Models and Model-driven Development

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

D3S

Software Engineering for Dependable Systems

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Tomas Bures

bures@d3s.mff.cuni.cz

Models

System modeling

- System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

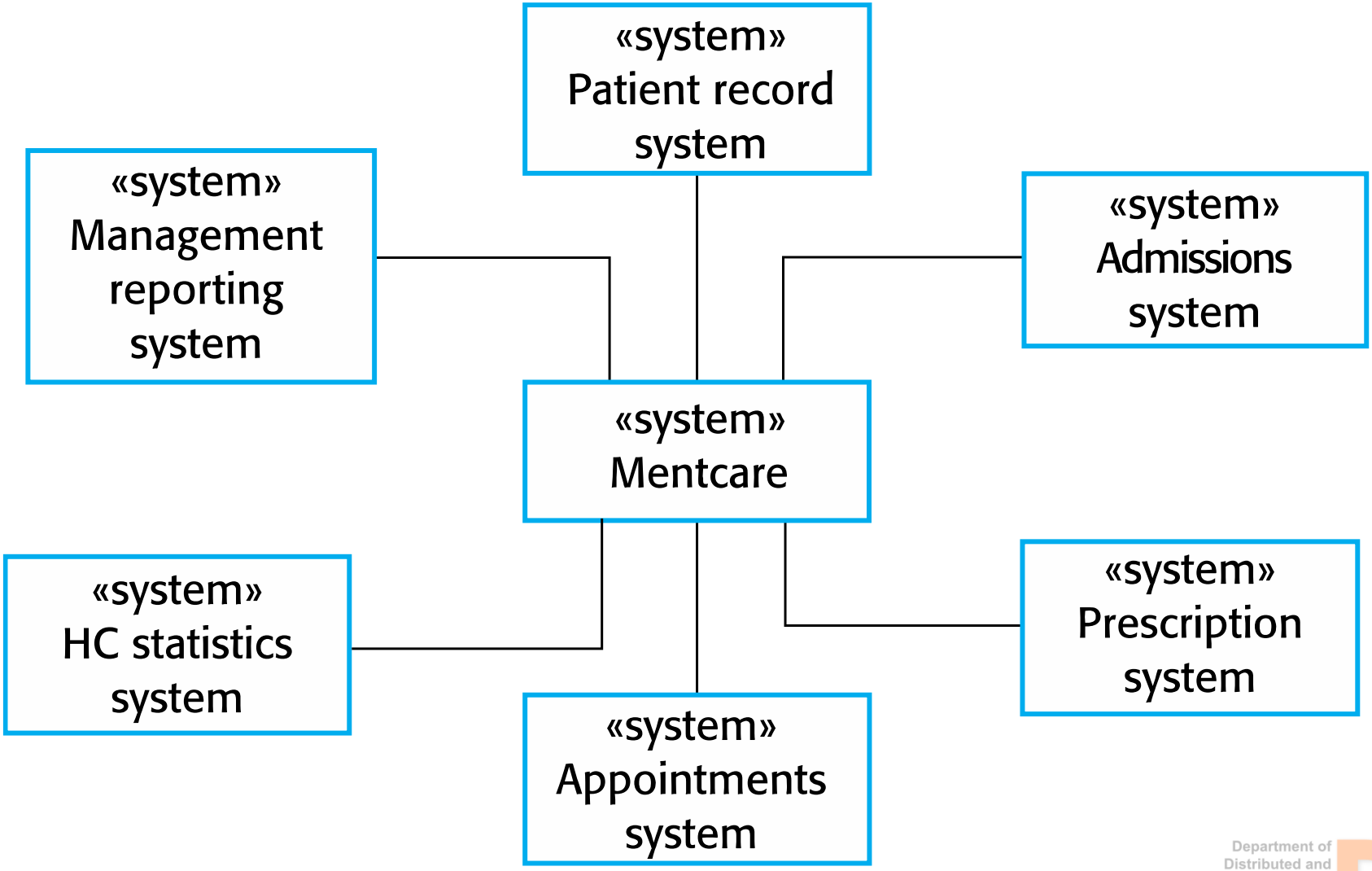
System perspectives

- External perspective
  - models the context or environment of the system
- Interaction perspective
  - models the interactions between a system and its environment, or between the components of a system
- Structural perspective
  - models the organization of a system or the structure of the data that is processed by the system
- Behavioral perspective
  - models the dynamic behavior of the system and how it responds to events

External perspective

- Context models illustrate the operational context of a system - they show what lies outside the system boundaries.
- System boundaries are established to define what is inside and what is outside the system.
  - They show other systems that are used or depend on the system being developed.

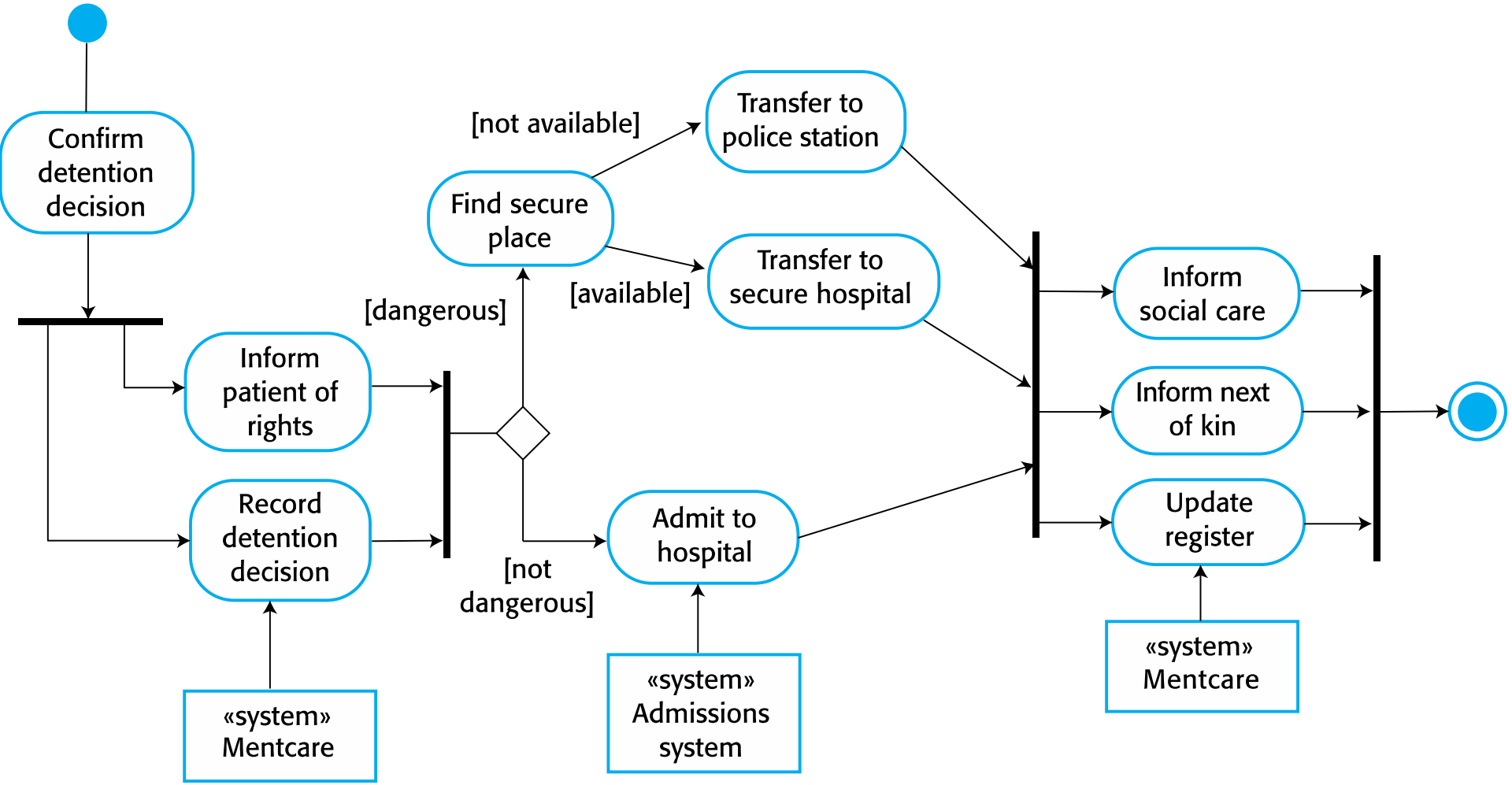
Example: Context model



Process models

- Context models simply show the other systems in the environment, not how the system being developed is used in that environment.
- Process models reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.

Example: Process model



Interaction perspective

- Modeling user interaction is important as it helps to identify user requirements.
- Modeling system-to-system interaction highlights the communication problems that may arise.
- Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- Use case diagrams and sequence diagrams may be used for interaction modeling.

Use case modeling

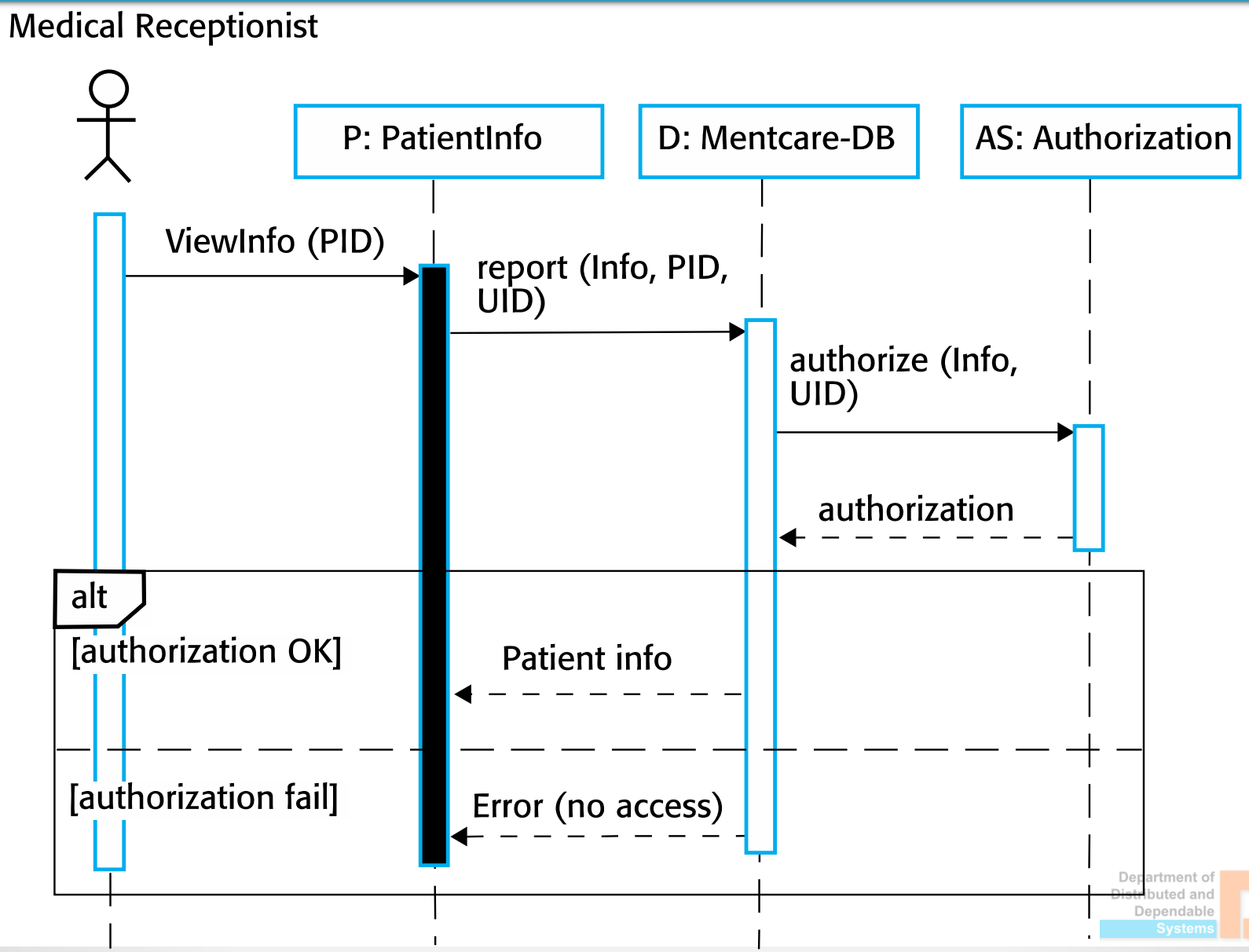
- Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Represented diagramatically to provide an overview of the use case and in a more detailed textual form.

Example: Use-case model

Sequence diagrams

- Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.

Example: System sequence diagram



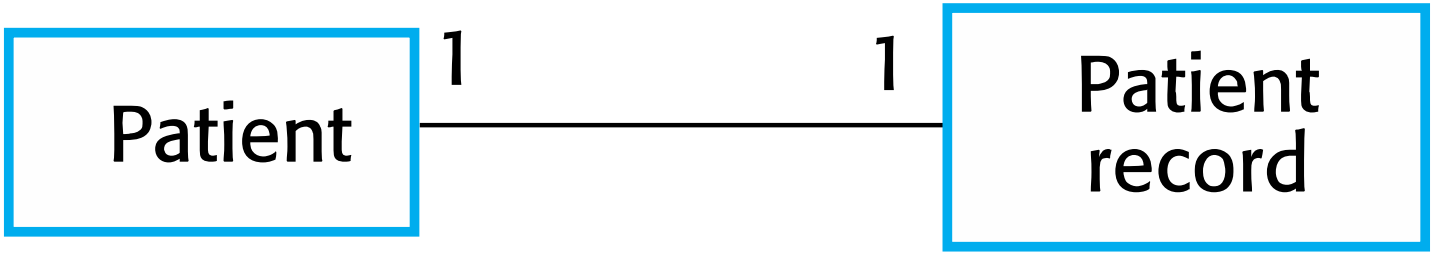
Structural perspective

- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.

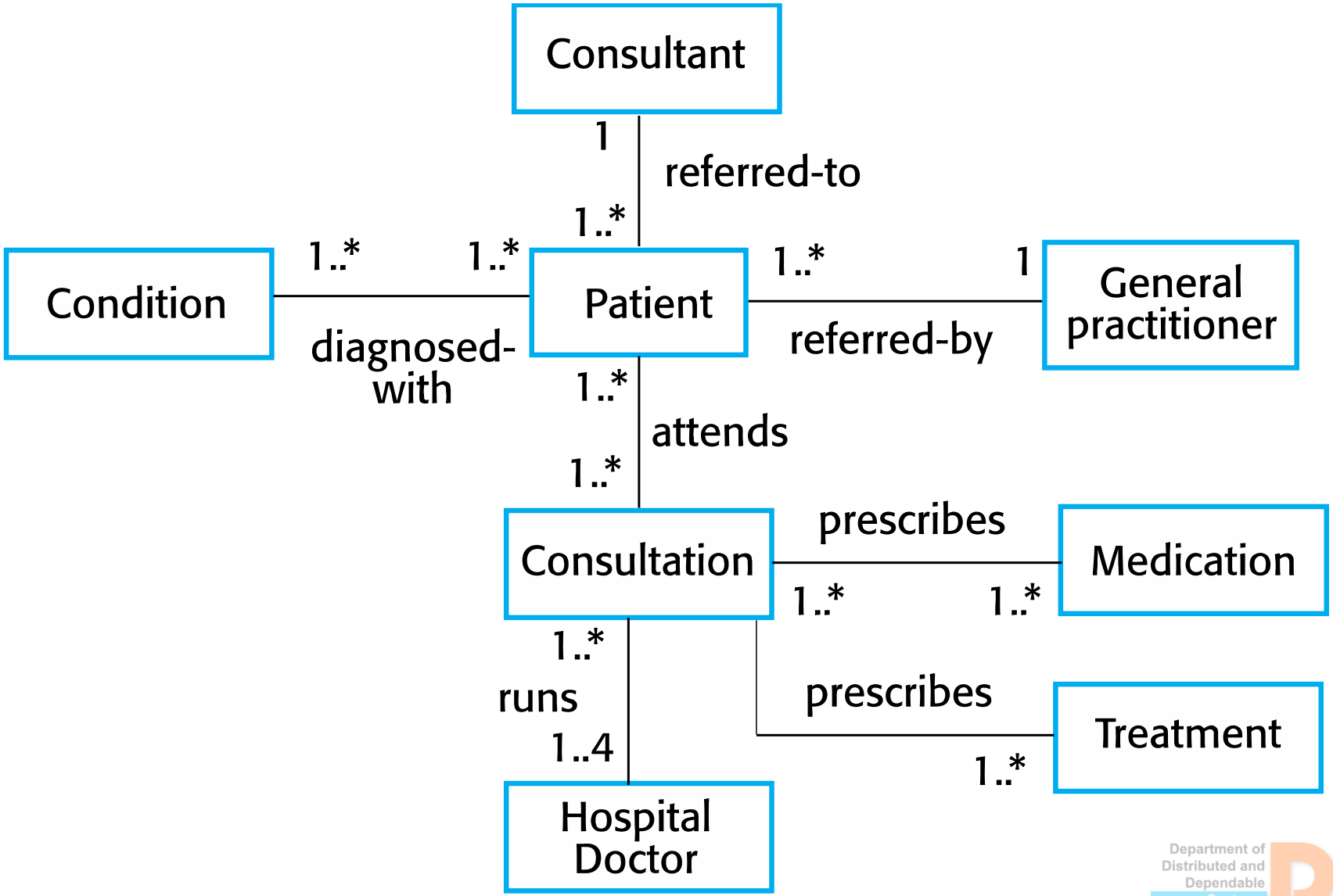
Class diagrams

- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.
- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

Class diagram – Association

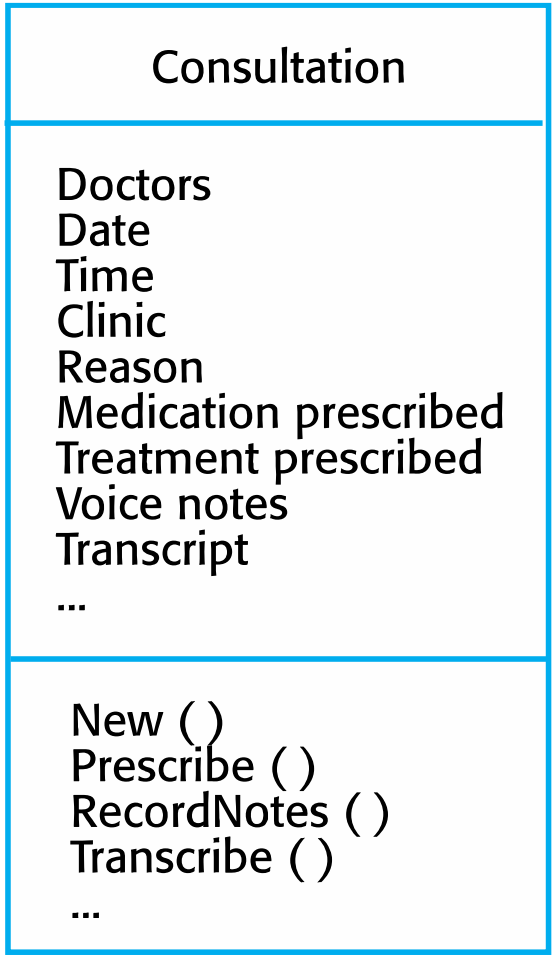


Class diagram – Associations

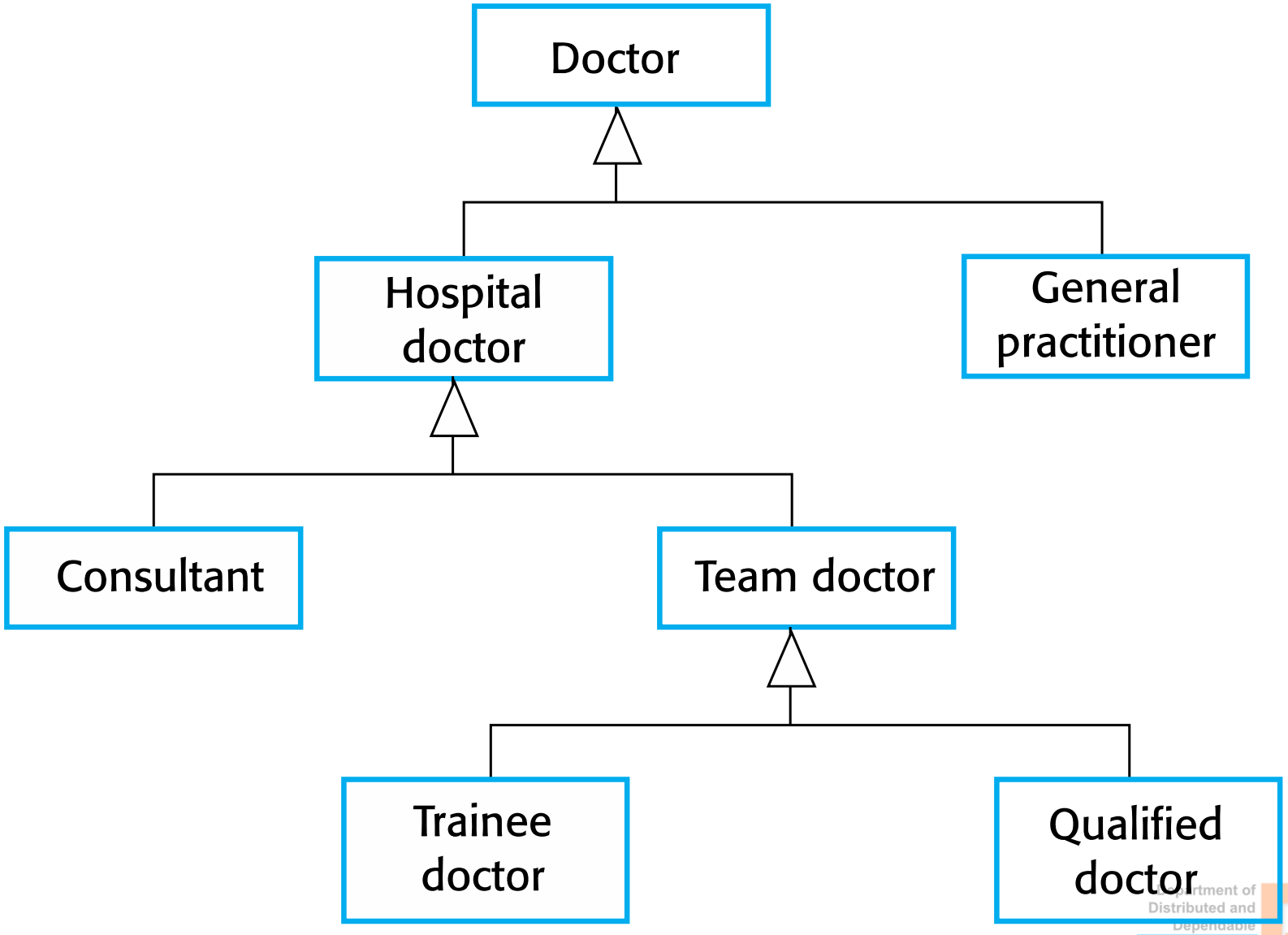


Department of Distributed and Dependable Systems

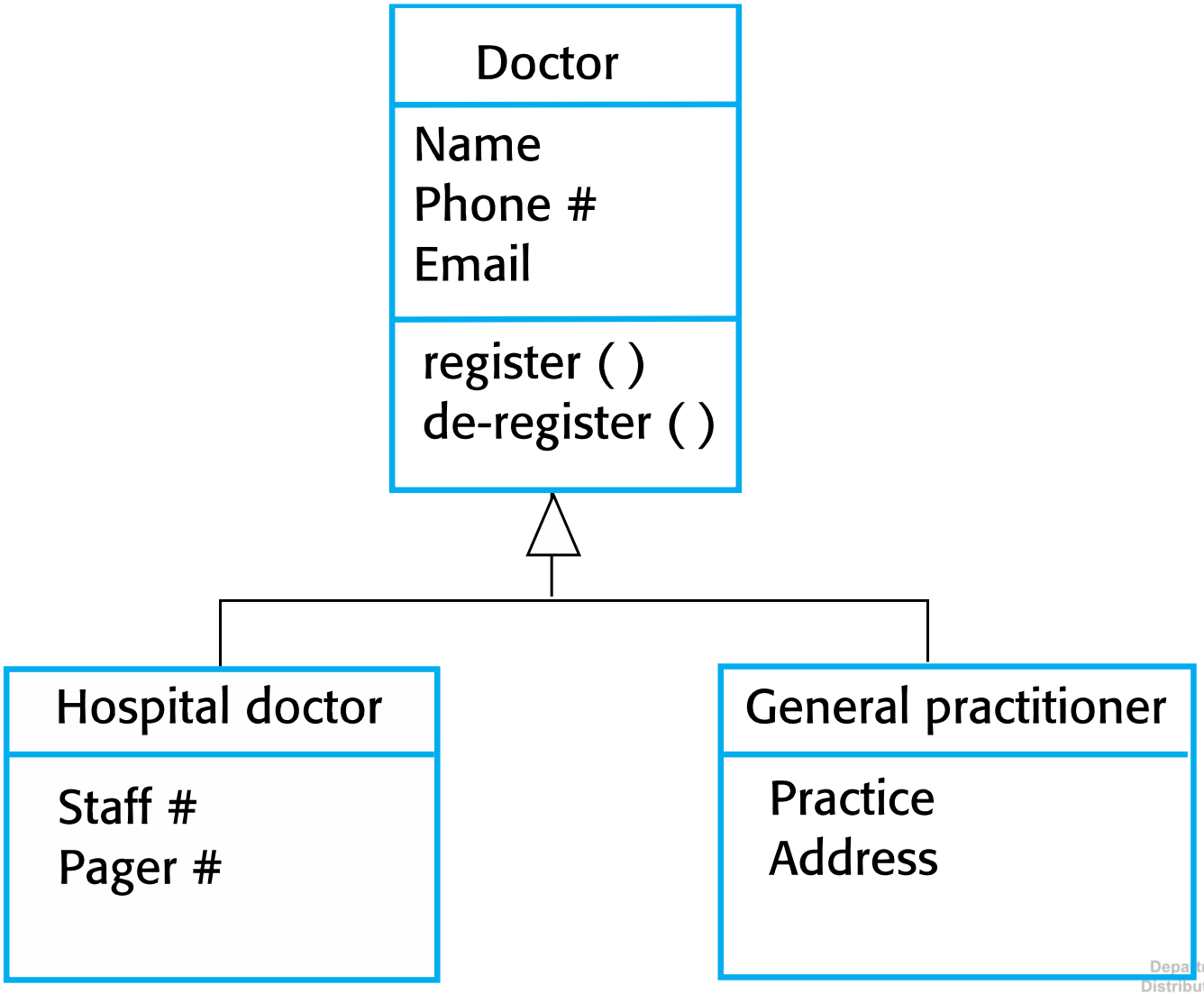
Class diagram – class compartments



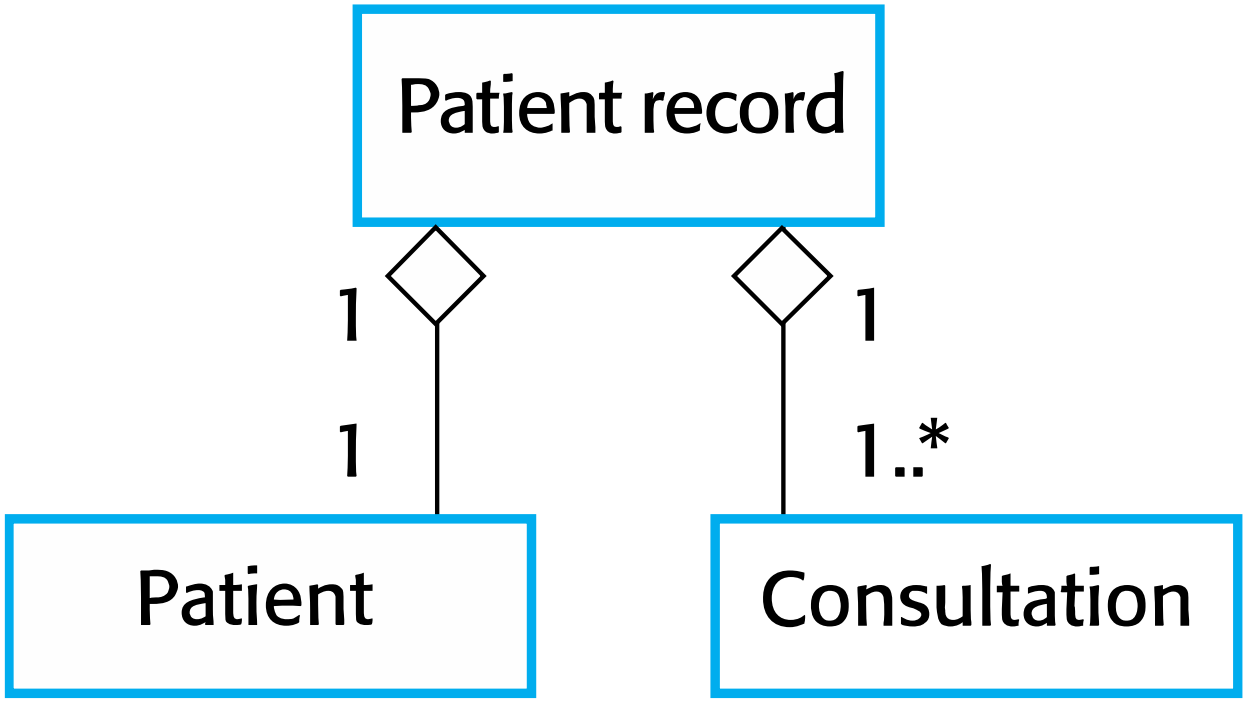
Class diagram – Generalization



Class diagram – Inheritance



Class diagram – Aggregation



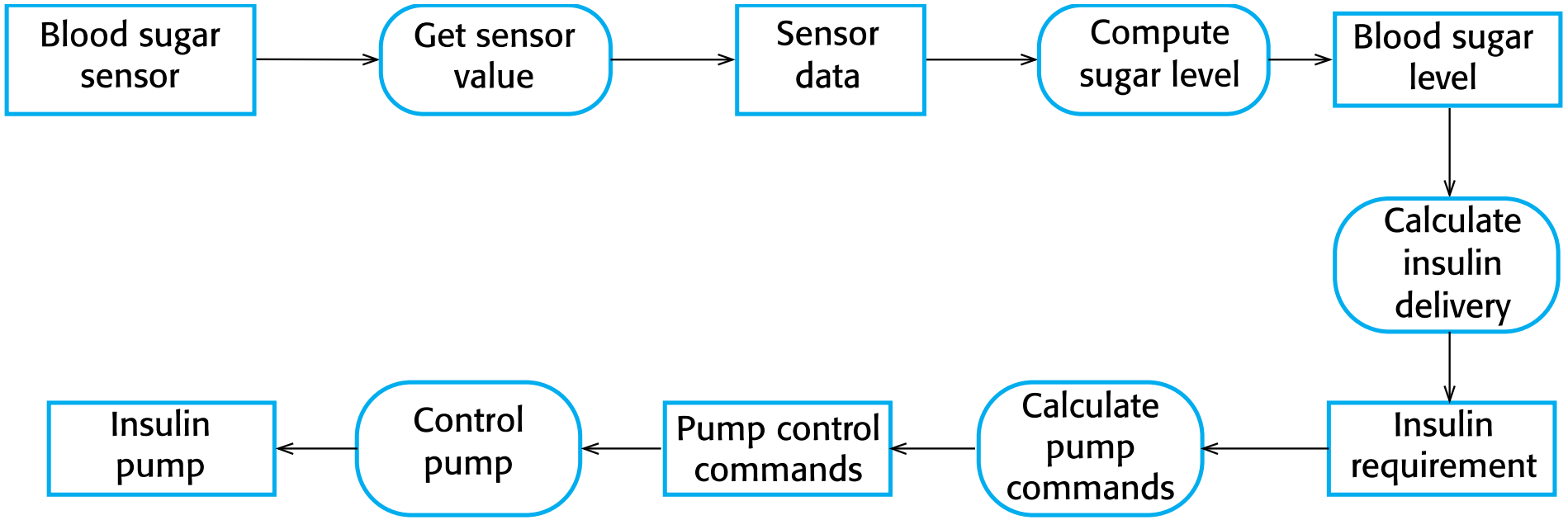
Behavioral perspective

- Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- You can think of these stimuli as being of two types:
  - Data Some data arrives that has to be processed by the system.
  - Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

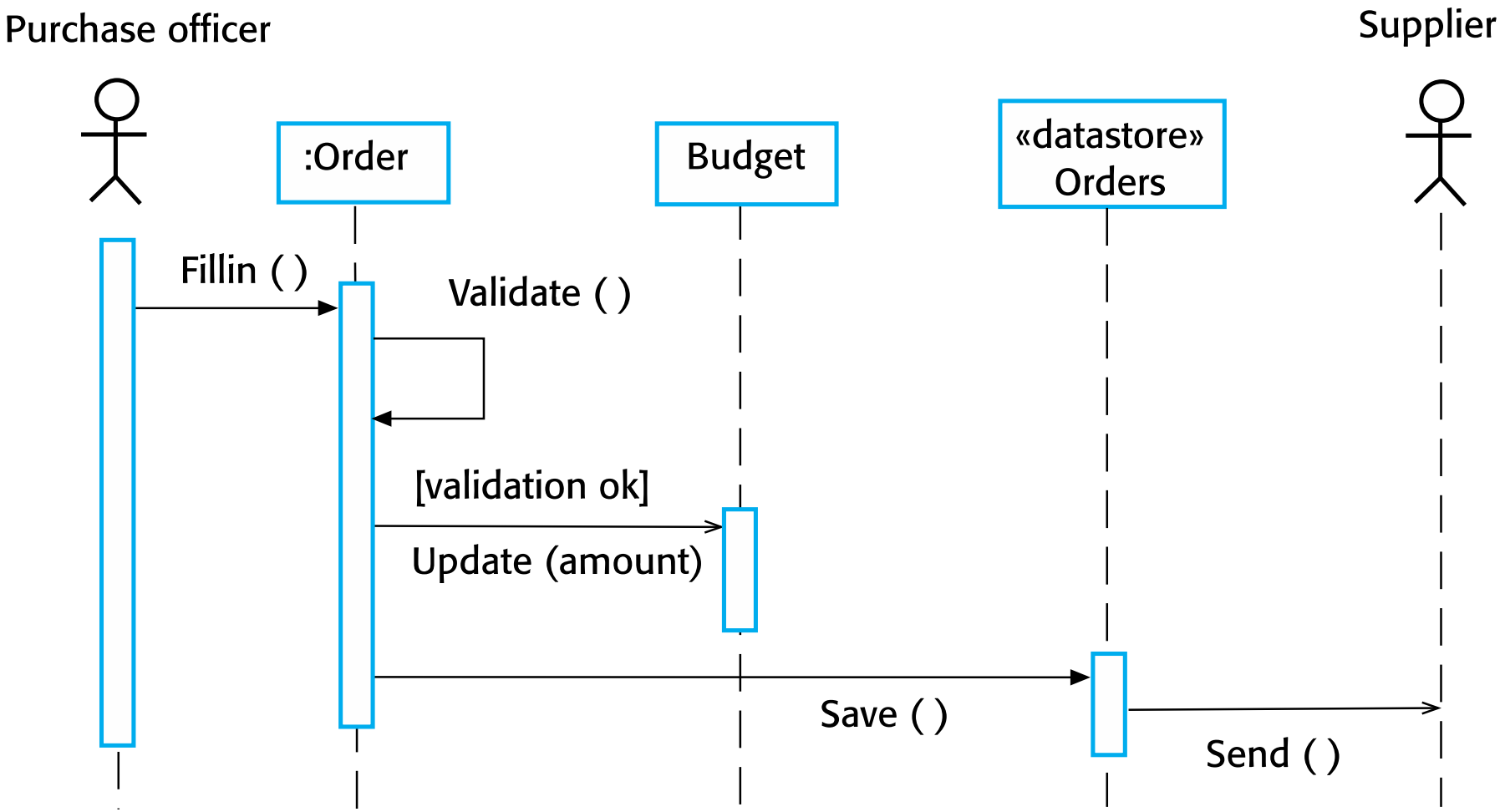
Data-driven modeling

- Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

Example: Activity model



Example: Sequence diagram



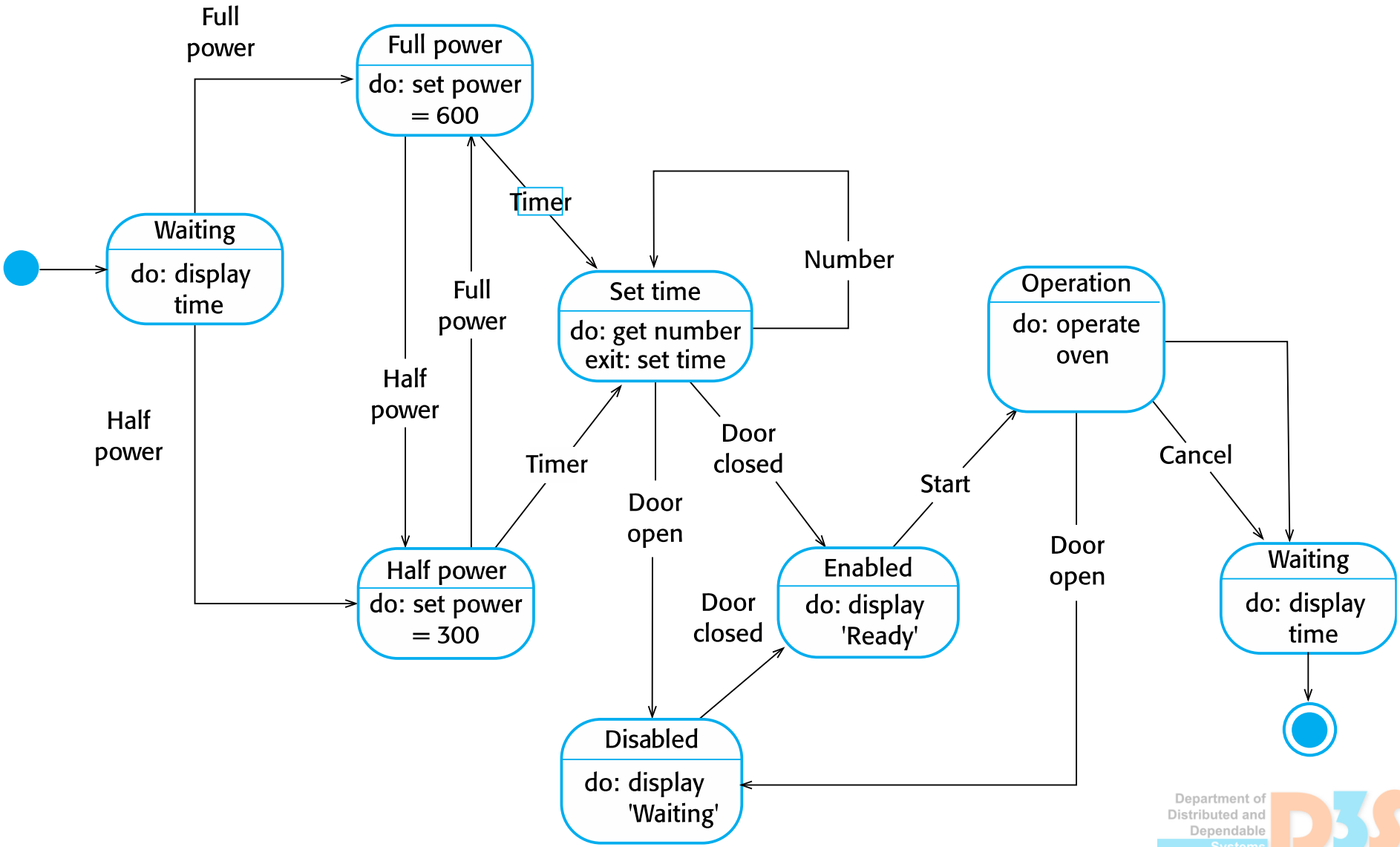
Event-driven modeling

- Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

Example: State diagram



Example: State diagram

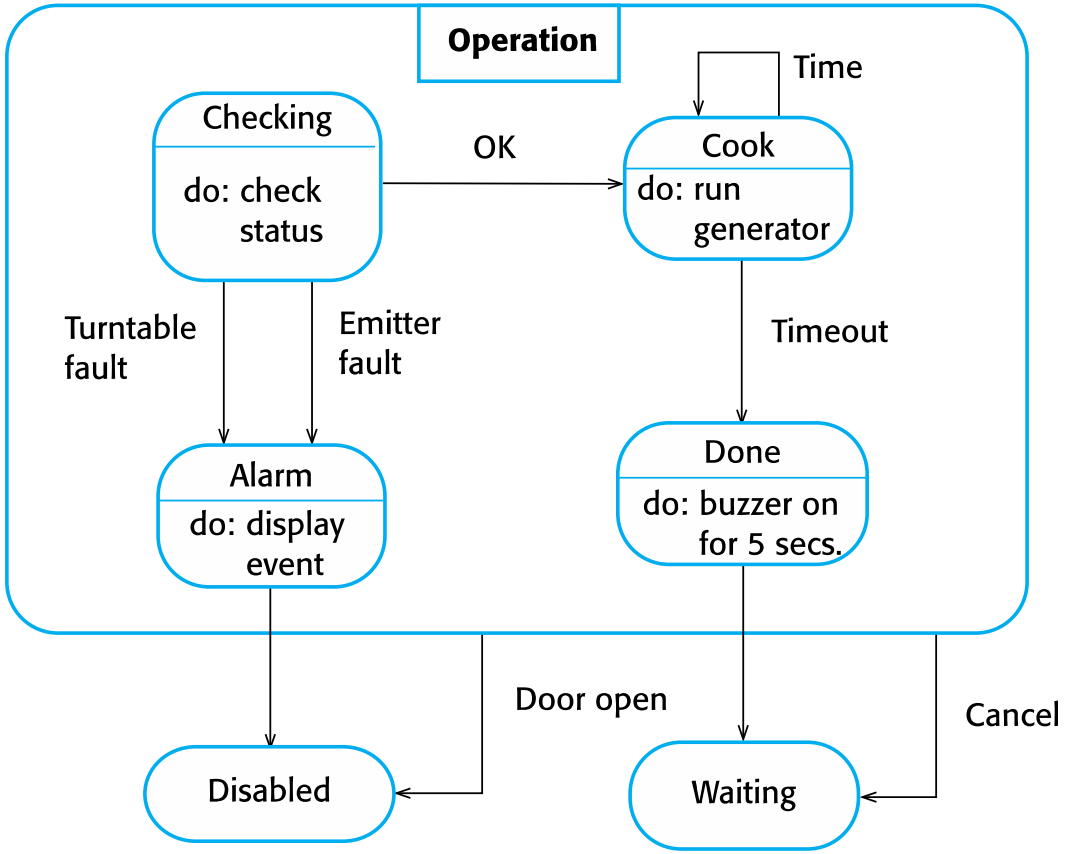


Table of states

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows ‘Half power’.
Full power	The oven power is set to 600 watts. The display shows ‘Full power’.
Set time	The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

