CS445 / SE463 / ECE 451 / CS645

Software requirements specification & analysis

A reference model for requirements engineering

Fall 2015

Mike Godfrey & Daniel Berry & Richard Trefler

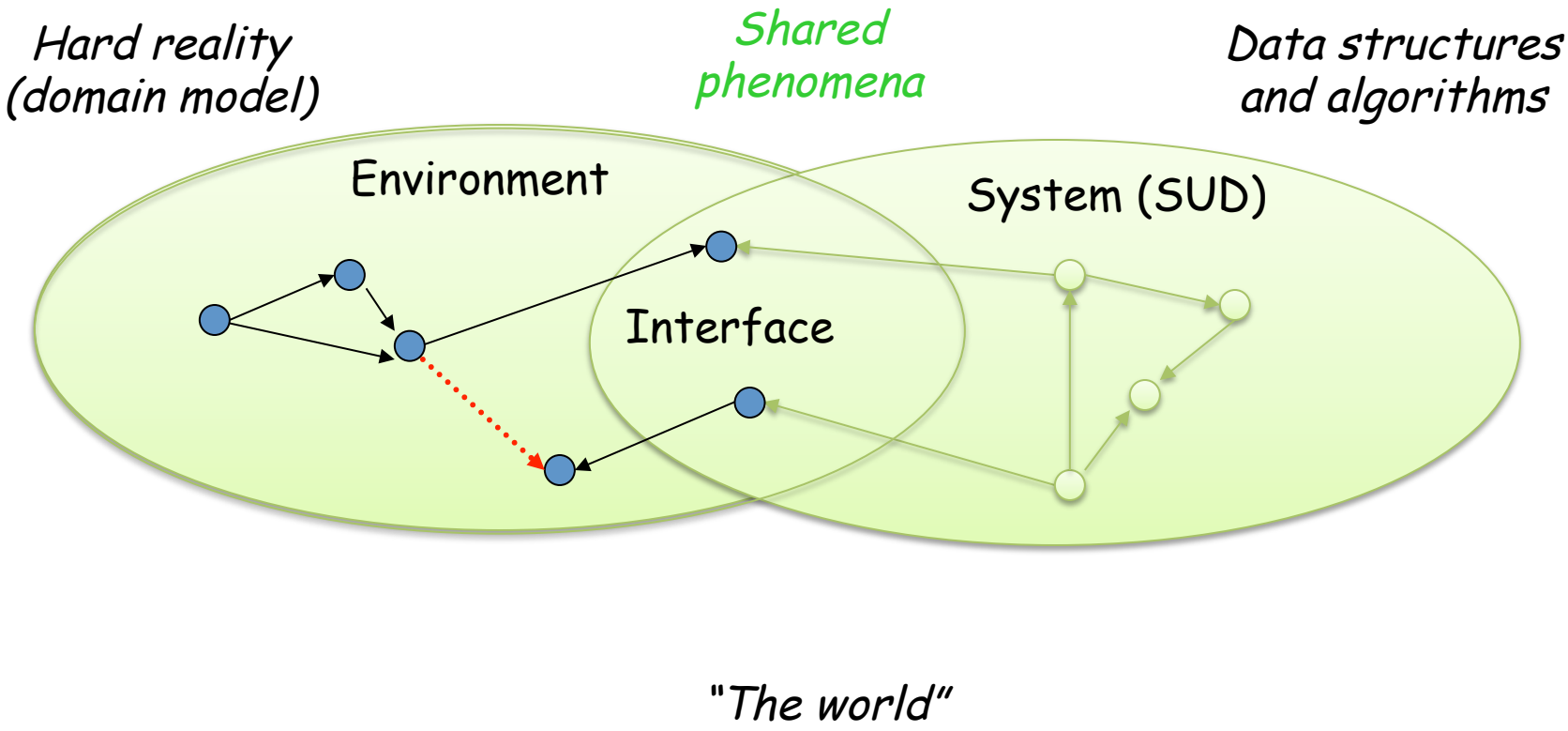
Overview

Goal: A clear understanding of how requirements relate to both their environment and the SUD (System Under Development)

- Topics:
  - Reference model for requirements engineering
  - System, environment, interface
  - Context diagrams
  - Deriving specifications from requirements
  - Domain knowledge

[Much of this is based on the work of P. Zave and M. Jackson with C. and E. Gunter "A Reference Model for Requirements and Specifications" *IEEE Software* 17:3, 37-43, 2000]

Reqs, specs, and programs



Reqs, specs, and programs

- We view the hardware / software as building a *System*
  - … that operates within a specified *Environment*
  - … interacting with it through a set of *Shared Phenomena:*
    - *Sensors* sense phenomena in the environment (e.g., keyboard clicks)
    - *Actuators* cause change in the environment (e.g., screen display)

- Both the System and the Environment sit in the *World*.
  - Systems are built to "improve" the Environment.

The system

- A *System* can be any socio-technical artifact that is to be constructed
  - It can be composed of some mix of software and hardware (incl. humans) and processes

- However, in general only the software is easily modifiable, and that is what we will concentrate on.
  - Hence, we will talk about

    Software Engineering and *Software Requirements*,

    but we really mean

    Systems Engineering and *System Requirements*.

The environment

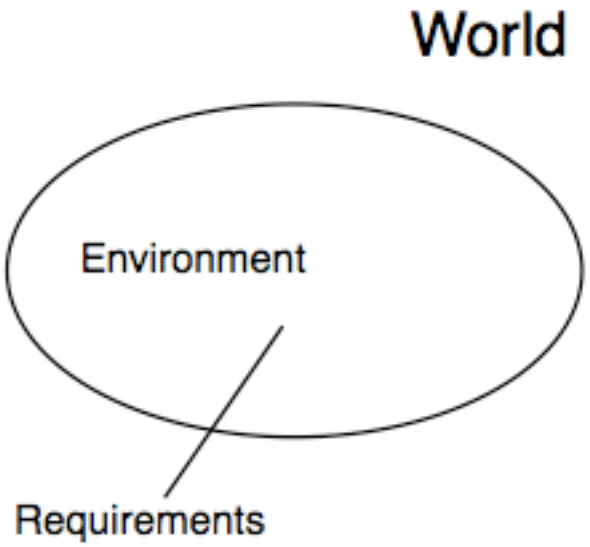
- We scope the *Environment* to include only those aspects of the real world that are relevant to the particular problem at hand

- The generalized environment for this *kind* of system is sometimes called the *application domain*
  - Examples: travel agency, online stores, telephony systems
  - A *domain model* is a diagram that shows how various kinds of domain entities relate to each other; it's usually drawn as a UML class diagram
    - We'll examine domain models in more detail later

