Requirement analysis. Use Case Diagrams. Relationships

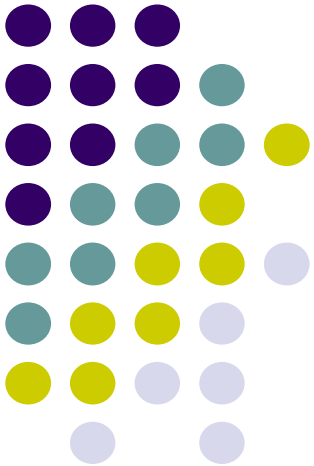
Definitions

Diagrams

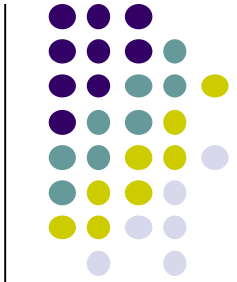
Relationships

Examples

Case Study



Bibliography



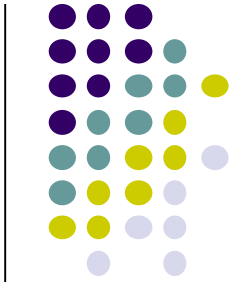
Basic

- Roger S. Pressman. Software Engineering : A Practitioner's Approach, 8th edition (2014), McGraw Hill, ISBN-10: 0078022126
- Ian Sommerville. Software Engineering, 10th edition (2015), Addison-Wesley Pub Co; ISBN-10: 0133943038

Additional

- Software Engineering : Theory and Practice by S. Pfleeger and J. Atlee, 4th edition (2009), Pearson International Edition, ISBN-10: 0136061699
- SDLC (Software Development Life Cycle) Phases, Methodologies, Process, And Models, https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/

Modeling principles



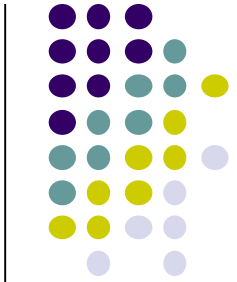
In software engineering work, two classes of models can be created:

- ***Requirements models* (also called *analysis models*)** represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain.
- ***Design models*** represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Requirements modeling principles

- Principle #1. The information domain of a problem must be represented and understood.
- Principle #2. The functions that the software performs must be defined.
- Principle #3. The behavior of the software (as a consequence of external events) must be represented.
- Principle #4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
- Principle #5. The analysis task should move from essential information toward implementation detail.

What is a requirement?

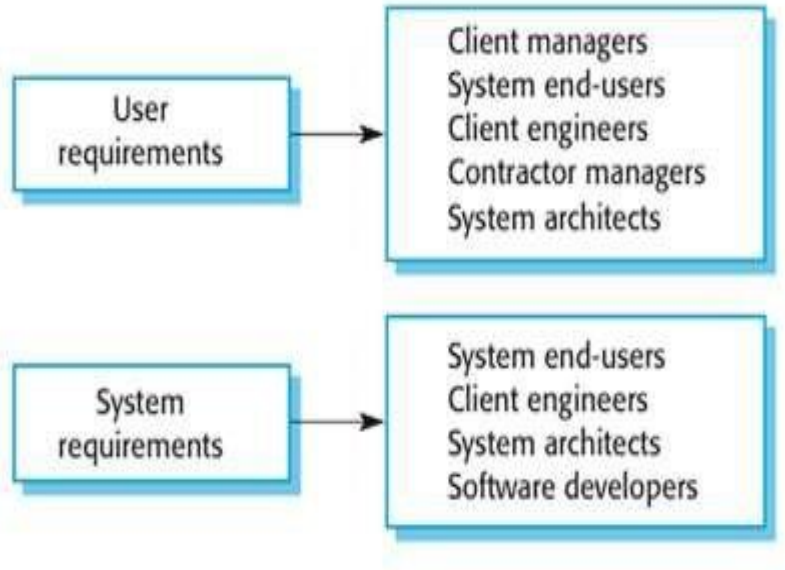


- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- Requirements may serve a dual function:
  - May be the basis for ***a bid for a contract*** – therefore must be open to interpretation;
  - May be ***the basis for the contract itself*** – therefore must be defined in detail;
  - Both these statements may be called ***requirements***.

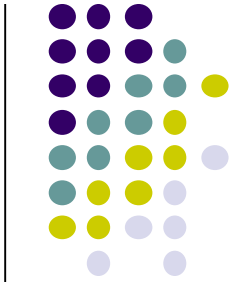
User- and system requirements

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. *Written for customers*.
- System requirements
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor. *Written for dev-op's*.

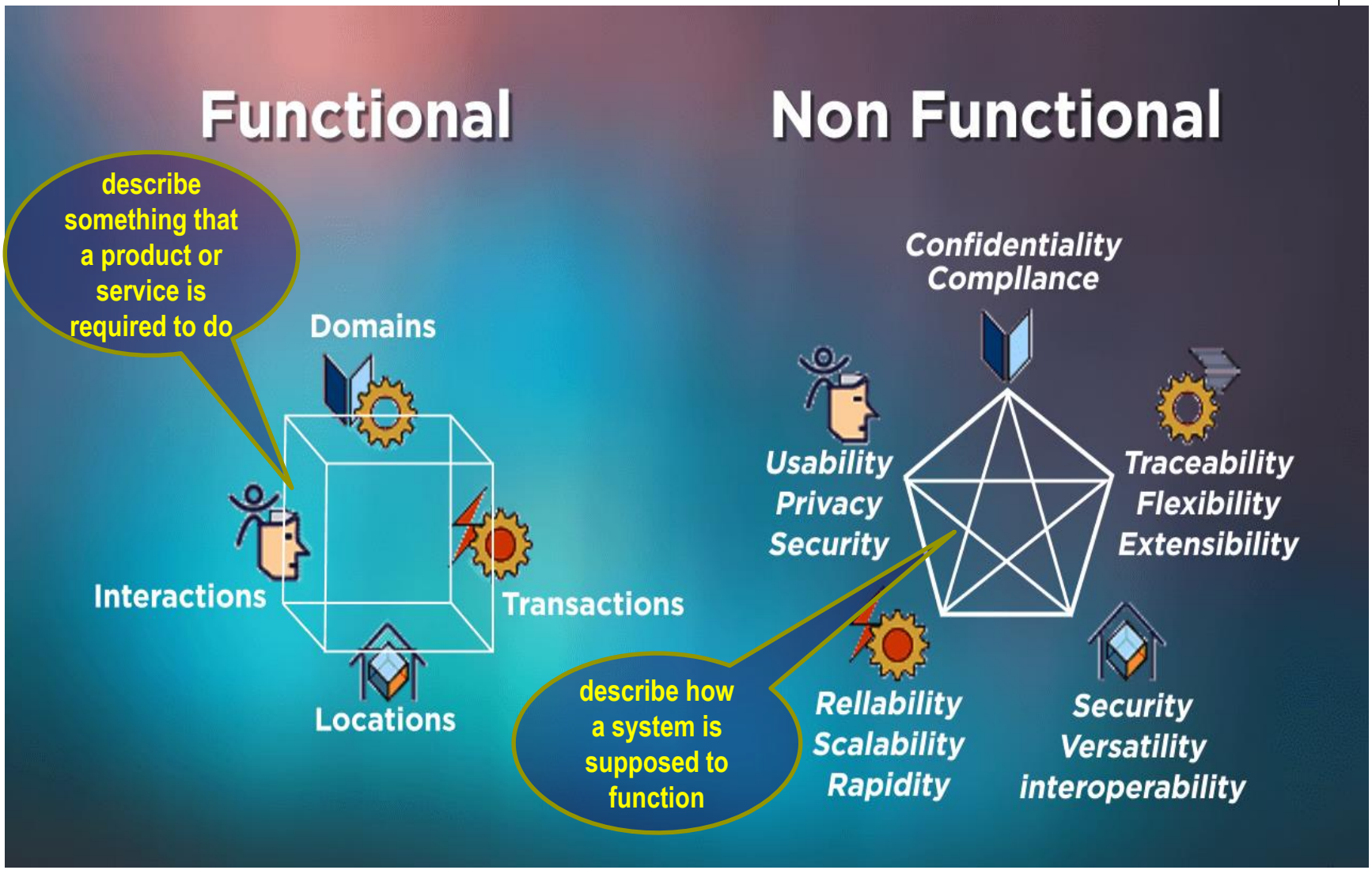
Who are they important for?



Functional and non-functional req's

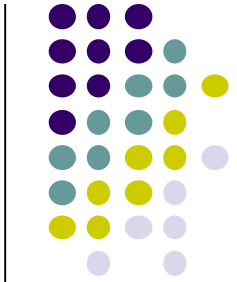


- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.
- Domain requirements (often seen as a subset of the functional req's)
  - Constraints on the system from the domain of operation



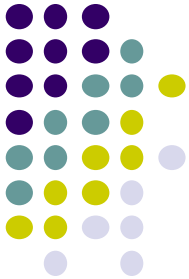
Source: https://vironit.com/functional-vs-non-functional-requirements/

Functional requirements



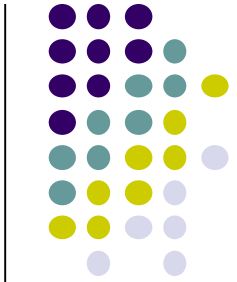
- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.

