SEG 2105 - LECTURE 04

Developing Requirements

SOFTWARE REQUIREMENTS

DOMAIN ANALYSIS

A **domain** is a field of study that defines a set of **common requirements**, **terminology**, and **functionality** for any software program constructed to solve a problem in the area of computer programming, known as domain engineering. The word domain is also taken as a synonym of **application domain**.

https://en.wikipedia.org/wiki/Domain_(software_engineering)

A **subject-matter expert (SME)** or **domain expert** is a person who is an **authority** in a particular area or **topic**. A domain expert is a person with **special knowledge** or skills in a particular area of endeavour.

The ***development of accounting software*** requires **knowledge** in two different domains: **accounting** and **software**.

https://en.wikipedia.org/wiki/Subject-matter_expert

Domain analysis is the process of **analyzing related software systems** in a domain to find their **common** and **variable** parts. It is a model of **wider business context** for the system.

https://en.wikipedia.org/wiki/Domain_analysis

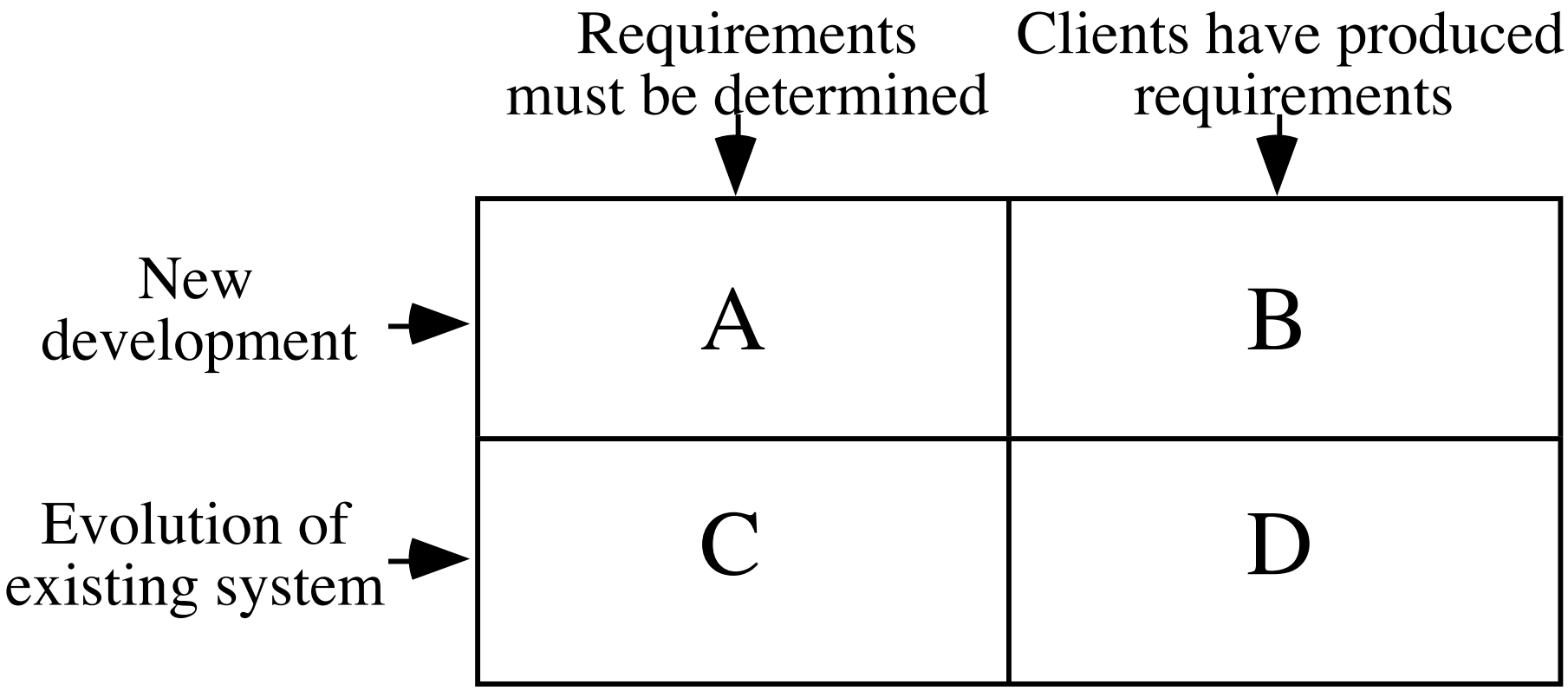
Benefits of Domain Analysis

- Faster development
- Better system
- Anticipation of extensions

Domain Analysis Template

- Summary
- Glossary
- General Knowledge
- Customers and Users
- The Environment
- Tasks and Procedures
- Competing Software
- Similarities to other domains

Starting Point for Software Projects



SOFTWARE REQUIREMENTS

DEFINING THE PROBLEM

Expressing the problem

- A difficulty the users or customers are facing,
- Or as an opportunity that will result in some benefit such as improved productivity or sales.
- A good problem statement is short and succinct

The **solution** to the problem normally will entail **developing software**

SOFTWARE REQUIREMENTS

DEFINING THE SCOPE

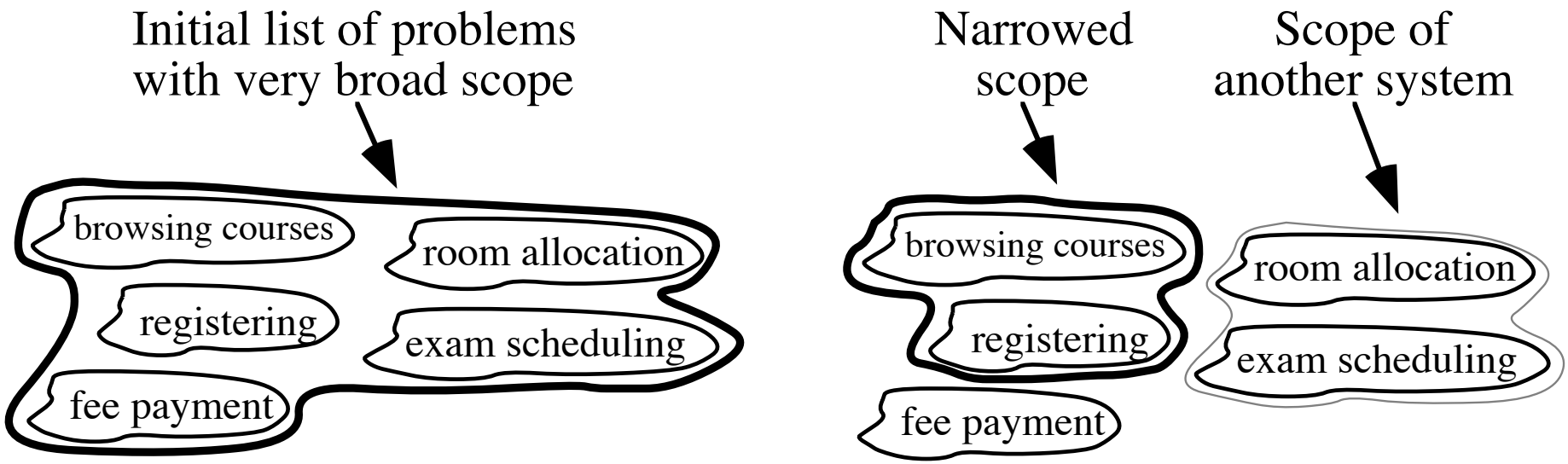
Scope involves getting **information** required to **start a project**, and the **features** the product would have that would **meet** its stakeholders **requirements**. The features and functions that **characterize a product, service, or result**."

https://en.wikipedia.org/wiki/Scope_(project_management)

Narrow the scope by defining a more precise problem

- List all the things you might imagine the system doing
- Exclude some of these things if too broad
- Determine high-level goals if too narrow

EXAMPLE: A UNIVERSITY REGISTRATION SYSTEM



SOFTWARE REQUIREMENTS

WHAT IS A REQUIREMENT?