Table of stimuli

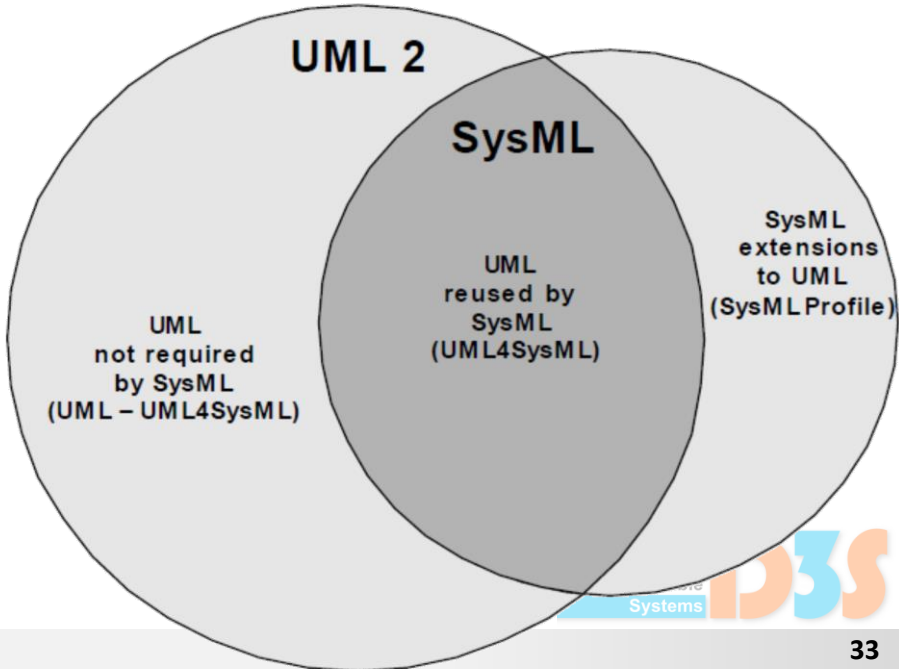
Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Systems Modeling Language (SysML)

SysML

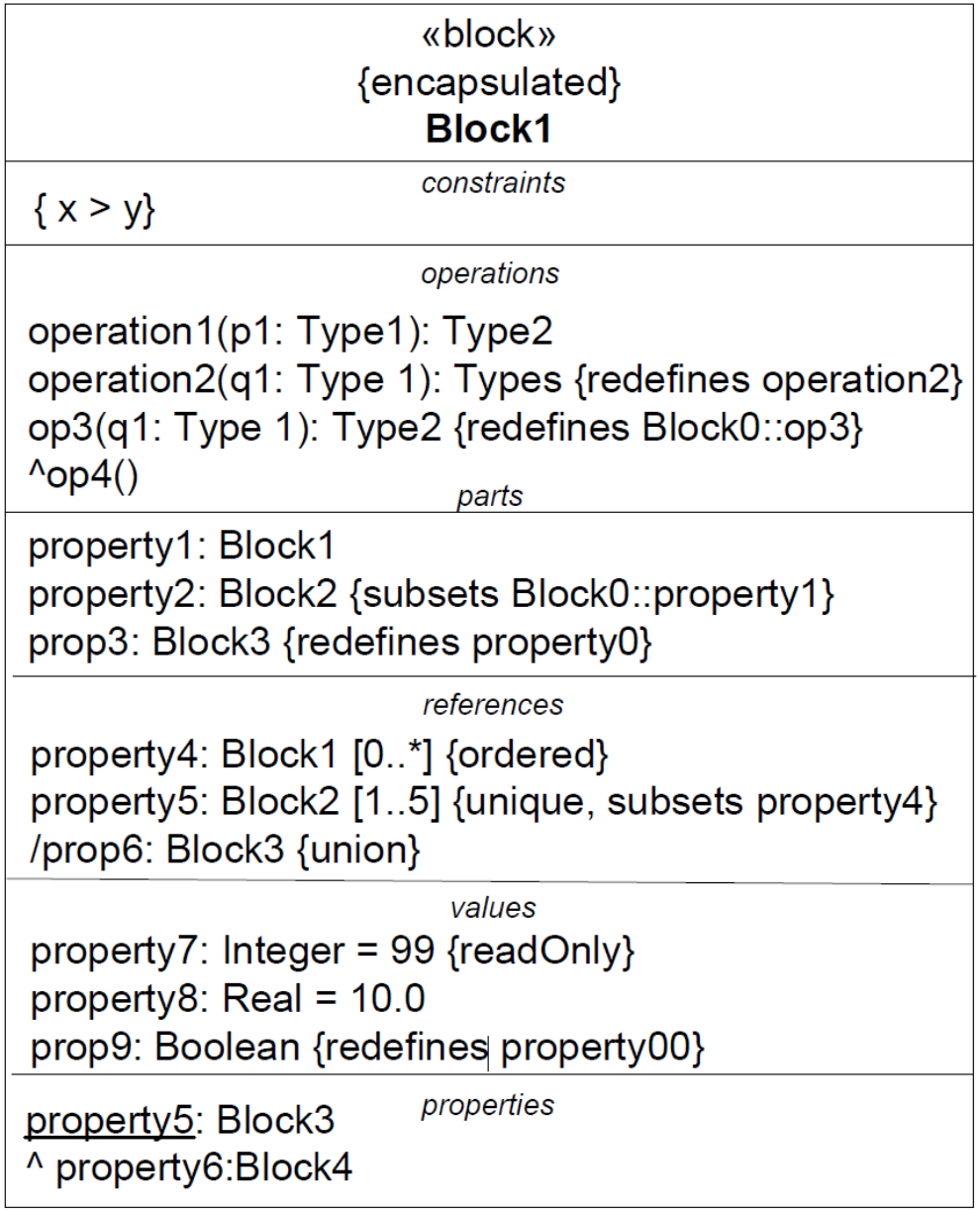
- General-purpose modeling language for systems engineering applications
- Developed by OMG
- Based on UML (extension and a subset)

- Currently in version 1.4

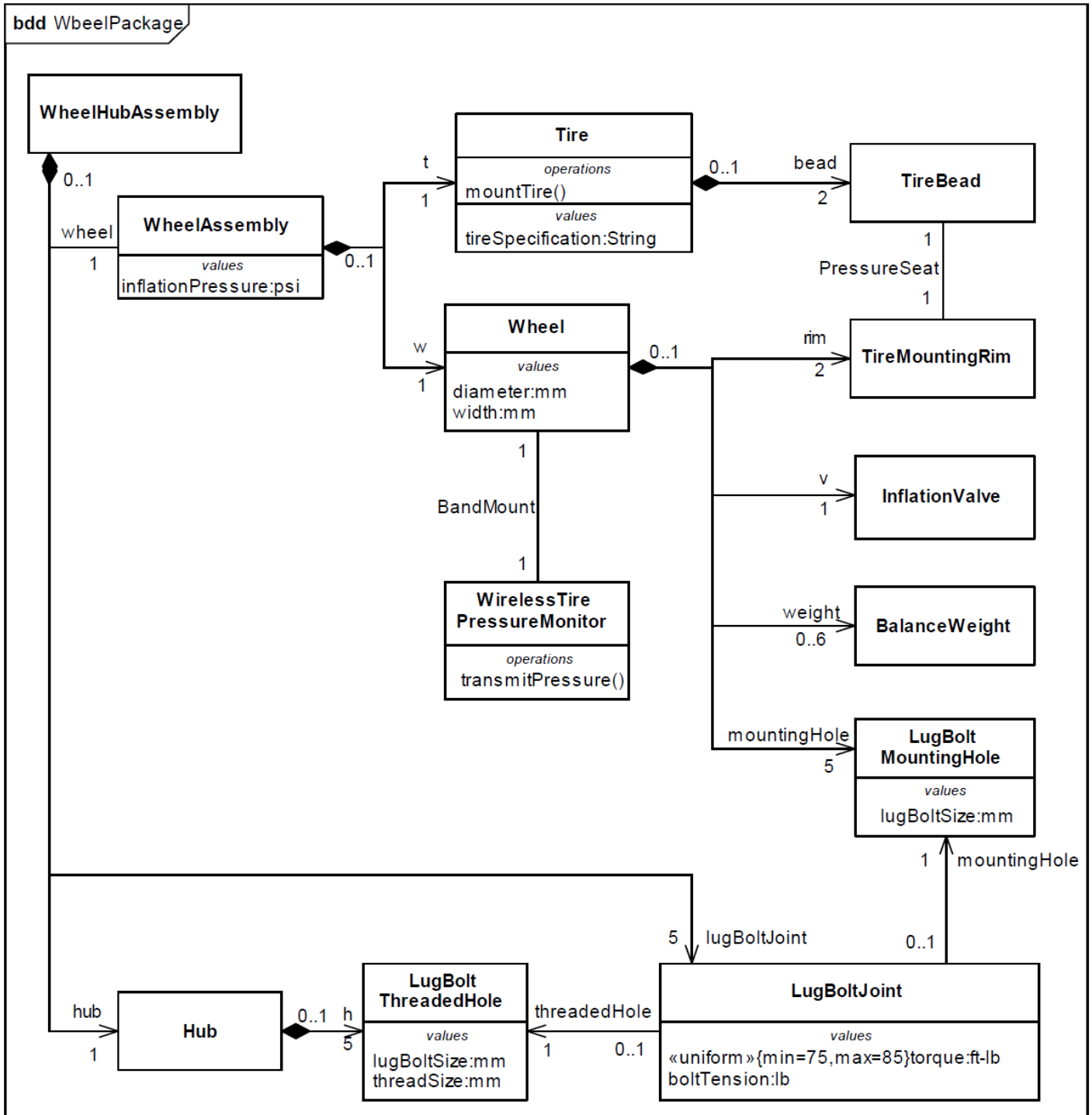


Blocks

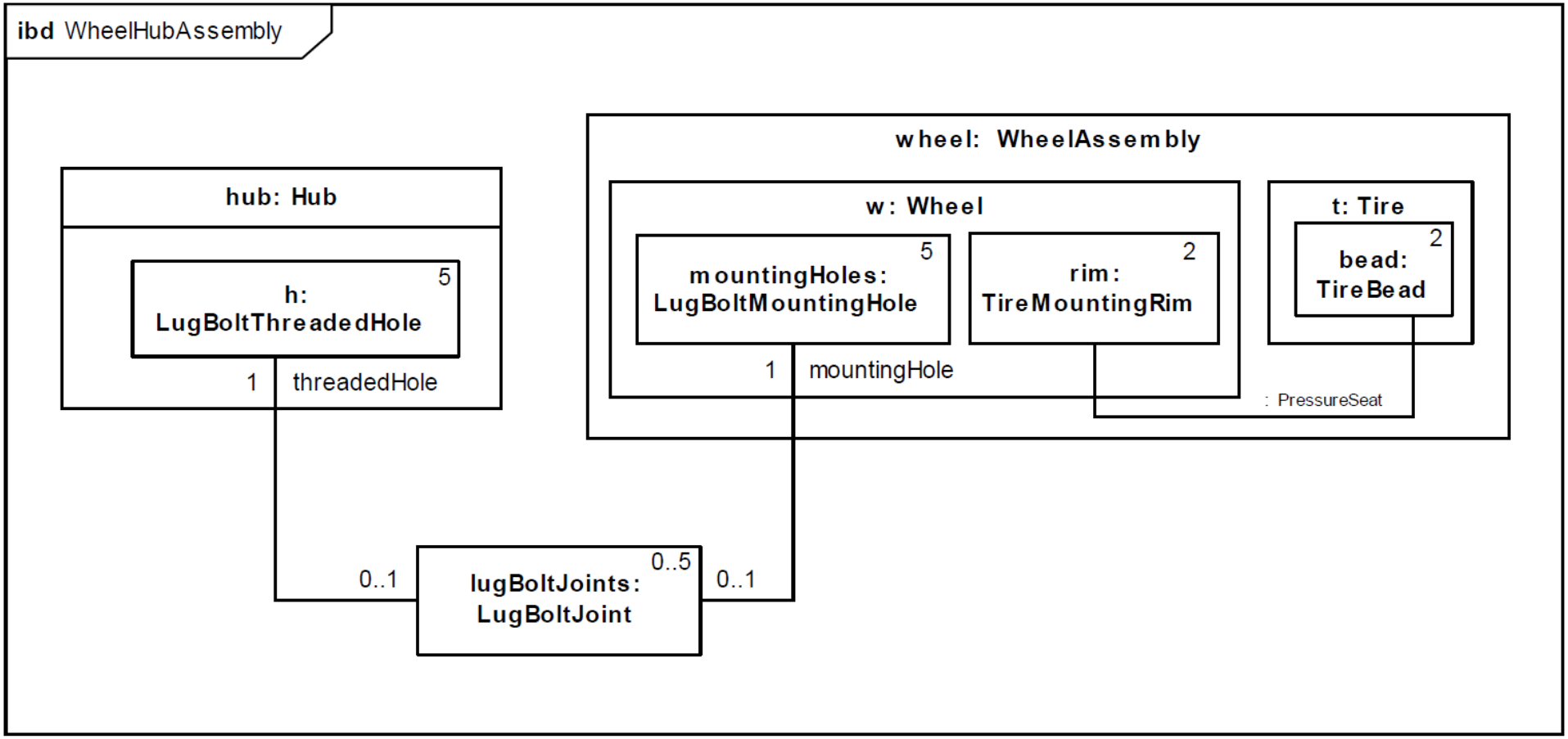
- Block is a basic building block of structural models
- Replaces UML “class”, which has too software-ish flavor
- Specified in Block definition diagrams (BDDs) and Internal block diagrams (IBDs)



Example: Block definition diagram



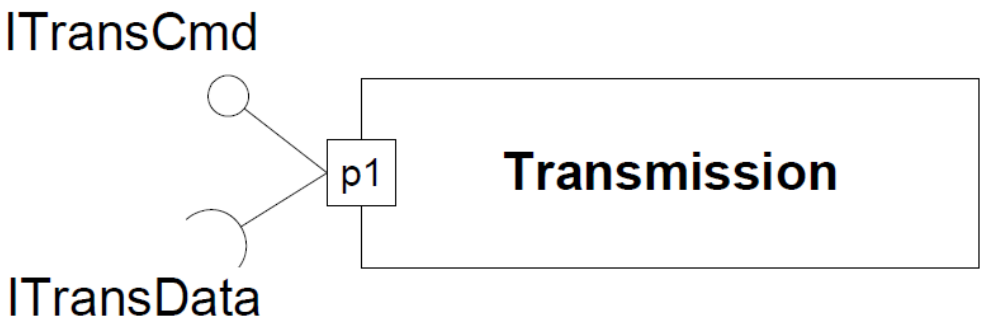
Example: Internal block diagram

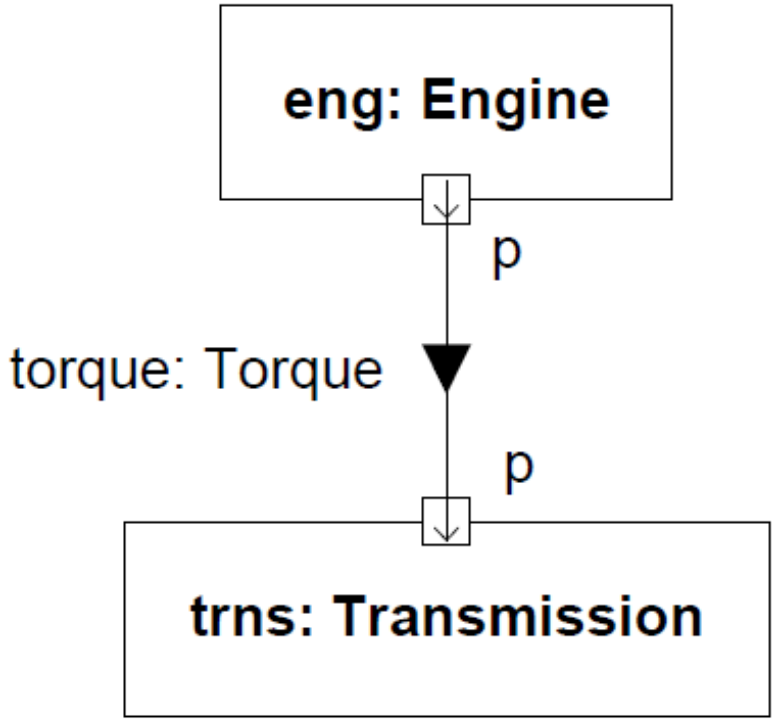


Ports

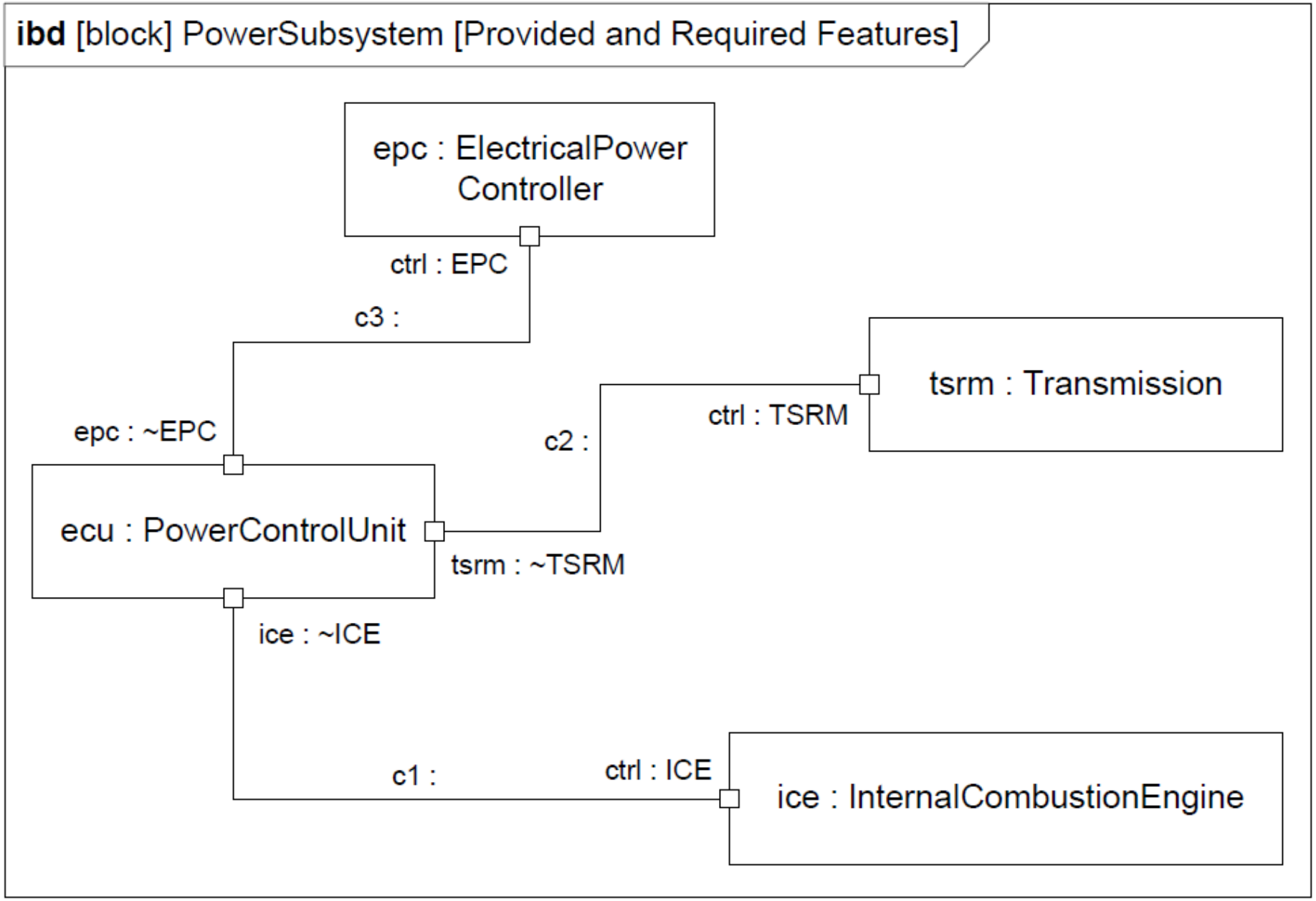
- Blocks can have defined ports
- Ports expose only a certain block properties or internal structure
- A port may consist of multiple ports
- A port may have:
  - Client-server style operations
  - Flow properties (with specified direction)

Examples: Ports

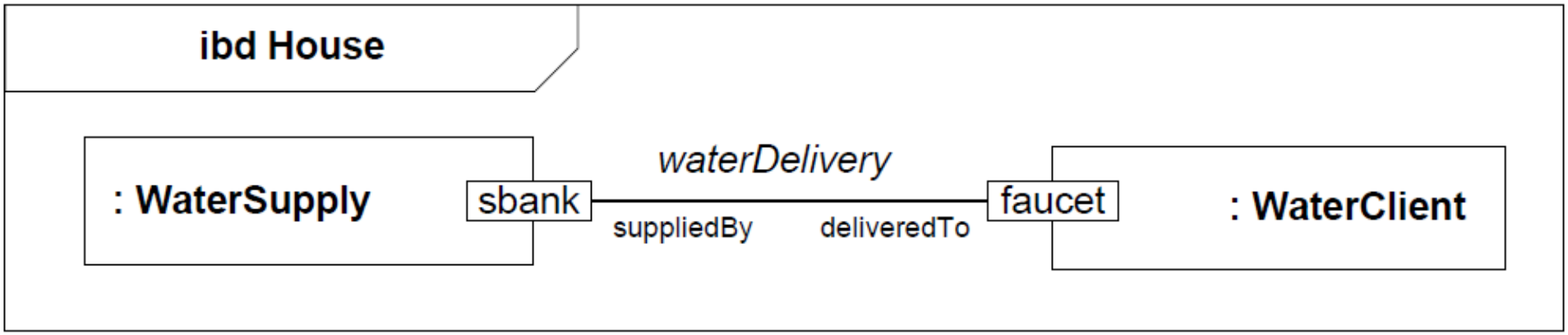


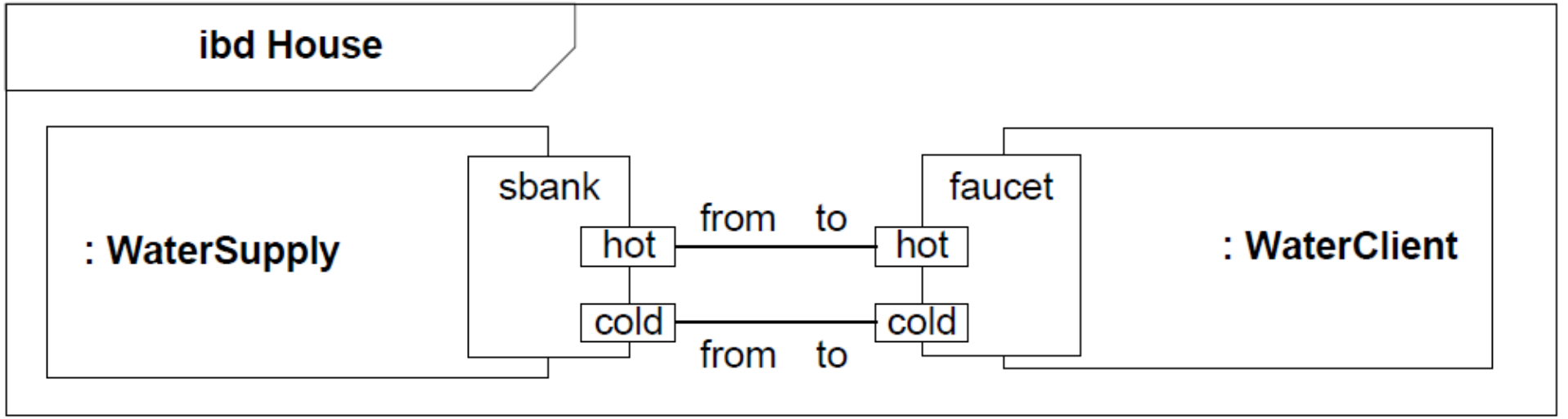


Example: Ports



Example: Ports



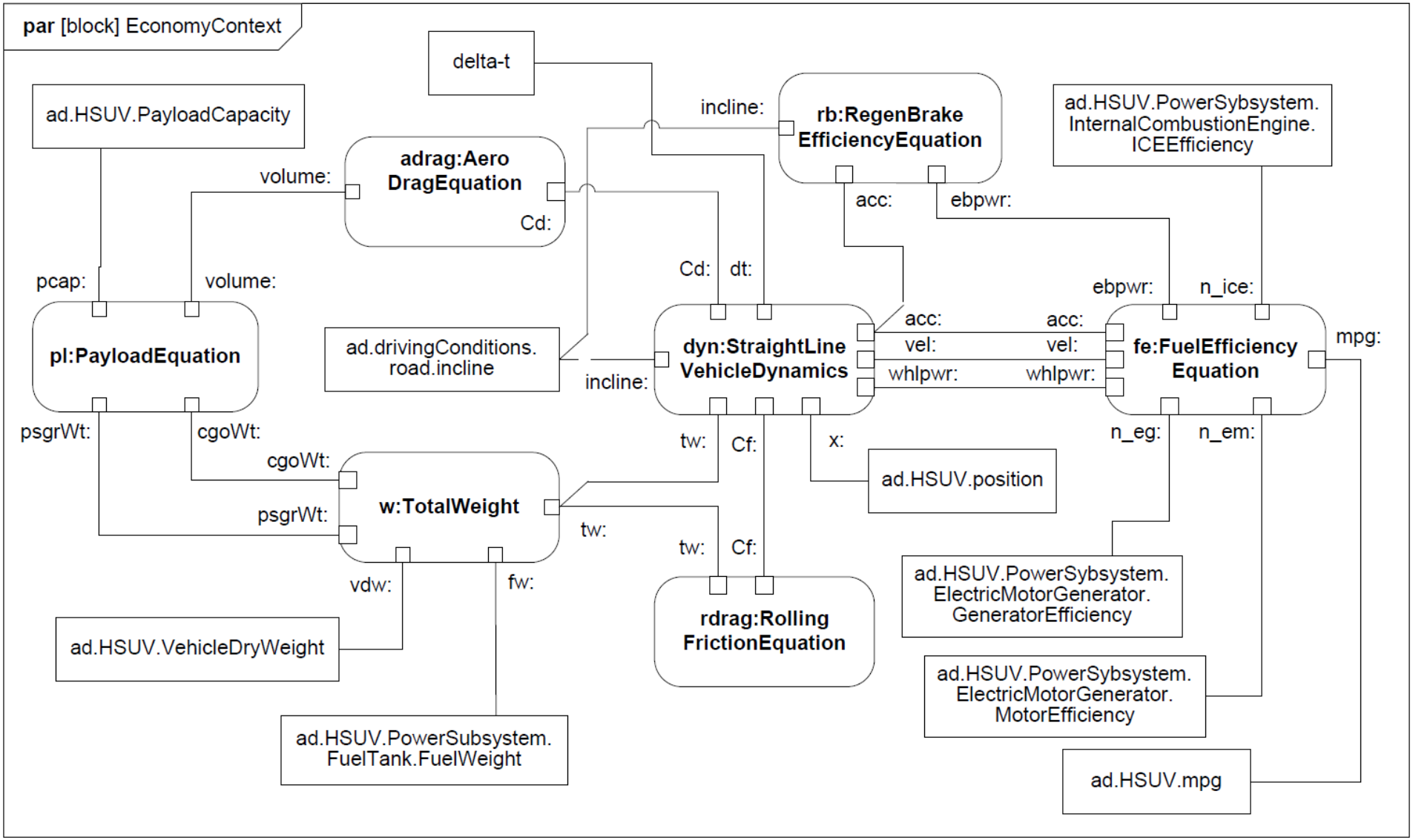


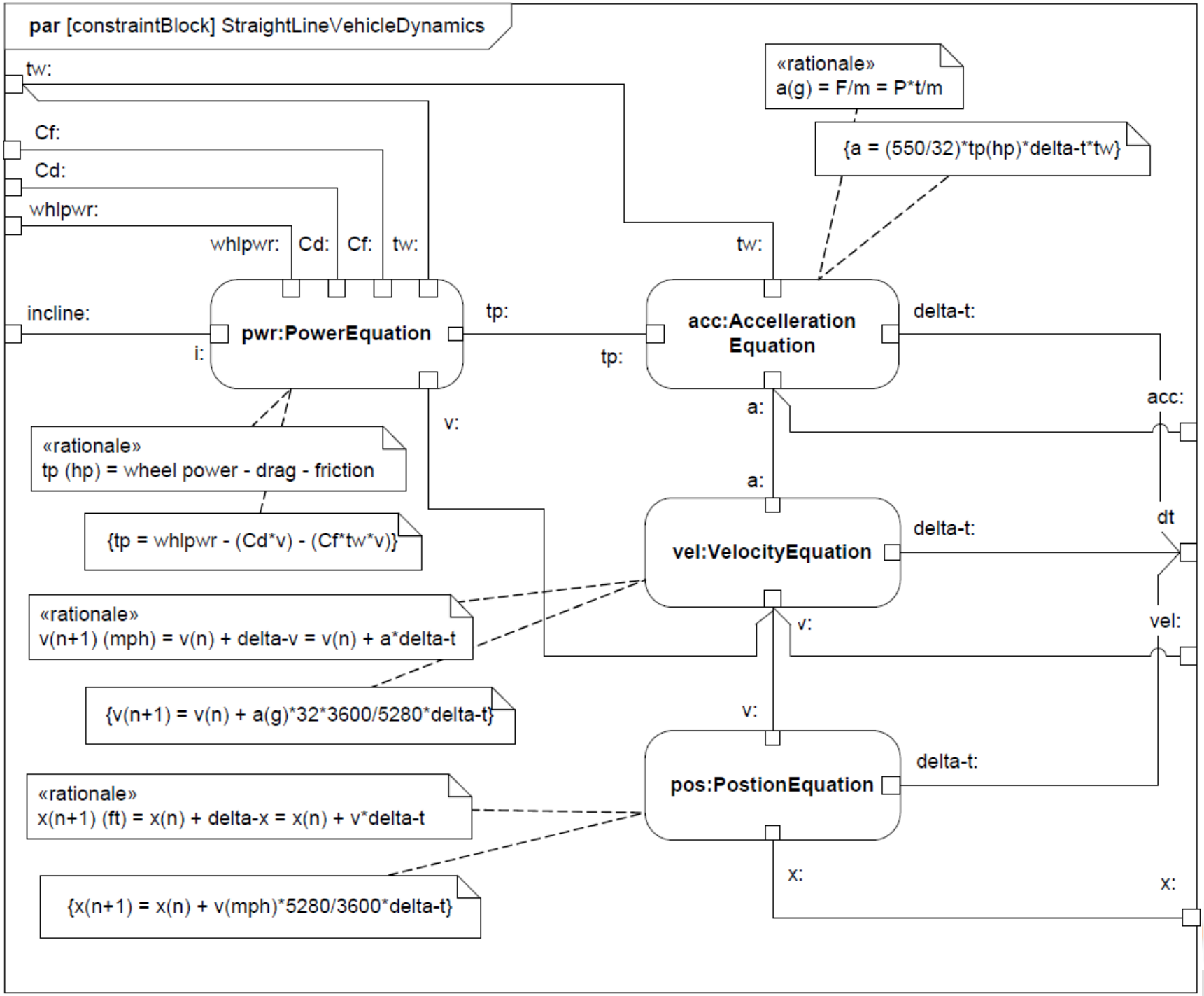
Constraint blocks

- Specify constraints over block properties
- Provide mechanism for integrating engineering analysis such as performance and reliability analysis with other SysML models

- Defined in Parametric diagrams

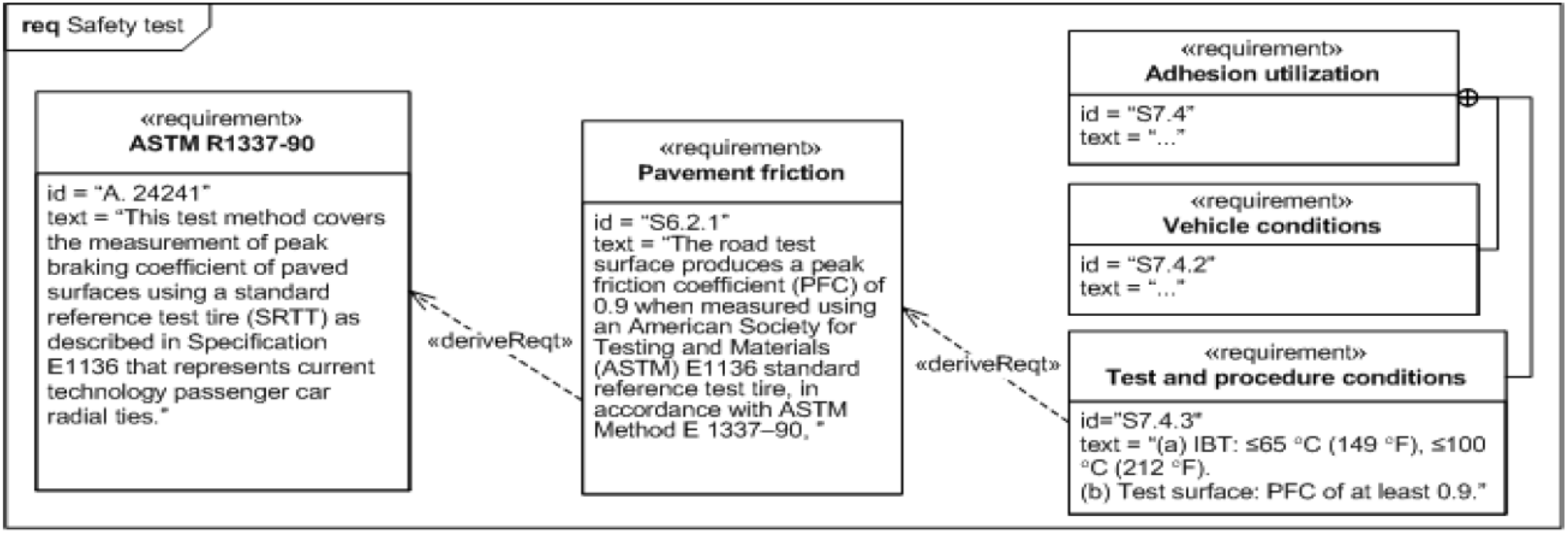
Example: Parametric diagram

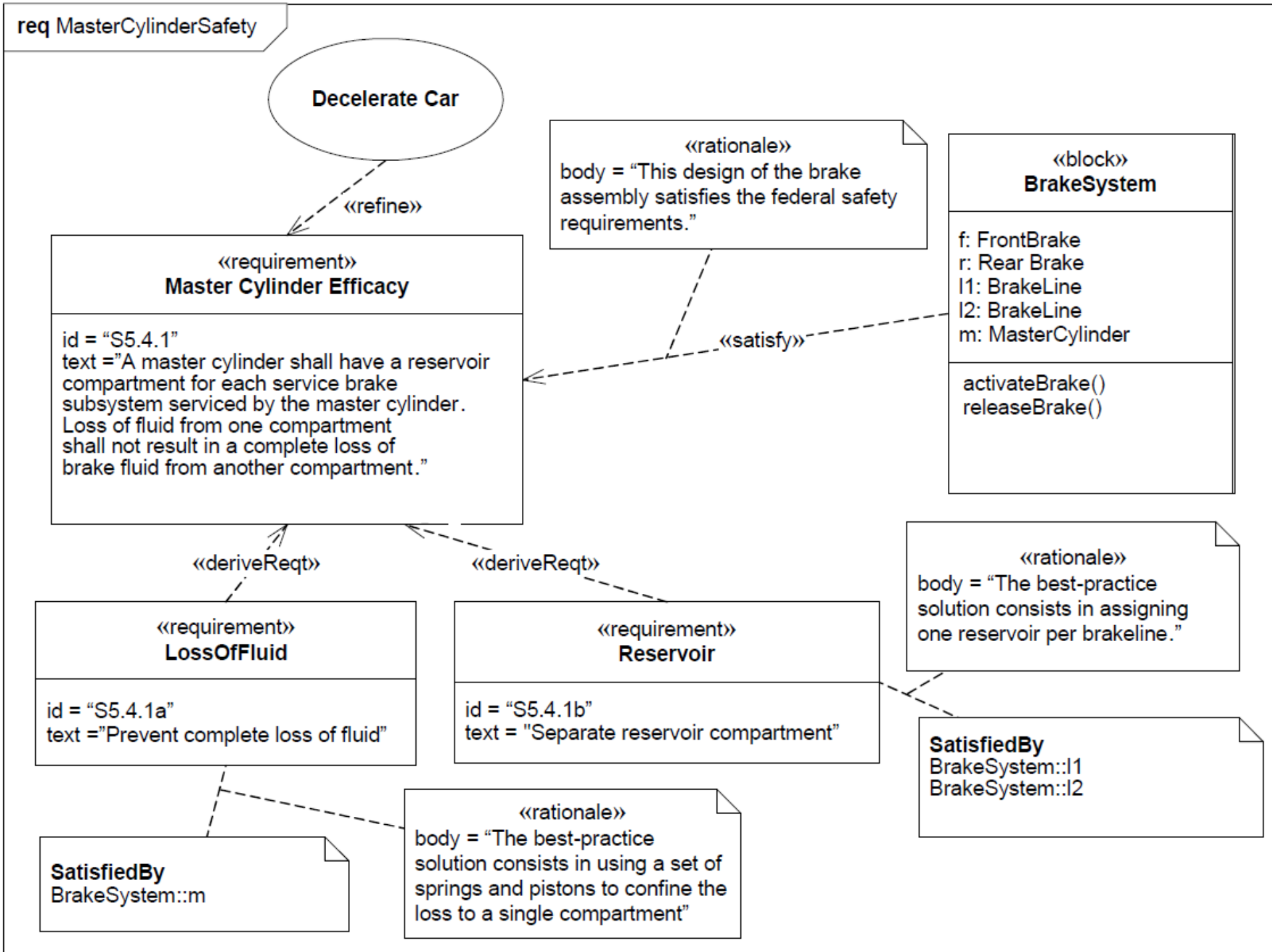



par [constraintBlock] StraightLineVehicleDynamics
tw:
Cf:
Cd:
whlpwr:
whlpwr:
Cd:
Cf:
tw:
incline:
i:
pwr:PowerEquation
tp:
tp:
tw:
acc:Accelleration Equation
delta-t:
acc:
a:
v:
«rationale»
tp (hp) = wheel power - drag - friction
{tp = whlpwr - (Cd*v) - (Cf*tw*v)}
a:
vel:VelocityEquation
delta-t:
dt
«rationale»
v(n+1) (mph) = v(n) + delta-v = v(n) + a*delta-t
{v(n+1) = v(n) + a(g)*32*3600/5280*delta-t}
v:
vel:
v:
pos:PostionEquation
delta-t:
«rationale»
x(n+1) (ft) = x(n) + delta-x = x(n) + v*delta-t
{x(n+1) = x(n) + v(mph)*5280/3600*delta-t}
x:
x:
«rationale»
a(g) = F/m = P*t/m
{a = (550/32)*tp(hp)*delta-t*tw}

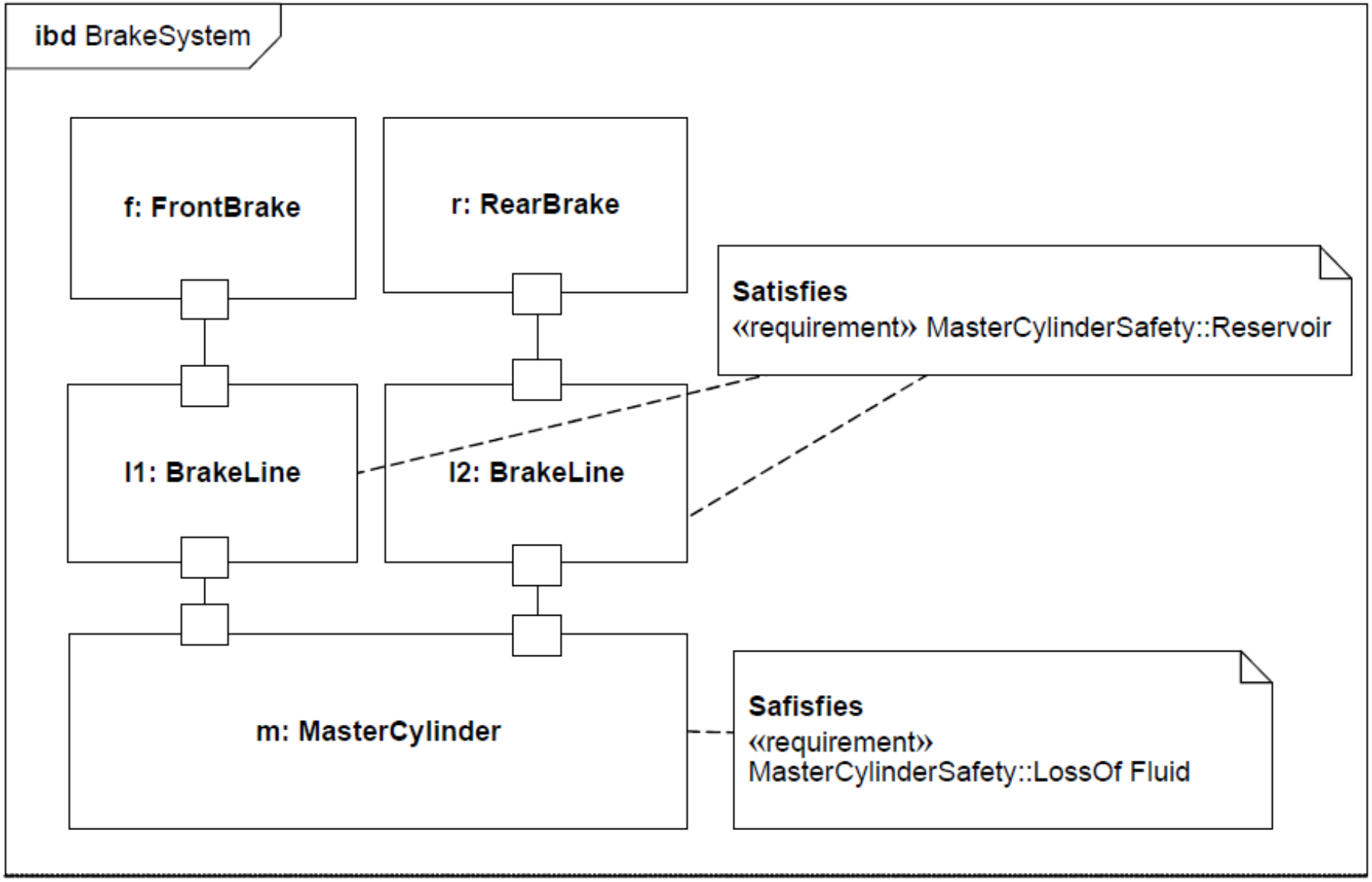
Requirements

- Block to represent requirements
- Organized in Requirements diagrams
- Tabular form also possible




req MasterCylinderSafety
Decelerate Car
«refine»
«rationale»
body = "This design of the brake assembly satisfies the federal safety requirements."
«block»
BrakeSystem
f: FrontBrake
r: Rear Brake
l1: BrakeLine
l2: BrakeLine
m: MasterCylinder
activateBrake()
releaseBrake()
«requirement»
Master Cylinder Efficacy
id = "S5.4.1"
text ="A master cylinder shall have a reservoir compartment for each service brake subsystem serviced by the master cylinder. Loss of fluid from one compartment shall not result in a complete loss of brake fluid from another compartment."
«satisfy»
«deriveReqt»
«deriveReqt»
«requirement»
LossOfFluid
id = "S5.4.1a"
text ="Prevent complete loss of fluid"
«requirement»
Reservoir
id = "S5.4.1b"
text = "Separate reservoir compartment"
«rationale»
body = "The best-practice solution consists in assigning one reservoir per brakeline."
SatisfiedBy
BrakeSystem::l1
BrakeSystem::l2
SatisfiedBy
BrakeSystem::m
«rationale»
body = "The best-practice solution consists in using a set of springs and pistons to confine the loss to a single compartment"