Requirements completeness and consistency



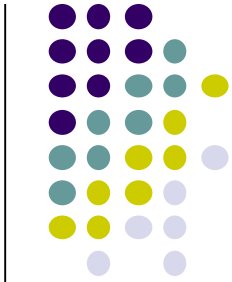
- In principle, requirements should be both complete and consistent.
- ***Complete***
  - They should include descriptions of all facilities required.
- ***Consistent***
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, it is very difficult to produce a complete and consistent requirements document.

Non-functional requirements



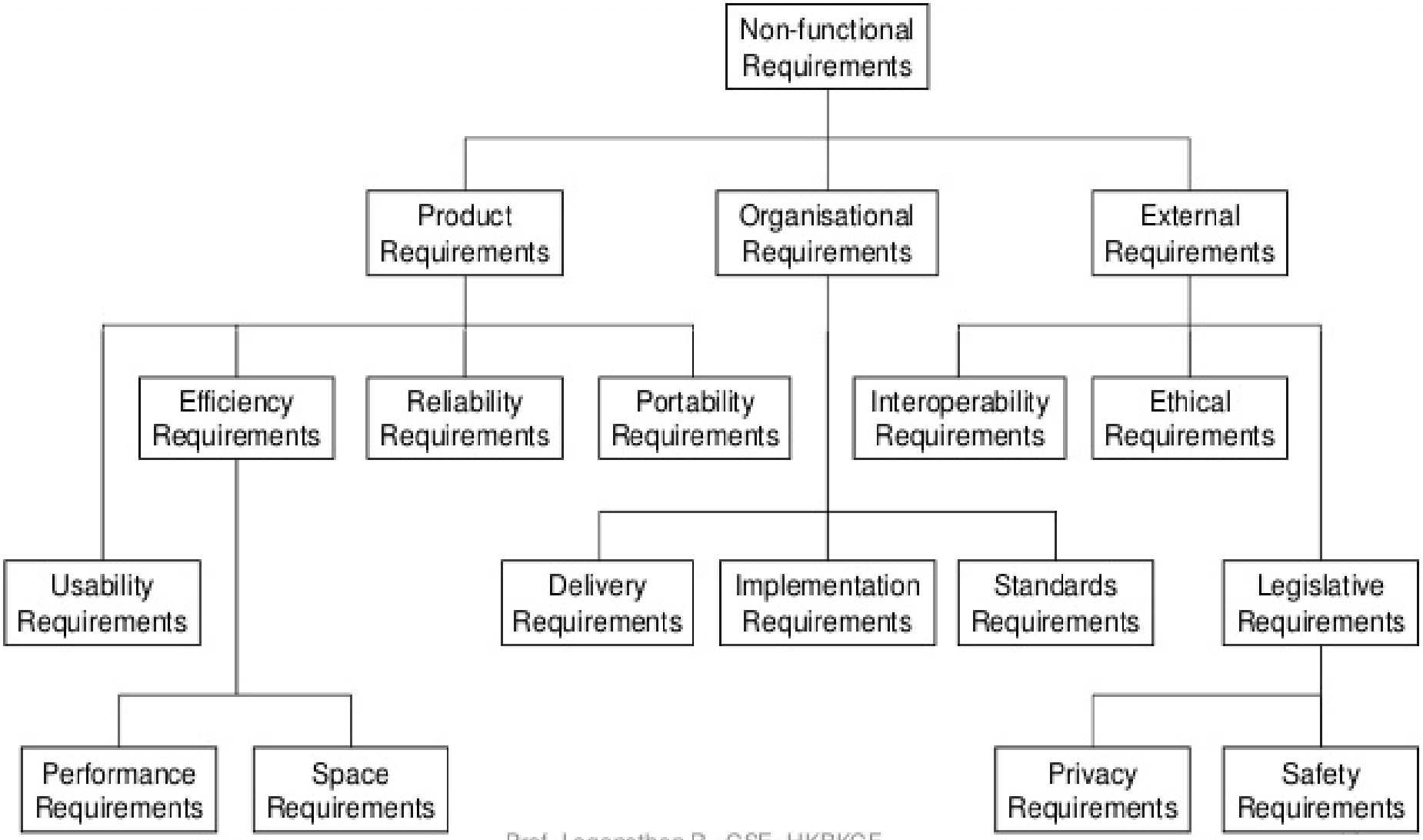
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Non-functional classifications

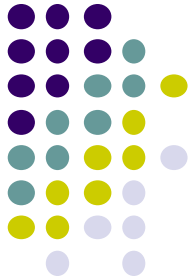


- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Types of non-functional req's

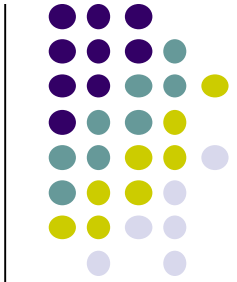


Metrics for specifying non-functional req's



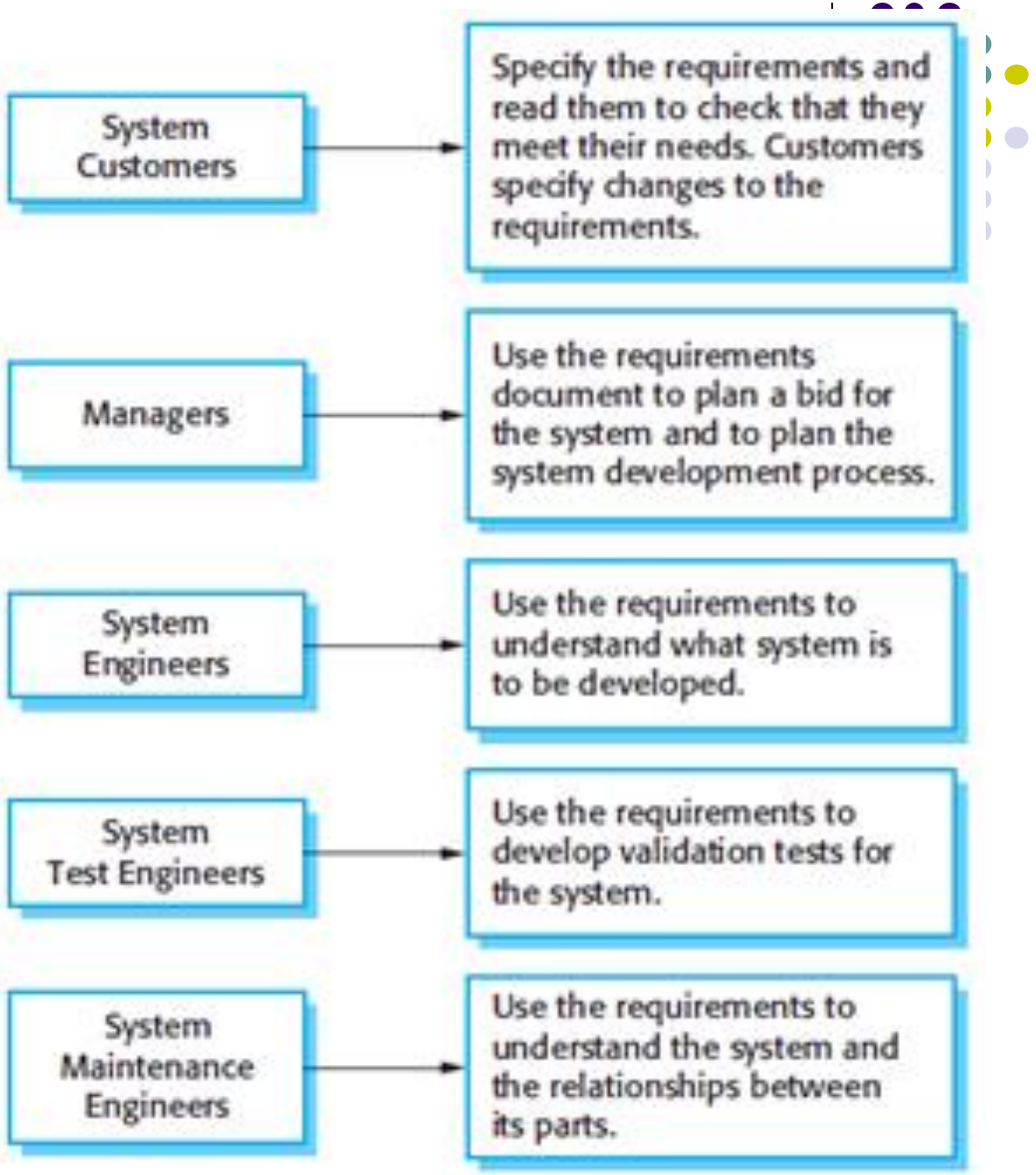
Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

The software requirements document

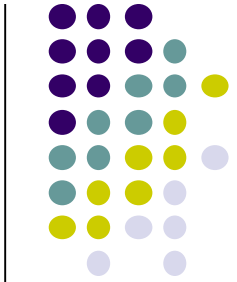


- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Users of a requirements document

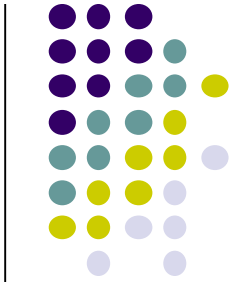


Requirements' document structure 1/2



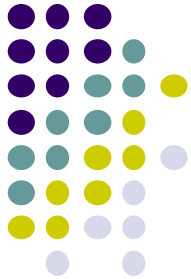
Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

Requirements' document structure 2/2



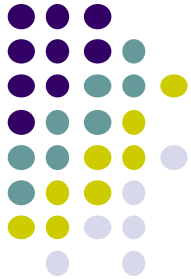
Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Requirements specification



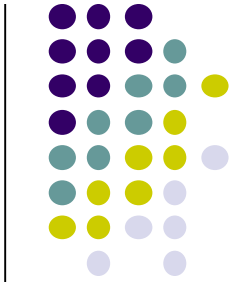
- The process of writing of the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete as possible.

