Object Constraint Language (OCL) Timing Diagrams

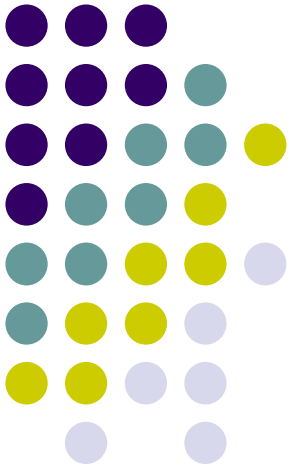
Model Driven Architecture (MDA)

OCL

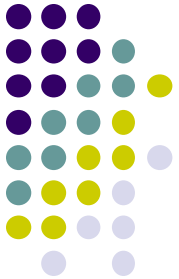
Characters of OCL

Timing diagrams

Examples

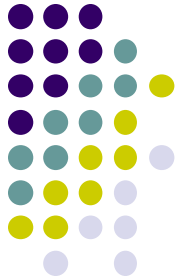


Literature



- OCL website: http://www.omg.org/uml/
- Textbook: "*The Objection Constraint Language: Precise Modeling with UML*", by Jos Warmer and Anneke Kleppe
- This presentation includes some slides by:
  - Yong He
  - Tevfik Bultan
  - Brian Lings
  - Lieber.

Model Driven Architecture (MDA)

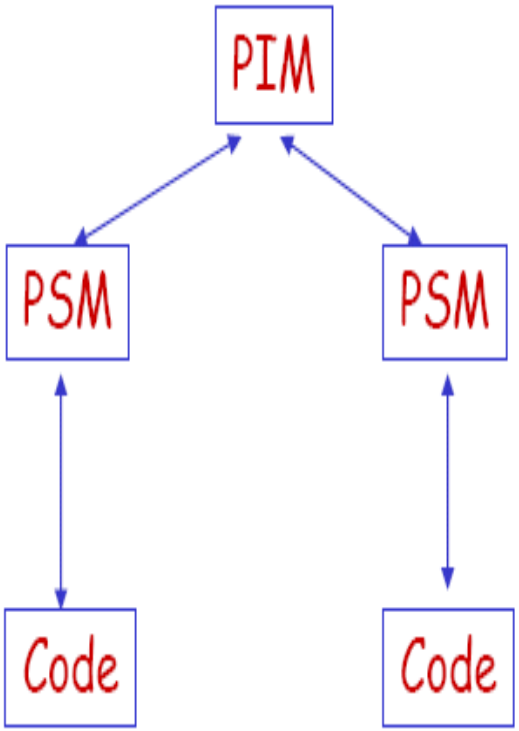


- In Model Driven Architecture (MDA), the software development process is driven by the activity of modeling.
- MDA approach separates the specification of system functionalities from the specification of the implementation of these functionalities on a particular technological platform.
- The MDA framework defines how to specify and transform models at different abstraction levels.
- MDA is under supervision of the Object Management Group (OMG).
- More info:
  - *MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.*
  - *O. Pastor, J.C. Molina. Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling, 2007*

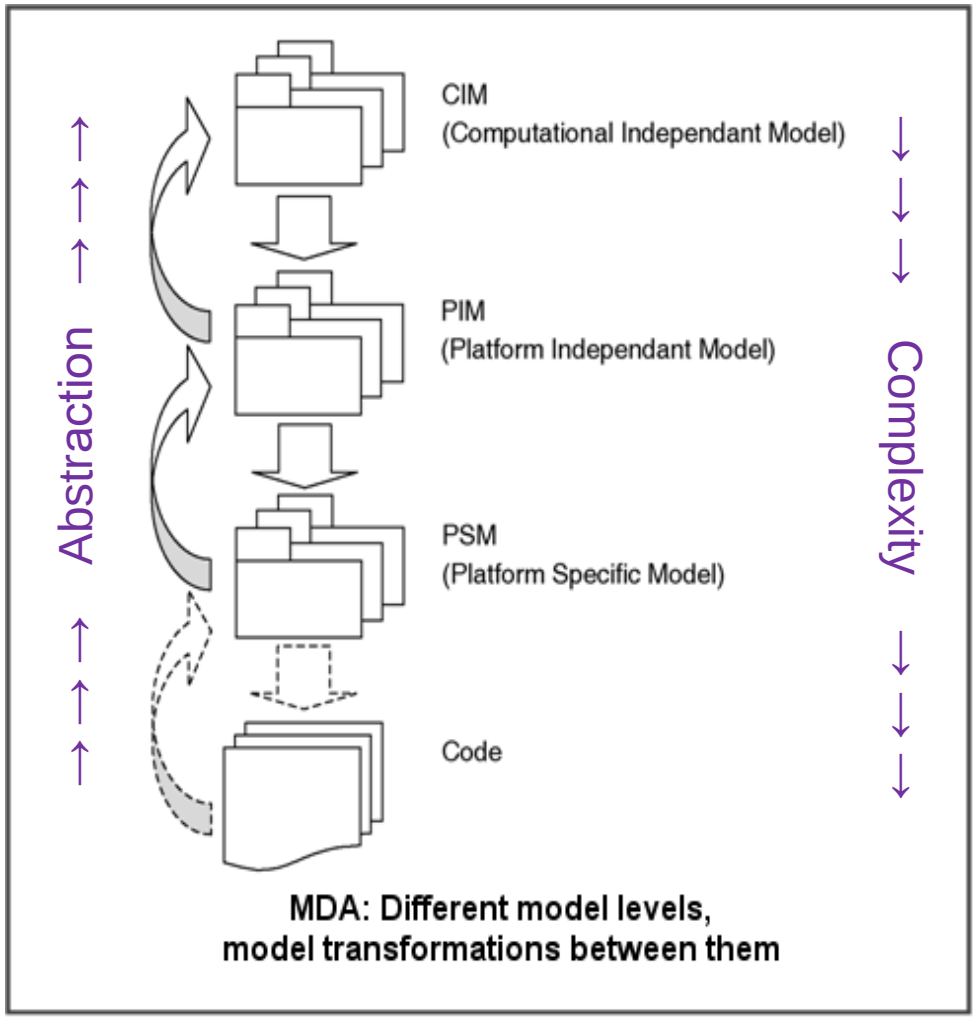
The MDA Process – year 2003

The MDA process consists of three steps:

- Build a model with a high level of abstraction, called ***Platform Independent Model (PIM).***
- Transform the PIM into one or more ***Platform Specific Models (PSMs),*** i.e., models that are specified in some specific implementation technology.
- Transform the PSMs to code.



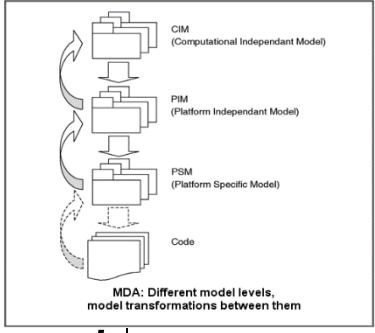
The MDA Process – Modern Version



MDA: Different model levels, model transformations between them

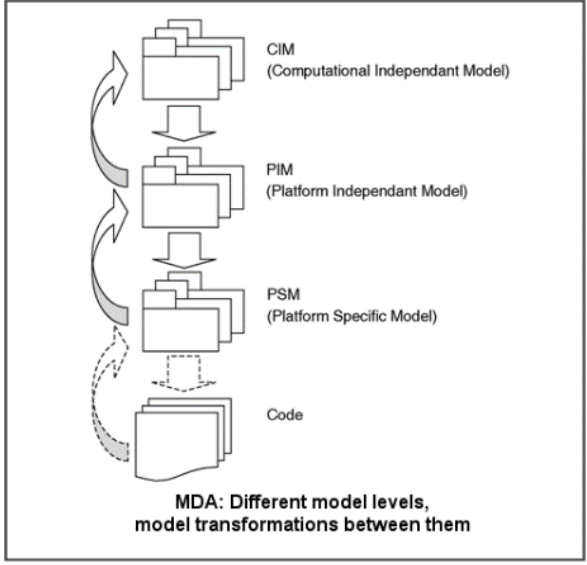
- **CIM** does not show details of the system structure. CIM always uses a vocabulary that is familiar to the practitioners of the domain.
- **PIM** represents the business model to be implemented by an information system. PIM describes processes and structure of the system without reference to the delivery platforms. PIM ignores operating system, programming languages, hardware and networking.
- **PSM** combines specifications in the PIM with details that specify how a system uses a particular type of platform.

MDA Elements



- Modeling languages describe models. *Models must be consistent and precise,* and contain as much information as possible.
- *CIM is a requirements model* for the future application with its services offered to other entities with which it interacts.
- *PIM represents the system-specific business logic or the design model*. It represents the functioning of entities and services. It must be durable and lasting over time.
- PSM - the system model is linked to *an execution platform*
- PIM-to-PSM is more challenging than PSM-to-Code.
- Transformation definitions map one model to another.
- These definitions must be independent on the tools.

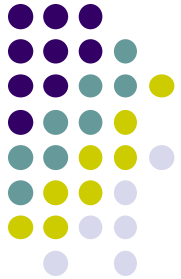
MDA Benefits



MDA: Different model levels, model transformations between them

- ***Transformation*** tools do the dirty work.
- ***Portability*** - PIMs can be transformed to different PSMs.
- ***Productivity*** - developers work at a higher level abstraction.
- Cross-platform ***interoperability*** - PIMs serve as a bridge between different PSMs.
- Easier ***maintenance*** and ***documentation***
- Maintaining PIMs is much easier than maintaining code.

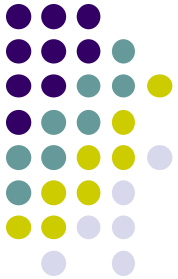
Maturity Levels



The maturity level indicates the gap between the model and the system.

- Level 0: No specification - Everything is in mind.
- Level 1: Textual description - Informal English description.
- Level 2: Text with Diagrams - Use diagrams to help understanding.
- Level 3: Models with Text - Models have well-defined meaning
- Level 4: Precise models - Precise enough to enable automatic model-to-code transformation.
- Level 5: Models only - Code is invisible.

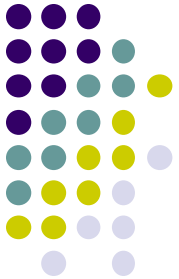
UML, OCL, and MDA



- UML uses diagrams to express software design.
- Diagrams are easier to understand, but many properties cannot be expressed using diagrams alone.
- The use of OCL (Object Constraint Language) can add additional and necessary info to UML diagrams.
- OCL uses expressions that have solid mathematic foundation but still maintains the ease of use.
- *Combining UML and OCL is necessary to construct models at maturity level 4* - models precise enough to enable automatic model-to-code transformation.
- The application of MDA relies on Level 4 models.

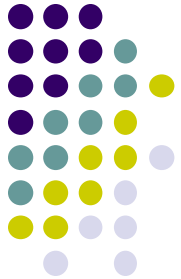
OCL Language Description

(*SOURCE*: Object Constraint Language, OMG Spec. Ver.2.0)



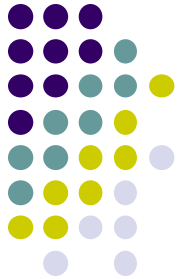
- OCL - a *formal language* that remains easy to read and write
- OCL is a *pure specification language* - no side effects (an OCL expression simply returns a value and change anything in the model)
- OCL is not a programming but *modeling language* - it is not possible to write program logic or flow control in OCL
- The evaluation of an OCL expression is *instantaneous*. This means that *the states of objects in a model cannot change during evaluation*

Advantages of Formal Constraints



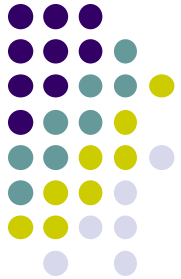
- Better documentation
  - Constraints add information about the model elements and their relationships to the visual models used in UML
  - It is way of documenting the model
- More precision OCL constraints have formal semantics, hence, can be used to reduce the ambiguity in the UML models
- Communication without misunderstanding
  - UML models are used to communicate between developers
  - Using OCL constraints modelers can communicate unambiguously

OCL - History



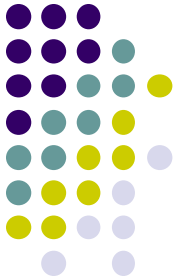
- First developed in 1995 as IBEL by IBM's Insurance division for business modelling
- IBM proposed it to the OMG's call for an object-oriented analysis and design standard. OCL was then merged into UML 1.1.
- OCL was used to define UML 1.2 itself.
- Supported by OMG, Microsoft, HP, IBM, Oracle, Sterling, Unisys, ICON, IntelliCorp, Softeam …

Where to use OCL?



- Specify invariants for classes and types
- Specify pre- and post-conditions for methods
- Guard conditions
- As a navigation language – specifies targets for messages and actions
- To specify constraints on operations
- Test requirements and specifications

OCL Context

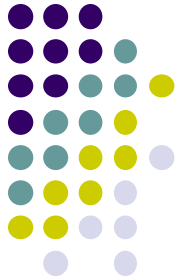


- “Intuitive” syntax – reminds OO programming languages
- The *context keyword introduces the context for the expression – class, attribute, operation, operation param., ...*
- *The keyword **inv, pre,** and **post** denote the stereotypes, respectively «invariant», «precondition», and «postcondition» of the constraint.*
- *The actual OCL expression comes after the colon.*

context TypeName inv:

'this is an OCL expression with stereotype <<invariant>> in the context of TypeName’ = 'another string'

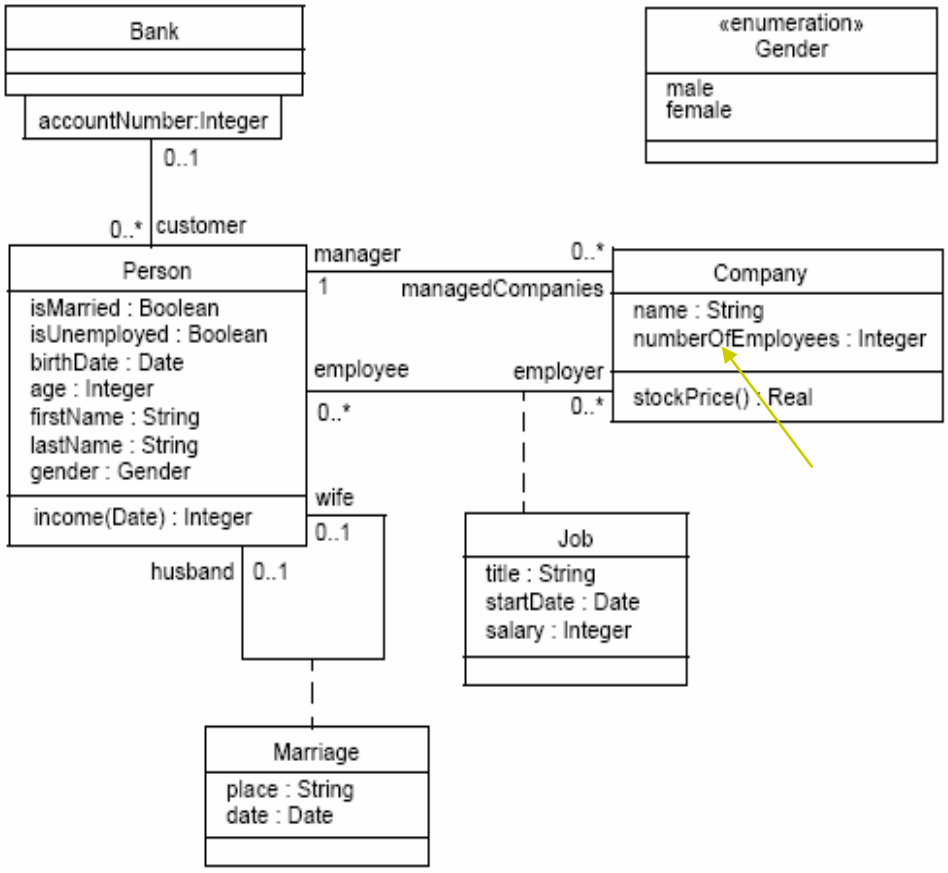
Invariants 1/2



- The OCL expression can be part of an Invariant which is a constraint stereotyped as an «invariant».
- An OCL expression is an invariant of the type and must be true for all instances of that type at any time.
- All OCL expressions that express invariants are of type Boolean.