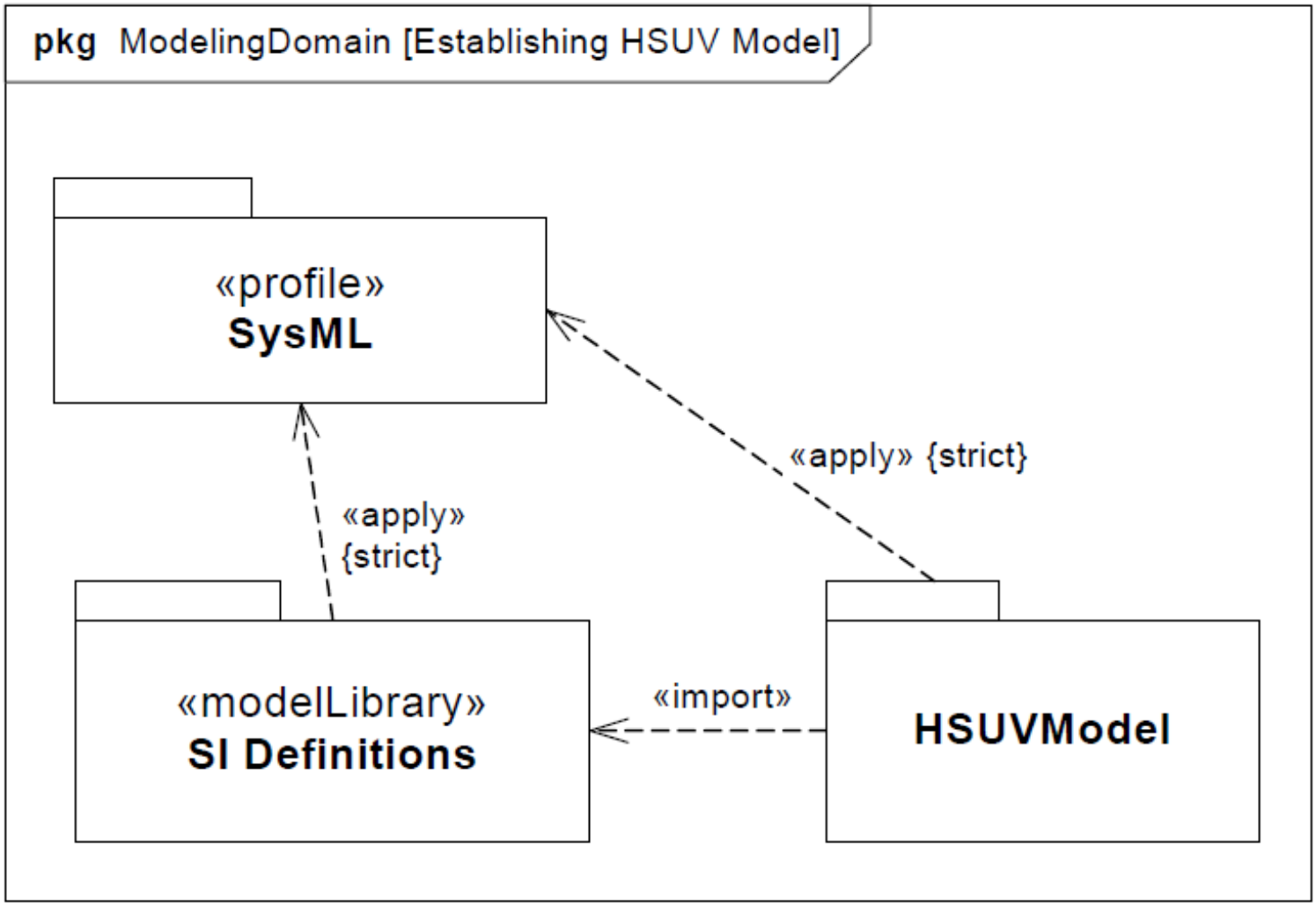
Example: Requirements



SysML Sample Problem

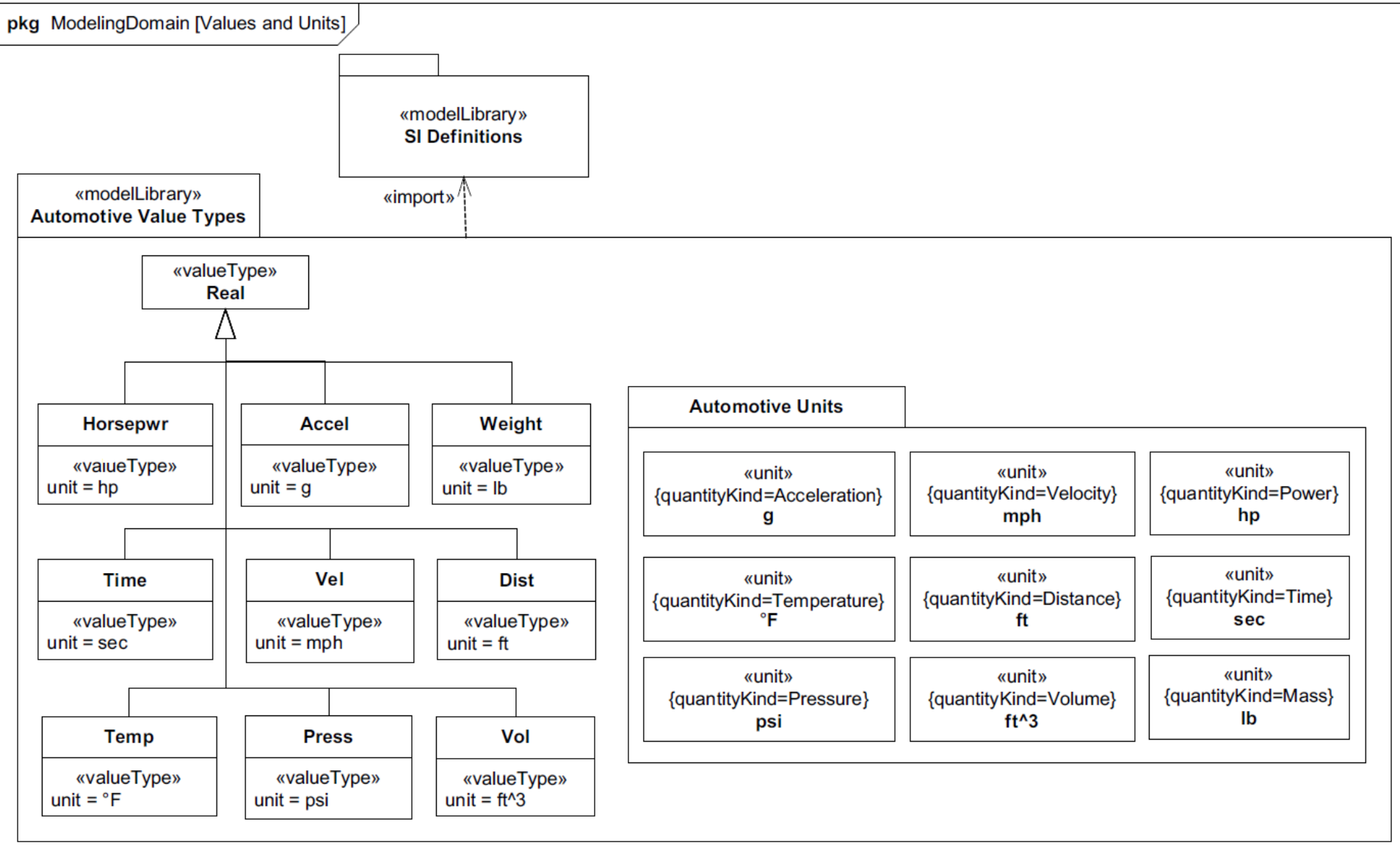
HSUV

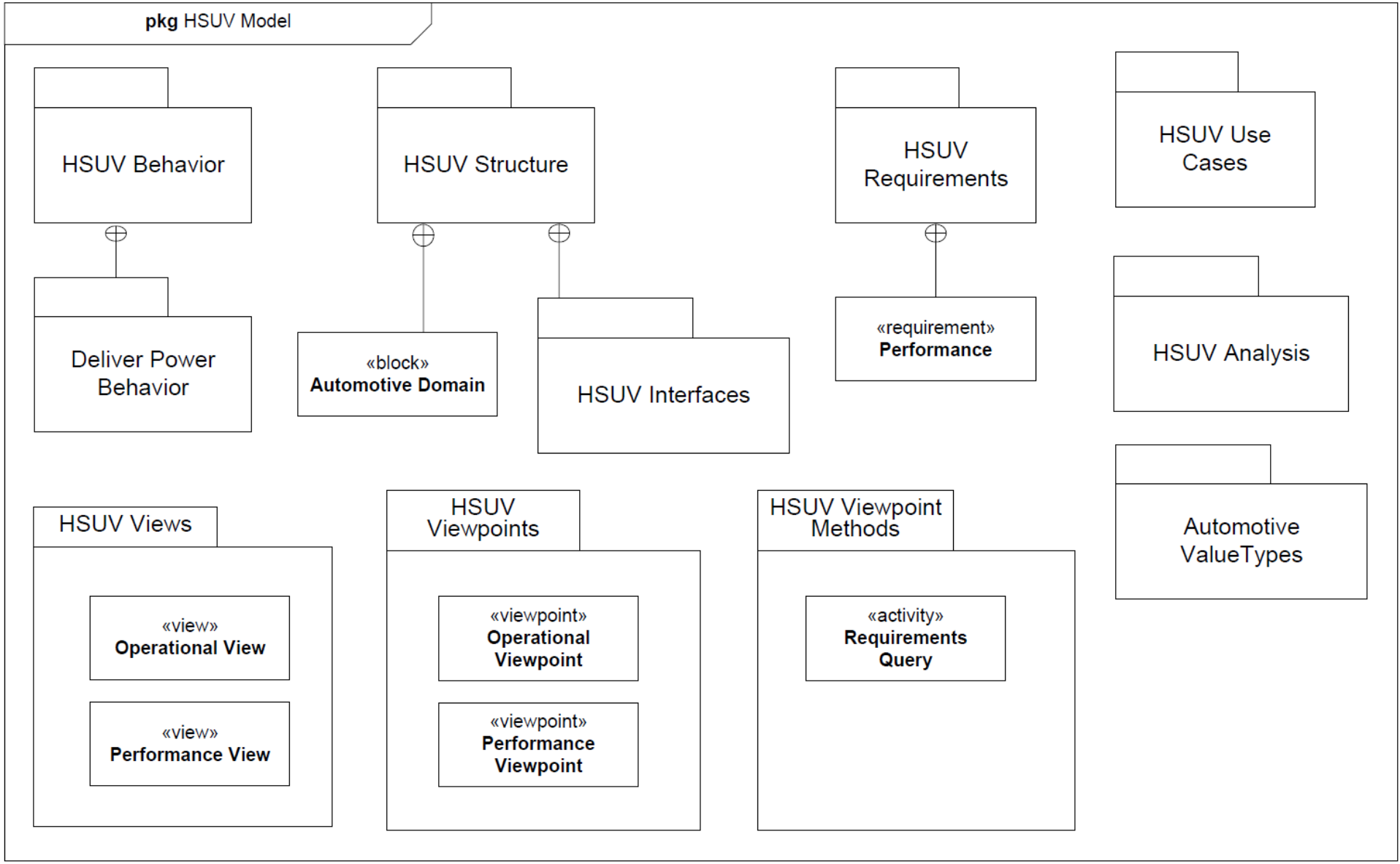
- Model of a Hybrid gas/electric powered Sport Utility Vehicle
- Part of OMG's SysML specification



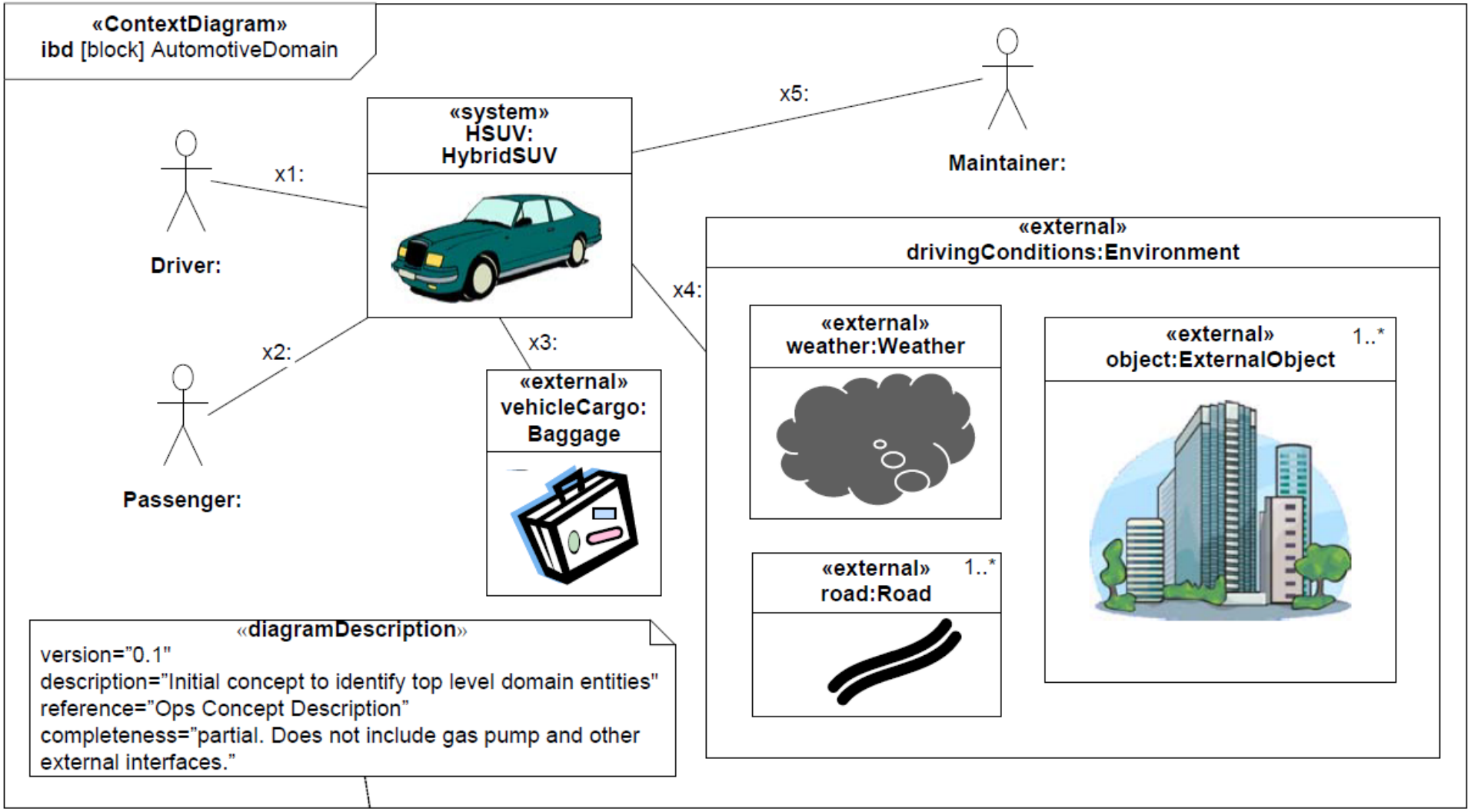
Establishing the User Model by Importing and Applying SysML Profile & Model Library (Package Diagram)

Defining valueTypes and units to be Used in the Sample Problem

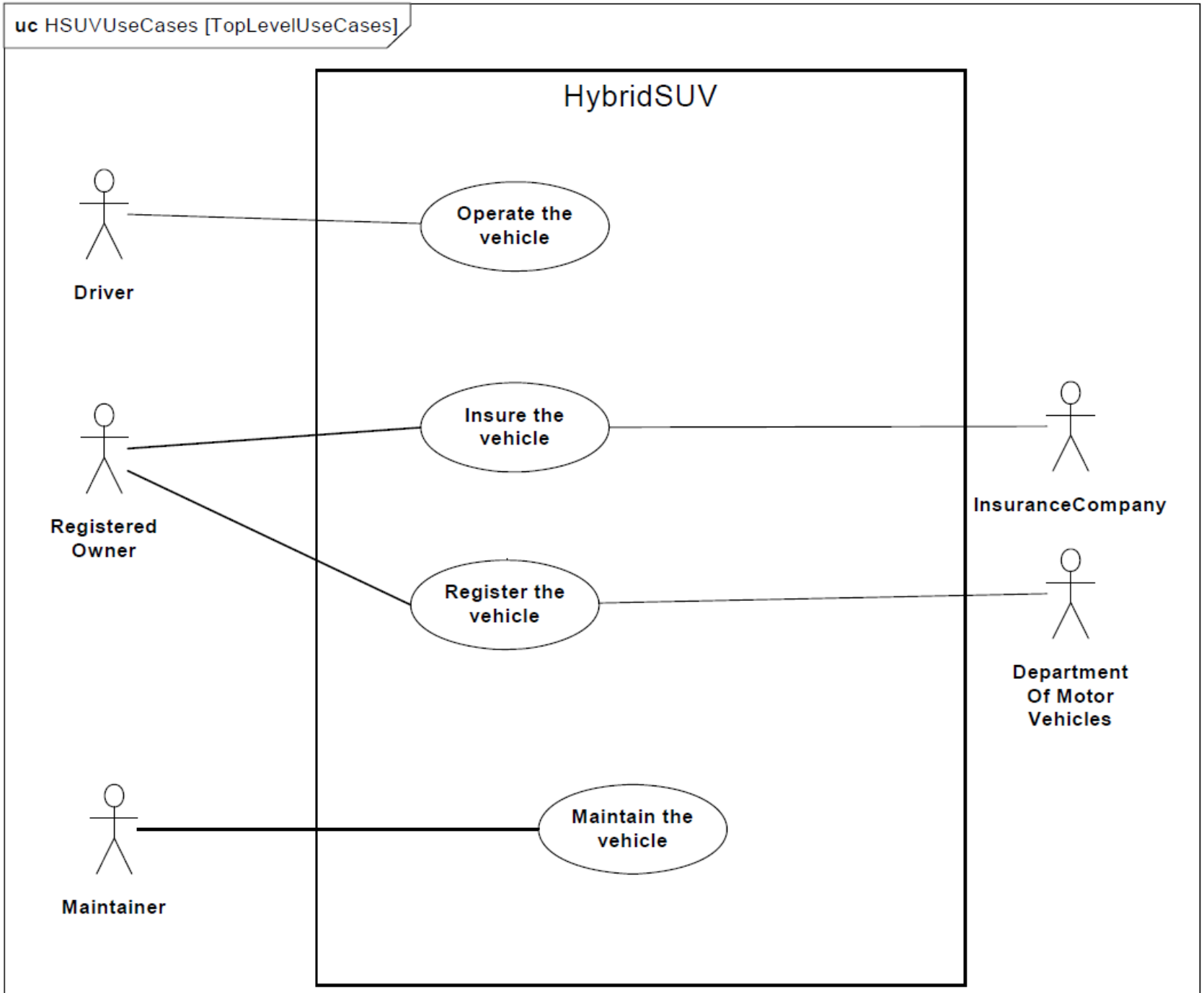




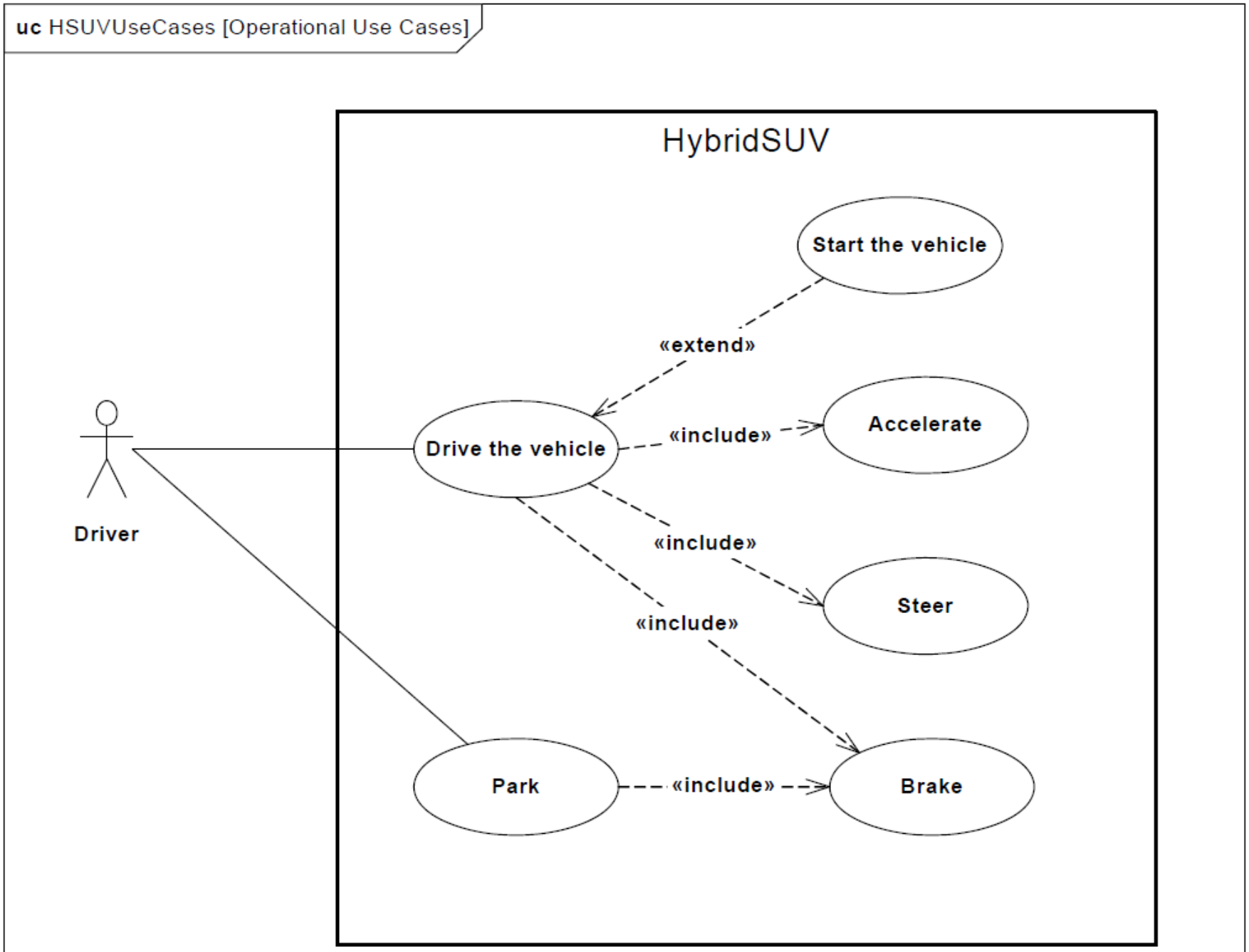
Establishing Structure of the User Model using Packages and Views (Package Diagram)



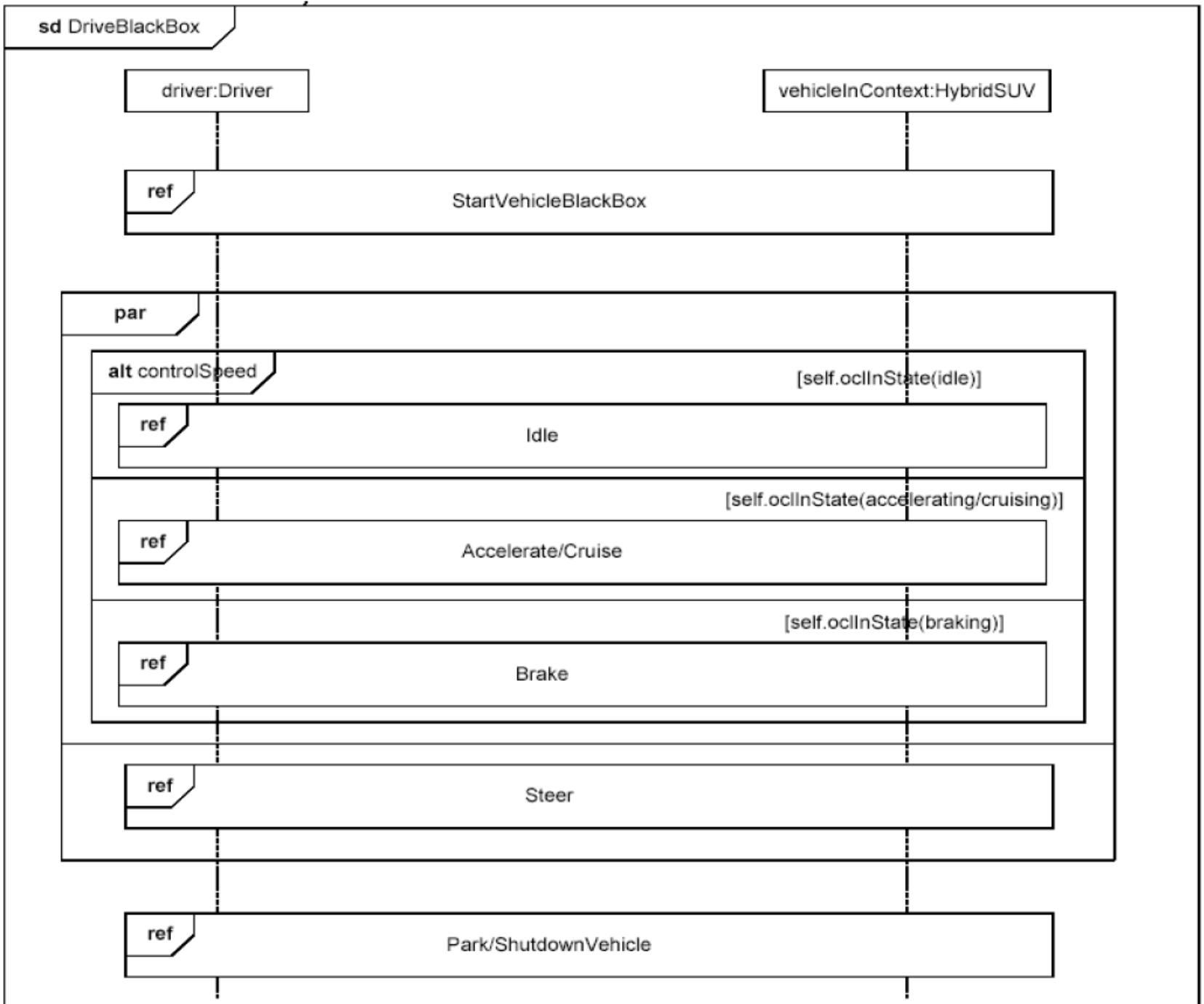
Establishing the Context of the Hybrid SUV System using a User-Defined Context Diagram. (Internal Block Diagram) Completeness of Diagram Noted in Diagram Description



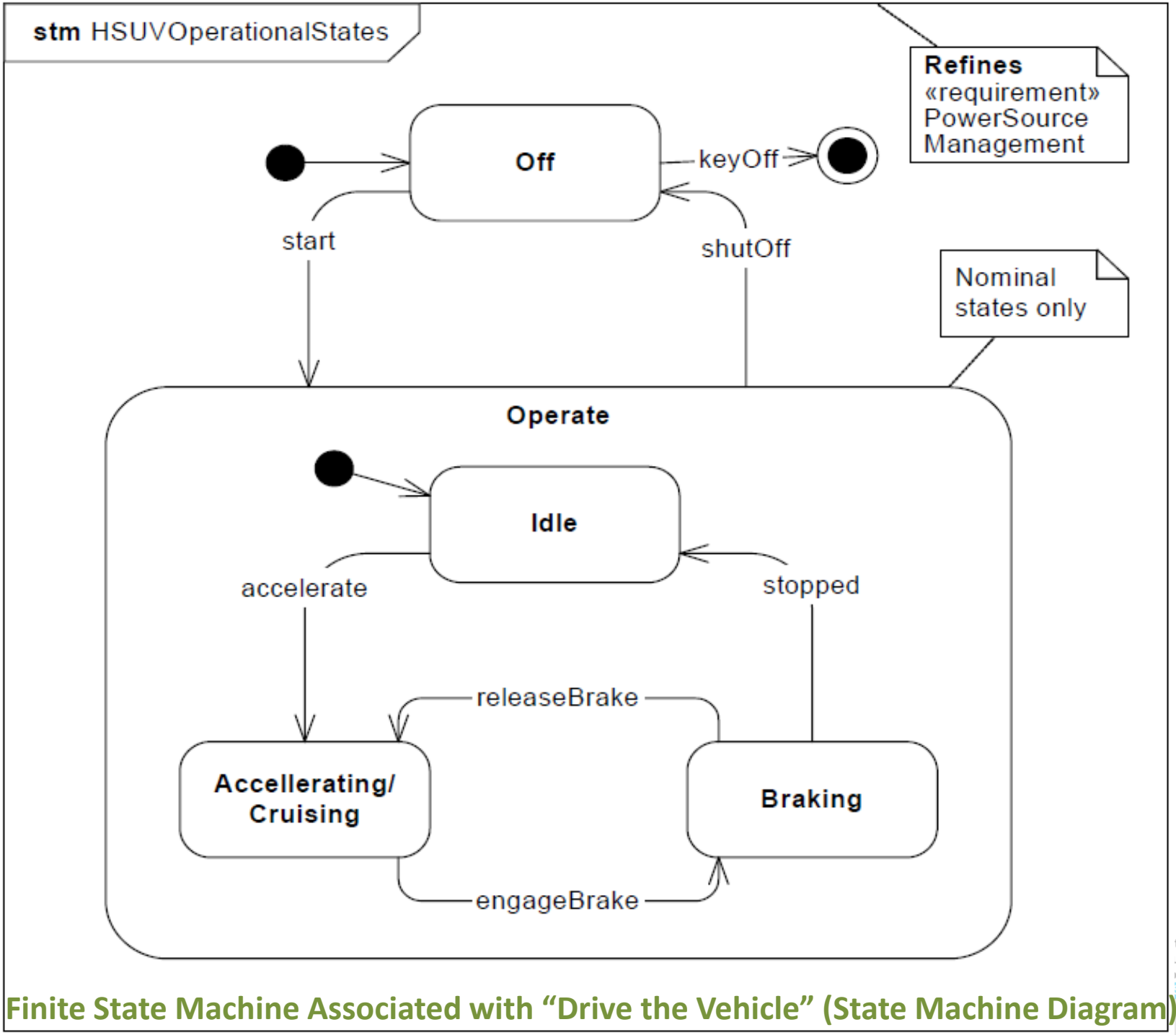
Establishing Top Level Use Cases for the Hybrid SUV (Use Case Diagram)



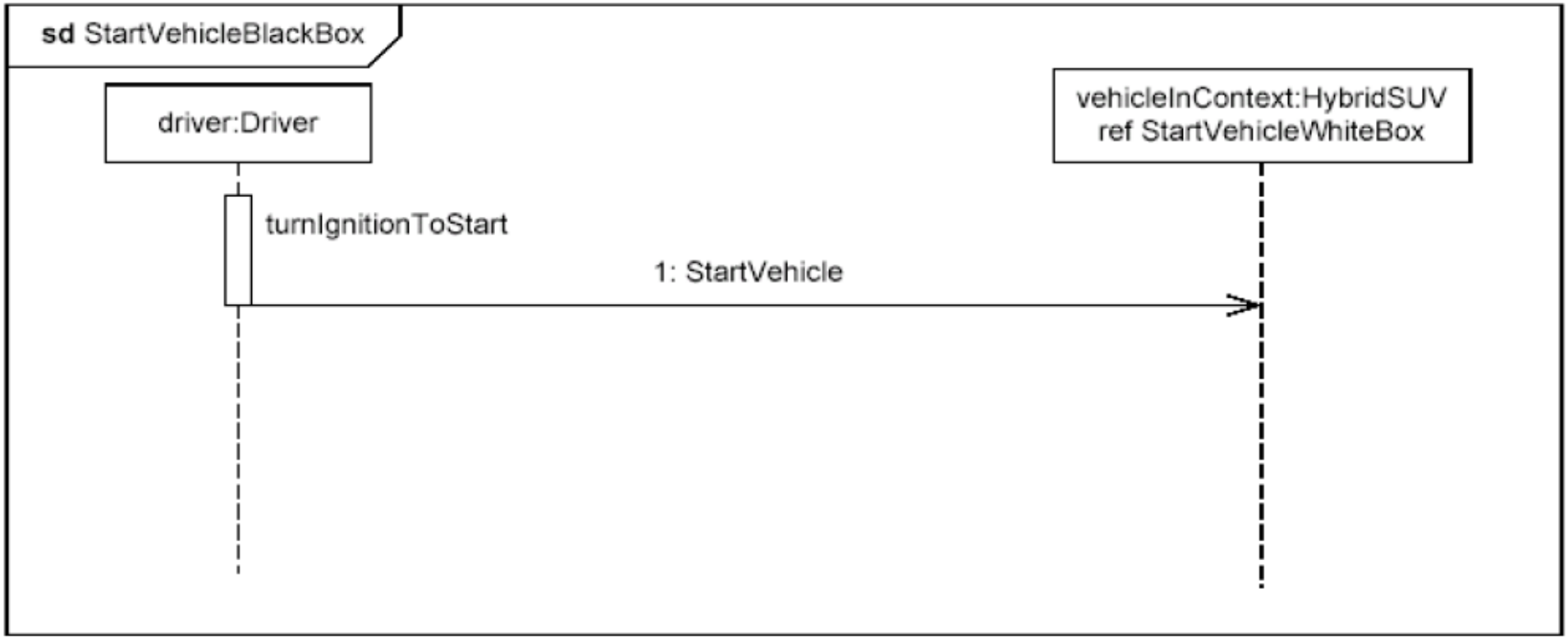
Establishing Operational Use Cases for “Drive the Vehicle” (Use Case Diagram)



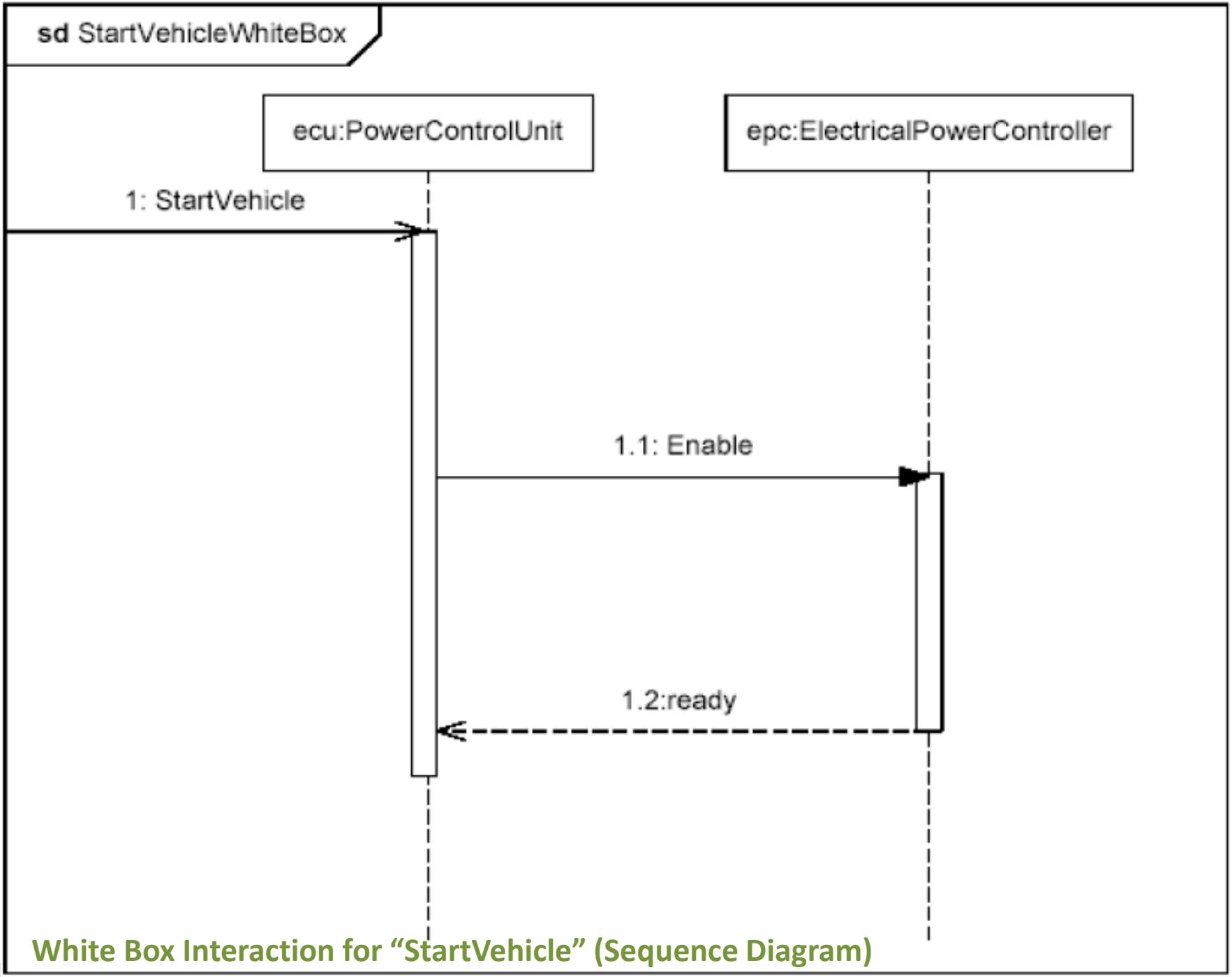
Elaborating Black Box Behavior for the "Drive the Vehicle" Use Case (Sequence Diagram)



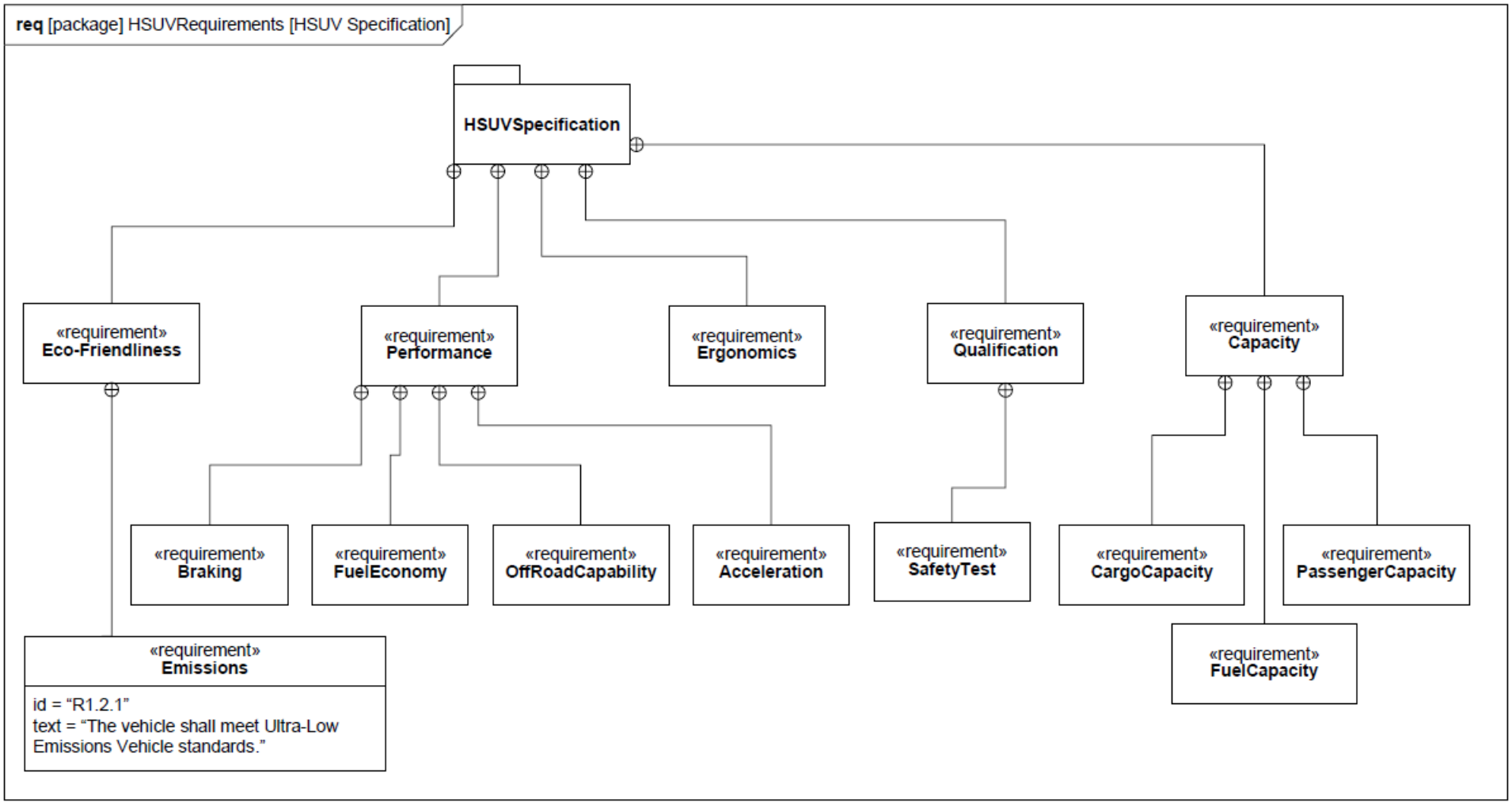
Finite State Machine Associated with "Drive the Vehicle" (State Machine Diagram)



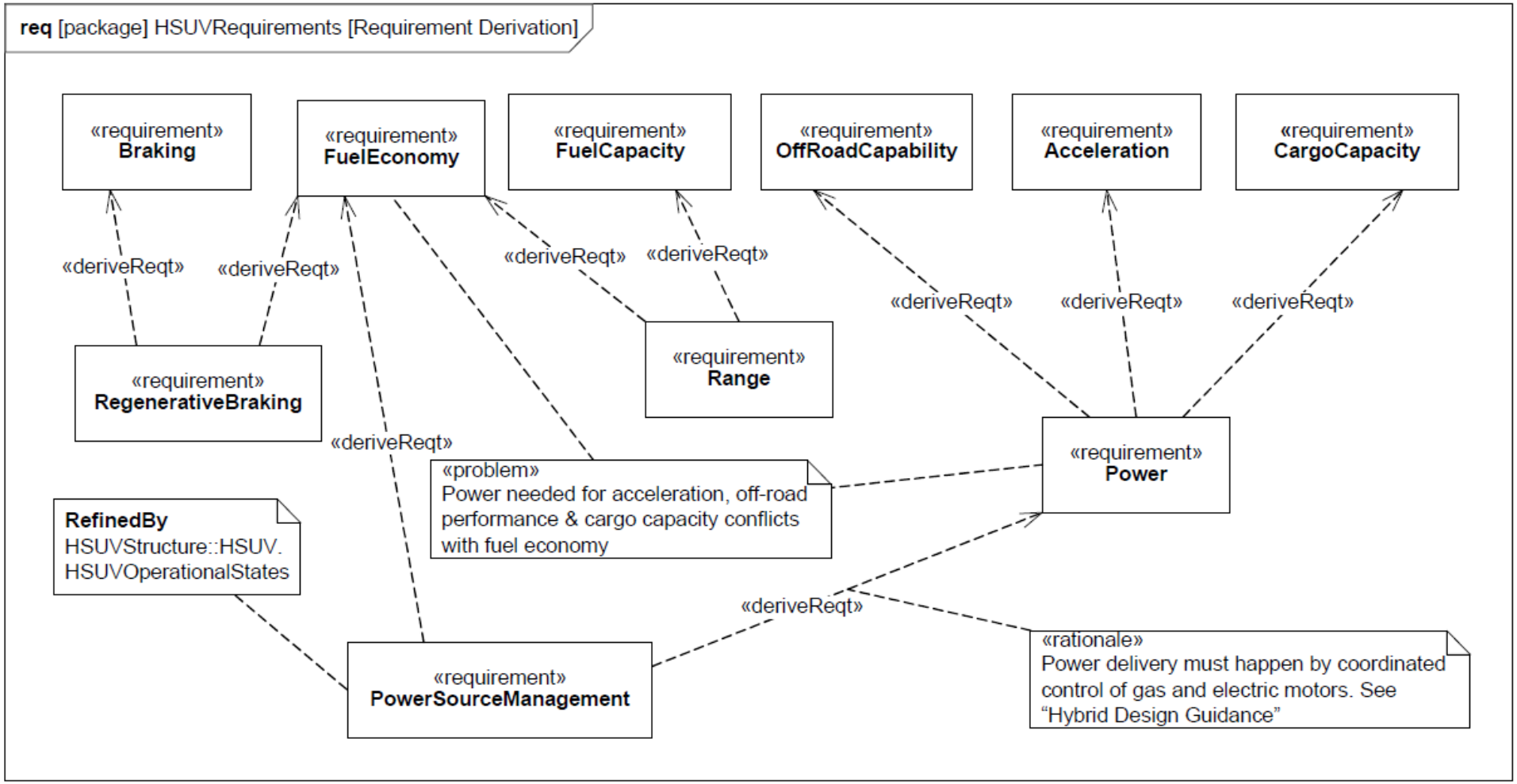
Black Box Interaction for “StartVehicle,” referencing White Box Interaction (Sequence Diagram)