Ways of writing a system requirements specification



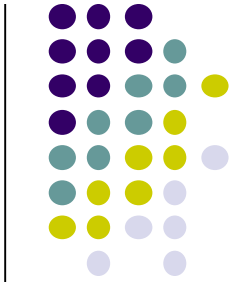
Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on *a standard form or template*. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now *rarely used* although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; *UML use case and sequence diagrams are commonly used*.
Mathematical specifications	These notations are based on mathematical concepts such as *finite-state machines or sets*. Although these unambiguous specifications *can reduce the ambiguity* in a requirements document, *most customers don't understand a formal specification*. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Scenarios



- Scenarios are real-life examples of how a system can be used.
- They should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.

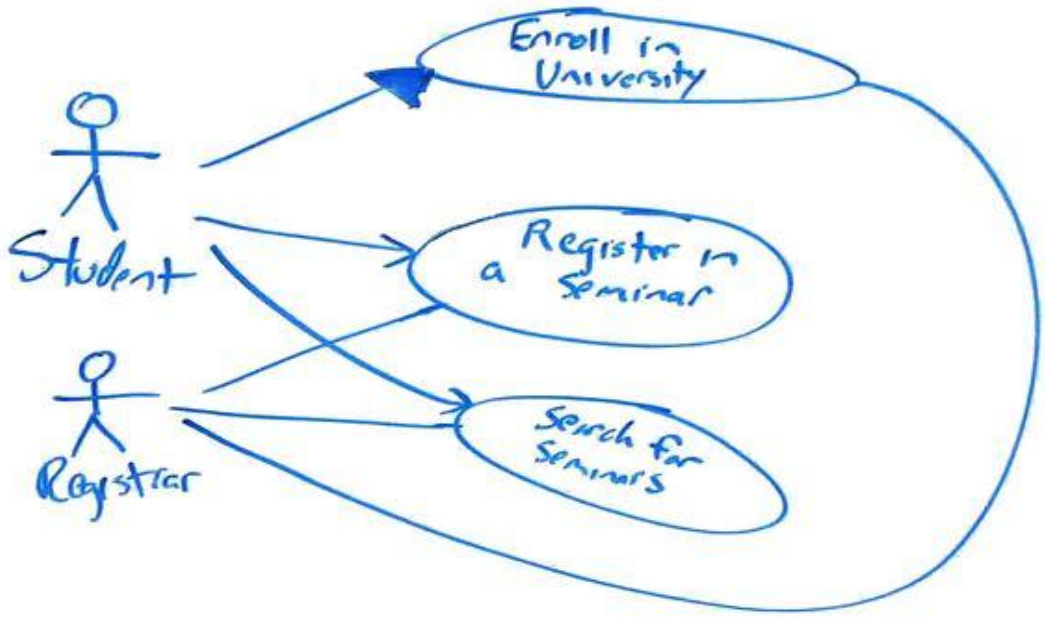
Use cases



- Use-cases are a scenario based technique in **UML (Unified Modelling Language),** which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- High-level graphical model supplemented by Sequence diagrams (a more detailed tabular description) may be used to add detail to use-cases by showing the sequence of event processing in the system.

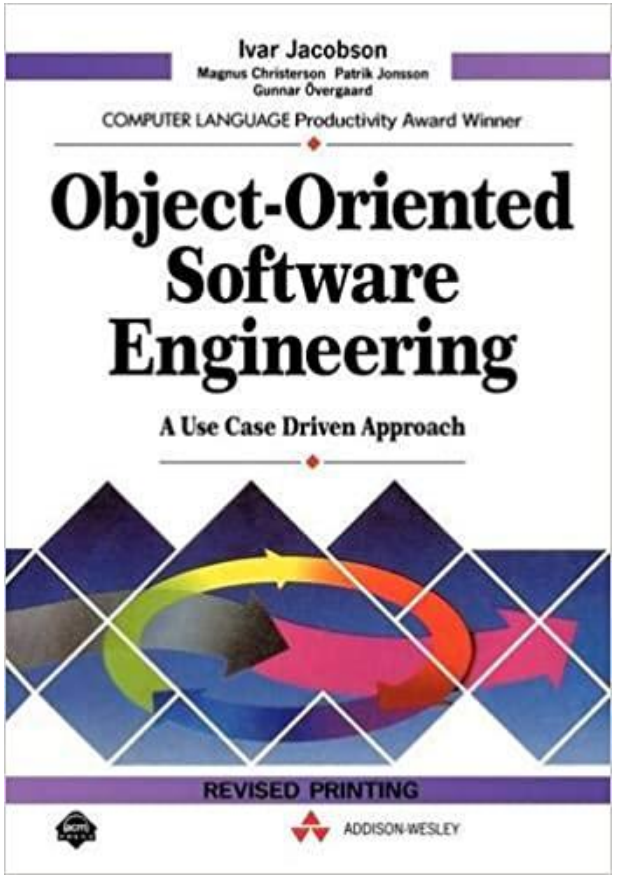
UML use case diagrams

- UML use case diagrams provide overview of usage requirements for a system.
- For actual system or software engineering use case diagrams describe actual system/software requirements
- Useful also for simple presentations to management and/or project stakeholders

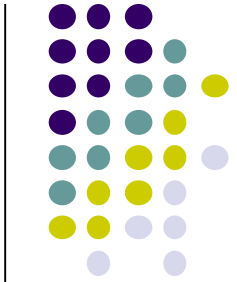


History

- 1986 - Ivar Jacobson formulated the textual, structural and visual modeling techniques for specifying use (originally *usage*) cases
- 1990 - use cases started becoming one of the most common practices for capturing functional requirements
- 1992 - Jacobson's published the book "Object-Oriented Software Engineering - *A Use Case Driven Approach*"
- 1995 - use case diagrams included into Unified Modeling Language (UML) and the Rational Unified Process (RUP)

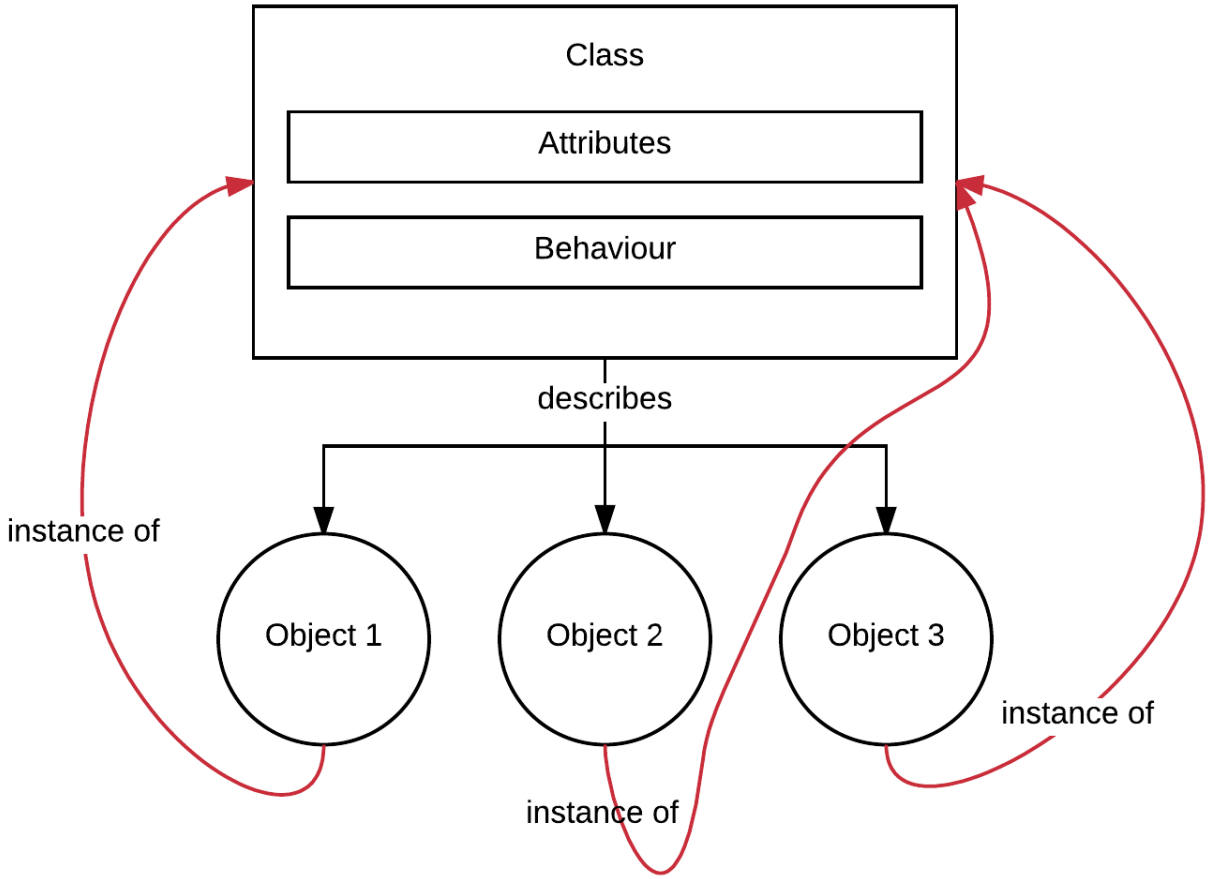


Elements



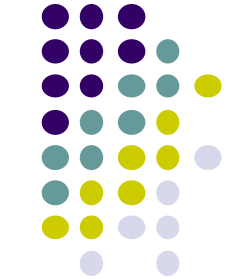
- **Actors** - a person, organization, or external system that plays a role in one or more interactions with your system
- **Use cases** - describe a sequence of actions that provide something of measurable value to an actor
- **Associations** - exist whenever an actor is involved with an interaction described by a use case
- **Other relations** – *include*, *extend*, *generalize* and *depend*
- **System boundary boxes** (optional) - rectangles around the use cases to indicates the scope of your system
- **Packages** (optional) - UML constructs that enable you to organize model elements (such as use cases) into groups.