Shared phenomena

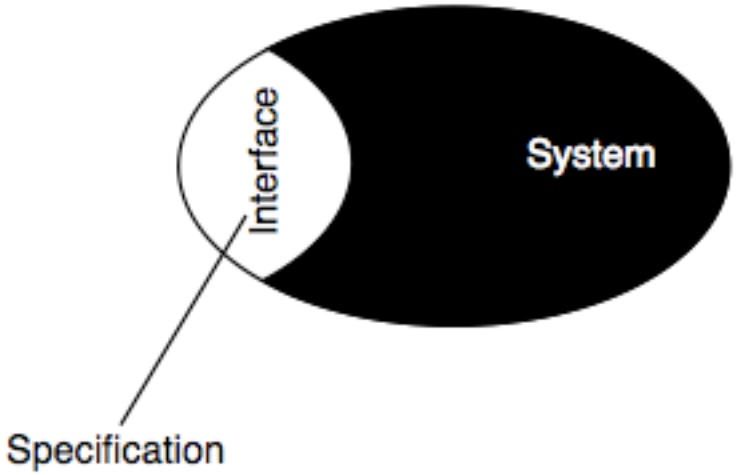
- *Shared Phenomena* are visible to both the Environment and the System, and form the *Interface* between the two.
  - A given interface entity may be sensed or controlled by the Environment or the System (but generally not by both).
    - It serves as a communication bridge from the one to the other.
  - Each such entity also has a concrete interface that describes how the system may interact with it (e.g, an API)
  - Anything that *has* to be in the Interface *has* to be shared by the Environment and the System.
  - Anything that *has* to be shared by the Environment and the System *has* to be in the Interface.

Requirements



- *Requirements* are desired changes to the World
  - They are expressed in terms of environmental phenomena:
    - No one should enter the park without paying
    - Anyone who has paid should be allowed to enter the park

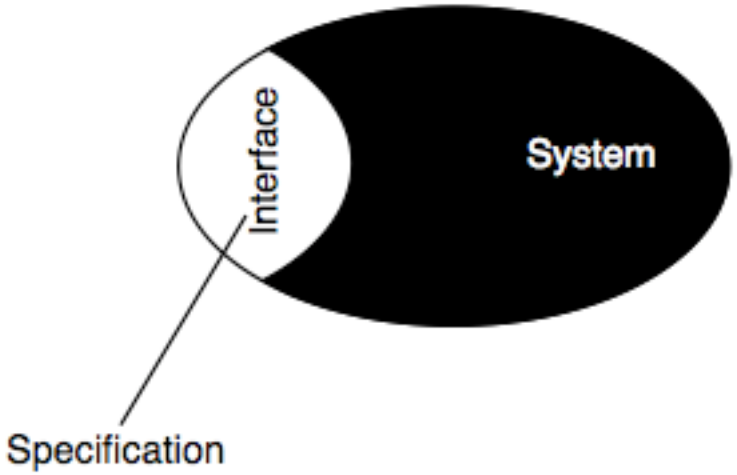
Requirements specification



- A *Requirements Specification* (aka "spec") is a description of the proposed behaviour of a CBS (Computer-Based System).
  - Want to avoid "implementation bias" in describing system.

  i.e., the spec should describe *what* the system is supposed to do, without indicating *how* the system will be realized.

Requirements specification



- A spec is expressed in terms of shared phenomena:
  - It describes how the system should react to various environmental events that it can sense.

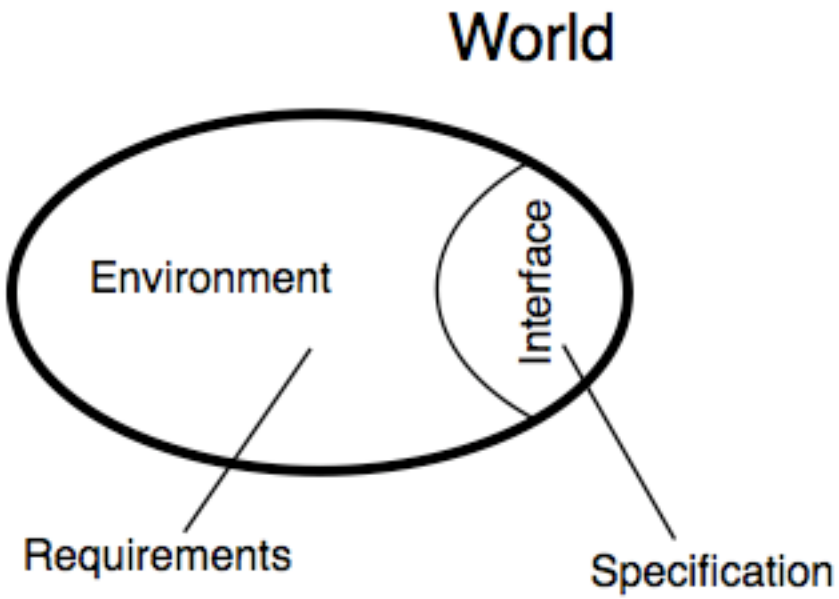
  e.g., if the number of coins inserted is greater than the number of times the turnstile has been pushed, then the turnstile is unlocked.

Requirements vs. specification

- *Requirements* are statements of *desired* properties
  - Often high level
  - May need to be elaborated, organized, analyzed
  - Heard during elicitation

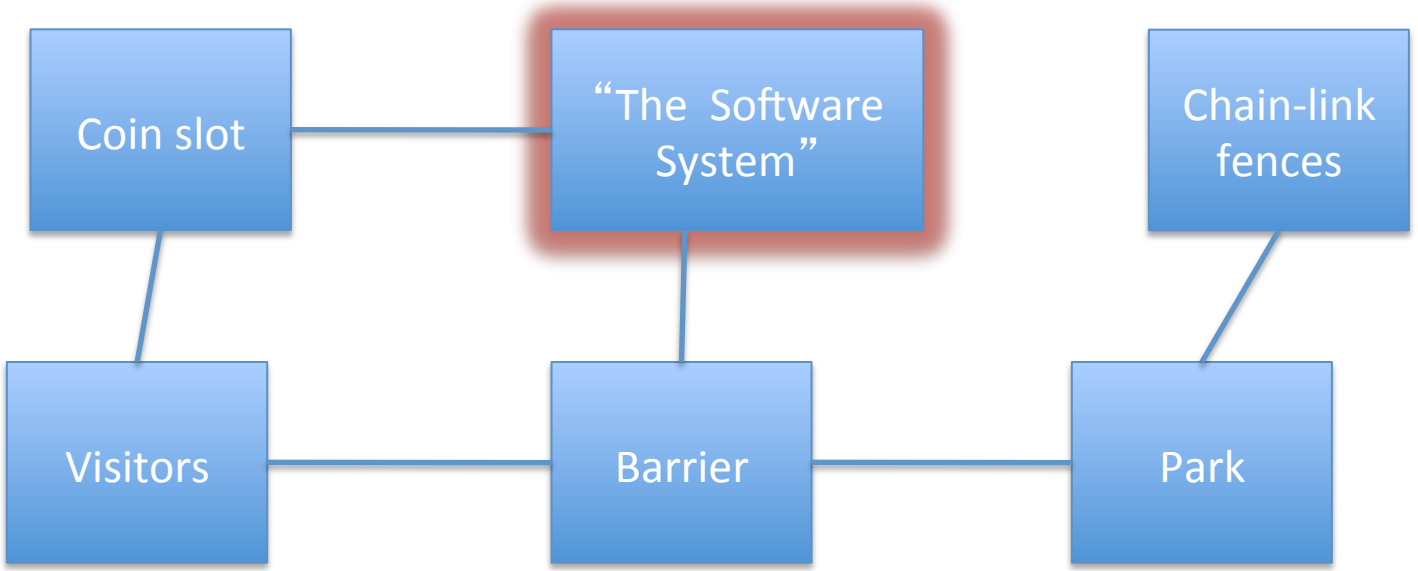
- A *specification* is a description of how the SUD will satisfy those properties, in terms of the shared phenomena
  - Concrete and detailed
  - Hammered out later on

