21st Century Requirements Engineering: A Pragmatic Guide to Best Practices

Erik Simmons, Intel Corporation

Requirements engineering is a core discipline to product development, whether an organization is large or small; involved in market-driven products, IT development, or contractual work; or using traditional or agile methods. There is no shortage of books, papers and courses on requirements, but what really works, and where to start?

In this session, we'll examine some of the core questions that govern how much detail is enough, which areas need it, and when to provide it – regardless of what software life cycle you are using. In addition, we will cover some of the practices that have proven most useful across projects of all types.

So, if you are confused about "agile requirements", can't find the right balance of detail level vs. cost and deadlines in your requirements work, or just want to see some broadly useful practices that you can start using immediately, stop by for the discussion.

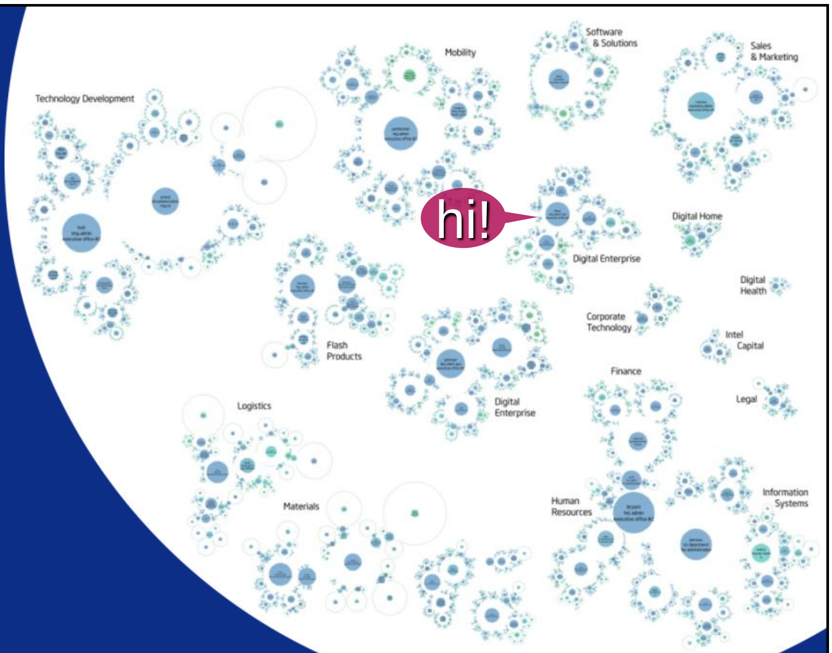
Erik Simmons works in the Corporate Platform Office at Intel Corporation, where he is responsible for creation, implementation, and evolution of requirements engineering practices and supports other corporate platform and product initiatives. Erik's professional interests include software development, decision making, heuristics, development life cycles, systems engineering, risk, and requirements engineering. He has made invited conference appearances in New Zealand, Australia, England, Belgium, France, Germany, Switzerland, Finland, Canada, and the US. Erik holds a Masters degree in mathematical modeling and a Bachelors degree in applied mathematics from Humboldt State University, and was appointed to the Clinical Faculty of Oregon Health Sciences University in 1991.

21st-Century Requirements Engineering: A Pragmatic Guide to Best Practices

Erik Simmons

PNSQC 2011

Version 1.0, 08/11



2

Contents

Introduction

- Fundamental Concepts
- The Setting and the Challenges

Detail Level and Timing Issues

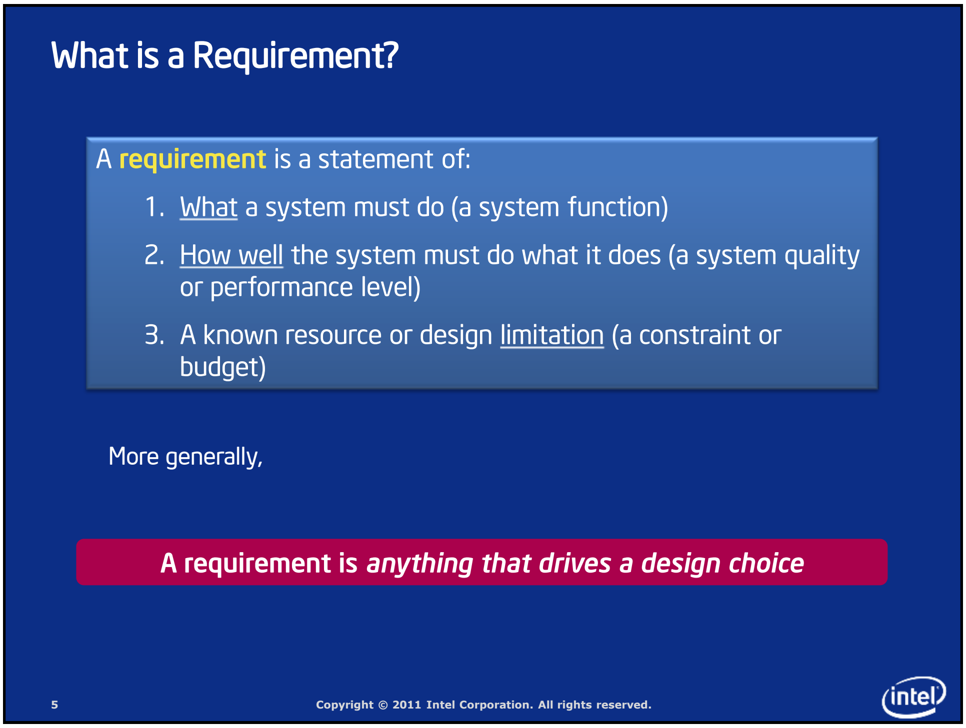
Requirements Practices for Today's Environment

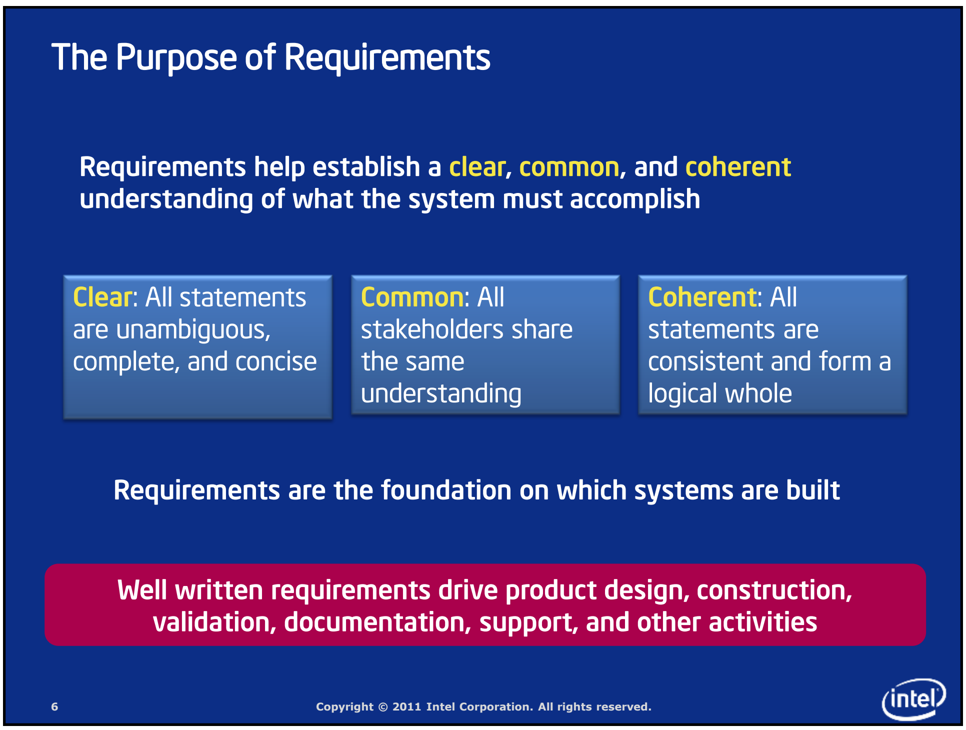
- The Three-Circle Model
- Specification Basics
- The Easy Approach to Requirements Syntax (EARS)
- Planguage
- The Landing Zone
- Specification Quality Control (SQC)