Invariants 2/2

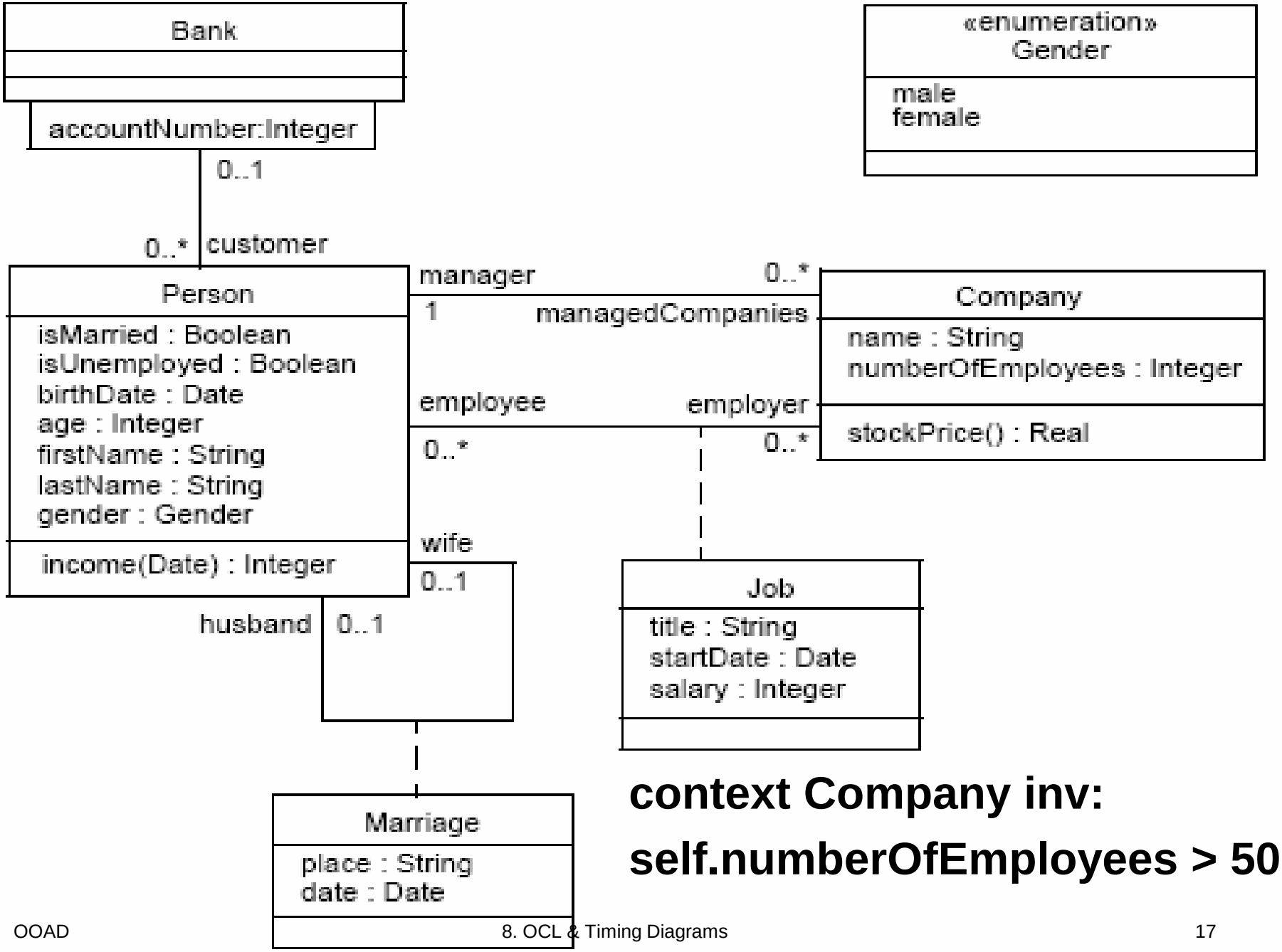
- in the context of the Company type, the following expression would specify an invariant that the number of employees must always exceed 50:



context Company inv:

self.numberOfEmployees > 50

-- *self is an instance of type Company (*refers to the contextual instance)


Bank
accountNumber:Integer
0..1
0..*
customer
«enumeration»
Gender
male
female
Person
isMarried : Boolean
isUnemployed : Boolean
birthDate : Date
age : Integer
firstName : String
lastName : String
gender : Gender
income(Date) : Integer
manager
1
0..*
managedCompanies
Company
name : String
numberOfEmployees : Integer
stockPrice() : Real
employee
0..*
employer
0..*
wife
0..1
husband
0..1
Job
title : String
startDate : Date
salary : Integer
Marriage
place : String
date : Date
context Company inv:
self.numberOfEmployees > 50

Omitting *self*

context Company inv:

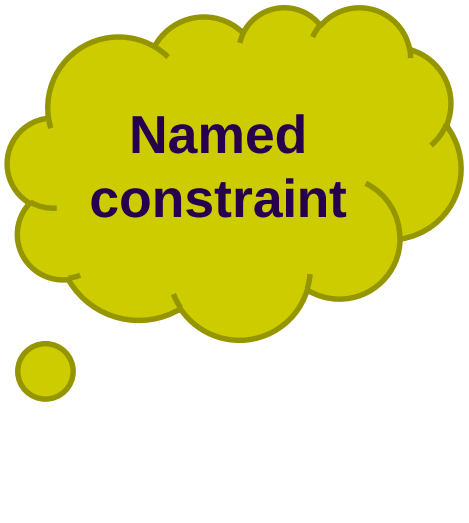
self.numberOfEmployees > 50

context c : Company inv:

c.numberOfEmployees > 50

context c : Company inv <u>enoughEmployees</u>:

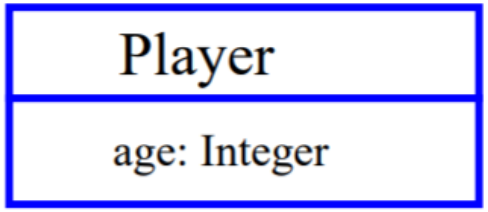
c.numberOfEmployees > 50



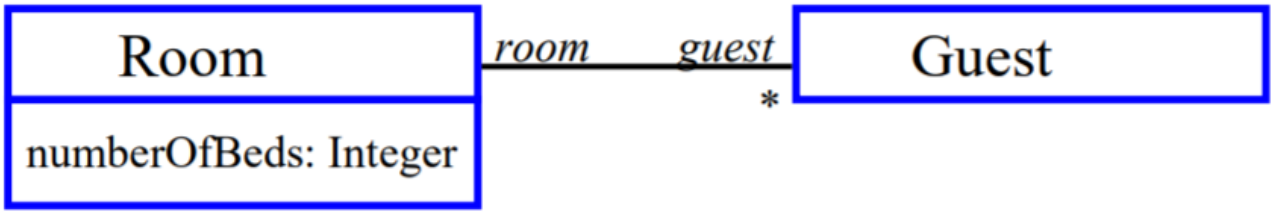
More examples

- All players must be over 18.

context Player **invariant:**
self.age >=18

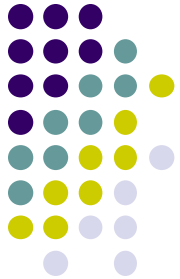


- The number of guests in each room doesn't exceed the number of beds in the room.



context Room **invariant:**
guests -> size <= numberOfBeds

Pre- and Post-Conditions



- The OCL expression can be part of a Precondition or Postcondition, corresponding to «precondition» and «postcondition» stereotypes of Constraint associated with an Operation or other behavioral feature.

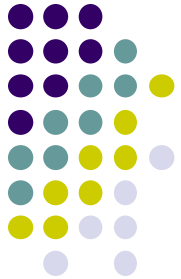
context Typename::operationName(param1 : Type1, ...): ReturnType

pre : param1 > 5

post: result = 55

- The name ***self*** *can be used in the expression referring to the object on which the operation was called.*
- *The reserved word* ***result*** *denotes the result of the operation, if there is one.*

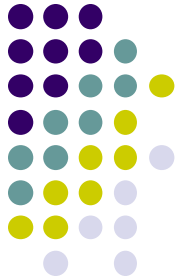
Example of a static UML Model 1/2



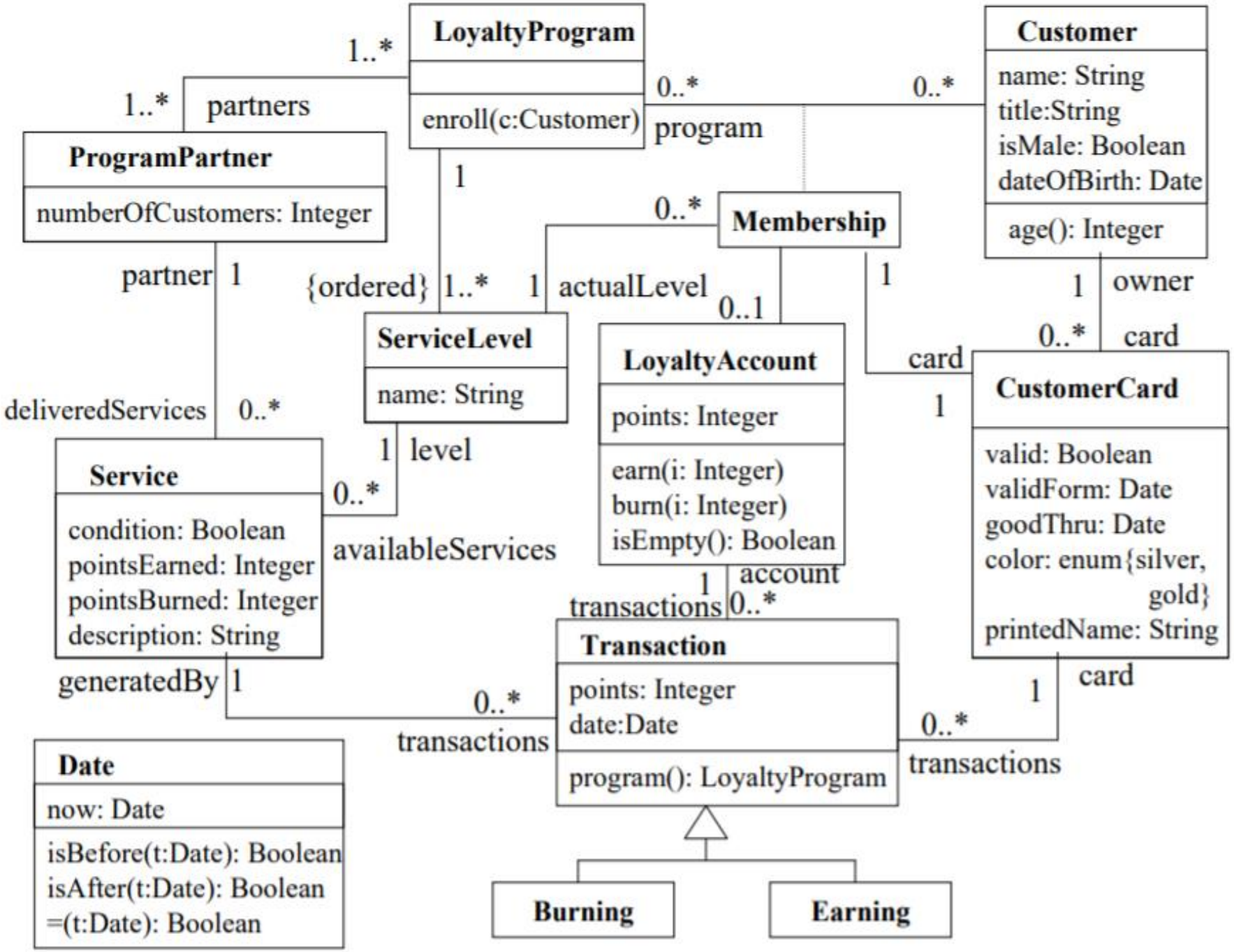
Problem story:

- A company handles loyalty programs (class LoyaltyProgram) for companies (class ProgramPartner) that offer their customers various kinds of bonuses.
- Often, the extras take the form of bonus points or air miles, but other bonuses are possible.
- Anything a company is willing to offer can be a service (class Service) rendered in a loyalty program.
- Every customer can enter the loyalty program by obtaining a membership card (class CustomerCard).
- The objects of class Customer represent the persons who have entered the program.

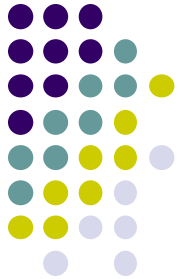
Example of a static UML Model 2/2



- A membership card is issued to one person, but can be used for an entire family or business.
- Loyalty programs can allow customers to save bonus points (class loyaltyAccount), with which they can “buy” services from program partners.
- A loyalty account is issued per customer membership in a loyalty program (association class Membership).
- Transactions (class Transaction) on loyalty accounts involve various services provided by the program partners and are performed per single card. There are two kinds of transactions: earning and burning. Membership durations determine various levels of services (class serviceLevel).

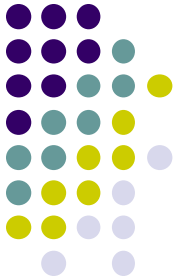

LoyaltyProgram
enroll(c:Customer)
Customer
name: String
title:String
isMale: Boolean
dateOfBirth: Date
age(): Integer
1..*
partners
1..*
0..*
program
0..*
ProgramPartner
numberOfCustomers: Integer
1
0..*
Membership
partner
1
{ordered}
1..*
1
actualLevel
0..1
1
owner
1
0..*
card
ServiceLevel
name: String
LoyaltyAccount
points: Integer
earn(i: Integer)
burn(i: Integer)
isEmpty(): Boolean
card
1
CustomerCard
valid: Boolean
validForm: Date
goodThru: Date
color: enum{silver, gold}
printedName: String
deliveredServices
0..*
1
level
Service
condition: Boolean
pointsEarned: Integer
pointsBurned: Integer
description: String
0..*
availableServices
1
account
transactions
0..*
Transaction
points: Integer
date:Date
program(): LoyaltyProgram
generatedBy
1
0..*
transactions
1
card
0..*
transactions
Date
now: Date
isBefore(t:Date): Boolean
isAfter(t:Date): Boolean
=(t:Date): Boolean
Burning
Earning

Objects and Properties



- *OCL expressions* can refer to ***classifiers***, e.g., types, classes, interfaces, associations (acting as types), and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types, etc. can be used.
- OCL refers to attributes, association-ends, and side-effect-free methods/operations as being *properties. A property is one of:*
- ***an Attribute***
- ***an AssociationEnd***
- ***an Operation with isQuery being true***

Invariants on Attributes

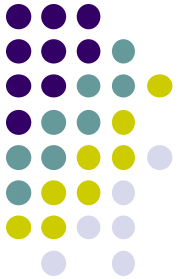


- Invariants on attributes:

```
context Customer
invariant ageRestriction: age >= 18
context CustomerCard
invariant correctDates:
validFrom.isBefore(goodThru)
//The type of validFrom and goodThru is Date.
//isBefore(Date):Boolean is a Date operation.
```

- The class on which the invariant must be put is the invariant context.
- For the above example, this means that the expression is an invariant of the Customer class.

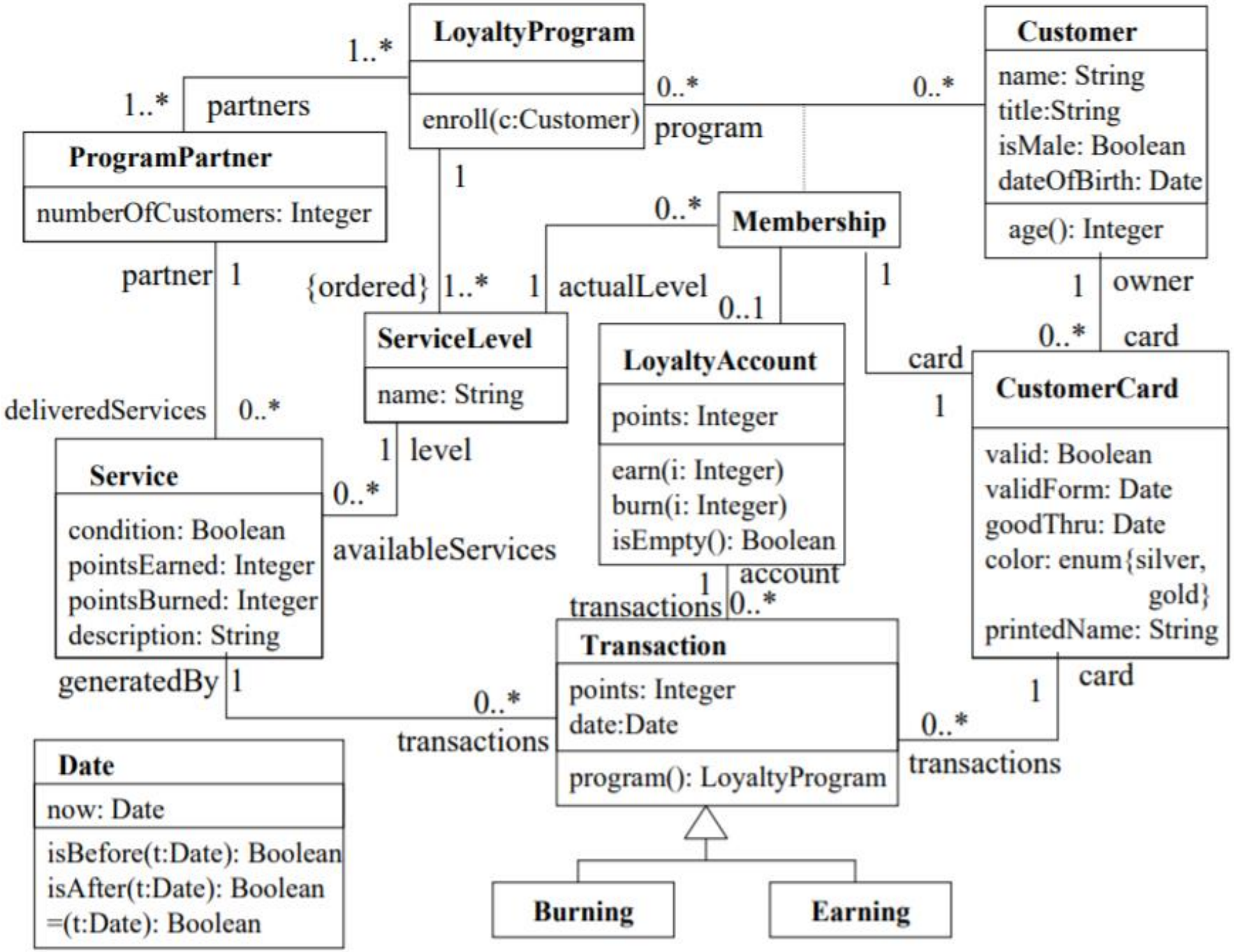
Invariants using Navigation over Association Ends – Roles



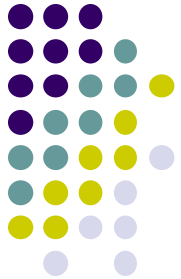
- Navigation over associations is used to refer to associated objects, starting from the context object:

```
context CustomerCard invariant:
owner.age >= 18
//owner is a Customer instance.
//owner.age is an Integer.
```

- Note: This is not the “right” context for this constraint! *If the role name is missing – use the class name at the other end of the association*, starting with a lowercase letter.
- Preferred: *Always give role names*.


LoyaltyProgram
enroll(c:Customer)
Customer
name: String
title:String
isMale: Boolean
dateOfBirth: Date
age(): Integer
1..*
partners
1..*
0..*
program
0..*
ProgramPartner
numberOfCustomers: Integer
1
0..*
Membership
partner
1
{ordered}
1..*
1
actualLevel
0..1
1
owner
1
0..*
card
ServiceLevel
name: String
LoyaltyAccount
points: Integer
earn(i: Integer)
burn(i: Integer)
isEmpty(): Boolean
card
1
CustomerCard
valid: Boolean
validForm: Date
goodThru: Date
color: enum{silver, gold}
printedName: String
deliveredServices
0..*
1
level
Service
condition: Boolean
pointsEarned: Integer
pointsBurned: Integer
description: String
0..*
availableServices
1
account
transactions
0..*
Transaction
points: Integer
date:Date
program(): LoyaltyProgram
generatedBy
1
0..*
transactions
1
card
0..*
transactions
Date
now: Date
isBefore(t:Date): Boolean
isAfter(t:Date): Boolean
=(t:Date): Boolean
Burning
Earning

Invariants using Navigation through Association Classes



- Navigation from a class through an association class uses the association class name to obtain all tuples of an object:

"The cards of the memberships of a customer are only the customer's cards":

```
context Customer
invariant correctCard:
cards->includesAll(Membership.card)
```

- This is exactly the same as this constraint:

"The owner of the card of a membership must be the customer in the membership":

```
context Membership
invariant correctCard:
card.owner = customer
```

- The `Membership correctCard` constraint is better

Invariants using Navigation through Associations with "Many" Multiplicity

- Navigation over associations roles with multiplicity greater than 1 yields a **Collection** type.
- Operations on collections are accessed using an arrow ->, followed by the operation name.