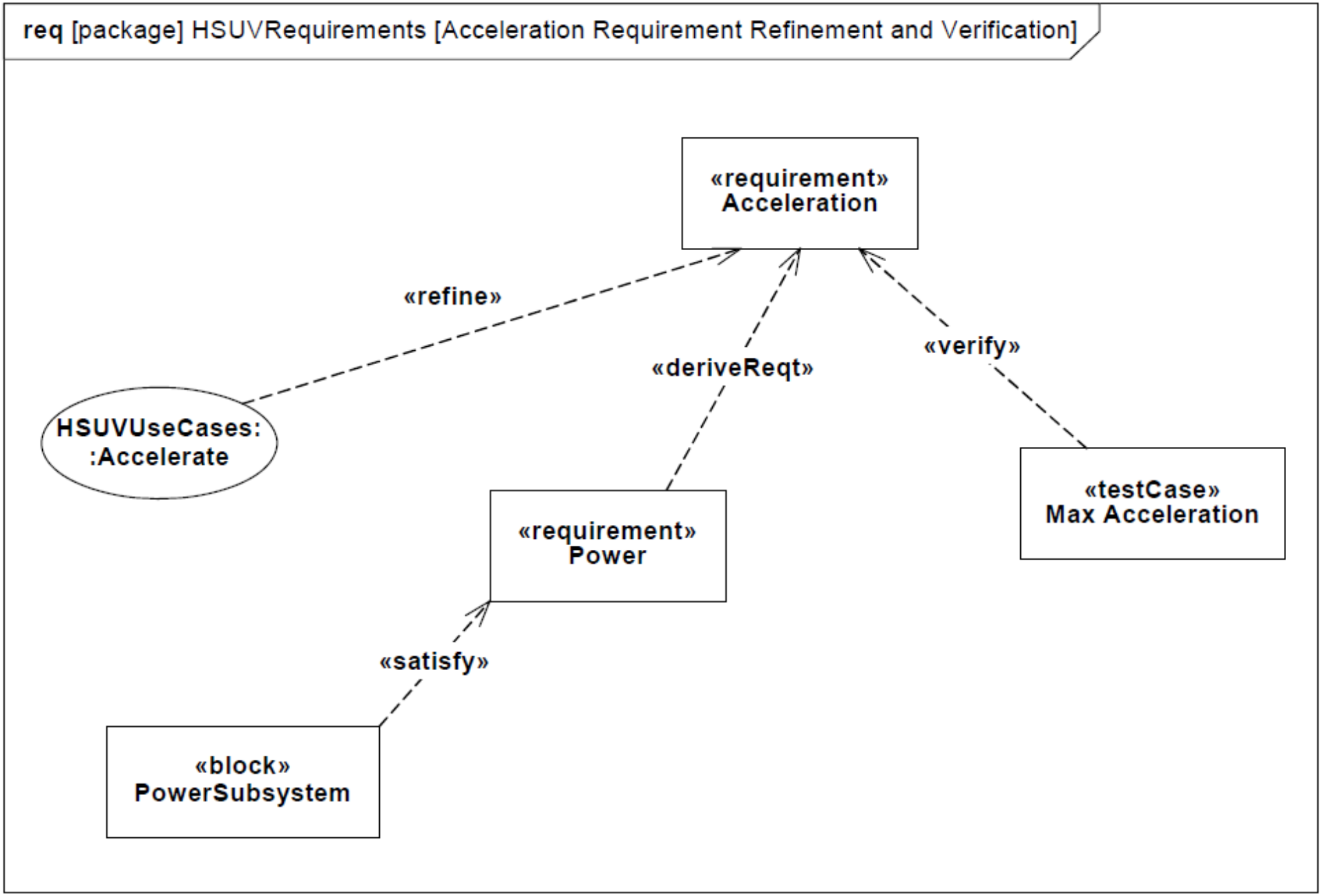
White Box Interaction for “StartVehicle” (Sequence Diagram)



Establishing HSUV Requirements Hierarchy (containment) - (Requirements Diagram)



Establishing Derived Requirements and Rationale from Lowest Tier of Requirements Hierarchy (Requirements Diagram)



Acceleration Requirement Relationships (Requirements Diagram)

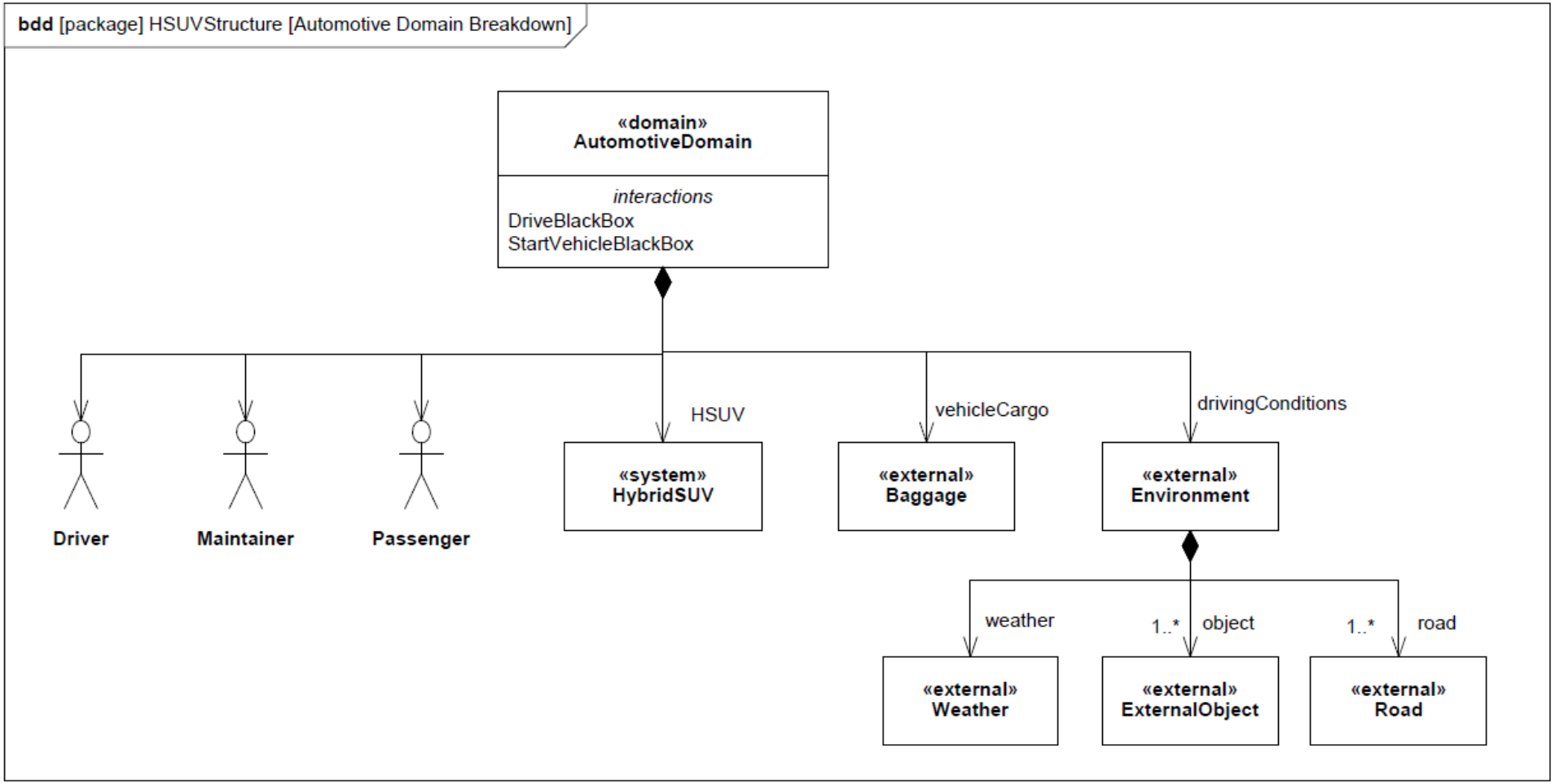
table [requirement] Performance [Decomposition of Performance Requirement]

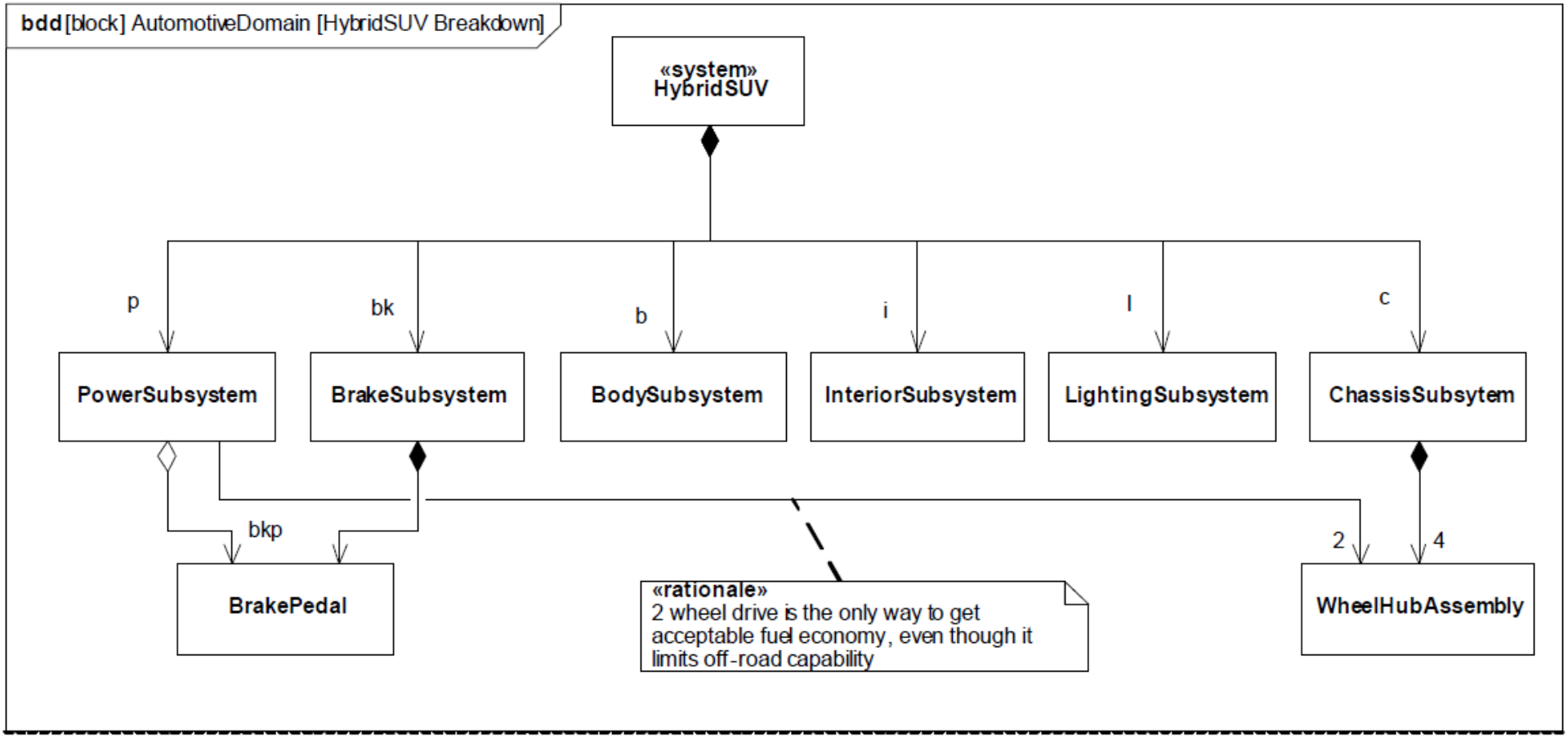
id	name	text
2	Performance	The Hybrid SUV shall have the braking, acceleration, and off-road capability of a typical SUV, but have dramatically better fuel economy.
2.1	Braking	The Hybrid SUV shall have the braking capability of a typical SUV.
2.2	FuelEconomy	The Hybrid SUV shall have dramatically better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

table [requirement] Performance [Tree of Performance Requirements]

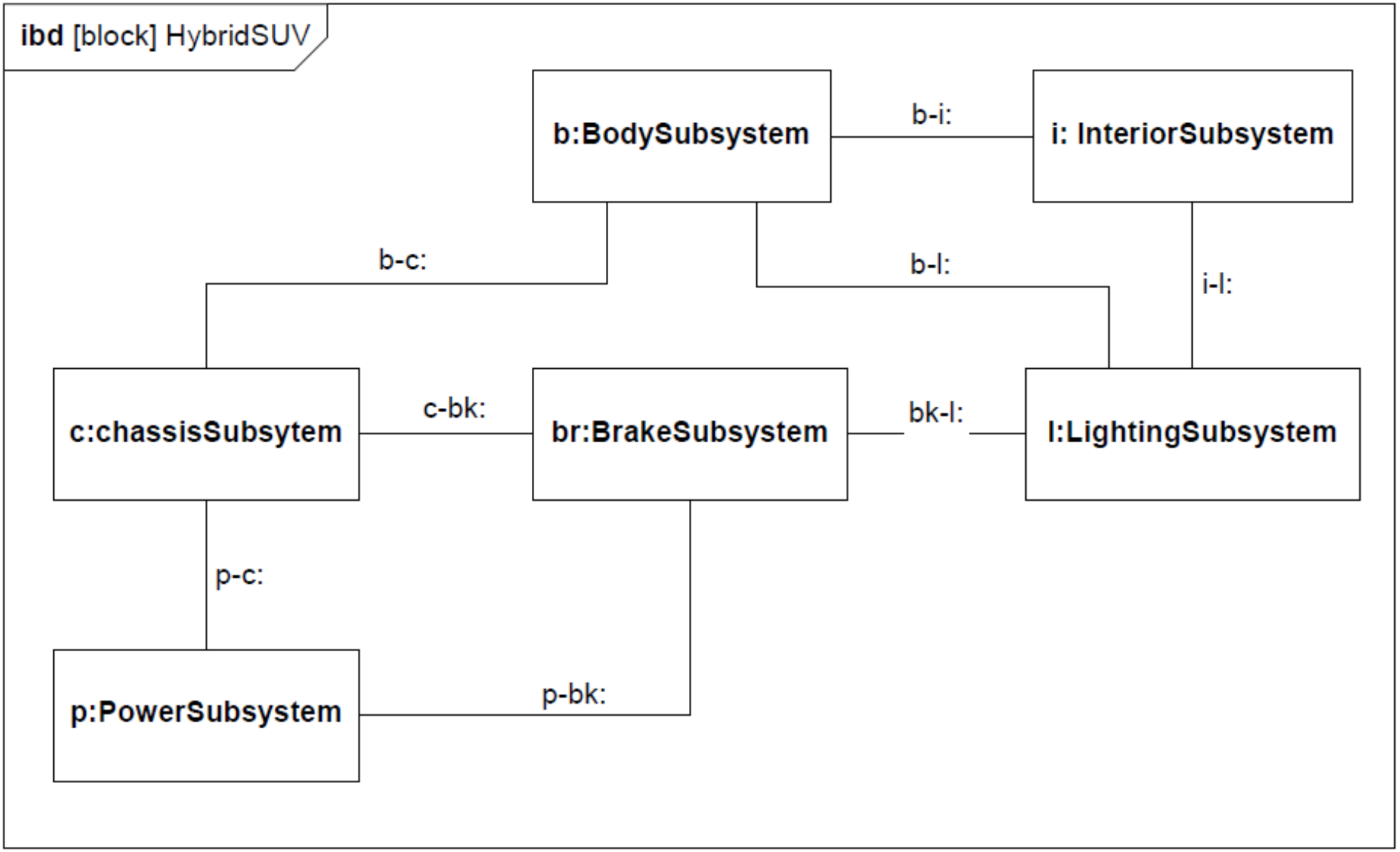
id	name	relation	id	name	relation	id	name
2.1	Braking	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.1	RegenerativeBraking			
2.2	FuelEconomy	deriveReqt	d.2	Range			
4.2	FuelCapacity	deriveReqt	d.2	Range			
2.3	OffRoadCapability	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
2.4	Acceleration	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement
4.1	CargoCapacity	deriveReqt	d.4	Power	deriveReqt	d.2	PowerSourceManagement

Requirements Relationships Expressed in Tabular Format (Table)

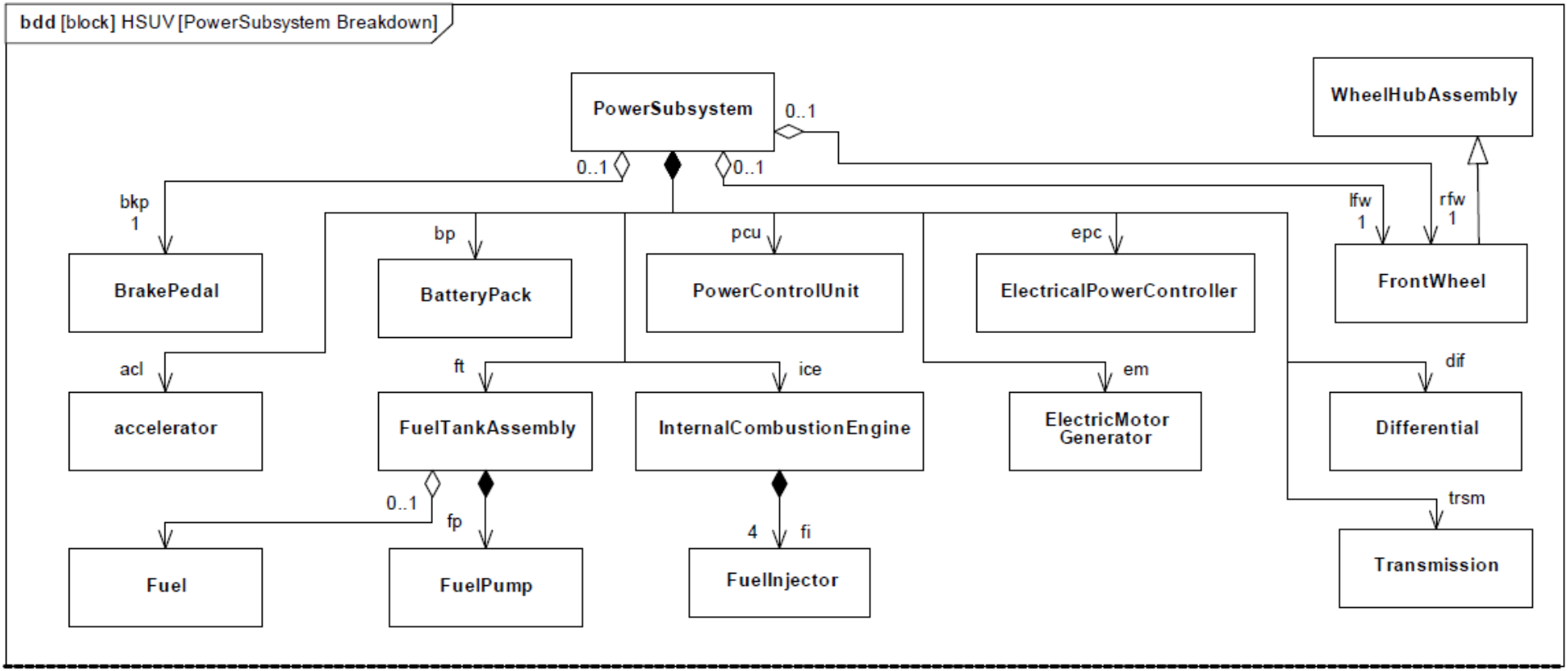

bdd [package] HSUVStructure [Automotive Domain Breakdown]
«domain»
AutomotiveDomain
interactions
DriveBlackBox
StartVehicleBlackBox
HSUV
vehicleCargo
drivingConditions
Driver
Maintainer
Passenger
«system»
HybridSUV
«external»
Baggage
«external»
Environment
weather
1..*
object
1..*
road
«external»
Weather
«external»
ExternalObject
«external»
Road



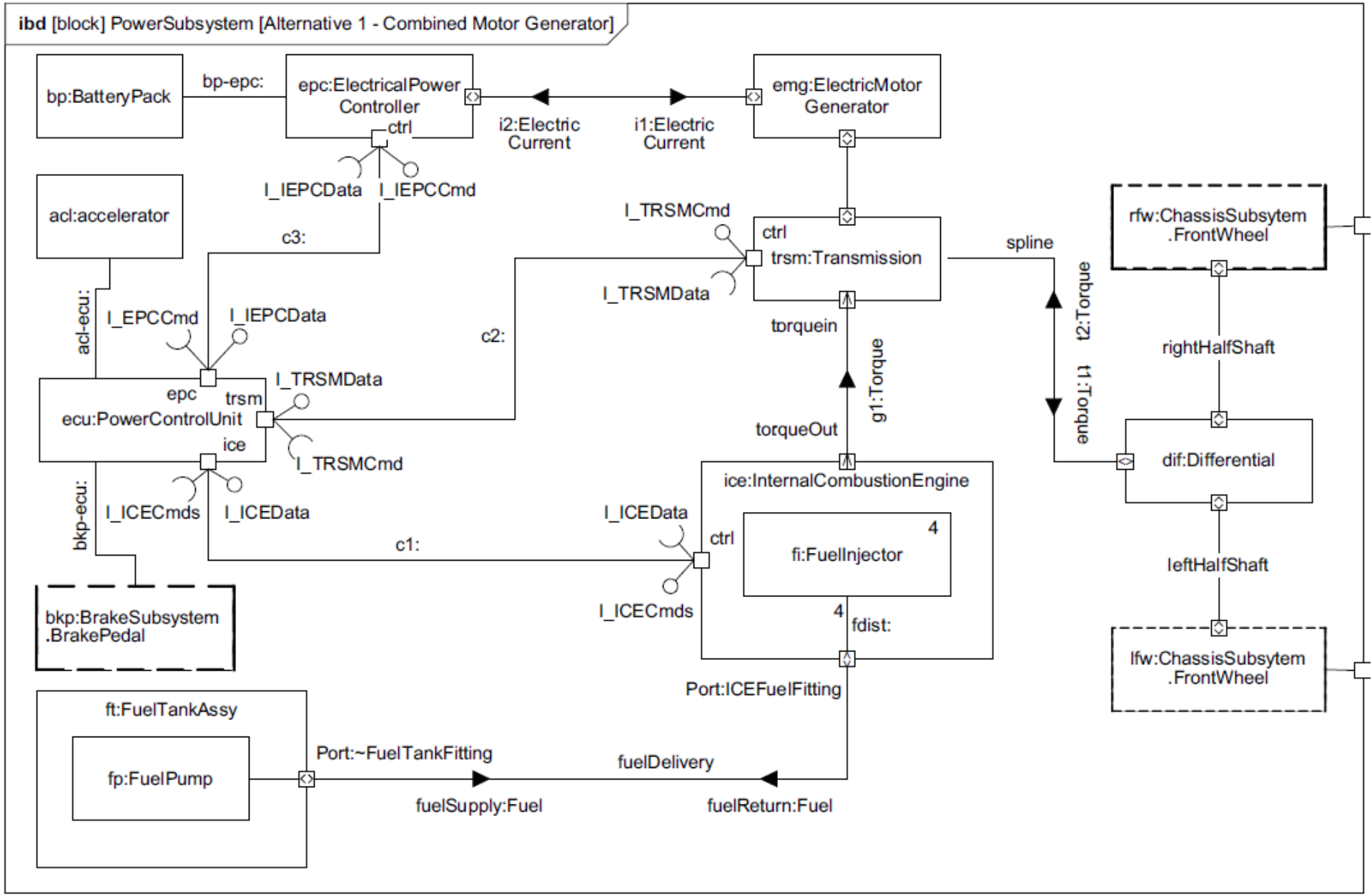
Defining Structure of the Hybrid SUV System (Block Definition Diagram)



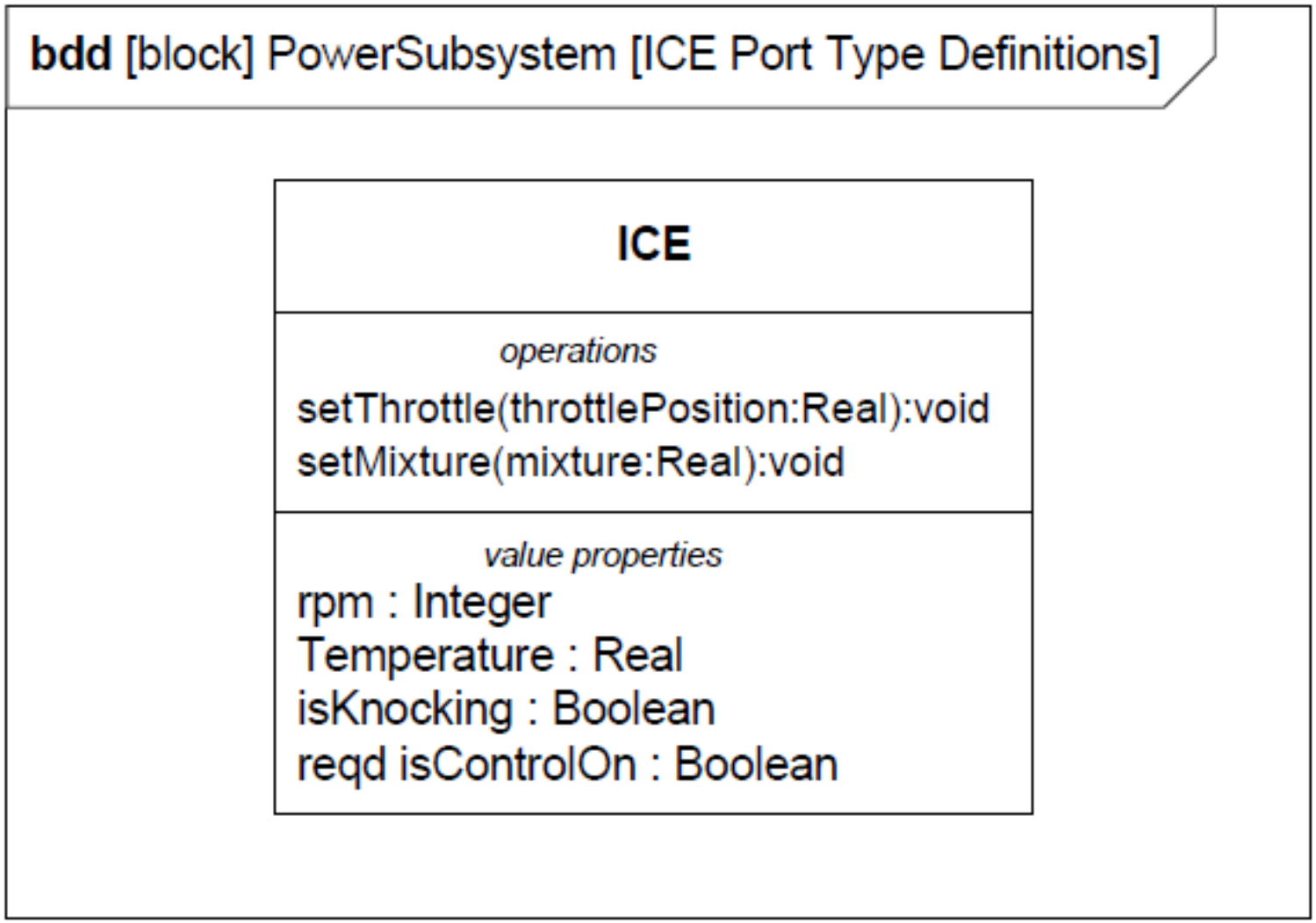
Internal Structure of Hybrid SUV (Internal Block Diagram)



Defining Structure of Power Subsystem (Block Definition Diagram)

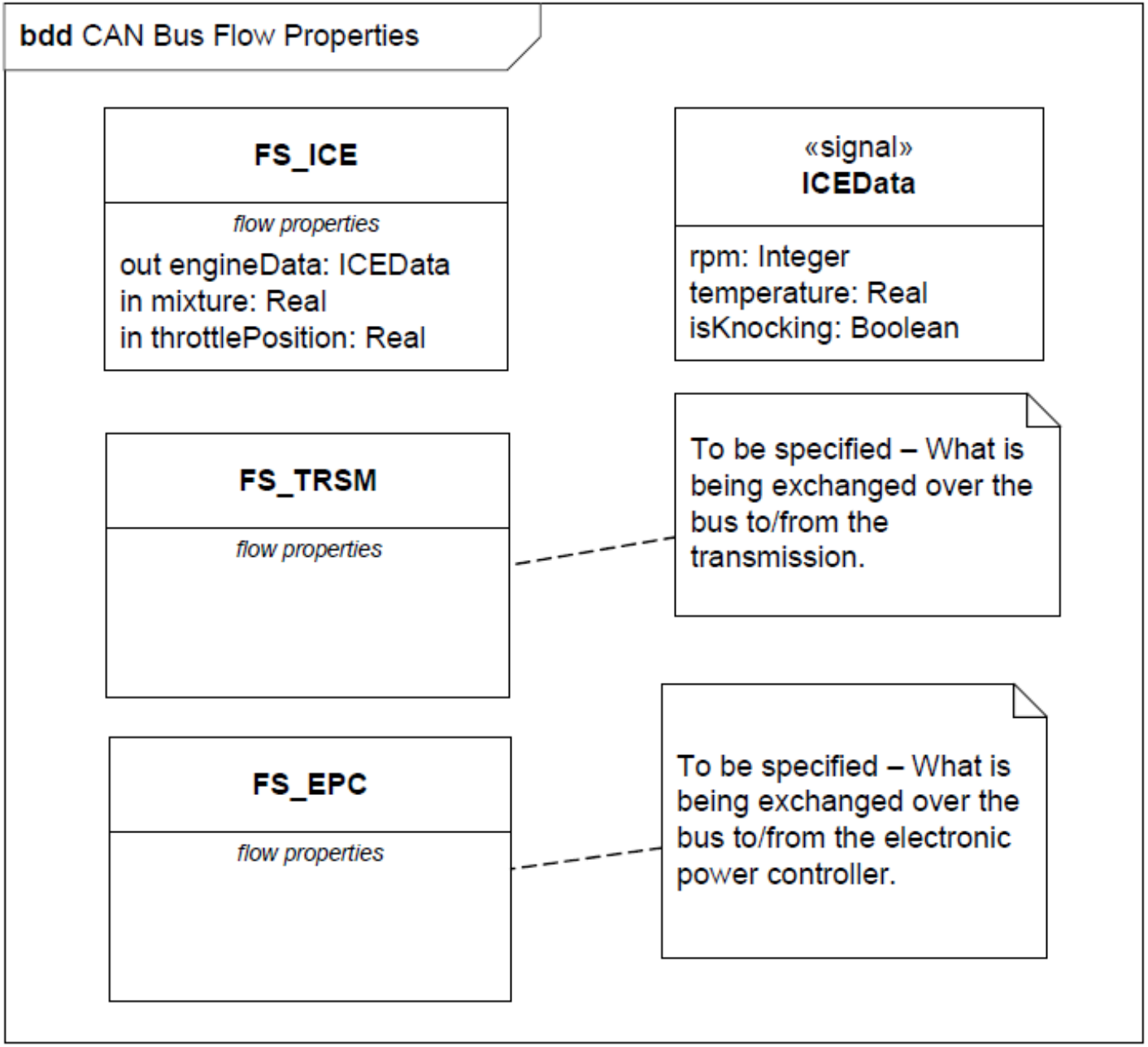


Internal Structure of the Power Subsystem (Internal Block Diagram)

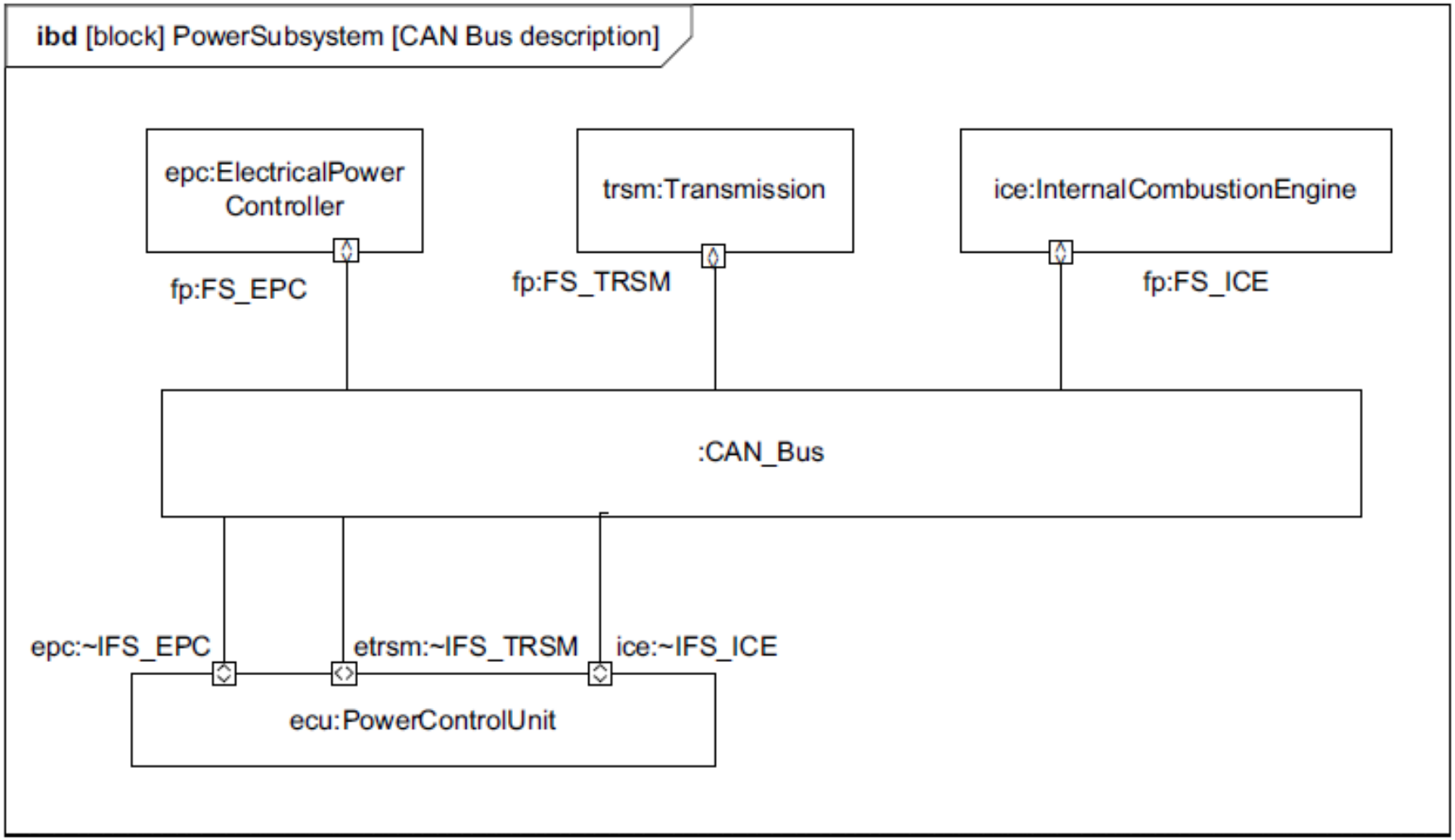


Department of Distributed and Dependable Systems

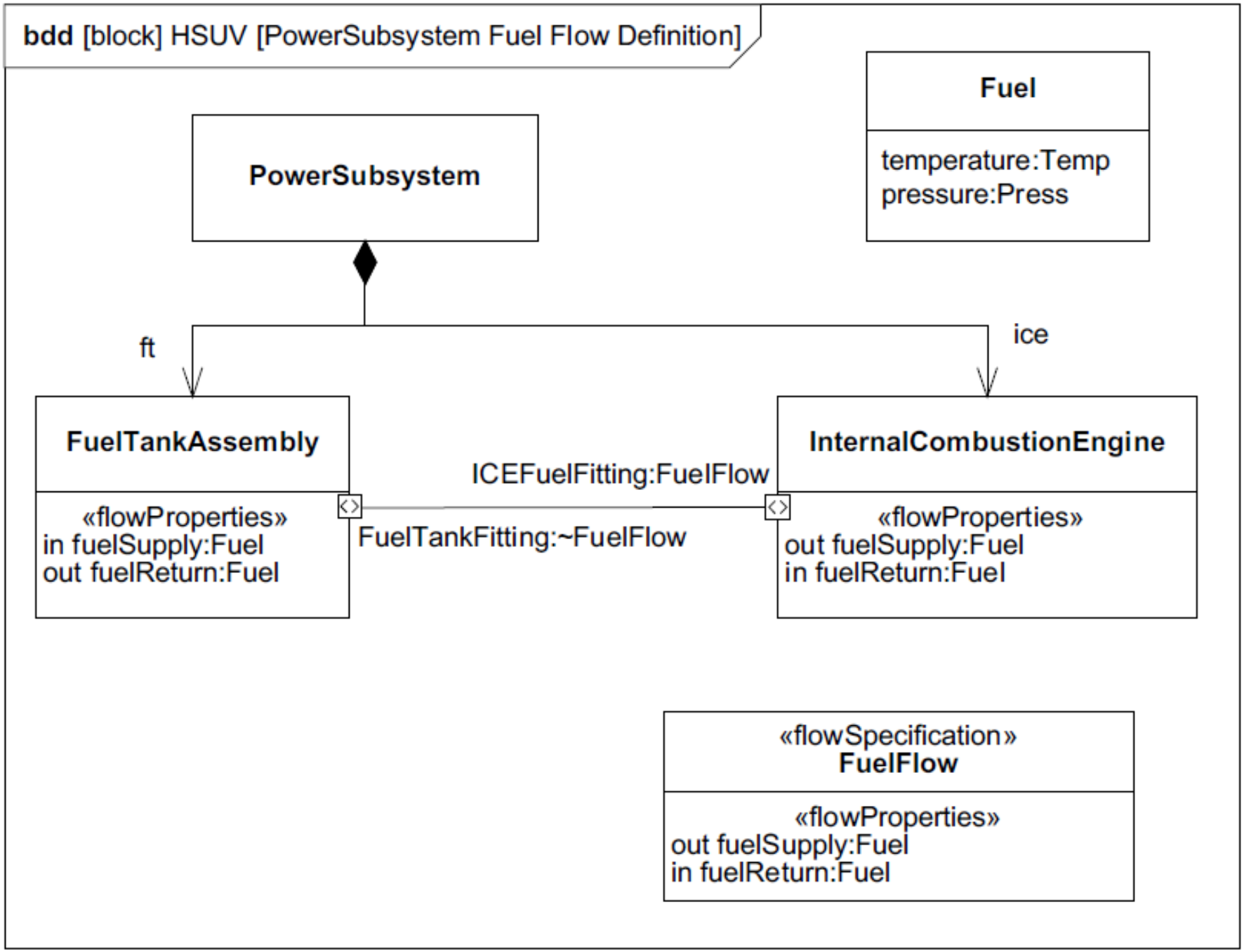
Blocks Typing Ports in the Power Subsystem (Block Definition Diagram)



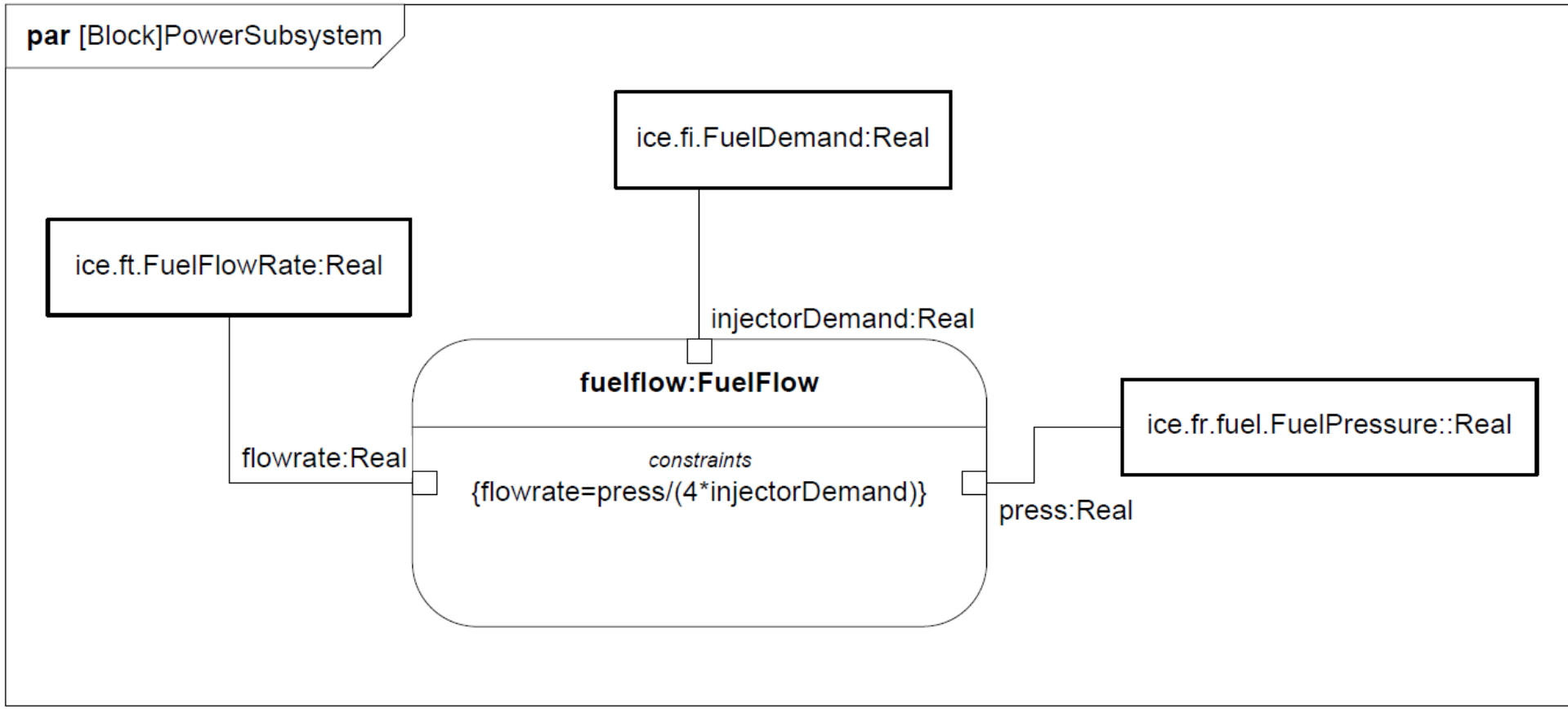
Initially Defining Port Types with Flow Properties for the CAN Bus (Block Definition Diagram)



Consolidating Connectors into the CAN Bus. (Internal Block Diagram)