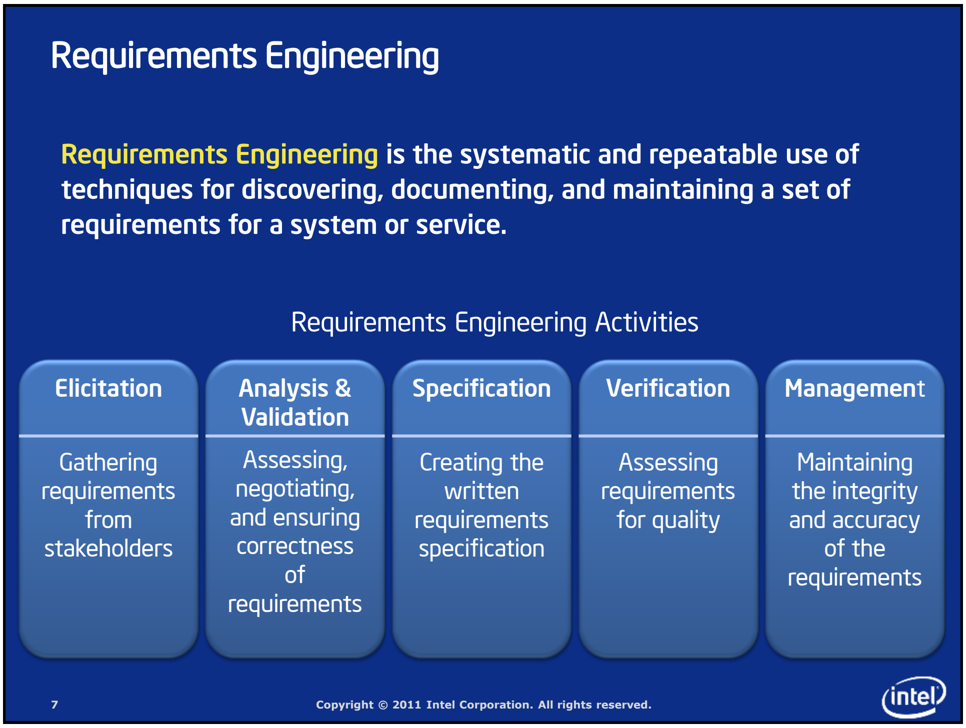
Sources for More Information

Note: Third-party brands and names are the property of their respective owners

Introduction


What is a Requirement?
A requirement is a statement of:
1. What a system must do (a system function)
2. How well the system must do what it does (a system quality or performance level)
3. A known resource or design limitation (a constraint or budget)
More generally,
A requirement is anything that drives a design choice
5
Copyright © 2011 Intel Corporation. All rights reserved.


The Purpose of Requirements
Requirements help establish a clear, common, and coherent understanding of what the system must accomplish
Clear: All statements are unambiguous, complete, and concise
Common: All stakeholders share the same understanding
Coherent: All statements are consistent and form a logical whole
Requirements are the foundation on which systems are built
Well written requirements drive product design, construction, validation, documentation, support, and other activities
6
Copyright © 2011 Intel Corporation. All rights reserved.


Requirements Engineering
Requirements Engineering is the systematic and repeatable use of techniques for discovering, documenting, and maintaining a set of requirements for a system or service.
Requirements Engineering Activities
Elicitation
Gathering requirements from stakeholders
Analysis & Validation
Assessing, negotiating, and ensuring correctness of requirements
Specification
Creating the written requirements specification
Verification
Assessing requirements for quality
Management
Maintaining the integrity and accuracy of the requirements
7
Copyright © 2011 Intel Corporation. All rights reserved.


Current Challenges | Complexity and Pace
Problem Complexity
We've solved most of the simple problems
Solution Complexity
We're not finding many simple solutions to complex problems
Design Tool Complexity
Multi-core, multi-threaded, distributed, cross-platform…yikes
Organizational Complexity
Larger, distributed software teams, more cross-domain interactions and dependencies
Software Development Process Complexity
This is a natural response to increasing solution complexity
Market Forces
The expectations placed on teams have not relaxed, even in the face of the other factors
8
Copyright © 2011 Intel Corporation. All rights reserved.

Current Challenges | Choice and Change

Today's markets are more fluid than ever, and consumers are more willing than ever to shift their thinking, spending, and brand loyalties

Companies must now innovate continuously, or risk loss of customer base to a more innovative rival

- Traditional barriers between product types are falling and new markets are emerging
- Usage models are evolving
- Shorter cycle times mean more threats to market-leading products

For example, think about how many ways music and video can be consumed today

Focus on customer delight and the rapid delivery of value to end users

9

The Need for Agility

Does Requirements Engineering Matter in an Agile World?

Yes! Complexity and pace mean we have define problem and solution, avoid rework, and maximize reuse

But this can't be "your grandfather's requirements engineering" – 21st-century requirements engineering must be different:

Less	More
Front-loaded, static, stand-alone	Incremental, fluid, integrated
Dictatorial	Collaborative, supportive
Exhaustive, speculative	Just-enough, just-in-time

The question is: How much requirements engineering, and when?

10

The Need for Abstraction & Hierarchy

Complexity requires better ways to address various views and subsets of a problem or solution

Aspect-oriented development and cross-cutting concerns are good examples

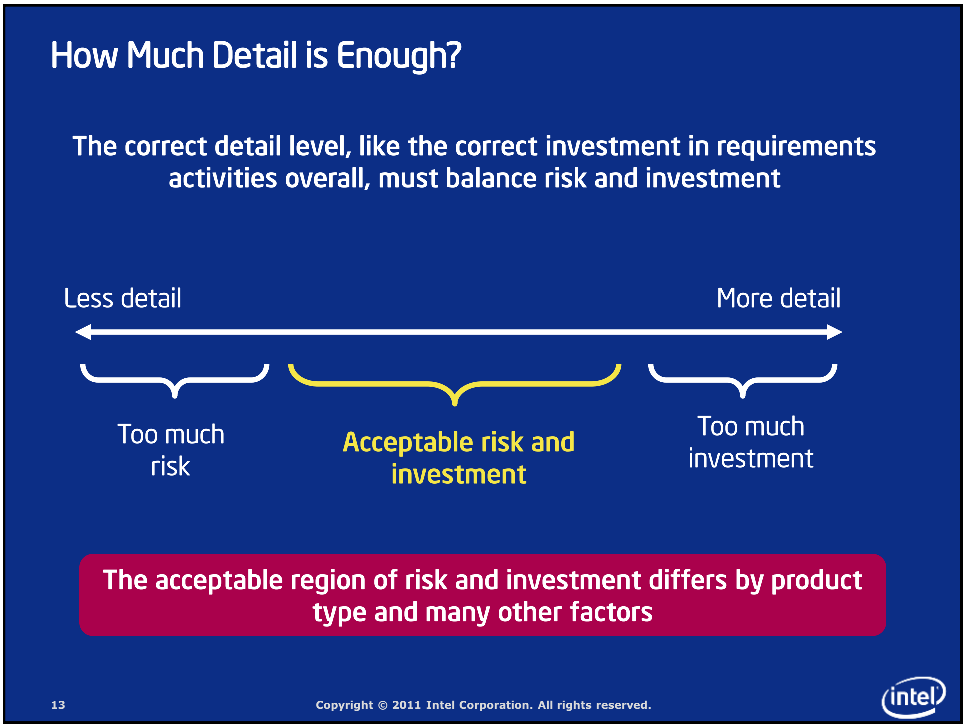
The biggest value in today's systems comes from emergent behaviors, and is not found in any single component

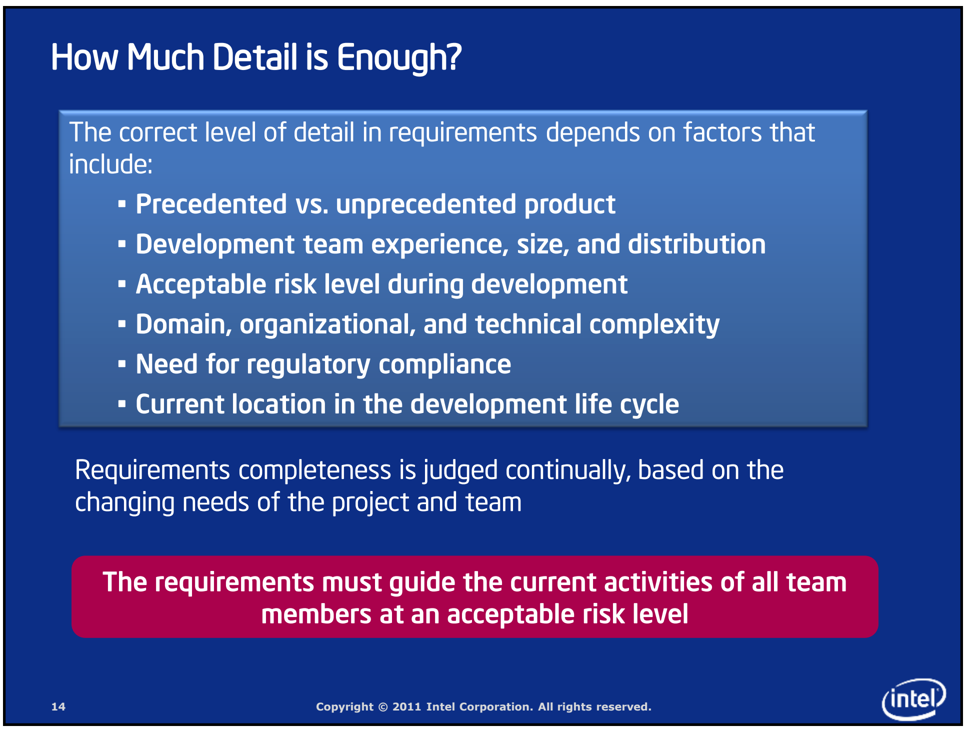
Requirements engineering, done correctly in partnership with architecture and design, can provide helpful abstraction and hierarchy

- What is it that is most valuable in your systems?
- Is that value found in a single component?
- Is it delivered by a single team?