"A customer card belongs only to a membership of its owner":

```
context CustomerCard
invariant correctCard:
owner.Membership->includes(membership)
```

owner -> is a Customer instance.

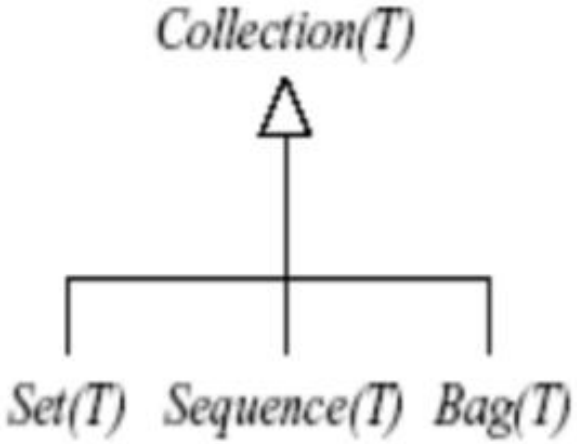
owner.Membership -> is a set of Membership instances.

membership -> is a Membership instance.

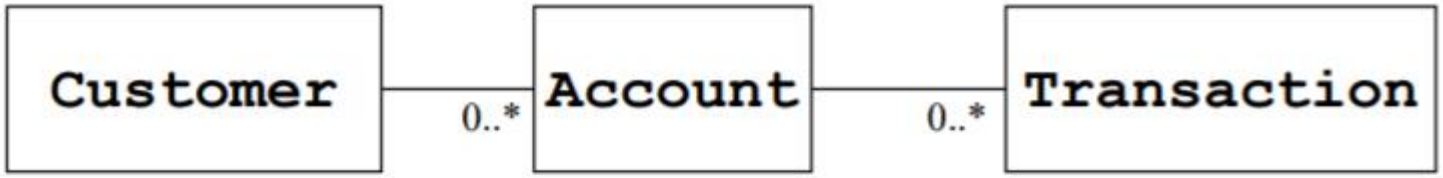
includes is an operation of the OCL Collection type.

The OCL Collection Types

- Collection is a predefined OCL type
  - Operations are defined for collections
  - They never change the original
- Three different collections:
  - Set (no duplicates)
  - Bag (duplicates allowed)
  - Sequence (ordered Bag)
- With collections type, an OCL expression either states a fact about all objects in the collection or states a fact about the collection itself, e.g. the size of the collection.
- Syntax:
  - collection->operation



Navigating to collections



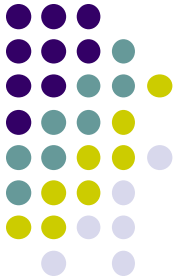
context *Customer*
`account` produces a **set** of Accounts

context *Customer*
`account.transaction` produces a **bag** of transactions

If we want to use this as a set we have to do the following

```
account.transaction -> asSet
```

Navigation to Collections 1/2



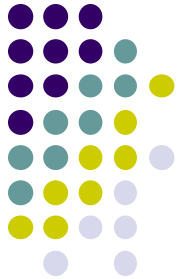
"The partners of a loyalty program have at least one delivered service":

```
context LoyaltyProgram
invariant minServices:
partners.deliveredservices->size() >= 1
```

"The number of a customer's programs is equal to that of his/her valid cards":

```
context Customer
invariant sizesAgree:
program->size() = card->select(valid=true)->size()
```

Navigation to Collections 2/2



"When a loyalty program does not offer the possibility to earn or burn points, the members of the loyalty program do not have loyalty accounts. That is, the loyalty accounts associated with the Memberships must be empty":

```
context LoyaltyProgram
invariant noAccounts:
partners.deliveredservices->
      forAll(pointsEarned = 0 and pointsBurned = 0)
            implies Membership.account->isEmpty()
//and, or, not, implies, xor are logical connectives.
```

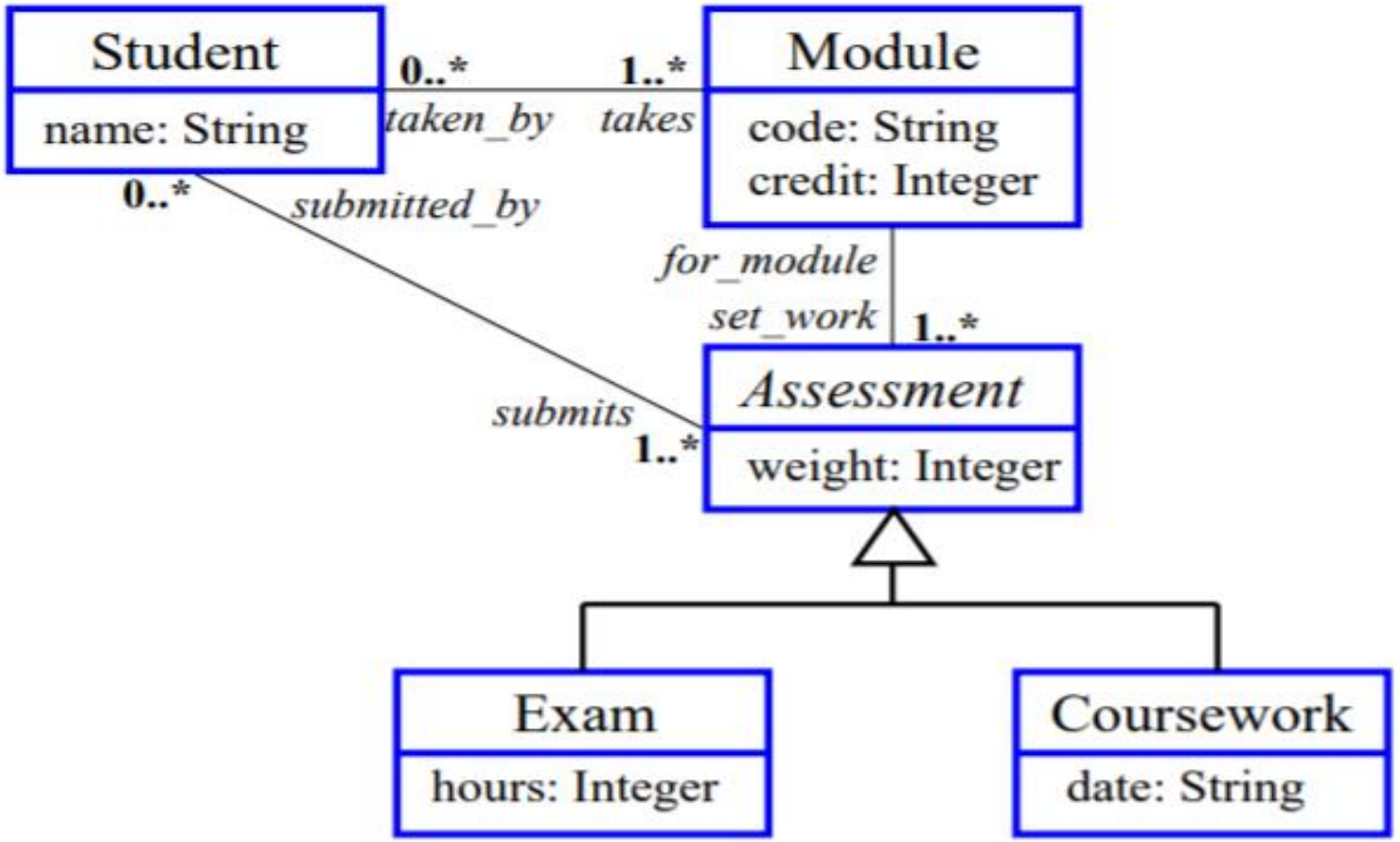
Collection Operations

<collection> → size
→ isEmpty
→ notEmpty
→ sum ( )
→ count (object)
→ includes (object)
→ includesAll (collection)

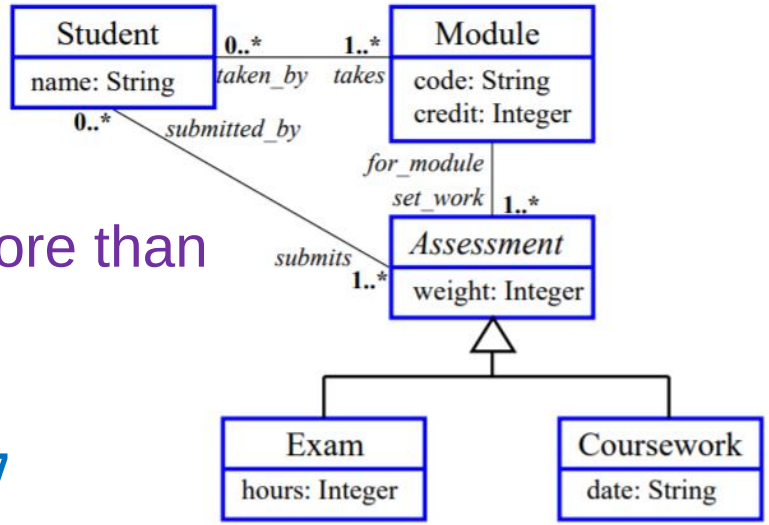
<collection> → select (e:T | <b.e.>)
→ reject (e:T | <b.e.>)
→ collect (e:T | <v.e.>)
→ forAll (e:T* | <b.e.>)
→ exists (e:T | <b.e.>)
→ iterate (e:T_1; r:T_2 = <v.e.> | <v.e.>)

b.e. stands for: boolean expression
v.e. stands for: value expression

Example UML diagram



Constraints



Modules can be taken iff they have more than seven students registered

```
context Module
invariant: taken_by -> size > 7
```

The assessments for a module must total 100%

```
context Module
invariant: set_work.weight -> sum( ) = 100
```

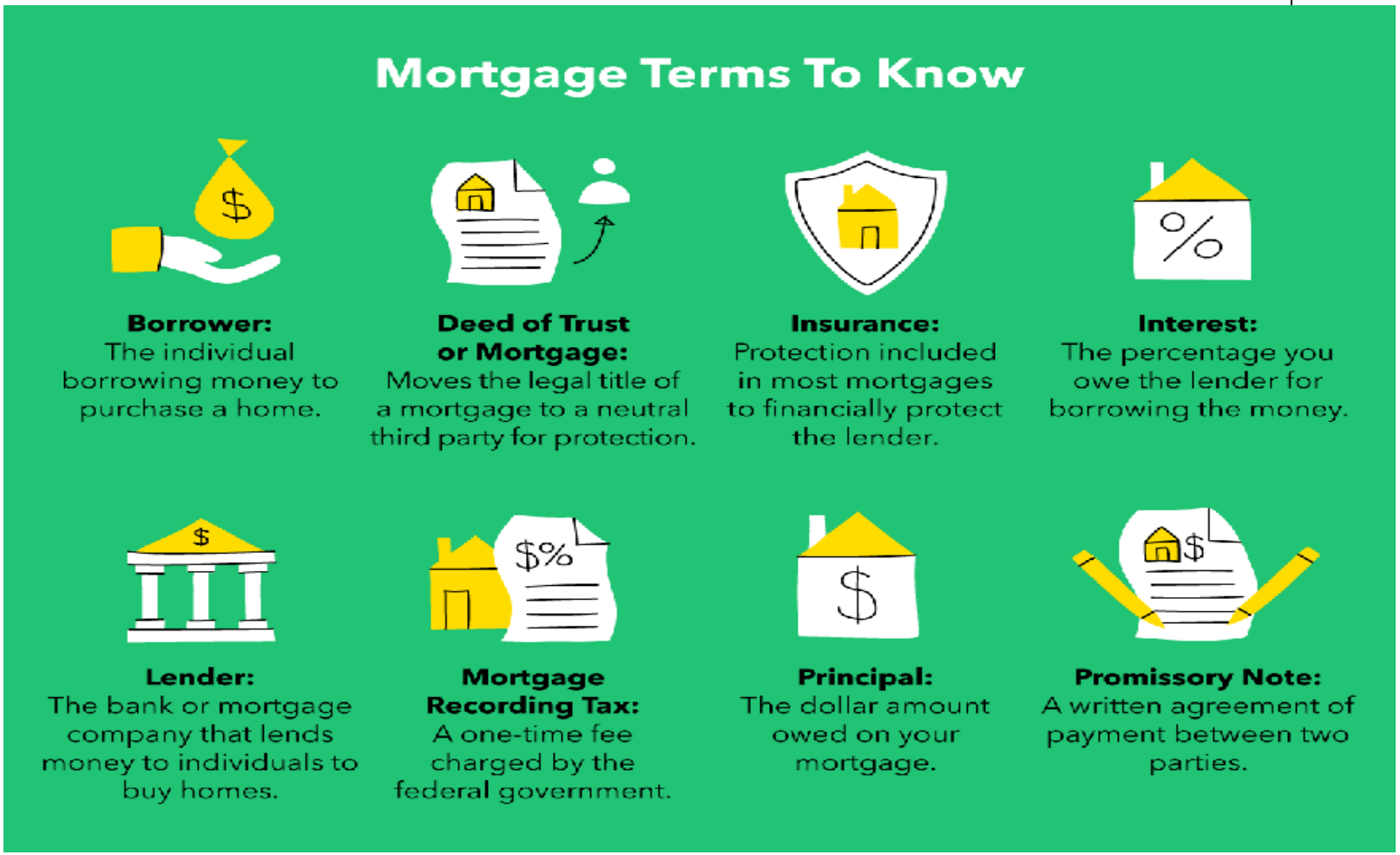
Students must register for 120 credits each year

```
context Student invariant:
takes.credit -> sum( ) = 120
```

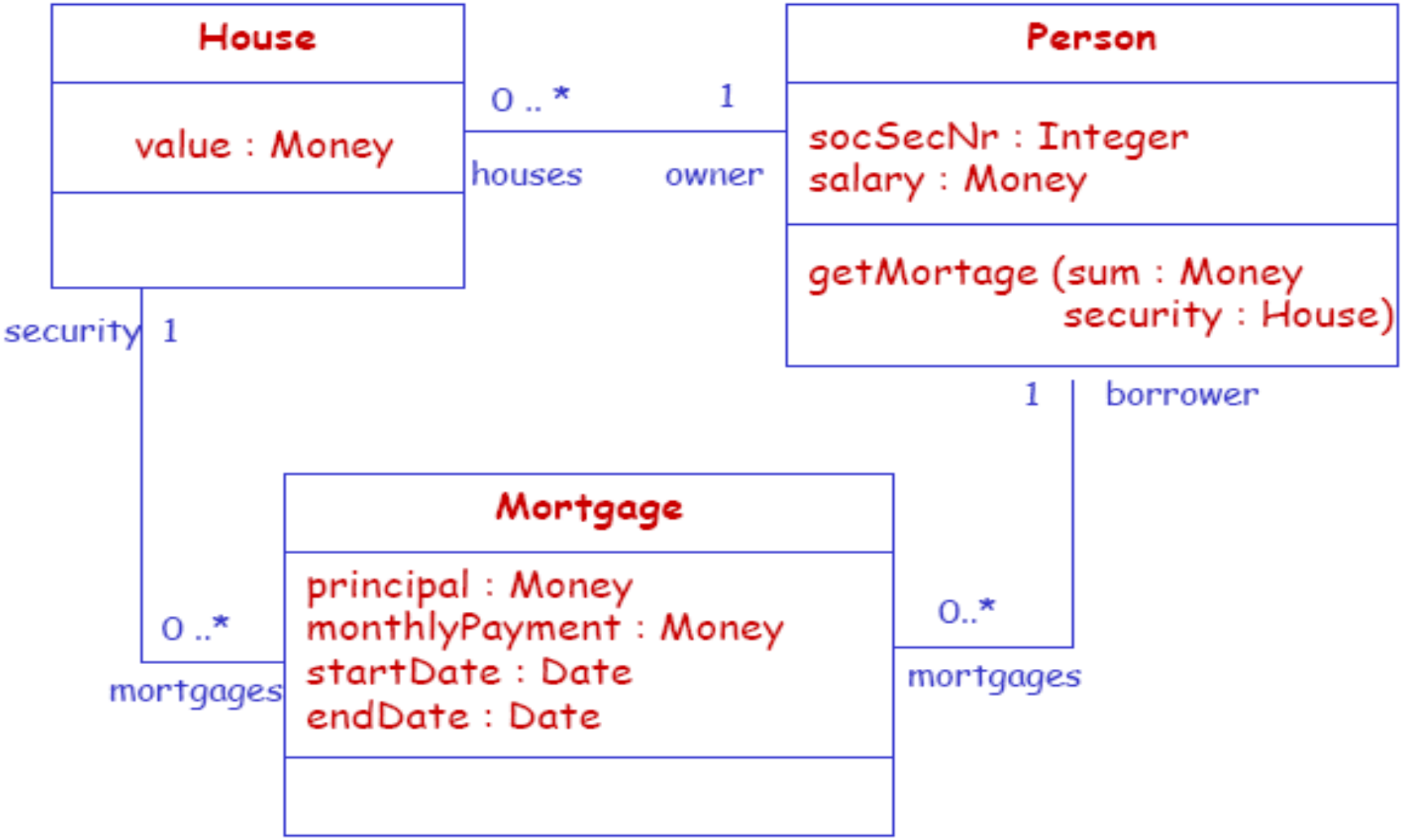
Students must take at least 90 credits of CS modules each year

```
context Student
invariant: takes -> select(code.substring(1,2) =
'CS').credit -> sum( ) >= 90
```

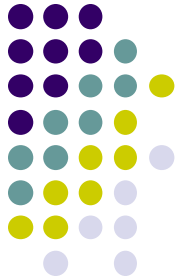
OCL (Object Constraint Language) – The Mortgage Example