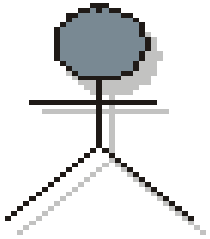
Retrospection



An **instance** is a specific **object** created from a particular **class**.

Defining Actors





Actor

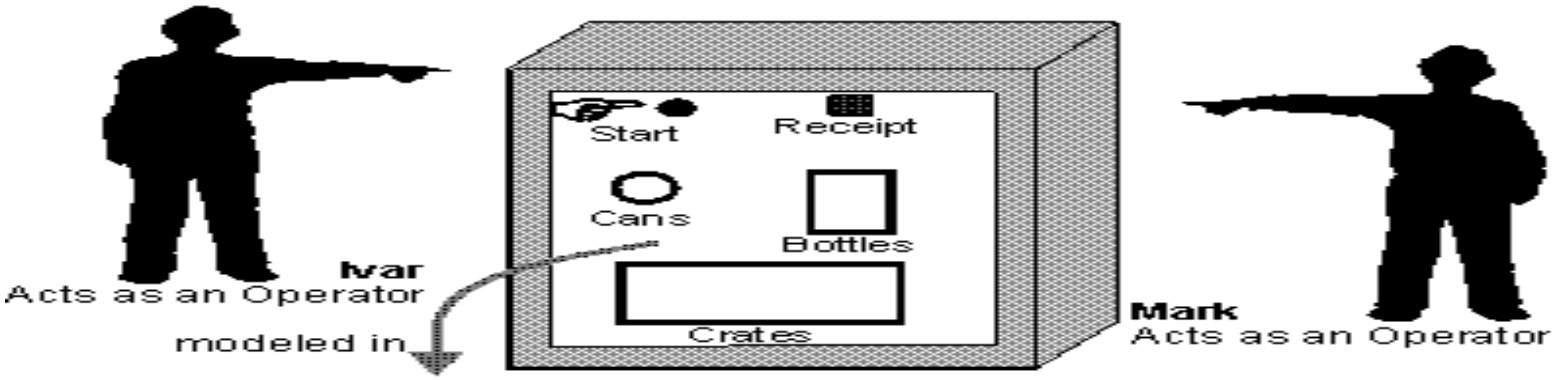
An **actor** ***instance*** is *someone* or *something* outside the system that interacts with the system.

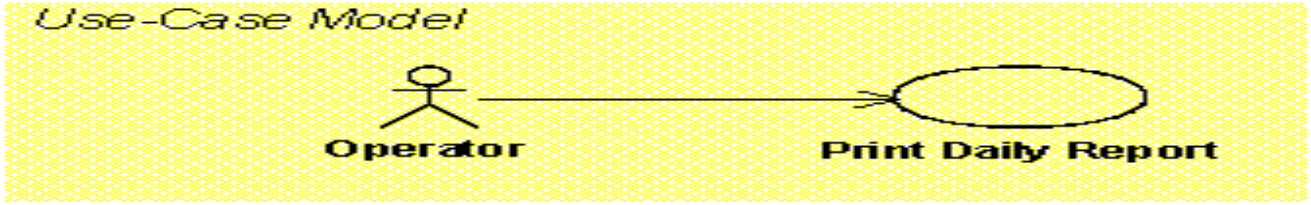
An **actor** ***class*** defines a set of actor instances, in which each actor instance plays the same role in relation to the system.

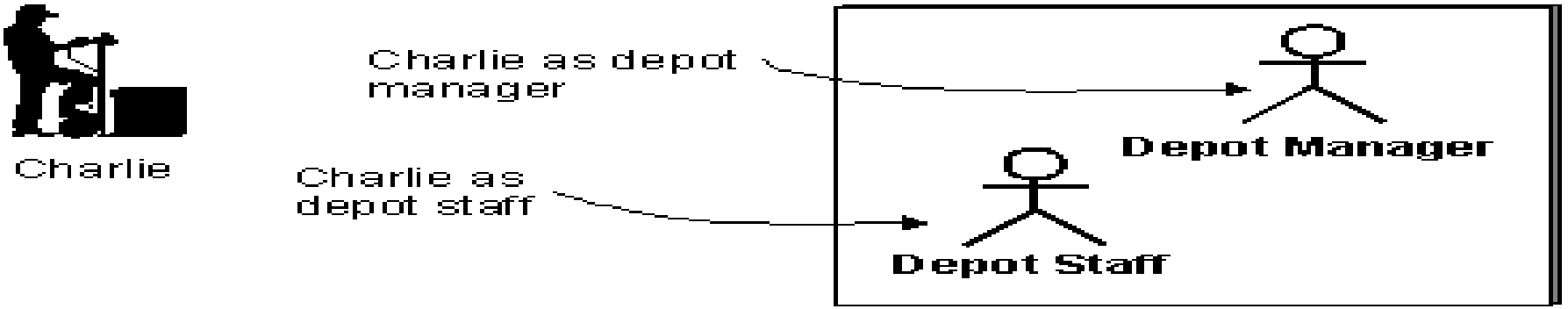
To fully understand the system's purpose you must know **who** the system is for, or who will use the system. Different user types are represented as actors.

An actor is **anything** that exchanges data with the system. An actor can be a user, external hardware, or another system

Differences between an Actor and an Individual System User

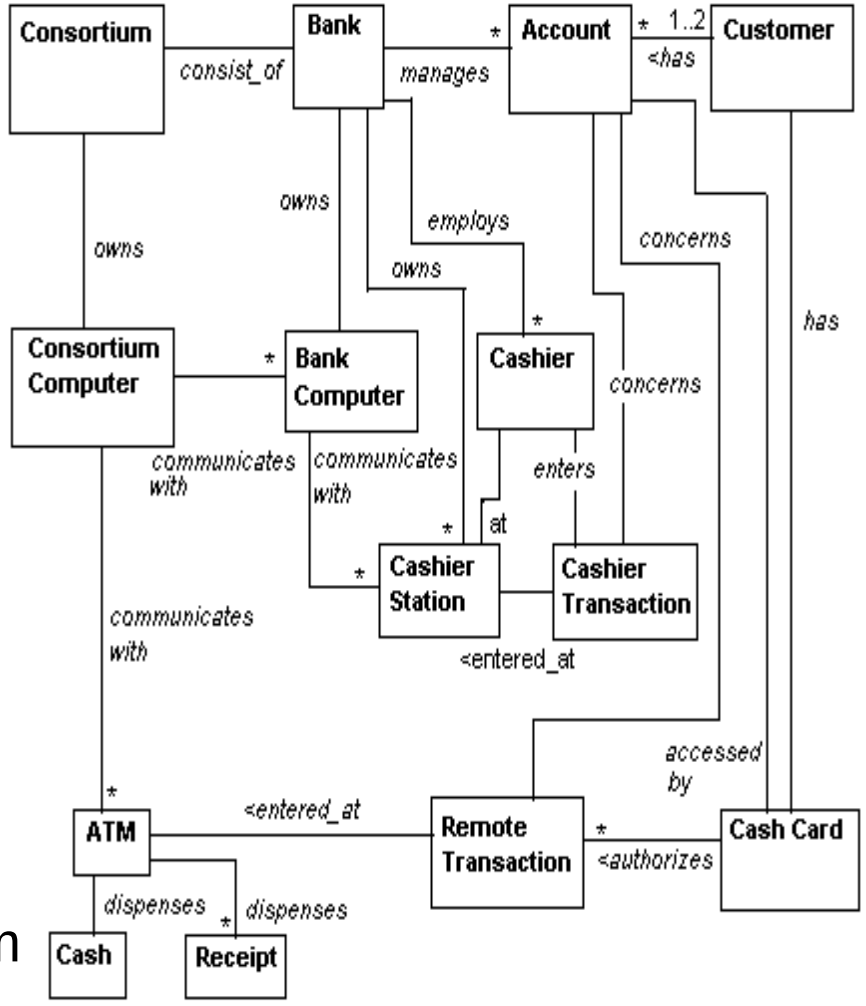






Actors as Different Aspects of System's Surroundings

- Users who execute the system's main functions – for a Depot-Handling System: Depot Staff and Order Registry Clerk.
- Users who execute the system's secondary functions, such as system administration - Depot Manager.
- External hardware the system uses - a ventilation system that controls the temperature in a building continuously gets metered data from sensors in the building. Sensor is therefore an actor.
- Other systems interacting with the system - An ATM (Automated Teller Machine) must communicate with the central system that holds the bank accounts. The central system is probably an external one, and should therefore be an actor.