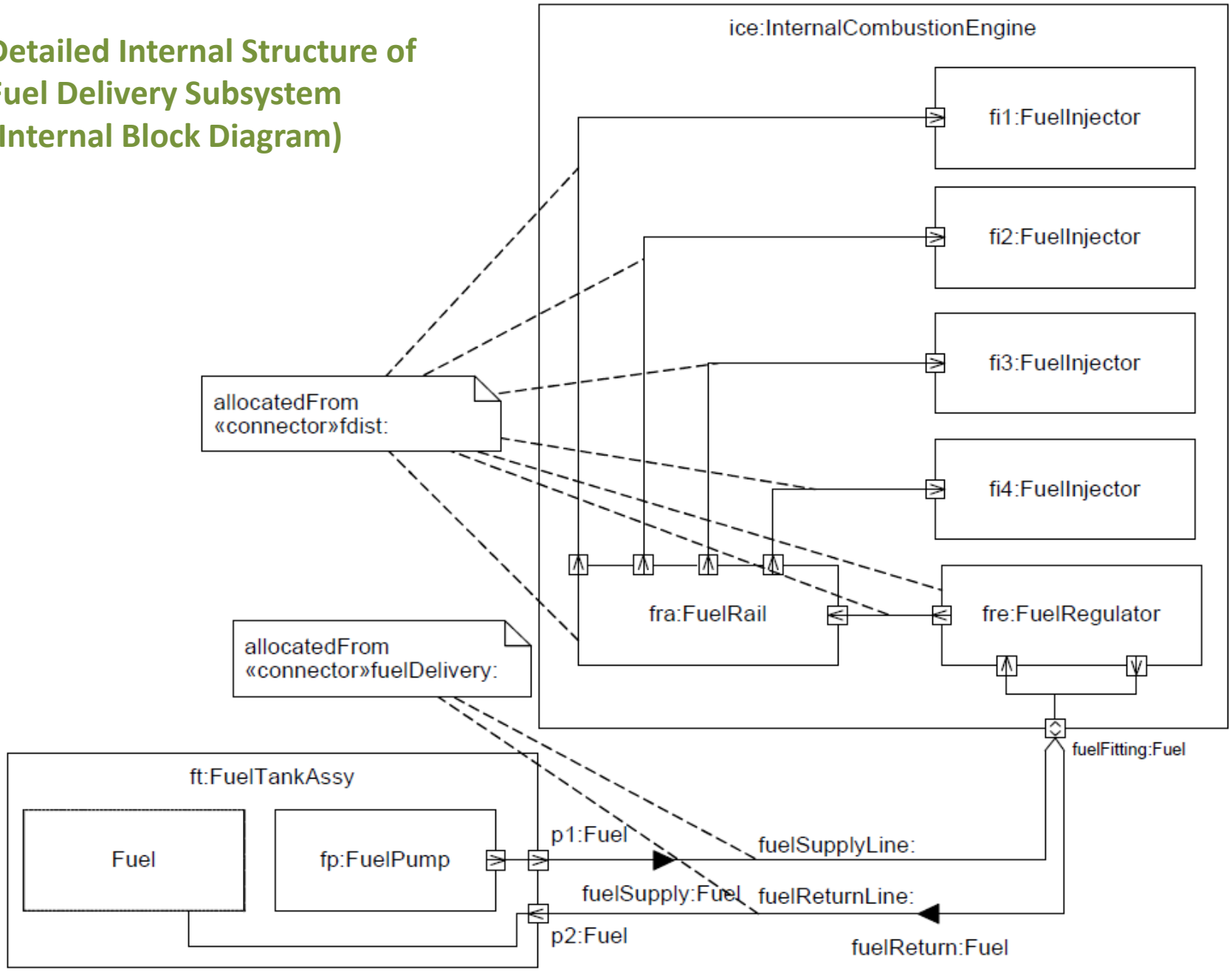


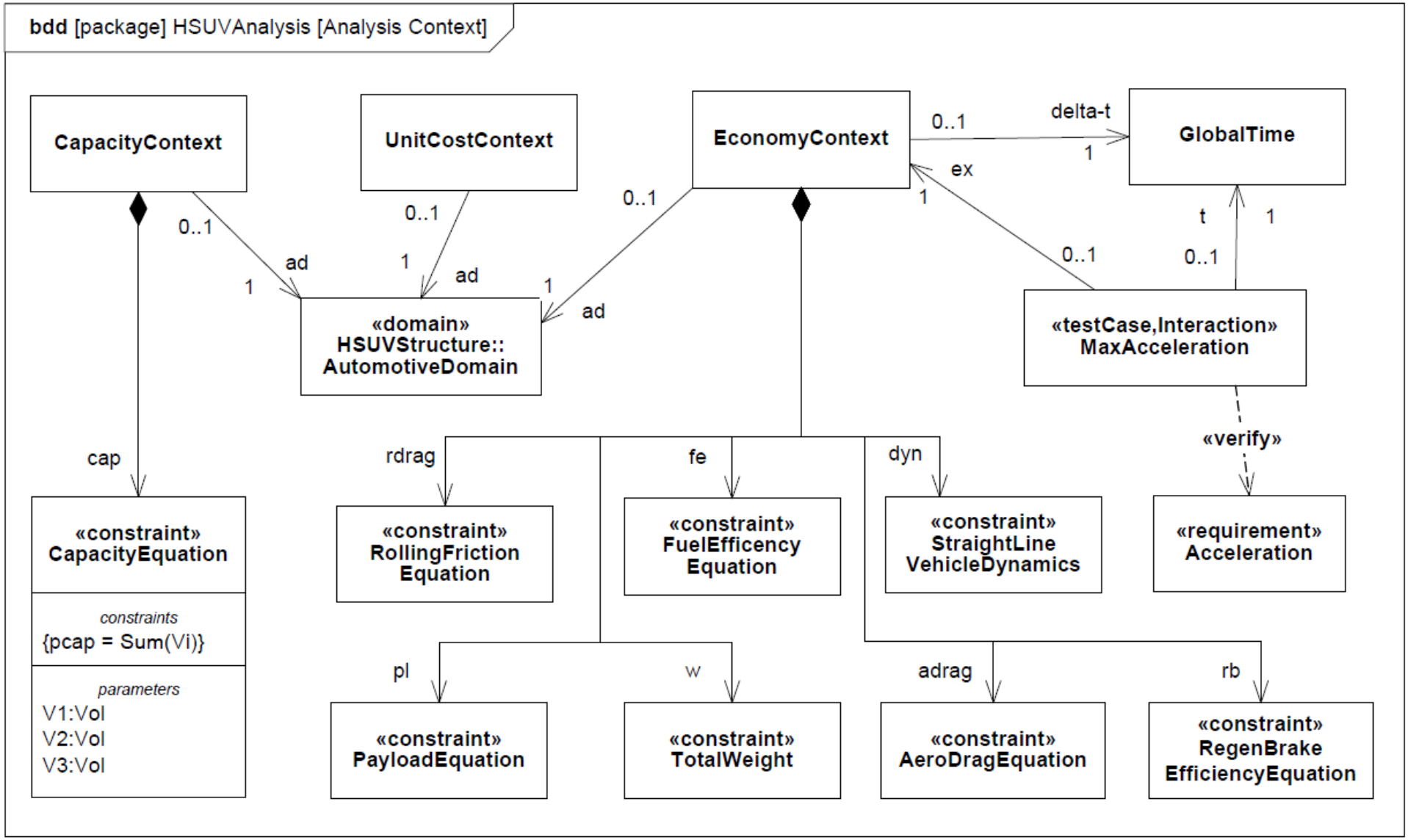
Elaborating Definition of Fuel Flow. (Block Definition Diagram)

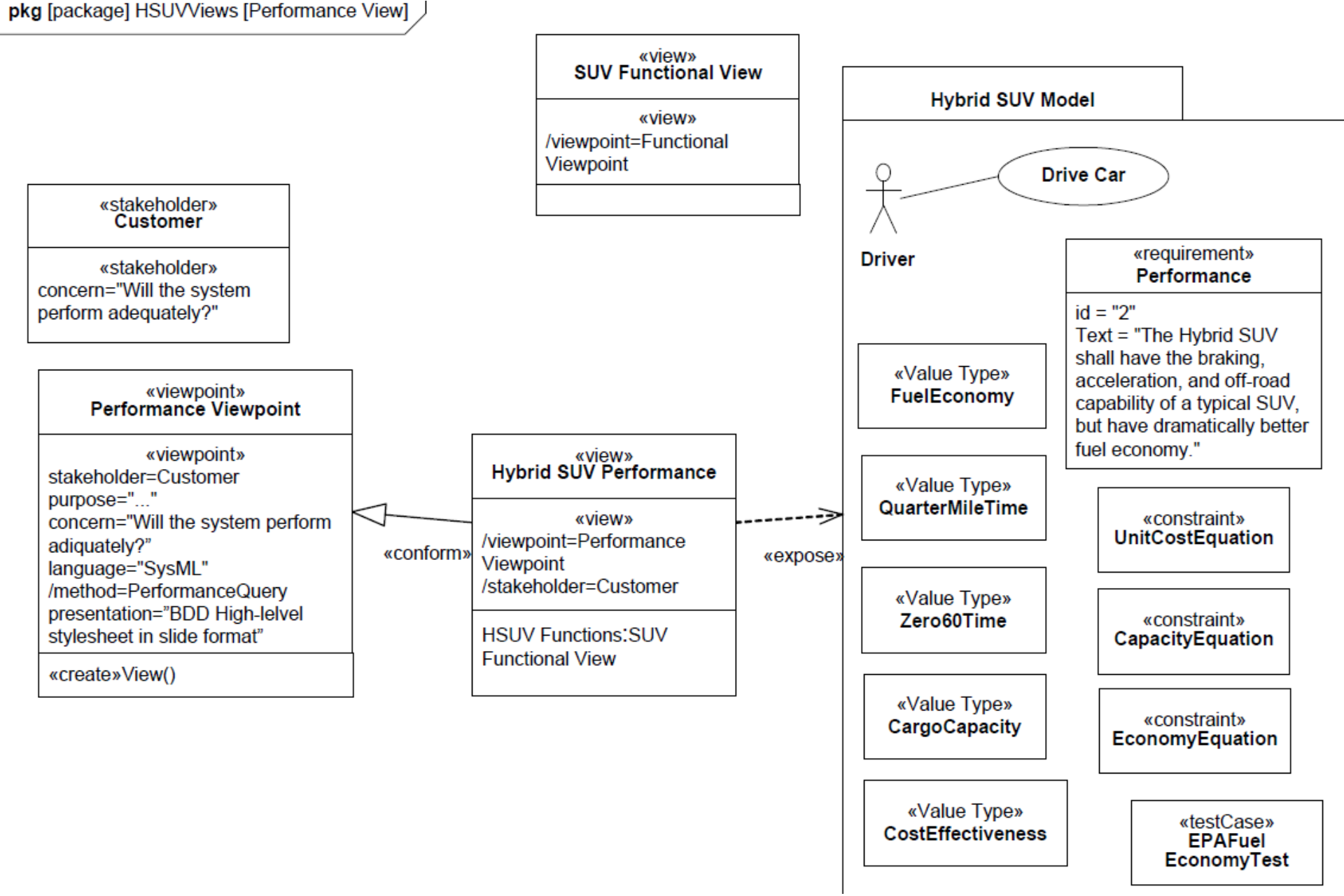


Defining Fuel Flow Constraints (Parametric Diagram)

Detailed Internal Structure of Fuel Delivery Subsystem (Internal Block Diagram)

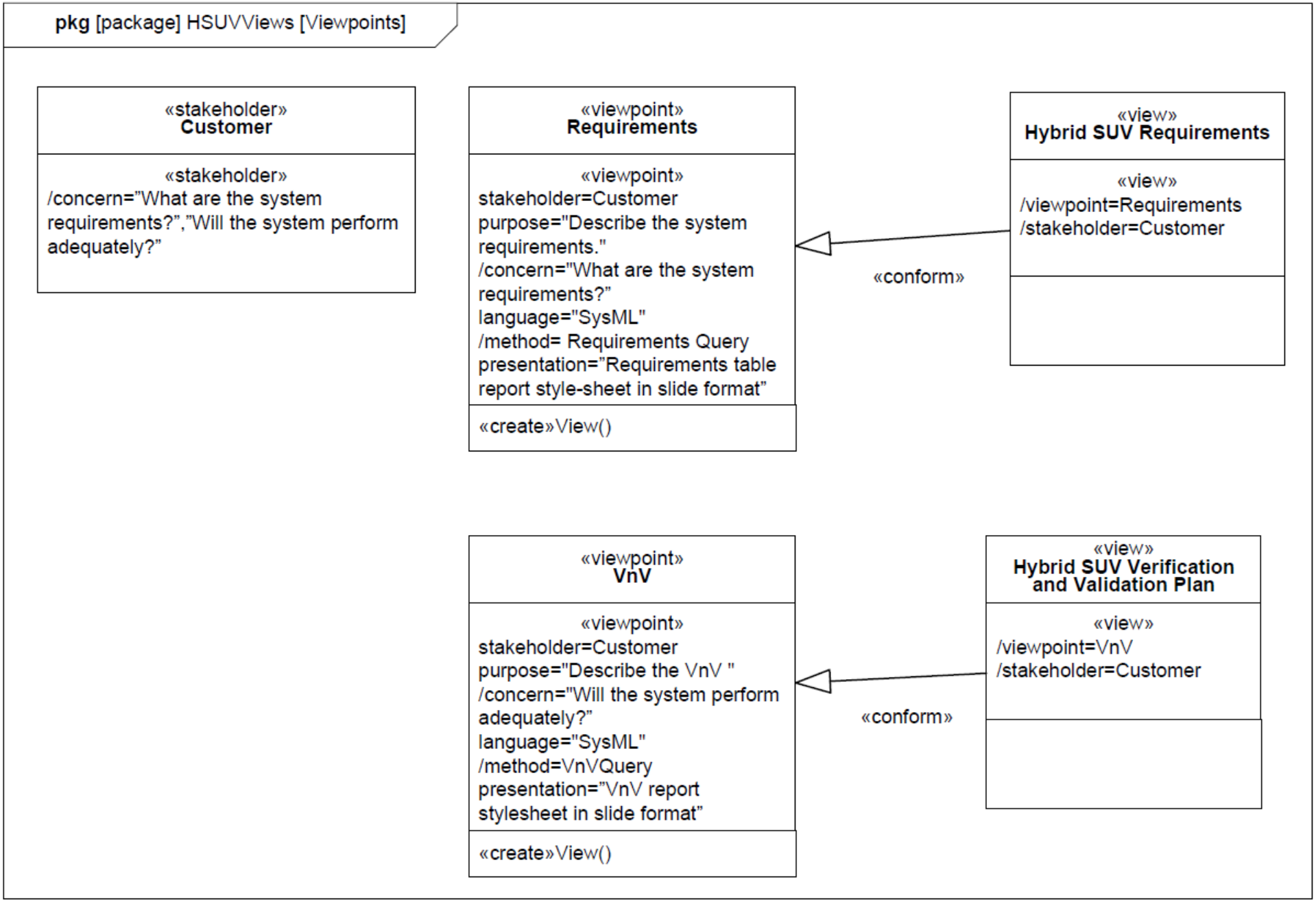



bdd [package] HSUVAnalysis [Analysis Context]
CapacityContext
UnitCostContext
EconomyContext
GlobalTime
«domain»
HSUVStructure::
AutomotiveDomain
«testCase,Interaction»
MaxAcceleration
«constraint»
CapacityEquation
constraints
{pcap = Sum(Vi)}
parameters
V1:Vol
V2:Vol
V3:Vol
«constraint»
RollingFriction
Equation
«constraint»
FuelEfficency
Equation
«constraint»
StraightLine
VehicleDynamics
«requirement»
Acceleration
«constraint»
PayloadEquation
«constraint»
TotalWeight
«constraint»
AeroDragEquation
«constraint»
RegenBrake
EfficiencyEquation
0..1
1
ad
cap
rdrag
fe
dyn
«verify»
delta-t
ex
t
pl
w
adrag
rb

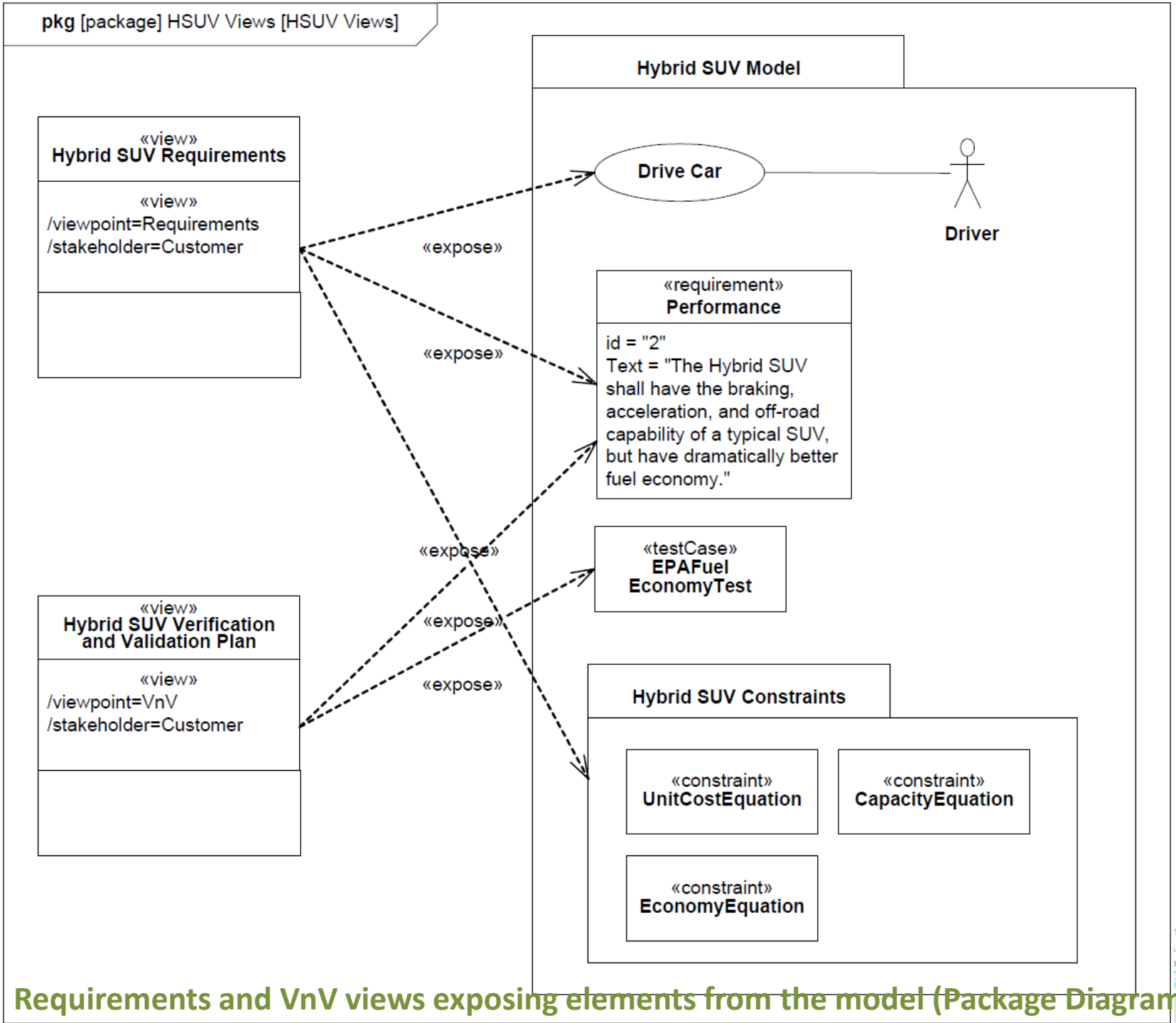


Department of Distributed and Dependable Systems

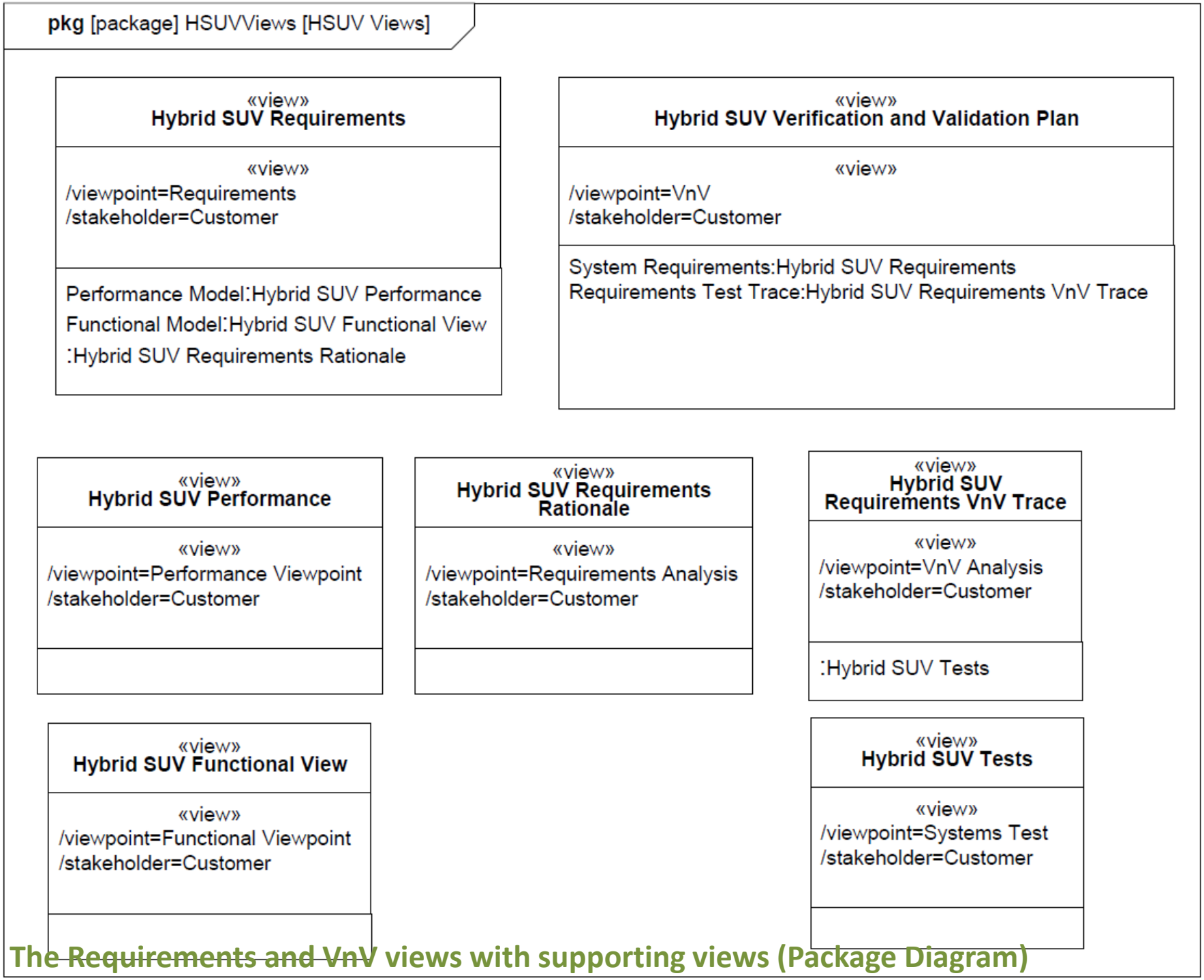
Establishing a Performance View of the User Model (Package Diagram)



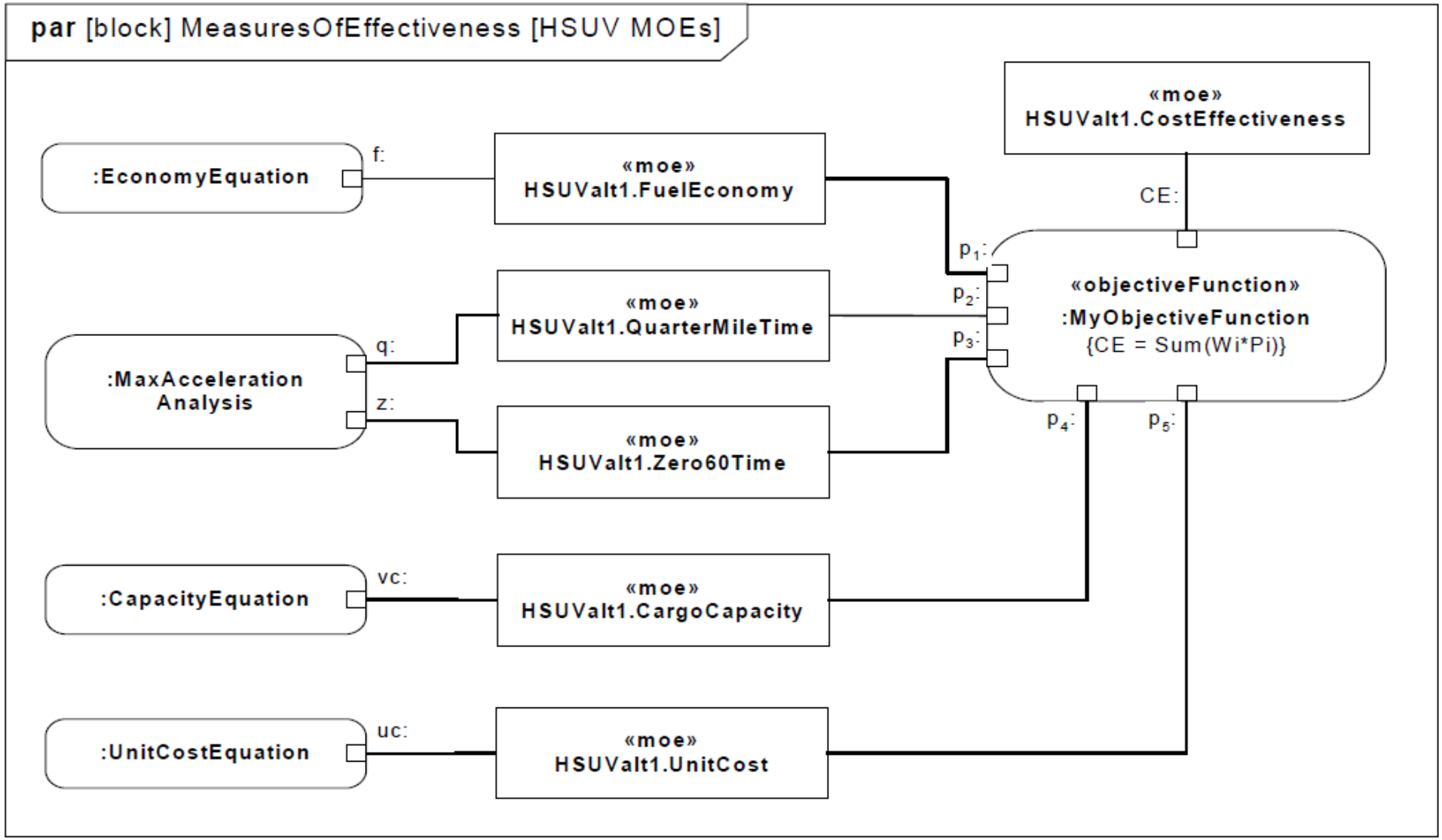
Defining Requirements and VnV viewpoints (Package Diagram)

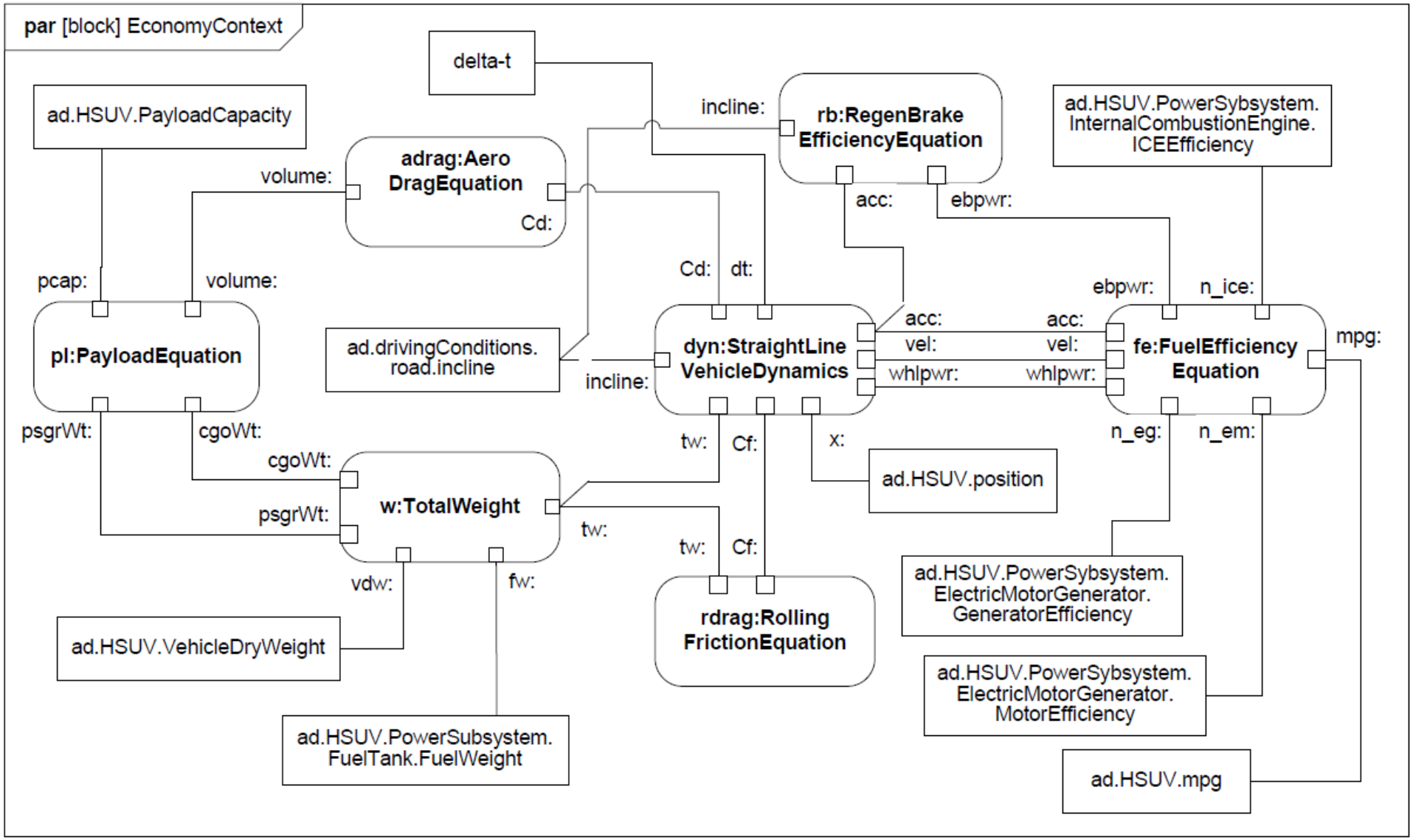


Requirements and VnV views exposing elements from the model (Package Diagram)

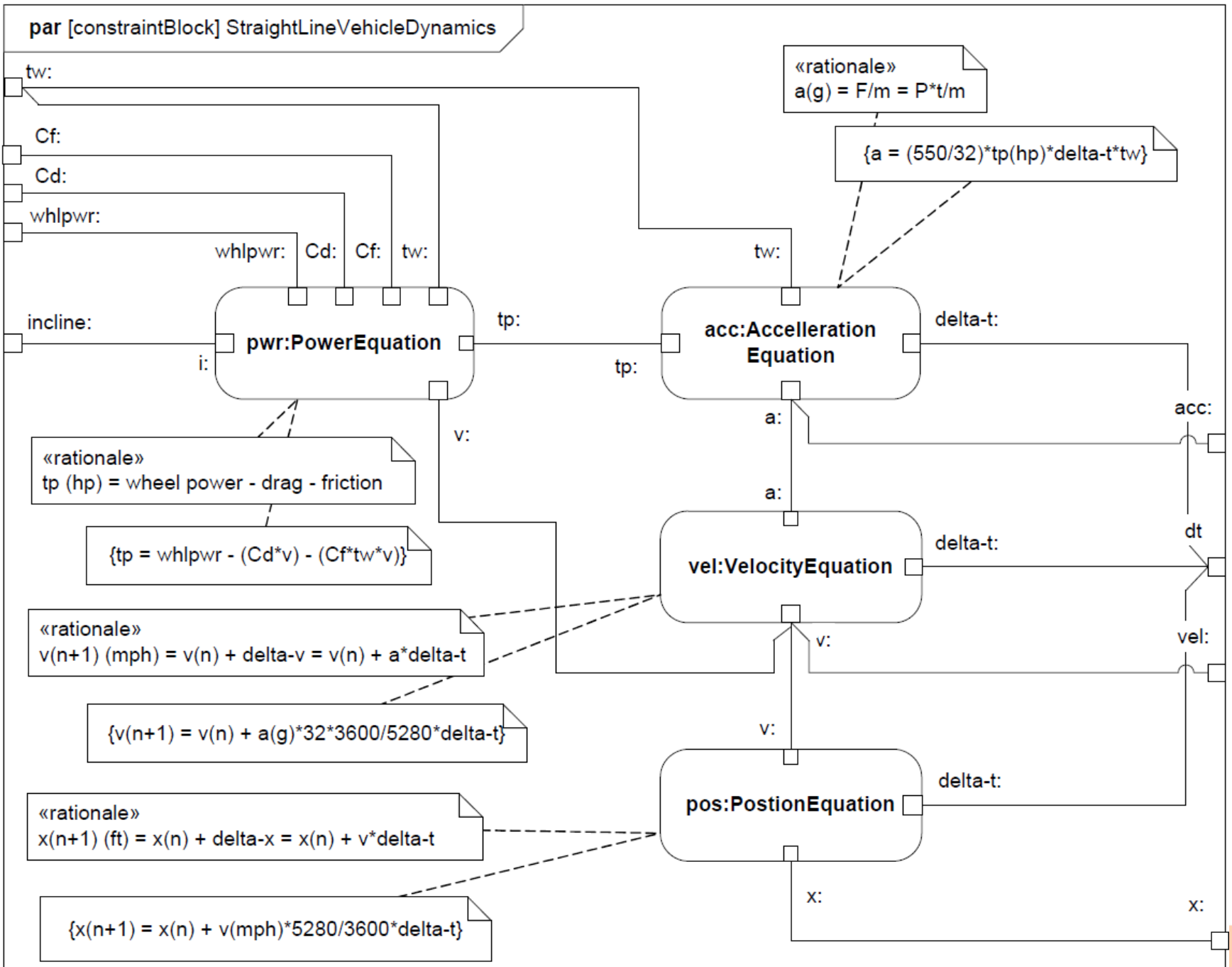


The Requirements and VnV views with supporting views (Package Diagram)


par [block] MeasuresOfEffectiveness [HSUV MOEs]
«moe»
HSUValt1.CostEffectiveness
CE:
:EconomyEquation
f:
«moe»
HSUValt1.FuelEconomy
p1:
p2:
p3:
«objectiveFunction»
:MyObjectiveFunction
{CE = Sum(Wi*Pi)}
«moe»
HSUValt1.QuarterMileTime
:MaxAcceleration
Analysis
q:
z:
«moe»
HSUValt1.Zero60Time
p4:
p5:
:CapacityEquation
vc:
«moe»
HSUValt1.CargoCapacity
:UnitCostEquation
uc:
«moe»
HSUValt1.UnitCost

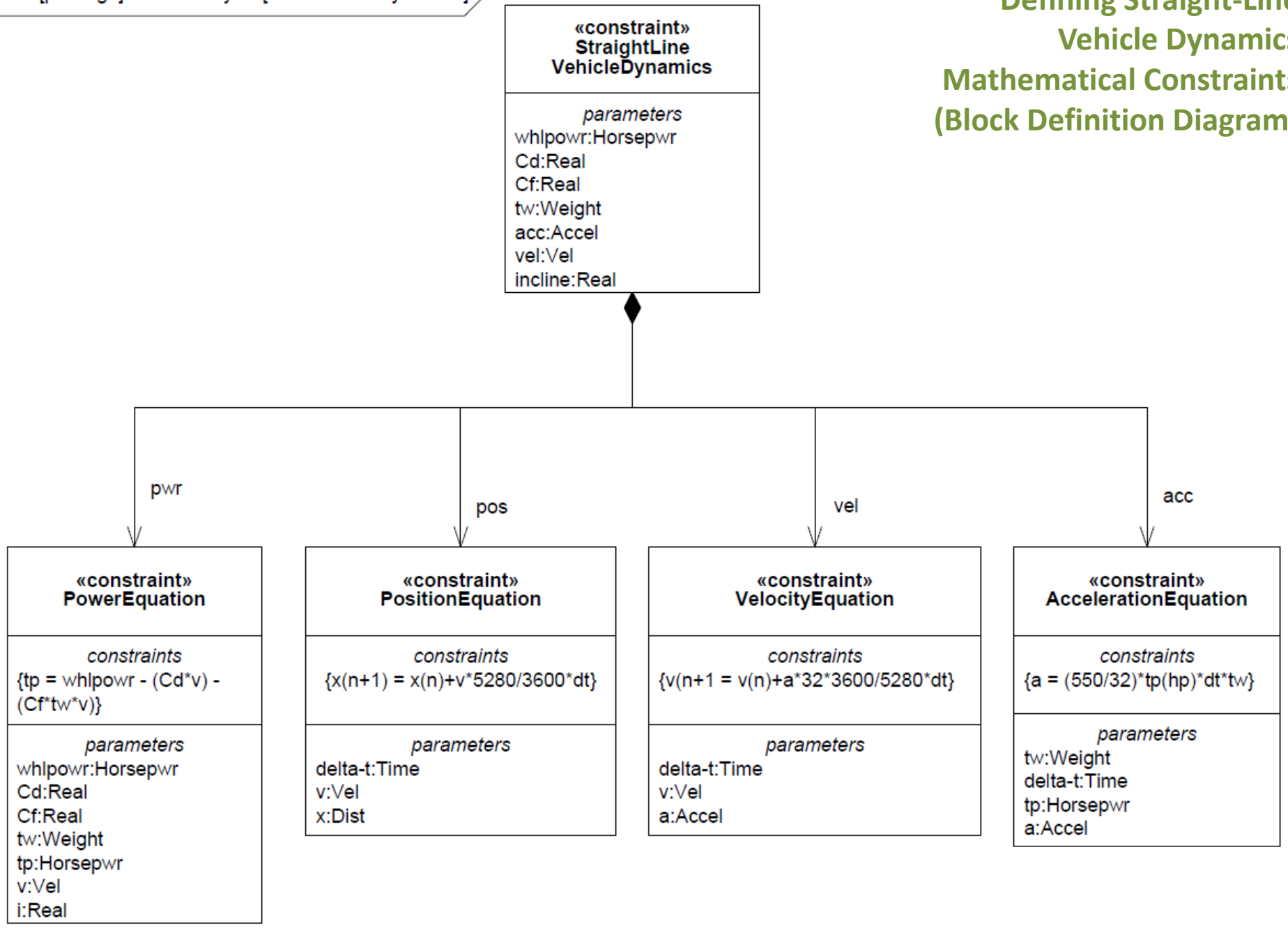


Establishing Mathematical Relationships for Fuel Economy Calculations (Parametric Diagram)



Straight Line Vehicle Dynamics Mathematical Model (Parametric Diagram)

Defining Straight-Line Vehicle Dynamics Mathematical Constraints (Block Definition Diagram)



Design with patterns

Overall Scope

Requirements analysis

System and Software Design

Implementation

Designing **a solution in terms of collaborating software objects.**
I.e. which classes, what methods, what interaction patterns.

Problem

- How to systematically come up with design (structure and responsibilities) of individual building blocks/classes?