11

Detail Level and Timing Issues


How Much Detail is Enough?
The correct detail level, like the correct investment in requirements activities overall, must balance risk and investment
Less detail
More detail
Too much risk
Acceptable risk and investment
Too much investment
The acceptable region of risk and investment differs by product type and many other factors
13
Copyright © 2011 Intel Corporation. All rights reserved.


How Much Detail is Enough?
The correct level of detail in requirements depends on factors that include:
▪ Precedented vs. unprecedented product
▪ Development team experience, size, and distribution
▪ Acceptable risk level during development
▪ Domain, organizational, and technical complexity
▪ Need for regulatory compliance
▪ Current location in the development life cycle
Requirements completeness is judged continually, based on the changing needs of the project and team
The requirements must guide the current activities of all team members at an acceptable risk level
14
Copyright © 2011 Intel Corporation. All rights reserved.

How Much Detail is Enough?

No requirements specification is ever truly complete

There isn't enough time or resources available to write them all – *and you shouldn't have to anyway…*

Provide detail where it's needed most: risky, unprecedented, or complex features and usages

Writing hundreds of pages of documentation may feel like productivity, but:

- If what gets documented is what everyone already understood, what is the effect on project risk?
- Large specifications can lead to a false sense of security

Make a conscious decision on what *NOT* to write

15

BRUF Versus Agile

Big Requirements Up Front (BRUF) involves asking stakeholders for "all their requirements", then "freezing" the requirements before design and development begins

BRUF forces stakeholders to defensively protect their interests by stating *every possible requirement they can think of*, even if it is unlikely they will ever need some of them

It is unreasonable to expect people to foresee all the contingencies and challenges up front

"Any attempt to formulate all possible requirements at the start of a project will fail and would cause considerable delays." Pahl and Beitz, *Engineering Design: A Systematic Approach*

Make a conscious decision on *WHEN* to write what you do write

16

A Flexible Approach to Scope and Details

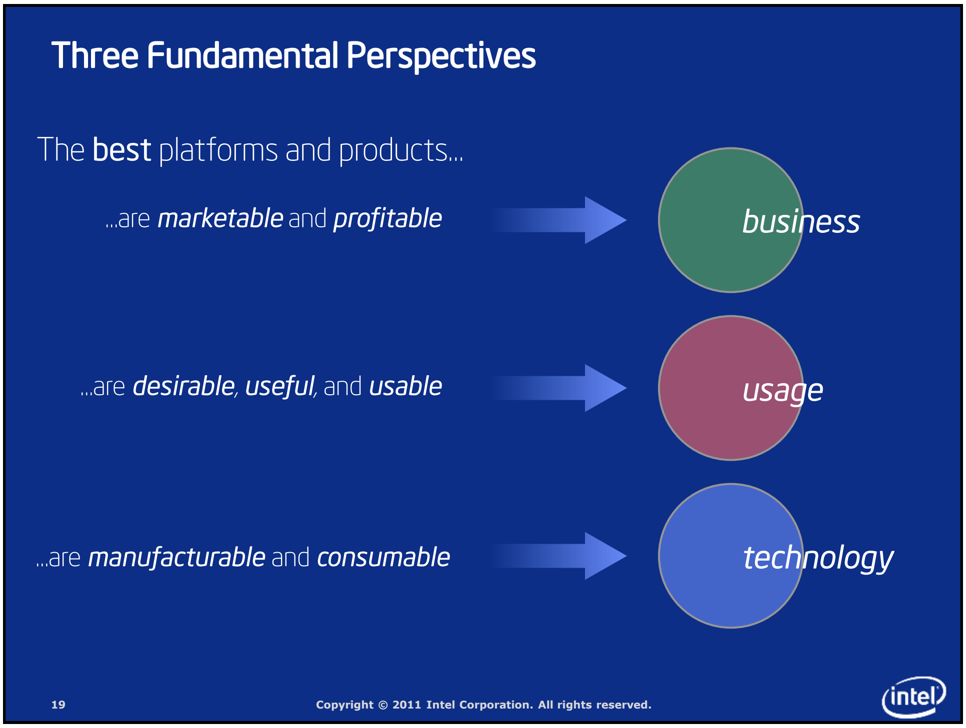
Regardless of what type of system you are building, use an *evolutionary* approach to requirements engineering:

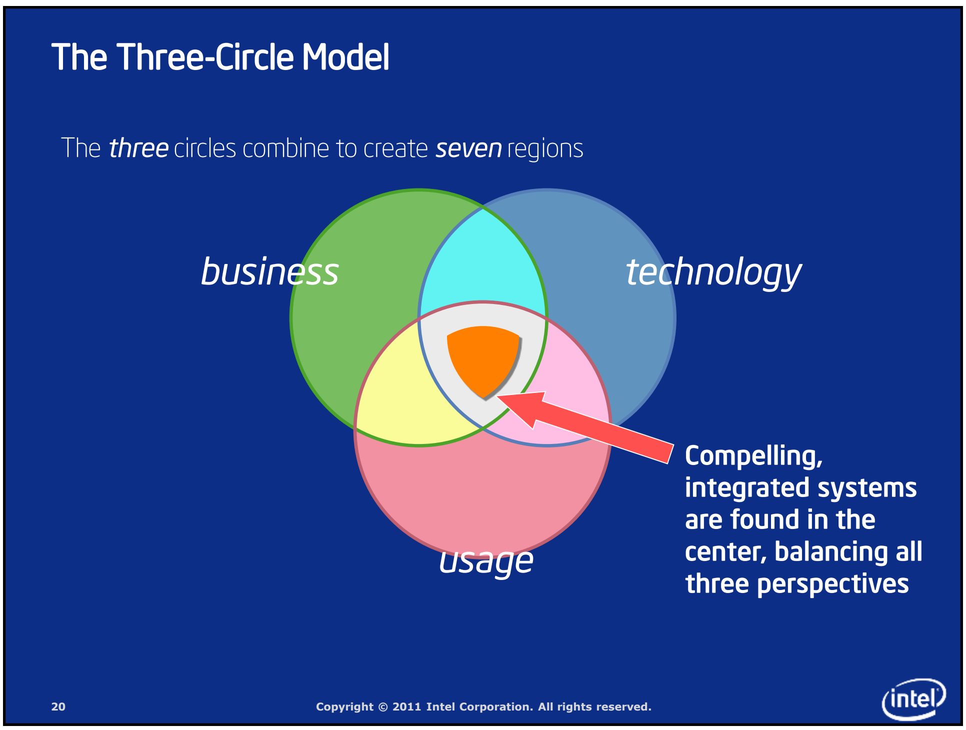
1. Start by generating requirements that define the scope of the system – full breadth, but minimum depth
2. Decide ***what not to write***
3. Decide ***when to write*** what you will write
4. Create the necessary details at the right time, always using business value and risk reduction as guides
5. Revisit steps 2 and 3 often based on what you learn as you make progress and the requirements evolve

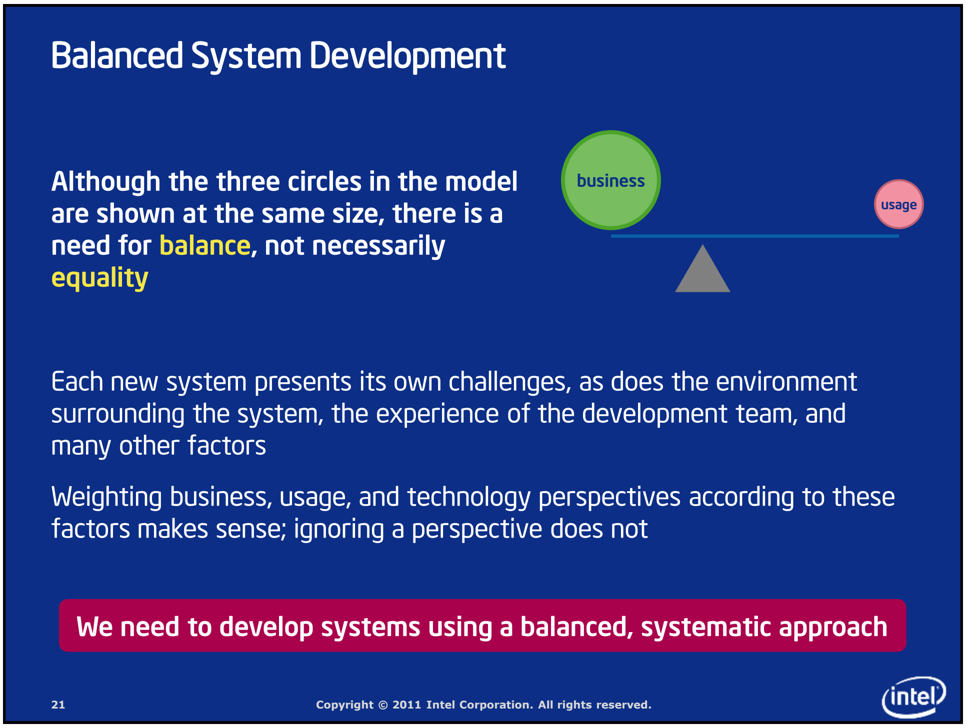
Iterative and incremental work in an agile environment