Scoping the environment



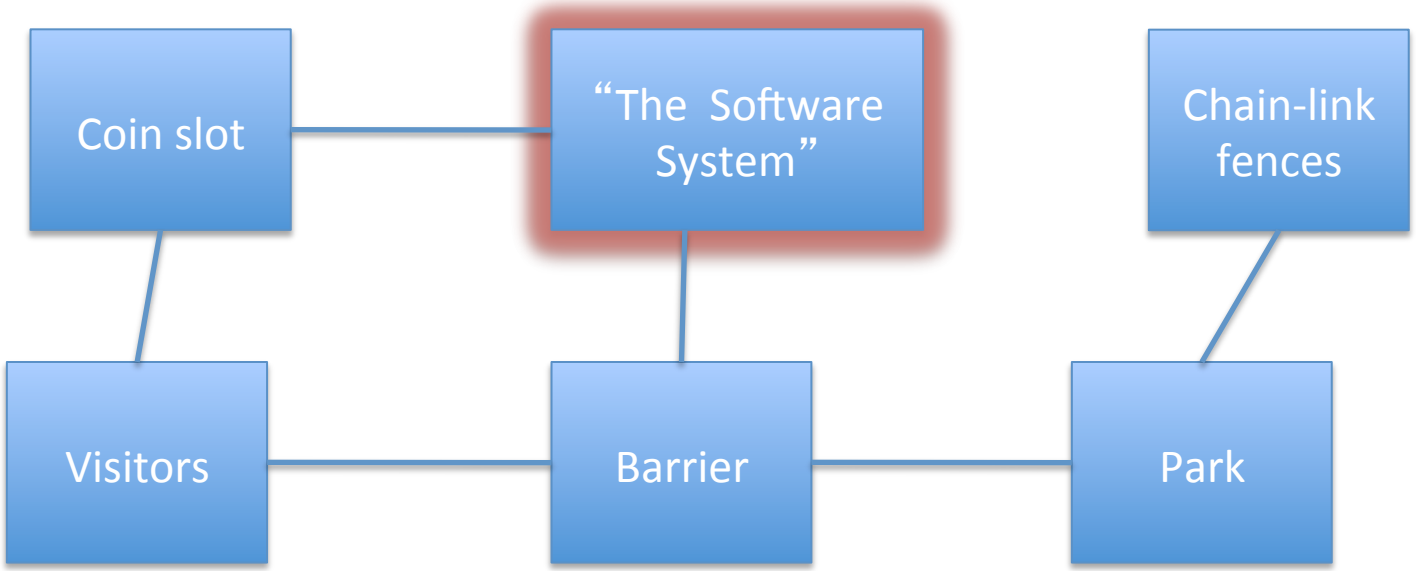
- The environment defines our area of discourse
  - It is a subset of the world
  - Want to model only as much of the world as is necessary to express the reqs and the spec

Context diagram

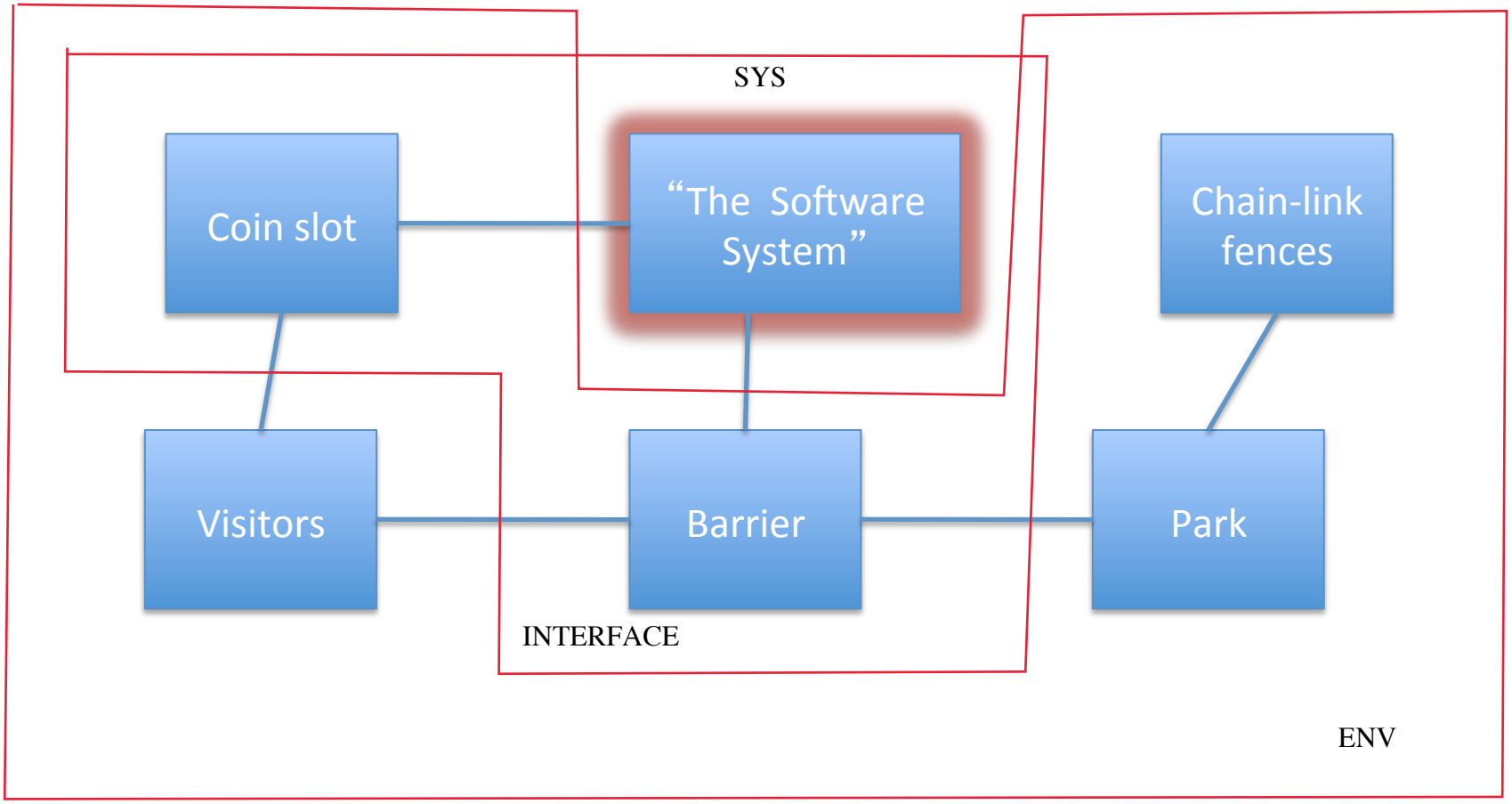


- A *Context Diagram* is a graphical model of the environment plus systems' s sub-domains.
  - A *sub-domain* is a coherent part of the environment plus system
  - Each sub-domain connected to the system shares phenomena with it
  - Sometimes need to know about sub-domains that don't directly interact with SUD

Context diagram



- Something in the Environment but not the System almost never has direct links to the Software System.
- Instead, it links directly to things in the Interface, which in turn, link directly to the Software System.