Object-Oriented Design

- Goal:
  - Create software objects with responsibilities
  - Create interaction diagrams for the software objects in the system
    - Based on system sequence diagrams
  - Create class diagrams
    - Based on the classes in the domain model

- Note that the three steps above are really done in parallel

Procedure

1. Take an SSD and elaborate each system message with a collaboration/sequence diagram
2. Create necessary software classes and assign them responsibilities
   - Based on domain model conceptual classes

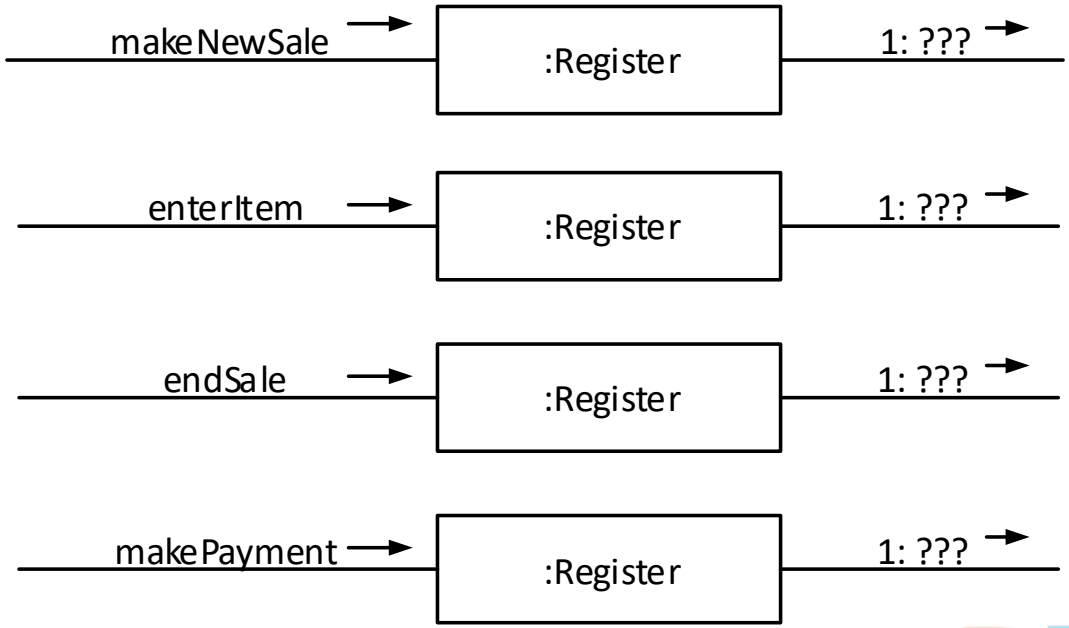
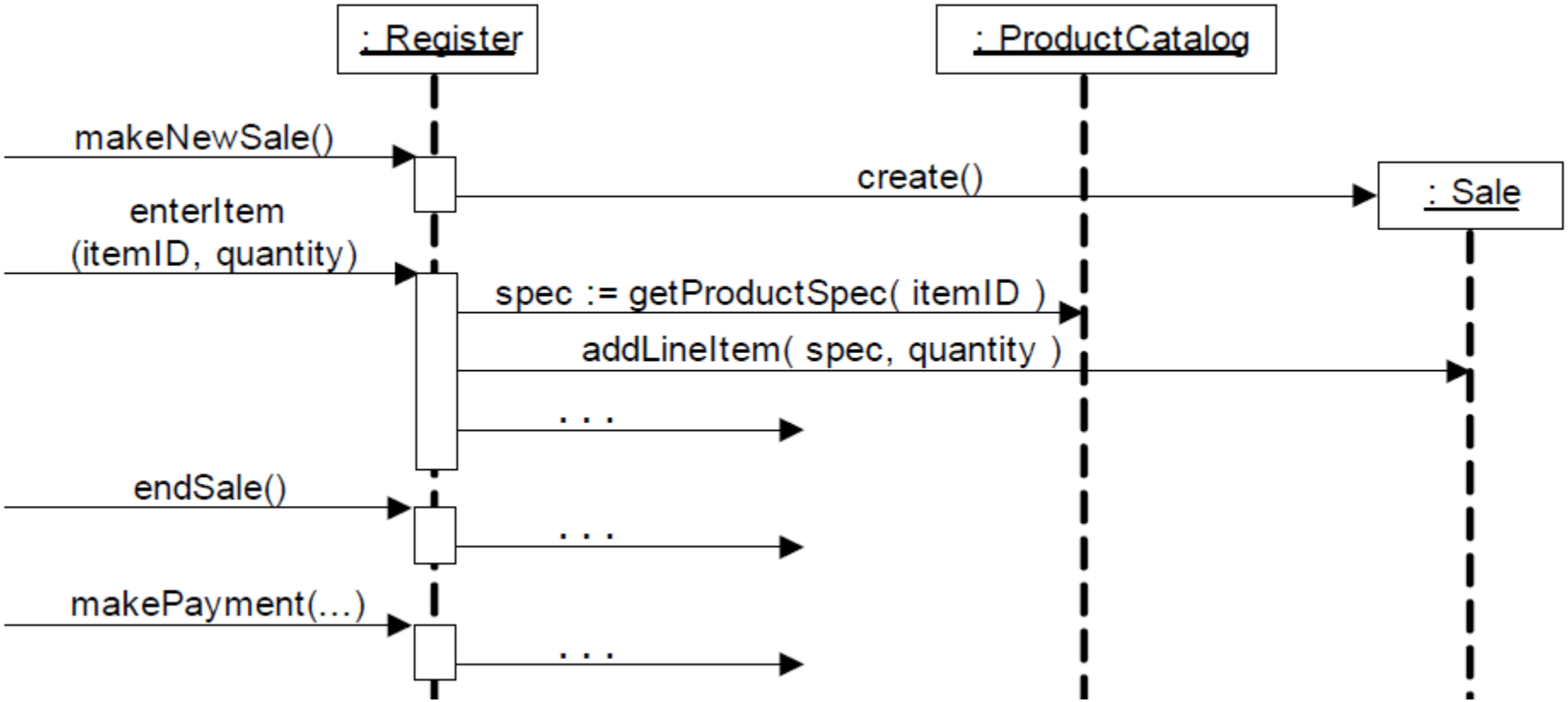


Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources

Procedure



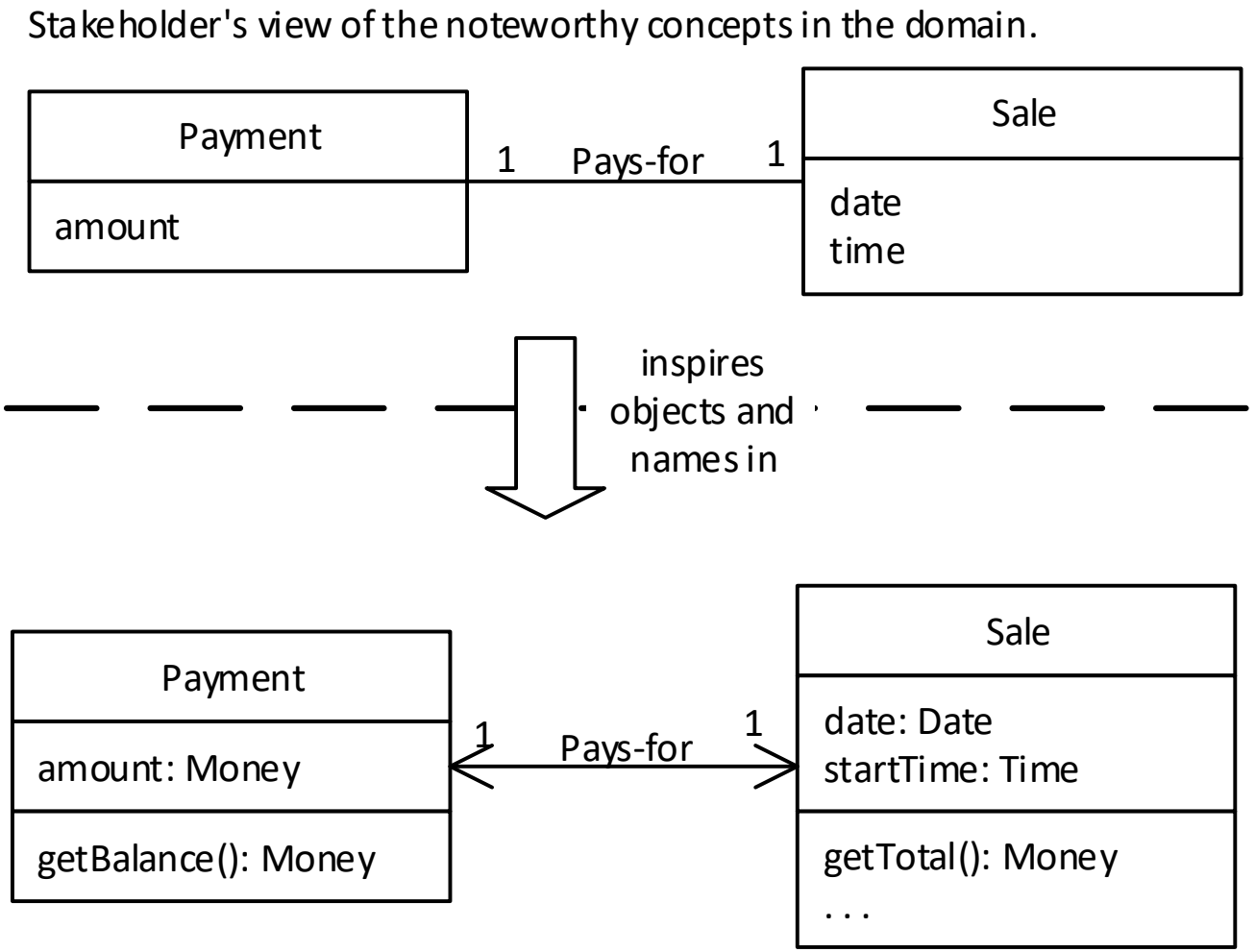
Department of Distributed and Dependable Systems

Figure taken from C. Larman: Applying UML and Patterns (Second Edition), Prentice Hall

Software classes

- Represent domain concepts
  - Physical objects (wheels, other robots, obstacles)
    - Along with the estimation of their state
  - Sensors, actuators
    - Along with methods reading sensors and controlling actuators
  - Virtual objects
    - Map
  - Processes
    - Movement/trajectory

Creating Software Classes



Responsibilities

- Assigning responsibilities to classes
  - “a contract or obligation of a classifier”

- Basically two types of responsibilities
  - Doing
    - doing something itself, such as creating an object or doing a calculation
    - initiating action in other objects
    - controlling and coordinating activities in other objects
    - sensing or actuating
  - Knowing
    - knowing about private encapsulated data
    - knowing about related objects
    - knowing about things it can derive or calculate
    - keeping internal state (of processes – e.g. movement -> position)

Responsibilities

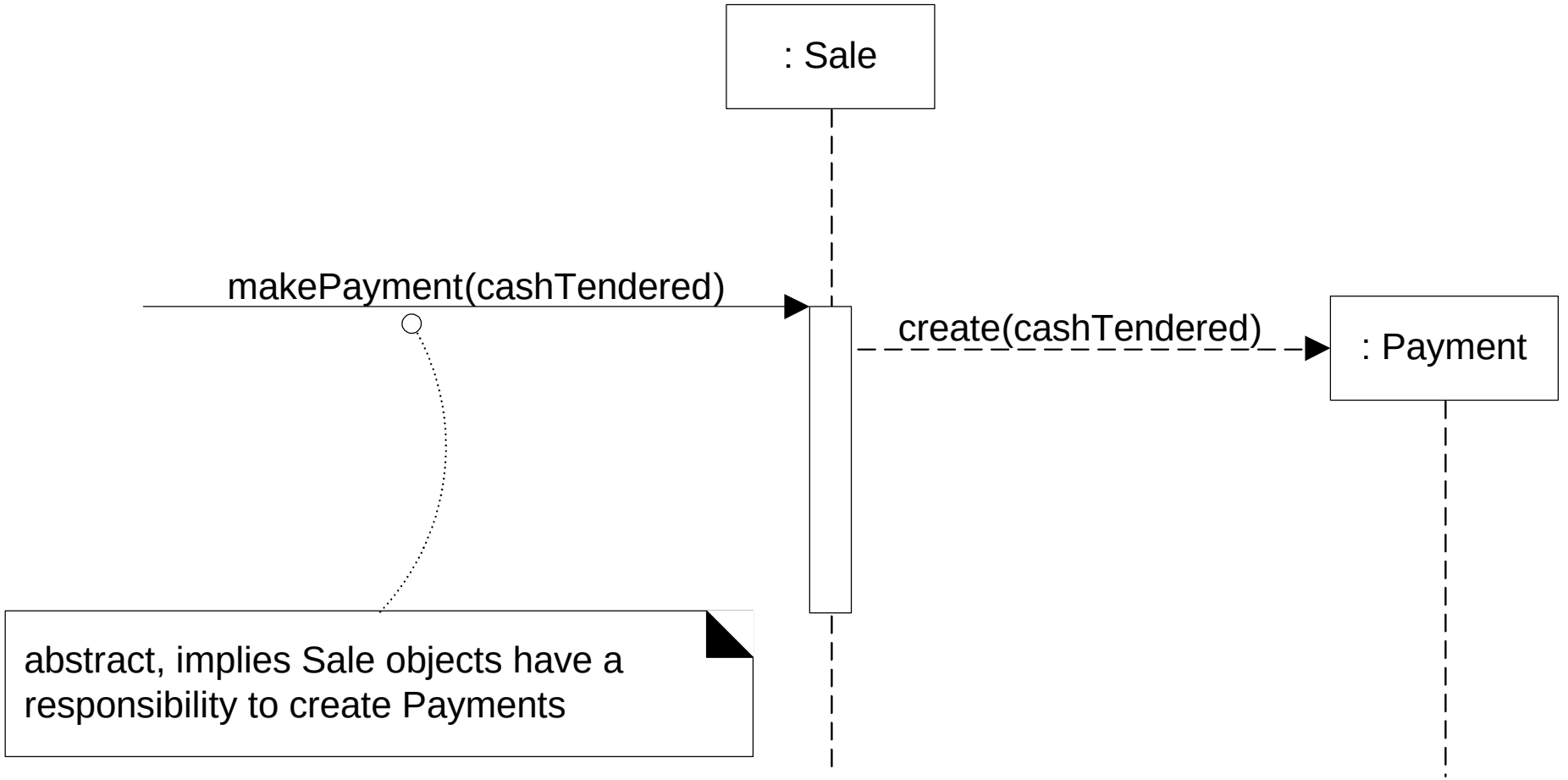


Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources

Assigning Responsibilities

- GRASP patterns
  - principles and idioms that guide the creation of classes and assigning responsibilities to them
  - simplify communication

- GRASP patterns are
  - Information Expert
  - Creator
  - High Cohesion
  - Low Coupling
  - Controller
  - …

Information Expert

Problem

What is a general principle of assigning responsibilities to objects?

Solution

Assign a responsibility to the *information expert* – the class that has the information necessary to fulfill the responsibility

Information Expert – Example

Example

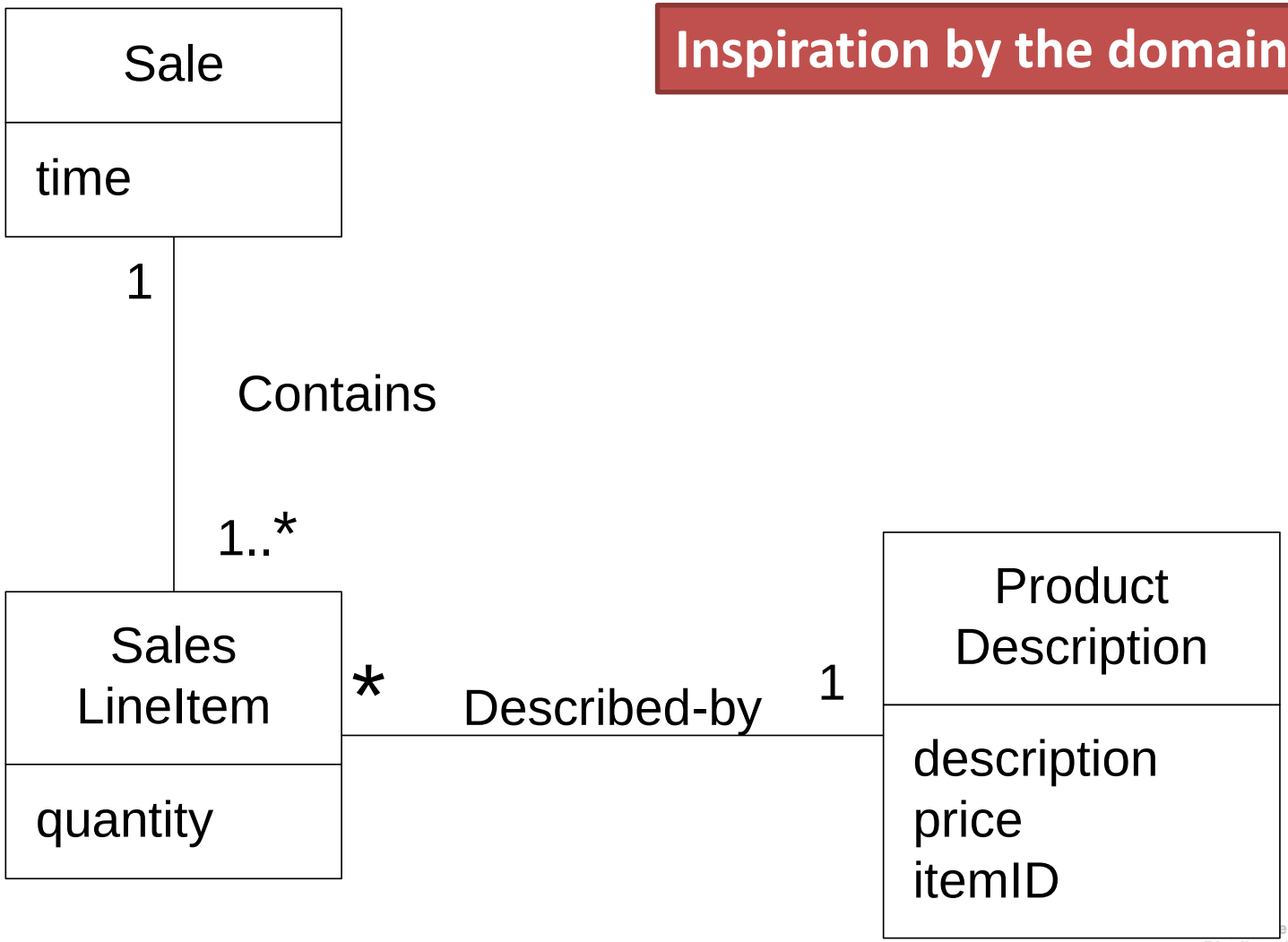
- Who should be responsible for knowing the grand total of a sale?
  - By information expert, it should be the class that has all the information needed – i.e. the class *Sale*

Finding the Right Design Class

- Where to find the class to assign the responsibility to?
  - First look for a suitable class in the design model
  - If there is no suitable class there, look for such a class in the domain model
  - use (or expand) it to inspire the creation of a corresponding design class
- This approach supports *low representational gap*

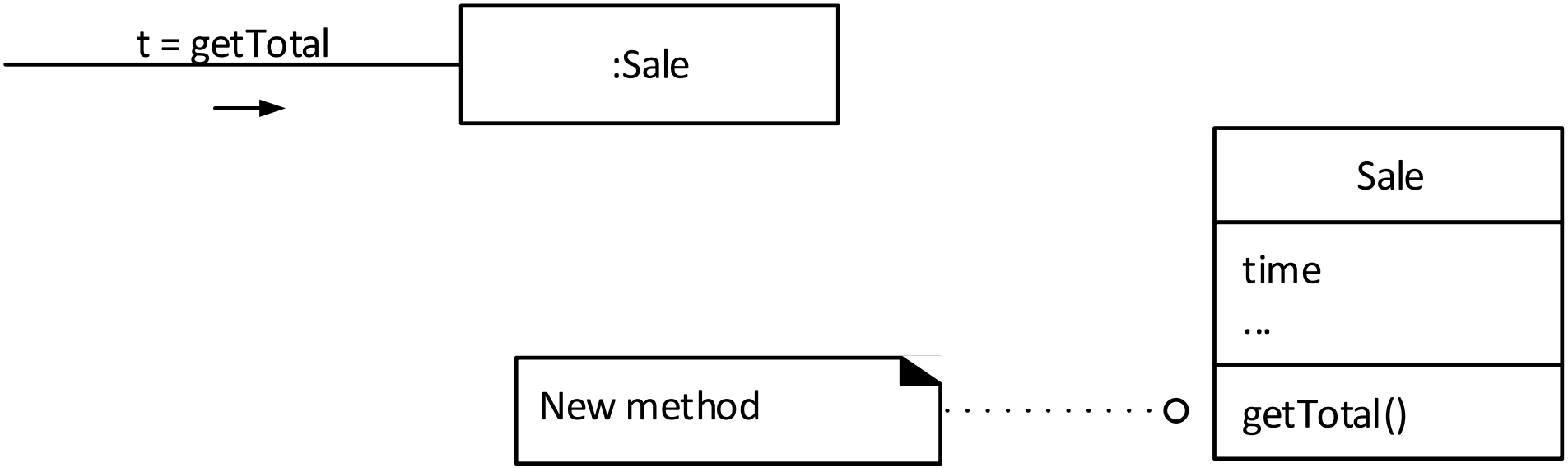
Information Expert – Example contd.

Inspiration by the domain model



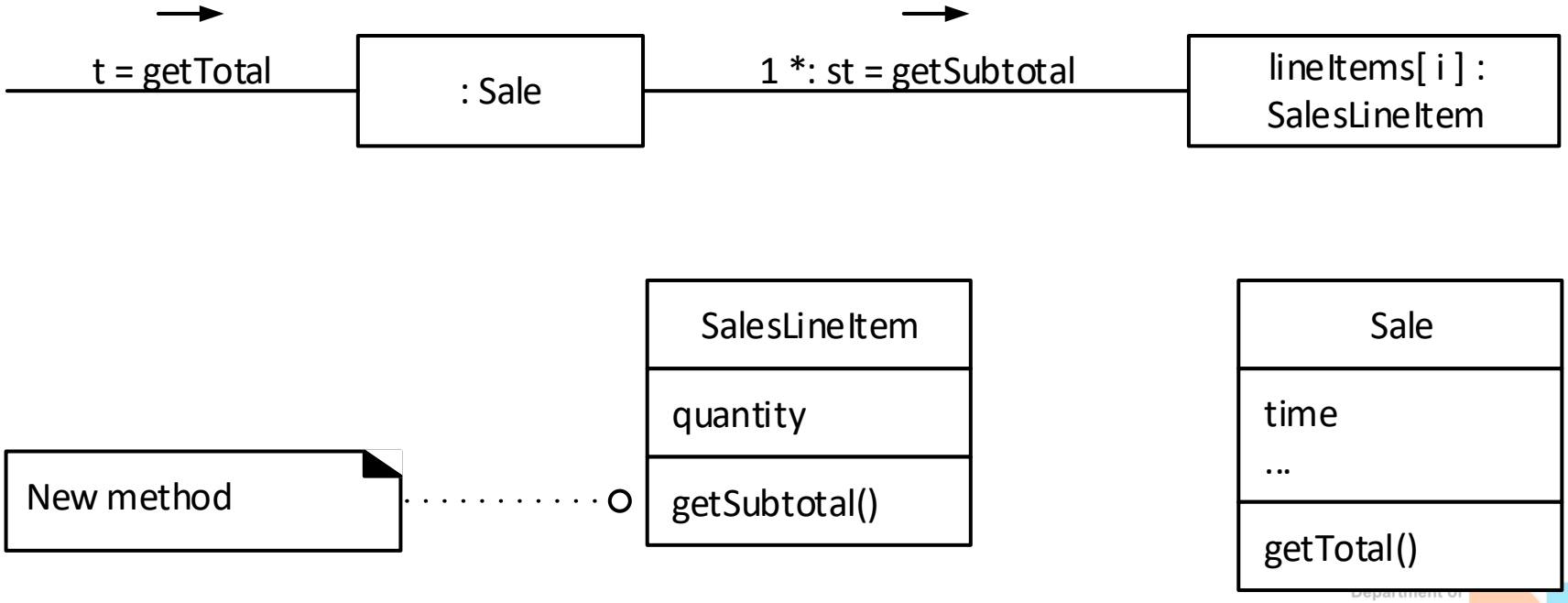
Information Expert – Example contd.

Creation of a partial communication diagram
Creation of a class in the design model
Creation of a method in the Sale class



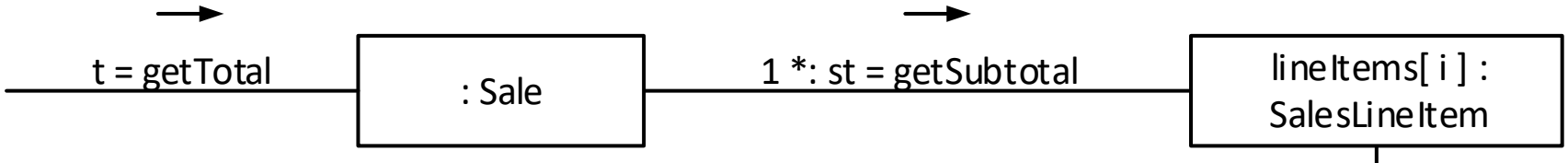
Information Expert – Example contd.

Elaboration of the communication diagram
Creation of an additional class in the design model
Creation of a corresponding method



Information Expert – Example contd.

Further elaboration to complete to communication diagram



Product Description
description price itemID
getPrice()

Sale
time ...
getTotal()

SalesLineItem
quantity
getSubtotal()

Information Expert – Example contd.

- We ended up with this assignment or responsibilities

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

- Note that all of the classes are partial information experts, which cooperate to provide the getTotal method

Information Expert

Contraindications

- Sometimes leads to problems in coupling and cohesion
- E.g. who should be responsible to save a sale into a database?

Benefits

- Information encapsulation is maintained
- Encourages more cohesive “lightweight” classes

Creator

Problem

Who should be responsible for creating a new instance of some class?

Solution

Assign class B the responsibility to create an instance of class A (i.e. B will be a creator of A objects) if one or more of the following is true:

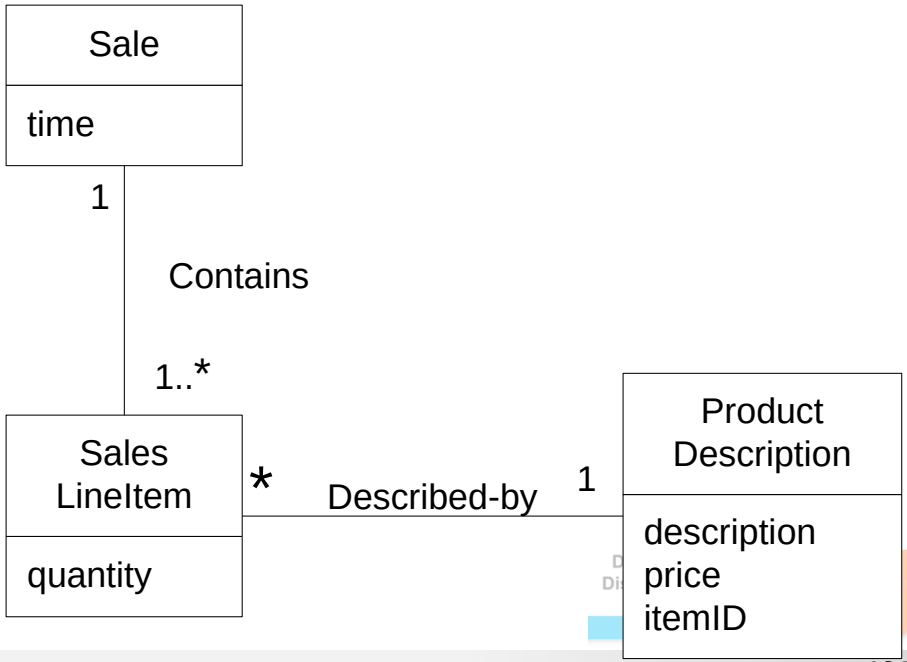
- B aggregates A objects;
- B contains A objects;
- B records instances of A objects;
- B closely uses A objects;
- B has the initializing data that will be passed to A when it is created (thus B is an Expert with respect to creating A).

If more than one option applies, prefer a class B which aggregates or contains class A.

Creator

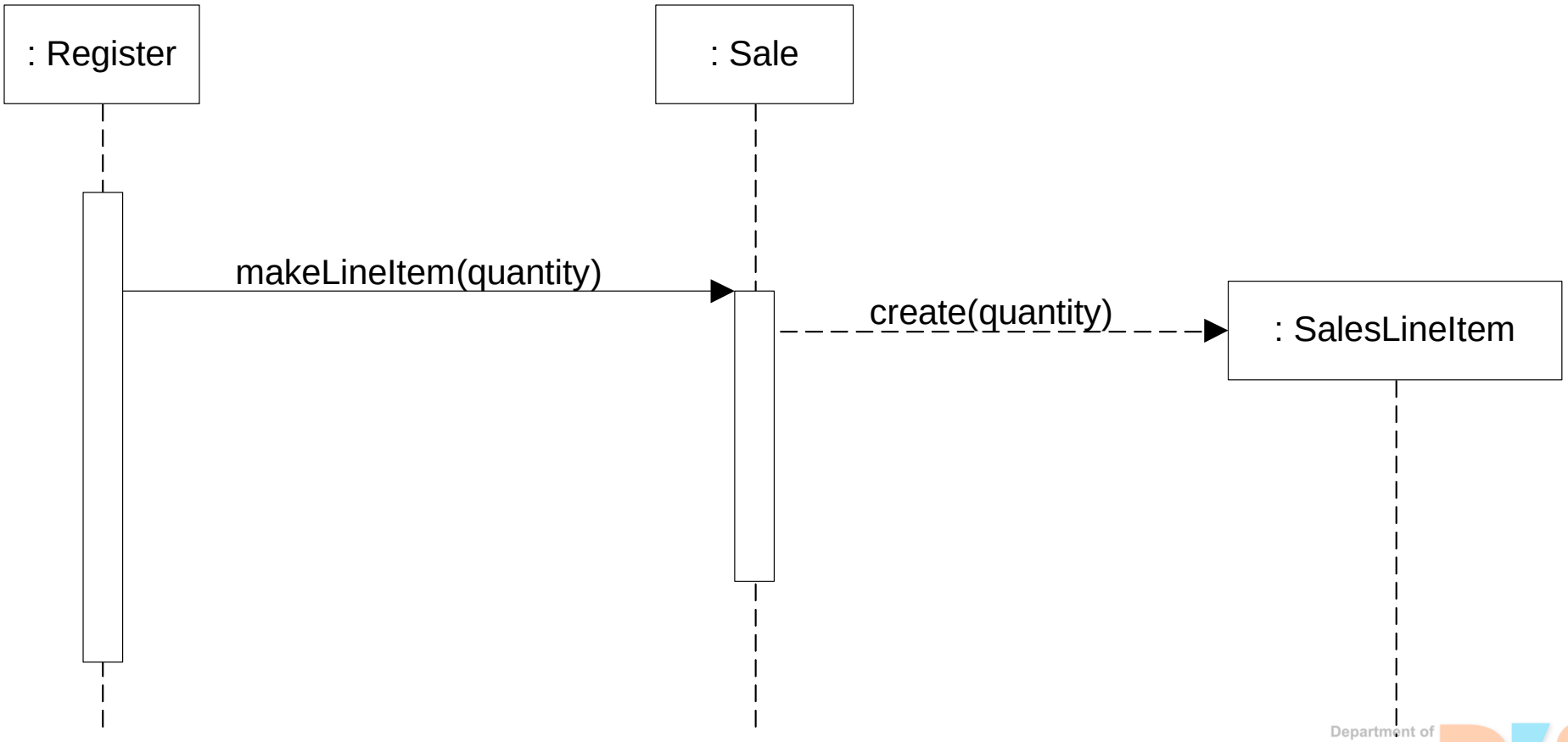
Example

- Who should be responsible for creating a *SalesLineItem* instance?
  - By Creator, we look for a class that aggregates, contains, etc. *SalesLineItem* instances
  - From the domain model we see that *Sale* is a good candidate.



Creator – Example contd.

Leads to the following interations



Creator

Contraindications

- Sometimes the creation may involve recycling object instances (e.g. due to performance), conditional creation of instances based on some external property, etc. In those cases it is better to use a separate *Factory* class

Benefits

- Maintains low coupling since the creator anyway has all the knowledge of the object it creates

Low Coupling

Problem

How to support low dependency, low change impact, and increased reuse?

Solution

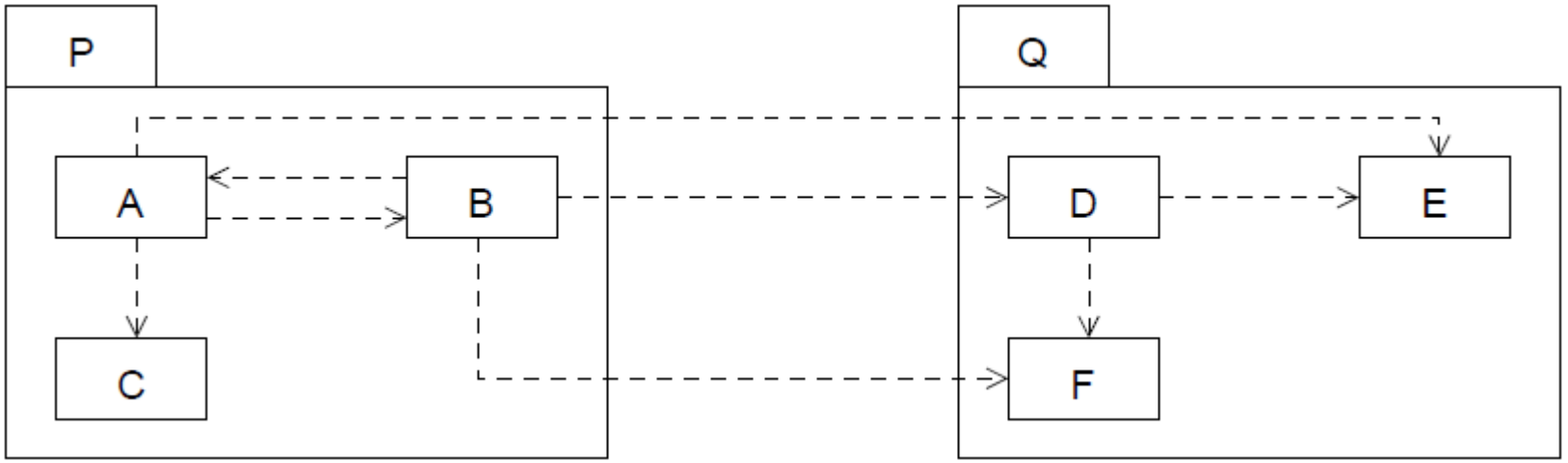
Assign a responsibility so that coupling remains low

Coupling

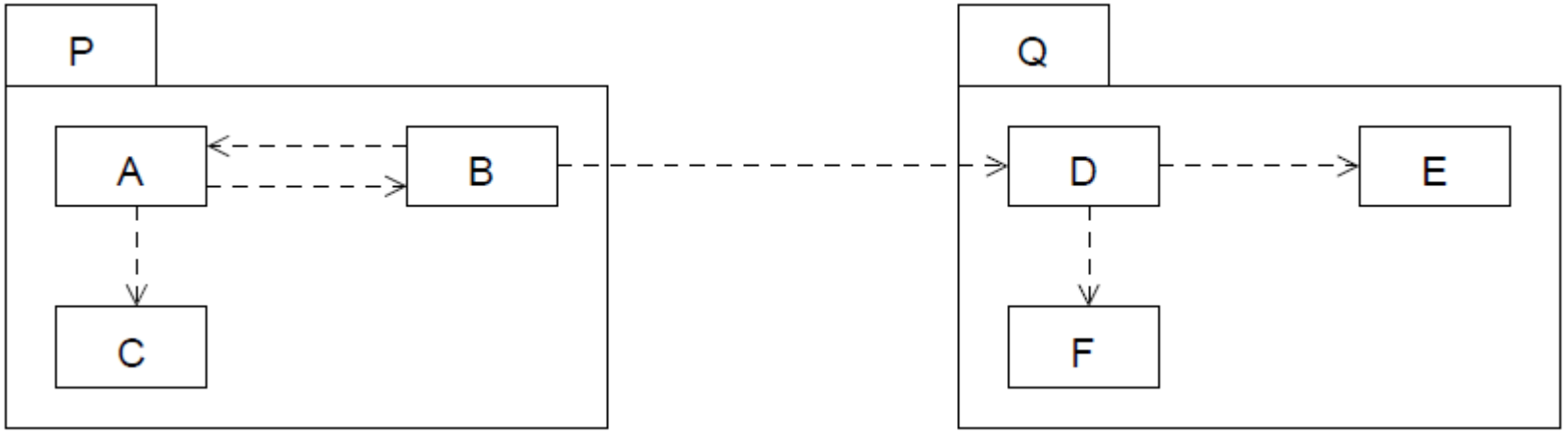
- **Coupling** – a measure of how strongly one element is connected to, has knowledge of, or relies on other elements

- An element with low (or weak) coupling is not dependent on too many other elements.
- A class with high (or strong) coupling relies on many other classes.
- High-coupling leads to problems such as:
  - Changes in related classes force local changes
  - Harder to understand in isolation
  - Harder to reuse because its use requires the additional presence of the classes on which it is dependent

High coupling



Low coupling



Coupling

- In common object-oriented languages, the coupling comes from:
  - *TypeA* has an attribute (data member or instance variable) that refers to a *TypeB* instance, or *TypeB* itself
  - A *TypeA* object calls on services of a *TypeB* object
  - *TypeA* has a method that references an instance of *TypeB,* or *TypeB* itself, by any means
    Typically include a parameter or local variable of type *TypeB,* or the object returned from a message being an instance of *TypeB*
  - *TypeA* is a direct or indirect subclass of *TypeB*
  - *TypeB* is an interface, and *TypeA* implements that interface

Low Coupling – Example

Example

- Which class(es) should be responsible for creating a *Payment* and associating it with *Sale*?

- By Information Expert, the *Register* should create the *Payment*. Then it would associate it with *Sale*.

Couples Register with the knowledge of Payment class!

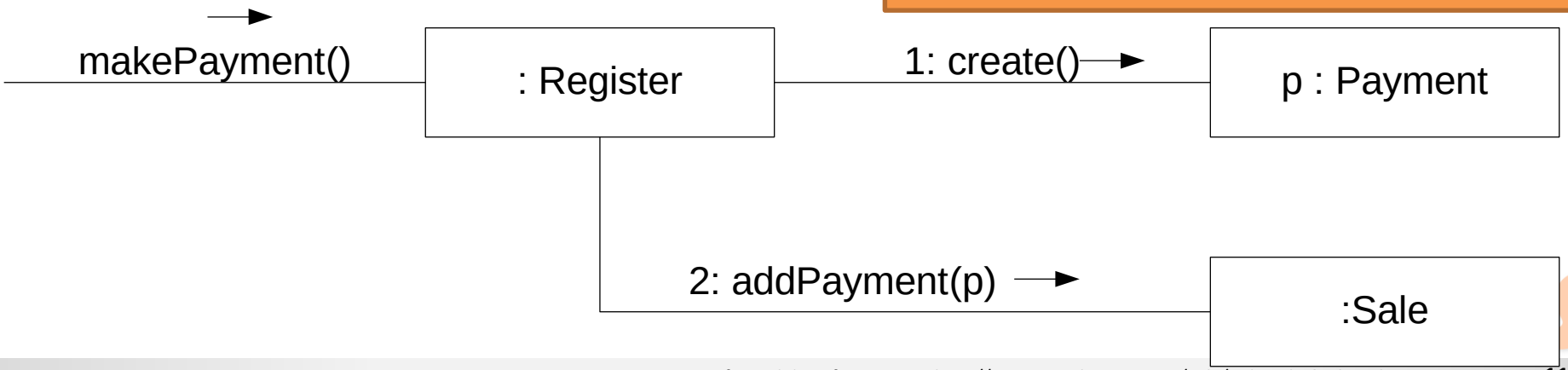
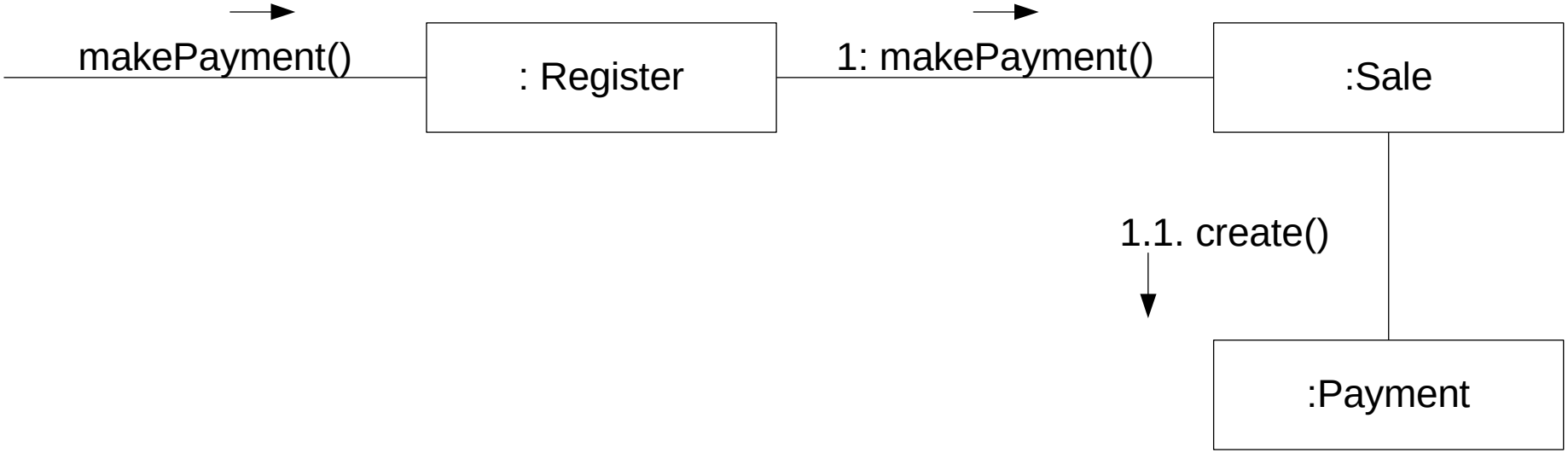


Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources

Low Coupling – Example

- Better solution from the perspective of Low coupling is to delegate the creation to the *Sale* class because it knows about the *Payment* anyway (it aggregates it)



Low Coupling

Contraindications

- High-coupling is a problem only when depending on types which are potentially unstable or where variability is required.
- It is contraproductive in cases such as depending on standard Java API
  - In such cases direct use of the API (i.e. high-coupling) yields a better readable and simpler code

Benefits

- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse

High Cohesion

Problem

How to keep complexity manageable?

Solution

Assign a responsibility so that cohesion remains high

Cohesion

- **Cohesion** – a measure of how strongly related and focused the responsibilities of an element are

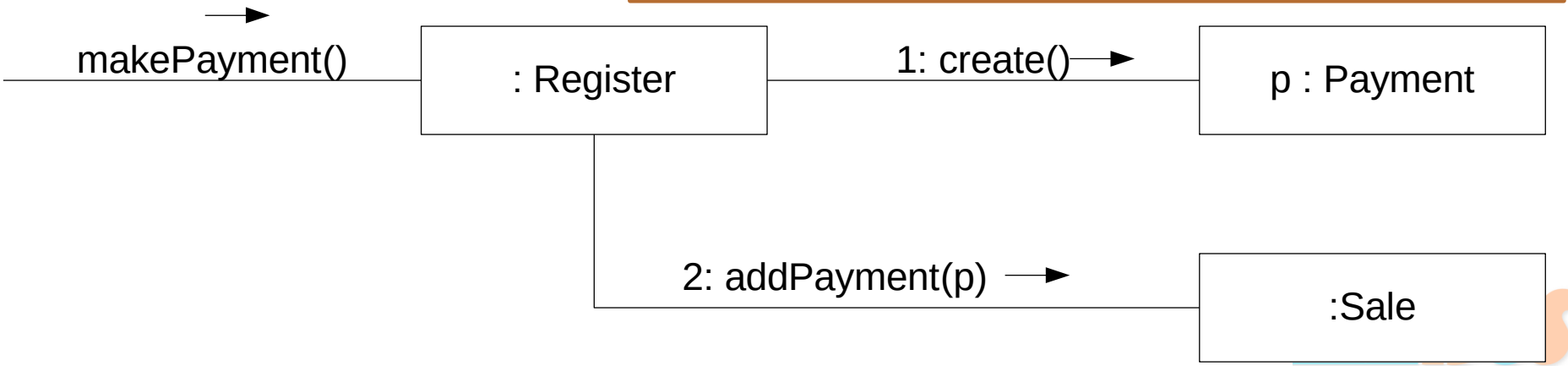
- An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion
- A class with low cohesion does many unrelated things, or does too much work
- Low-cohesion leads to classes which are :
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - constantly effected by change

High Cohesion – Example

Example (similar to the Low-Coupling Example)

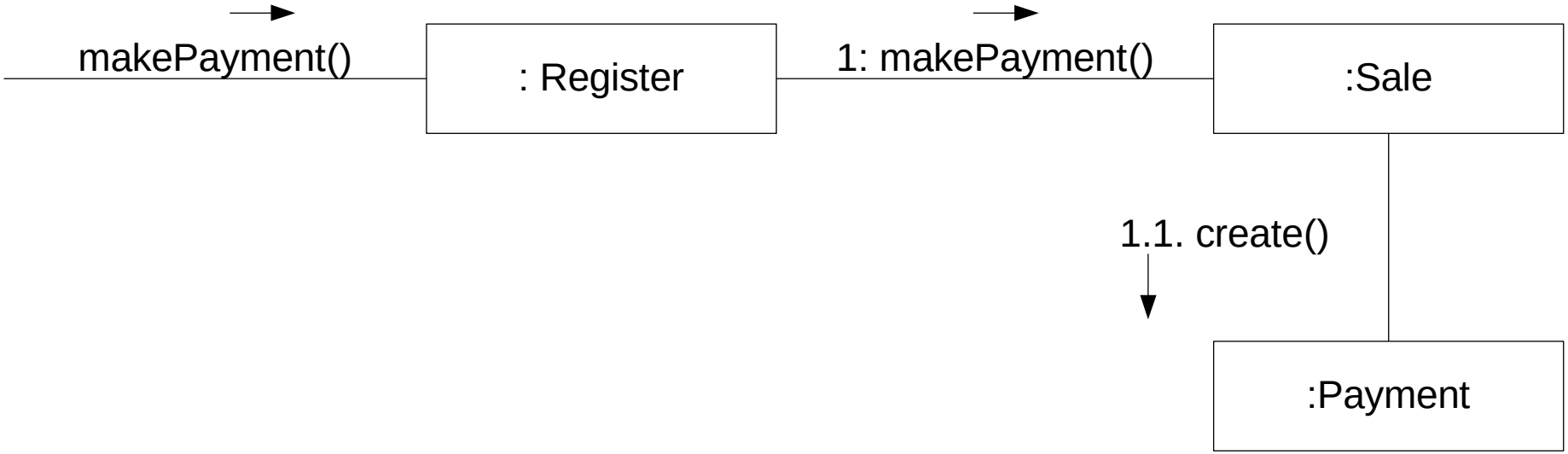
- Which class(es) should be responsible for creating a *Payment* and associating it with *Sale*?

Over time if we keep using the Information Expert / Creator patter, the Registry would get many unrelated methods, which will lower its cohesion.



High Cohesion – Example

- Better solution from the perspective of High cohesion



Cohesion – Examples

- *Very low cohesion*
  - Class is responsible for many things in very different functional areas.
  - E.g. *RDB-RPC-Interface* completely responsible for interacting with relational databases and for handling remote procedure calls.

- *Low cohesion*
  - Class responsible for a complex task in one functional area.
  - E.g. *RDBInterface* completely responsible for interacting with relational databases.

Cohesion – Examples

- *Moderate cohesion*
  - Class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept, but not to each other.
  - E.g. *Company* which is completely responsible for (a) knowing its employees and (b) knowing its financial information.
- *High cohesion*
  - Class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
  - E.g. *RDBInterface* which is only partially responsible for interacting with relational databases. Interacts with a dozen other classes related to RDB access in order to retrieve and save objects.