Source: https://www.cs.vassar.edu/~cs335/ATM/ATMExample.html

How to Find Actors

- Who will supply/use/remove information?
- Who will use this functionality?
- Who is interested in any requirement?
- Where in the organization is the system used?
- Who will support/maintain the system?
- What are the system's external resources?
- What other systems will need to interact with this one?

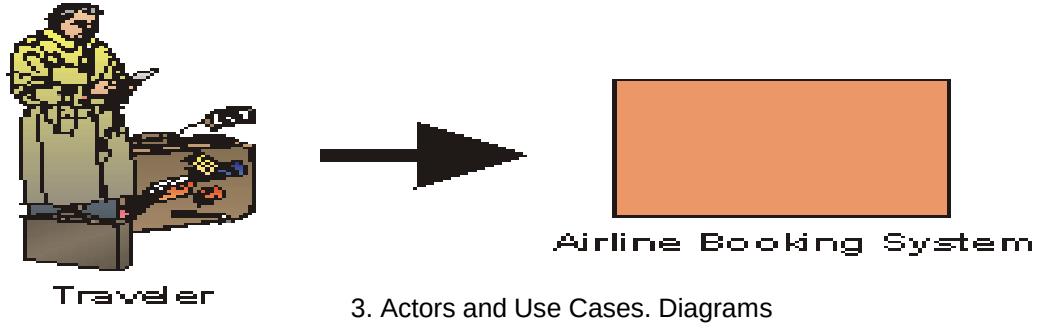
Actors Help Define System Boundaries

Only those who ***directly communicate with the system*** need to be considered as ***actors***. Otherwise, you are attempting to model the business in which the system will be used, not the system itself.

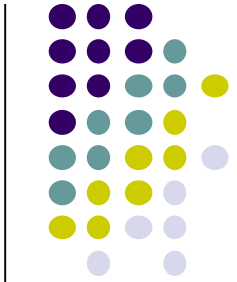
In an airline booking system (e.g., Amadeus), what would the actor be?

1. If the system is to be used by a travel agent, the actor would be travel agent.

2. When users to connect via the Internet:



Documenting Actor Characteristics



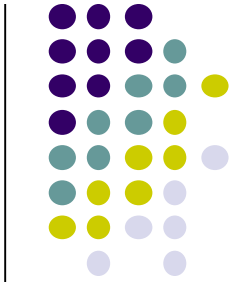
Brief description:

- What or who the actor represents?
- Why the actor is needed?
- What interests the actor has in the system?

Actor characteristics might influence how the system is developed:

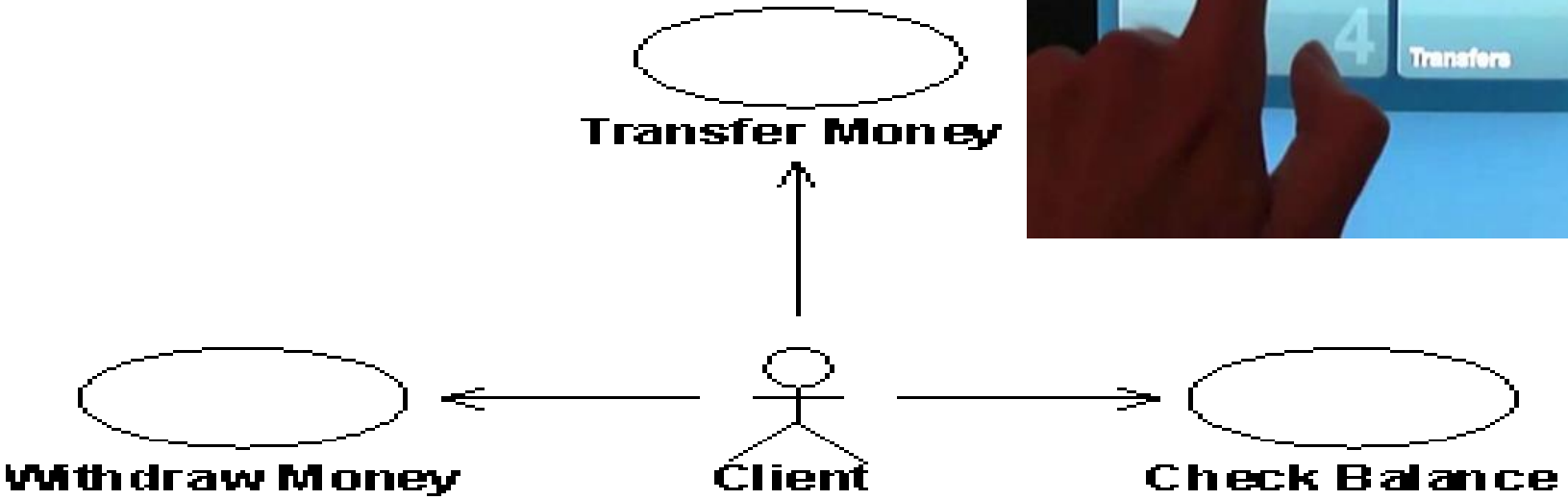
- The actor's scope of responsibility.
- The physical environment in which the actor will be using the system.
- The number of users represented by this actor.
- Others.

Defining Use Cases



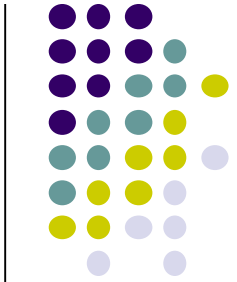
 **Use Case**	A **use case *instance* (scenario)** is a sequence of actions a system performs that yields an observable result of value for one or more particular actors or other stakeholders of the system. A **use case (*class*)** defines a set of use-case instances.

Use Case Example



An ATM example - the system functionality is defined by different use cases, each of which represents a specific flow of events, defines what happens in the system when the use case is performed, and has a task of its own to perform.

How to Find Use Cases

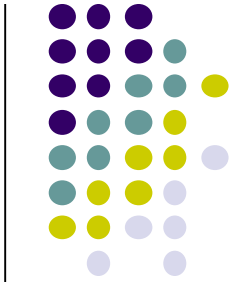


- What are the system tasks for each actor you have identified?
- Does the actor need to be informed about certain occurrences in the system?
- Will the actor need to inform the system about sudden, external changes?
- Does the system supply the business with the correct behavior?
- Can all features be performed by the use cases you have identified?
- What use cases will support and maintain the system?
- What information must be modified or created in the system?

Use cases types:

- System start and stop.
- Maintenance of the system (add user, …).
- Maintenance of data stored in the system.
- Functionality needed to modify behavior in the system.

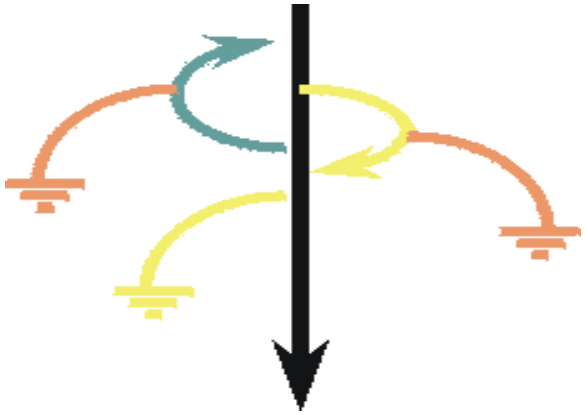
Use Case Documenting – Flow of Events

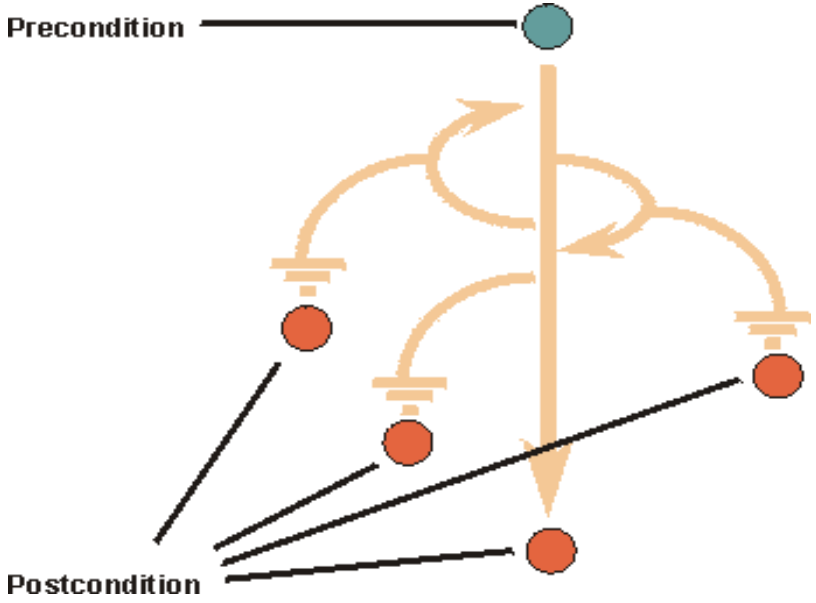


The **Flow of Events** of a use case contains the most important information derived from use-case modeling work. Its contents:

- Describe how the use case starts and ends
- Describe what *data is exchanged* between the actor and the use case
- Do not describe *the details of the user interface*, unless it is necessary to understand the behavior of the system
- Describe *the flow of events*, not only the functionality. To enforce this, start every action with "When the actor ... "
- Describe *only the events that belong to the use case*, and not what happens in other use cases or outside of the system
- Avoid *vague terminology* such as "for example", "etc. " and "information" - description stile
- Detail the flow of events - all "*whats*" should be answered.

