Requirements Engineering

2019/2020

Course 4

Course 4 outline

- SysML
  - Requirements diagram
- Executable Specification
  - Behavior Driven Development

Course 4 bibliography

- OMG Systems Modeling Language:
  http://www.omgsysml.org/
- Mario Cardinal, *Executable Specifications with Scrum. A Practical Guide to Agile Requirements Discovery,* Addison-Wesley, 2013
- Behaviour-Driven Development
  http://behaviour-driven.org/
- Dan North, Introducing BDD
  http://dannorth.net/introducing-bdd/
- Andrew Glover, *Drive development with easyb*

http://www.ibm.com/developerworks/java/tutorials/j-easyb/

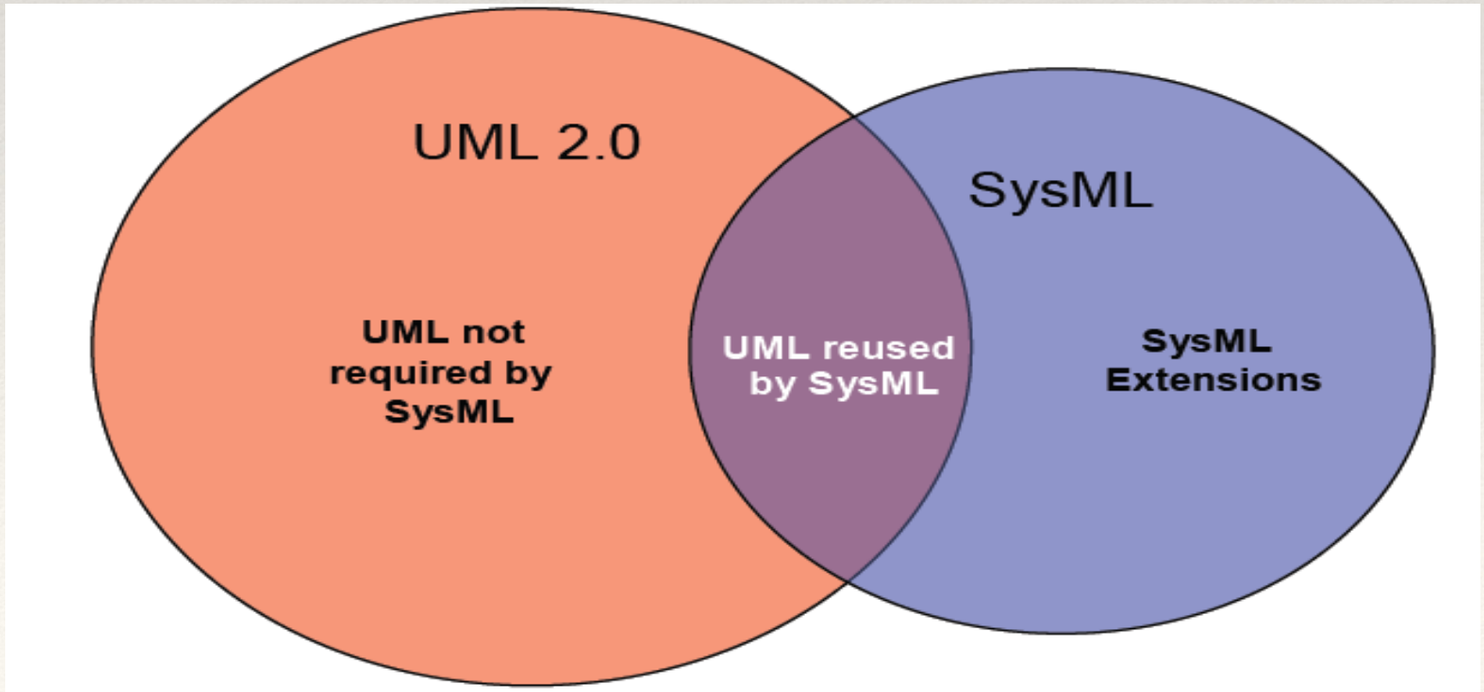
- http://www.easyb.org/

OMG Systems Modeling Language

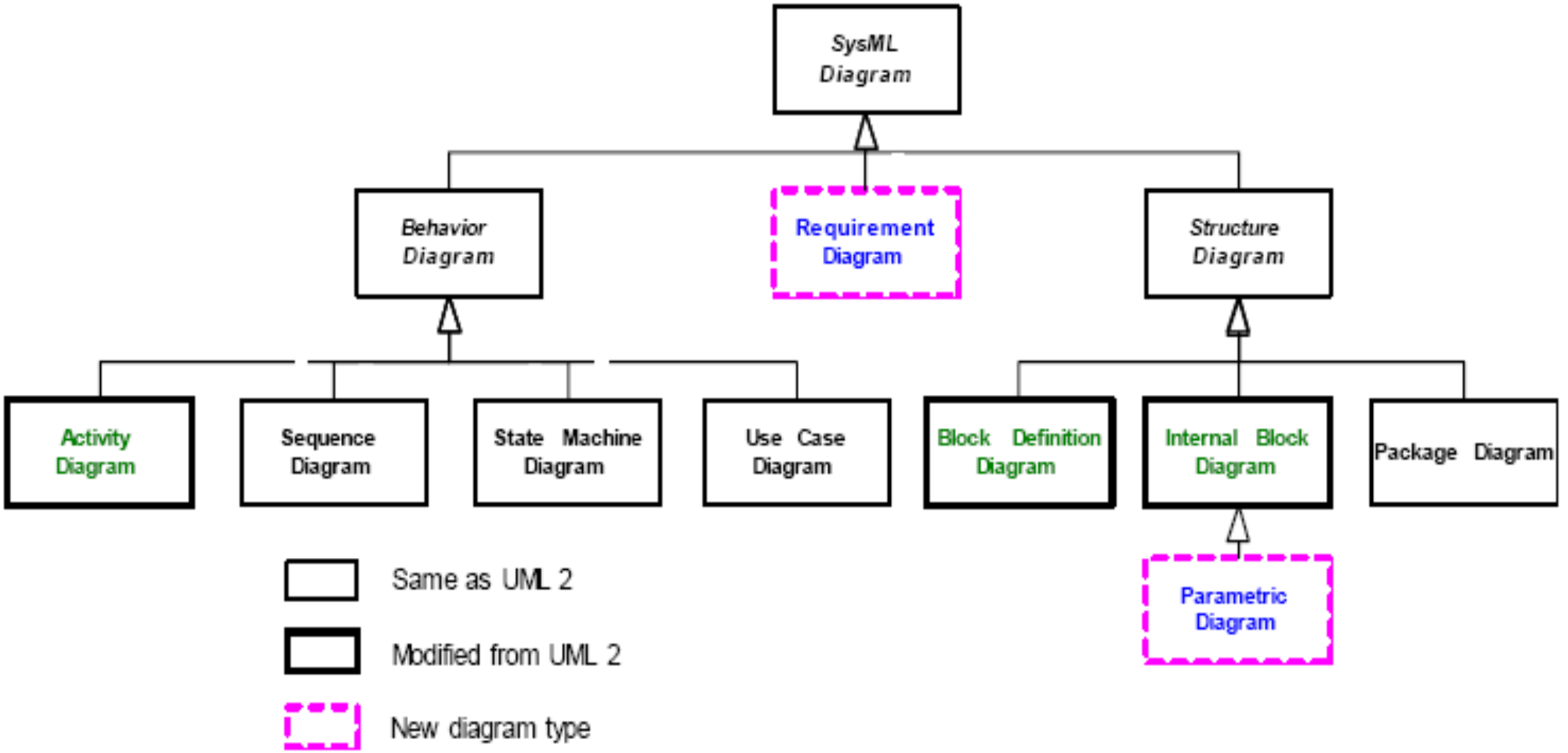
- Defines a general-purpose modeling language for systems engineering applications (also known as SysML).
- SysML supports the specification, analysis, design, verification, and validation of a broad range of complex systems.
- These systems may include hardware, software, information, processes, personnel, and facilities.
- SysML reuses a subset of UML 2 and provides additional extensions needed to address the requirements in the UML for Systems Engineering Request for Proposal (2003).
- SysML uses the UML 2 extension mechanisms (stereotypes, etc).
- It is particularly effective in specifying requirements, structure, behavior, allocations, and constraints on system properties to support engineering analysis.

OMG SysML

- SysML reuses many of the major diagram types of UML. In some cases, the UML diagrams are strictly reused (use case, sequence, state machine, and package diagrams), in other cases they are modified so that they are consistent with SysML extensions.
- SysML does not use all of the UML diagram types: the object diagram, communication diagram, interaction overview diagram, timing diagram, deployment diagram, and profile diagram.
- Two new diagram types: the requirements and the parametric diagram.