High Cohesion

Contraindications

- Sometimes it is better to provide a class that encapsulates all behavior related to some particular technology. Such class is then not very cohesive.
  - E.g. RDB mapper – implements storing and retrieving data to/from database
- Sometimes data have to be handle more coarse grained in order to achieve performance
  - E.g. RPC interfaces – would have setData instead of having setFirstName, setSurname, setSalary, etc.

Benefits

- Clarity and ease of comprehension of the design is increased
- Maintenance and enhancements are simplified
- Low coupling is often supported

Controller

Problem

Who should be responsible for handling an input system event?

Solution

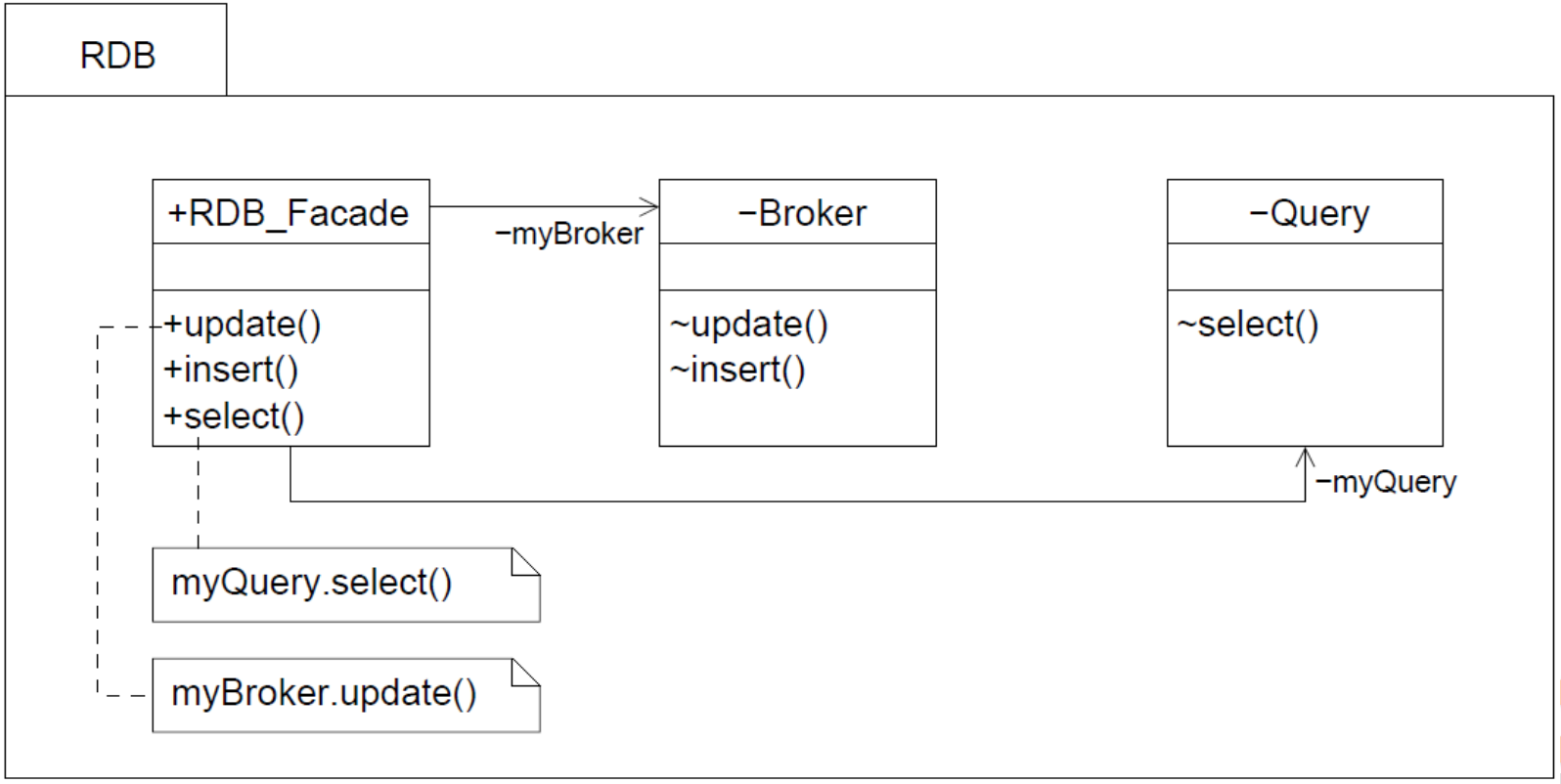
Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system, device, or subsystem *(facade controller)*
- Represents a use case scenario within which the system event occurs *(use-case or session controller);*
  - Often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session
  - Use the same controller class for all system events in the same use case scenario

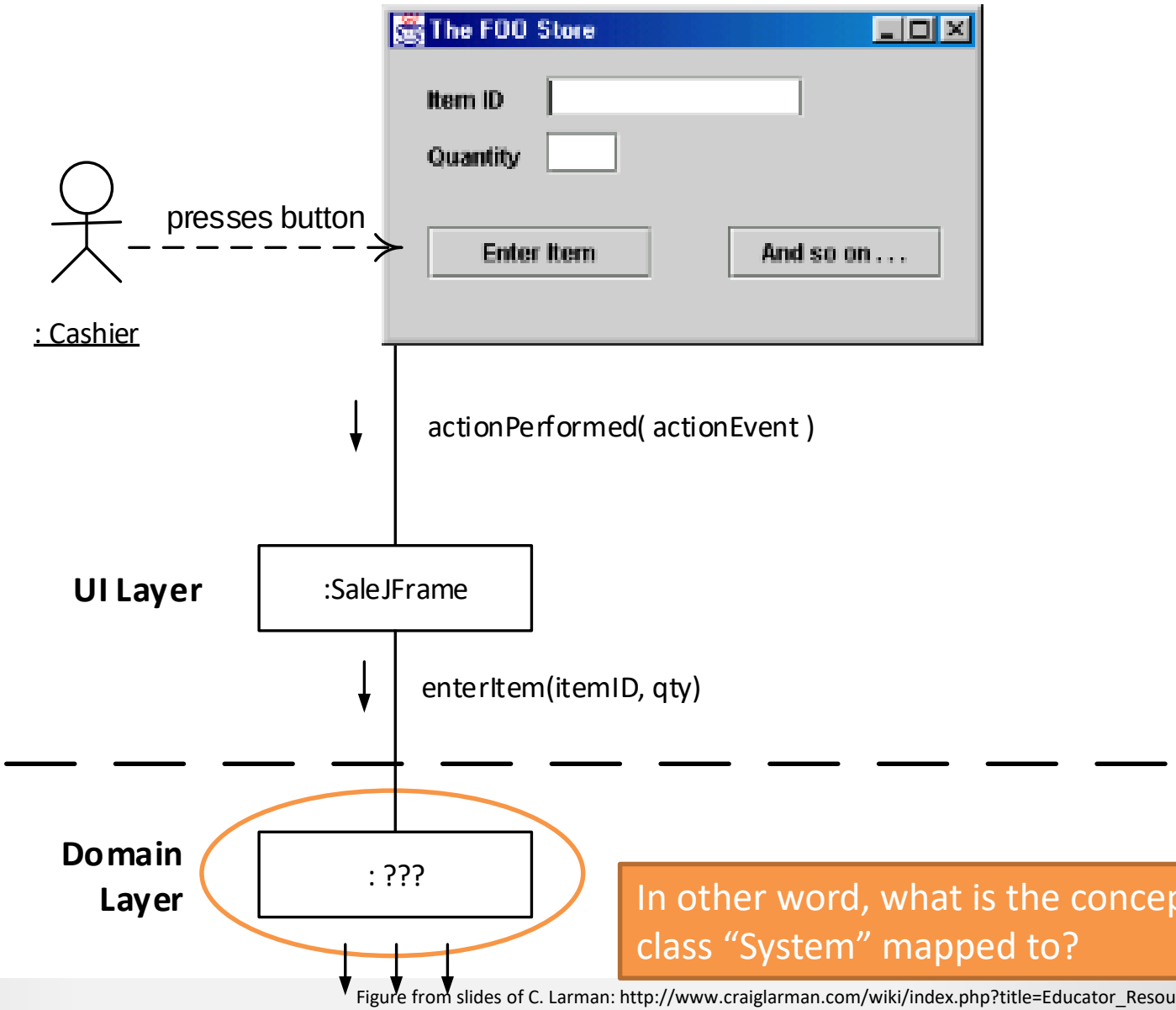
Note that “window”, “widget”, “view”, etc. are not on this list. They should rather delegate the events to the controller.

Façade

- Controller often acts as a façade class
  - Groups together the functionality of several classes of a subsystem and delegates calls to the appropriate objects.



Controller – Example

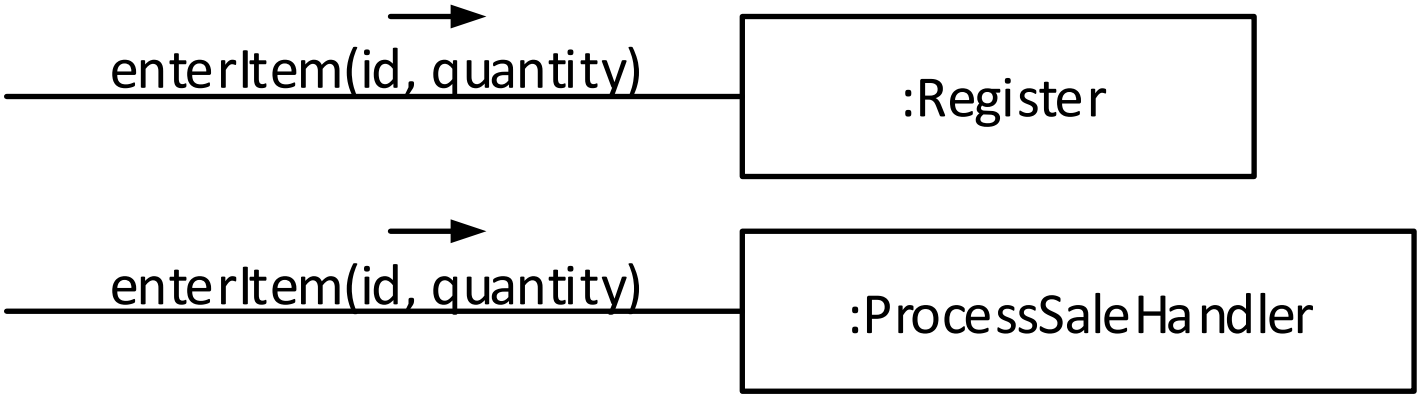


In other word, what is the conceptual class "System" mapped to?

Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources

Controller – Example

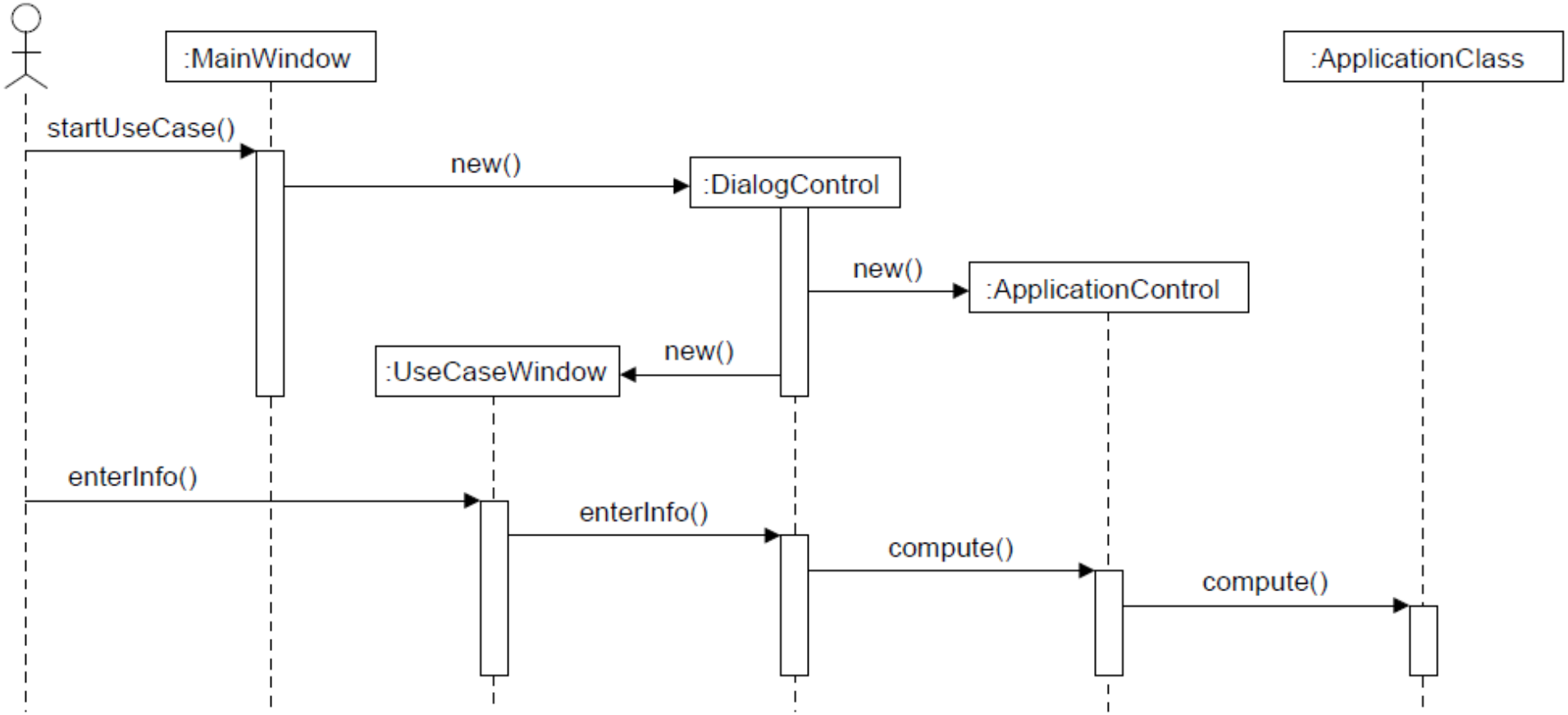
- By Controller pattern, there are several options
  - *Register, POSSystem* – the overall system, device, or subsystem
  - *ProcessSaleHandler, ProcessSaleSession* – a receiver or handler of all system, events of a use case scenario



The controller further delegates the work to other objects.
It coordinates and controls the activity.
But it does not do much work itself.

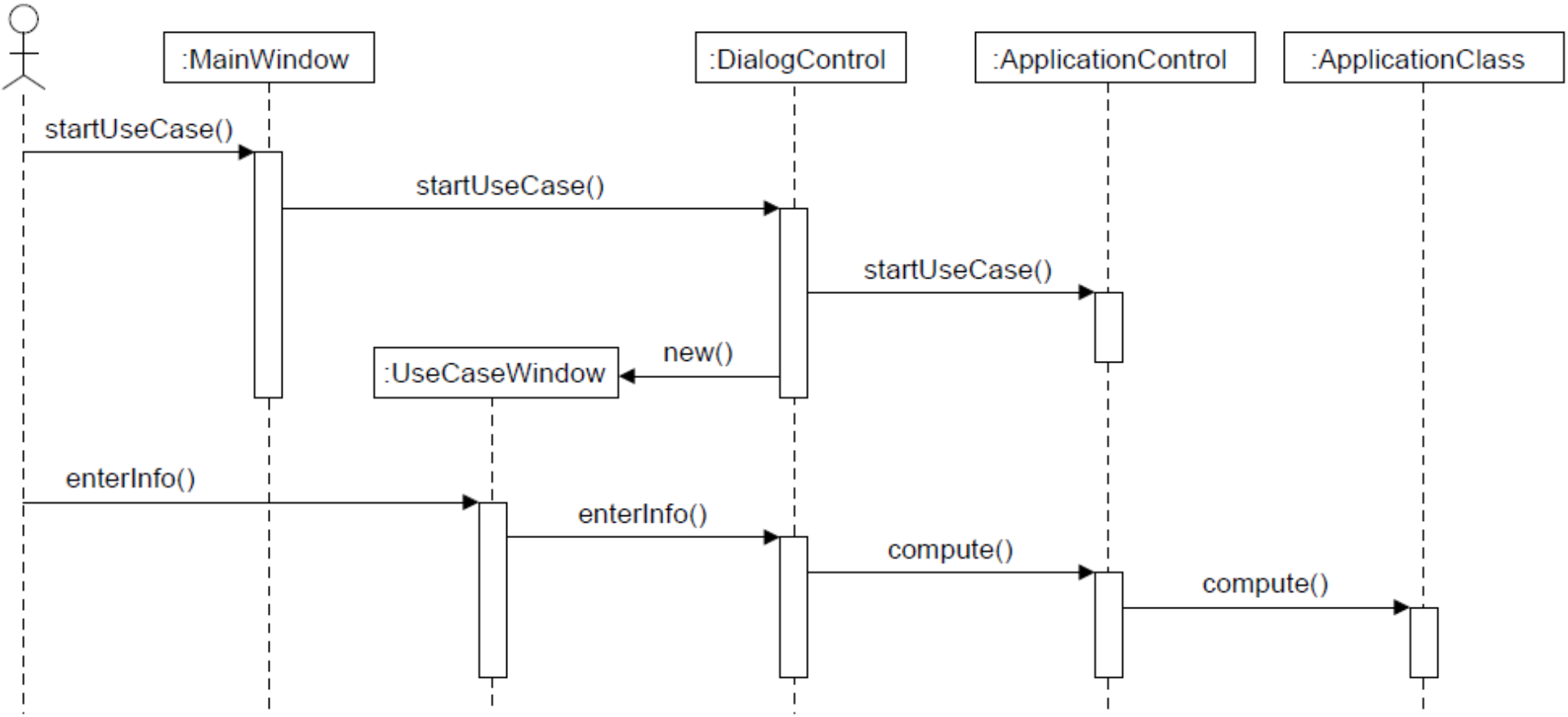
Multiple Controllers

Typical Interaction Pattern with Control Objects



Multiple Controllers

Variant: Typical Interaction Pattern with Control Objects



Controller

Discusssion

- Façade controller chosen when no too many system events
- Use-case controller when the façade controller would be too bloated

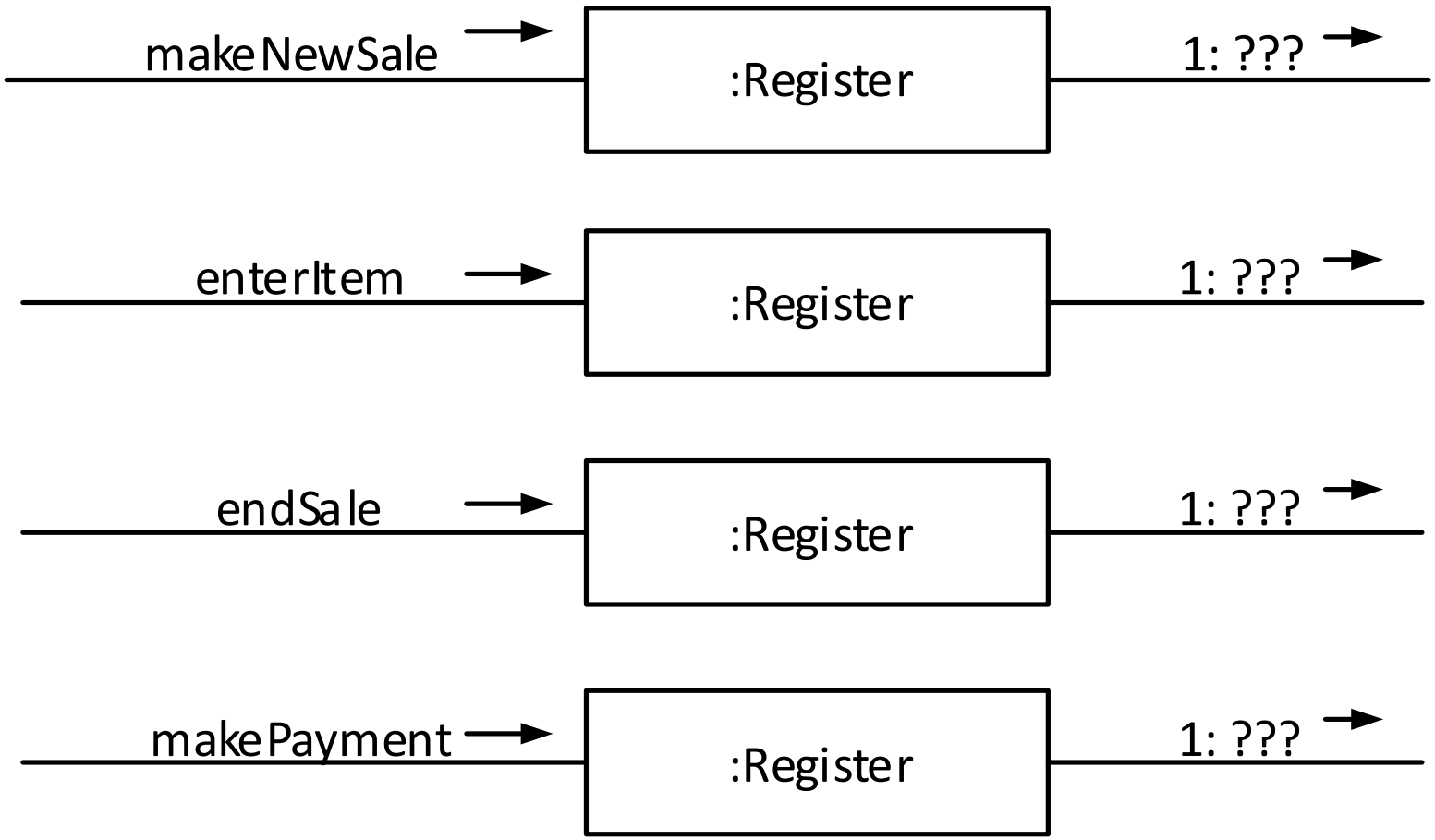
Use-case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

Benefits

- Pluggable interfaces
  - Application logic is separate from the presentation
- Easy to reason about a state of a use-case
  - The controller represents a state-machine of a use-case

POS System – Examples

SSD Events


makeNewSale
:Register
1: ???
enterItem
:Register
1: ???
endSale
:Register
1: ???
makePayment
:Register
1: ???

SSD Events

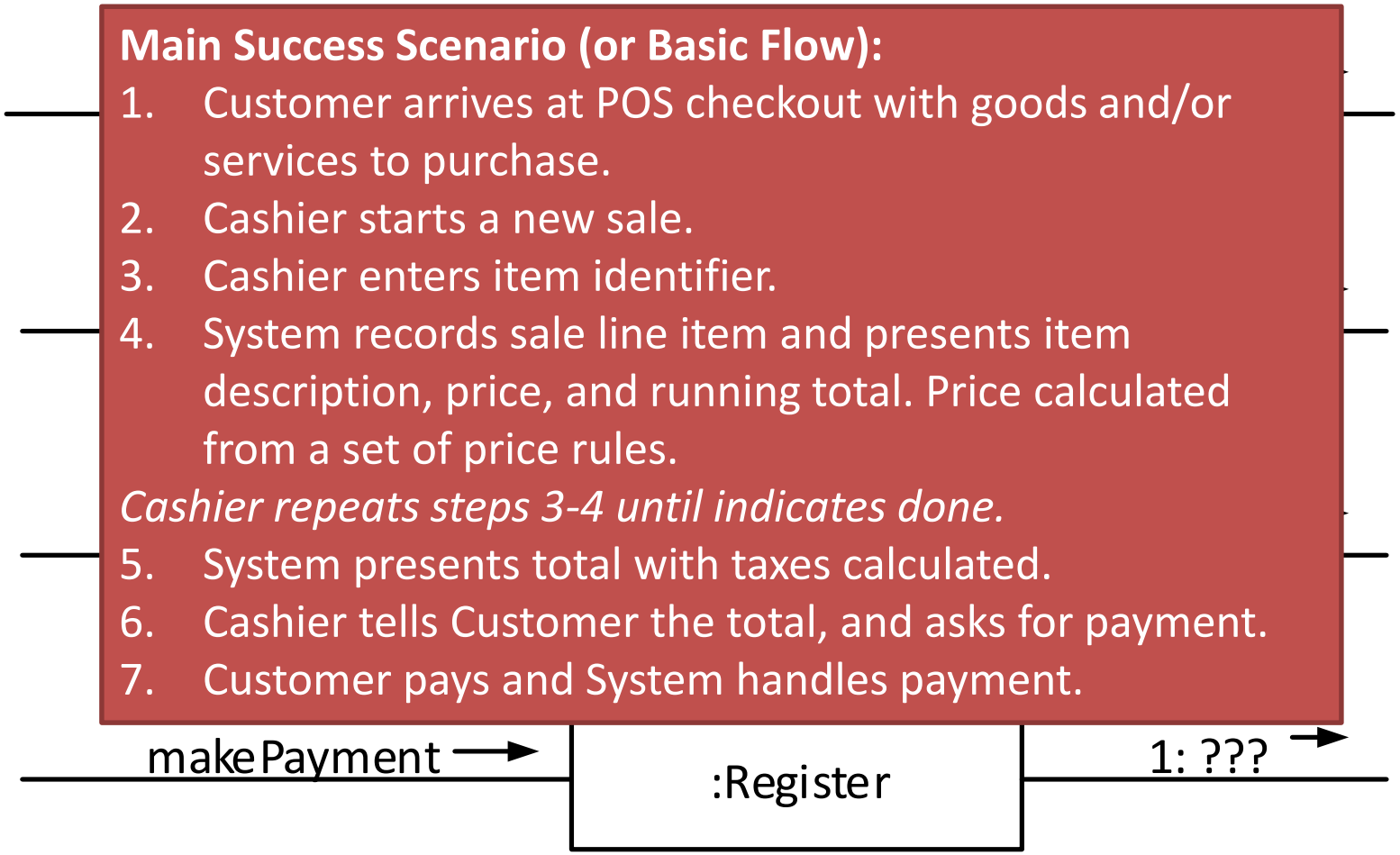
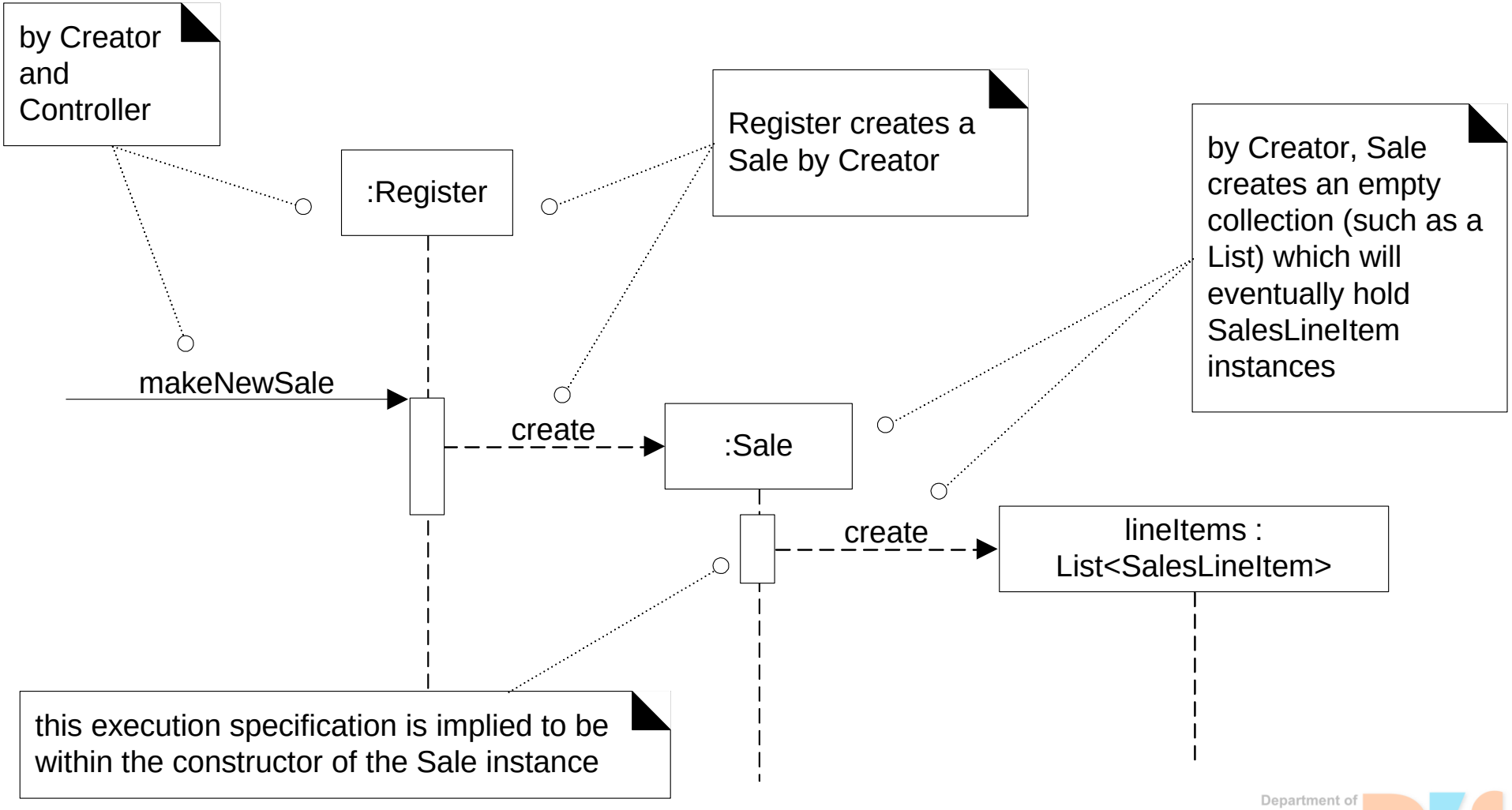
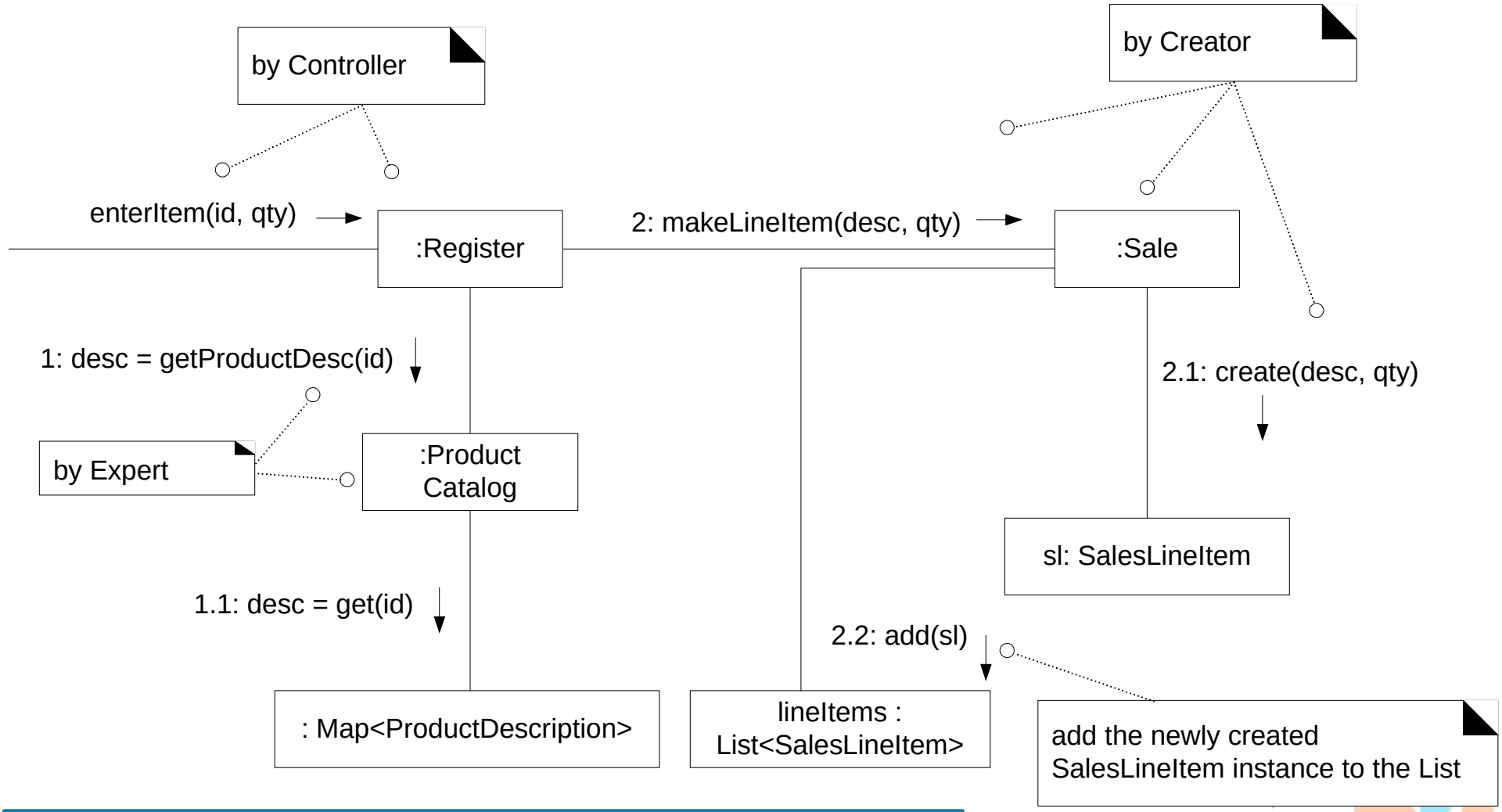


Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources

Creating New Sale



Entering an Item



Note that Register must have visibility to the ProductCatalog

Distributed and Dependable Systems

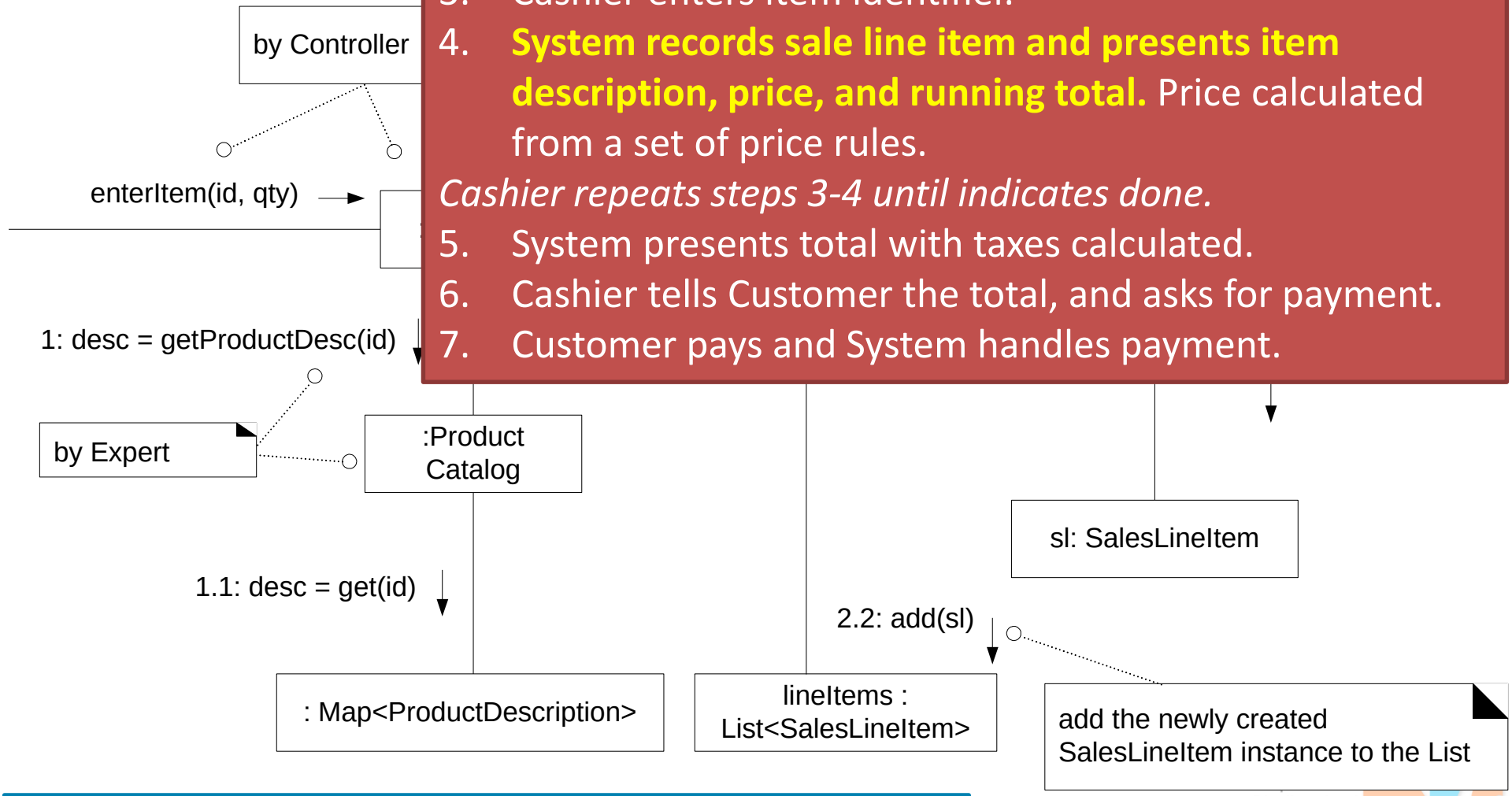
Entering ar

Main Success Scenario (or Basic Flow):

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. **System records sale line item and presents item description, price, and running total.** Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.



Note that Register must have visibility to the ProductCatalog

Ending Sale

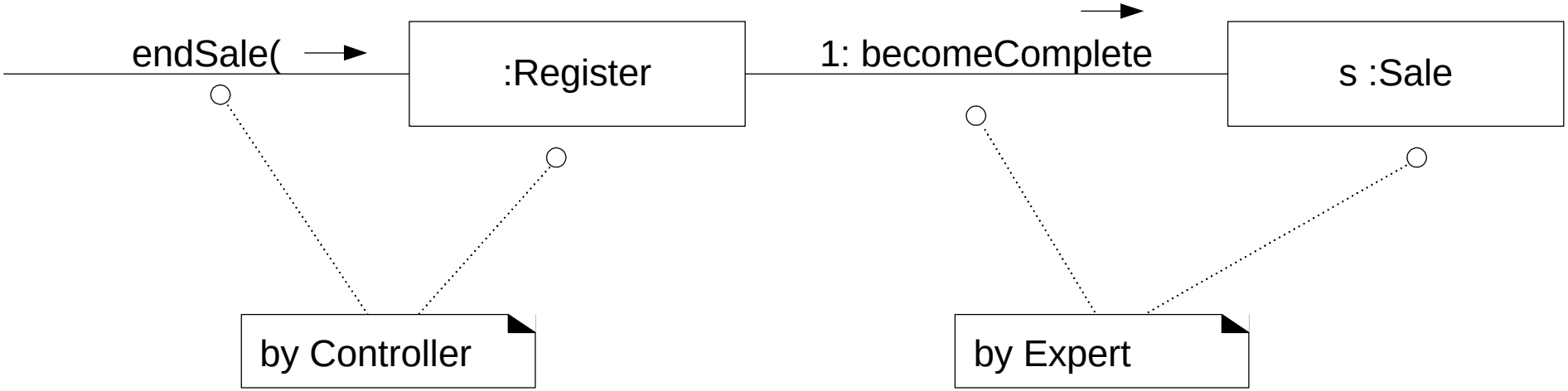
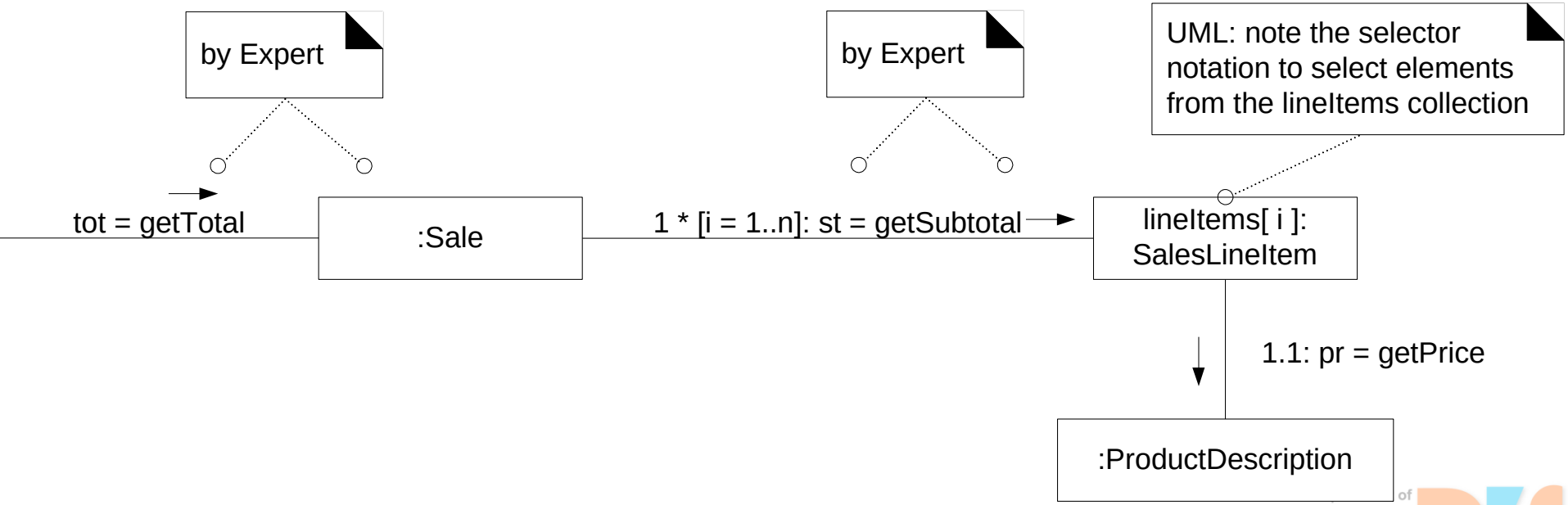


Figure from slides of C. Larman: http://www.craiglarman.com/wiki/index.php?title=Educator_Resources

Calculating Sale Total



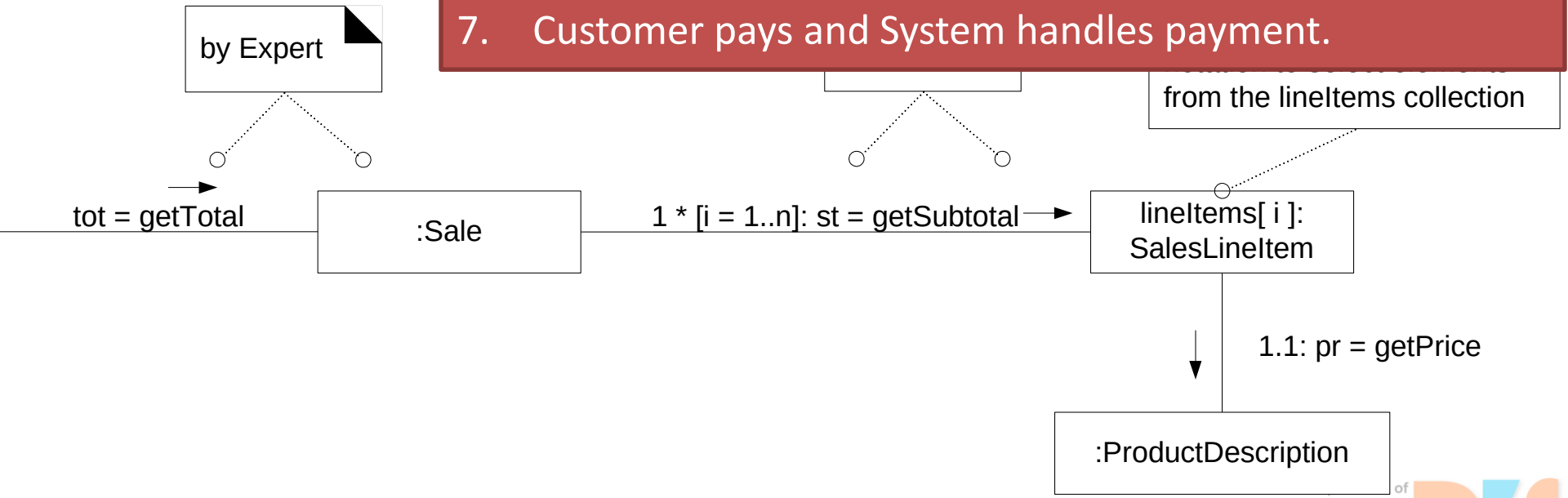
Calculating

Main Success Scenario (or Basic Flow):

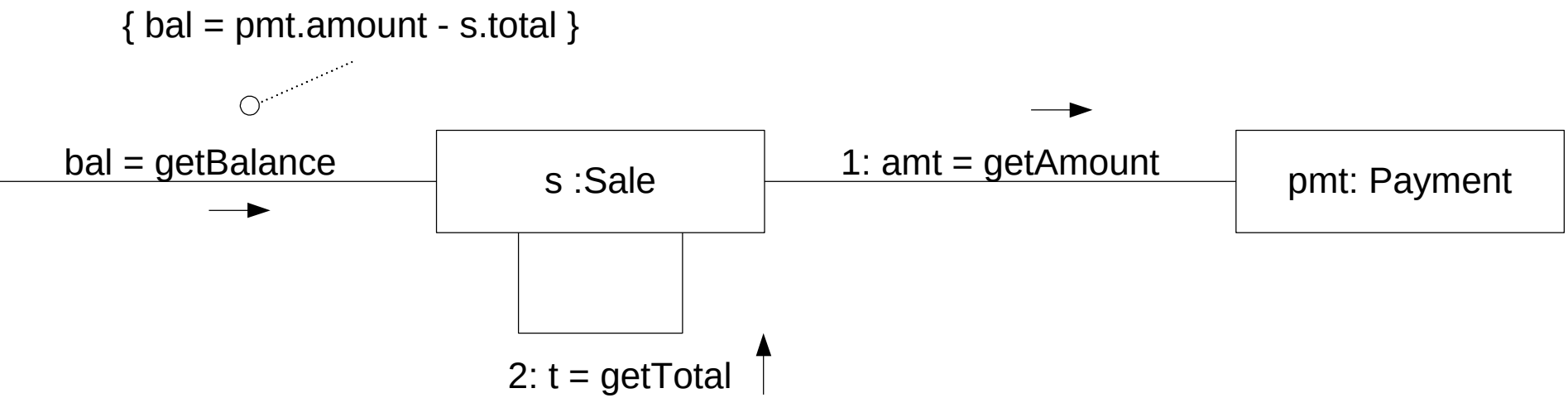
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. **System presents total with taxes calculated.**
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.