OCL (Object Constraint Language) – The Mortgage Example

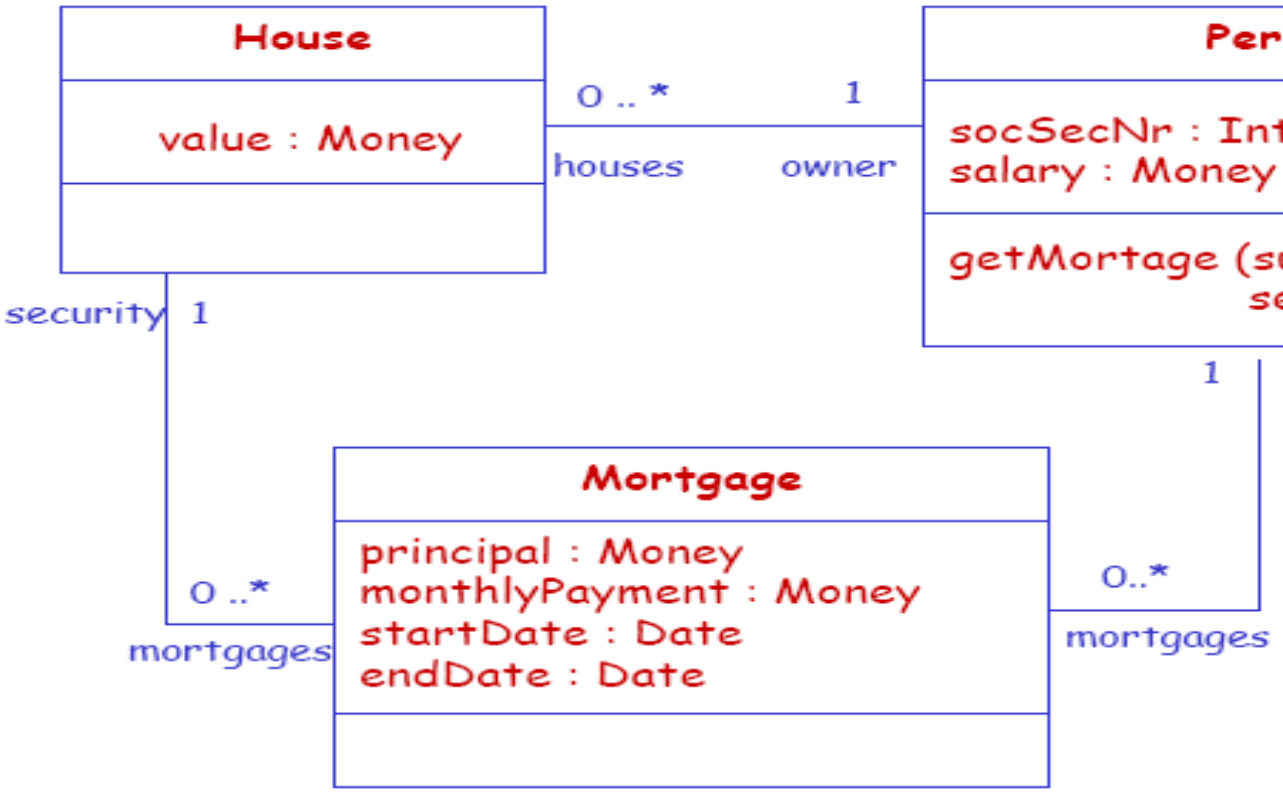


How to express constraints?



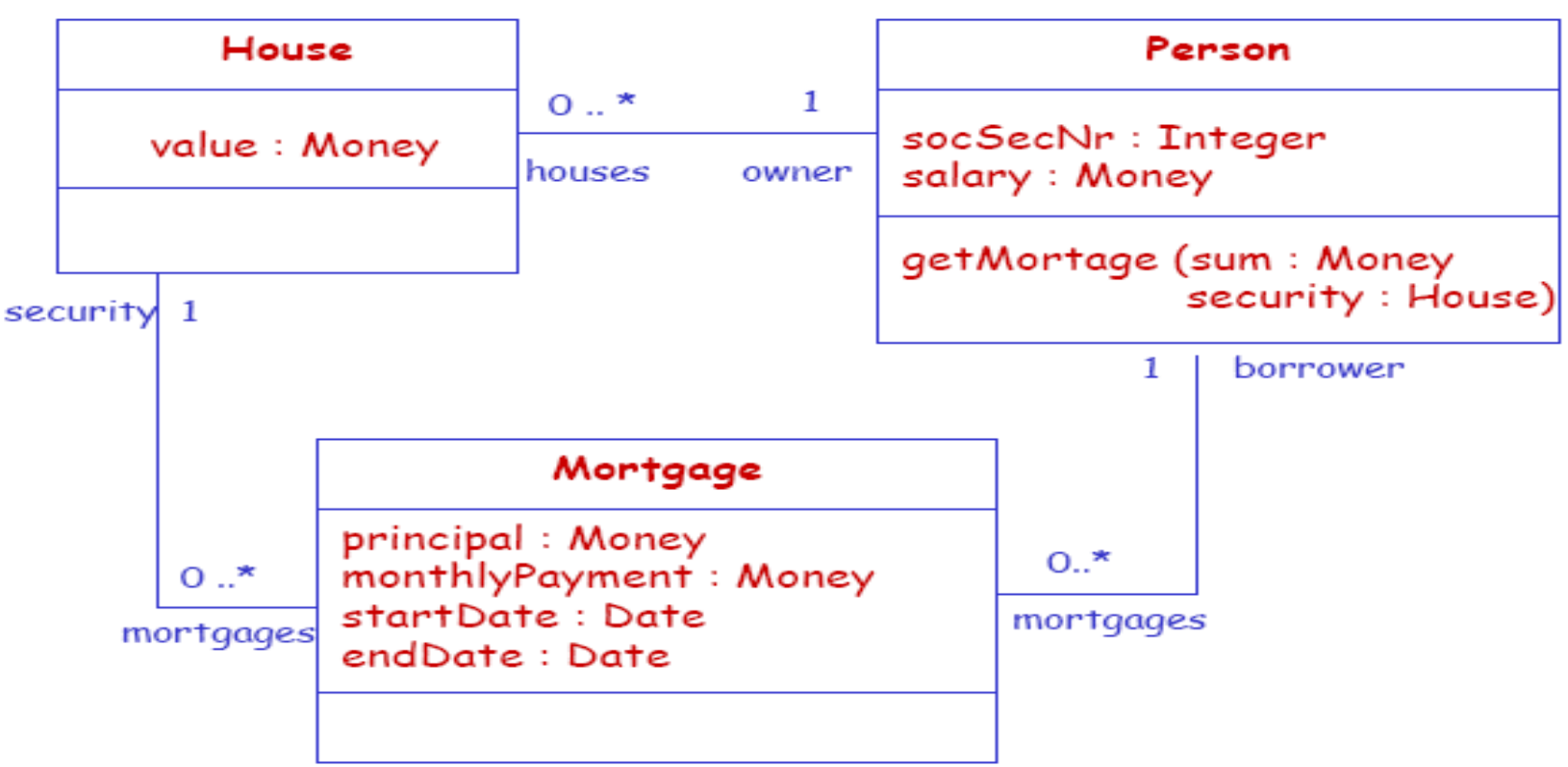
Can we express the following info on the diagrams?

1. A person may have a mortgage on a house only if that house is owned by himself.
2. The start date for any mortgage must be before the end date.
3. The social security number of all persons must be unique.
4. A new mortgage will be allowed only when the person's income is sufficient.
5. A new mortgage will be allowed only when the counter value of the house is sufficient.



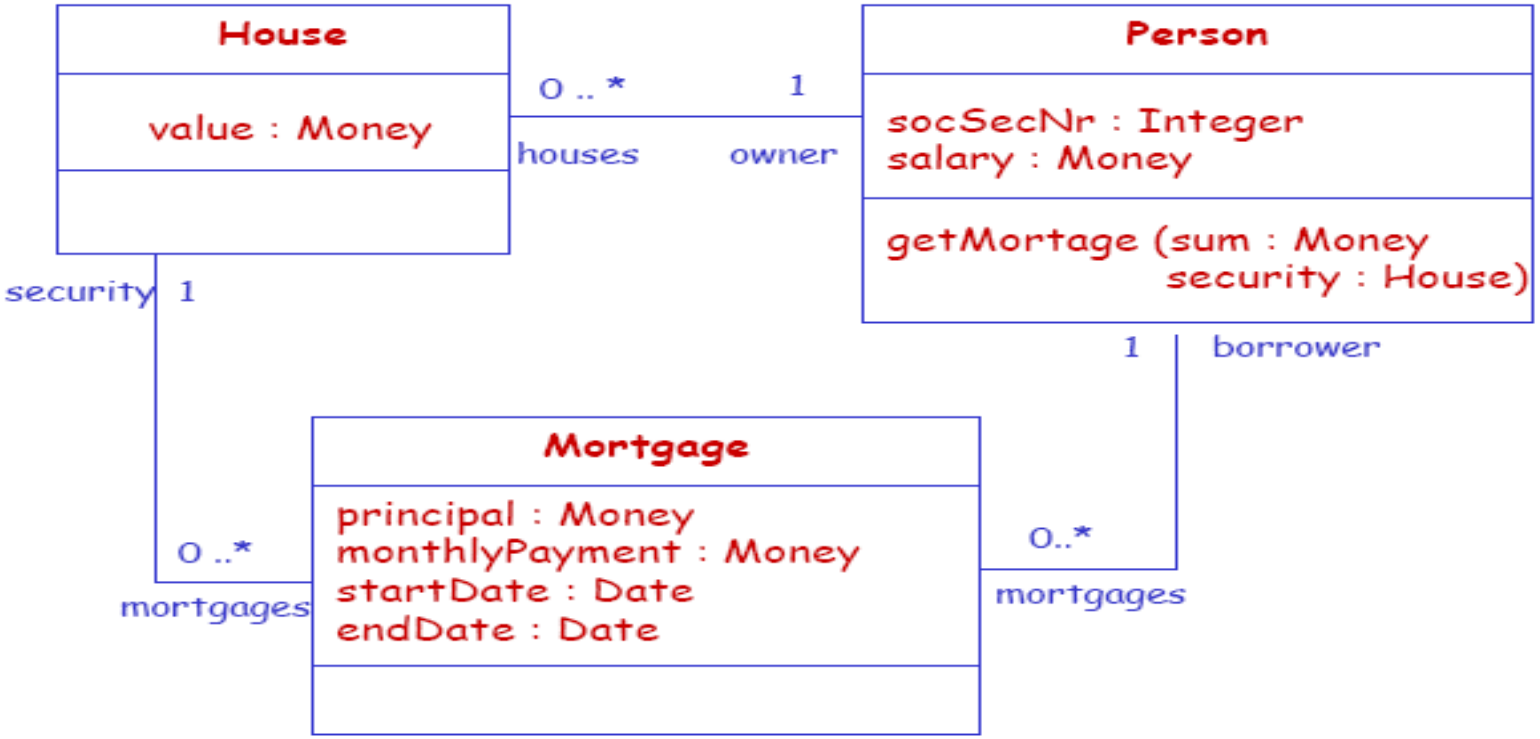
context Mortgage
inv: security.owner = borrower

/* A person may have a mortgage on a house only if that house is owned by himself.*/



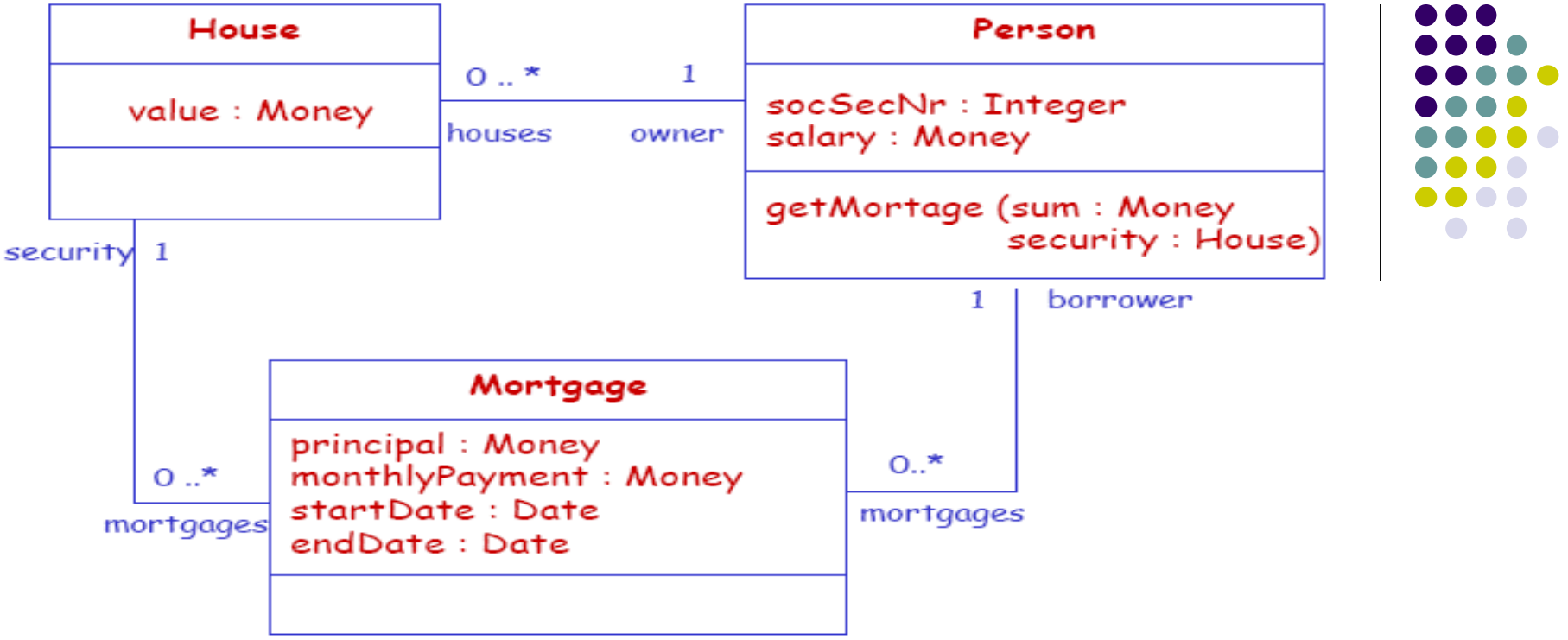
context Mortgage
inv: startDate < endDate

/* The start date for any mortgage must be before the end date.*/



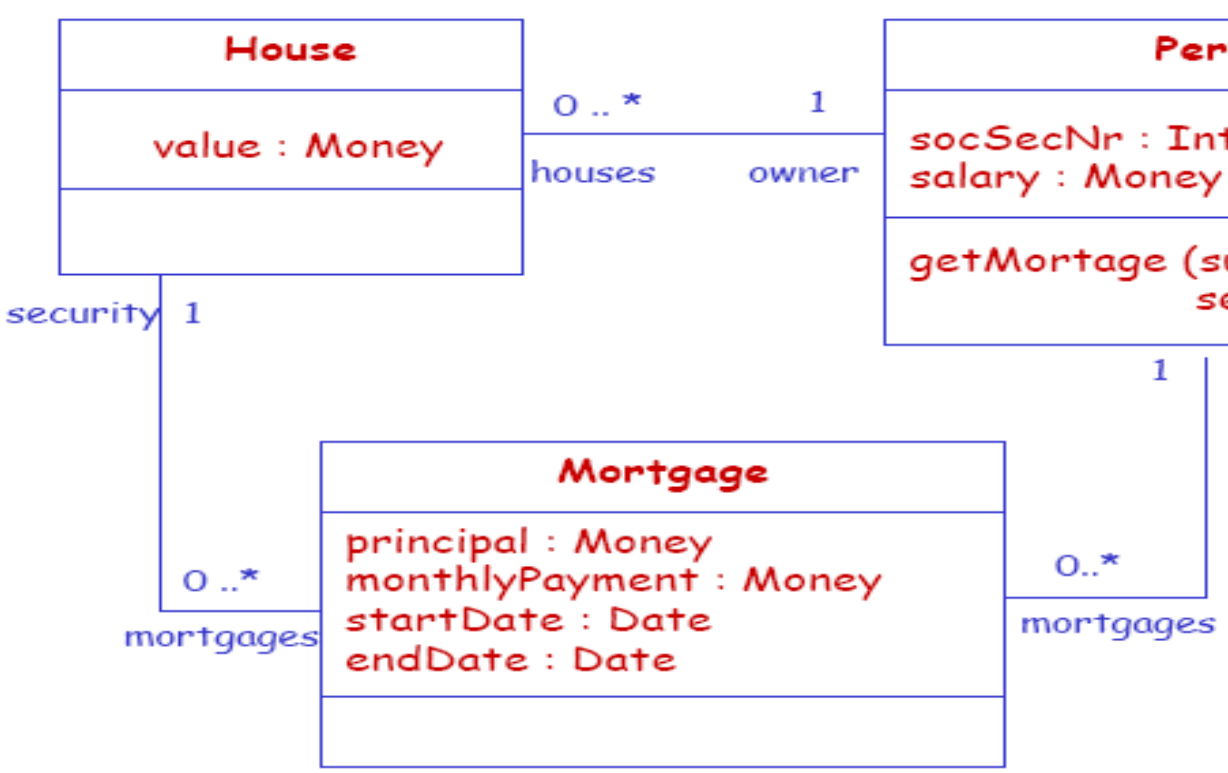
context Person
inv: Person::allInstances() -> isUnique (socSecNr)

/* The social security number of all persons must be unique.*/



context Person::getMortgage(sum: Money, security: House)
Pre: self.mortgages.monthlyPayment -> sum() <=
self.salary * 0.30

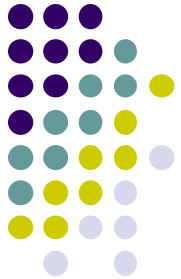
/* A new mortgage will be allowed only when the person's income is sufficient.*/



context Person::getMortgage(sum: Money, security: House)
Pre: security.value >= self.mortgages.principal->sum()

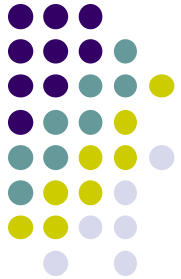
/* A new mortgage will be allowed only when the counter value of the house is sufficient.*/

Constraints Usage



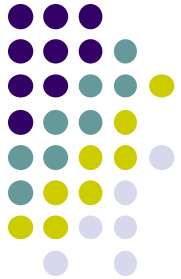
- Avoid any potential misunderstandings
  - Not everyone is aware of these constraints
  - People may make different assumptions.
- Enable automatic model analysis/transform.
  - Computer has no “intuition”.
  - Software tools are possible only if the model contains complete information.
- Document your design decisions.