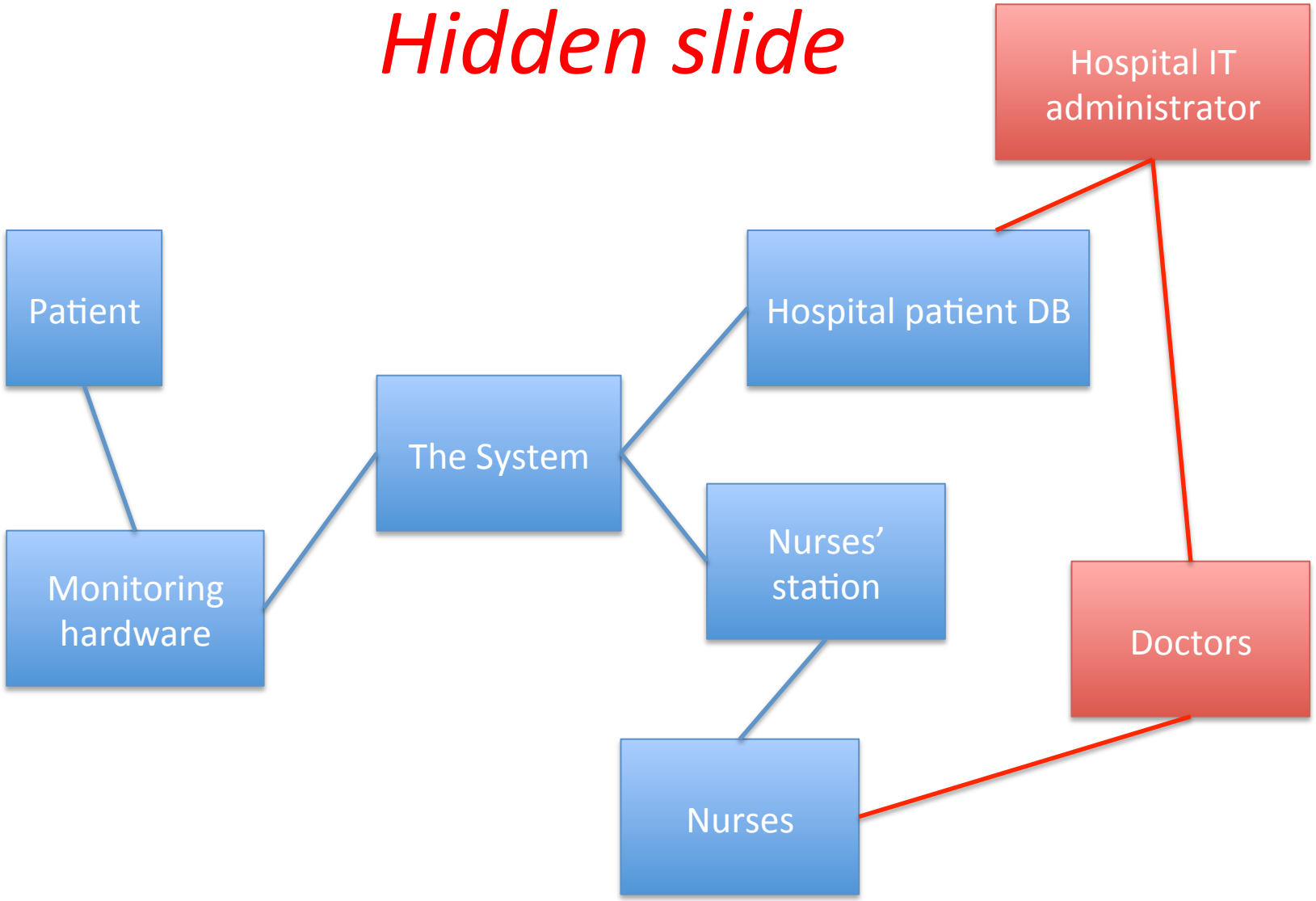

SYS
Coin slot
"The Software System"
Chain-link fences
Visitors
Barrier
Park
INTERFACE
ENV

Example: Patient Monitor

- Patients in an intensive-care ward in a hospital are monitored by electronic analog devices attached to their bodies by sensors of various kinds. Through the sensors, the devices measure the patient's vital factors: pulse rate, temperature, blood pressure, and so on. A program is needed to read the factors, at a frequency specified for each patient, and store them in a database. The factors read are to be compared with safe ranges specified for each patient, and readings that exceed the safe ranges are to be reported by alarm messages displayed on the screen of the nurse's station.

  [Stevens, Myers, and Constantine, "Structured Design", IBM Systems Journal, 13(2), 1974.]

Hidden slide



Red means "not in problem description, but subsequent analysis might reveal these subdomains as being of interest".

Hidden slide

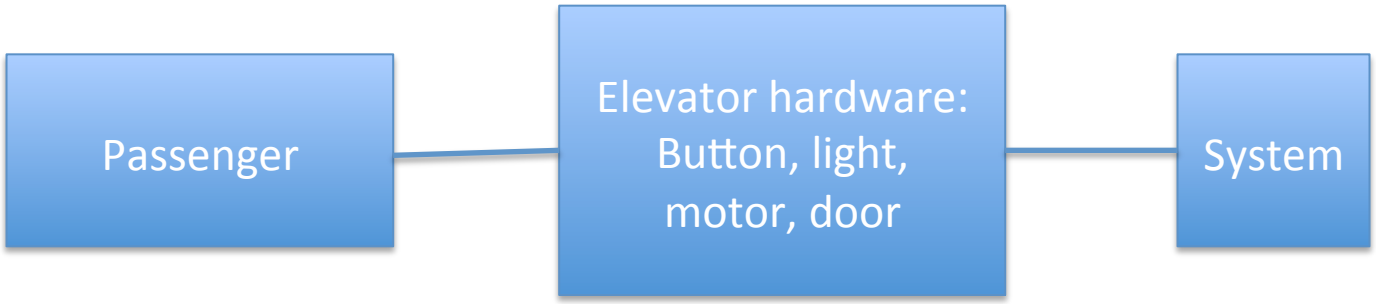
- Patients: context diagram not restricted to just components that interface with software.
  - include any sub-domain needed to talk about requirement
  - may include sub-domain that is part of the system if part of the system's interface (e.g., database).
    - Moreover, database may be a component that is used by other programs, whose requirements are specified elsewhere and are needed to satisfy requirements.