SysML Diagram Taxonomy



Requirements in SysML

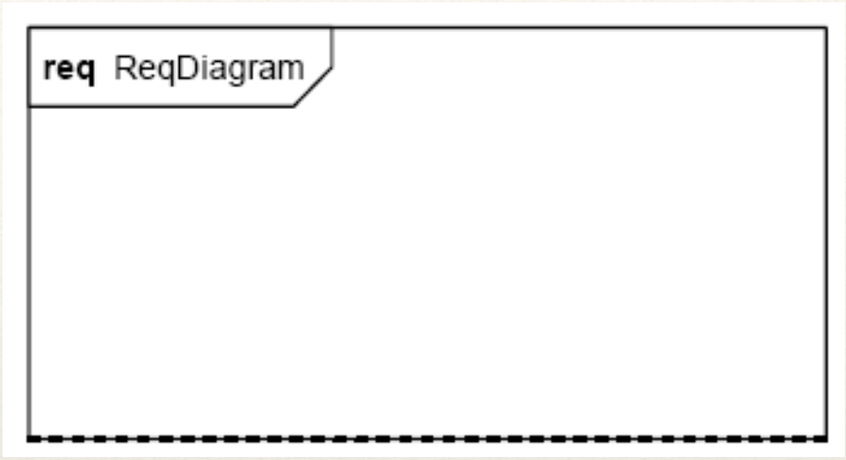
- The SysML Requirements diagram is intended to help in better organizing requirements, and also to explicitly show the various kinds of relationships between different requirements.
- The SysML requirements constructs are intended to provide a bridge between traditional requirements management tools and the other SysML models.
- When combined with UML for software design, the requirements constructs can also fill the gap between user requirements specification, normally written in natural language, and Use Case diagrams, used as initial specification of system requirements.
- A SysML requirement can also appear on other diagrams to show its relationship to other modeling elements.

Requirements in SysML

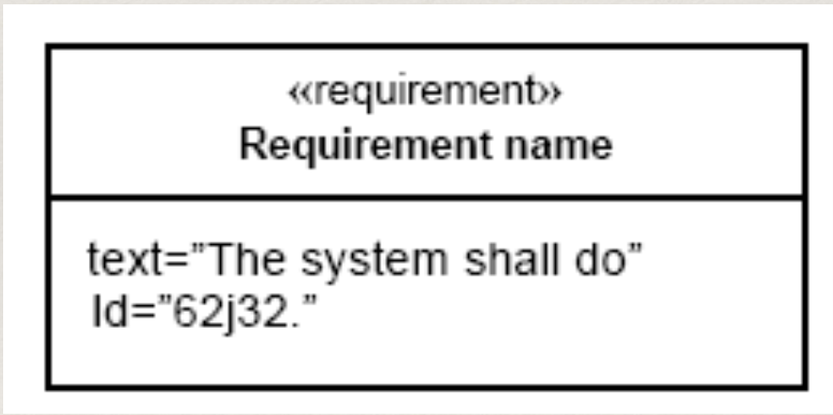
- Text-based requirements can be represented and related to other modeling elements.
- The requirements diagram can depict the requirements in graphical, tabular, or tree structure format.
- A requirement is defined as a stereotype of UML Class subject to a set of constraints.
- A standard requirement includes properties to specify its unique identifier and text requirement. Additional properties such as verification status, can be specified by the user.

Requirements in SysML

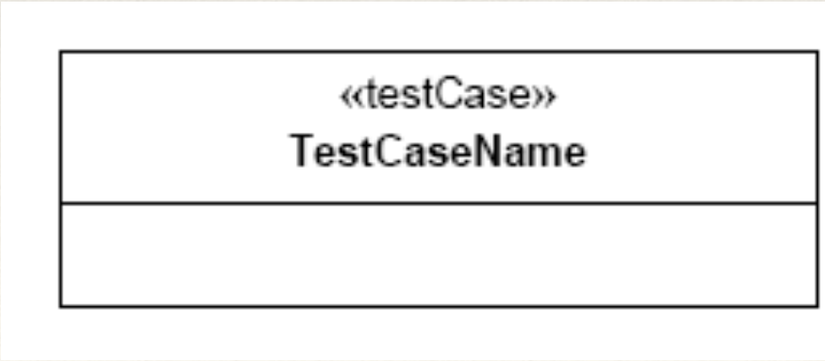
- Requirements diagram:



- Requirement:



- TestCase:



Requirements in SysML

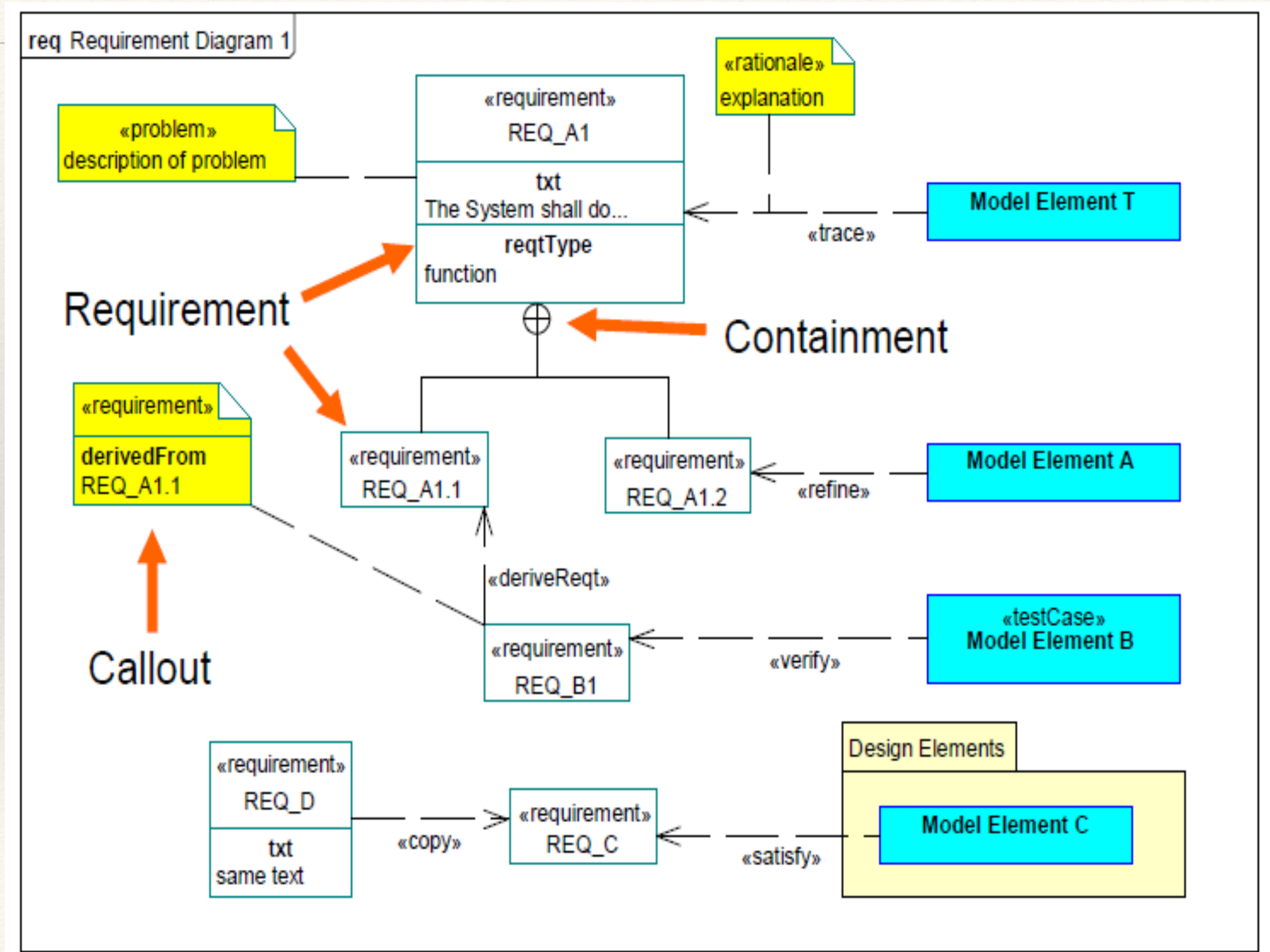
- Several requirements relationships are specified to relate requirements to other requirements or to other model elements.
- Relationships for:
  - defining a requirements hierarchy,
  - deriving requirements,
  - satisfying requirements,
  - verifying requirements,
  - refining requirements.
- The relationships can improve the specification of systems, as they can be used to model requirements.

