11. More on Requirements: The EARS approach and the Doorstop tool

David Pereira José Proença Eduardo Tovar

RAMDE 2021/2022

Requirements and Model-driven Engineering

CISTER – ISEP
Porto, Portugal

https://cister-labs.github.io/ramde2122

The EARS Approach to Requirements Specification

Getting to Know EARS

What is EARS?

- The acronym **EARS** stands for *"Easy Approach to Requirements Syntax"*
- EARS is a mechanism to gently constrain textual requirements
- EARS patterns provide structured guidance that enable authors to write high quality textual requirements.

Building Blocks

- There is a set syntax (structure), with an underlying ruleset.
- A small number of keywords are used to denote the different clauses of an EARS requirement.
- The clauses are always in the same order, following temporal logic.
- The syntax and the keywords closely match common usage of English and are therefore intuitive.

How EARS Came to Life

Context

- **When:** while the author and colleagues at Rolls-Royce PLC were analysing the airworthiness regulations for an engine jet control system.
- **"Inputs":**The regulations contained high level objectives, a mixture of implicit and explicit requirements at different levels, lists, guidelines and supporting information.
- **How?:**
  - In the process of extracting and simplifying the requirements, Mav noticed that the requirements all followed a similar structure.
  - He found that requirements were easiest to read when the clauses always appeared in the same order. These patterns were refined and evolved to create EARS.
- **Inception:** The notation was first published in 2009 and has been adopted by many organisations across the world.

Motivations for Adopting EARS (1/2)

Why adopt EARS?

- System requirements are usually written in unconstrained natural language, which being to the table its inherent imprecision and ambiguity.
- It is not unusual that authors of requirements have no training on how to write requirements.
- During system development, requirements problems propagate to lower levels.

This creates unnecessary volatility and risk, impacting schedules and costs.

Motivations for Adopting EARS (2/2)

Why adopt EARS?

- EARS reduces or even eliminates common problems found in natural language requirements.
- It is especially effective for requirements authors who must write requirements in English, but whose first language is not English.
- EARS has proved popular with practitioners because it is lightweight, there is little training overhead, no specialist tool is necessary, and the resultant requirements are easy to read.

In the words of the author, **"(...) because the EARS method imposes just a slight constraint on natural language while providing a simple, logical method for constructing clear, concise, unambiguous requirements."**

On the adoption of EARS

Who is using EARS?

- the EARS methodology was first presented to the 17th IEEE International Requirements Engineering Conference in 2009
- since then, it has been adopted by numerous organizations (Bosch, Honeywell, Intel, Rolls-Royce and Siemens) and included in the requirements engineering curricula of many universities (China, France, Sweden, UK, USA, and now Portugal ☺).

The EARS Patterns

EARS Patterns - Overview

Vocabulary

`<system>`	the system name (only 1 per requirement)
`<response>`	the system response (1 or more per requirement)
`<pre>`	a precondition, *i.e.*, a set of properties from a state that need to be true for the requirement to be active (zero or many per requirement)
`<trigger>`	the trigger that activates a requirement (zero or one per requirement)
`<feature>`	a feature of the system (zero or one)

Specific keywords

while, when, where, if, then, the, shall... we will see in a moment when and where they are used.

Ubiquitous requirements

These refer to requirements that must be always active during system operation.

```
the <system> shall <response>
```

Moreover:

- typically state fundamental aspects of the system
- No EARS specific keyword is present when specifying this particular type of requirement.
- Which makes sense! These requirements do not depends on pre-condition(s) or triggers to become active. They must remain active all the time.

Ubiquitous Requirements Examples

Example

- **the** distances computed between two sets of coordinates **shall** account for curvature of the earth
- **the** compiler **shall** transform source code into semantically equivalent binary code
- **the** surveillance UAV **shall** fly only inside of the designated flight zone
- **the** software package **shall** contain an installer
- **the** software **shall** be written in programming language X

State driven requirements

These are requirements that are active as long as the specified state, hereby represented by a pre-condition, remains true. These requirements start with the keyword while.

Syntactic pattern

```
while <pre(s)> the <system> shall <response>
```

Examples of State Driven Requirements

Example

- **while** there is no card in the ATM **the** ATM **shall** display "insert card to begin".
- **while** in maintenance mode **the** kitchen system **shall** reject all input.
- **while** in Low Power Mode **the** software **shall** keep the display brightness at the Minimum Level
- **while** the heater is on **the** software **shall** close the water intake valve
- **while** the autopilot is engaged **the** software **shall** display a visual indication to the pilot

Event driven requirements

Event driven requirements are initiated **when and only when** a trigger occurs or is detected. They are denoted by the keyword **when**.

Syntactic pattern

```
when <trigger> the <system> shall <response>
```

Event driven requirements

Event driven requirements are initiated **when and only when** a trigger occurs or is detected. They are denoted by the keyword **when**.

Syntactic pattern

```
when <trigger> the <system> shall <response>
```

Simple translation exercise

THE EARS Patterns - Event Driven Requirements

Event driven requirements

Event driven requirements are initiated **when and only when** a trigger occurs or is detected. They are denoted by the keyword **when**.

