable
  
  - limited expressiveness
    
    • in particular when machine analyzeability is required
  
  - will not be understood by all persons involved in the development process
    
    • software designers and programmers?
    
    • customers?
Requirements Specification Languages

- Criteria for the Evaluation of Specification Languages according to Ardis (c.f. [Pfleeger], 4.11)
  - applicability
  - implementability (code generation, executability)
  - testability
  - validatability (amongst others, readability for application experts)
  - maintainability
  - modularity
  - abstractness and expressiveness
  - logic completeness (e.g., does the language have a formal semantics so that inconsistencies can be discovered)
  - verifiability
  - runtime-safety
Requirements Specification Languages

♦ Criteria for the Evaluation of Specification Languages according to Ardis (c.f. [Pfleeger], 4.11)
  ▸ maturity of tools
  ▸ looseness (are incompletenesses and nondeterminisms permitted)
  ▸ learning curve
  ▸ technical maturity (certification or standardization)
  ▸ data modeling (representation of data and data relationships)
  ▸ discipline (does it force users to write well-structured, easily understandable and well-behaving specifications)

♦ Conclusion
  ▸ specification languages are highly domain specific
Requirement Specification Properties

♦ correctness
  ▪ all facts stated in the requirements specification represent required properties of the system to be designed
  ▪ counterexample
    – specification requires that callee receives a dial tone when the caller hangs up, while the system is supposed to provide a busy tone in this situation
Requirement Specification Properties

♦ unambiguity

♦ all facts stated in the specification have a single interpretation
♦ natural language descriptions are frequently ambiguous
  – for up to 12 aircraft on the control screen the small display format is to be used, otherwise the large format is required
  • to satisfy the abstract system requirement of avoidance of clutter it is irrelevant whether this is interpreted as < 12 or ≤ 12
  • this may, however, lead to problems if both display formats are designed by different design teams, one none of the two teams feels responsible for the case =12 (empty display in that case?)
Requirement Specification Properties

♦ completeness
  ‣ definition 1: every property required of the system is expressed in the specification
    – does this imply that the specification needs to include the description of behaviour that is not permitted?
      • may be difficult to achieve due to large number of possible system behaviours
      • formal specifications may help in this situation, e.g.,
        \[
        (\forall A, B) (Call(A, B) \land Idle(B) \Rightarrow \\
        (Ring(B) \land (\forall X \neq B)(\neg Ring(X)))
        \]
        however, there may be good reasons for allowing other phones to ring…
  ‣ definition 2: the responses of the software system on all types of possible input values are specified.
  ‣ for further definitions of completeness, c.f. [Davis]
Requirement Specification Properties

♦ verifiability

♦ there exists an effective, either manual or automated procedure for checking whether a software product satisfies the required properties

♦ examples:
  – formal verification: use of mathematical proof
    • formalization
      * implementation ⇒ specification
  – formal validation: experimentation with a model to check validity of a property
    • testing
    • model checking
    • simulation

♦ many requirements are not verifiable
  – after every command the operating system is supposed to return control to the user
  – the software is never supposed to enter an infinite loop
  – the user interface is required to be easy to operate
Requirement Specification Properties

♦ consistency
  ‣ no two requirements are logically a contradiction
  ‣ possible inconsistencies
    – contradictory behaviour
      • when the receiver is being picked up, a dial tone will be heard
      • when the receiver is being picked up, a rint to will be heard
    – contradictory expressions
    – contradictory properties
    – temporal inconsistency
      • entering $a$ leads to an output $b$ at the same time
      • $b$ may never be observed less than 15 seconds after observing $a$
 Requirement Specification Properties

♦ traced
  ‣ requirements specification is traced if origin of every requirement is clear
    – e.g., requires documentation of rationale behind the choice of certain values (for instance, timing bounds)

♦ traceable
  ‣ the requirements specification is edited in such a way that it is easy to reference every single requirement
    – often, achieved through numbering scheme
    – important when relating design or code to requirements
      • essential in testing