A **requirement** is a singular documented **physical** or **functional need** that a particular design, **product** or process **aims to satisfy**. It is a broad concept that could speak to any necessary (or sometimes desired) **function**, **attribute**, **capability**, **characteristic**, or **quality of a system** for it to have **value** and utility to a **customer**, organization, internal user, or other stakeholder.

https://en.wikipedia.org/wiki/Requirement

A STATEMENT DESCRIBING EITHER

A. An aspect of what the proposed system **must do**

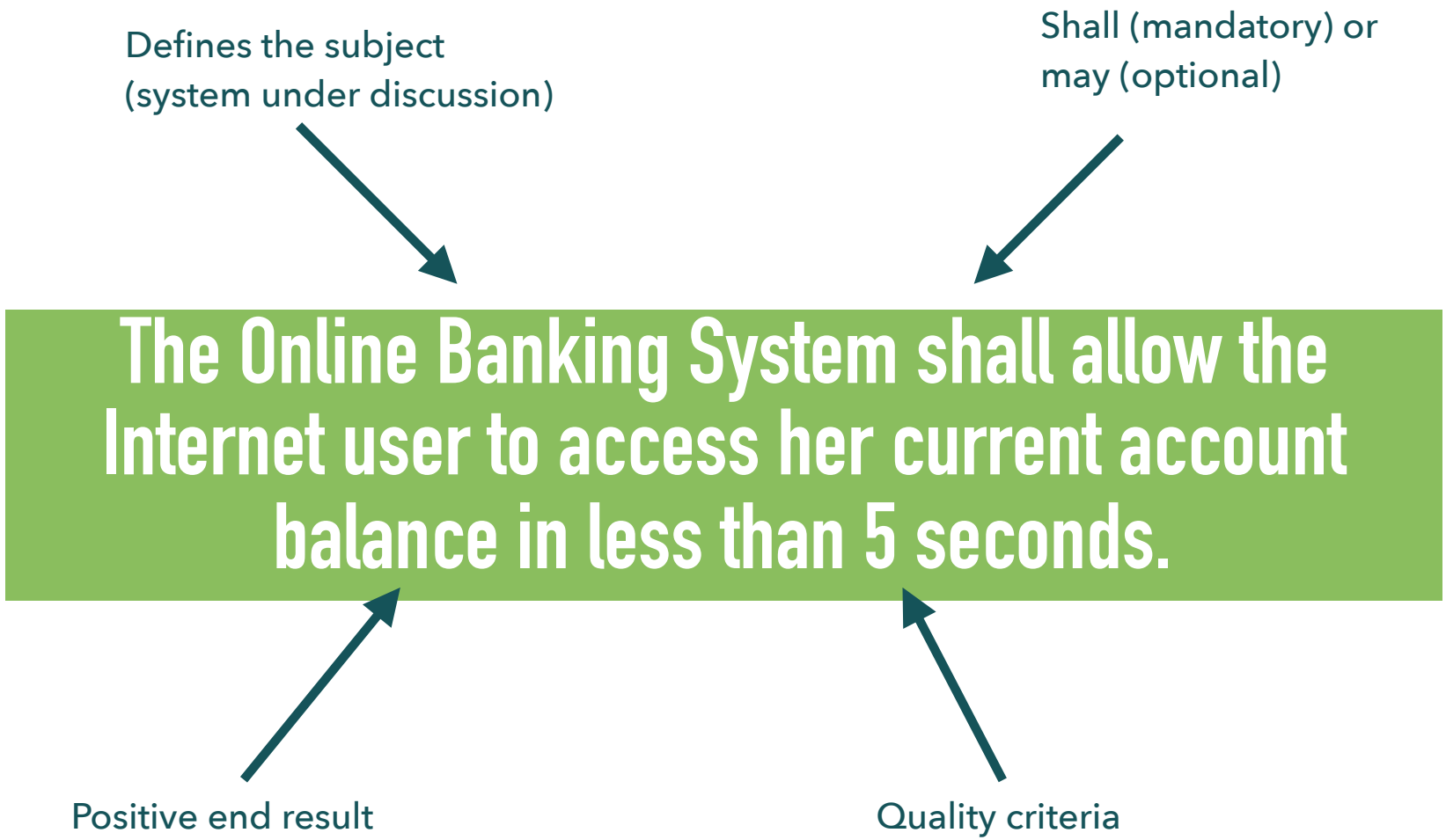
B. A **constraint** on the system's development

In either case it must **contribute** in some way towards adequately **solving** the customer's **problem**;

The **set of requirements** as a whole represents a **negotiated agreement** among the stakeholders.

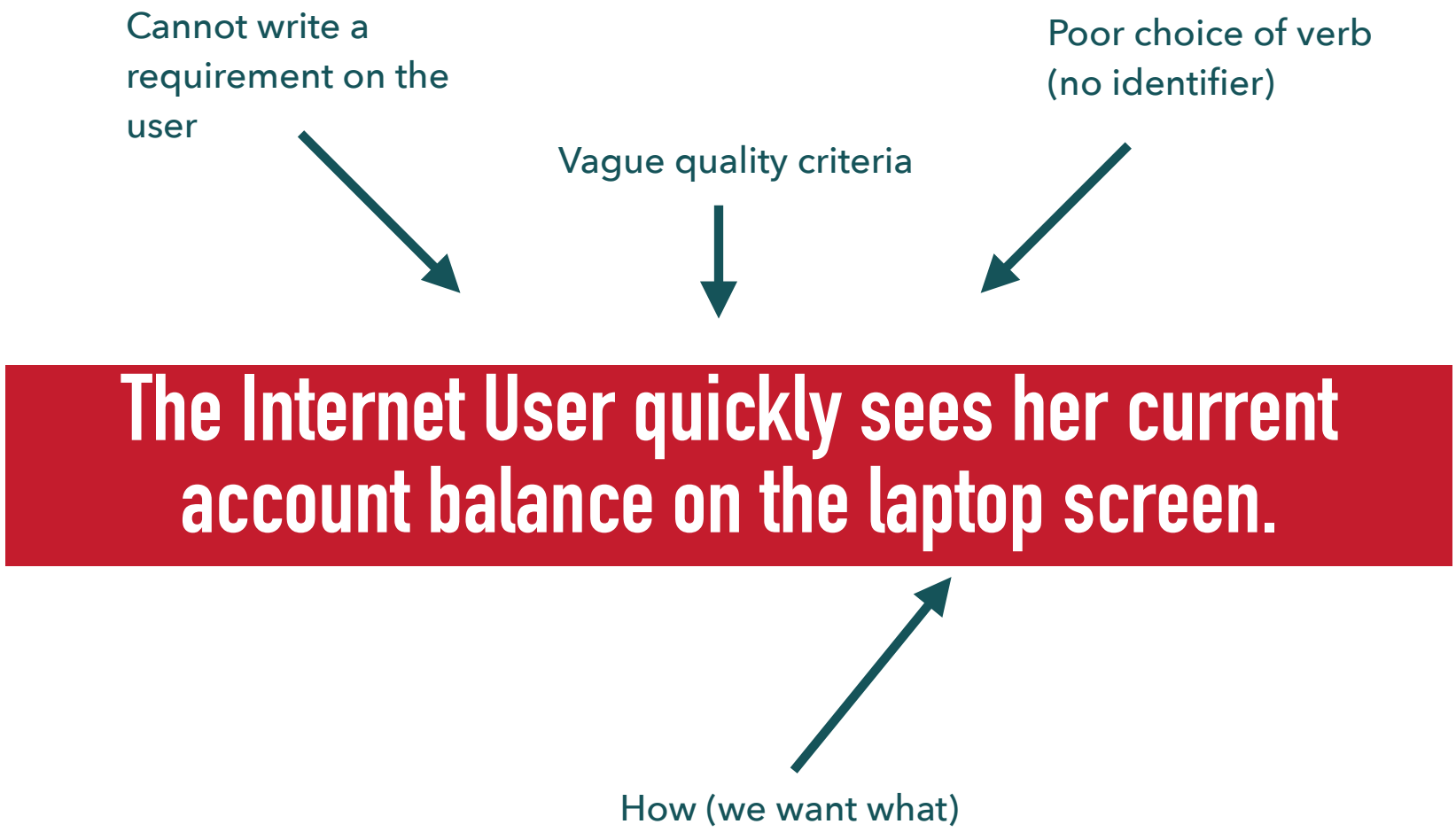
Anatomy of a Good User Requirement

- Identifies the system under discussion and a desired end result that is wanted within a specified time that is measurable
- The challenge is to seek out the system under discussion, end result, and success measure in every requirement
- Each requirements includes:
  - Subject
  - Action (shall or may)
  - Positive end result
  - Quality measure


Defines the subject
(system under discussion)
Shall (mandatory) or
may (optional)
The Online Banking System shall allow the Internet user to access her current account balance in less than 5 seconds.
Positive end result
Quality criteria