Requirements relationships in SysML

Types of relationships:

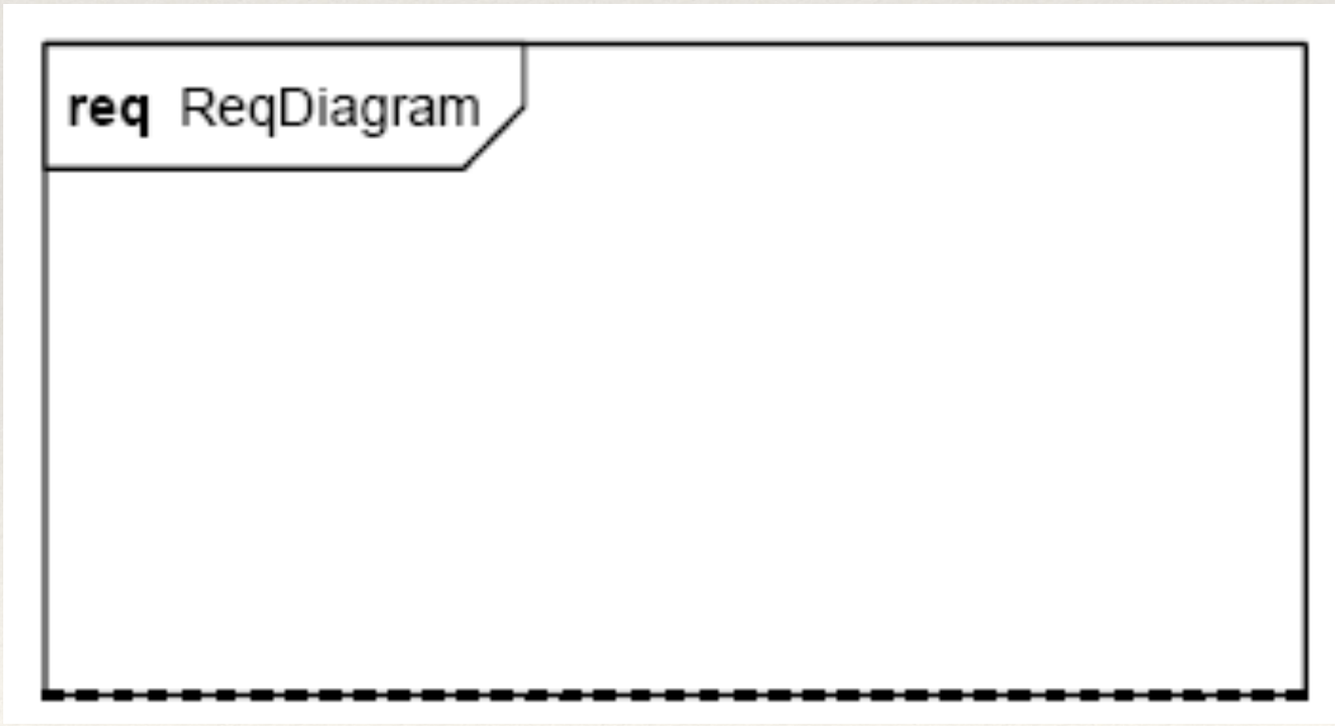
- *master/slave*: A slave requirement is a requirement whose text property is a read-only copy of the text property of a master requirement.
- *derive*: relates a derived requirement to its source requirement
- *satisfy*: describes how a design or implementation model satisfies one or more requirements.
- *verify*: defines how a test case or other model element verifies a requirement.
- *refine*: can be used to describe how a model element or set of elements can be used to further refine a requirement
- *trace*: provides a general-purpose relationship between a requirement and any other model element.

Requirements diagram elements



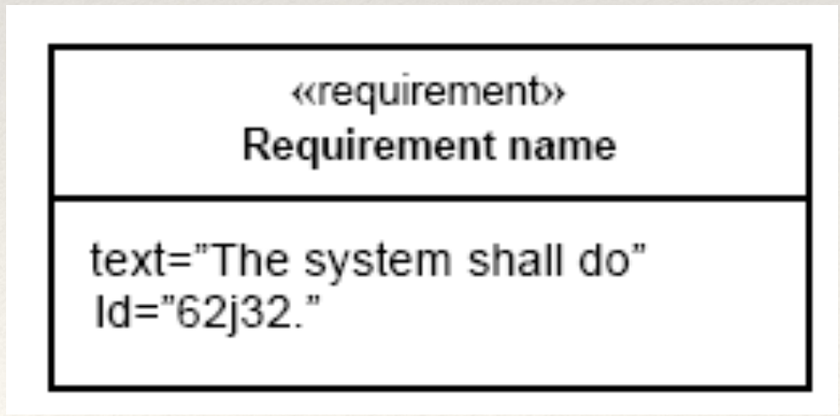
Requirement Diagram

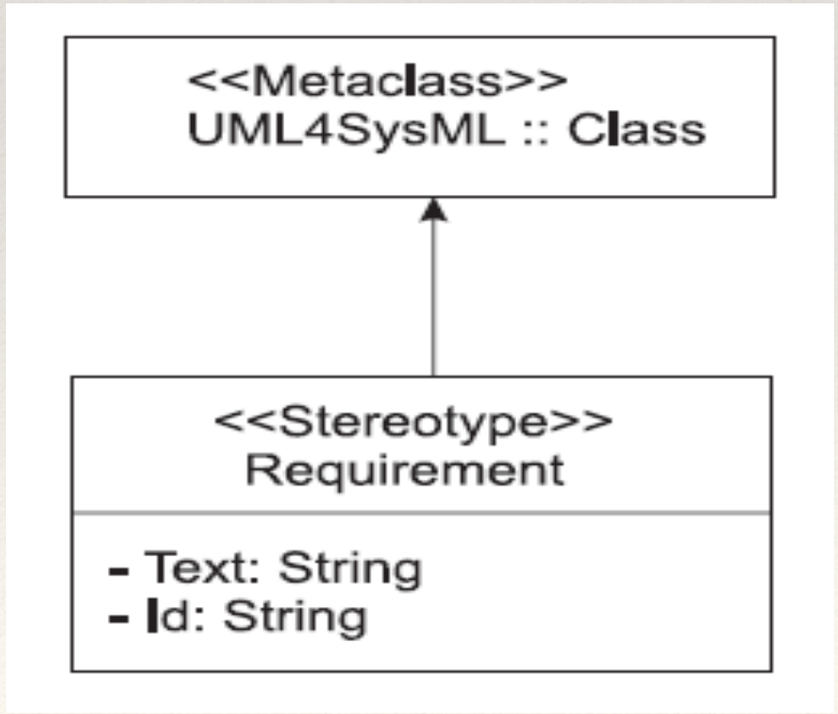
- The Requirement Diagram can only display requirements, packages, other classifiers, test cases, and rationale.
- The relationships for *containment*, *deriveReqt*, *satisfy*, *verify*, *refine*, *copy*, and *trace* can be shown on a requirement diagram.
- SysML notation:



Requirement Notation

- A requirement may specify a function that a system must perform or a performance condition that a system must satisfy.
- Requirements are used to establish a contract between the customer (or other stakeholder) and those responsible for designing and implementing the system.
- A requirement is a stereotype of UML Class.
- SysML notation:





Requirement Notation

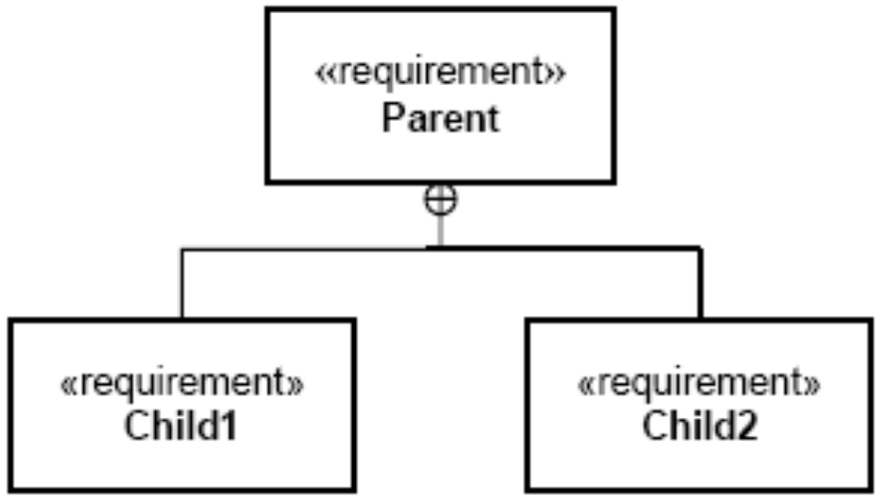
- Attributes:
  - text: String The textual representation or a reference to the textual representation of the requirement.
  - id: String The unique id of the requirement.
- Constraints:
  - Classes stereotyped by «requirement» may not participate in associations.
  - Classes stereotyped by «requirement» may not participate in generalizations.
  - A nested classifier of a class stereotyped by «requirement» must also be stereotyped by «requirement».