Characteristics of OCL 1/2



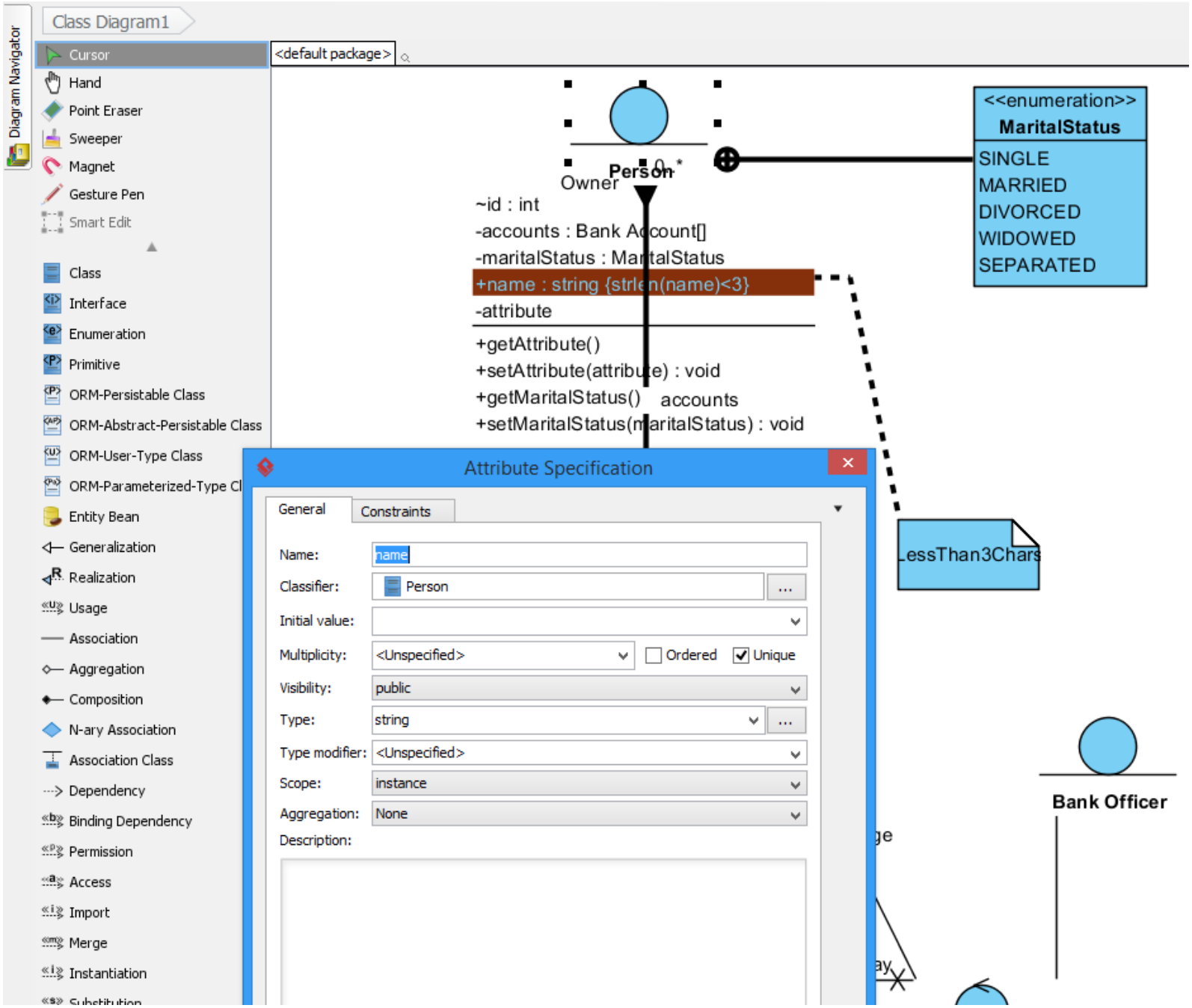
- OCL is a constraint and query language
  - A constraint is a restriction on one or more values of a model.
  - OCL can be used to write not only constraints, but *any query expression*.
  - It is proved that OCL has the same capability as SQL.
- OCL has a formal foundation, but maintain the ease of use.
  - The result is a precise language that should be easily read and written by average developers.

Characteristics of OCL 2/2

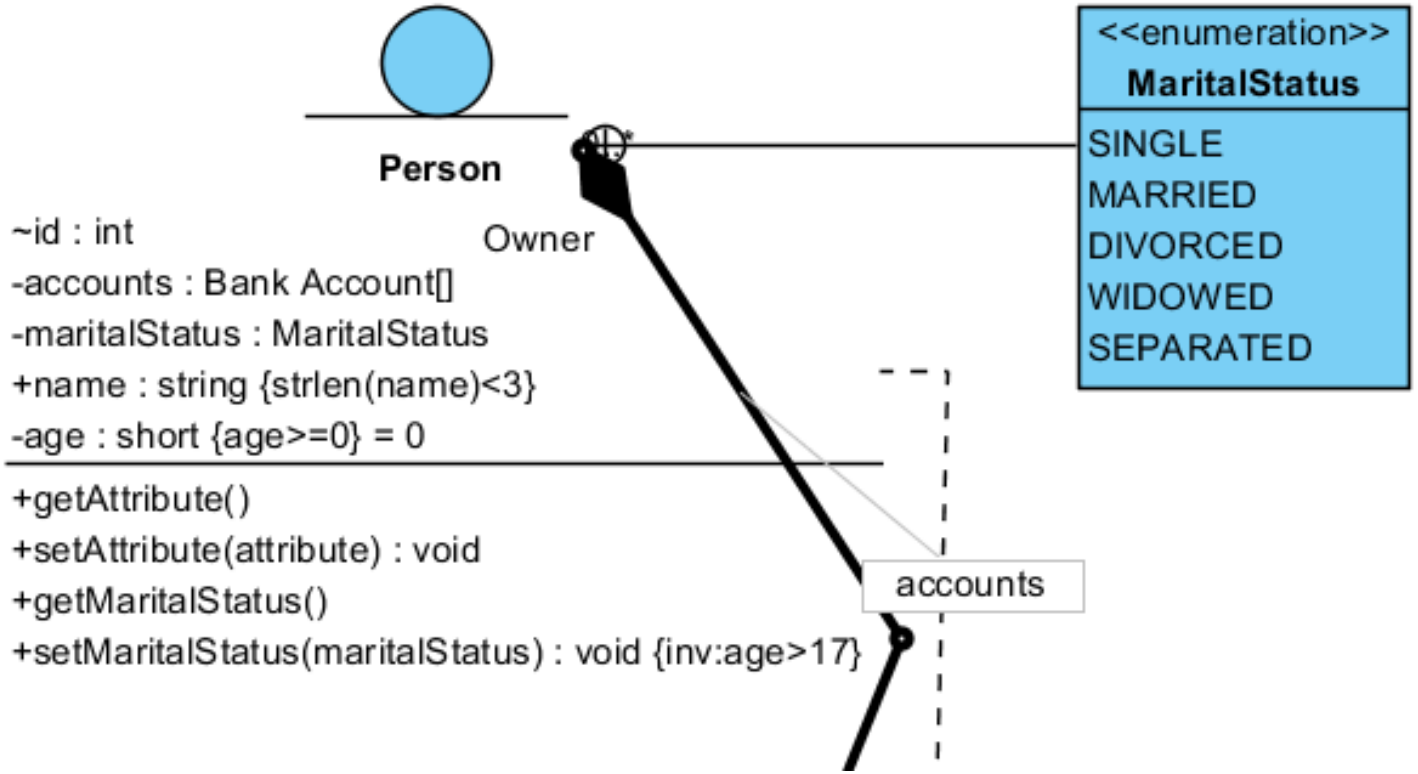


- OCL is strongly-typed
  - This allows OCL expressions can be checked during modeling, before execution.
  - What is the benefit?
- OCL is a declarative language
  - OCL expressions state what should be done, but not how.

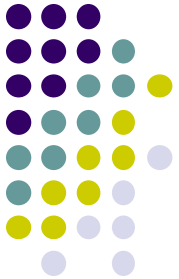
OCL in VP


Class Diagram1
Diagram Navigator
Cursor
Hand
Point Eraser
Sweeper
Magnet
Gesture Pen
Smart Edit
Class
Interface
Enumeration
Primitive
ORM-Persistable Class
ORM-Abstract-Persistable Class
ORM-User-Type Class
ORM-Parameterized-Type Cl
Entity Bean
Generalization
Realization
Usage
Association
Aggregation
Composition
N-ary Association
Association Class
Dependency
Binding Dependency
Permission
Access
Import
Merge
Instantiation
<default package>
Owner
Person
0..*
~id : int
-accounts : Bank Account[]
-maritalStatus : MaritalStatus
+name : string {strlen(name)<3}
-attribute
+getAttribute()
+setAttribute(attribute) : void
+getMaritalStatus()
+setMaritalStatus(maritalStatus) : void
accounts
<<enumeration>>
MaritalStatus
SINGLE
MARRIED
DIVORCED
WIDOWED
SEPARATED
LessThan3Chars
Attribute Specification
General
Constraints
Name:
name
Classifier:
Person
Initial value:
Multiplicity:
<Unspecified>
Ordered
Unique
Visibility:
public
Type:
string
Type modifier:
<Unspecified>
Scope:
instance
Aggregation:
None
Description:
Bank Officer

OCL for age and setMeritalStatus



More about OCL

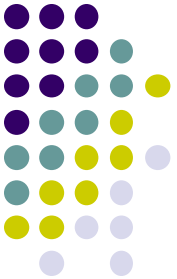


- **UML 2.0 OCL Specification -** http://www.lri.fr/~wolff/teach-material/2008-09/IFIPS-VnV/UML2.0OCL-specification.pdf
- **The university model - An UML/OCL example** - http://dresden-ocl.sourceforge.net/usage/ocl22sql/modelexplanation.html
- **Verification of UML/OCL Class Diagrams using Constraint Programming,** by J. Cabot

Timing Diagrams (UML 2.0)

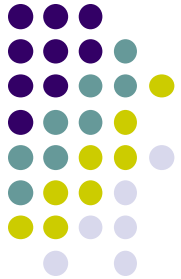
Source for these slides:

Learning UML 2.0,
by Kim Hamilton, Russell Miles
.....
Publisher: **O'Reilly**
Pub. Date: **April 2006**



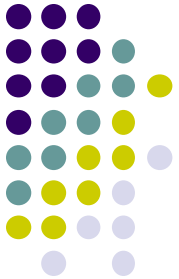
- **Interaction diagrams:**
  - sequence diagrams focus on message order
  - communication diagrams show the links between participants
  - *But: we need interaction diagrams to model detailed timing information!*
- **In timing diagrams:**
  - each event has timing information associated with it

Event Timing Information:

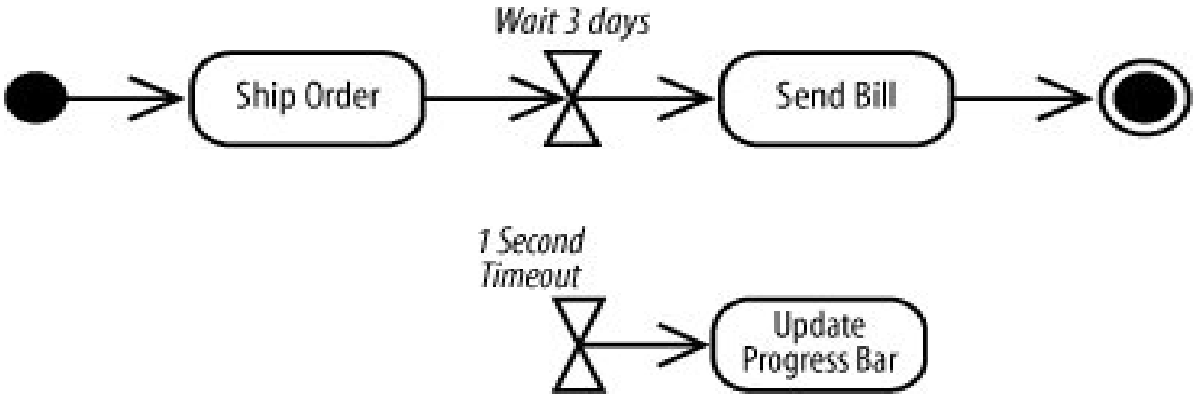


- Describes:
  - when the event is invoked,
  - how long it takes for another participant to receive the event, and
  - how long the receiving participant is expected to be in a particular state.
- Event timing could be expressed within activity diagrams (UML 2.x).

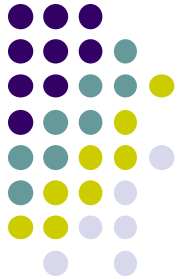
Time Events (slide from previous lesson)



- A time event with no incoming flows models a repeating time event

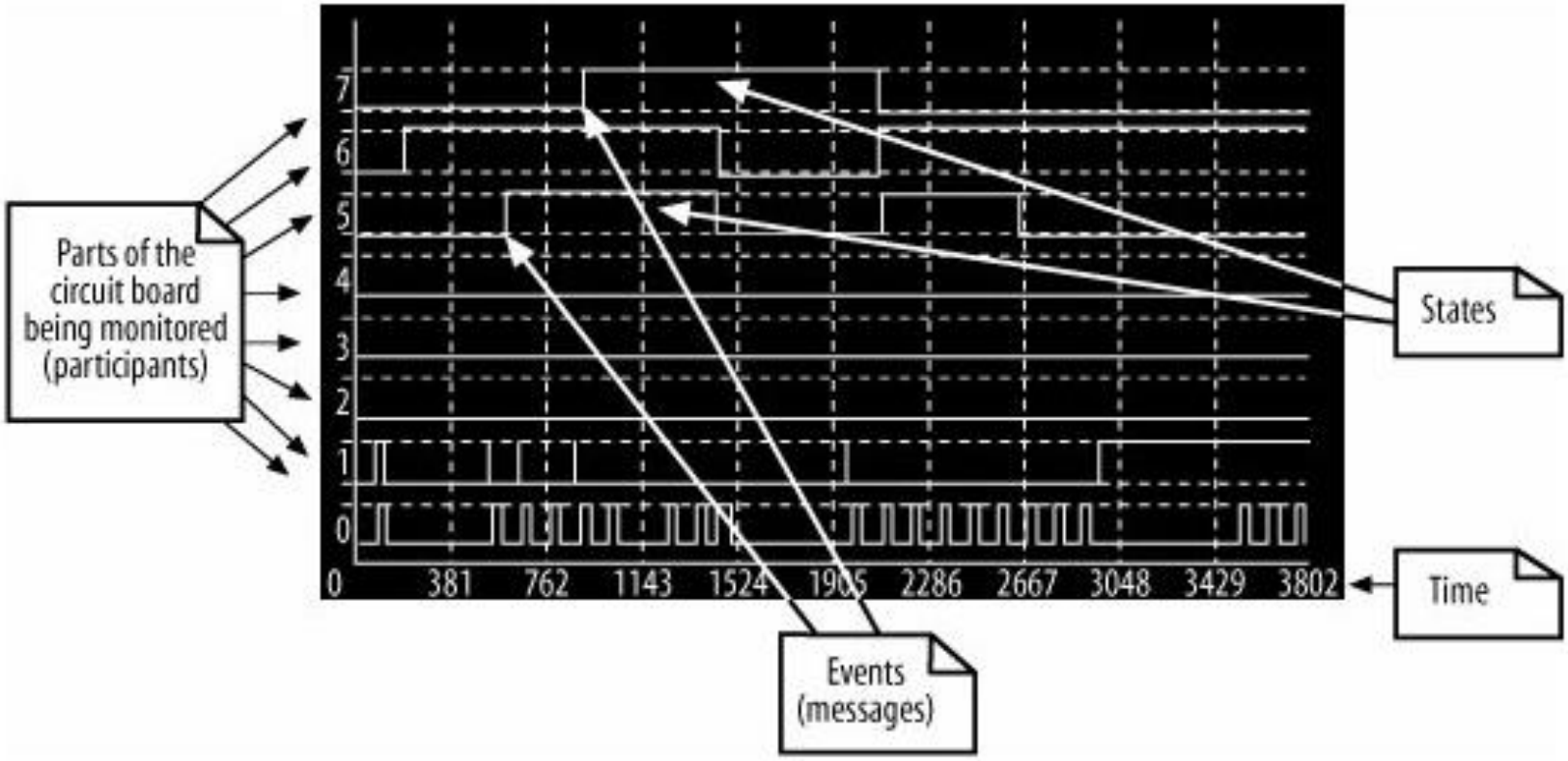


Need of Timing Diagrams



- Although sequence diagrams and communication diagrams are very similar,
- timing diagrams add completely new information
- that is not easily expressed on any other form of UML interaction diagram.