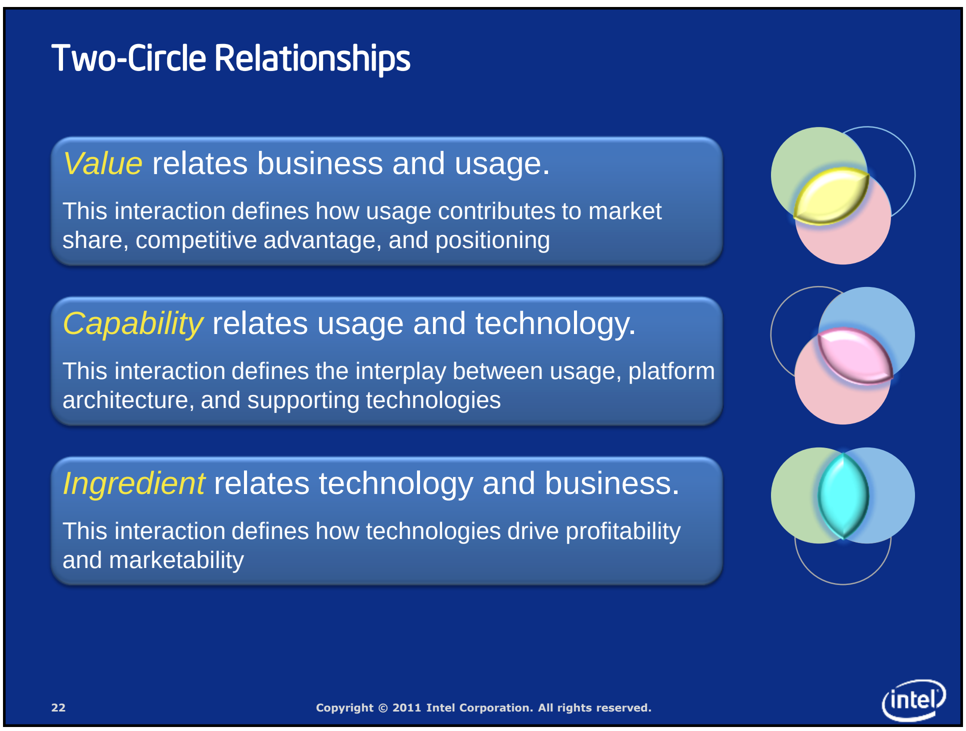
17

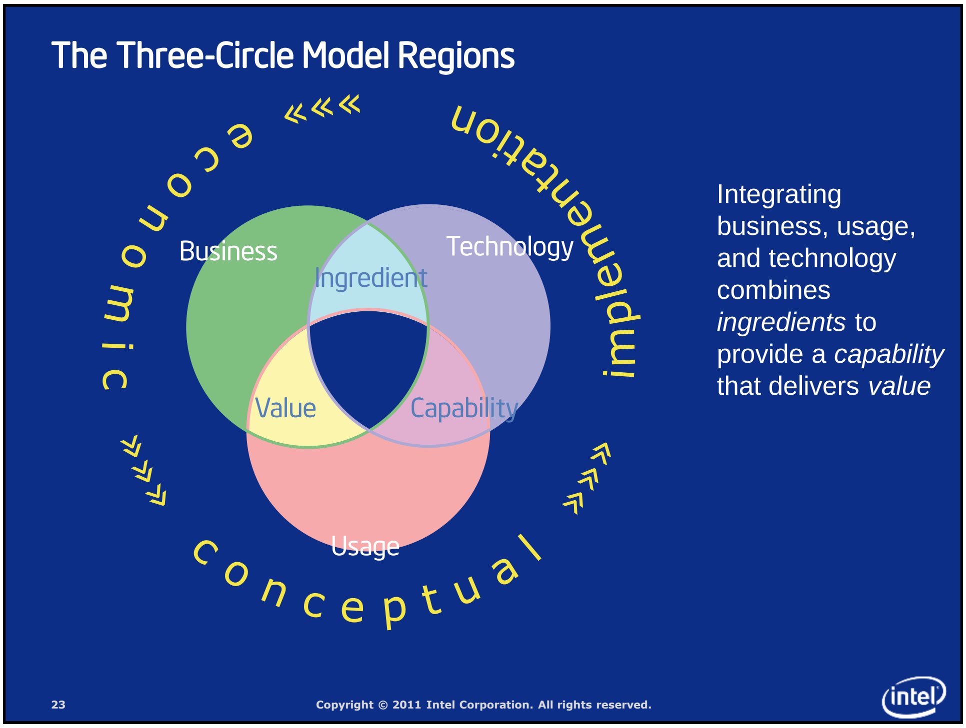
The Three-Circle Model


Three Fundamental Perspectives
The best platforms and products…
…are marketable and profitable
business
…are desirable, useful, and usable
usage
…are manufacturable and consumable
technology
19
Copyright © 2011 Intel Corporation. All rights reserved.
intel

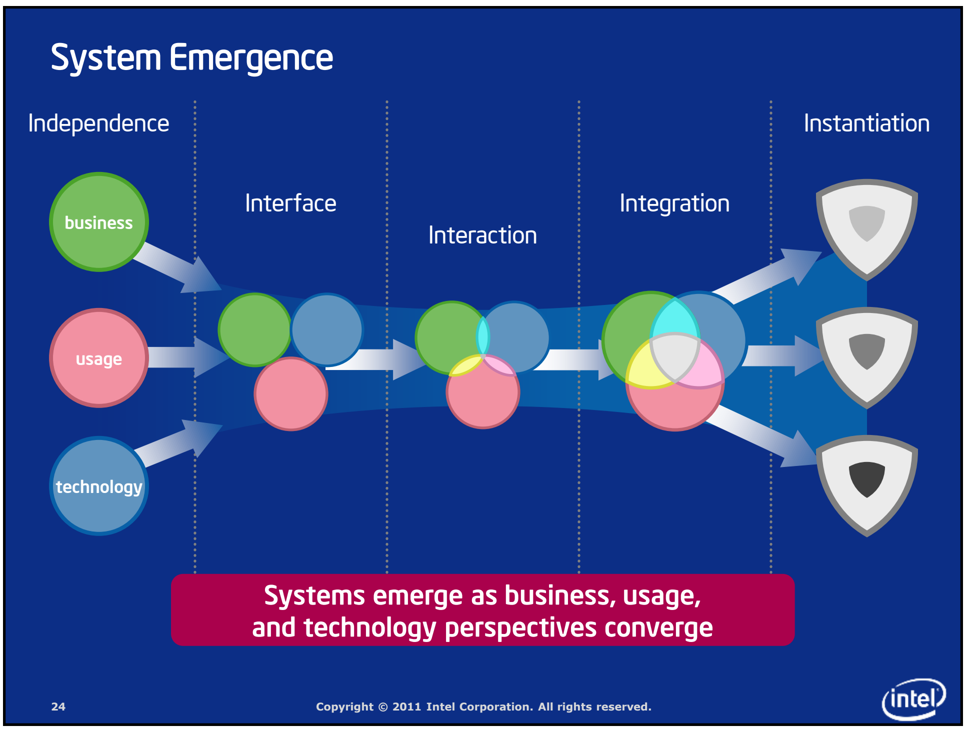

The Three-Circle Model
The three circles combine to create seven regions
business
technology
usage
Compelling, integrated systems are found in the center, balancing all three perspectives
20
Copyright © 2011 Intel Corporation. All rights reserved.
intel


Balanced System Development
Although the three circles in the model are shown at the same size, there is a need for balance, not necessarily equality
business
usage
Each new system presents its own challenges, as does the environment surrounding the system, the experience of the development team, and many other factors
Weighting business, usage, and technology perspectives according to these factors makes sense; ignoring a perspective does not
We need to develop systems using a balanced, systematic approach
21
Copyright © 2011 Intel Corporation. All rights reserved.
intel


Two-Circle Relationships
Value relates business and usage.
This interaction defines how usage contributes to market share, competitive advantage, and positioning
Capability relates usage and technology.
This interaction defines the interplay between usage, platform architecture, and supporting technologies
Ingredient relates technology and business.
This interaction defines how technologies drive profitability and marketability
22
Copyright © 2011 Intel Corporation. All rights reserved.
intel



The Three-Circle Model Regions
economic
implementation
conceptual
Business
Technology
Ingredient
Value
Capability
Usage
Integrating business, usage, and technology combines *ingredients* to provide a *capability* that delivers *value*
23
Copyright © 2011 Intel Corporation. All rights reserved.



System Emergence
Independence
Interface
Interaction
Integration
Instantiation
business
usage
technology
Systems emerge as business, usage, and technology perspectives converge
24
Copyright © 2011 Intel Corporation. All rights reserved.

Specification Basics

Specification Basics

These basic practices have a high return on investment:

- **Use a template** for requirements specification
- **Move from unconstrained natural language to constrained natural language** to reduce ambiguity and improve completeness with minimal effort
- **Do not include design statements** in the requirements unless they are there as intentionally-imposed constraints
- **Supplement natural language where needed** with other representations to improve comprehension and reduce ambiguity

Specification Basics, cont.