Compound requirements

- They can be created by using the nesting capability of the class definition mechanism.
- The default interpretation of a compound requirement (unless stated differently by the compound requirement itself) is that all its subrequirements must be satisfied for the compound requirement to be satisfied.
- Subrequirements can be accessed through the “nestedClassifier” property of a class.
- When a requirement has nested requirements, all the nested requirements apply as part of the container requirement.
- Deleting the container requirement deletes the nested requirements, a functionality inherited from UML.

Composite requirements

- A composite requirement can contain subrequirements in terms of a requirements hierarchy, specified using the UML namespace containment mechanism.
- This relationship enables a complex requirement to be decomposed into its containing child requirements.
- A composite requirement may state that the system shall do A and B and C, which can be decomposed into the child requirements that the system shall do A, the system shall do B, and the system shall do C.
- An entire specification can be decomposed into children requirements, which can be further decomposed into their children to define the requirements hierarchy.
- SysML



Slave requirements

- The concept of requirements reuse is very important in many applications.
- SysML introduces the concept of a slave requirement.
- A slave requirement is a requirement whose text property is a read-only copy of the text property of a master requirement.
- The text property of the slave requirement is constrained to be the same as the text property of the related master requirement.
- The master/slave relationship is indicated by the use of the copy relationship.

Copy relationship

- A Copy relationship is a dependency between a supplier requirement and a client requirement that specifies that the text of the client requirement is a read-only copy of the text of the supplier requirement.
- A Copy dependency created between two requirements maintains a master/slave relationship between the two elements for the purpose of requirements re-use in different contexts.
- SysML notation:

- Constraints:
  - A Copy dependency may only be created between two classes that have the "requirement" stereotype, or a subtype of the "requirement" stereotype applied.
  - The text property of the client requirement is constrained to be a read-only copy of the text property of the supplier requirement.
  - Previous constraint is applied recursively to all subrequirements.

DeriveReqt relationship

- A DeriveReqt relationship is a dependency between two requirements in which a client requirement can be derived from the supplier requirement.
- The arrow direction points from the derived (client) requirement to the (supplier) requirement from which it is derived.
- SysML notation:

- Constraints
  - The supplier must be an element stereotyped by «requirement» or one of «requirement» subtypes.
  - The client must be an element stereotyped by «requirement» or one of «requirement» subtypes.

Satisfy relationship

- A Satisfy relationship is a dependency between a requirement and a model element that fulfills the requirement.
- The arrow direction points from the satisfying (client) model element to the (supplier) requirement that is satisfied.
- SysML notation:

- Constraints
  - The supplier must be an element stereotyped by «requirement» or one of «requirement» subtypes.

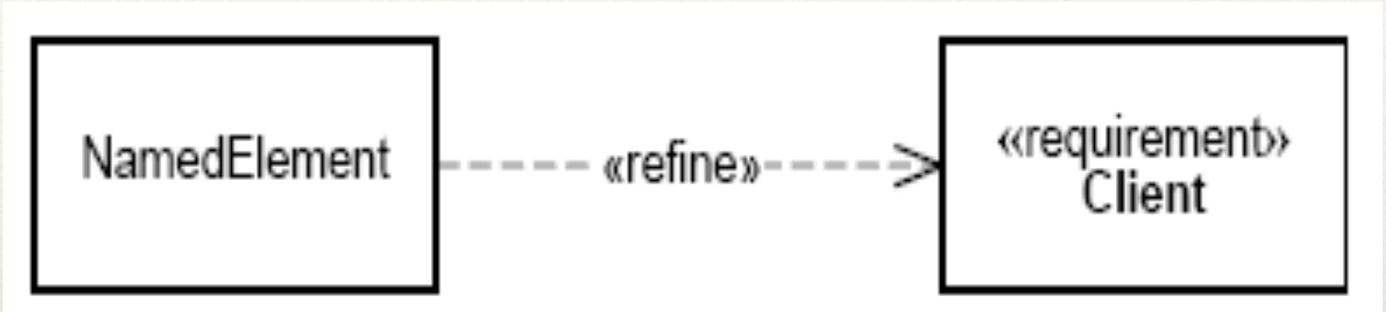
Verify relationship

- A Verify relationship is a dependency between a requirement and a test case or other model element that can determine whether a system fulfills the requirement.
- The arrow direction points from the (client) element to the (supplier) requirement.
- SysML notation:

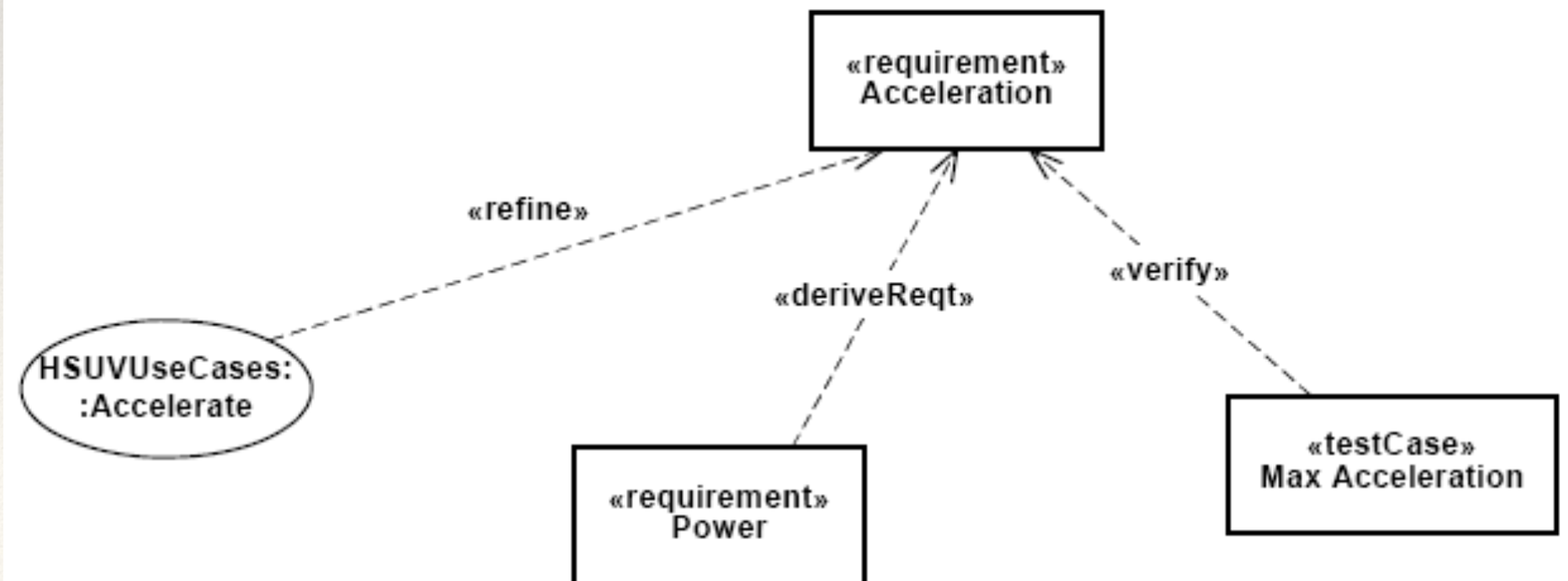
- Constraints
  - The supplier must be an element stereotyped by «requirement» or one of «requirement» subtypes.

Refine relationship

- The refine requirement relationship can be used to describe how a model element or set of elements can be used to further refine a requirement.
- SysML notation:
- Eg.



- a use case or activity diagram may be used to refine a text-based functional requirement.
- refine relationship may be used to show how a text-based requirement refines a model element.