♦ design independence
  ‣ the requirement specification does not imply a specific software architecture and does not require specific algorithmic implementations
  ‣ contrary: overspecification
Classification of Software Systems

♦ **transformational systems**
  - transformation of (possibly empty) set of input data into output data
  - describes a function from an initial state $S_i$ into a final state $S_k$
  - examples
    - scientific computations
    - compilers
    - database batch processing (e.g., payroll)
  - correctness criteria
    - termination
    - correctness of the function $S_i \rightarrow S_k$
    - correctness of input - output transformation
Classification of Software Systems

♦ reactive systems
  ‣ maintain an ongoing interaction with their environment
  ‣ driven by environment events (stimuli) on which they react with responses to the environment
  ‣ examples
    – operating systems
    – communications protocols
    – control systems
  ‣ correctness criteria
    – non-termination under normal execution conditions
    – correctness of stimuli-response pairs
  ‣ formal description
    – models that describe continuous, infinite stimulus/response sequences
    – functions on stimulus are insufficient
      • necessary to include history (state)
Classification of Software Systems

♦ **embedded systems**
  - usually reactive systems
    - tightly intertwined with the hardware that they control
  - often executed on dedicated target system hardware
  - examples
    - flight management systems in aircraft
    - automobile control processors
    - timing control in microwave oven
    - control software in telephony switch or cellular phone
  - correctness criteria and formal description methods as for reactive systems
Classification of Software Systems

♦ real-time systems

- so far: correctness does not depend on relative or absolute processing speed of system components
  - e.g., pressing the request button will *eventually* lead to the elevator arriving at the requested floor
    - not particularly helpful if the time span between request and arrival exceeds human life expectancy...
- therefore, requirements specification often contain real-time bounds
  - pressing the request button will lead to the elevator arriving *within* 30 seconds at the requested floor
- distinction
  - soft real-time systems
  - hard real-time systems
- c.f. chapter 1 of [Heitmeyer und Mandrioli]
Classification of Software Systems

♦ soft real-time systems

- real-time bounds will be specified, but violation of these bounds will not lead to an invalidation of the system
- usually, probabilities attached to the satisfaction or violation of a real-time bound, e.g.,
  - with a probability of 0.99999 a dial-tone will be obtained within 500 ms of lifting the receiver
- also called quality-of-service requirements, in particular in the context of high-speed and multimedia computer networks
- specification method: stochastic extensions of timed system models
  - timed stochastic petri nets
  - timed markov chains
- correctness criteria: satisfaction of both the stochastic and the timed model
- note: "real-time system" sometimes used synonymously with
  - "reactive system" or
  - "embedded system" or
  - "soft real-time system"
Classification of Software Systems

♦ hard real-time systems
  ‣ correctness of system depends on the system satisfying the real-time constraints
    – *the system must have closed the compartment door within 20 seconds after retracting the undercarriage*
    – *the reactor shutdown mechanism must add additional coolant to the kernel within 30 seconds after the water pressure falls below threshold X*
  ‣ typical formats of real-time requirements
    – maximal and minimal value between
      • stimulus / response
      • response / response
      • stimulus / stimulus
      • response / stimulus
  ‣ requirements specification
    – real-time extended models
      • timed Petri nets and automata
      • real-time temporal logic
Classification of Software Systems

♦ hybrid systems
  ‣ dynamic systems, in which state is characterized by discrete as well as continuous variables
  ‣ example:
    – thermostat or chemical, controlled process
    – train crossing control
      • discrete variables
        * state of gate
        * train in crossing
      • continuous
        * speed of train
        * acceleration of train
        * timing of opening-closing of gate
  • requirement example:
    there may never be a train in crossing while gate is not closed
  ‣ specification using hybrid extensions of discrete models
Classification of Software Systems

♦ **Focus for this course**
  - reactive systems
  - discrete systems
  - no real-time constraints
  - concurrent systems
    - message exchange
    - common storage (shared variables)
Bibliographische Referenzen