- **Quantify qualitative requirements** so they are verifiable
- **Define terms early and centrally** to ensure accurate use throughout the project
- **Validate requirements with stakeholders frequently** as a test of understanding
- **Rigorously review and inspect requirements** to prevent defects and maximize requirements quality

27

Attributes of a Good Requirement

- **Complete**: A requirement is complete when it contains sufficient detail for those that use it to guide their work
- **Correct**: A requirement is correct when it is error-free
- **Concise**: A requirement is concise when it contains just the necessary information, expressed in as few words as possible
- **Feasible**: A requirement is feasible if there is at least one design and implementation for it
- **Necessary**: A Requirement is necessary when it:
  - Is included to be market competitive
  - Can be traced to a stakeholder need
  - Establishes a new product differentiator or usage model
  - Is dictated by business strategy, roadmaps, or sustainability

28

Attributes of a Good Requirement

- **Prioritized:** A requirement is prioritized when it is ranked or ordered according to its importance
- **Unambiguous:** A requirement is unambiguous when it possesses a single interpretation
- **Verifiable:** A requirement is verifiable if it can be proved that the requirement was correctly implemented
- **Consistent:** A requirement is consistent when it does not conflict with any other requirements at any level
- **Traceable:** A requirement is traceable if it is uniquely and persistently identified with a Tag

29

Requirements vs. Design

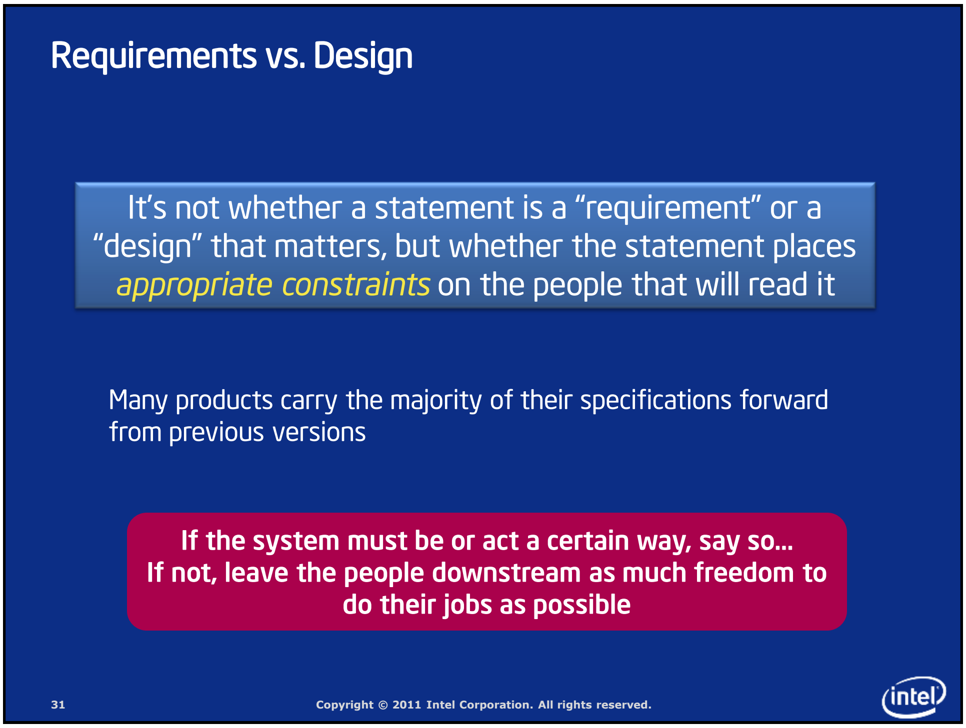
"Requirements are the *what*, design is the *how*..."

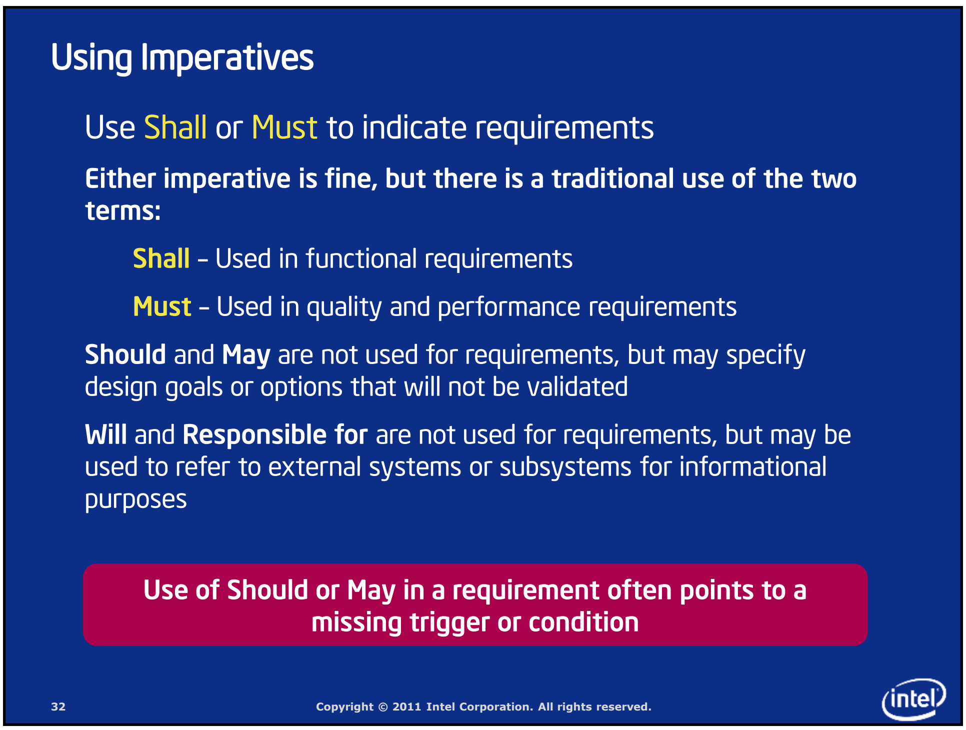
This is true – to a point, but the main difference between requirement and design is one of perspective:

How you look at a statement dictates whether it is a requirement or a design

Executive management: ***"A design to meet financial goals"*** → Build a media center PC ← Product development: ***"My requirements for this year"***

30


Requirements vs. Design
It's not whether a statement is a "requirement" or a "design" that matters, but whether the statement places appropriate constraints on the people that will read it
Many products carry the majority of their specifications forward from previous versions
If the system must be or act a certain way, say so…
If not, leave the people downstream as much freedom to do their jobs as possible
31
Copyright © 2011 Intel Corporation. All rights reserved.
intel


Using Imperatives
Use Shall or Must to indicate requirements
Either imperative is fine, but there is a traditional use of the two terms:
Shall – Used in functional requirements
Must – Used in quality and performance requirements
Should and May are not used for requirements, but may specify design goals or options that will not be validated
Will and Responsible for are not used for requirements, but may be used to refer to external systems or subsystems for informational purposes
Use of Should or May in a requirement often points to a missing trigger or condition
32
Copyright © 2011 Intel Corporation. All rights reserved.
intel

Negative Specification

It is appropriate to state what the system shall not do, but keep in mind that *the system shall not do much more than it shall do*

- Use negative specification sparingly, for emphasis
- Don't use negative specification for requirements that could be stated in the positive
- Avoid double negatives altogether

NO: "Users shall not be prevented from deleting data they have entered"

YES: "The system shall allow users to delete data they have entered"

33

Writing Functional Requirements

An excellent way to structure functional requirements is to use the following generic syntax:

[Trigger] [Precondition] Actor Action [Object]

Example:

When an Order is shipped and Order Terms are not "Prepaid", the system shall create an Invoice.

- Trigger: *When an Order is shipped*
- Precondition: *Order Terms are not "Prepaid"*
- Actor: *the system*
- Action: *create*
- Object: *an Invoice*

34

Writing Functional Requirements: EARS

A recent refinement of the generic syntax is the ***Easy Approach to Requirements Syntax*** (EARS) that contains patterns for specific types of functional requirements

Pattern Name	Pattern
Ubiquitous	The <system name> shall <system response>
Event-Driven	WHEN <trigger> <optional precondition> the <system name> shall <system response>
Unwanted Behavior	IF <unwanted condition or event>, THEN the <system name> shall <system response>
State-Driven	WHILE <system state>, the <system name> shall <system response>
Optional Feature	WHERE <feature is included>, the <system name> shall <system response>
Complex	(combinations of the above patterns)

35

Examples of Functional Requirement Syntax

The system shall allow the user to select a custom wallpaper for the display from any of the image files stored on the device.

When a user commands installation of an Application that accesses Communications Functions, the system shall prompt the user to acknowledge the access and agree before continuing installation.

When the system detects the user's face in proximity to the display while the phone function is active and Speaker Mode is off, the system shall turn off the display and deactivate the display's touch sensitivity.

While in Standby, if the battery capacity falls below 5% remaining, the system shall change the LED to flashing red.

36

Specifying Requirements Using Planguage

What is Planguage?