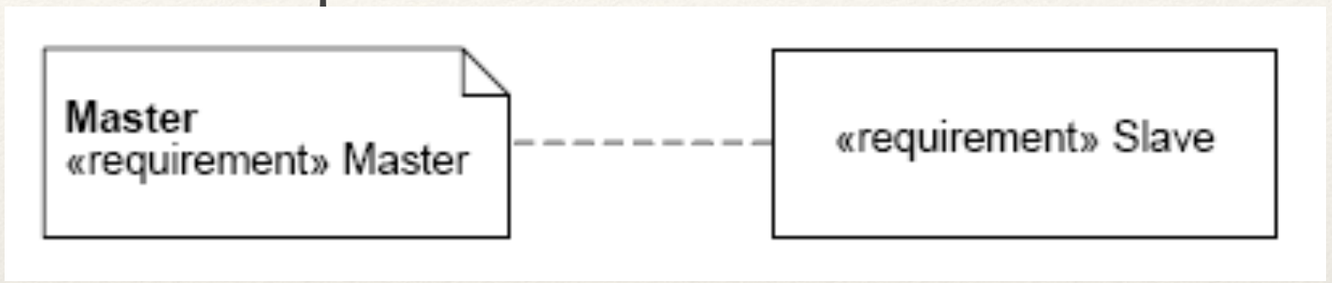


Trace relationship

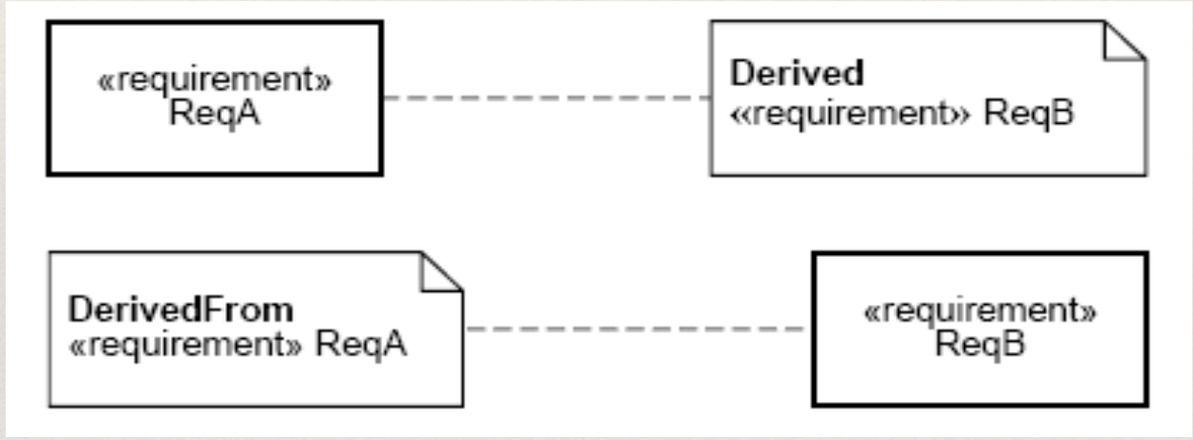
- A generic trace requirement relationship provides a general-purpose relationship between a requirement and any other model element.
- The semantics of trace include no real constraints and therefore are quite weak.
- It is recommended that the trace relationship not be used in conjunction with the other requirements relationships.

Requirement Property Callout Format

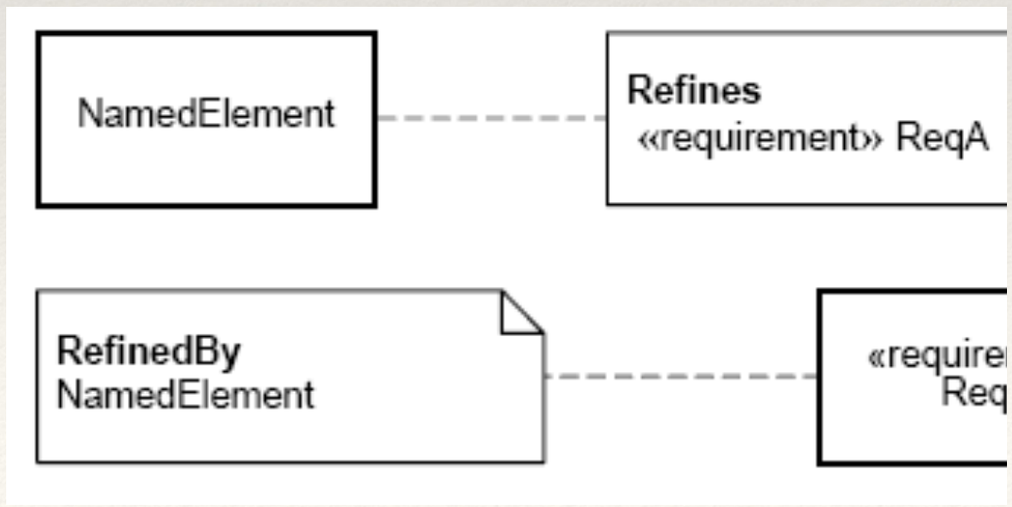
- A callout notation can be used to represent derive, satisfy, verify, refine, copy, and trace relationships.
- Master callout:



- Derive callout:

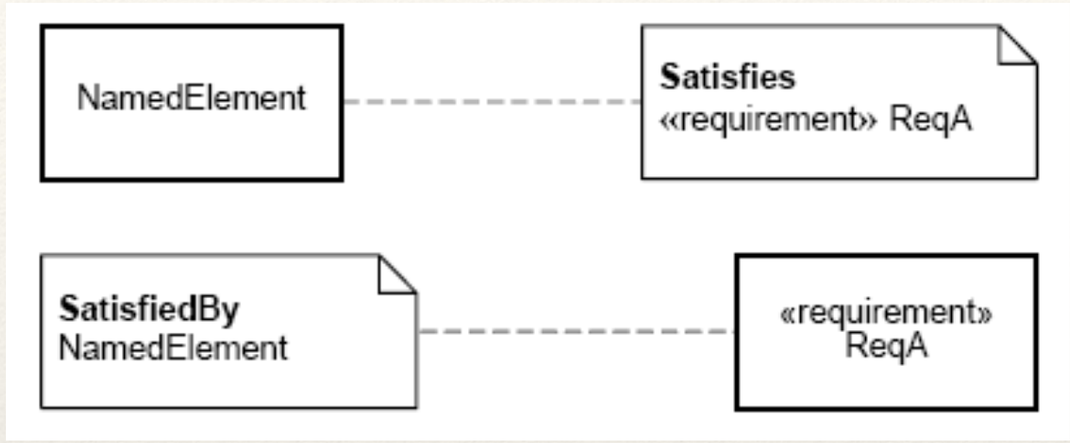


- Refine callout:

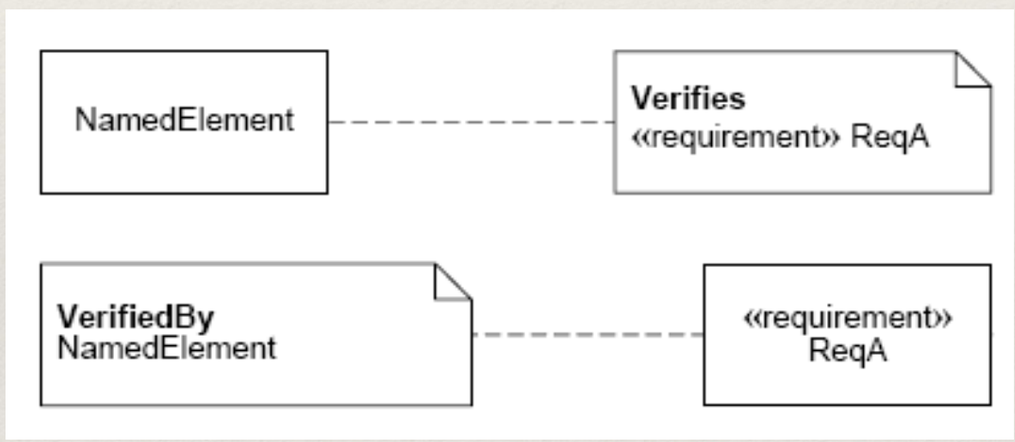


Requirement Property Callout Format

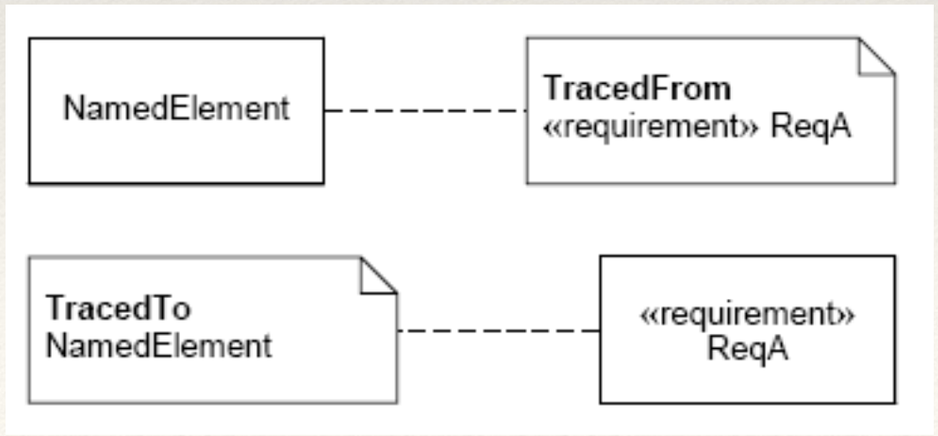
- Satisfy callout:



- Verify callout:



- Trace callout:



Requirements Table

- SysML allows the representation of requirements, their properties and relationships in a tabular format.
- The table may contain:
  - Requirements with their properties in columns.
  - A column that includes the supplier for any of the dependency relationships (Derive, Verify, Refine, Trace).
  - A column that includes the model elements that satisfy the requirement.
  - A column that represents the rationale for any of the above relationships, including reference to analysis reports for trace rationale, trade studies for design rationale, or test procedures for verification rationale.
- The relationships between requirements and other objects can also be shown using a sparse matrix style.
- The table should include the source and target elements names (and optionally kinds) and the requirement dependency kind.