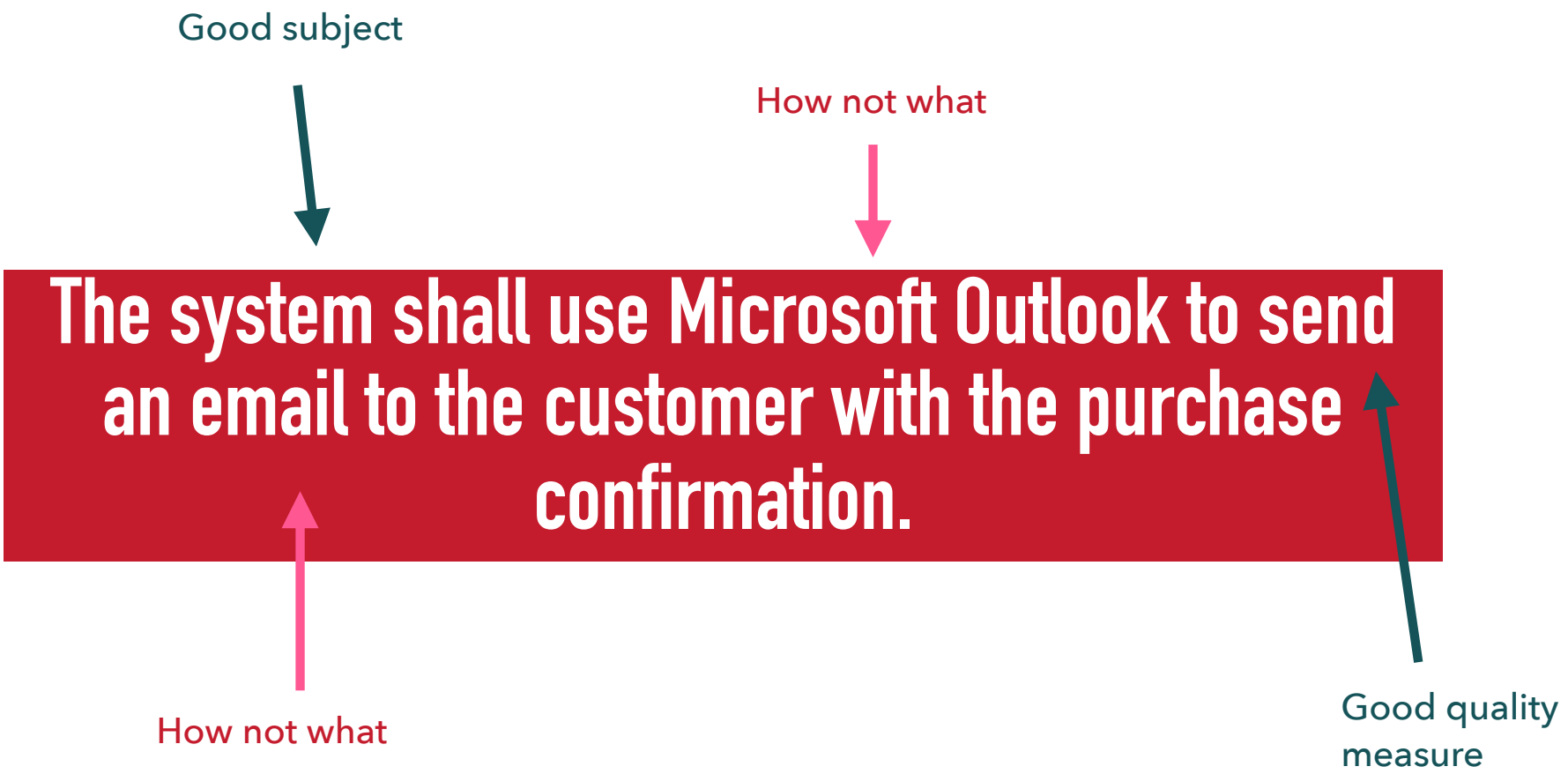
Never tell a child not to run, instead **ask them to walk**. Because jumping, skipping, frolicking, racing, jogging, and any number of other activities is also not running.

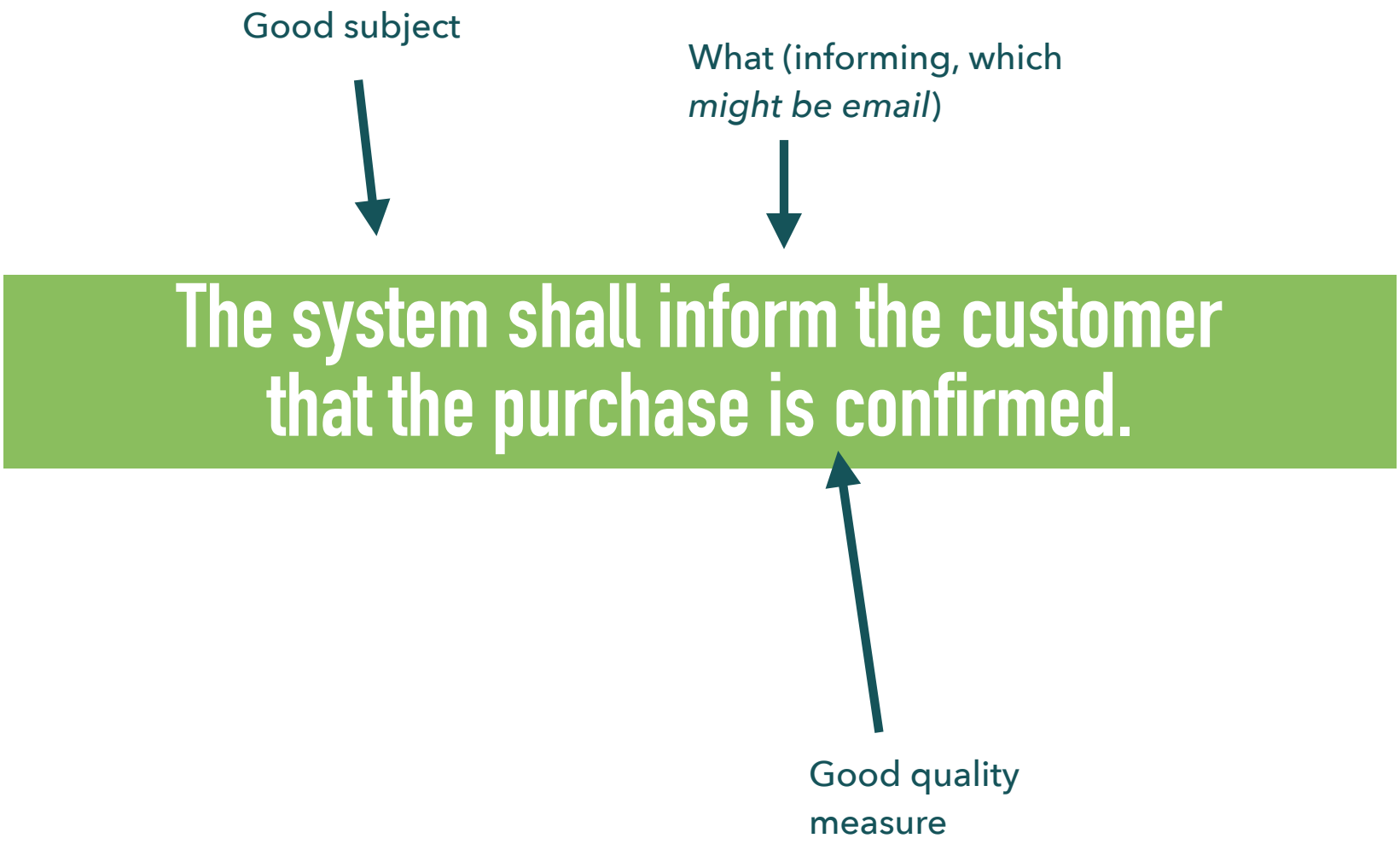
https://en.wikipedia.org/wiki/Requirement


Cannot write a requirement on the user
Vague quality criteria
Poor choice of verb (no identifier)
The Internet User quickly sees her current account balance on the laptop screen.
How (we want what)

Standard for Writing a Requirement

- Each requirement must form **a complete sentence**
- Each requirement contains:
  - **Subject**: a user type (watch out!) or the system under discussion
  - **Predicate**: a condition, action, or intended result
  - **Verb**: “shall” / “will” / “must” to show mandatory nature;
    “may” / “should” to show optionality
- The whole requirement provides the specifics of a **desired end goal** or result
- Contains a **success criterion** or other **measurable** indication of the **quality**


Good subject
How not what
The system shall use Microsoft Outlook to send an email to the customer with the purchase confirmation.
How not what
Good quality measure


Good subject
What (informing, which *might be email*)
The system shall inform the customer that the purchase is confirmed.
Good quality measure

WRITING REQUIREMENTS

WRITING PITFALLS TO AVOID

Do not speculate

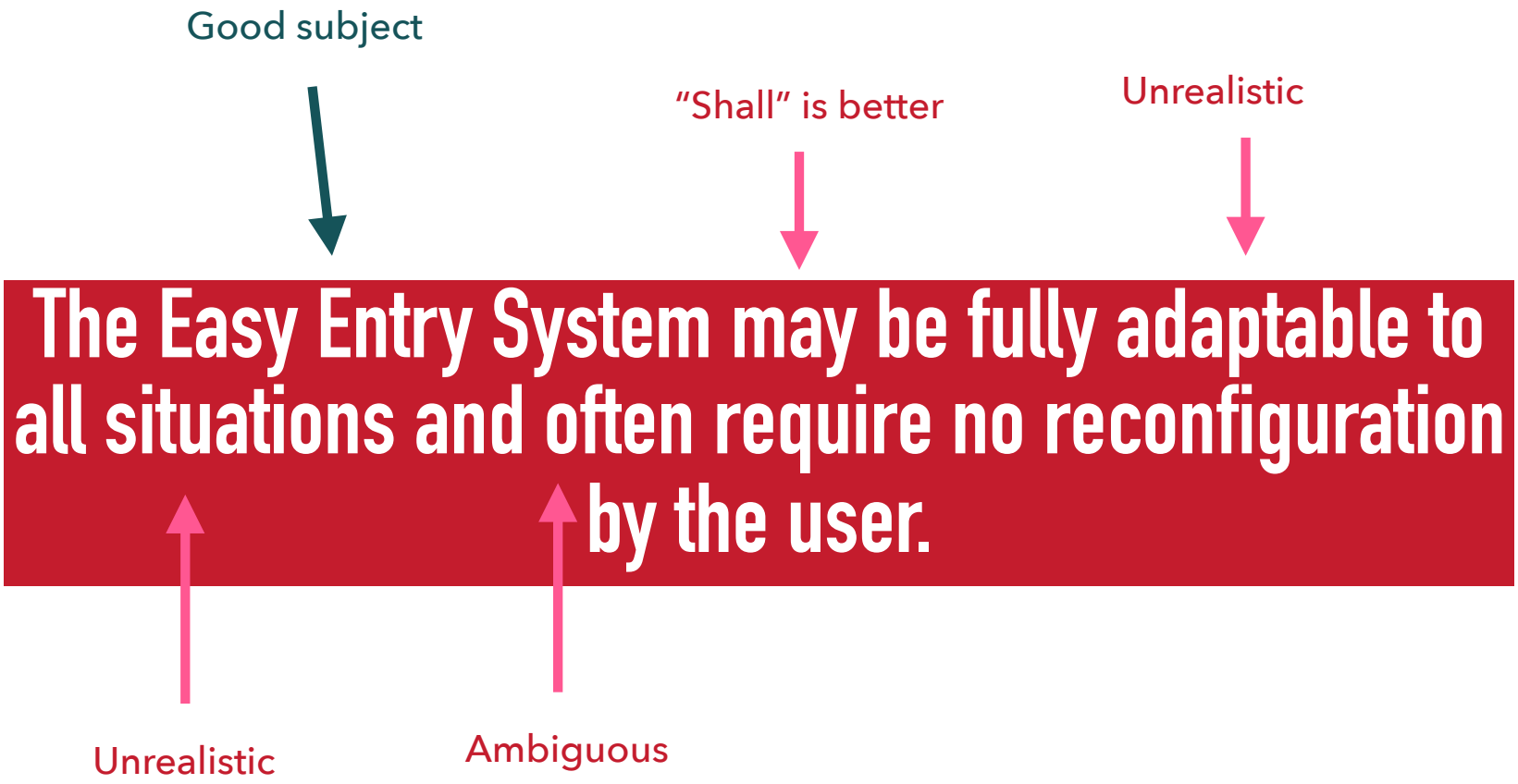
- There is no room for “wish lists” – general terms about things that somebody probably wants
- Danger signs: vague subject type and generalization words such as
  - usually,
  - normally,
  - generally,
  - typically
  - often,