Example: Elevator

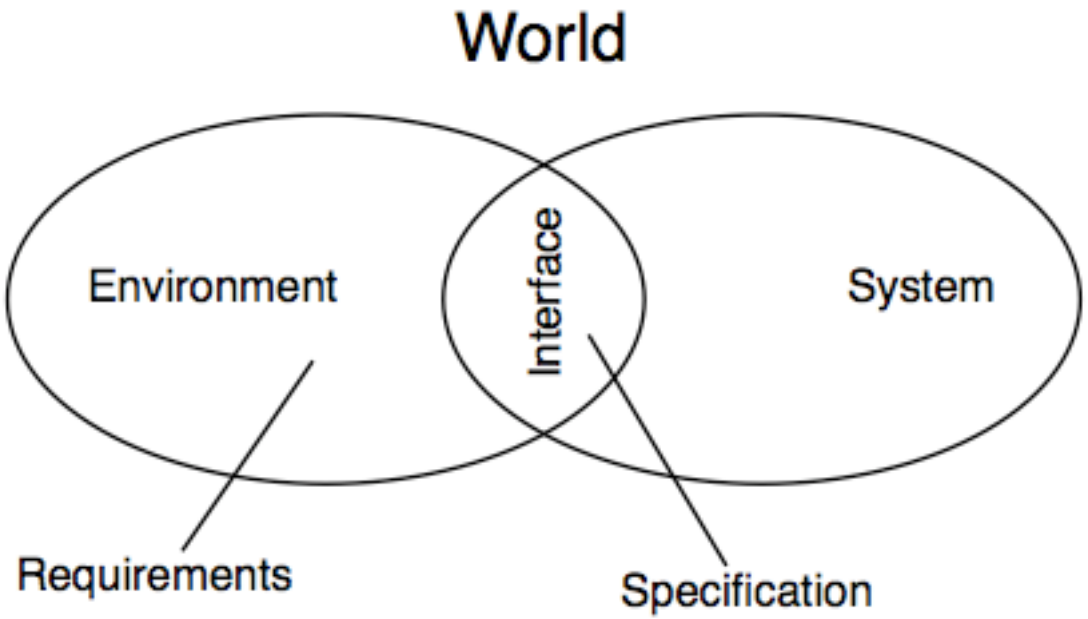
- An elevator passenger who wants to travel from one floor to another (higher) floor presses the "up" button at his current floor. The light beside the button must then be lit, if it was not lit before. The elevator must arrive reasonably soon, travelling in an upwards direction. The direction of travel is indicated by an arrow illuminated when the elevator arrives. The doors must open, and stay open long enough for the passenger to enter the elevator. The doors must never be open except with the elevator is stationary at a floor.

Michael Jackson, *Software Requirements and Specifications, Addison-Wesley, 1995.*

Hidden slide



Deriving specifications



- Deriving specifications is the process of identifying actions, functions, operations, and constraints on shared phenomena that achieve each of the requirements such that $S \vdash R$

Domain knowledge

- Requirements are concerned with describing things that we want the System to help make true.
  - The System might not be able to accomplish these things by itself.
  - Guarantees of properties of the Environment might be necessary for the System to meet the Requirements.

- These properties are called Domain Knowledge, D.

- *Domain Knowledge* is thus the set of properties that we know (or assume) to be true of the Environment that are relevant to the problem.

Elevator domain knowledge

- *The elevator is constrained to move in a shaft, so that it never goes from one floor to another without passing all the intermediate floors*
- *If the motor polarity is set to "up" and the motor is activated, then the elevator will rise.*
- *If the elevator arrives at a floor travelling upwards, the floor sensor switch is set on when the elevator is nine centimeters below the home position at the floor.*
- *The lift doors take 2250 msec to reach the fully closed state from the fully open state*

Domain knowledge

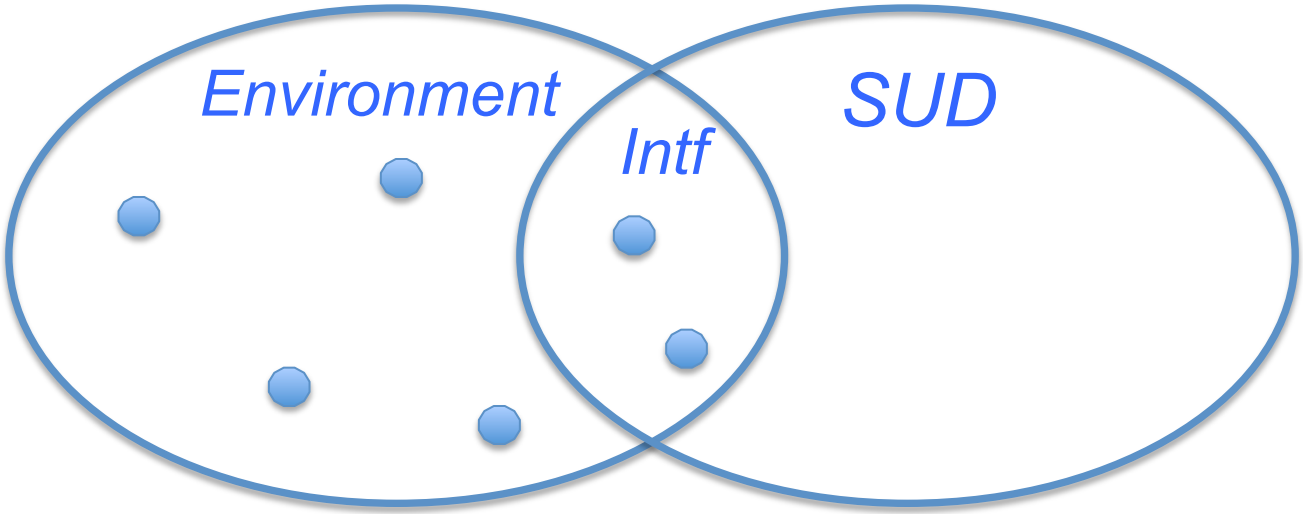
- The elevator specification will be expressed in terms of the shared phenomena:
  - states of the sensor switches, button pressings, setting and activations of the motor and doors, …
- Without domain knowledge, you could not ensure that any system you designed would be capable of satisfying the stated requirements!

Hidden: Traffic light example

- D = drivers behave legally and cars function correctly
- S = spec of traffic light that guarantees that perpendicular directions do not show green at same time
- R = perpendicular traffic does not collide

Problem: make D unnecessary, steel walls pop up on red, light controls cars by wireless

Reference model



R – Requirements live in ENV (incl. INTF)
S – Spec lives in INTF, describes behaviour of SUD
D – Domain knowledge lives in ENV (incl. INTF)

Reference model

- Thus, if we enlarge our model to include domain knowledge, then the following relationship must hold:

$$\mathbf{D, S \vdash R}$$

  - D is domain knowledge
  - S is the specification
  - R is the requirements

- The specification describes the behaviour of a system that realizes the requirements.
- The domain assumptions are needed to argue that any system that meets the specification (and that manipulates the interface phenomena) will satisfy the original requirements.