Requirements Table

table [requirement] Performance [Decomposition of Performance Requirement]

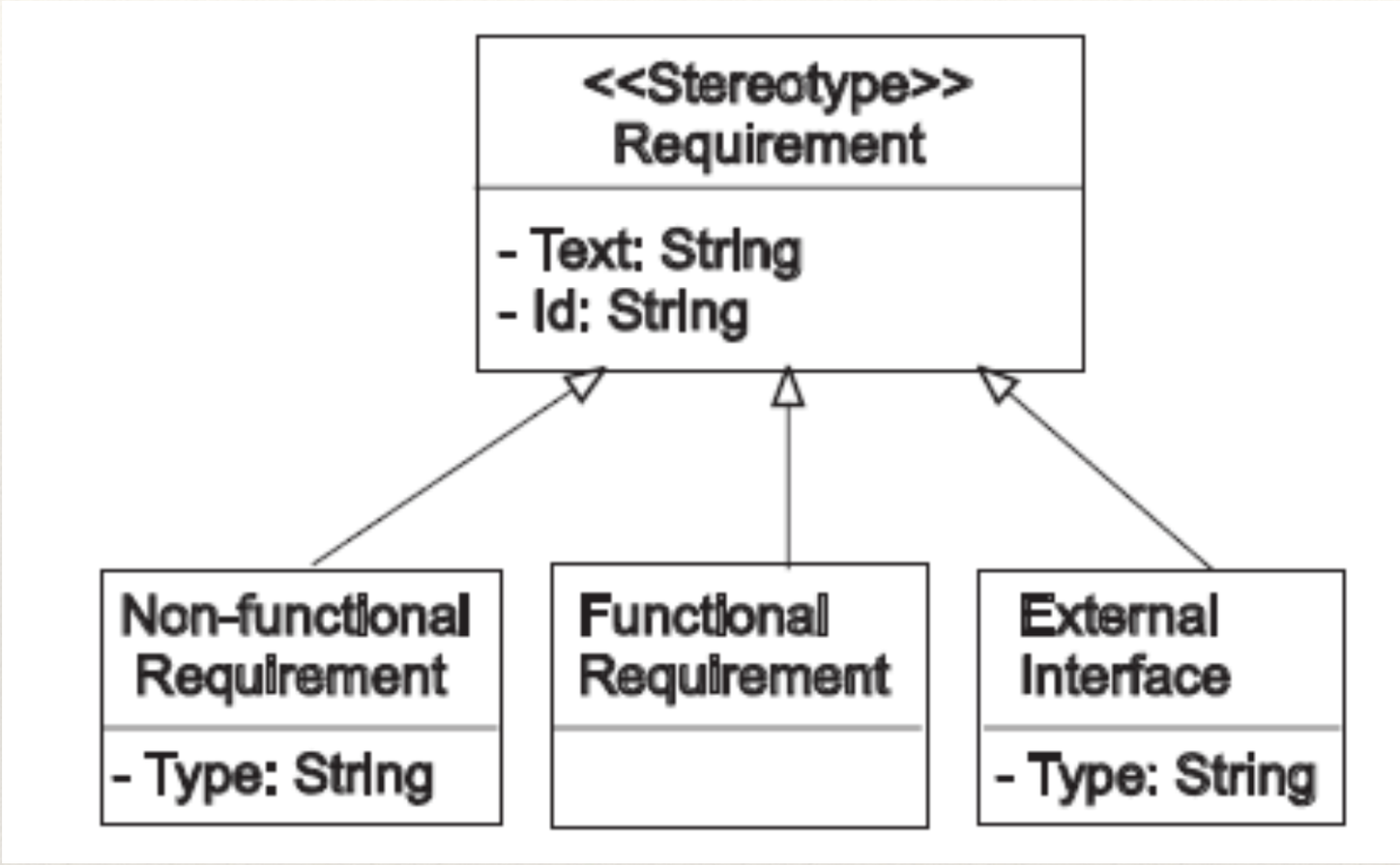
id	name	text
2	Performance	The Hybrid SUV shall have the braking, acceleration, and off-road capability of a typical SUV, but have dramatically better fuel economy.
2.1	Braking	The Hybrid SUV shall have the braking capability of a typical SUV.
2.2	FuelEconomy	The Hybrid SUV shall have dramatically better fuel economy than a typical SUV.
2.3	OffRoadCapability	The Hybrid SUV shall have the off-road capability of a typical SUV.
2.4	Acceleration	The Hybrid SUV shall have the acceleration of a typical SUV.

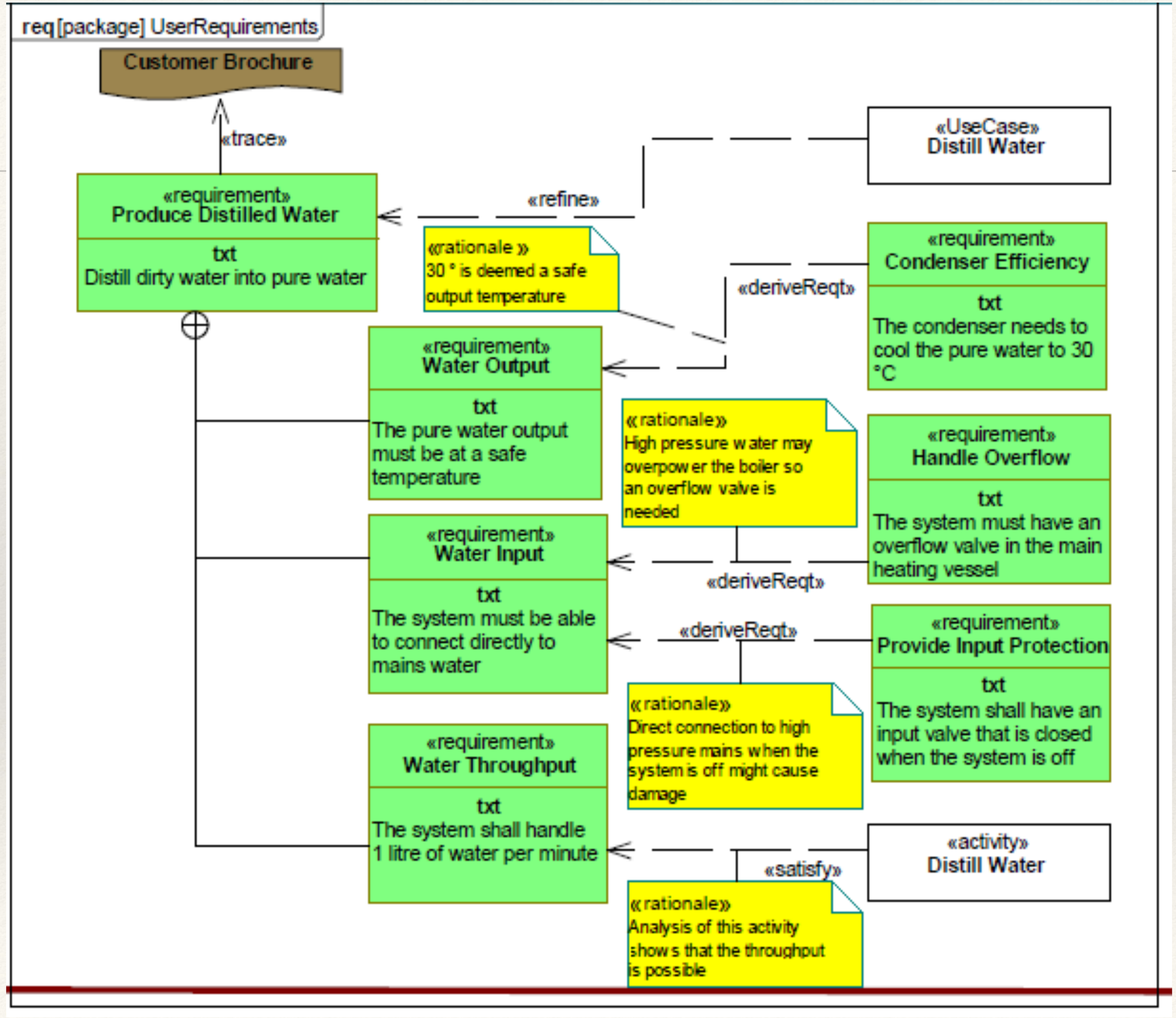
Id	Name	RelatesTo	RelatesHow	Type
TMC17	Traffic management configuration information	{TMC15, TMC16}	deriveReqt	Functional
TMC20	Make function/context obvious	{TMC18, TMC19}	deriveReqt	Functional
TMC21	Education material	{TMC18, TMC19}	deriveReqt	Functional

Requirements Extensions

- Modelers can customize requirements taxonomies by defining additional subclasses of the Requirement stereotype.
- E.g., a modeler may want to define requirements categories to represent operational, functional, interface, performance, physical, storage, activation/deactivation, design constraints, and other specialized requirements such as reliability and maintainability, or to represent a high level stakeholder need.
- The stereotype enables the modeler to add constraints that restrict the types of model elements that may be assigned to satisfy the requirement.
- E.g. a functional requirement may be constrained so that it can only be satisfied by a SysML behavior such as an activity, state machine, or interaction.