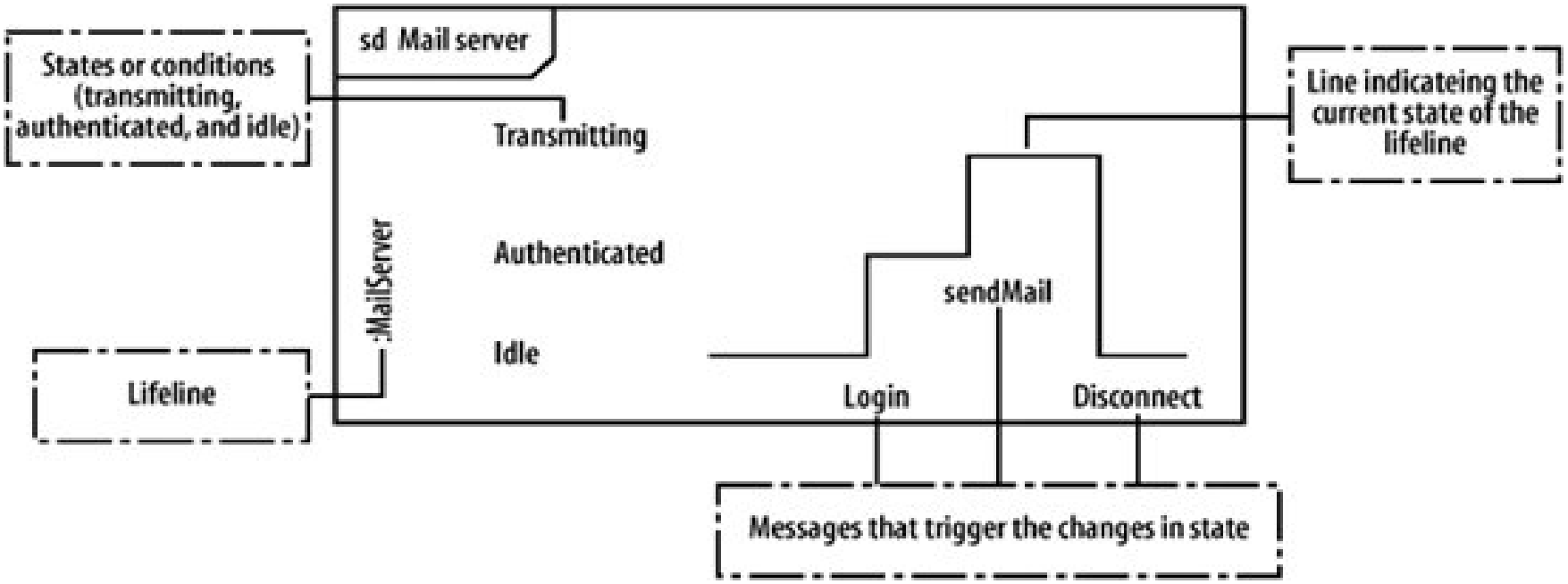
Introducing timing: osciloscope views



A logic analyzer captures a sequence of events as they occur on an electronic circuit board

Events and states on timing diagrams

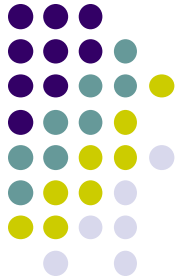
- On a timing diagram:
  - events are the logic analyzer's signals, and
  - states are the states that a participant is placed in



Sample timing diagram for a mail server

Source: UML 2.0 in a Nutshell (O'Reilly).

From use case and requirements…

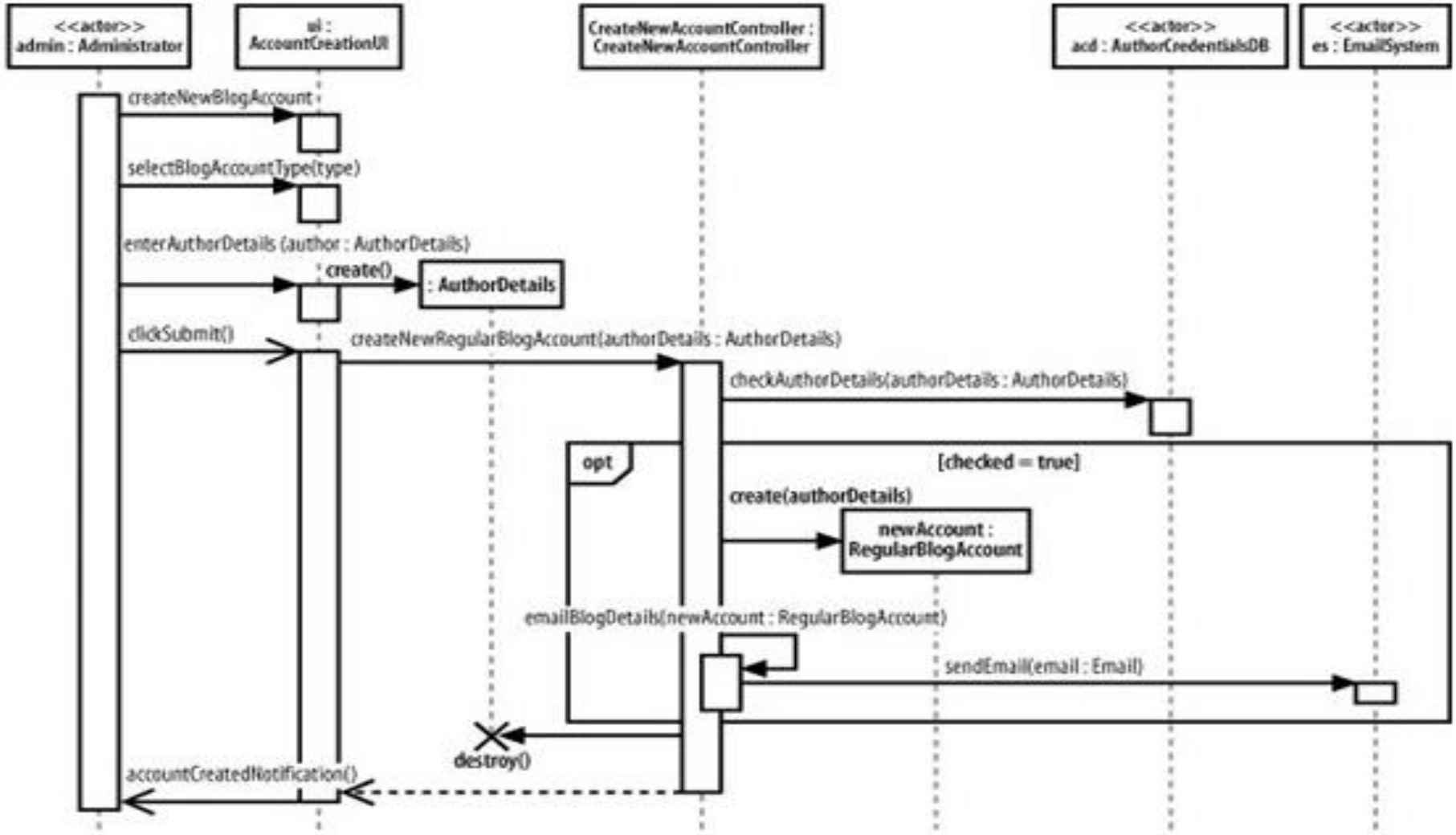


Sample use case: Create a new Regular Blog Account

Requirement A.2

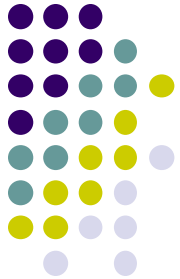
- The content management system shall allow an administrator to create a new regular blog account, provided the personal details of the author are verified using the Author Credentials Database.

…To sequence diagrams – sequential ordering…



Create a new Regular Blog Account interaction

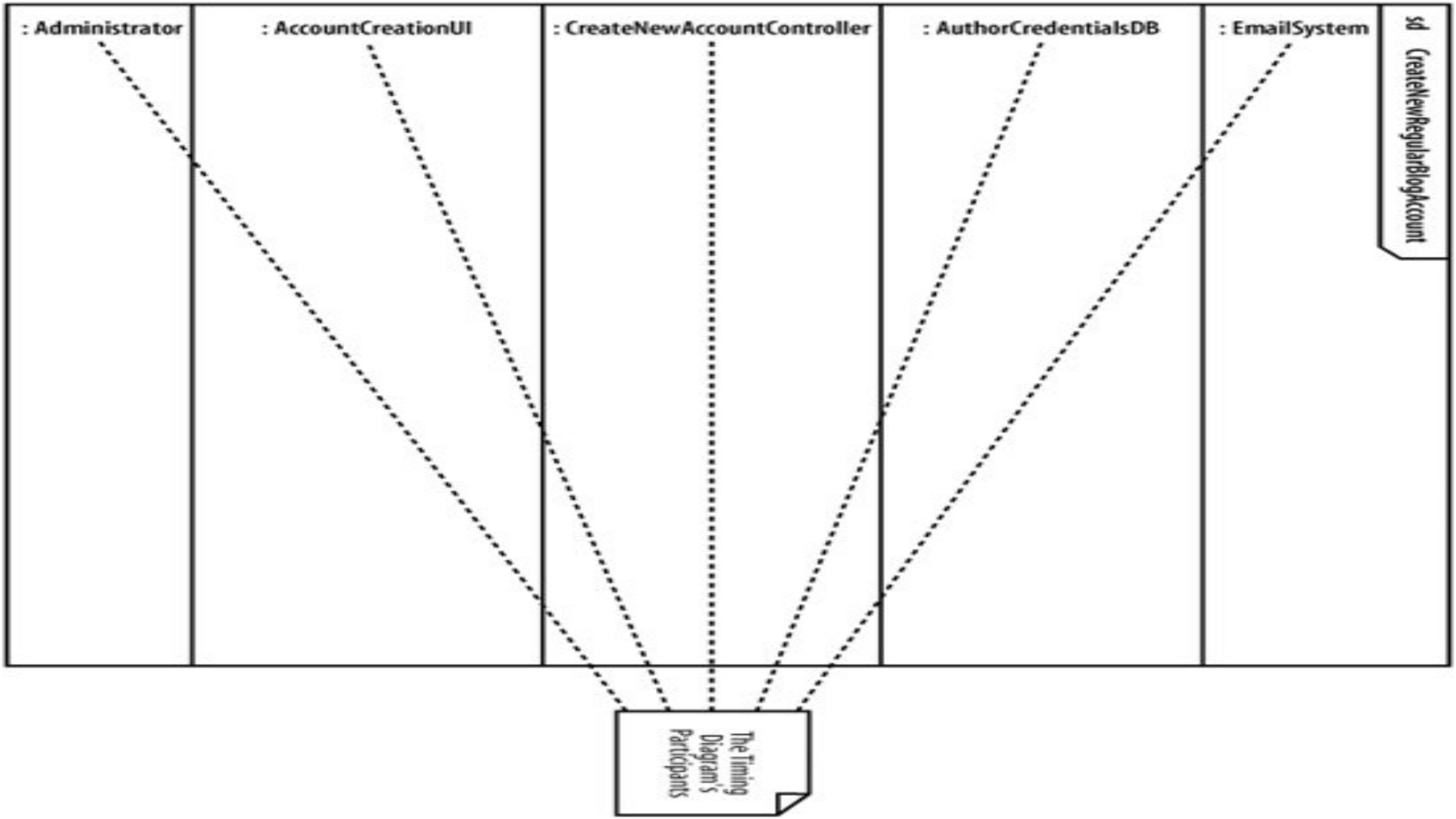
…Through Adding Timing Constraints in System Requirements…



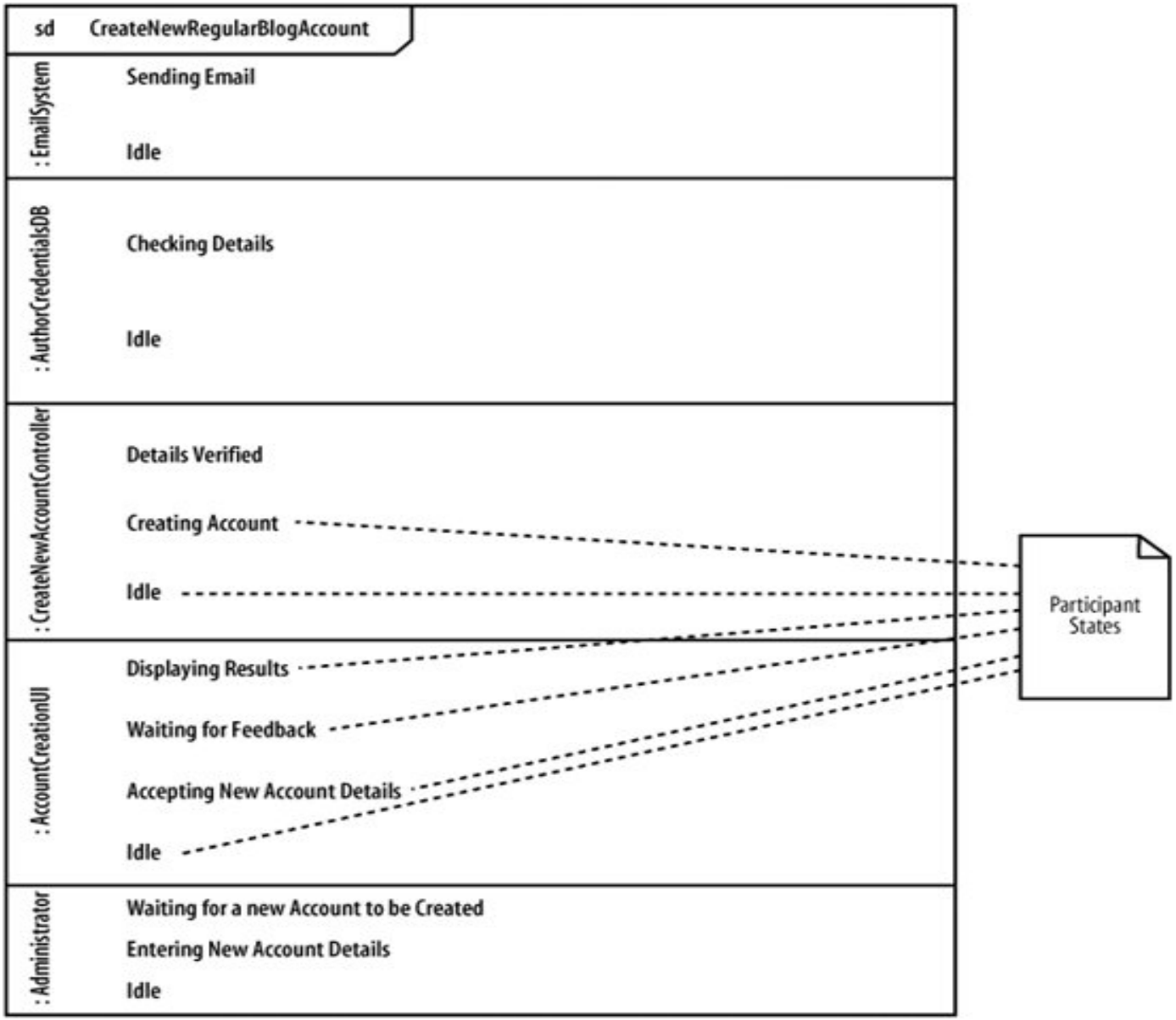
Requirement A.2 (Updated)

- The content management system shall allow an administrator to create a new regular blog account *within five seconds* of the information being entered,
- *provided the personal details of the author are verified* using the Author Credentials Database.

…To a Timing Diagram – first define the Participants

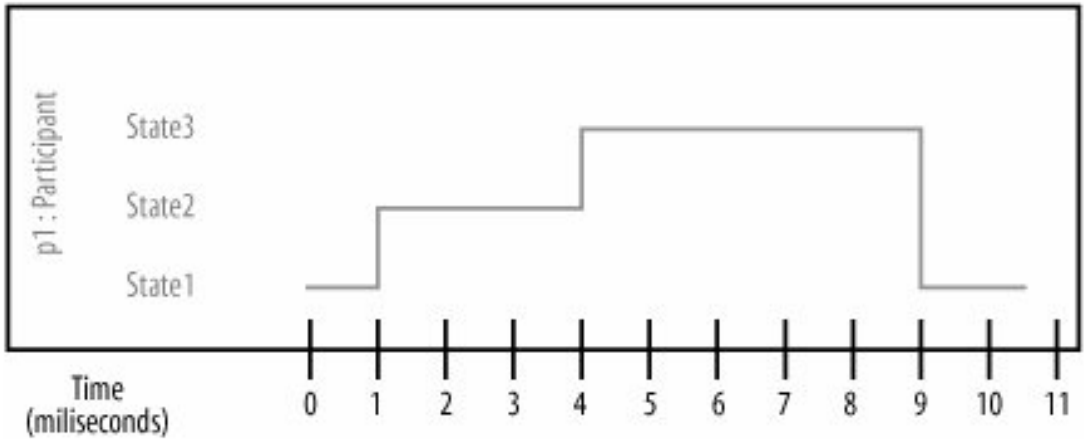


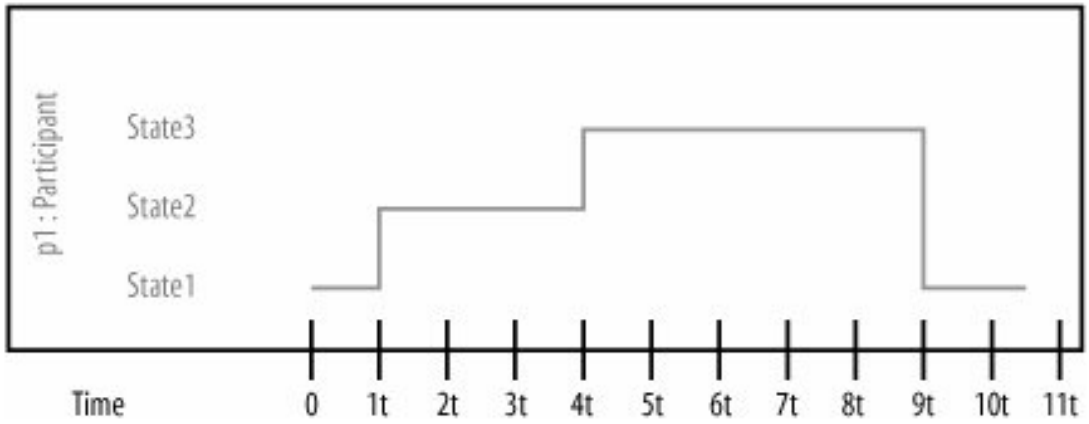
Next – add States



States **are written horizontally on a timing diagram and next to the** ***participant*** **that they are associated with**