Syntactic pattern

```
when <trigger> the <system> shall <response>
```

Simple translation exercise

Original req: *In the event of a fire, the security system shall Unlock the fire escape doors*

THE EARS Patterns - Event Driven Requirements

Event driven requirements

Event driven requirements are initiated **when and only when** a trigger occurs or is detected. They are denoted by the keyword **when**.

Syntactic pattern

```
when <trigger> the <system> shall <response>
```

Simple translation exercise

Original req: *In the event of a fire, the security system shall Unlock the fire escape doors*

In EARS: **when** *a fire is detected* **the** *security system* **shall** *unlock the fire escape doors*

Examples of Event Driven Requirements

Example

- **when** mute is selected **the** laptop **shall** suppress all audio output.
- **when** potato is inserted into the input hatch **the** kitchen system **shall** peel the potato.
- **when** continuous ignition is commanded by the aircraft **the** control system **shall** switch on continuous ignition
- **when** an unregistered device is plugged into a USB port **the** OS **shall** tries to locate and load the driver for the device.
- **when** the water level falls below the Low Water Threshold **the** software **shall** open the water valve to fill the tank to the High Water Threshold

Optional feature requirements

Optional feature requirements apply in products or systems that include the specified feature and are denoted by the keyword Where.

Syntactic pattern

```
where <feature> the <system> shall <response>
```

THE EARS Patterns - Optional feature requirements

Example

- **where** the car has a sunroof **the** car **shall** have a sunroof control panel on the driver door.
- **where** the kitchen system has a food freshness sensor **the** kitchen system **shall** detect rotten foodstuffs.
- **where** a thesaurus is part of the software package **the** installer **shall** prompt the user before installing the thesaurus
- **where** hardware encryption is installed **the** software **shall** encrypt data using the hardware instead of using a software algorithm
- **where** a HDMI port is present **the** software **shall** allow the user to select HD content for viewing

Unwanted behaviour requirements

These are used to specify the required system response to undesired situations and are denoted by the keywords if and then.

Syntactic pattern

```
if <trigger> then the <system> shall <response>
```

A note on "trigger"

As in the case of event requirements, for unwanted behaviours one needs to identify the trigger/event. It is on you, the requirement specification responsible to understand if it refers to something wanted or unwanted. EARS just ensures a syntactic distinction, i.e., using **when** or **if** depending on the concrete case.

THE EARS Patterns - Unwanted behaviour requirements

Example

- **if** an invalid credit card number is entered **then the** website **shall** display "please re-enter credit card details"
- **if** a spoon is inserted to the input hatch **then the** kitchen system **shall** eject the spoon
- **if** the memory checksum is invalid **then the** software **shall** display an error message
- **if** the ATM card inserted is reported lost or stolen **then the** software **shall** confiscate the card
- **if** the measured and calculated speeds vary by more than 10% **then the** software **shall** use the measured speed

EARS Patterns - Complex Requirement Pattern

Syntax of a complex EARS requirement

Is the more general pattern of requirement that exists in EARS. It uses a combination of EARS keywords to allow for such complexity.

Syntactic pattern

```
<multiple conditions> the <system> shall <response>
```

such that `multiple conditions` is a combination of:

- a pre-condition, using the **while** keyword
- a trigger, using the **when** keyword
- an unwanted condition, using the **if** keyword
- a specific feature, using the **where** keyword

THE EARS Patterns - Example of Complex Requirements

Example

- **while** the aircraft is on ground **when** reverse thrust is commanded **the** engine control system **shall** enable reverse thrust
- **while** a second optical drive is installed **when** the user selects to copy disks **the** software **shall** display an option to copy directly from one optical drive to the other optical
- **while** in start up mode **when** the software detects an external flash card **the** software **shall** use the external flash card to store photos
- **when** the landing gear button is depressed once **if** the software detects that the landing gear does not lock into position **then the** software **shall** sound an alarm

Exercises

Ex. 11.1: Rewrite using EARS patterns

1. The user can have tea after having 2 consecutive coffees.
2. It is possible to do *a* after 3 *b*'s, but not more than 1 *a*.
3. It must be possible to do *a* after [doing *a* and then *b*].
4. If a taxi is allocated to a service, it must first collect the passenger and then plan the route.
5. On detecting an emergency the taxi becomes inactive.
6. The user can only have coffee after the coffee button is pressed.
7. The used must have coffee after the coffee button is pressed.
8. It is always possible to turn off the coffee machine.
9. It is always possible to reach a state where the coffee machine can be turned off.
10. It is never possible to add chocolate right after pressing the *latte button*.

Applying and Troublshooting EARS

How to apply EARS

- Identify whether you are working with a requirement, or something else (e.g., note, example, remark, etc)
- Identify compound requirements, i.e., whether the requirement needs to be split/decomposed
- Identify the acting system, person, or process
- Analise the needed sentence type(s)
- Identigy possible missing requirements
- Analyse the translated requirements for ambiguity, conflict, and repetition
- Review requirements if possible
- Interate as required

What are the issues that can occur when using EARS?