Flow of Events - Structure

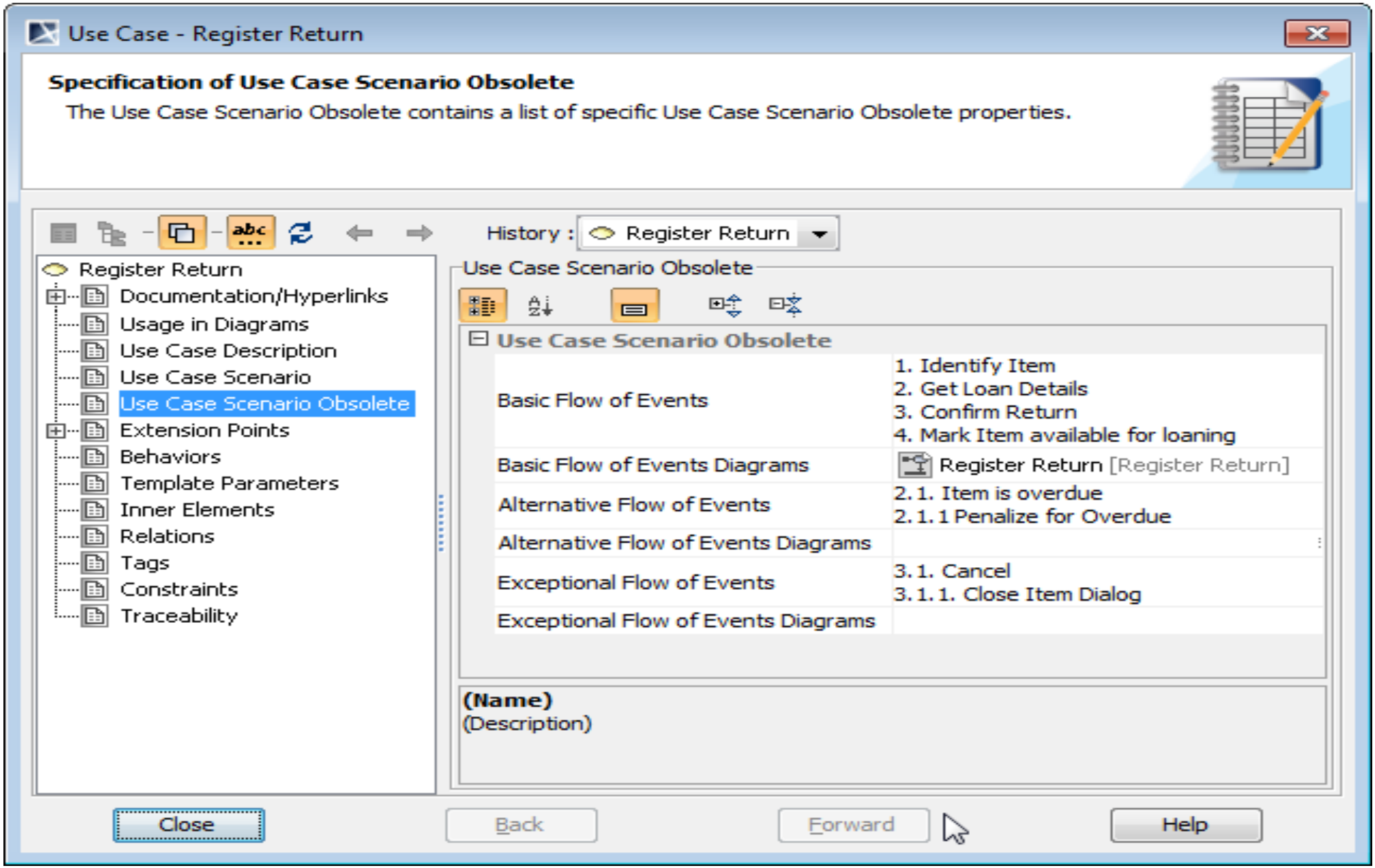




Basic (the straight arrow) and ***alternative*** flows of events (the curves).

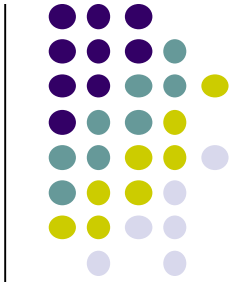
A ***pre-condition*** is the state of the system and its surroundings that is required before the use case can be started.

A ***post-condition*** - the states the system can be in after the use case has ended.



Source: https://docs.nomagic.com/display/MD184/The+Use+Case+scenario+created+with+the+version+17.0.1+or+earlier

How to describe a use case scenario?

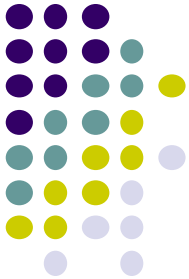


*While a **use case** is an abstraction that describes all possible scenarios, a **scenario** is an use case instance with concrete set of actions.* A three fields template [Bruegge&Dutoit, 2004]:

- Name – unambiguous, underlined;
- Participating actors – underlined names;
- Flow of events – sequence of numbered interactions for the use case; accomplished either by the actor (left column) or by the system (right)

  There are no Entry and Exit conditions – as they are abstractions to describe a range of conditions under which a use case is invoked.

How to describe a use case?



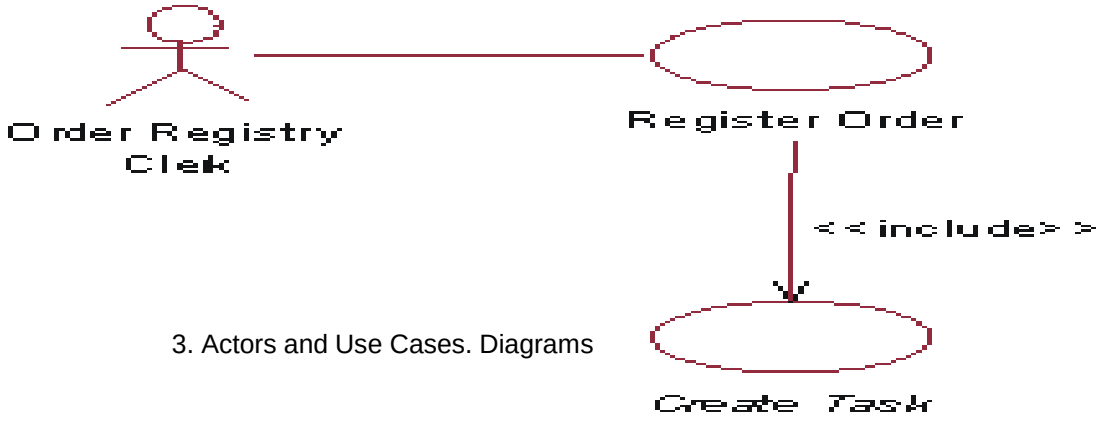
A six fields template [Bruegge&Dutoit, 2004]:

- Name – unambiguous, unique across the system;
- Participating actors;
- Entry conditions – need to be TRUE before use case initiation;
- Flow of events:
  - sequence of numbered interactions for the use case;
  - accomplished either by the actor (left column) or by the system (right)
- Exit conditions - need to be TRUE after use case completion;
- Quality conditions – non-functional requirements…

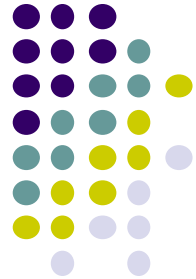
Concrete and Abstract Use Cases

A **concrete** use case is initiated by an actor and constitutes a complete flow of events (instance of the use case performs the entire operation called for by the actor).

An ***abstract*** use case (written in *italics*) is never instantiated in itself. Abstract use cases are *included in, extended into, or generalizing* other use cases. When a concrete use case is initiated, an instance of the use case is created. This instance also exhibits the behavior specified by its associated *abstract* use cases. Thus, no separate instances are created from *abstract* use cases.