R, S, D, Design, & Code

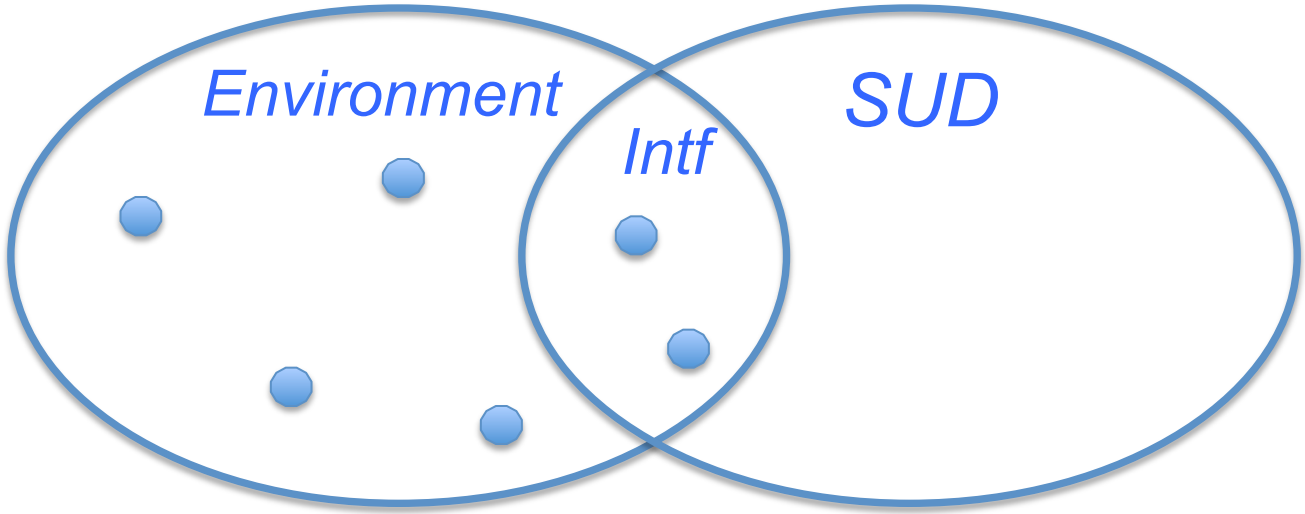
- R: If the user presses the “K” key, then he sees “K” on the screen.
- S: If the “K” key is pressed, then display “K” on the screen.
- D: The user has fingers with which to press keys and eyes with which to see.
- Design: A press of any key causes emission of an ASCII code that is used as an index into a table of bitmaps in a font table, the bit map that is put on screen.
- Code: C realization of the Design.

Reqs that live in only ENV – INTF?

Is there some notion of requirements that live in only ENV, saying only what is desired in the world, independent of any system that might achieve it?

We could call these “high-level requirements” or “goals”!

Add Goal = G, with R ⊢ G



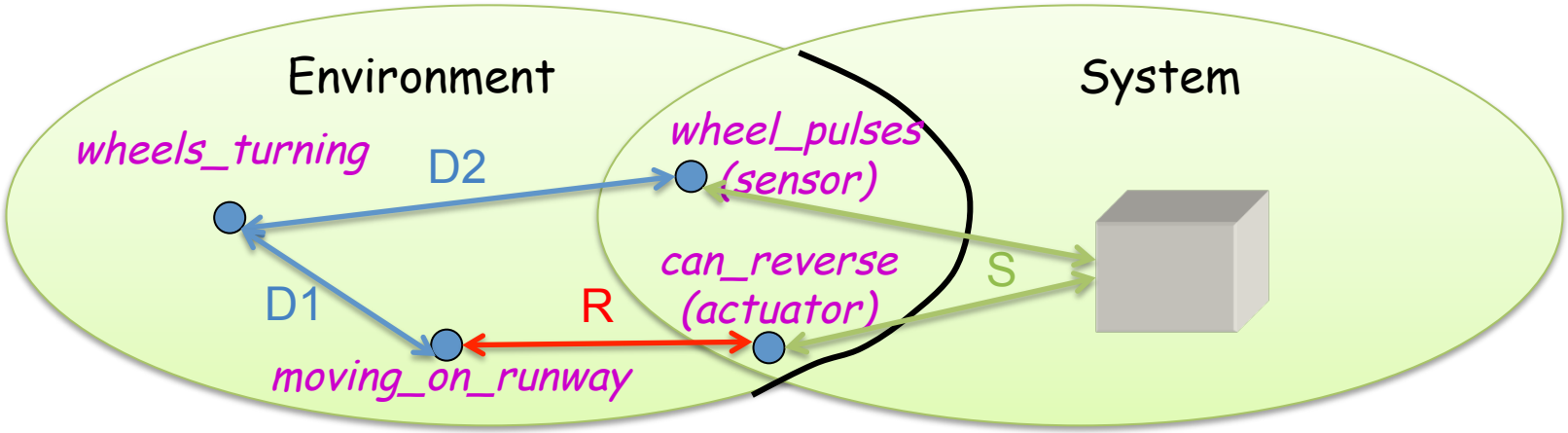
G – High Level Reqs, Goals live in ENV – INTF
R – Requirements live in ENV (incl. INTF)
S – Spec lives in INTF, describes behaviour of SUD
D – Domain knowledge lives in ENV (incl. INTF)

Reference model

- If you can't prove this, then at least one of 3 things must have gone wrong:
  - requirements are incorrect / unreasonable
  - system doesn't do enough
  - we aren't assuming enough about the environment

- A real world example: [M. Jackson]
  - An airplane overshot the runway on landing. The pilot had tried to engage reverse thrust, but the system wouldn't permit it. What's wrong?

A real world example



R: An airplane may engage reverse thrust iff it's moving on the runway
D1: Moving on runway iff wheels turning
D2: Wheel pulses detected iff wheels turning
S: Can reverse iff wheel pulses detected

Hidden slide

- The reason for the crash that the runway was wet, and the wheels were hydroplaning instead of turning.
  - Reverse thrust could only be engaged if pulses from the wheel sensors indicated that the wheels were turning.

- The developers made domain assumptions, but D1 was wrong.
  - If airplane is hydroplaning, then MOVING_ON_RUNWAY is true (and would like to engage reverse thrust), but WHEELS_TURNING is false.
  - The error was in the step from requirements to specification.

Correctness

Trivia: What's this symbol called?

- To evaluate a specification:

$$D, S \vdash R$$

  - Must be able to argue that the SUD spec plus the domain assumptions are enough to satisfy the requirements.

- If you can't make this argument successfully, then you need to do one (or more) of:
  1.
  2.
  3.