- *No sentence type fits:* Maybe you are not translating a requirement?
- *Can't identify the actor:* either use higher abstraction level until it makes sense, or get more information from the relevant stakeholder
- *There is no system response:* typically the case with non-functional requirements; it can be expressed as "the system shall be ..."
- *There is no template for "shall not":* try using "shall be immune" or similar or, as last resort, use the "shall not" pattern
- *EARS produces too many atomic requirements:*

The Doorstop Tool

What is Doorstop?

In a Nutshell

- its is both a Python tool and API that allows to write requirements in a text based manner and that uses version control.
- This solution allows a project to utilize its existing development tools to manage versions of the requirements using a lightweight, developer-friendly interface.

Doorstop was created to enable the utilization of existing version control systems and usage of tools developers are already familiar with (namely, the command line and text editors).

What is Doorstop?

How it works (high-level description)

- When a project leverages this tool, each linkable item (requirement, test case, etc.) is stored as a YAML file in a designated directory.
- The items in each directory form a document.
- The relationship between documents forms a tree hierarchy.
- Doorstop provides mechanisms for modifying this tree, validating item traceability, and publishing documents in several formats.

Where to download and install

Requirements

- Python 3.5+
- version control system

Installing

- pip install doorstop

Running from Docker

- access `https://github.com/doorstop-dev/docker-doorstop` and follow the instructions on how run the Docker container

Documentation

`https://doorstop.readthedocs.io`

Doorstop - creating documents

Parent Document

```
$ doorstop create REQ ./reqs
created document: REQ (@/reqs)
```

Child Document

```
$ doorstop create TST ./reqs/tests --parent REQ
created document: TST (@/reqs/tests)
```

Doorstop - adding and editing documents

Adding items/requirements

```
$ doorstop add REQ
added item: REQ001 (@/reqs/REQ001.yml)
```

Editing items/requirements

```
$ doorstop edit REQ1
opened item: REQ001 (@/reqs/REQ001.yml)
```

Doorstop - linking items

Example

```
$ doorstop create REQ ./reqs
created document: REQ (@/reqs)
$ doorstop add REQ
added item: REQ001 (@/reqs/REQ001.yml)
$ doorstop create TST ./reqs/tests --parent REQ
created document: TST (@/reqs/tests)
$ doorstop add TST
added item: TST001 (@/reqs/tests/TST001.yml)
$ doorstop link TST1 REQ1
linked item: TST001 (@/reqs/tests/TST001.yml) -> REQ001 (@/reqs/REQ001.yml)
```

Doorstop - validating and publishing

Validation

```
$ doorstop
```

Publishing as text

```
$ doorstop publish TST
```

Publishing as HTML

```
$ doorstop publish all ./dist/
```

Validation warnings

REQ00X: no text

All requirements should have some description.

REQ00X: no links from child document: CHILD

Each requirement in a parent should be linked from some child requirement.

CHILD00X: no links to parent document: REQ

Each child requirement should link to some parent requirement.

REQ00X: unreviewed changes

A requirement was change and not marked as “reviewed”.

More online:

`https://doorstop.readthedocs.io/en/latest/cli/validation/`

Lets go online now...

We will now replicate the commands presented in the CLI to see how doorstop behaves...

Some exercises for training with EARS

The problem

A farmer wants to transport a fox, a goose, and some beans across a river (from the left margin to the right margin). Unfortunately, he can only carry one at a time. Furthermore, if the farmer is not present, the fox will eat the goose and the goose will eat the beans.

Ex. 11.2: Identify requirements

The goal of this exercise is for you to identify the requirements for this problem, and classify and write them using the EARS patterns. If necessary, elicit other requirements that are not in the text but that should be present.

Ex. 11.3: Another scenario that you've seen in the classes

Lets consider a vending machine with 2 products, apples and bananas, costing 1eur and 2eur respectively. Its users have only 1eur and 2eur coins to interact with the machine. Now, write using EARS patterns the following requirements:

- The user must be able to get apples and bananas;
- The machine accepts up to 3eur, and not more than that;
- The machine must give change back when applicable;
- The machine can be powered off and powered on;

Now, a more complex scenario

The problem

I would like the vending machine to sell 3 items: apples, bananas, and chocolates. It should be possible to buy chocolates for 2€ and fruit for 1€ . Only 1€ and 2€ coins are accepted. The machine has a maximum capacity for 1€ coins and for 2€ coins. The machine does not accept coins if its capacity is full. The machine should give change back when buying fruit after inserting 2€. If the machine has already 2€ inserted, it refuses another coin. If the machine has no 1€ coins, it cannot not sell fruit with a 2€ coin. The user can request the money back after inserting coins.

Ex. 11.4: Identify requirements

Proceed with identifying requirements, classifying and writing them following the EARS patterns, and use the doorstop tool to produce the corresponding requirement specification requirements.