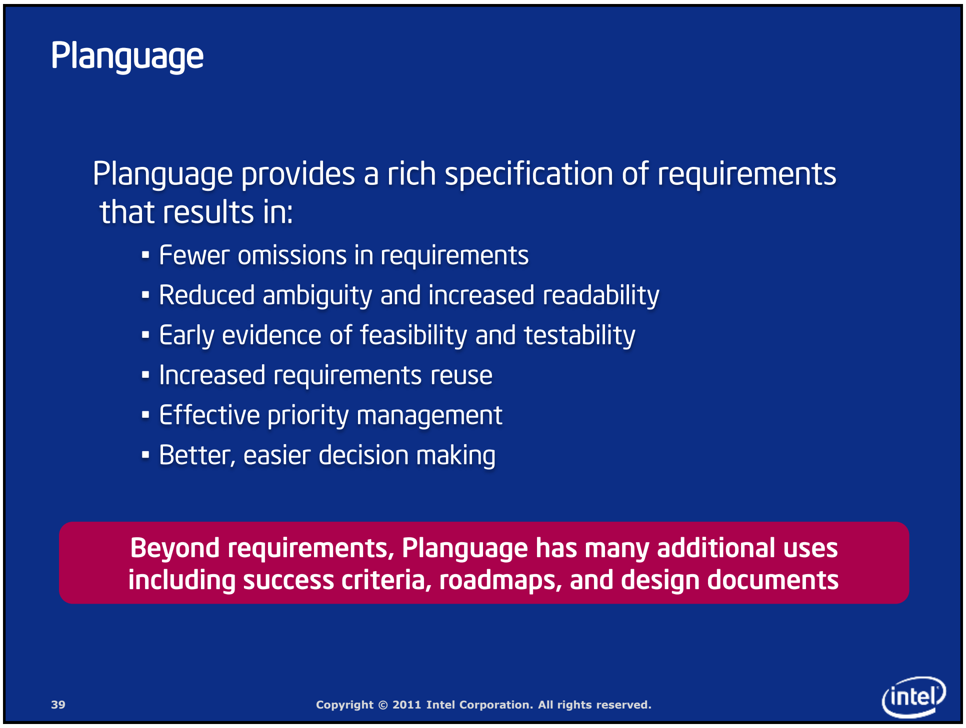
Planguage is an informal, but structured, keyword-driven planning language

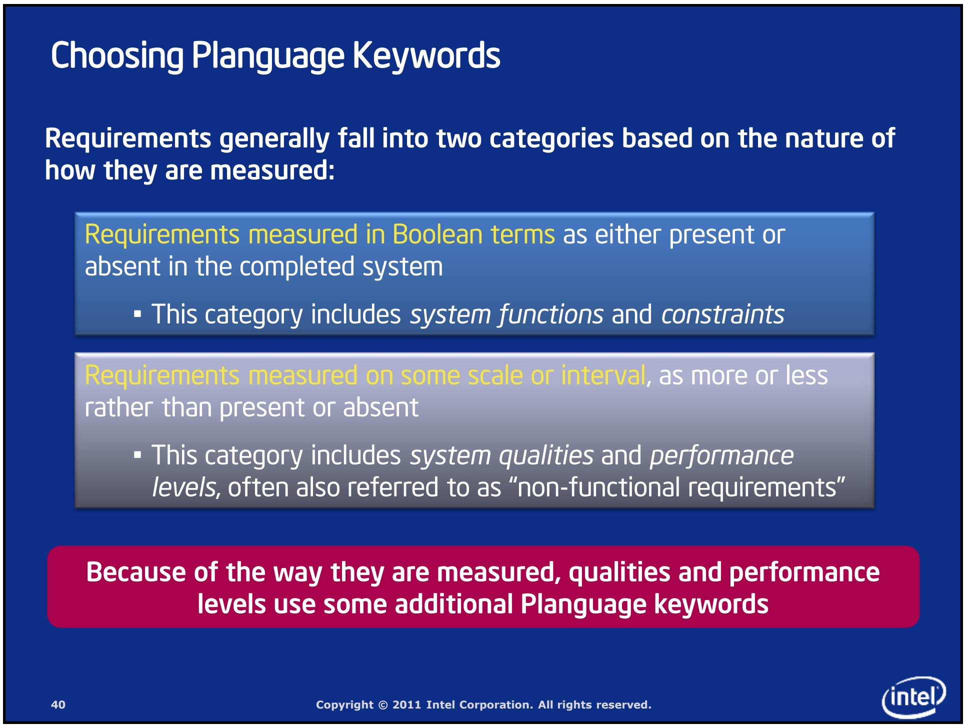
It can be used to create all types of requirements

The name Planguage is a combination of the words *Plan*ning and Lan*guage*

Planguage is an example of a Constrained Natural Language

Planguage aids communication about complex ideas


Planguage
Planguage provides a rich specification of requirements that results in:
▪ Fewer omissions in requirements
▪ Reduced ambiguity and increased readability
▪ Early evidence of feasibility and testability
▪ Increased requirements reuse
▪ Effective priority management
▪ Better, easier decision making
Beyond requirements, Planguage has many additional uses including success criteria, roadmaps, and design documents
39
Copyright © 2011 Intel Corporation. All rights reserved.


Choosing Planguage Keywords
Requirements generally fall into two categories based on the nature of how they are measured:
Requirements measured in Boolean terms as either present or absent in the completed system
▪ This category includes system functions and constraints
Requirements measured on some scale or interval, as more or less rather than present or absent
▪ This category includes system qualities and performance levels, often also referred to as "non-functional requirements"
Because of the way they are measured, qualities and performance levels use some additional Planguage keywords
40
Copyright © 2011 Intel Corporation. All rights reserved.

Basic Planguage Keywords for Any Requirement

ID: A unique, persistent identifier (often system-assigned)

Requirement: The text that details the requirement itself

Rationale: The reasoning that justifies the requirement

Priority: A rating of priority (numeric, HML, etc.)

Priority Reason: A short description of the requirement's claim on scarce resources; why is it rated as it is?

Tags: A set of keywords or phrases useful for sorting and searching

Stakeholders: a person or organization that influences a system's requirements or is impacted by that system

41

Basic Planguage Keywords for Any Requirement, cont.

Status: The status of the requirement (draft, committed, etc.)

Contact: The person who serves as a reference for the requirement

Author: The person that wrote the requirement

Revision: A version number for the statement

Date: The date of the most recent revision

Fuzzy concepts requiring more details: <*fuzzy concept*>

The source for any statement: ←

42

A Simple Planguage Requirement

ID: Invoice ← Christine Walsh

Requirement: When an Order is shipped and Order Terms are not "Prepaid", the system shall create an Invoice.

Rationale: Task automation decreases error rate, reduces effort per order. Meets corporate business principle for accounts receivable.

Priority: High. If not implemented, it will cause business process reengineering and reduce program ROI by $400K per year.

Stakeholders: Shipping, finance

Contact: Atul Gupta

Author, Revision, Date: Julie English, rev 1.0, 5 Oct 05

43

Additional Keywords for Quality and Performance Requirements

Ambition: A description of the goal of the requirement (this replaces the Requirement keyword used in functional requirements)

Scale: The scale of measure used to quantify the statement

Meter: The process or device used to establish location on a Scale

Minimum: The minimum level required to avoid political, financial, or other type of failure

Target: The level at which good success can be claimed

Outstanding: A stretch goal if everything goes perfectly

Past: An expression of previous results for comparison

Trend: An historical range or extrapolation of data

Record: The best known achievement

44

Quantifying Learnability

ID: Learnable ←C. Smith

Ambition: Make the system easy to learn ← VP marketing

Rationale: Upcoming hiring reflected in business plans makes learnability for order entry a critical success factor for new offices

Scale: Average time required for a Novice to complete a 1-item order using only the online help system for assistance.

Meter: Measurements obtained on 100 Novices during user interface testing.

Minimum: No more than 7 minutes

Target: No more than 5 minutes

Outstanding: No more than 3 minutes

Past: 11 minutes ← Recent site statistics

Defined: Novice: A person with less than 6 months experience with Web applications and no prior exposure to our Website.

45

Using Qualifiers

Qualifiers are expressed within square braces [ ] and may be used with *any* keyword

- They allow for conditions and events to be described, adding specificity to a requirement
- They most often contain data on *where*, *when*, etc.

Example: Instead of

Past: 11 minutes ← Recent site statistics

We could write

Past: [1st quarter average, all orders, all regions, new customers only] 11 minutes ← Recent site statistics

46

The Landing Zone

The Landing Zone

A Landing Zone is a table that defines a "region" of success for a product or project

The rows of the table contain the subset of requirements that directly define success or failure (*not* all the requirements)

The columns of the table contain a range of performance levels; usually, a Landing Zone covers the range between great success (Outstanding) and failure avoidance (Minimum)

Landing Zones can be used in agile development to help define success of an iteration or Scrum sprint

Landing Zones focus attention on what will create success

48

Example Landing Zone

Requirement	Outstanding	Target	Minimum
Retail On Shelf	Nov 15th	Nov. 22nd	Dec 1st
Manufacturing Cost	$9.00	$10.00	$11.50
Peak Project Headcount	250	350	400
Markets at Launch	US, APAC, EMEA	US, APAC	US, APAC
Design Wins at Launch	40+	30+	20+
Total First Year Volume	125K	110K	95K

49

Landing Zone Usage

Landing Zones are useful for several things:

- **Gain explicit consensus** at the start of a project on the definition of success
- **Quantify the achievement levels required** as an input to feasibility and risk analysis
- **Drive tradeoff discussions** and decision making throughout the project
- **Monitor and communicate** product attribute status to decision forums and management during development

50

Landing Zone Usage

Landing Zones help clarify decision authority for a team:

Decisions that do not violate any row of the LZ are made by the team as a normal part of their work

- So long as the team meets all LZ rows, that is success

Any decision that would cause any LZ row to be violated requires ratification from a higher authority

- This would include falling below Minimum or a decision to pursue something beyond Outstanding

Landing Zones can be created for platforms, components, service offerings, user experiences, projects, etc.