Do not express suggestions or possibilities

- Suggestions that are not explicitly stated as requirements are invariably ignored by developers
- Danger signs:
  - may,
  - might,
  - should,
  - ought
  - could,
  - perhaps,
  - probably

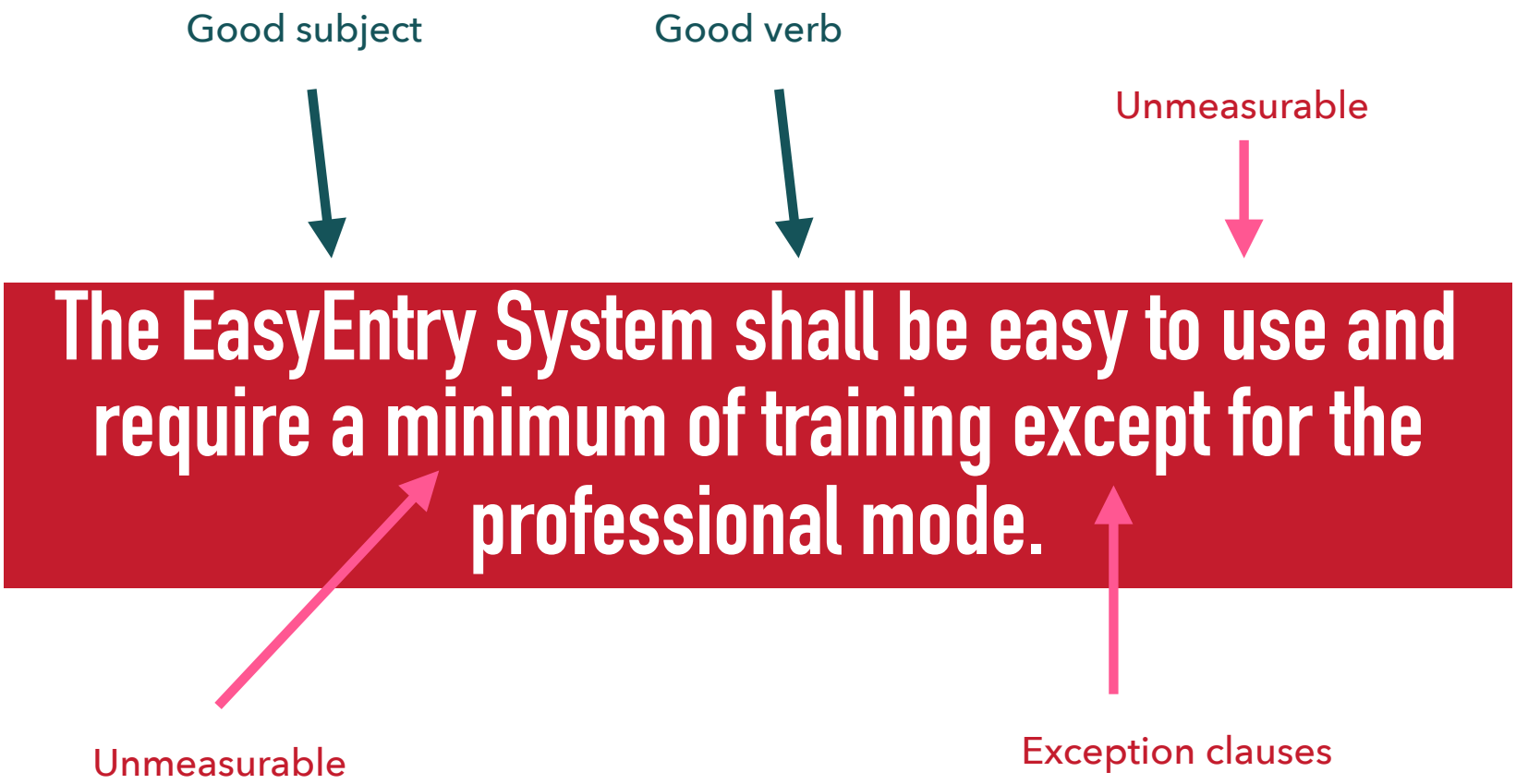
Avoid wishful thinking

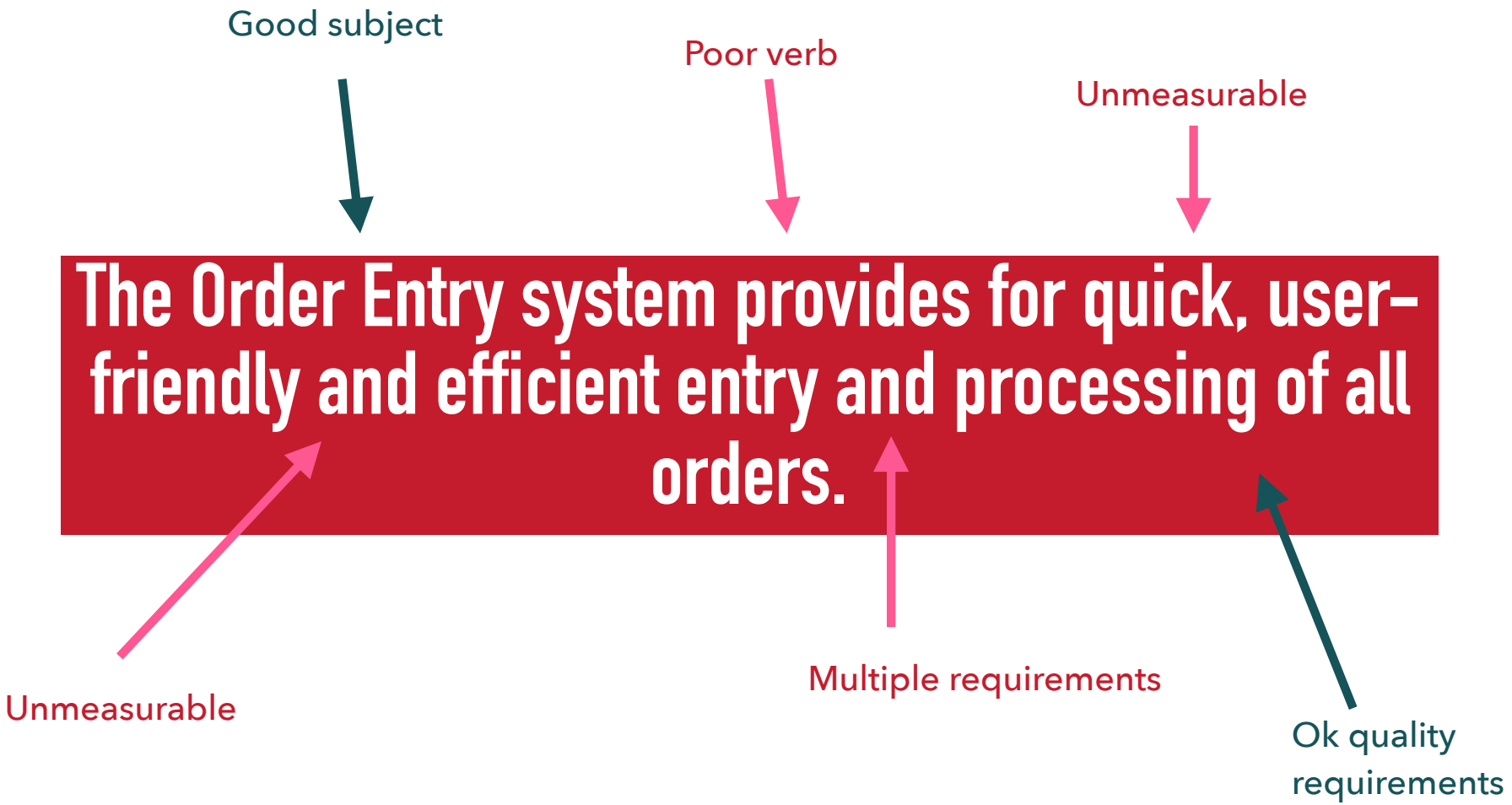
- Wishful thinking means asking for the impossible
  - 100% reliable,
  - safe,
  - handle all failures
  - fully upgradeable,
  - run on all platforms