Hidden slide

1. strengthen the specification
2. strengthen the domain knowledge
3. weaken the requirements

Example: Train crossing

Req: train is in crossing ⇒ gate must be down

S1: if approaching train is 200m away, lower gate

Hidden slide

- Is S1 enough? (no), then give D1, then D2.
  - D1: gate can be lowered in 10 sec
  - D2: trains move more slowly that 200m/10s

- Yes, this is enough now ... but ...
  - Is this enough to be *safe?* Is the requirement reasonable?
  - What about speed of cars/humans who might be crossing tracks? Do they have enough time to clear? Will the crossing coming down interfere with their leaving?

Park example

- Suppose that that the city of Waterloo decides to raise funds by instituting users fees for public parks.
  - Must implement a complete system of money collection, security, etc.

- Informal requirement:
  - Collect $1 fee from each human park user on entry to park.
    - Ensure that no one may enter park without paying.
    - Ensure that anyone who has paid may enter the park.

- These are "pure" requirements:

  i.e., high-level goals, not stated in terms of interface of system

Park example: Possible solutions

Solution #1:

- Employ human fee collectors.
- Enforce perimeter security by instituting the Waterloo Park Militia, armed guards who ensure no one uses a park w/o paying a user fee.

Solution #2:

- Use chain link fences for security, use turnstiles with automated coin collection.
- After some research, we find appropriate turnstile hardware, but it's brand new technology so we must create the embedded software system.
  - There is a barrier through which to enter a park.
  - A person inserts a coin, the barrier unlocks, allowing the person to push the barrier and enter the park.

Park turnstile details

- The turnstile consists of a rotating barrier and a coin slot, and is fitted with an electrical interface.
- This mechanical apparatus has already been chosen, and the development job is to write the controlling software.
  - The software will run on a small computer; this is the SUD.
  - The environment is the turnstile mechanism itself and its use by visitors to the park.
- To enter the park, a visitor must first insert a coin and then push on the turnstile barrier, moving it to an intermediate position from which it will continue rotating of its own accord, returning to its initial position and gently pushing the visitor into the park.
- The turnstile is equipped with a locking device: when locked, it prevents the barrier from being pushed to the intermediate position. (It's not clear whether system is to lock the turnstile OR the turnstile locks itself after turning far enough to let one person in!)

Park example

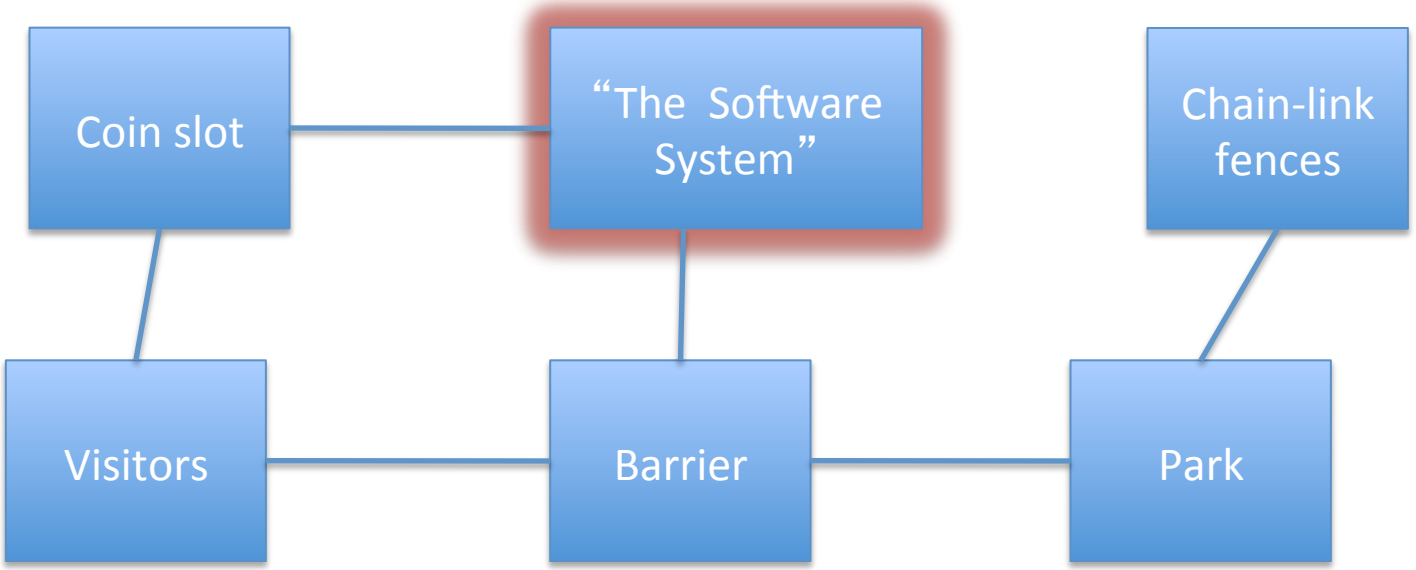
- Problem:
  - The requirements talk about visitors, coins, and the park
  - … but the system will interact only with the shared phenomena

    i.e., turnstile hardware: coin slot, barrier, including: lock, rotation detector, …

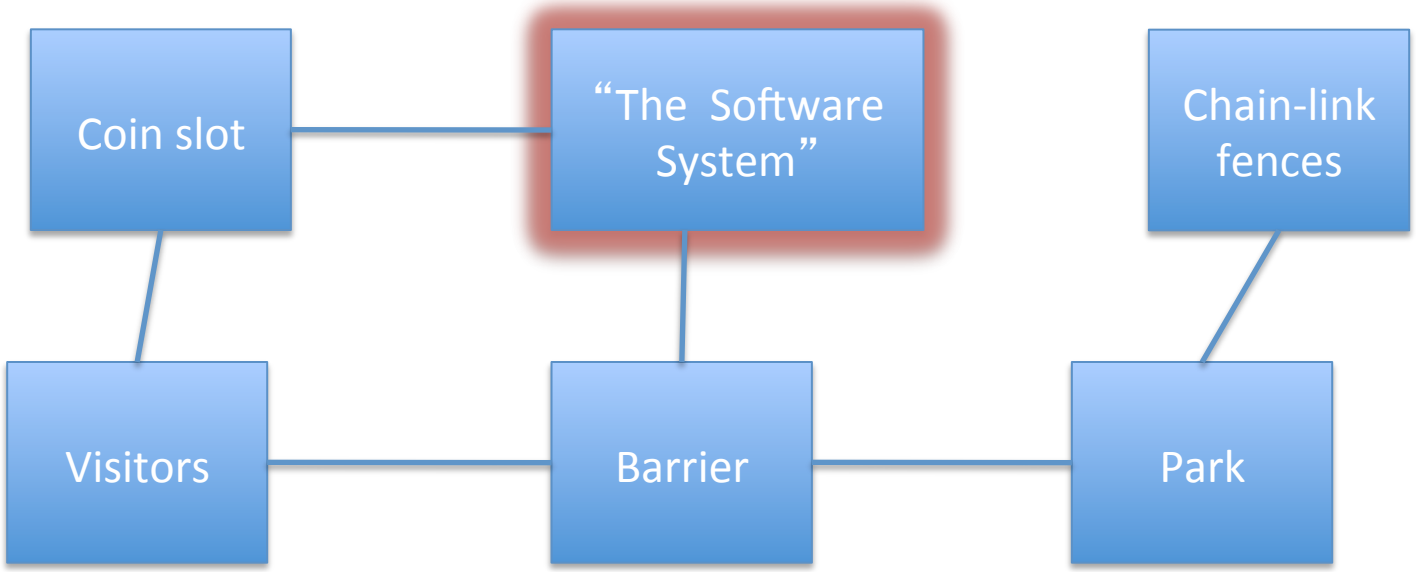
- Goal: Analyze the requirements that we have and "refine" them to a spec that we can (eventually) implement
  - Need to make assumptions about the environmental phenomena and how they can relate to system directly