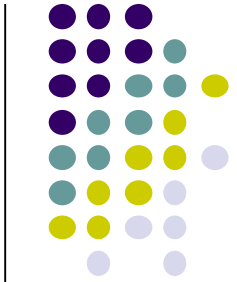


ID, Rank, Leaf and Root use cases



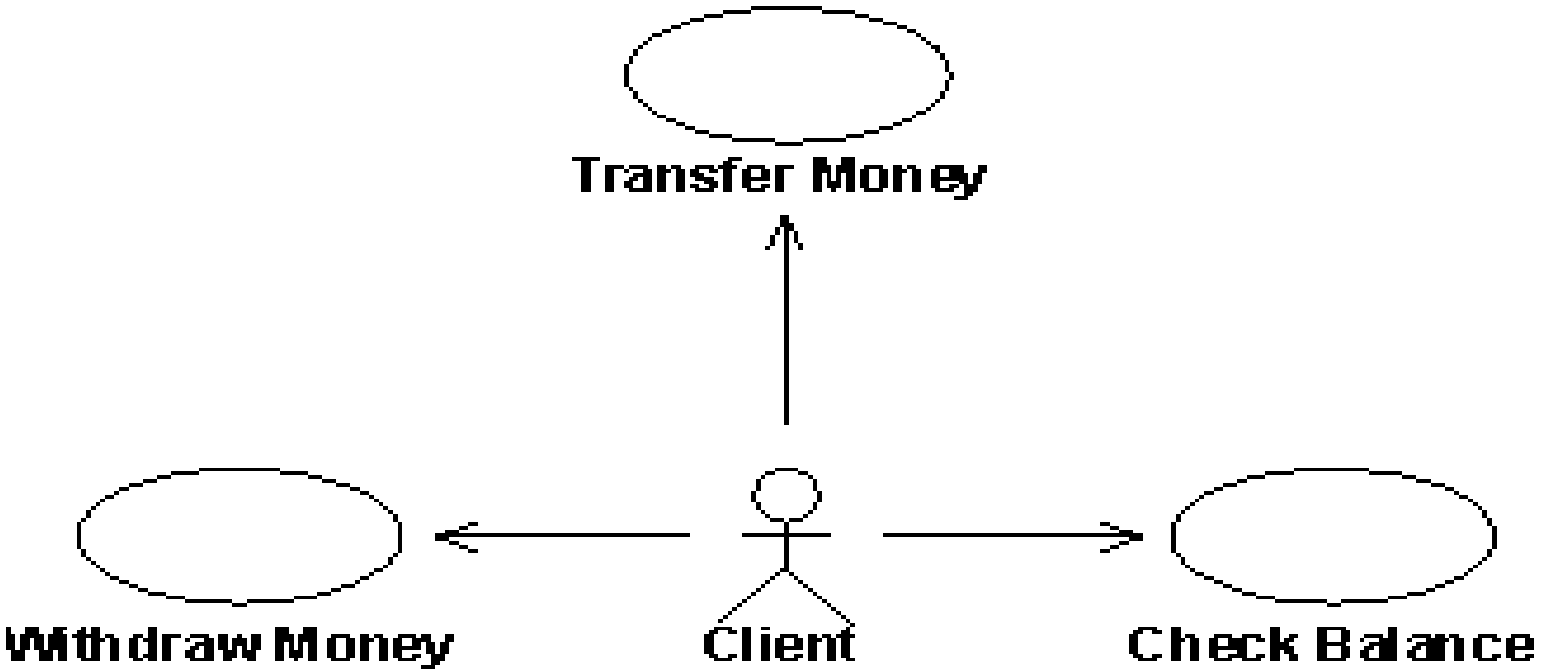
ID	A unique value for identifying the use case.
Rank	Describe the importance of the use case. The higher the ranking implies that more attention is needed.
Leaf	Indicates whether it is possible to further specialize an use case. If the value is true, then it is not possible to further specialize the use case.
Root	Indicates whether the use case has no ancestors (true for no ancestors).

Associations (relationships)



- Associations between actors and/or use cases are indicated in use case diagrams by solid lines.
- An association exists whenever an actor is involved with an interaction described by a use case.
- Associations are modeled as lines connecting use cases and actors to one another
- The arrowhead is often used *to indicate the direction of the initial invocation of the relationship* (but *not the direction of information exchange*)

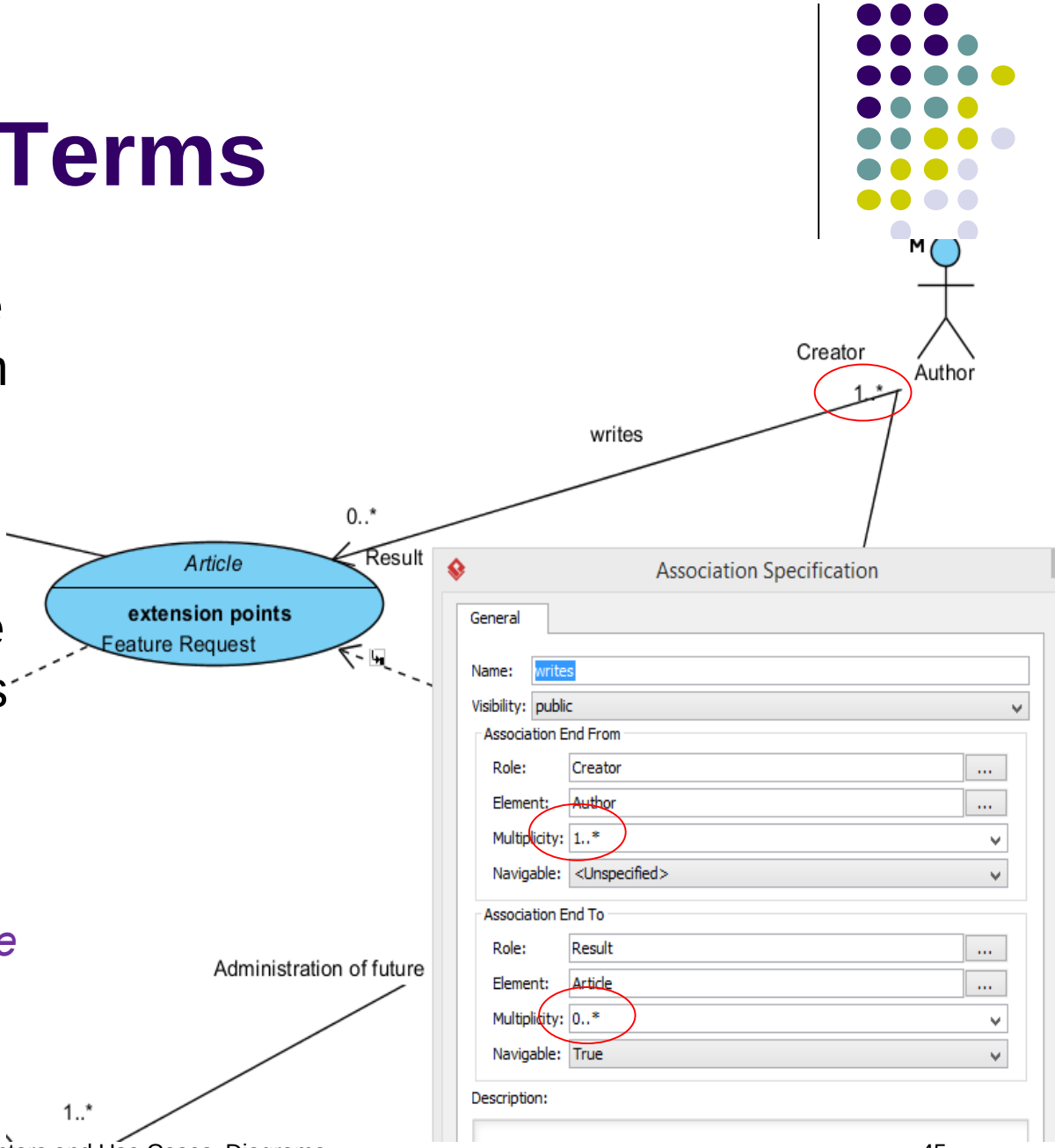
Associations



UML 2.0 New Terms

- Use Case ***Multiplicities*** lie on the association between actors and use cases.
- The definition of ***multiplicities*** in the use case diagram is exactly the same as they are in a class diagram – it shows the number of instances associated each other.

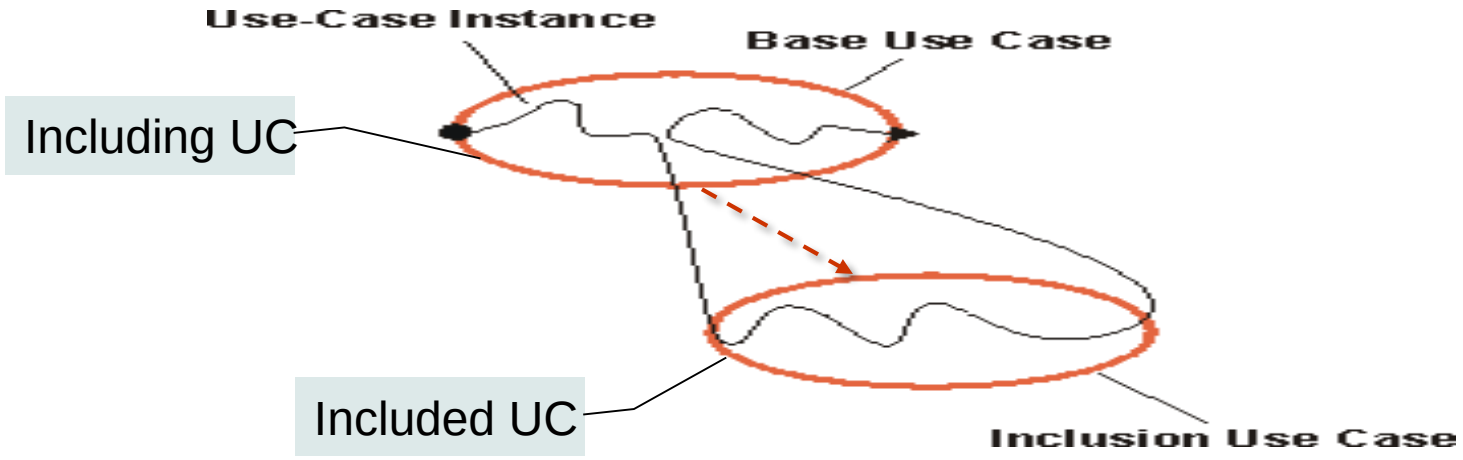
* *What's New in UML 2? The Use Case Diagram–by Randy Miller, June 30, 2003*



Include-Relationship

An **include-relationship** is a directed relationship from a base use case to an inclusion use case, specifying how the behavior defined for the inclusion use case is *non-optionally*, *explicitly* inserted into the behavior defined for the base use case.