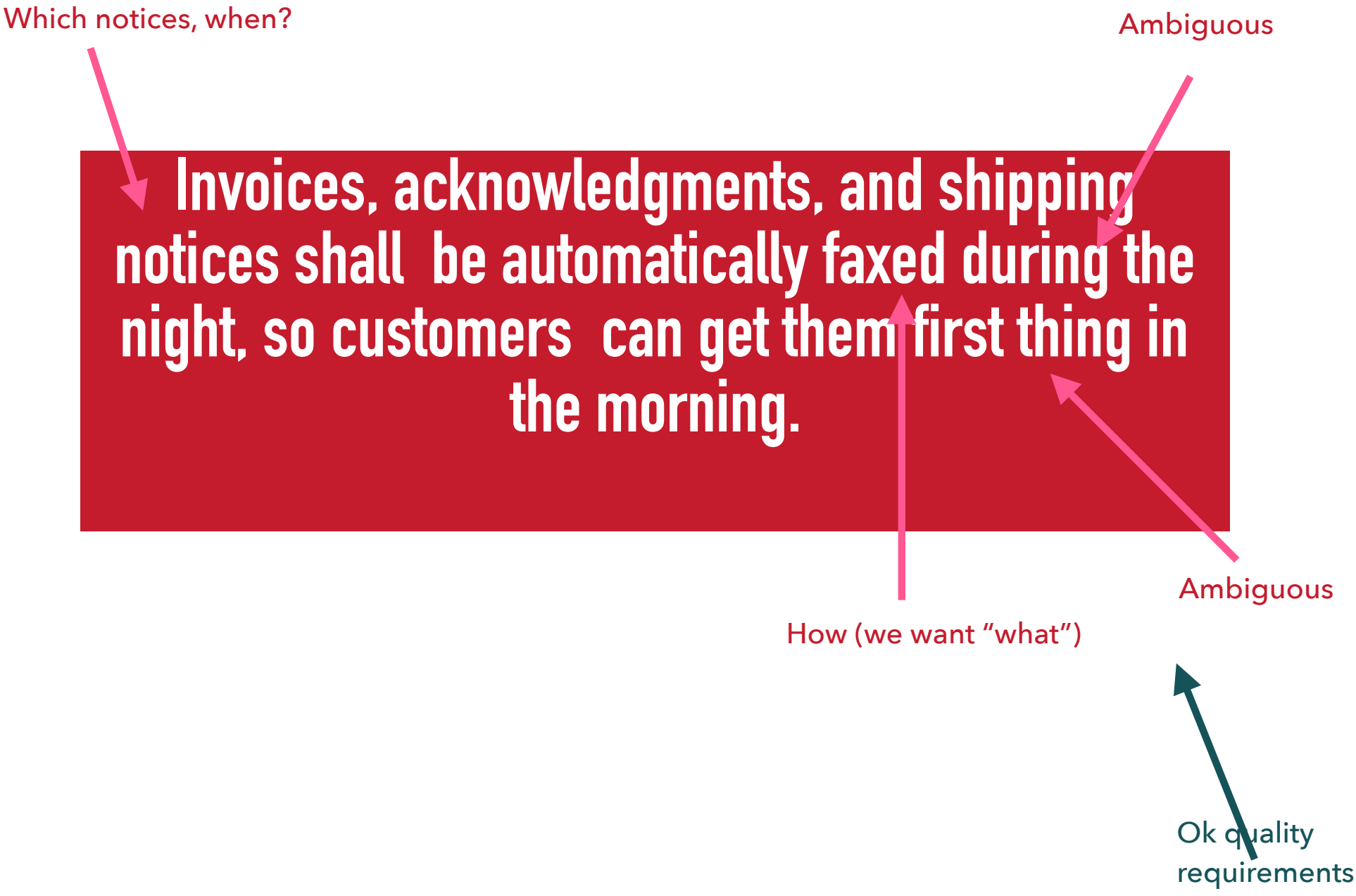

Good subject
"Shall" is better
Unrealistic
The Easy Entry System may be fully adaptable to all situations and often require no reconfiguration by the user.
Unrealistic
Ambiguous