Exact Time Measurements and Relative Time Indicators

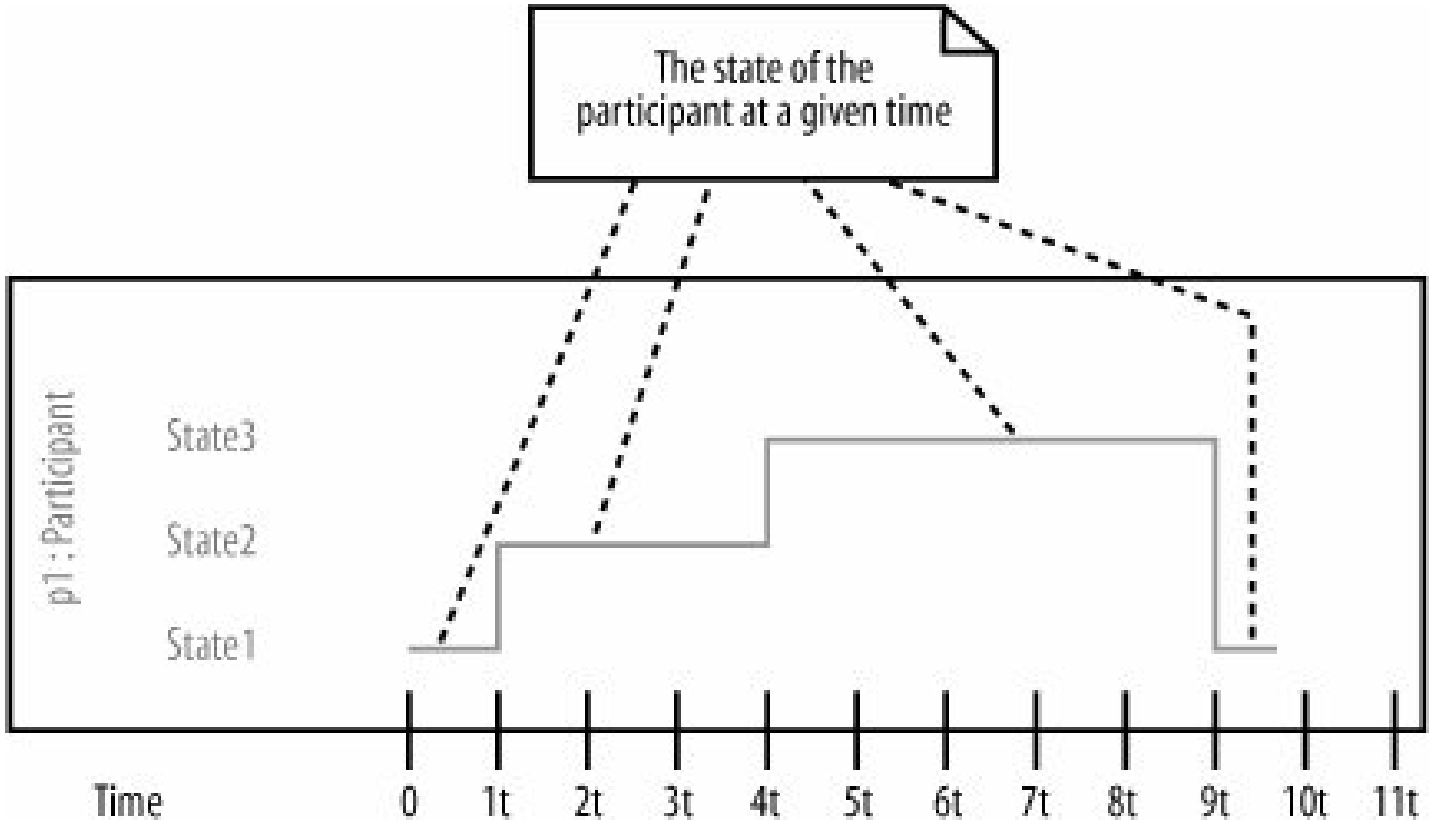




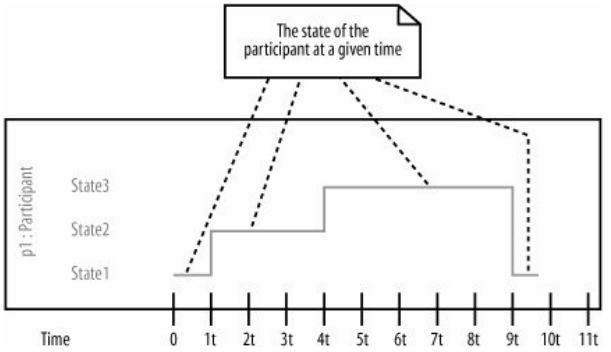
- **Time measurements are placed on a timing diagram as a ruler along the bottom of the page**

- **Relative time indicators are particularly useful when you have timing considerations such as "ParticipantA will be in State1 for half of the time that ParticipantB is in State2"**

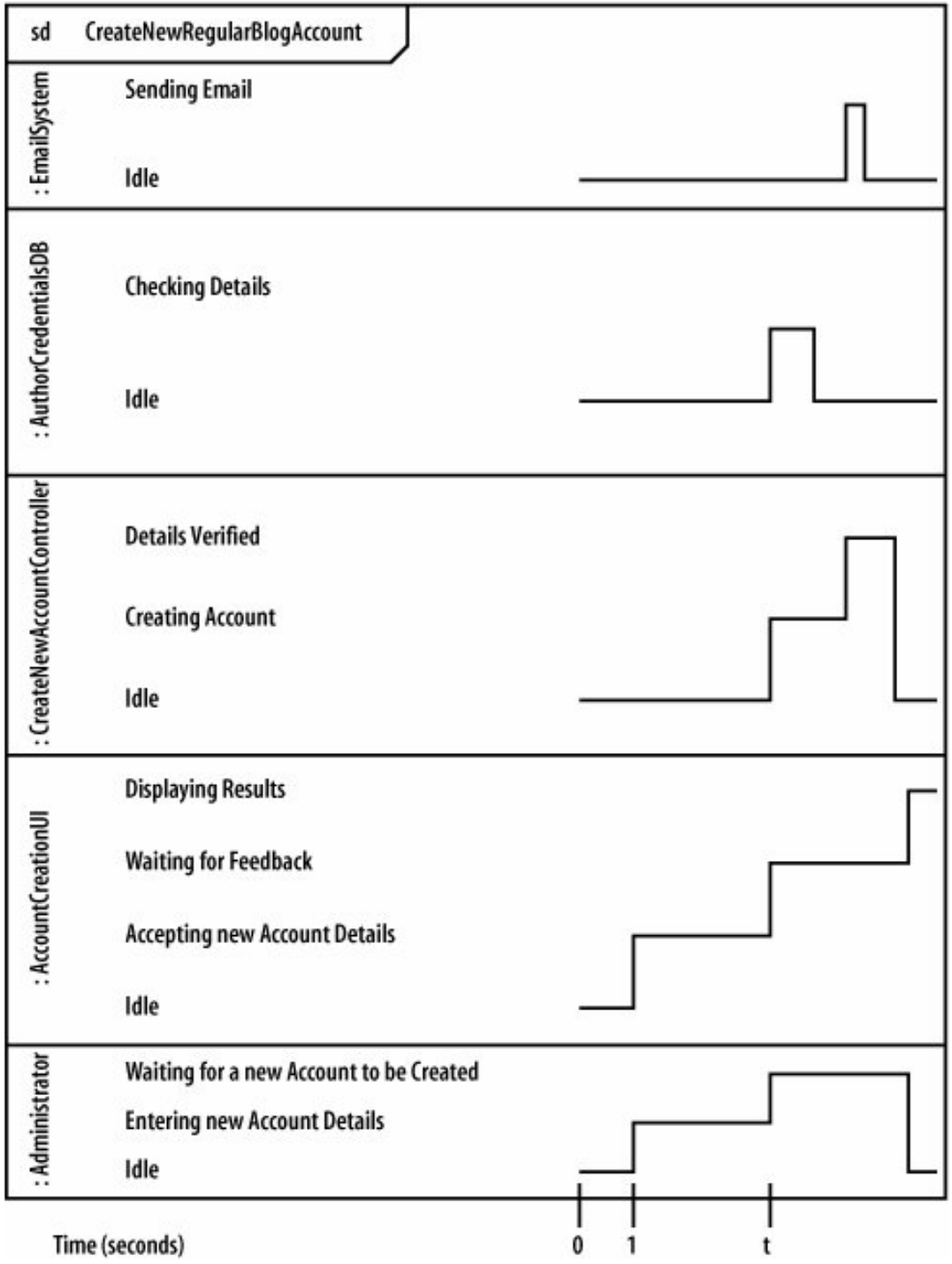
The Participant's State-Line 1/2



The Participant's State-Line 2/2



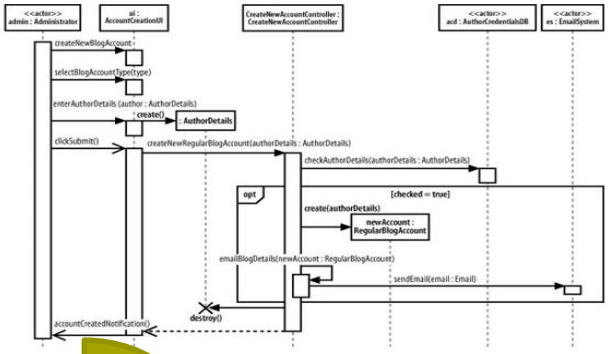
- Create a new Regular Blog Account timing diagram - updated to show the state of each participant at a given time during the interaction.
- p1:Participant's state-line indicates that it is in State1 for 1 unit of time, State2 for three units of time, and State3 for roughly five units of time (before returning to State1 at the end of the interaction)
- In practice, you would probably add both events and states to a timing diagram at the same time.

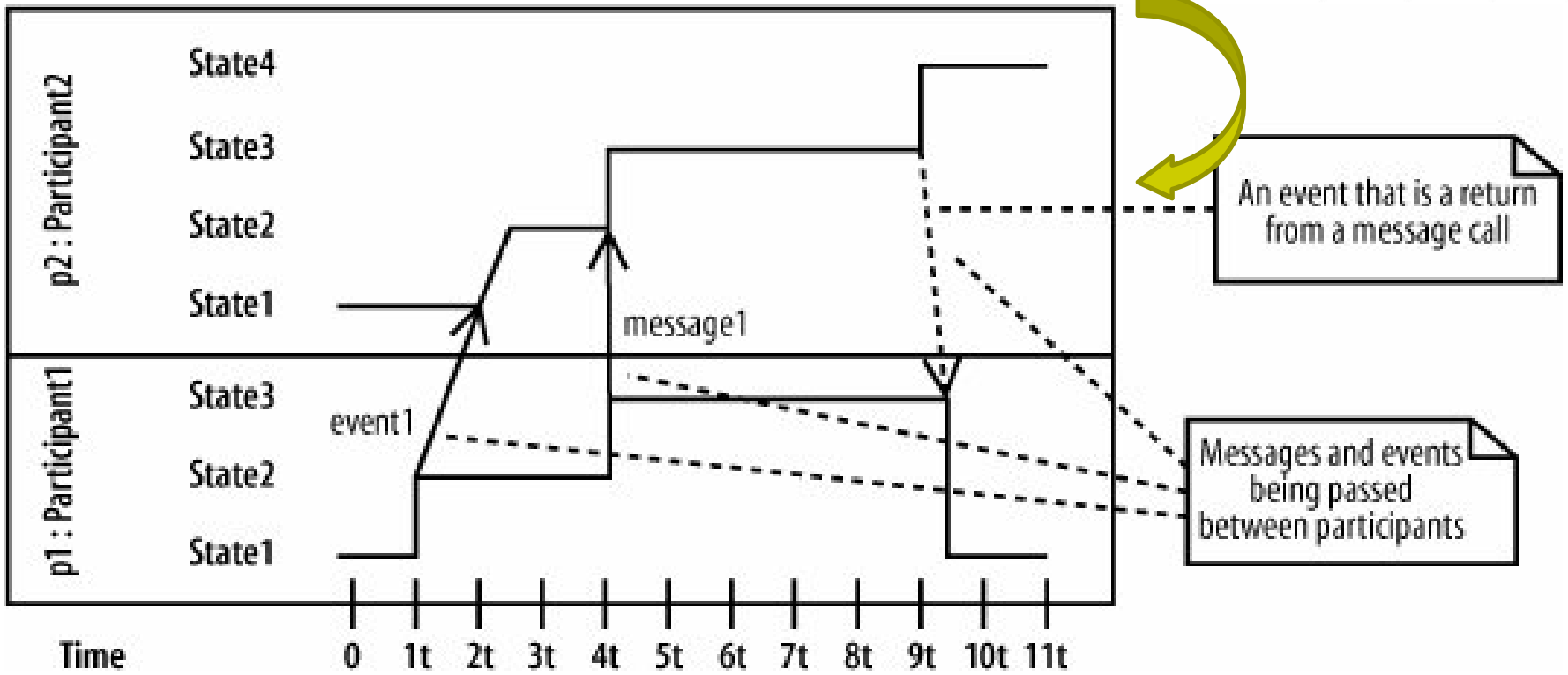


Create a new Regular Blog Account timing diagram

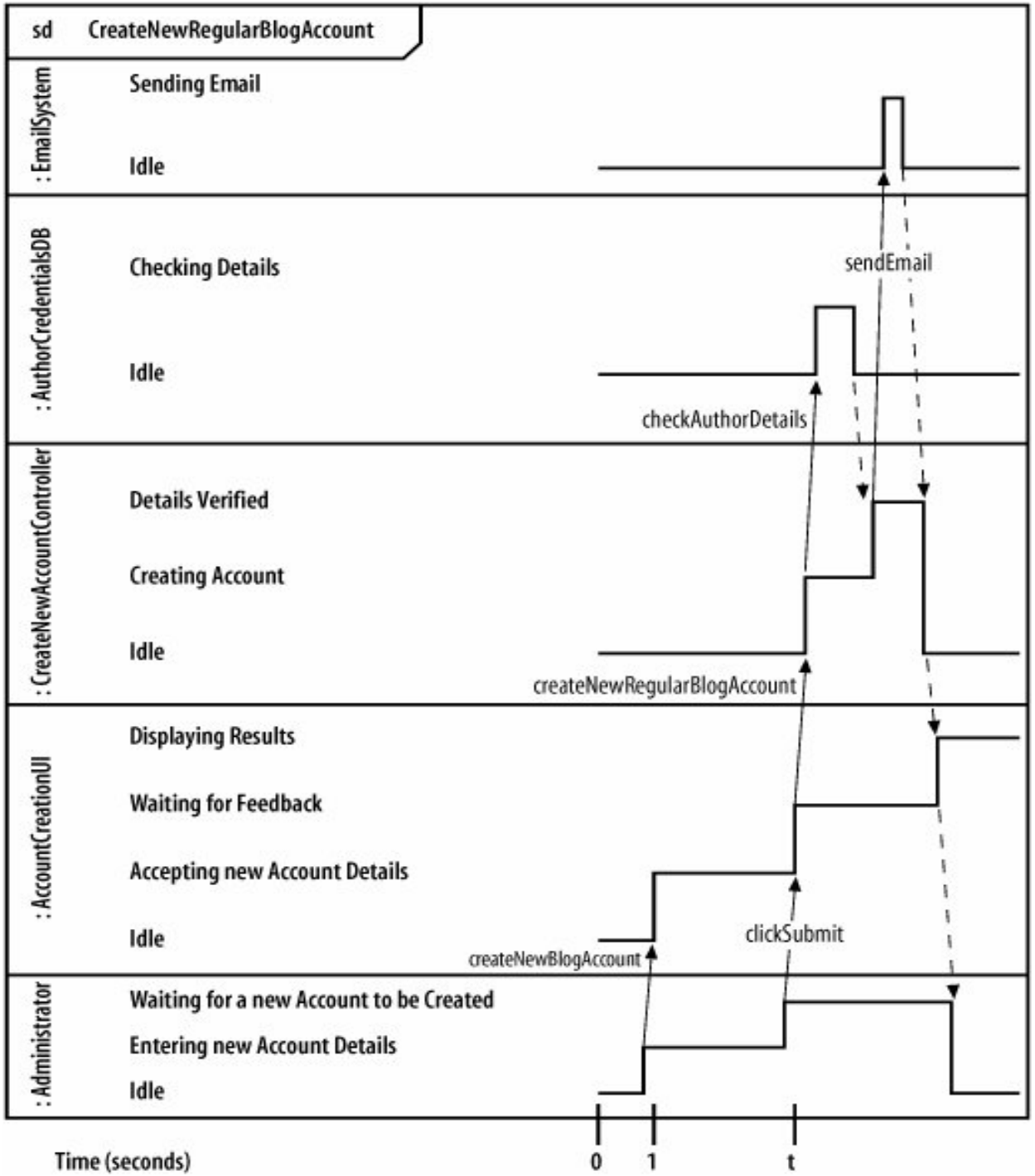
(the single **t** value below represents a single second wherever it is mentioned on any further timing constraints on the diagram)

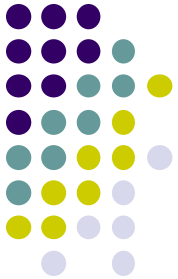
Adding events and messages





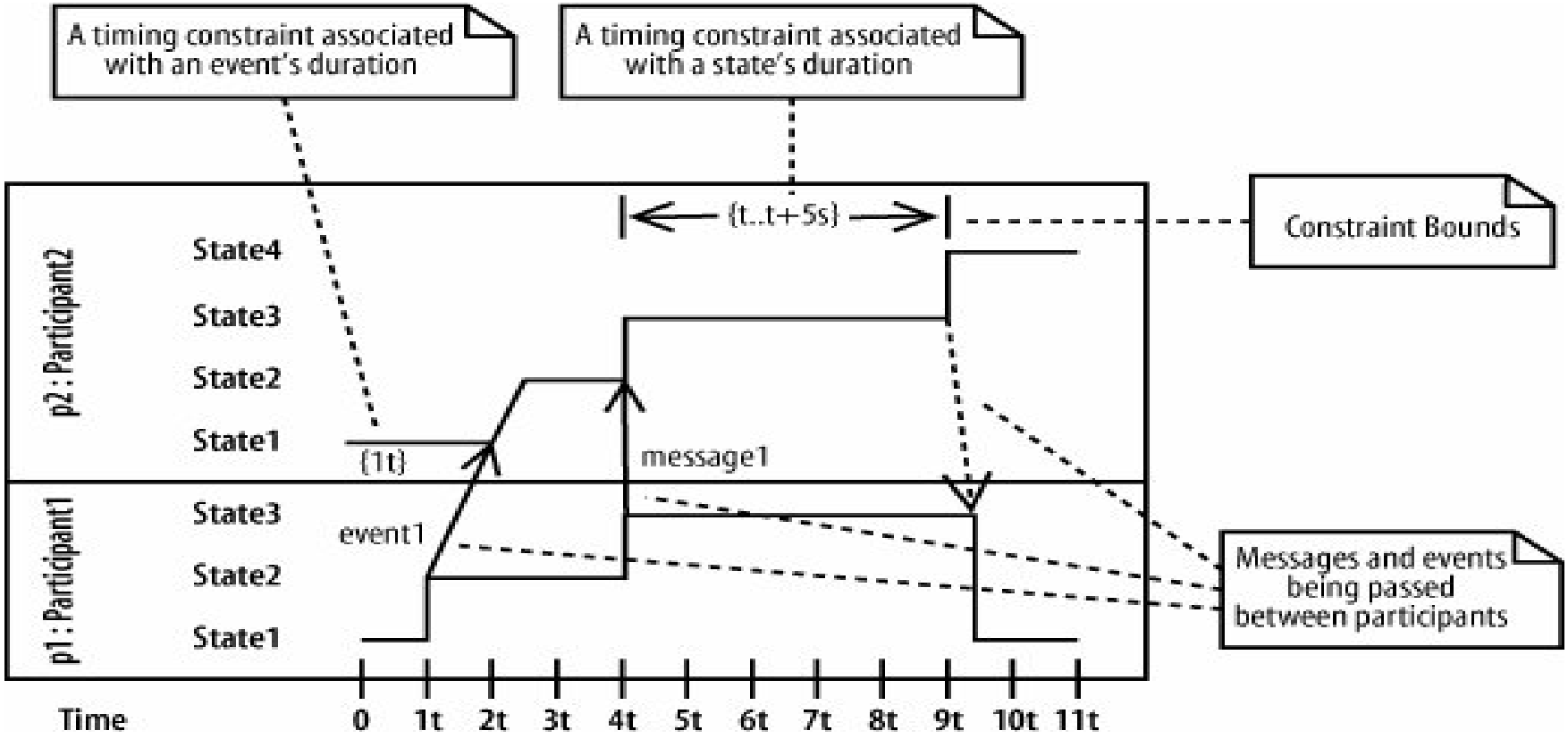
Events on a timing diagram can even have their own durations, as shown by event1 taking 1 unit of time from invocation by p1:Participant1 and reception by p2:Participant2