Landing Zone Variants

One Landing Zone variant adds a fourth column to monitor the level that the engineering team has committed to deliver:

Requirement	Outstanding	Target	Minimum	Commit

Another version drops the Outstanding level and replaces it with a Kill Switch level that, if reached, triggers a review meeting to consider stopping the project:

Requirement	Target	Minimum	Kill Switch

Customize Landing Zone format and content to meet your needs

Placing Functions in a Landing Zone

Landing Zone rows typically represent qualities and performance requirements that are measured across Minimum, Target, and Outstanding

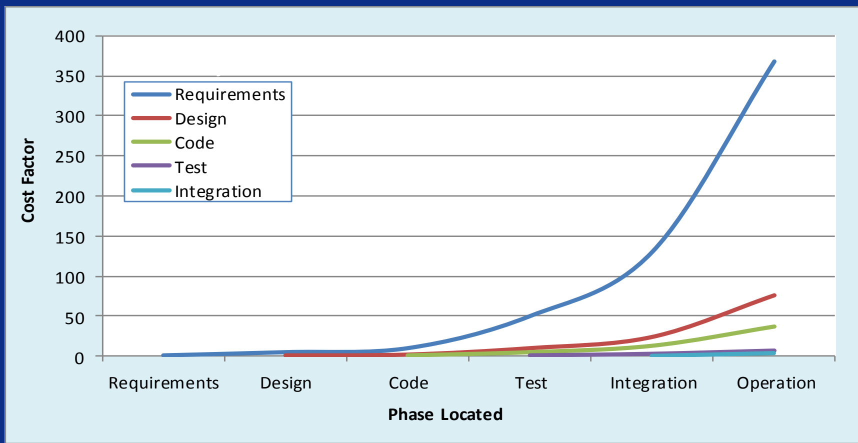
Functions do not fit this pattern, but can be included in a Landing Zone by placement in a single row, where Minimum – Outstanding show different lists of functions:

Requirement	Outstanding	Target	Minimum
Retail On Shelf	Nov 15th	Nov. 22nd	Dec 1st
Functions	Target + HTML5 support	Min + Quad monitor, 4G	Dual monitor support, 3G

53

Specification Quality Control

Defect Removal Cost
Relative to Phase Located



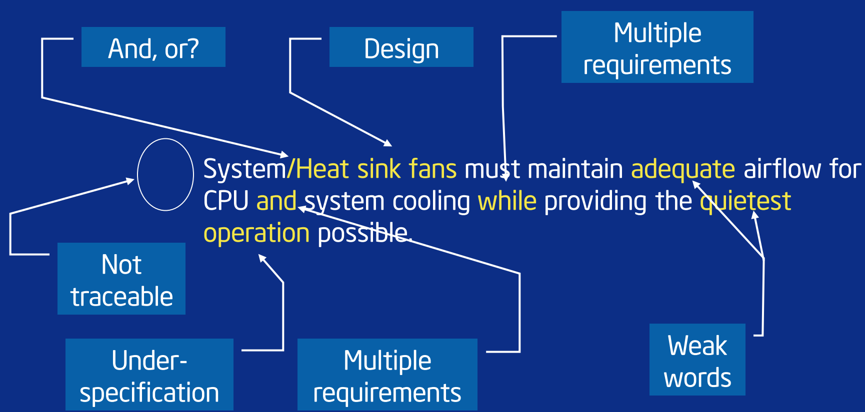
Source: NASA data, 2006

What's Wrong With My Requirements?

System/Heat sink fans must maintain adequate airflow for CPU and system cooling while providing the quietest operation possible.

See anything wrong?...

A Lot is Wrong, Actually…



Missing data: Source, status, rationale, priority, contact, etc.

Not verifiable as written

57

Peer Review Methods | Pros and Cons

	Pros	Cons
Informal Review	▪ Flexible ▪ Least threatening	▪ Finds fewer defects than other types ▪ Variable, inconsistent results
Walkthrough	▪ More systematic than reviews ▪ Identifies defects reviews miss	▪ May lack follow-up ▪ More time intensive and inconvenient than reviews
Inspection	▪ Most defects located ▪ Controlled, repeatable ▪ Industry proven practice	▪ Intimidating to some ▪ Requires training ▪ Can be too much effort without sampling

58

An Optimal Approach

An optimal requirements verification process would:

- Emphasize defect prevention and organizational learning
- Limit participant investment of time and energy to manageable levels
- Address the unique needs of each author and project
- Be suitable for all types and sizes of specification
- Rely on objective definitions and standards, not opinions
- Provide relevant, understandable metrics and indicators

59

The Answer: Specification Quality Control

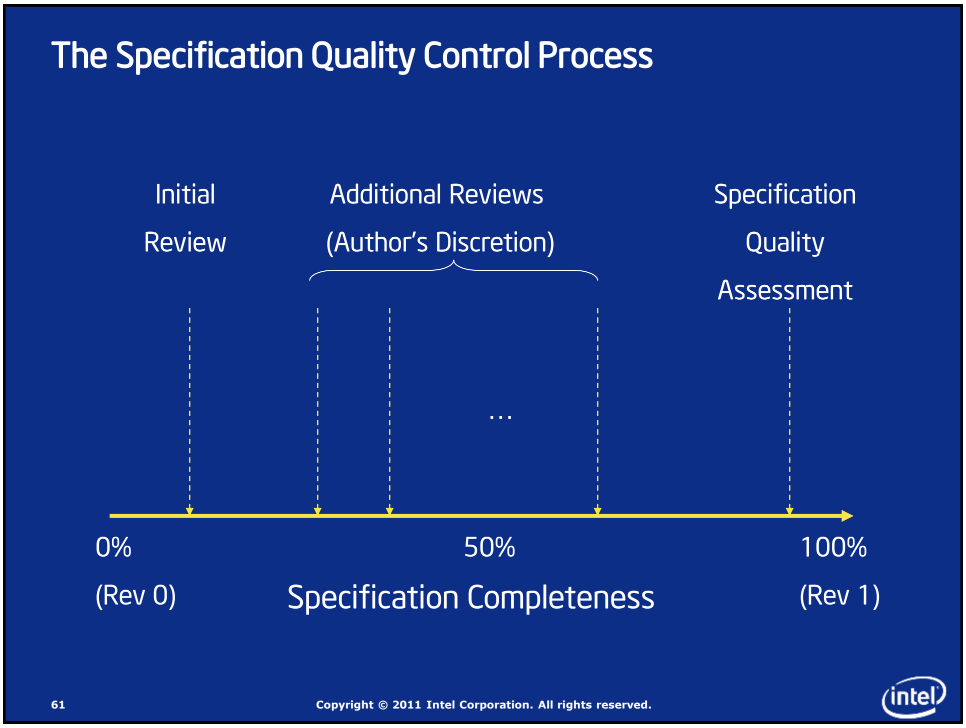
Specification Quality Control (SQC) is a method for ensuring *specifications* meet established quality goals according to objective, measured standards.

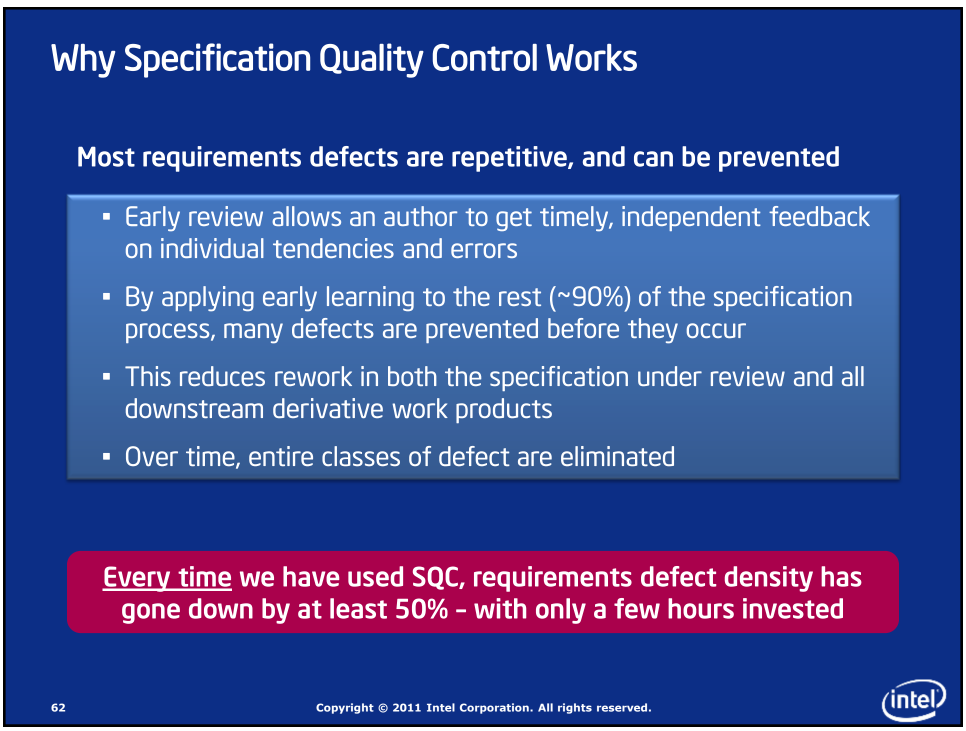
Specification Quality Control emphasizes:

- Cost and TTM reduction
- **Defect prevention**
- Resource efficiency
- Early learning
- Author confidentiality
- Quantified specification quality

Specification: Any representation (electronic or otherwise) of a requirement, constraint, design idea, plan, etc.