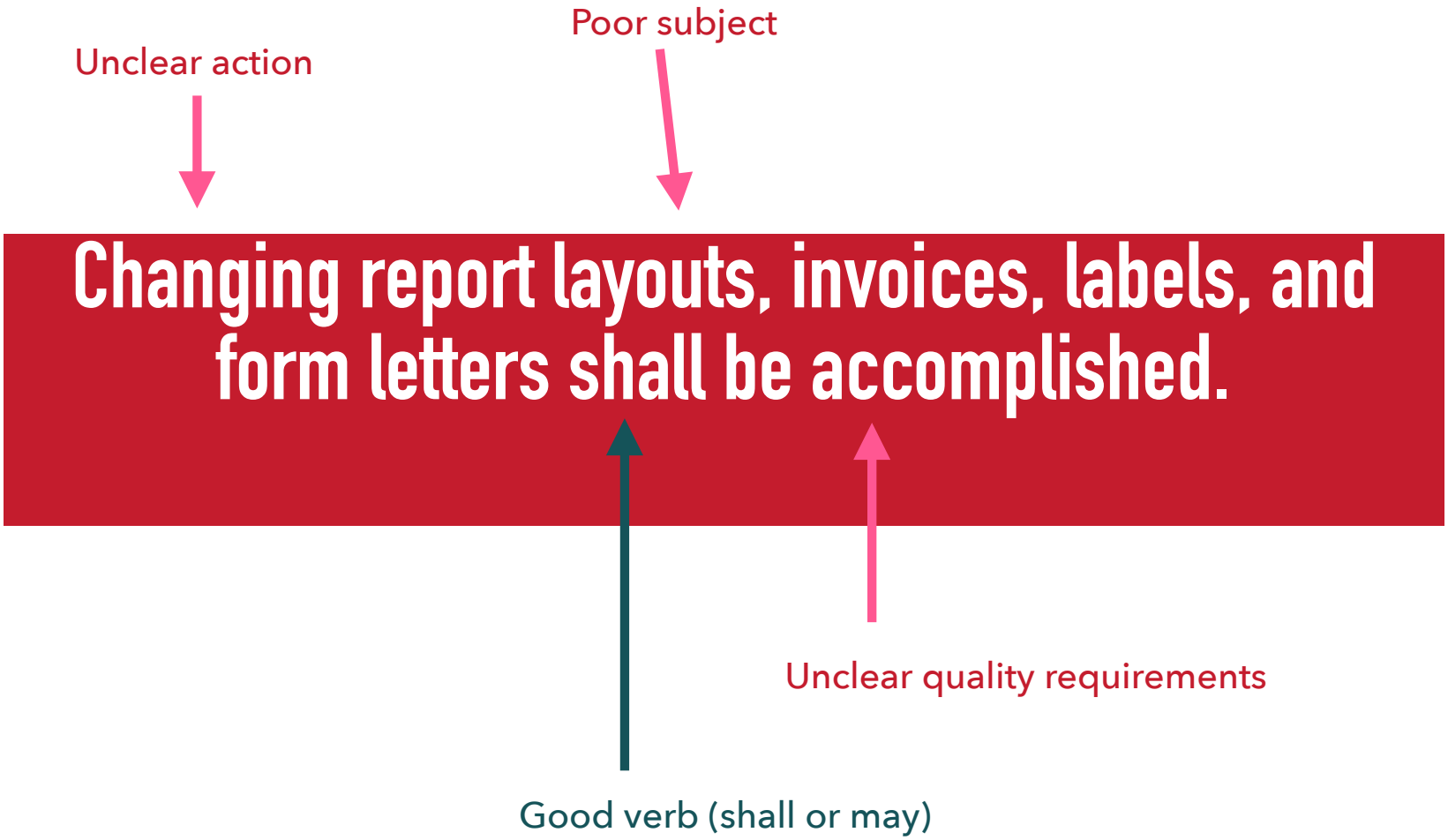
Do not use vague indefinable terms

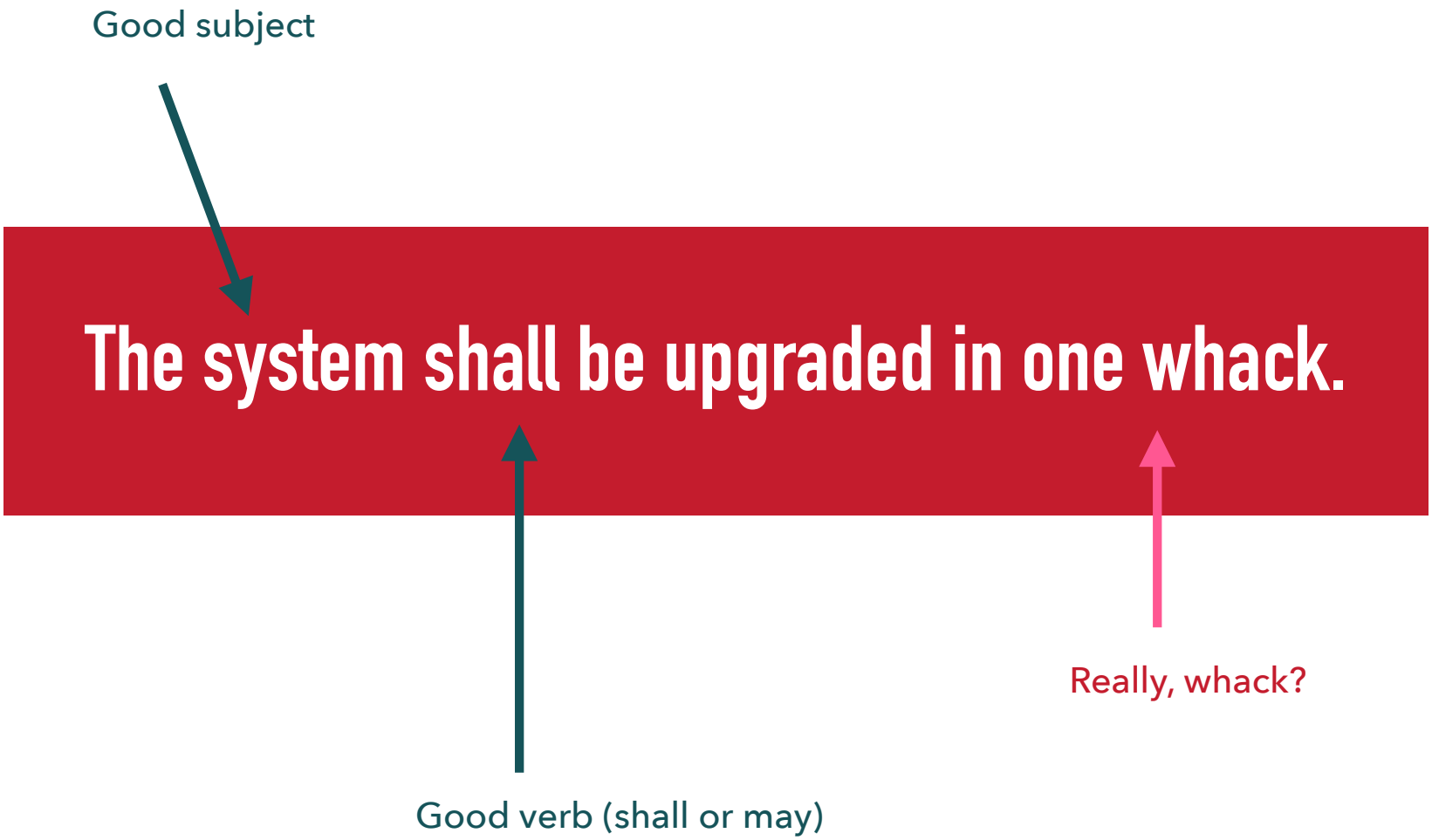
- Many words used informally to indicate quality are too vague to be verified
- Danger signs:
  - user-friendly,
  - highly versatile,
  - flexible,
  - to the maximum extent
  - approximately,
  - as much as possible,
  - minimal impact

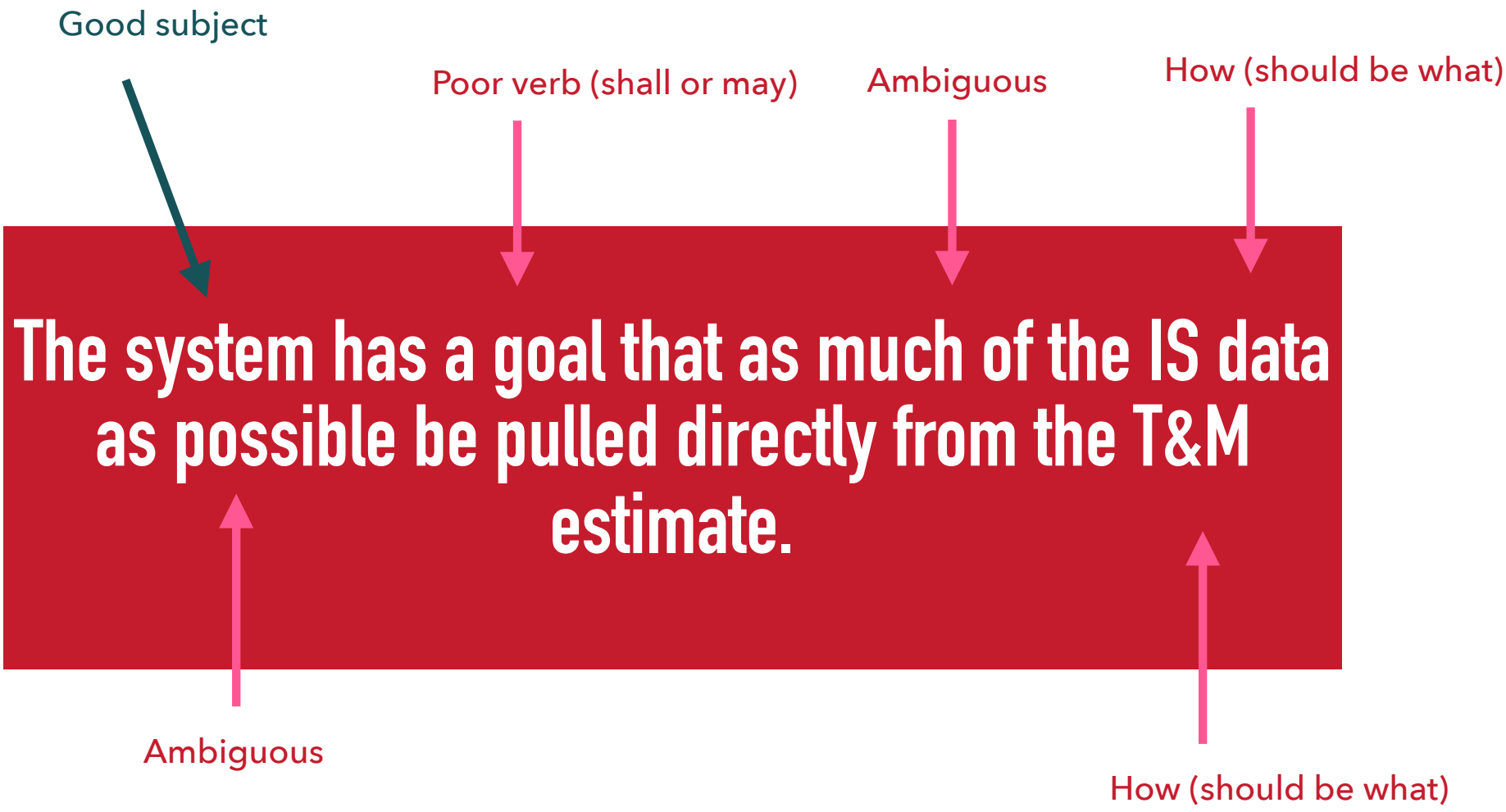

Good subject
Good verb
Unmeasurable
The EasyEntry System shall be easy to use and require a minimum of training except for the professional mode.
Unmeasurable
Exception clauses


Good subject
Poor verb
Unmeasurable
The Order Entry system provides for quick, user-friendly and efficient entry and processing of all orders.
Unmeasurable
Multiple requirements
Ok quality requirements


Which notices, when?
Ambiguous
Invoices, acknowledgments, and shipping notices shall be automatically faxed during the night, so customers can get them first thing in the morning.
How (we want "what")
Ambiguous
Ok quality requirements


Unclear action
Poor subject
Changing report layouts, invoices, labels, and form letters shall be accomplished.
Unclear quality requirements
Good verb (shall or may)


Good subject
The system shall be upgraded in one whack.
Really, whack?
Good verb (shall or may)


Good subject
Poor verb (shall or may)
Ambiguous
How (should be what)
The system has a goal that as much of the IS data as possible be pulled directly from the T&M estimate.
Ambiguous
How (should be what)

TYPES OF

REQUIREMENTS

Functional

Quality
(non functional)

Platform

Process

Functional Requirements

- Inputs
- Outputs
- Data (storage / other systems)
- Computations
- Timing / Synchronization

Quality Requirements

- Response time
- Throughput
- Resource usage
- Reliability
- Availability
- Recovery from failure
- Allowances for maintainability and enhancement
- Allowances for reusability

DESCRIBING REQUIREMENTS USING

USE CASES

A use case

- Sequence of actions to complete a task
- Model the system for
  - How users interact with the system
  - Achieving objectives
- Key activity in requirements analysis
- Use case model describes a set of use case and a diagram of their relationships

A USE CASE SHOULD . . .