Requirements Extensions




req [package] UserRequirements
Customer Brochure
«trace»
«requirement»
Produce Distilled Water
txt
Distill dirty water into pure water
«refine»
«UseCase»
Distill Water
«rationale »
30 ° is deemed a safe output temperature
«deriveReqt»
«requirement»
Condenser Efficiency
txt
The condenser needs to cool the pure water to 30 °C
«requirement»
Water Output
txt
The pure water output must be at a safe temperature
«rationale»
High pressure water may overpower the boiler so an overflow valve is needed
«requirement»
Handle Overflow
txt
The system must have an overflow valve in the main heating vessel
«deriveReqt»
«requirement»
Water Input
txt
The system must be able to connect directly to mains water
«deriveReqt»
«requirement»
Provide Input Protection
txt
The system shall have an input valve that is closed when the system is off
«rationale»
Direct connection to high pressure mains when the system is off might cause damage
«requirement»
Water Throughput
txt
The system shall handle 1 litre of water per minute
«satisfy»
«activity»
Distill Water
«rationale»
Analysis of this activity shows that the throughput is possible

SysML editors

- TopCase
  - Eclipse Plugin
- Online Visual Paradigm
- Magic Draw

Executable Specifications

- People interact with software to satisfy their desires.
- Their most important desires become requirements (their desirements).
- *User stories* are one of the most efficient techniques when the focus is on identifying stakeholders' desirements.
- A user story is a short description written in natural language that represents a discrete piece of demonstrable functionality.
- *"A user story describes functionality that will be valuable to either a user or purchaser of a system or software."* *

* Mike Cohn, *User Stories Applied: For Agile Software Development*. Boston, MA: Addison-Wesley, 2004.

Executable Specifications -User stories

- A user story is a placeholder containing just enough information so that the stakeholders can rank it and the team can produce a reasonable estimate of the effort it will take to implement it.
- A user story should be free of technical jargon;
- It needs to be comprehensible to both developers and stakeholders.
- It should be written as a one-liner using the classic template:

  As a <**role**>, I want <**desire**> so that <**benefit**>.

  - The desire describes an activity done by a stakeholder.

Example:

As a <student>, I want <to buy a pass valid only on school days> so that I can <go to school>.

Executable Specifications- User stories

- User stories are a quick way of documenting a stakeholder's desirable outcome without being preoccupied by the writing of detailed requirements specification.
- The template for user stories helps to answer the classic "Who," "What," "Why" questions:

  Who = role, What = desire, Why = benefit
- A well-written user story follows the INVEST criteria:
  - ***Independent***: A story should stand alone and be self-contained without depending on other stories.
  - ***Negotiable***: A story is a placeholder that facilitates conversation and negotiation between the team and stakeholders. At any time, the story can be rewritten or even discarded.
  - ***Valuable***: A story needs to deliver value to the stakeholders (either the end user or the purchaser).
  - ***Estimable***: The team needs to be able to roughly estimate the size of the effort to complete the story.
  - ***Small***: A story can start its life as a big placeholder. As time goes by and you better understand it, the placeholder will be split into smaller stories.
  - ***Testable***: A story must provide the necessary information to clearly define the acceptance criteria that confirm the story is completed.

Executable Specifications- User stories

- Examples
  - As a <**student**>, I want <**to buy a pass valid only on school days**> so that I can <**go to school**>.
  - As a <**student**>, I want <**to buy a monthly pass**> so that I can <**go to school and get around**>.
  - As a <**tourist**>, I want <**to buy a daily pass**> so that I can <**visit the city for one day**>.
  - As a <**tourist**>, I want <**to buy a multiple day pass>** so that I can **<visit the city for more than one day**>.
  - As a <**worker**>, I want <**to buy a monthly pass**> so that I can <**go to work**>.

Executable Specifications- Scenarios

- Words have different meanings to different people.
- Relying only on user stories can easily lead to miscommunication between the software development team and the stakeholders.
- We need to enhance the conversation with a more effective practice.
- The solution is to confirm user stories using concrete examples.
- By tying the meanings of words to concrete examples, it is much harder for people to misunderstand.
- Two categories of concrete examples used to illustrate the expected behavior:
  - ***Storyboard***: A rough, even hand-sketched, sequence of drawings that illustrate the important steps of the user experience for the purpose of previsualizing the behavior of a user story.
  - ***Scenario***: A script written in natural language to express a specific behavior within a user story.

Executable Specifications- Scenarios

- Scenarios provide additional information about the user story.
- This additional information says in the words of the stakeholders how they plan to verify the desirable outcome.
- This desirable outcome determines the success criteria.
- Success criteria establish the conditions of acceptation from the stakeholders' point of view.
- Scenarios are the perfect medium for expressing the success criteria.
- The scenario contains a precondition, an action, and a consequence.
  - A *precondition* is the current state of the software before action is taken.
  - An *action* is something that is accomplished to perform the behavior of the scenario.
  - A *consequence* is the result of the action.

Executable Specifications- Scenarios

- Scenarios connect the human concept of cause and effect to the software concept of input-process-output.
- It provides a clear and precise language to help improve communication not only with humans, but with software development tools as well.
- With enough formality, a tool can be written that interprets the intent of these scenarios and tests the software under construction.
- This ensures the specifications work as stated.
- There are two scripting techniques with enough formalism to create a foundation on which you can automate validation.
  - *Framework for Integrated Test (FIT)*. It uses tables and analyzes scenarios, from left to right, within columns.
  - *Given- When-Then syntax*. This technique follows a top-to-bottom method of scripting an action.

Executable Specifications- Scenarios

- Given-When-Then syntax, using the Gherkin grammar, structures the description of the behavior expected by the stakeholders.
- It limits each scenario to only one transition.
- They are easy to maintain by people and simple for tools to parse, because they are formalized using a well-known sequence.
- The Given-When-Then syntax is expressed as follows:

  Given one precondition

  And another precondition

  And yet another precondition

  When an action occurs

  Then a consequence

  And another consequence

Executable Specifications- Scenarios

- *Given*: The necessary preconditions that put the system in a known state. These should define all and no more than the required context.
- *When*: The key action that creates a state transition. There should be only one action per scenario.
- *Then*: The consequences of the action. This is the observable outcome of the scenario.
- *And*: A placeholder that replaces “Given” or “Then” clauses when there are several of them.

Executable Specifications- Scenarios

- User story:

  "As a student, I want to buy a monthly pass so that I go to school and get around."

- Scenario:

  Given *the buyer is a student*
  And *the buyer selects a monthly pass*
  When *the buyer pays*
  Then *a sale occurs with an amount of 76 dollars*

- To express several different values quickly a scenario outline can be used.
- A scenario outline is a template with placeholders. Placeholders must be contained within < > in the scenario steps.

Executable Specifications- Scenarios

- **Given** the buyer is a <buyer_category>
  And the buyer selects a monthly pass
  When the buyer pays
  Then a sale occurs with an amount of <price> dollars

Example:

buyer_category	price
Student	$76
Senior	$98
Standard	$146

Executable Specifications- Scenarios

Given an empty shopping cart is created

And a <fare> is added to shopping cart

When the buyer removes all items from shopping cart

Then the shopping cart is empty

Examples:

fare
Single Ticket
Booklet of 10 Single tickets
1-Day Card
3-Day Card
Weekly Pass
Monthly Pass

Executable Specifications- Scenarios

Given buyer is < buyer_category >

And an empty shopping cart is created

And a <fare> is added to shopping cart

When the buyer pays

Then a <price> sale occurred

Examples:

buyer _category	**fare**	**price**
student	Single Ticket	$1.50
student	Booklet of 10 Single tickets	$9.75
student	1-Day Card	$4.00
student	3-Day Card	$11.00
student	Weekly Pass	$15.00
student	Monthly Pass	$76.00
senior	Single Ticket	$1.75

Executable Specifications- Scenarios

Given-When-Then for queries

Given the buyer is a student
When the buyer requests the list of fares

Then response is

Id	**Name**	**Price**
001	Single Ticket	$1.50
002	Booklet of 10 Single tickets	$9.75
003	1-Day Card	$4.00
004	3-Day Card	$11.00
005	Weekly Pass	$15.00
006	Monthly Pass	$76.00

Executable Specification Tools

- JBehave (Java) (http://jbehave.org)
- JDave (Java) (http://jdave.org/)
- easyb (Java) (http://easyb.org/)
- NBehave (.NET)
- SpecFlow (.NET) (http://www.specflow.org/specflownew/)
- RSpec (Ruby) (http://rspec.info/)
- Cucumber
- etc.