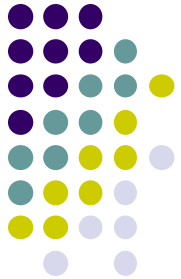
Participant state changes make much more sense when you can see the events that cause them

Timing Constraints

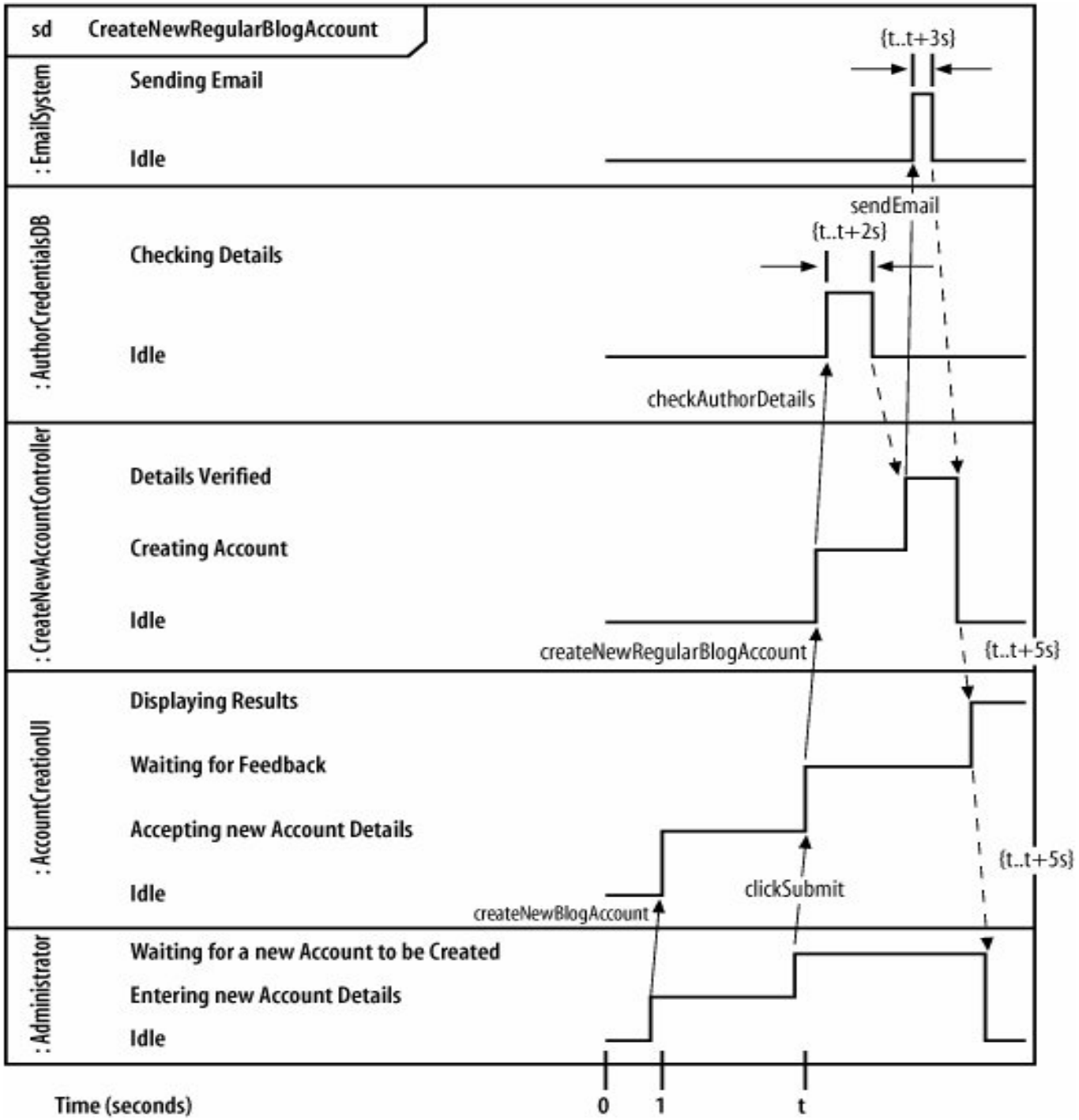


Timing constraints can be associated with an event or a state and may or may not be accompanied by constraint boundary arrows

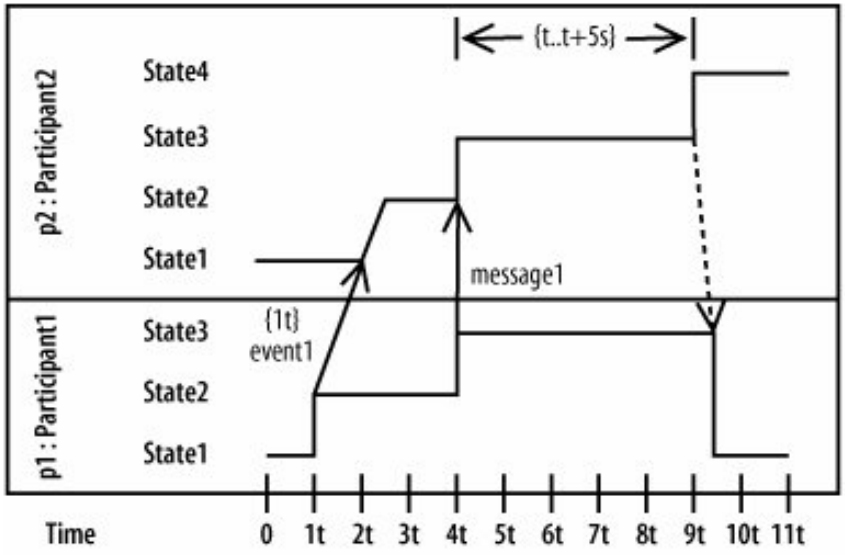
Timing Constraints Format



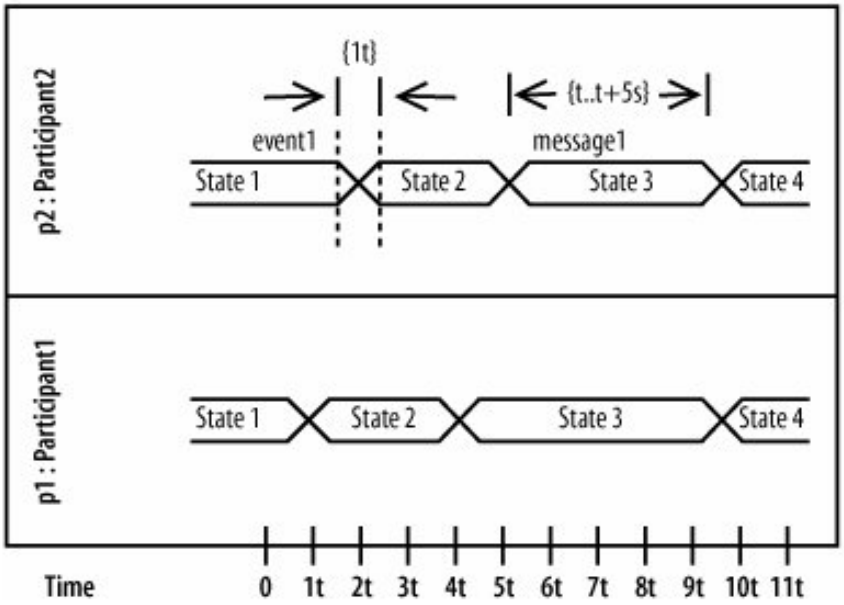
Timing Constraint	Description
{t..t+5s}	The duration of the event or state should be 5 seconds or less.
{<5s}	The duration of the event or state should be less than 5 seconds. This is a slightly less formal than {t..t+5s}.
{>5s, <10s}	The duration of the event or state should be greater than 5 seconds, but less than 10 seconds.
{t}	The duration of the event or state should be equal to the value of t. This is a relative measure, where t could be any value of time.
{t..t*5}	The duration of the event or state should be the value of t multiplied 5 times. This is another relative measure (t could be any value of time).



From when the :Administrator clicks on submit until the point at which the system has created a new account, no more than five seconds have passed



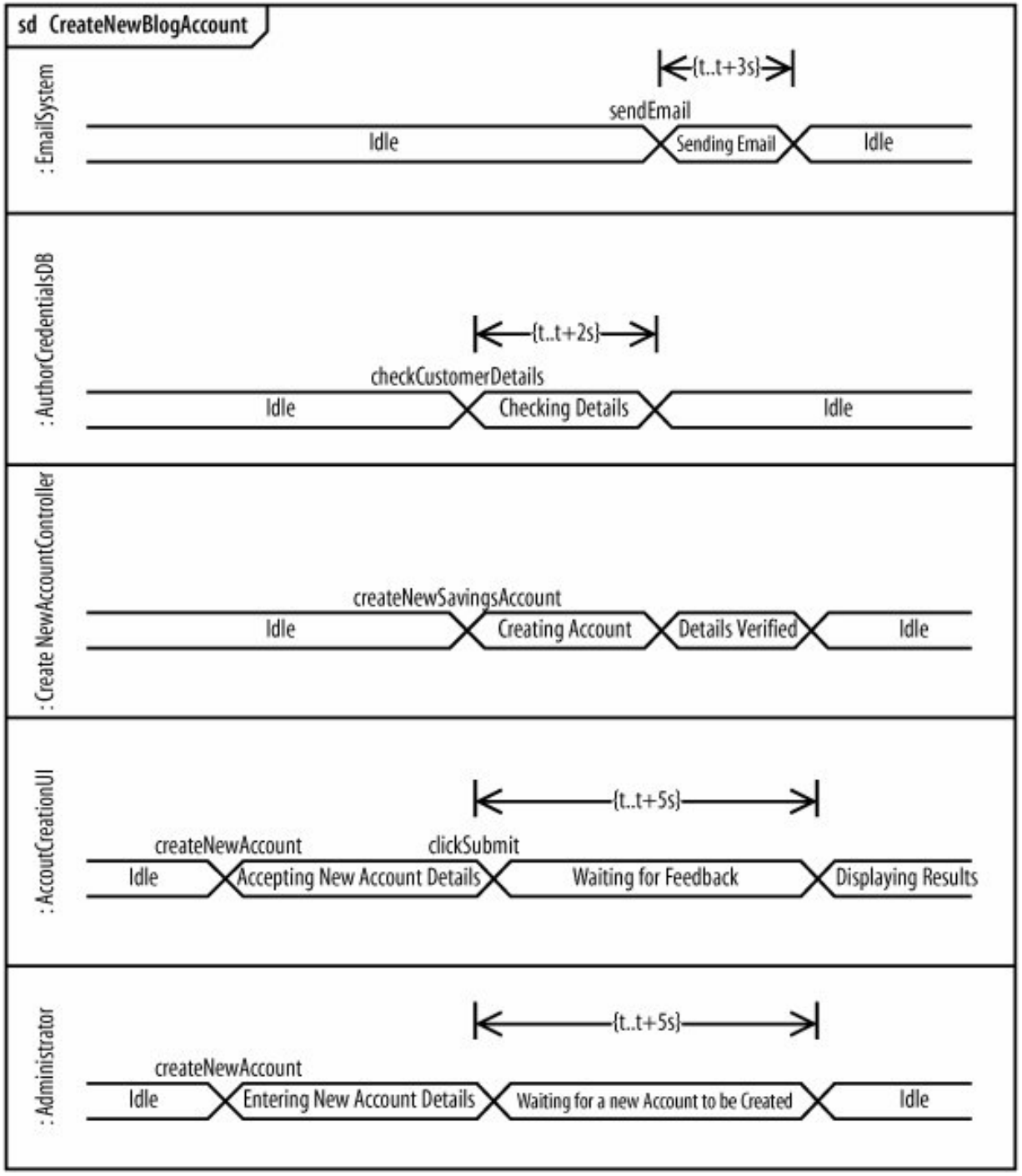
The regular Timing Diagram notation : good when *fewer states* need to be shown.



The alternative Timing Diagram Notation : good when *many states* need to be shown.

- The regular timing diagram notation (over) does not scale well when you have many participants that can be put in many different states during an interaction's lifetime.

- If a participant is placed in many different states during the course of the interaction, then it is worth considering using the alternative notation (below).

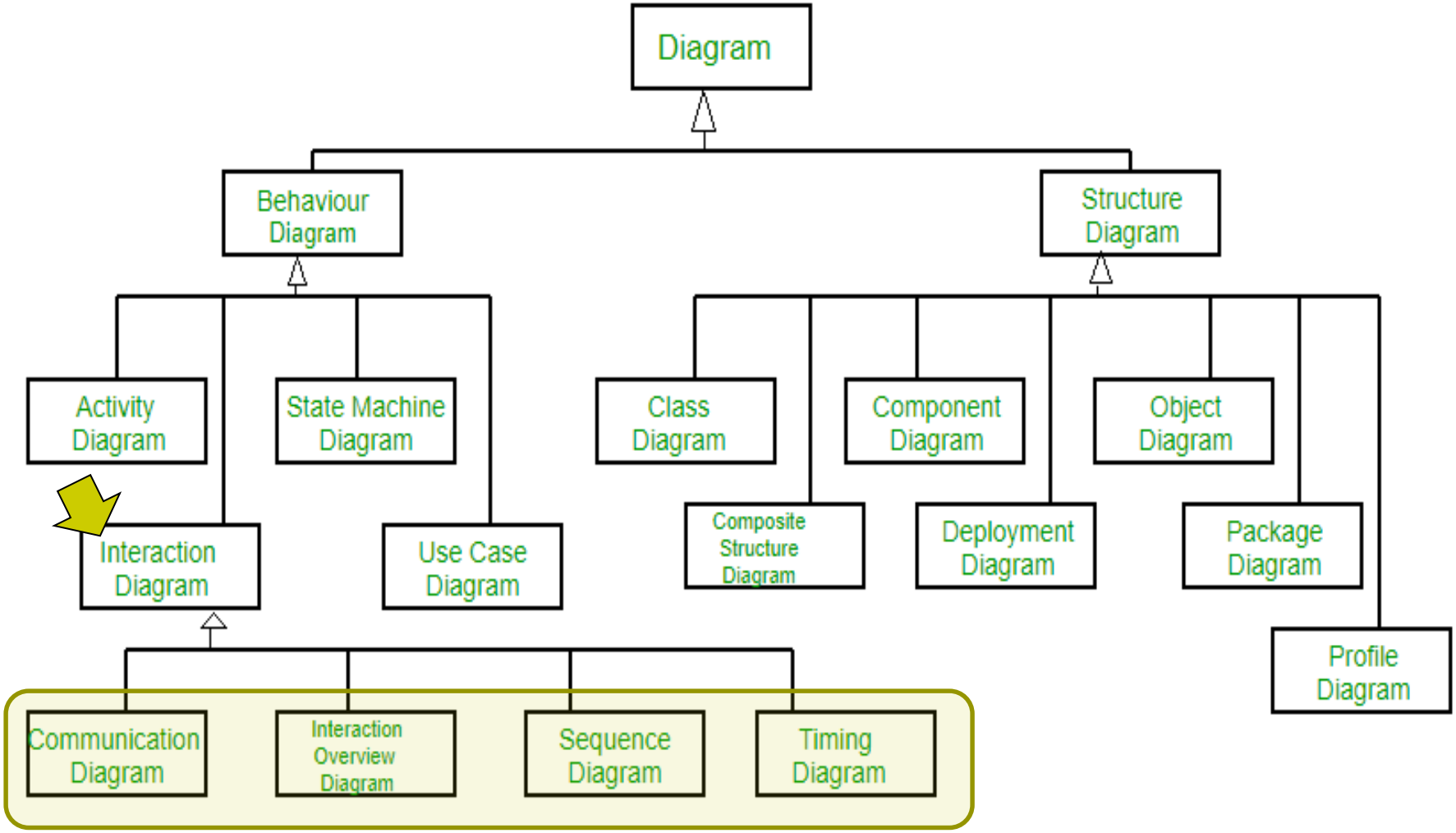


Note:
the alternate notation is more compact and manageable in a situation where there are *many states per participant*

Conclusions

- Interaction timing is most commonly associated with real-time or embedded systems, but it certainly is not limited to these domains.
- In a timing diagram, each event has timing information associated with it that accurately describes:
  1. **when the event is invoked,**
  2. **how long it takes for another participant to receive the event, and**
  3. **how long the receiving participant is expected to be in a particular state.**
- Although sequence diagrams and communication diagrams are very similar, timing diagrams add completely new information that is not easily expressed on any other form of UML interaction diagram.

UML 2.x Diagrams




Q&A