- Cover a sequence of steps
- How the user interactions with the system
  - NOT, the computations of the system
  - Only user interactions with the system
  - Ignores manual user actions
- Independent from the design / UI

Template of a use case

A. **Name**: Short and descriptive.

B. **Actors**: Types of users that will do this

C. **Goals**: What are the actors trying to achieve.

D. **Preconditions**: State of the system before the use case.

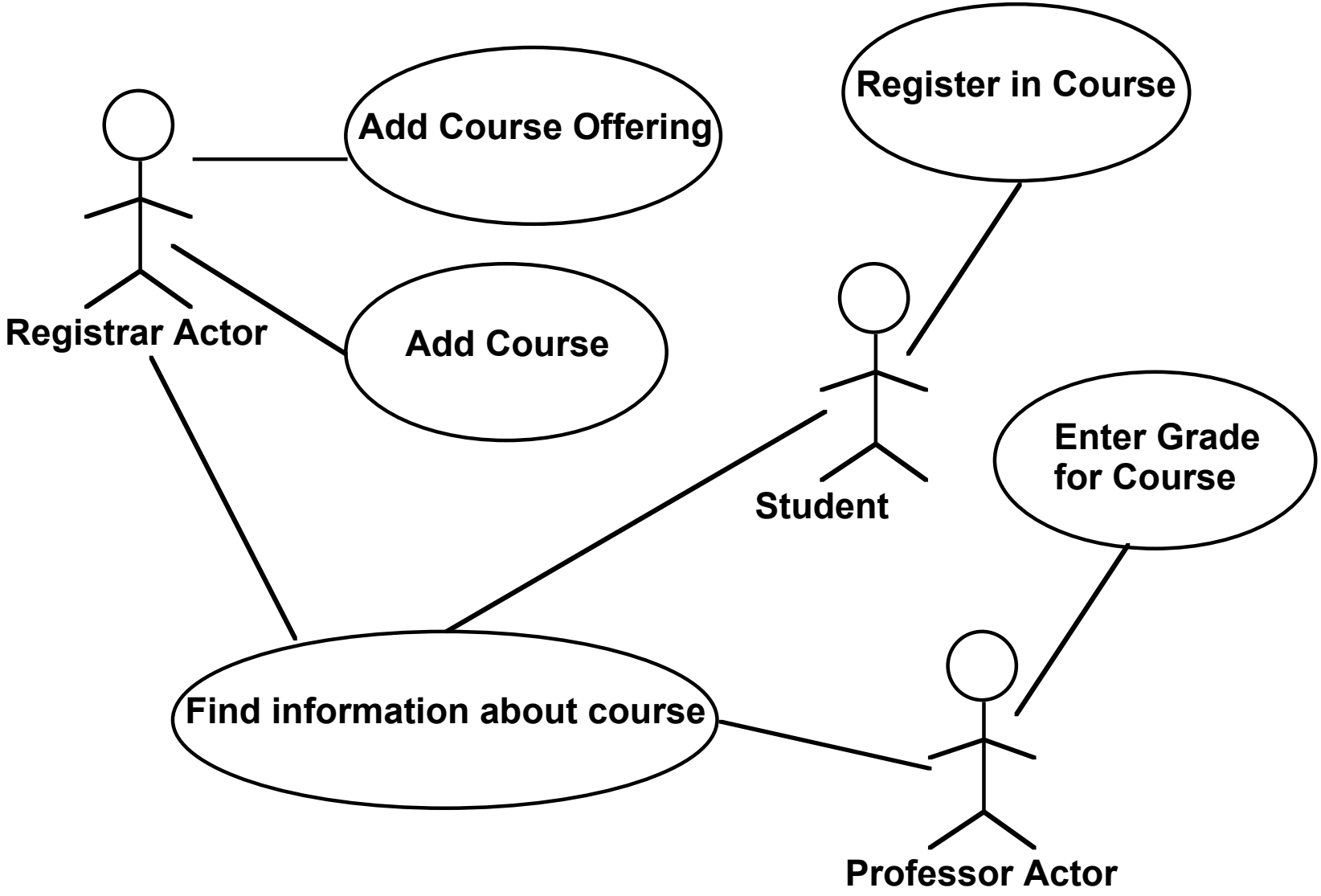
E. **Summary**: Short informal description.

F. **Related use cases**.

G. **Steps**: Describe each step using a 2-column format.

H. **Postconditions**: State of the system after completion.

USE CASE DIAGRAMS



Extensions

- Used to make optional interactions explicit or to handle exceptional cases.
- Keep the description of the basic use case simple.

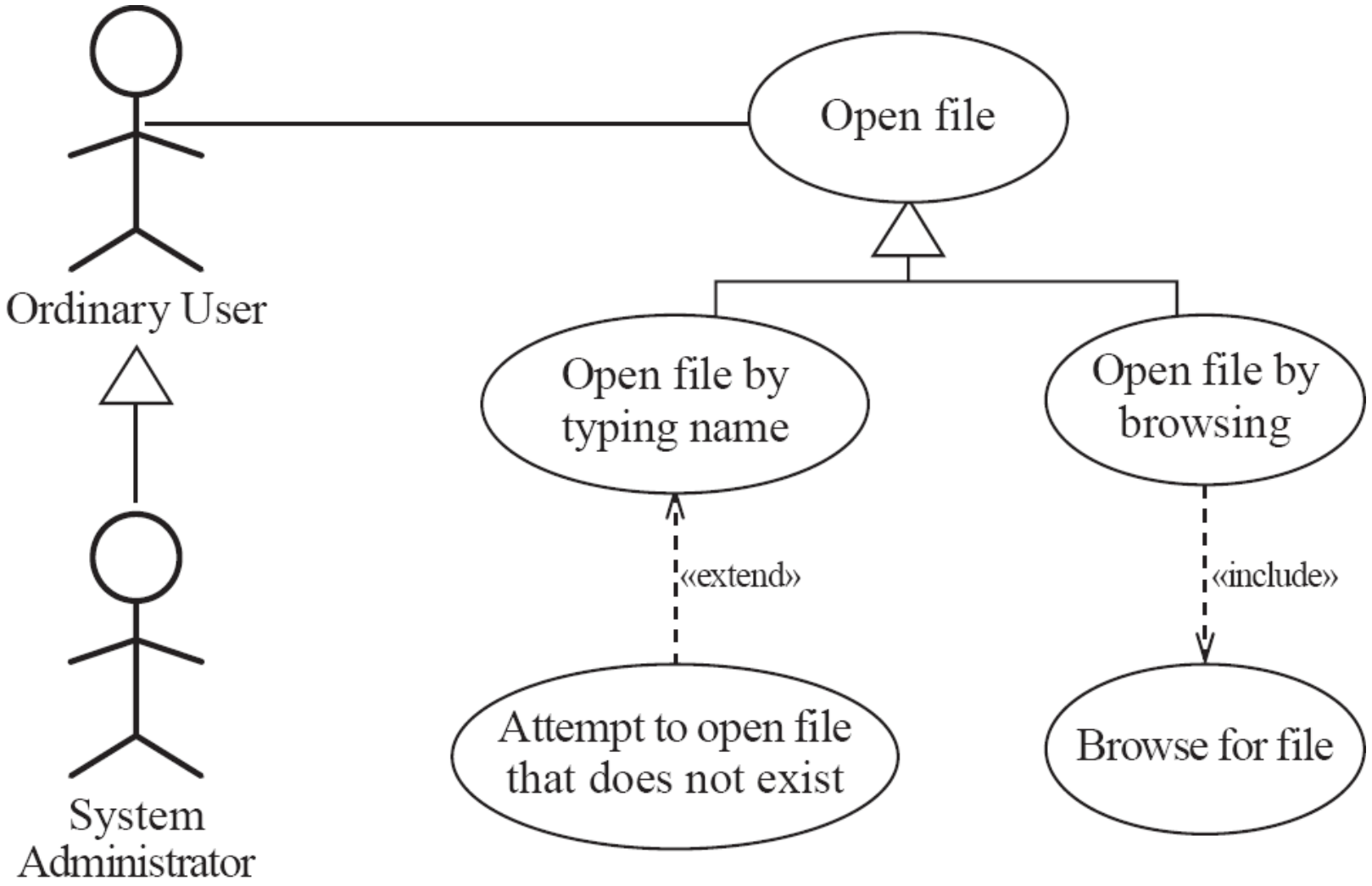
Generalizations

- Much like superclasses in a class diagram.
- A generalized use case represents several similar use cases.
- One or more specializations provides details of the similar use cases.

Inclusions

- Commonality between several different use cases.
- Are included in other use cases
  - Even very different use cases can share sequence of actions.
  - Enable you to avoid repeating details in multiple use cases.
- Lower-level task with a lower-level goal.

EXAMPLE OF GENERALIZATION, EXTENSION AND INCLUSION



Use case: **Open file**

Related use cases:

Generalization of:

- Open file by typing name
- Open file by browsing

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Use case: **Open file by typing name**

Related use cases:
Specialization of: Open file

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click ‘Open’	5. Dialog disappears

Use case: **Open file by browsing**

Related use cases:
Specialization of: Open file
Includes: Browse for file

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3. Browse for file	
4. Confirm selection	5. Dialog disappears

Use case: **Attempt to open file that does not exist**

Related use cases:
Extension of: Open file by typing name

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click ‘Open’	5. System indicates that file does not exist
6. Correct the file name	
7. Click ‘Open’	8 Dialog disappears

Use case: **Browse for file (inclusion)**

Steps:

Actor actions	System responses
1. If the desired file is not displayed, select a directory	2. Contents of directory is displayed
3. Repeat step 1 until the desired file is displayed	
4. Select a file	

Scenario

- A scenario is an **instance of a use case** that expresses a **specific occurrence** of the use case with a **specific actor** operating at a specific time and using **specific data**.