Behaviour-Driven Development (BDD)

- User Story example:
  - **Title**: Customer withdraws cash
  - As a customer,
  - I want to withdraw cash from an ATM,
  - so that I do not have to wait in line at the bank.
- Different scenarios:
  - the account may be in credit,
  - the account may be overdrawn but within the overdraft limit,
  - the account may be overdrawn beyond the overdraft limit,
  - the account is in credit but this withdrawal makes it overdrawn,
  - the dispenser has insufficient cash,
  - etc.

BDD Example

- **Scenario 1**: Account is in credit
  - Given the account is in credit
  - And the card is valid
  - And the dispenser contains cash
  - When the customer requests cash
  - Then ensure the account is debited
  - And ensure cash is dispensed
  - And ensure the card is returned

BDD Example

- **Scenario 2**: Account is overdrawn past the overdraft limit
  - Given the account is overdrawn
  - And the card is valid
  - When the customer requests cash
  - Then ensure a rejection message is displayed
  - And ensure cash is not dispensed
  - And ensure the card is returned

Easyb

- **easyb** is a behavior driven development framework for the Java platform.
- It aims to enable executable, yet readable documentation.
- easyb specifications are written in Groovy and run via a Java runner that can be invoked via the command line, Maven 2, or Ant.

Easyb example

- A customer who wants something to validate zip codes.
  - Given that someone mistypes a zip code
  - And the zip code validation service is up and running
  - When validate is invoked
  - Then the service should indicate the zip code is invalid
- Easyb scenario:

```
given "an invalid zip code", {
     invalidzipcode = "221o1"
   }
   and "given the zipcodevalidator is initialized", {
     zipvalidate = new ZipCodeValidator()
   }
   when "validate is invoked with the invalid zip code", {
     value = zipvalidate.validate(invalidzipcode)
   }
   then "the validator instance should return false", {
     value.shouldBe false
   }
```

Easyb Stories

- Each story is placed in a file ending with `.story`
- Story files have 0..N scenarios in them.
- Scenarios have a mixture of 0..N givens, whens, and thens, with ands linking them:

```
scenario "text", {
 given "text", {}
 when "text", {}
 then "text", {}
}
```

```
scenario "text",{
 given "text", {}
 and
 given "text", {}
 when "text", {}
 then "text", {}
 and  then "text", {}
 and “text”
}
```

Easyb Stories

- The body of any phrase (given, when, then) is optional.
- Given "text" without the trailing , {} implementation code is valid syntax.
- Whens are not required:

```
scenario "text", {
 given "text", {}
 then "text", {}
}
```

- Example:

```
scenario "increasing an empty account", {
    given "an empty account",{
        account = new Account()
    }
    when "1 is added", {
        account.add(BigDecimal.ONE)
    }
    then "the balance should be 1", {
        account.getBalance().shouldBe BigDecimal.ONE
    }
}
```

Easyb Stories

- `description` and `narrative` syntax can be used to capture additional information regarding stories, such as a story's description and some detail regarding the features, benefits, and roles of a person related to a story.
- Description:

```
description "some description“
```

or

```
description """some long description that requires
multiple lines, etc
"""
scenario "text"
```

- Narrative:

```
narrative "regarding currency management", {
 as a "person who uses money"
 i want "to be able to add currencies together"
 so that "that I can become rich over time"
}
```

- Both the narrative and description keywords are optional and they do not have to be used together.

Easyb Stories

- Fixture:
  - `before/after` the code is executed once before/after the entire story
  - `before_each/after_each` the code is executed before/after each scenario

```
before "start selenium", {
 given "selenium is up and running", {
  selenium = new DefaultSelenium("localhost",4444, ...)
  selenium.start()
 }
}
scenario "a valid person has been entered", {
 when "filling out the person form with a first and last name"
 and "the submit link has been clicked"
 then "the report should have a list of races for that person"
}
after "stop selenium" , {
 then "selenium should be shutdown", {
  selenium.stop()
 }
}
```

Easyb Stories

- Shared behaviors: you construct a basic scenario and then refer to it later in other scenarios: `shared_behavior, it_behaves_as`

```
shared_behavior "shared behaviors", {
  given "a string", {
    var = ""
  }
  when "the string is hello world", {
    var = "hello world"
  }
}
scenario "first scenario", {
  it_behaves_as "shared behaviors"
  then "the string should start with hello", {
    var.shouldStartWith "hello"
  }
}
scenario "second scenario", {
  it_behaves_as "shared behaviors"
  then "the string should end with world", {
    var.shouldEndWith "world"
  }
}
```

Easyb Example

Stakeholder: For the next release of our online store, our Gold level customers should receive a discount when they make a purchase.

Developer: What kind of discount? What criteria do they have to meet in order to receive it?

Stakeholder: When they have at least $50 in their shopping cart.

Developer: Does the discount increase based upon the amount or is it fixed regardless of the value of the shopping cart?

Stakeholder: The discount is fixed at 15 percent regardless of price. So, given a Gold level customer, when their shopping cart totals $50 or more, then they should receive a 15 percent discount off the total price.

Easyb Example

- Requirement 1 realization:

```
scenario "Gold level customer with \$50 in shopping cart", {
 given "a Gold level customer"
 when "their shopping cart totals \$50 or more"
 then "they should receive a 15 percent discount off the total price"
}
```

- Requirement 1 implementation

```
scenario "Gold level customer with \$50 in shopping cart", {
 given "a Gold level customer", {
  customer = new Customer()
  customer.goldLevel=true
 }
 when "their shopping cart totals \$50 or more", {
  customer.shoppingCart << new Item("widget", 50.00)
 }
 then "they should receive a 15% discount off the total price" , {
  customer.orderPrice.shouldBe 42.50
 }
}
```

Remark: The scenario is now executable within the context of the application.

Easyb Example

Stakeholder: Our Gold level customers are loving the fact that they're receiving a 15 percent discount on orders of $50 or more. We think it would be a good idea to provide some incentives to non-Gold level customers.

Developer: What kind of incentives?

Stakeholder: When a non-Gold level customer makes a purchase of $100 or more, they should receive a $10 discount; plus, they should receive a coupon for discounts on future purchases.

Developer: The $10 discount remains fixed regardless of the price (provided it is over $100)?

Stakeholder: Yes, it doesn't matter. Our analysis of customer buying habits shows that rarely do non-Gold level customers buy more than $250 worth of merchandise.

Developer: How will these customers receive a coupon?

Stakeholder: The system should e-mail it to them. We already have a coupon service that was implemented a few months ago, so we'll just reuse that.

Developer: OK, so given a non-Gold level customer, when they have $100 or more in their shopping cart at the time of purchase, then they should receive a $10 discount and they should be e-mailed a coupon.

Easyb Example

- Requirement 2 realization (v0):

```
scenario "Non-Gold level customer with \$100 or more" , {
 given "a non-Gold level customer"
 when "they have \$100 or more in their shopping cart"
 then "they should receive a \$10 discount"
 and "they should be emailed a coupon"
}
```

Easyb Example

Developer: When should the e-mail be sent out? We could send it immediately upon successful purchase, or we could send it out within some specified amount of time.

Stakeholder: Ah, we didn't really consider that. Let's send it 24 hours later. That way, maybe the customer will come back and use it. If we send it immediately, it might be forgotten because they just made a purchase.

Developer: And they only get an e-mail if they spend $100 or more?

Stakeholder: Correct.

Developer:The discount — is that applied for that order or is it a $10 discount at some later date?

Stakeholder: It should be applied to the current order price, so they should receive $10 off the total price.

Easyb Example

- Requirement 2 realization (v1):

```
scenario "Non-Gold level customer with \$100 or more" , {
 given "a non-Gold level customer"
 when "they have \$100 or more in their shopping cart"
 then "at the time of checkout, they should receive \$10 off the total
      price"
 and "they should be emailed a coupon within 24 hours"
}
```

Easyb Example

- An edge-case scenario (not mention during conversation)
- Requirement 3

```
scenario "Non-Gold level customer with less than \$100", {
 given "a non-Gold level customer"
 when "they have less than \$100 in their shopping cart"
 then "at the time of checkout, they should not receive any discounts"
 and "they should not be emailed a coupon"
}
```

SpecFlow example

Acceptance Test

```
[Binding]
public class ChecksOutPassTest
{
  [Given (@"an empty shopping cart is created")]
  public void AnEmptyShoppingCartIsCreated()
  {
  }

  [Given (@"a (.*) is added to shopping cart")]
  public void APassIsAddedToShoppingCart(string pass)
  {
  }

  [When (@"the buyer checks out the shopping cart")]
  public void TheBuyerChecksOutTheShoppingCart()
  {
  }

  [Then (@"a (.*) dollar sale occurs")]
  public void ANumberDollarSaleOccurs(int number)
  {
  }
}
```

1 Embed the scenario in a SpecFlow class

2 Given-When-Then steps