60


The Specification Quality Control Process
Initial
Review
Additional Reviews
(Author's Discretion)
Specification
Quality
Assessment
…
0%
(Rev 0)
50%
Specification Completeness
100%
(Rev 1)
61
Copyright © 2011 Intel Corporation. All rights reserved.
intel


Why Specification Quality Control Works
Most requirements defects are repetitive, and can be prevented
Early review allows an author to get timely, independent feedback on individual tendencies and errors
By applying early learning to the rest (~90%) of the specification process, many defects are prevented before they occur
This reduces rework in both the specification under review and all downstream derivative work products
Over time, entire classes of defect are eliminated
Every time we have used SQC, requirements defect density has gone down by at least 50% – with only a few hours invested
62
Copyright © 2011 Intel Corporation. All rights reserved.
intel

Sample SQC Results

- A team using Scrum reduced requirements defect density from **15** major defects per 600 words in one sprint to **4.5** in the next
- A security technology team reduced defect density from **35** major defects per 600 words to **15** on their first attempt, then went on to achieve less than **5** within another 12 months
- A large software team reduced defect density according to the following table:

Rev.	# of Defects	# of Pages	Defects/ Page (DPP)	% Change in DPP
0.3	312	31	10.06	
0.5	209	44	4.75	-53%
0.6	247	60	4.12	-13%
0.7	114	33	3.45	-16%
0.8	45	38	1.18	-66%
1.0	10	45	0.22	-81%
Overall % change in DPP revision 0.3 to 1.0:				**-98%**

63

Summary

64

Requirements Engineering in the early 21st Century

Requirements engineering is changing based on complexity, pace, consumer choice, and similar factors

Agility, hierarchy, and abstraction will be key to success in developing complex future systems

Adopting a systems engineering-based perspective helps ensure appropriate focus on emergent behaviors and cross-cutting concerns

Several pragmatic, simple requirements practices fit this environment well: EARS, Planguage, Landing Zones, and SQC are good examples

Questions?

Thank You!

65

Sources for More Information

Software Requirements (2nd ed.), Karl E. Wiegers, MS Press 2003

More About Software Requirements, Karl. E. Wiegers, MS Press 2005

Competitive Engineering, Tom Gilb, Elsevier 2005

Software & Systems Requirements Engineering: In Practice, Brian Barenbach et al, McGraw Hill 2009

Just Enough Requirements Management, Al Davis, Dorset House 2005

Requirements Engineering: From system goals to UML models to software specifications, Axel van Lamsweerde, Wiley 2009

Software Requirements – Styles and Techniques (2nd ed.), Søren Lauesen, Addison Wesley 2001

Customer-Centered Products, Creating Successful Products through Smart Requirements Management, Ivy Hooks and Kristin A. Farry, Amacom, 2001

66

Sources for More Information

Requirements Engineering: Processes and Techniques, Gerald Kotonya and Ian Sommerville, Wiley 1999

Exploring Requirements: Quality Before Design, Donald Gause and Gerald Weinberg, Dorset House 1988

Effective Requirements Practices, Ralph Young, Addison Wesley 2001

Managing Software Requirements: A Unified Approach (2nd ed.), Dean Leffingwell and Don Widrig, Addison Wesley 2003

Non-Functional Requirements in Software Engineering, Lawrence Chung et al., Kluwer Academic Publishers 2000

System and Software Requirements Engineering (2nd Ed.), Richard H. Thayer and Merlin Dorfman (ed), IEEE 1997

Mastering the Requirements Process (2nd Ed.), James and Suzanne Robertson, Addison Wesley 1999

Backup

Some Additional Planguage Keywords

Gist: A brief summary of the requirement or area addressed

Assumptions: All assumptions or assertions that could cause problems if untrue now or later

Risks: Anything that could cause malfunction, delay, or other negative impacts on expected results

Defined: The definition of a term (better to use a glossary)

Wish: A desirable level of achievement that may not be attainable through available means

Kill Switch: A level at which the project would be cancelled or the product withdrawn from the market

{*item1, item2, …*} A collection of objects

See *Competitive Engineering* by Tom Gilb, or visit www.gilb.com for a complete list of keywords

69

Planguage Synonyms

The default version of Planguage as created by Tom Gilb uses different terms for a few of the keywords than Intel:

Default Terms		Intel's Terms
Must	→	Minimum
Plan	→	Target
Stretch	→	Outstanding
Catastrophe	→	Kill Switch
Tag	→	ID

Rather than use Tag as the unique ID for a requirement, Intel uses Tags to capture keywords or phrases used to search and sort, in keeping with common social networking use of the term

70

Example of Early Planguage Use (1988)

Usability Attribute	Measuring Technique	Metric	Worst-Case Level	Planned Level	Best-Case Level
Initial use	NOTES benchmark task	Number of successful interactions in 30 minutes	1-2	3-4	8-10
Initial evaluation	Attitude questionnaire	Evaluation score (0 to 100)	50	67	83
Error recovery	Critical-incident analysis	Percent incidents “covered”	10%	50%	100%

This table comes from a Usability Specification written by DEC in 1988 for VAX NOTES Version 1.0. It bears a striking resemblance to a landing zone.

71

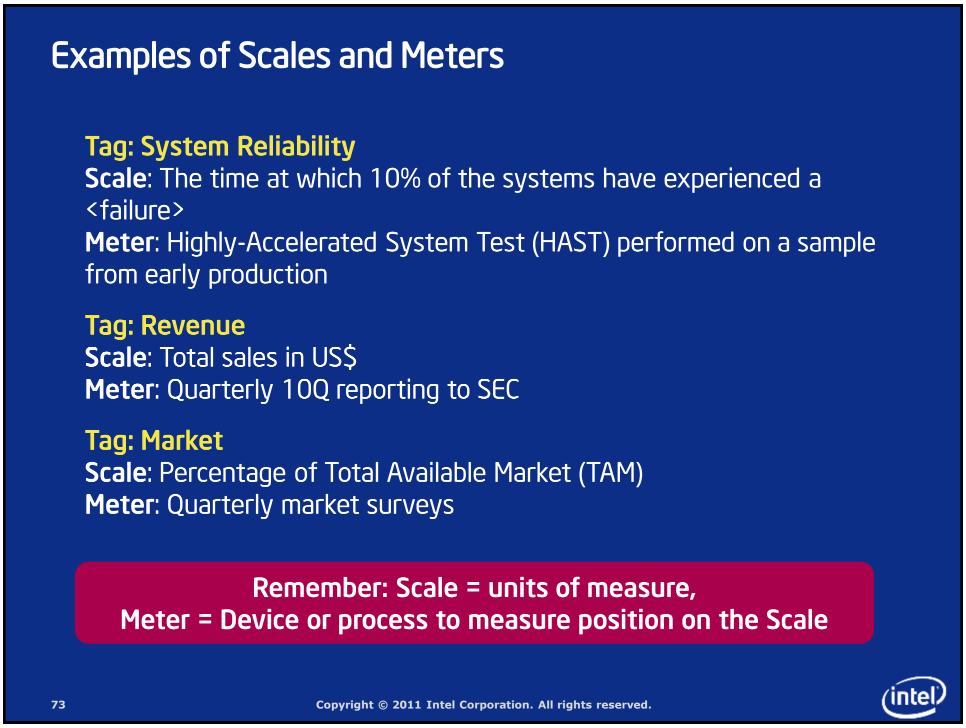
Examples of Scales and Meters

Tag: Environmental Noise
Scale: dBA at 1 meter
Meter: Lab measurements performed according to a <standard environmental test process>

Tag: Software Security
Scale: Time required to break into the system
Meter: An attempt by a team of experts to break into the system using commonly available tools

Tag: Software Maintainability
Scale: Average engineering time from report to closure of defects
Meter: Analysis of 30 consecutive defects reported and corrected during product development

72


Examples of Scales and Meters
Tag: System Reliability
Scale: The time at which 10% of the systems have experienced a <failure>
Meter: Highly-Accelerated System Test (HAST) performed on a sample from early production
Tag: Revenue
Scale: Total sales in US$
Meter: Quarterly 10Q reporting to SEC
Tag: Market
Scale: Percentage of Total Available Market (TAM)
Meter: Quarterly market surveys
Remember: Scale = units of measure,
Meter = Device or process to measure position on the Scale
73
Copyright © 2011 Intel Corporation. All rights reserved.
intel