Use case: Exit car park, paying cash

Actors: Car drivers

Goals: To leave the parking lot after having paid the amount due.

Preconditions: The driver must have entered the car park with his or her car, and must have picked up a ticket upon entry.

Summary: When a driver wishes to exit the car park, he or she must bring his or her car to the exit barrier and interact with a machine to pay the amount due.

Related use case: Exit car park by paying using a debit card

Steps:

Actor actions	*System responses*
1. Drive to exit barrier, triggering a sensor.	2a. Detect presence of a car. 2b. Prompt driver to insert his or her card.
3. Insert ticket.	4. Display amount due.
5. Insert money into slot.	6a. Return any change owing. 6b Prompt driver to take the change (if any). 6c. Raise barrier.
7. Drive through barrier, triggering a sensor.	8. Lower barrier.

Scenario

Steps:

Actor actions	*System responses*
Drives to the exit barrier.	Detects the presence of a car.
	Displays: 'Please insert your ticket'.
Inserts ticket.	Displays: 'Amount due $2.50'.
Inserts $1 into the slot.	Displays: 'Amount due $1.50'.
Inserts $1 into the slot.	Displays: 'Amount due $0.50'.
Inserts $1 into the slot.	Returns $0.50.
	Displays: 'Please take your $0. 50 change'.
	Raises barrier.
Drives through barrier, triggering sensor.	Lowers barrier.

Suppose an online banking system with actors: *Customer*, *Credit Agent*, and *User*, together with a mis-actor: *Thief*. The following use cases have been identified:

i) ***pay bills*** initiated by actor ***Customer*** to pay bills online,

ii) ***transfer funds*** initiated by actor ***Customer*** to transfer funds from an account to another account within the Bank,

iii) ***transfer funds externally*** initiated by actor ***Customer*** to transfer funds from an account in the Bank to an account at another Bank,

iv) ***get a loan*** initiated by actor ***Customer*** to obtain a loan,

v) ***log in*** initiated by all ***Users*** (***Customer***, ***Credit Agent***) to identify themselves to the system,

vi) ***check browser patches*** to check that web browsers are protected against phishing attacks and to warn users if this is not the case,

vii) ***steal login information***, a misuse case, initiated by mis-actor ***Thief*** to obtain a customer's information by using a phishing attack.

Additionally:

- Actor ***Credit Agent*** participates in use case ***get a loan***.
- Use case ***log in*** is performed as part of use cases ***pay bills***, ***transfer funds*** and ***get a loan***.
- Use case ***transfer funds externally*** is a variation of use case ***transfer funds*** with additional interactions.
- Misuse case ***steal login information*** hurts the objective of use case ***log in***.
- Use case ***check browser patches*** is performed every time a user logs in to counter misuse case ***steal login information***.

Choosing use cases on which to focus

- Central to the system
- Built around this particular use case
- Other reasons to focusing on particular use cases:
  - Some represent a high risk
  - Some have high political or commercial value

The benefits of basing software development on use cases

- Define the scope
- Plan the development
- Develop and validate requirements
- Definition of test cases
- Structure user manuals

User stories

- Agile development
  - Similar but not
  - Typically less formal
- Focus on a conversation with the client

GATHERING AND ANALYZING

REQUIREMENTS

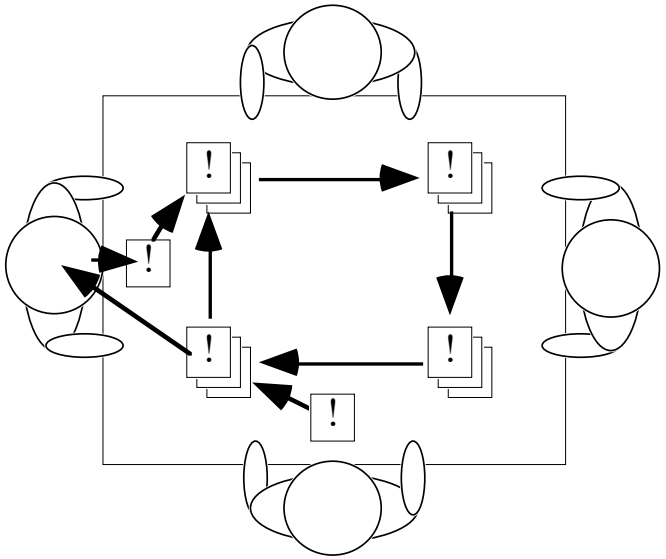
Observation

- Read documents and discuss requirements with users
- Shadowing important potential users as they do their work
  - ask the user to explain everything he or she is doing
- Session video taping

Interviewing

- Ask about specific details
- Ask about the stakeholder's vision for the future
- Ask if they have alternative ideas
- Ask for other sources of information
- Ask them to draw diagrams

Brainstorming



- Appoint an experienced moderator
- Arrange the attendees around a table
- Decide on a ‘trigger question’
- Ask each participant to write an answer and pass the paper to its neighbour

JOINT APPLICATION DEVELOPMENT (JAD)

https://www.youtube.com/watch?v=HizIOlMphhs

https://en.wikipedia.org/wiki/Joint_application_design

Prototyping

- Paper prototype.
  - Pictures of the system shown in sequence
- Mock-up of the system's UI
  - Written rapidly
  - Little computations, no database, no interactions
- May be limited to particular aspects

Use case analysis

- Determins the classes of users (actors)
- Determine actor tasks

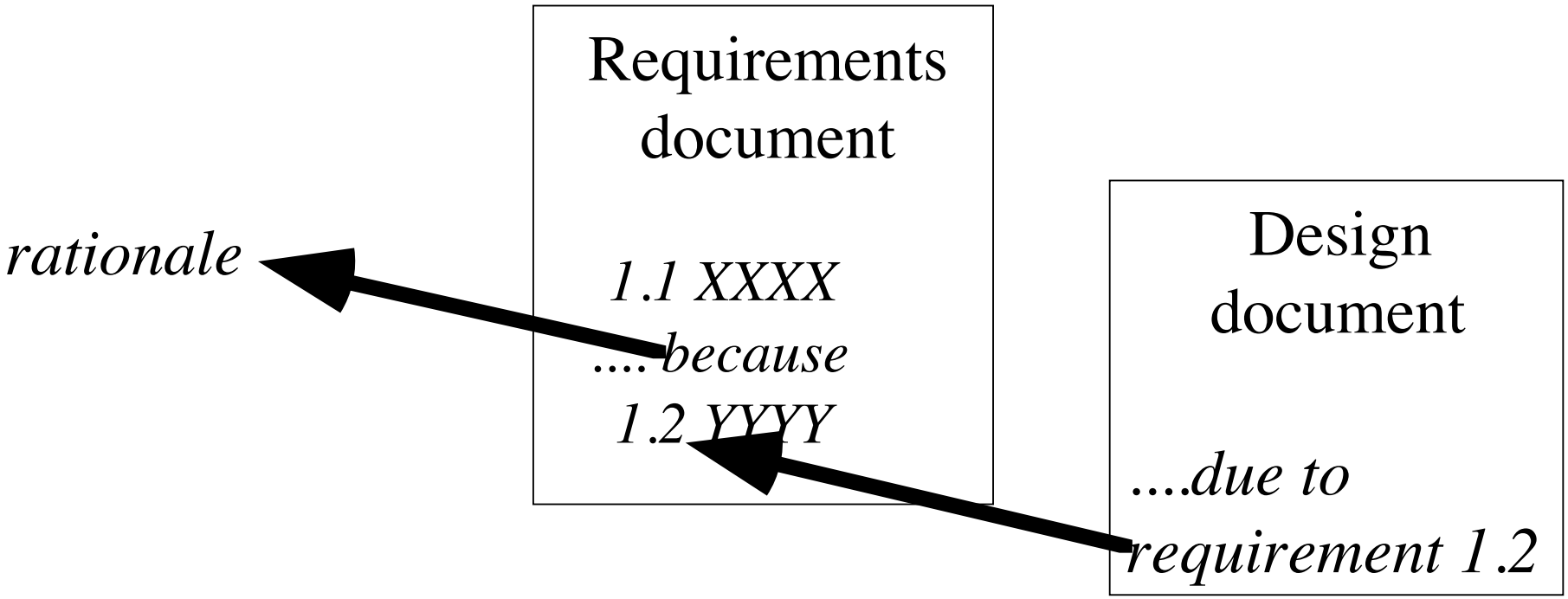
More detailed requirements are likely needed when

- Large and complex system
- Interface to hardware or other systems
- Very technical domain (engineering, science, finances)
- Large costs, safety or security issues for failures

Reviewing Requirements

Feasible	Clear	Identifiable	Non-Ambiguous
Needed	Concise	Modifiable	Not over contrained
Testable	Coherent	Prioritized	Important
	Independent of Design	Traceable	Quality

TRACEABILITY



Requirements change because:

- Business process changes
- Technology changes
- The problem becomes better understood

Requirements analysis never stops

- Continued interaction with clients / users
- Benefits of change outweighs costs
  - Quick and easy at little cost
  - Larger changes, not so much
    - Forcing unexpected changes result in poor design decisions
- Some changes are enhancements in disguise
  - Avoid making the system bigger, only make it better