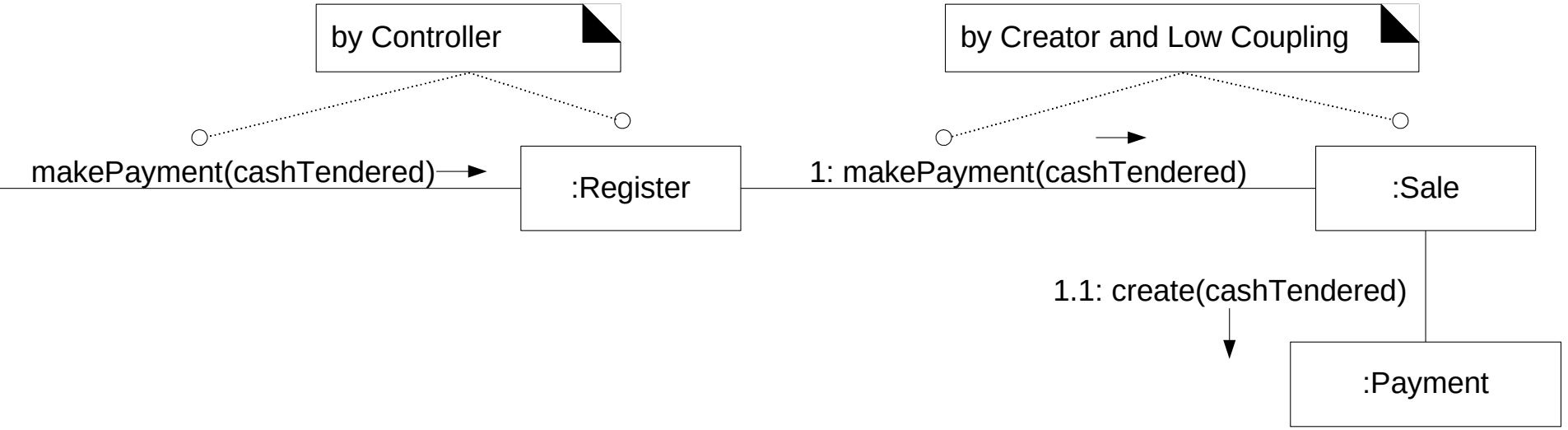


Calculating the Balance



7a. Paying by cash:
1. Cashier enters the cash amount tendered.
2. **System presents the balance due**, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

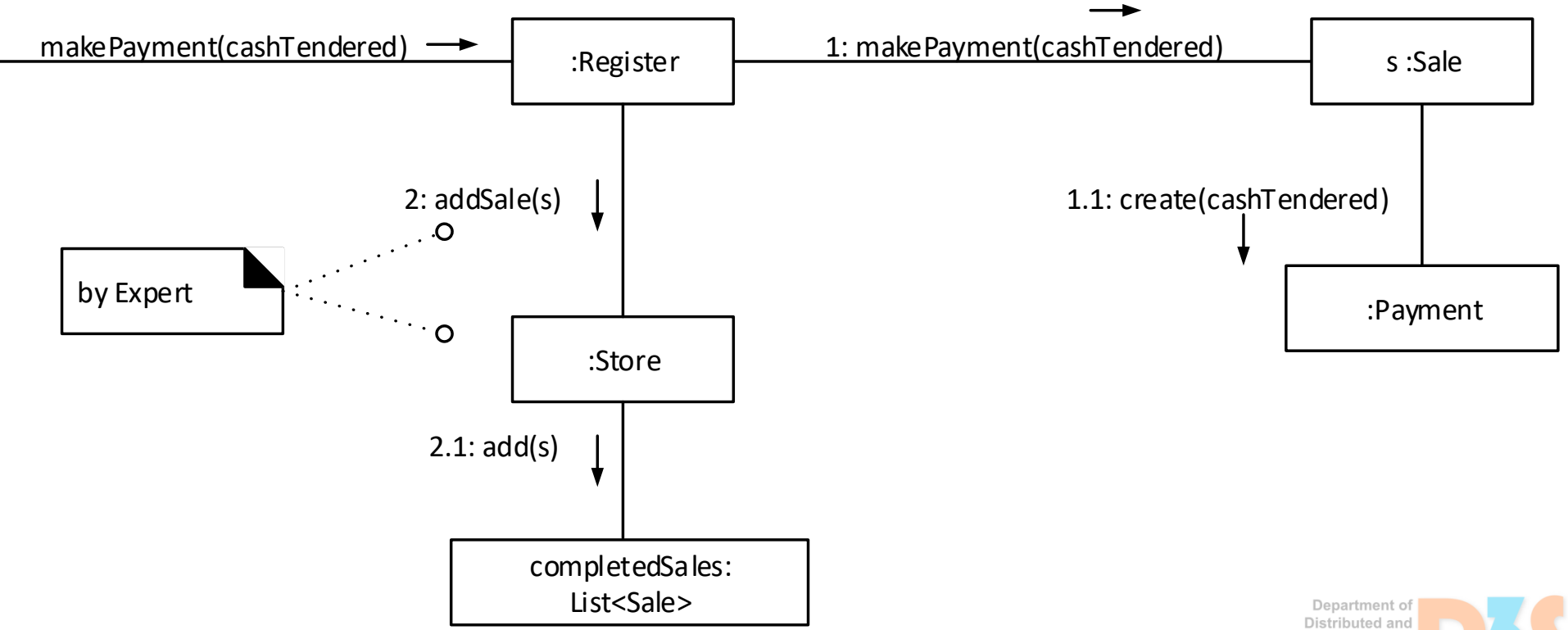
Making Payment



Logging Sales

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. **System records the cash payment.**



Model-driven development (MDD)

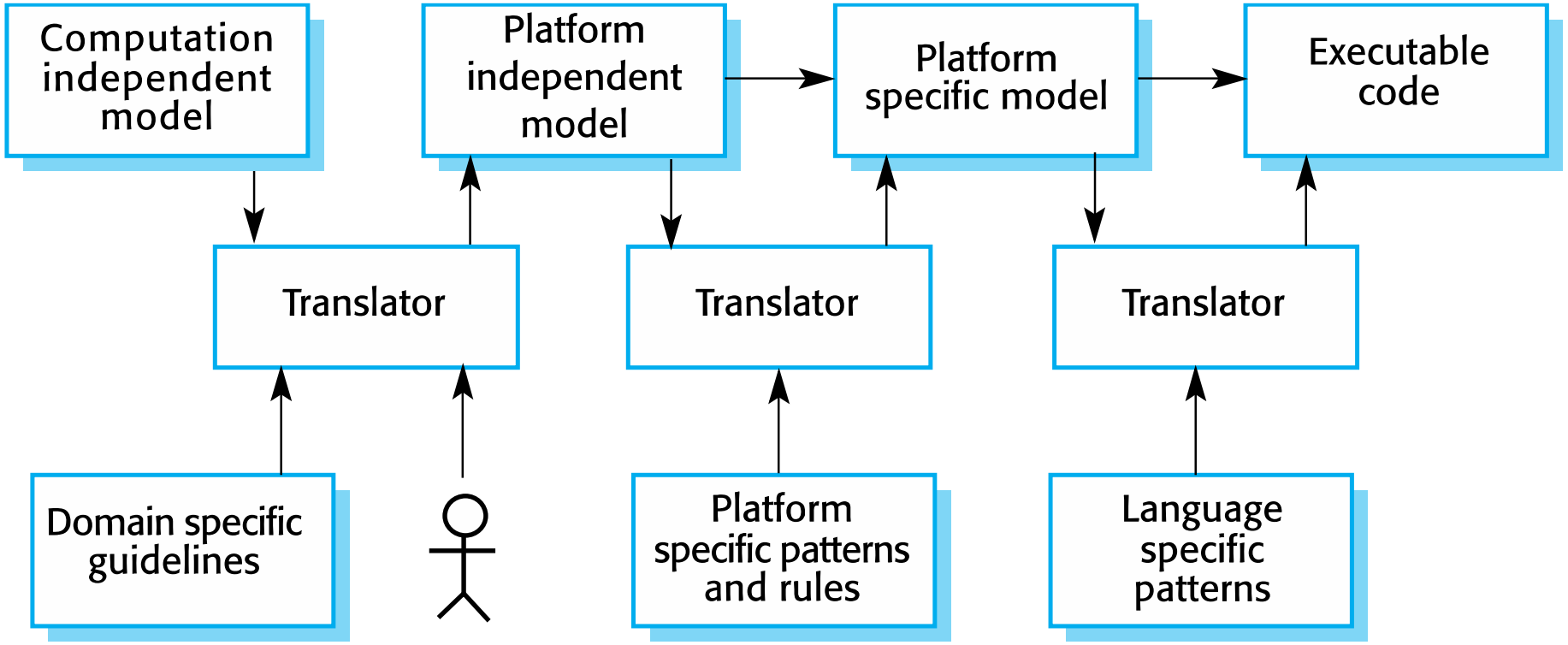
Model-driven development

- Uses models to capture system and its constitutents
- Benefit of models:
  - High-level documentation
  - Analysis
  - Code generation
- A model is essentially a domain specific langauge

Types of model

- A computation independent model (CIM)
  - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- A platform independent model (PIM)
  - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- Platform specific models (PSM)
  - These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDD transformations



AUTOSAR

Based on W. Hardt, M. Caspar, N. Englisch:
Practical Automtotive Software Engineering

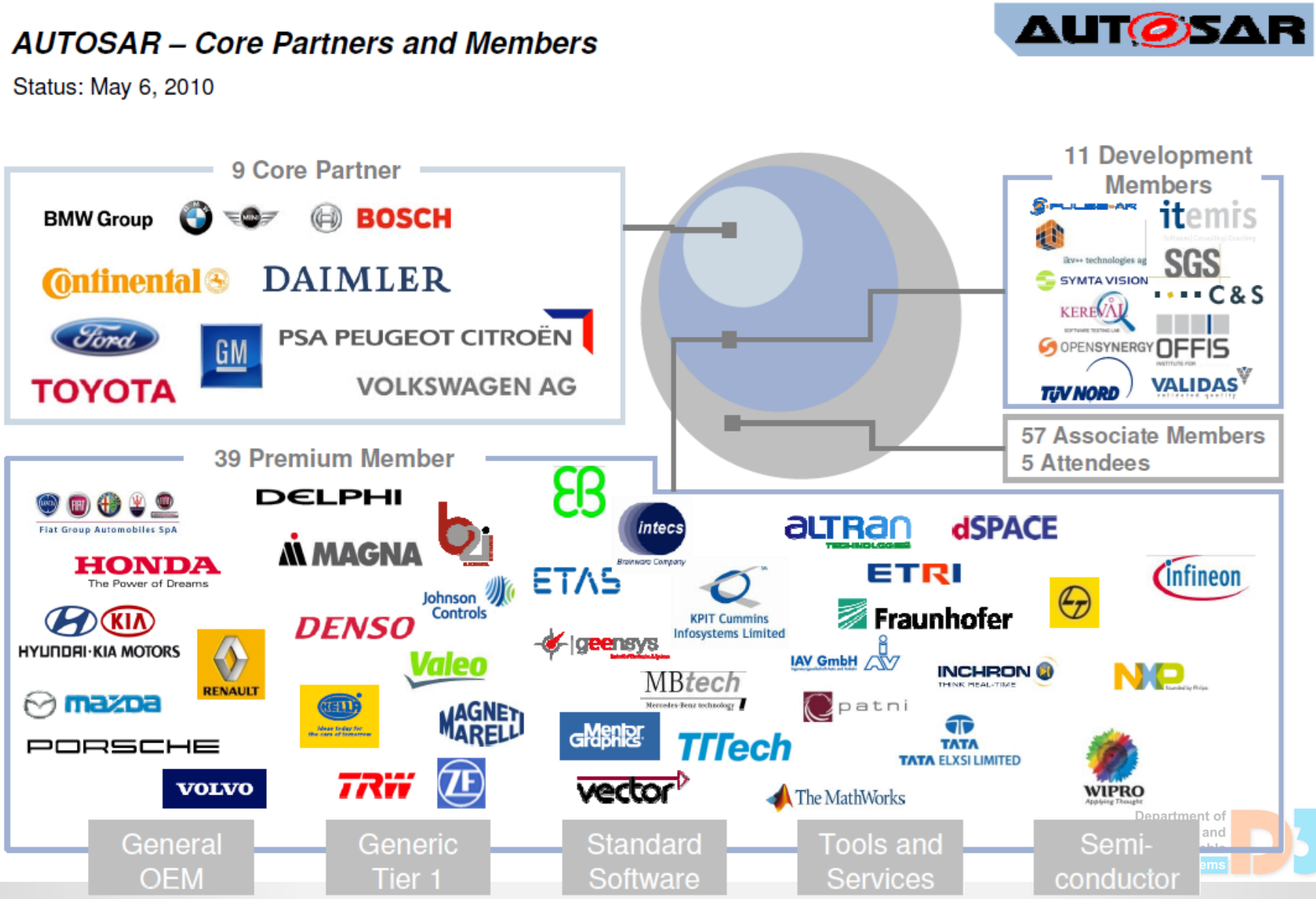
Department of Distributed and Dependable Systems
D3S

AUTOSAR

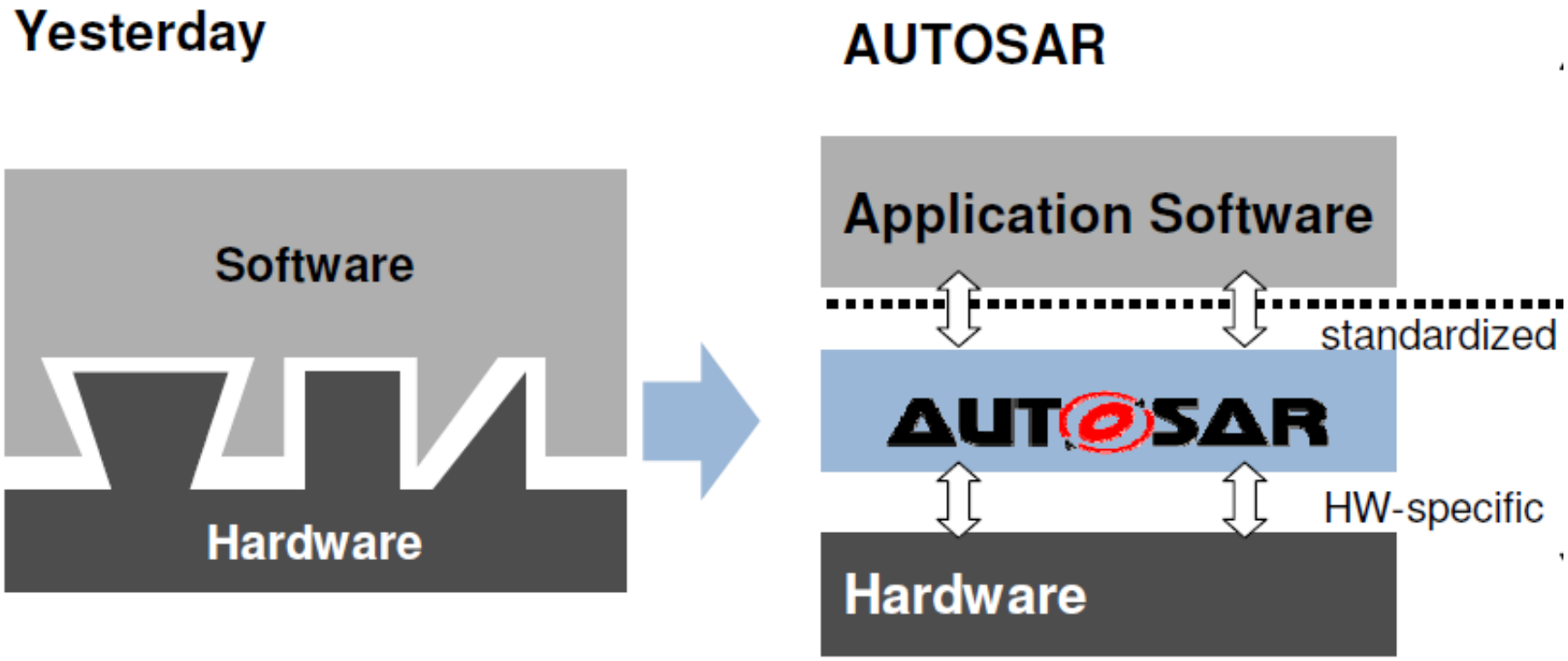
- Automotive Open System Architecture
- Founded by different OEMs and Tiers in 2003

- Component model for development of software for automotive systems
- MDD in action

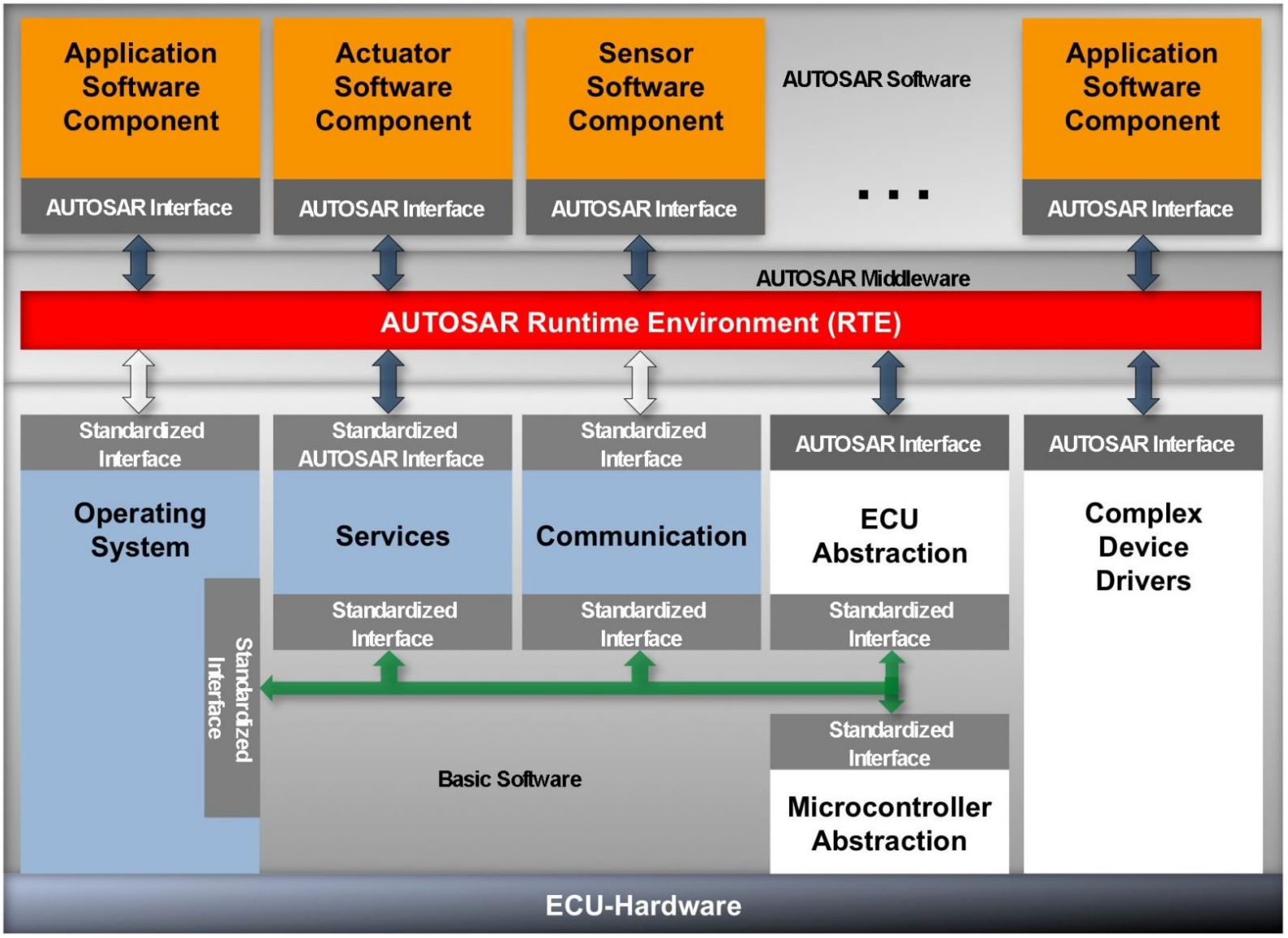
AUTOSAR Partners



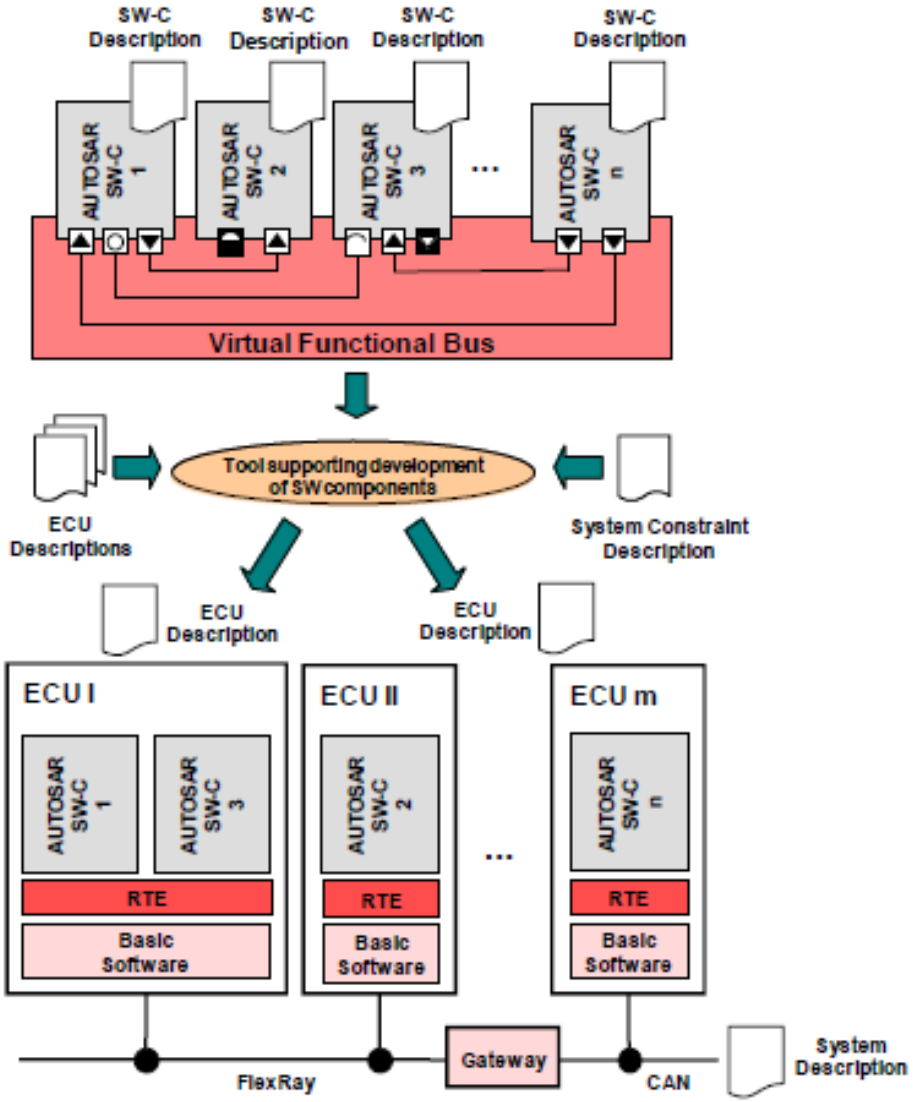
Basic Idea


Yesterday
Software
Hardware
AUTOSAR
Application Software
standardized
AUTOSAR
HW-specific
Hardware

Architecture



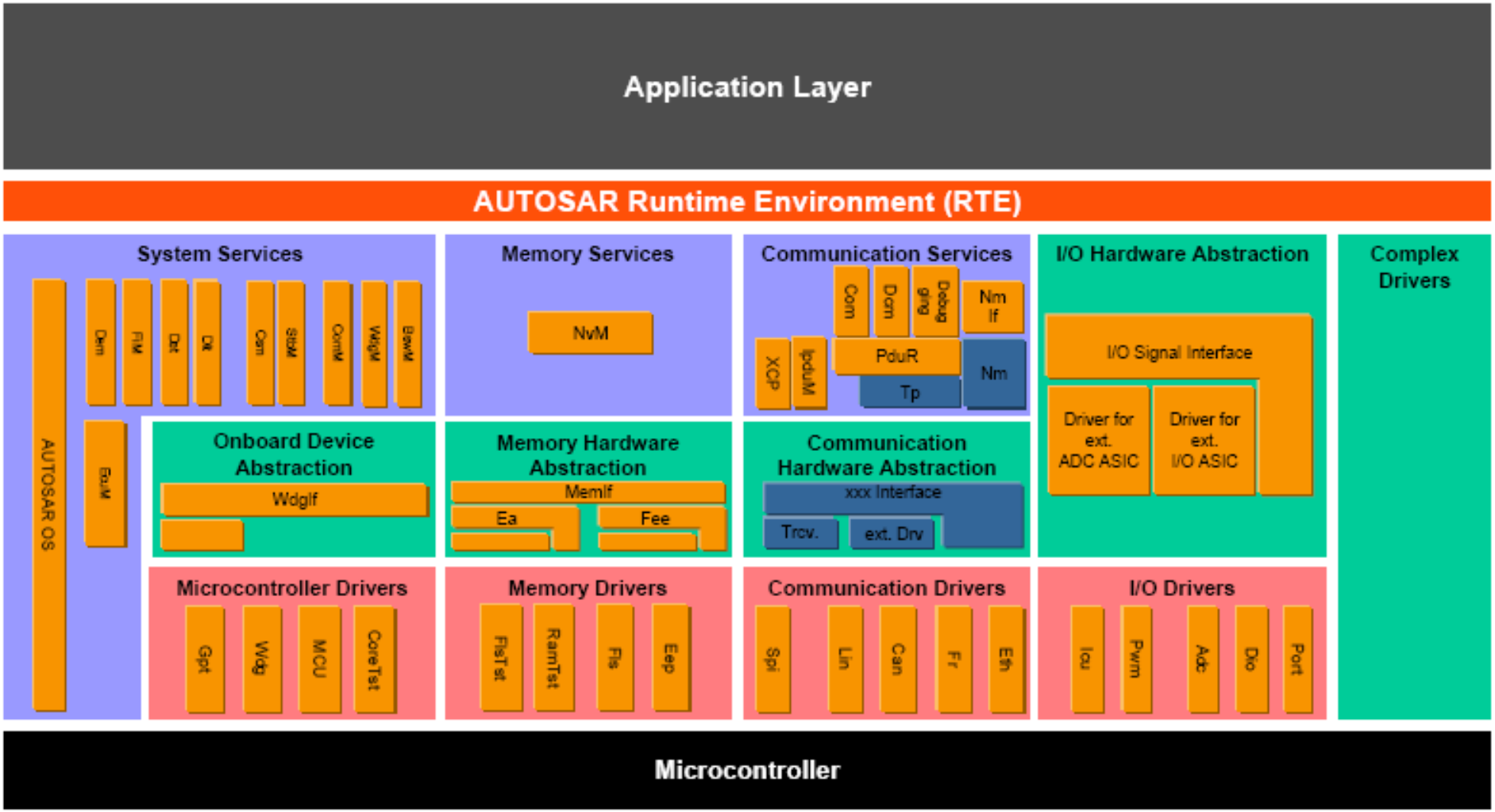
Following the AUTOSAR Methodology, the E/E architecture is derived from the formal description of software and hardware components.



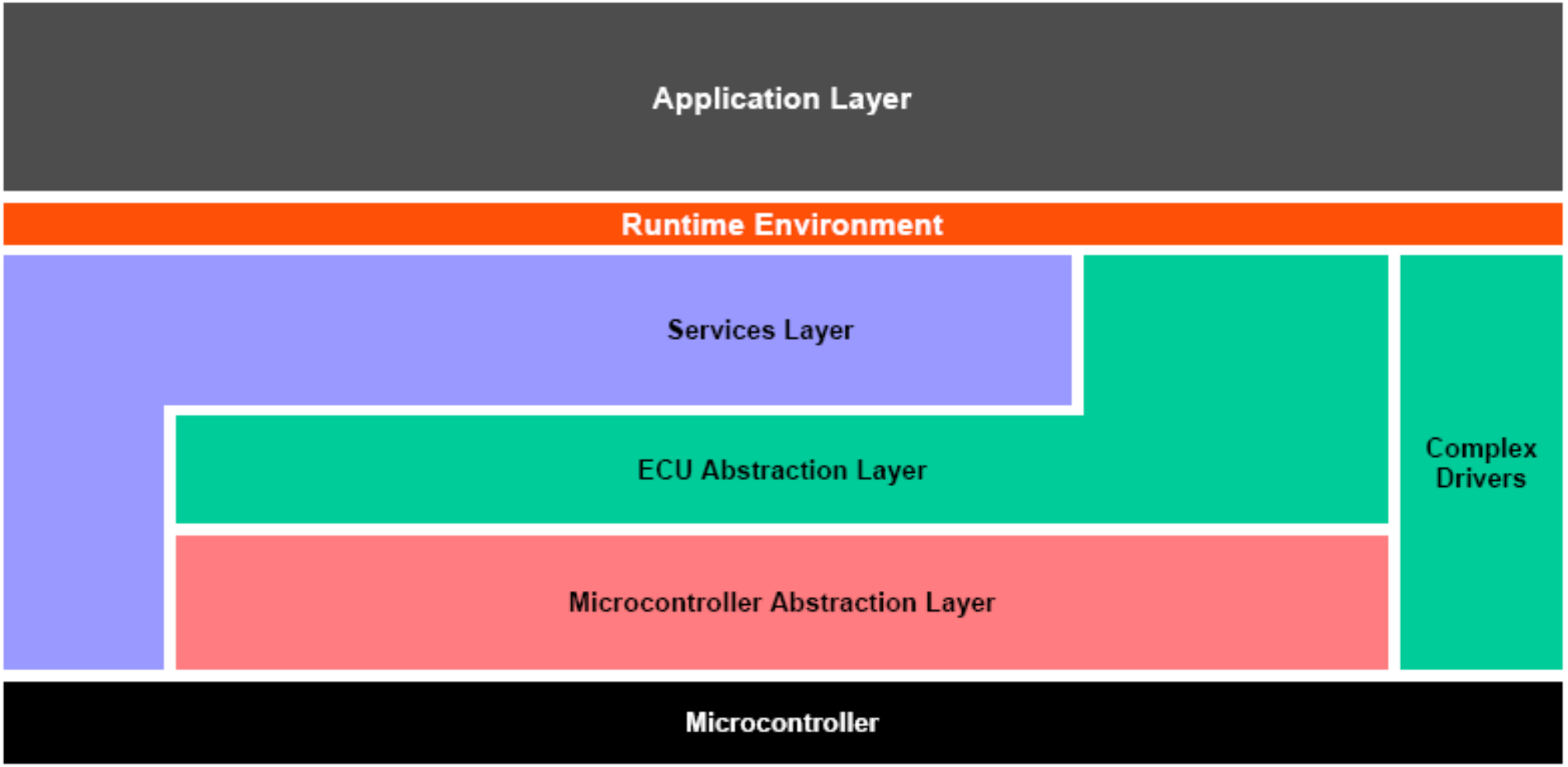
- Functional software is described formally in terms of "Software Components" (SW-Cs).
- Using „Software Component Descriptions" as input, the „**V**irtual **F**unctional **B**us" validates the interaction of all components and interfaces before software implementation.
- Mapping of "Software Components" to ECUs.

➔ The AUTOSAR Methodology supports the generation of an E/E architecture.

BSW Stack


Application Layer
AUTOSAR Runtime Environment (RTE)
System Services
AUTOSAR OS
Dem
FiM
Det
Dlt
Com
StbM
ComM
WdgM
BswM
EcuM
Onboard Device Abstraction
Wdgif
Memory Services
NvM
Memory Hardware Abstraction
MemIf
Ea
Fee
Communication Services
Com
Dcm
Debug ging
Nm If
XCP
IpduM
PduR
Nm
Tp
Communication Hardware Abstraction
xxx Interface
Trcv.
ext. Drv
I/O Hardware Abstraction
I/O Signal Interface
Driver for ext. ADC ASIC
Driver for ext. I/O ASIC
Complex Drivers
Microcontroller Drivers
Gpt
Wdg
MCU
CoreTst
Memory Drivers
FlsTst
RamTst
Fls
Eep
Communication Drivers
Spi
Lin
Can
Fr
Eth
I/O Drivers
Icu
Pwm
Adc
Dio
Port
Microcontroller

BSW Stack


Application Layer
Runtime Environment
Services Layer
ECU Abstraction Layer
Microcontroller Abstraction Layer
Complex Drivers
Microcontroller

Architecture

- Microcontroller Abstraction Layer
  - Lowest layer of BSW
  - Contains internal drivers
  - Hardware dependent
  - → Makes higher Layers independent
- ECU Abstraction Layer
  - API to access peripherals an devices
  - Driver for external devices
  - Make higher layers independent of ECU Hardware
  - Implementation microcontroller independent, ECU Hardware dependent

Architecture

- Service Layer
  - Highest Layer of BSW
  - Operating System, NetworkManagment, Memory Services, Diagnostic Services, ECU State Management
  - Provide basic services for applications and basic software modules

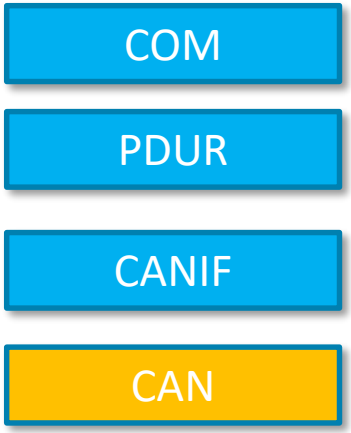
  → mostly µC and ECU hardware independent

- Complex Device Driver
  - Provide the possibility to integrate special purpose functionality :
    - which are not specified within AUTOSAR,
    - With very high time constraints
    - For Migration

  → Mostly Microcontroller an ECU dependent

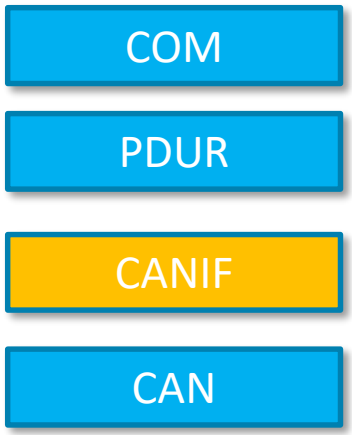
Layering Example: COM Stack

- CAN Driver (CAN)
  - Layer: MCAL
  - Provides functions for Initialization and CallBack Functions for higher Layers



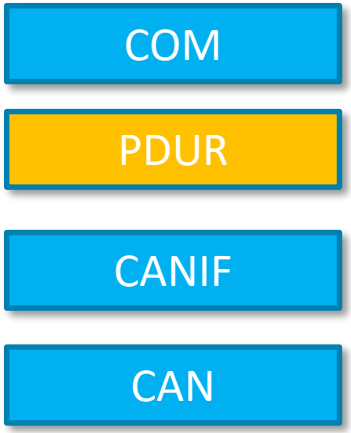
Layering Example: COM Stack

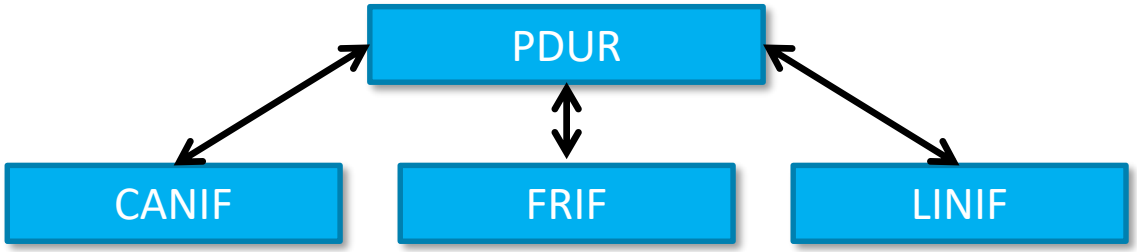
- CAN Interface Modul (CAN IF)
  - Layer: ECU Abstraction Layer
  - Access to CAN Channels
  - Abstracts between intern and extern CAN Controller
  - Hardware-independent CAN Functions



Layering Example: COM Stack

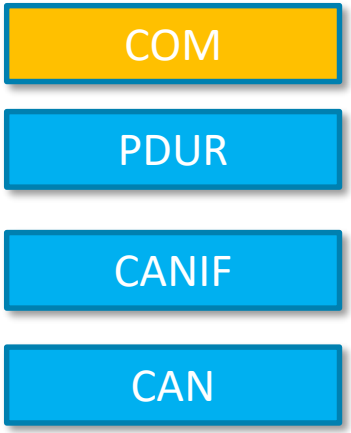
- PDU Router (PDUR)
  - Layer: Service Layer
  - Allocates PDUs between CAN, Flex Ray and LIN



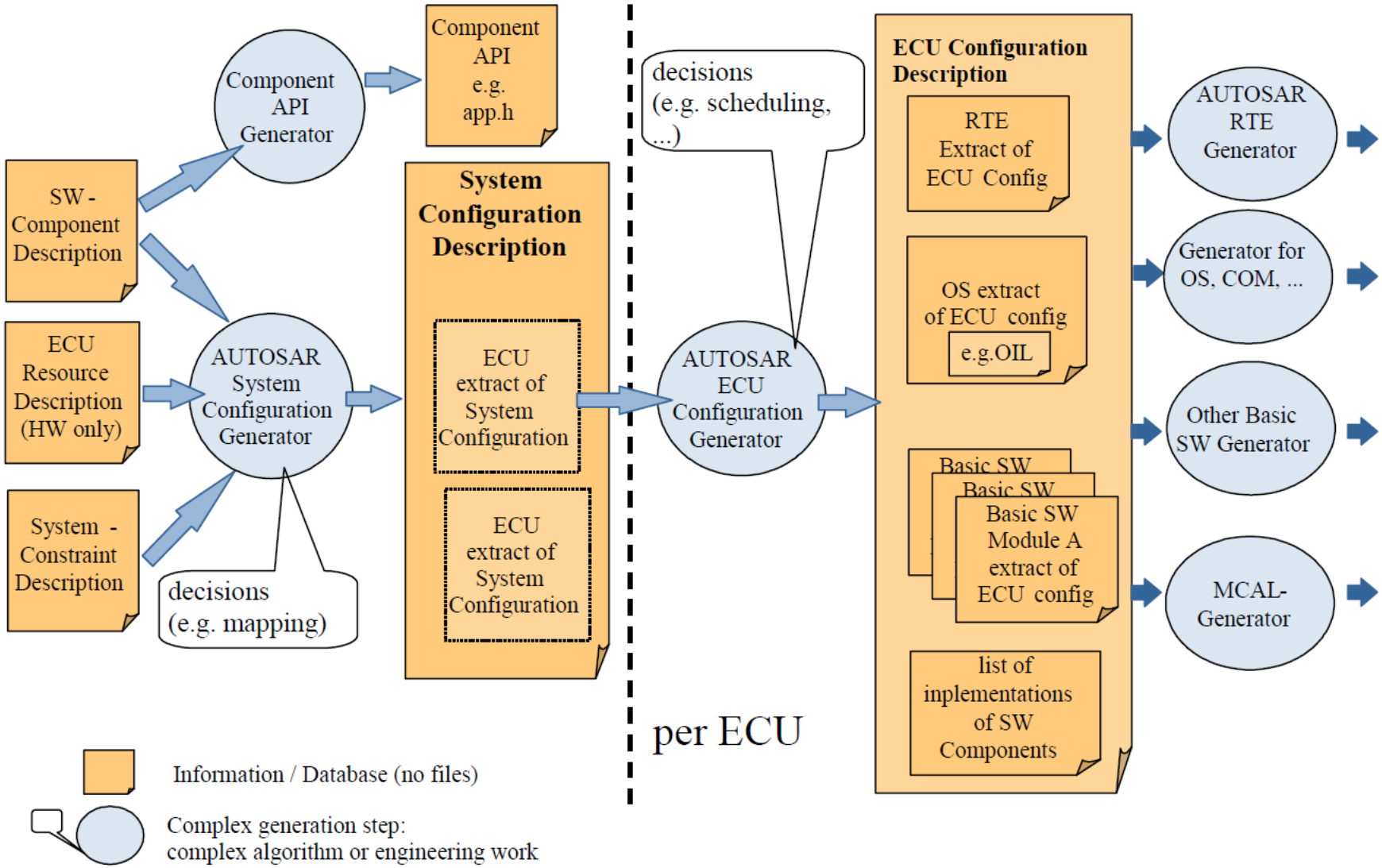


Layering Example: COM Stack

- AUTOSAR COM (COM)
  - Layer: Service Layer
  - Organises Inter and Intra ECU Communication
  - Pack and unpack signals from and into PDUs



Development Process



Application Layer – Components

- Software component kinds
  - Application
  - Sensor/Actuator
  - Parameter
  - NvBlock
  - ComplexDeviceDriver
  - Service
  - ServiceProxy
  - ECU Abstraction
  - Composite

Application Layer – Components

- Port types
  - Client/Server
  - Sender/Receiver
  - Parameter
  - NvData
  - ModeSwitch
  - Trigger

Development artifacts

- Examples from the Blinker app…