  e.g., turnstile rotation detected means human entering park

Park / turnstile context diagram



Hidden slide



Park / turnstile example

- How do important events in the environment relate to events of the interface?
  - What are the important input events that the system needs to detect?
  - What are the important output events that the system needs to generate?

Designations

- A *designation* is a mapping between a term in the reqs or spec and the environmental phenomenon it represents
  - We are getting precise about terminology and what each term represents wrt the environment and SUD

Assuming that system is to lock the turnstile

Term	Kind	Meaning
Push	input event	*visitor pushes the barrier to its intermediate position*
Enter	input event	*visitor gains entry to park (barrier rotation complete)*
Coin	input event	*a valid coin is inserted into the coin slot*
Lock	output event	*turnstile instructed to lock barrier*
Unlock	output event	*turnstile instructed to unlock barrier*
locked	internal state	*barrier is locked and cannot be pushed*
unlocked	internal state	*barrier is unlocked and can be pushed*

Designations

- A *designation* is a mapping between a term in the reqs or spec and the environmental phenomenon it represents
  - We are getting precise about terminology and what each term represents wrt the environment and SUD

Assuming that the turnstile locks itself after turning enough to let one person in

Term	Kind	Meaning
Push	input event	*visitor pushes the barrier to its intermediate position*
Enter	input event	*visitor gains entry to park (barrier rotation complete)*
Coin	input event	*a valid coin is inserted into the coin slot*
Unlock	output event	*turnstile instructed to unlock barrier*
locked	internal state	*barrier is locked and cannot be pushed*
unlocked	internal state	*barrier is unlocked and can be pushed*

Park / turnstile system interface

- A coin slot
  - system *observes* and
  - env *controls and observes*

  coin entering the coin slot (*Coin*)
- A barrier
  - locking and unlocking
    - system *controls and observes* and
    - env *observes*

    whether the barrier is locked or unlocked (*Lock* / *Unlock*)
  - motion
    - env *controls and observes* and
    - system *observes*

    the pushing of barrier (*Push, Enter*)

Specification

- The specification describes what needs to occur in terms of the shared phenomena

$$Spec \subseteq Env \cap Sys \quad (= Intf)$$

- The proper way to read this is that the vocabulary in which the specification is written must be a subset of the vocabulary of the interface, i.e., a subset of the vocabularies shared by the environment and the system.

Specification

- The specification describes what needs to occur in terms of the shared phenomena

$$Spec \subseteq Env \cap Sys \quad (= Intf)$$

- Example Specification:
  - If a coin has been inserted into the coin slot, then barrier is unlocked in a way that it can be pushed one rotation.

  Note that this doesn't say:

  - If a visitor puts a coin into the coin slot, then (s)he can push the barrier one rotation.
    - That references parts of *Env* that aren't in *Intf* and is a requirement!

Examples

Requirements:

1. No one should enter the park without paying
2. Anyone who has paid should be allowed to enter the park

Specifications:

1. Barrier is *locked* if

   # coins inserted so far < # of enters so far

2. Barrier is *unlocked* if

   # coins inserted so far >= # of enters so far

– How can we detect payments?

– How can we detect/control entry? [Not done yet!]

Domain knowledge

- The domain assumptions include the following facts and assumptions about the environment.
  1. coin = entrance fee.
  2. if someone pushes thru the barrier, (s)he will eventually enter the park (push leads to enter)
  3. if barrier is unlocked and pushed, it rotates enough to allow 1 visitor to pass, and then it locks (If barrier locks itself)

- Thus, we realize requirements in two ways:
  1. building a system that performs the specified actions, and
  2. making assumptions about how the environment will behave.

Domain knowledge

For every event / action, need to decide:

1. Can the system observe the event / action?
   (*i.e.,* is this event / action part of the interface)

2. If part of the intf, is the event / action is controlled by
   - the system, or
   - the environment?

3. If part of the environment, are there any domain assumptions about the event / action?

Uncertainty in "D, S ⊢ R"

- The formula D, S ⊢ R tries to be formal in the sense of describing what happens completely.
- One would expect computers and software and their combination to be formal in this sense.
- But, the real world intervenes to make this formula only a guideline and not an accurate, precise model.

Hidden: Uncertainty in "D, S ⊢ R"

- First, the real world *never* behaves as *any* model.
- Any model D is only an approximation.
- Generally, the simpler the model, the more of an approximation the model is, but the easier it is to prove things about the model.
- Modeling the real world accurately requires complexity to deal with all the weird exceptions.
- A mechanistic description generally has to be replaced by or tempered with a probabilistic model, e.g., 99.99% of drivers stop at a red light.

Hidden: Uncertainty in "D, S ⊢ R"

- At the lowest level, a CBS is mechanistic, e.g., a traffic light, the sqrt function, and can be modeled with a consistent S that is mechanistic, that always gives for any input the same answer that the CBS does.
- But floating point arithmetic is not the same as real numbers, and integer arithmetic suffers over & underflow.
- At higher levels, e.g., MS Word, an operating system, process control, etc., the CBS is so large that we cannot understand all of its code and all of its behavior. So, we begin to give probabilistic models of what the CBS does.

Hidden: Uncertainty in "D, S ⊢ R"

- All that applies to D, applies to R, because both are models of the real world, one as is, and the other as it is to be.
- R is always an approximation of what we want, because if we overlook something in the real world and it turns out to be relevant to the CBS's behavior, e.g., a gaggle of Canadian geese that fly near a jet engine, then R may not be correct.

Uncertainty in "D, S ⊢ R"

- The formula D, S ⊢ R tries to be formal in the sense of describing what happens completely.
- But, as we have seen, it cannot be completely formal because at least D and R have to describe the real world, which is not formal

What does this do to the hope of formally modeling computer systems?

Molecular Software

- Molecular SW, e.g., DNA, RNA, Proteins, Catalysts
- Molecules designed specifically to achieve a desired effect
- Molecule is shown empirically to behave as specified in S, with 99.95% certainty
- In this case, in D, S $\vdash$ R, also S is informal!

CS445 / SE463 / ECE 451 / CS645

Software requirements specification & analysis

A reference model for requirements engineering

Fall 2015

Mike Godfrey & Daniel Berry & Richard Trefler