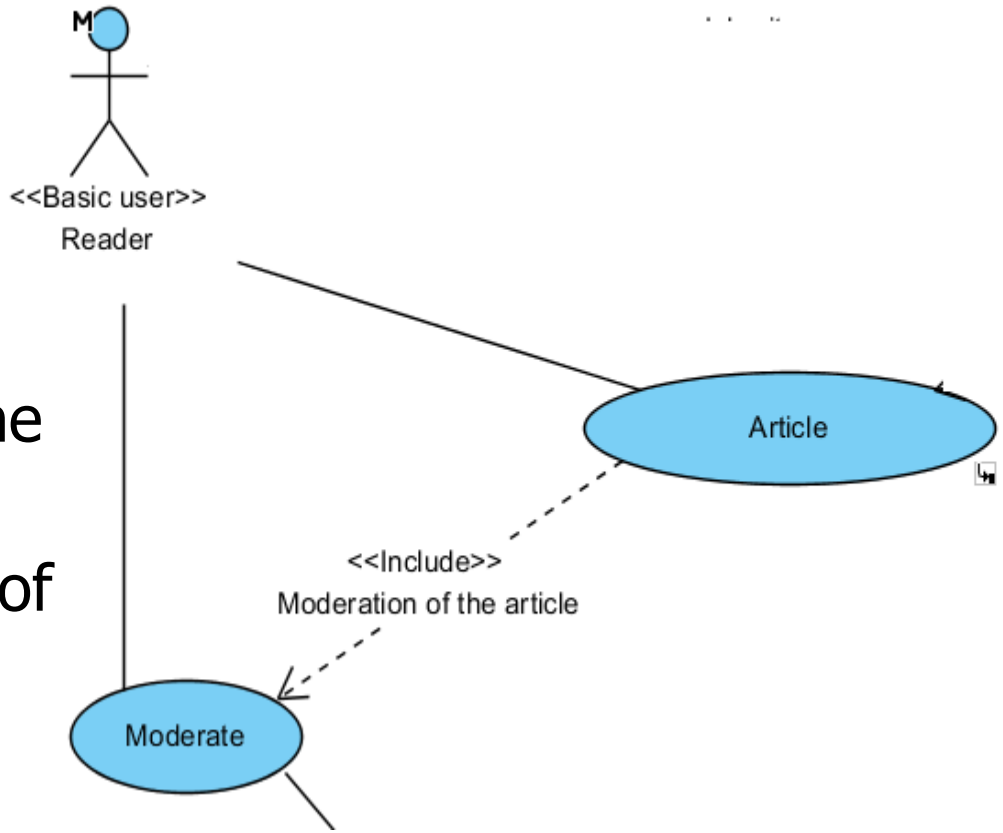
«include»



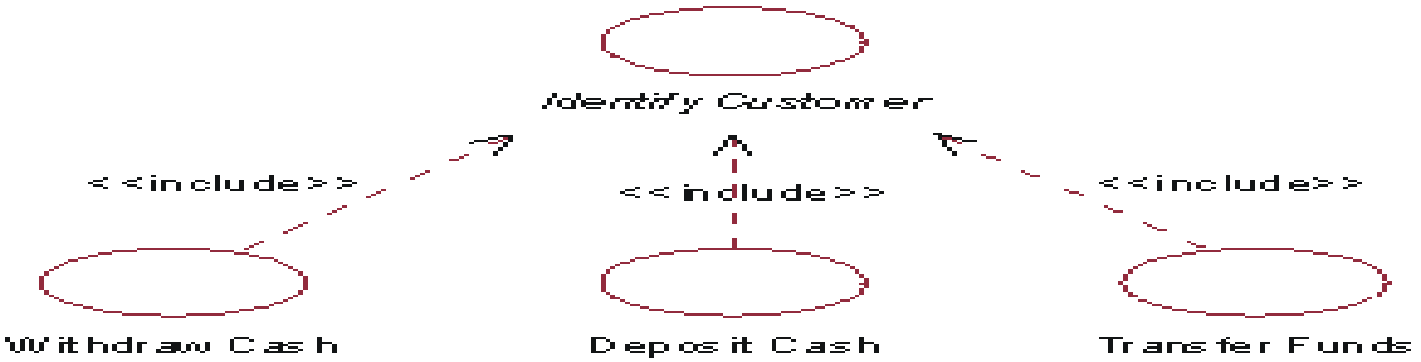
Executing a use-case instance following the description of a base use case including its inclusion.

More about <<Include>>

- Including use case includes the "addition" and owns the include relationship.
- Addition is use case that is to be included.
- The including use case may only depend on the result (value) of the included use case.
- This value is obtained as a result of the execution of the included use case.



Example of Includes



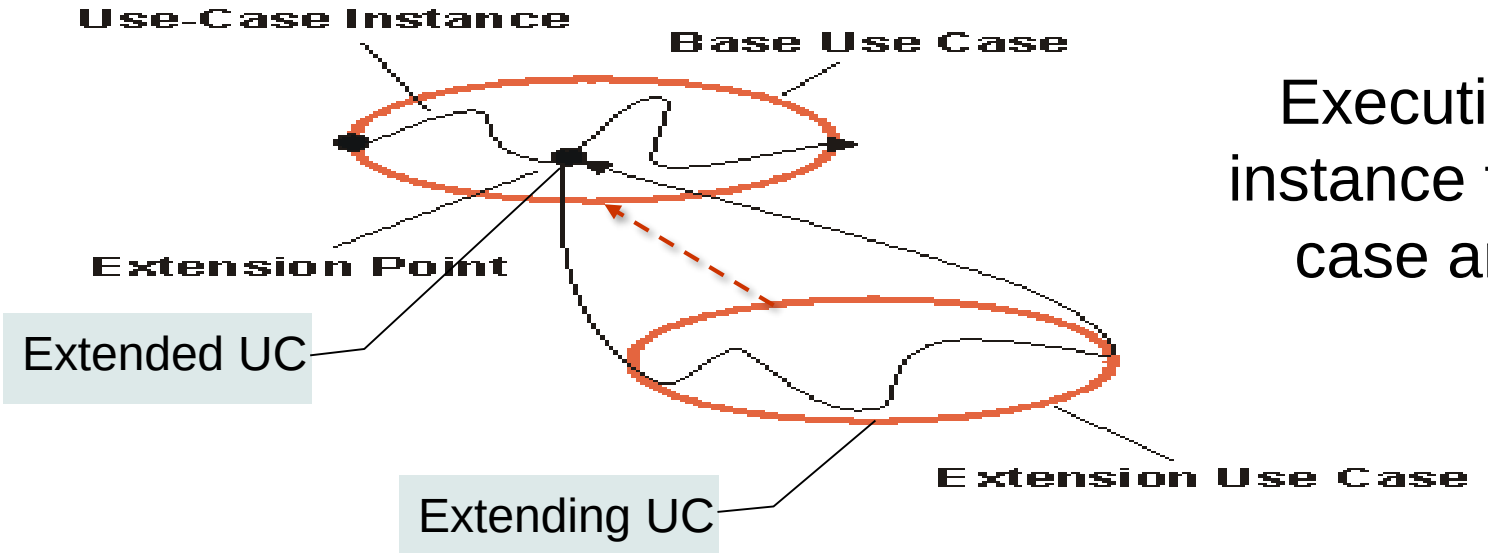
In the ATM system, the use cases Withdraw Cash, Deposit Cash, and Transfer Funds all include the use case Identify Customer.

- The base use case has control of the relationship to the inclusion and *can depend on the result* of performing the inclusion,
- but *neither the base nor the inclusion use case may access each other's attributes*.
- The inclusion is in this sense *encapsulated*, and represents behavior that can be *reused in different base use cases*.

Extend-Relationship

An **extend-relationship** goes from an extension use case to a base use case, specifying how the behavior defined for the extension use case *can be inserted* into the behavior of the base use case. It is implicitly inserted in the sense that the extension is not shown in the base use case.

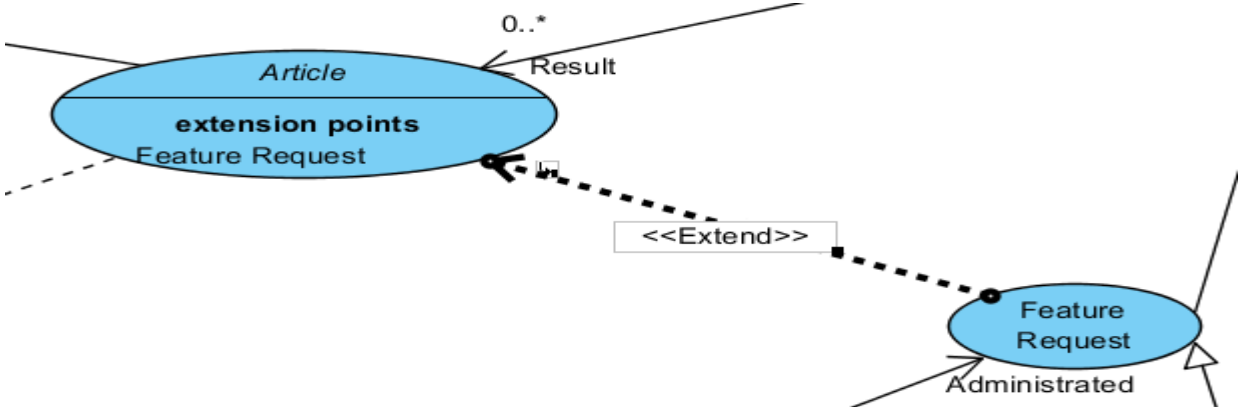
«extend»



Execution of a use-case instance follows a base use case and its extension.

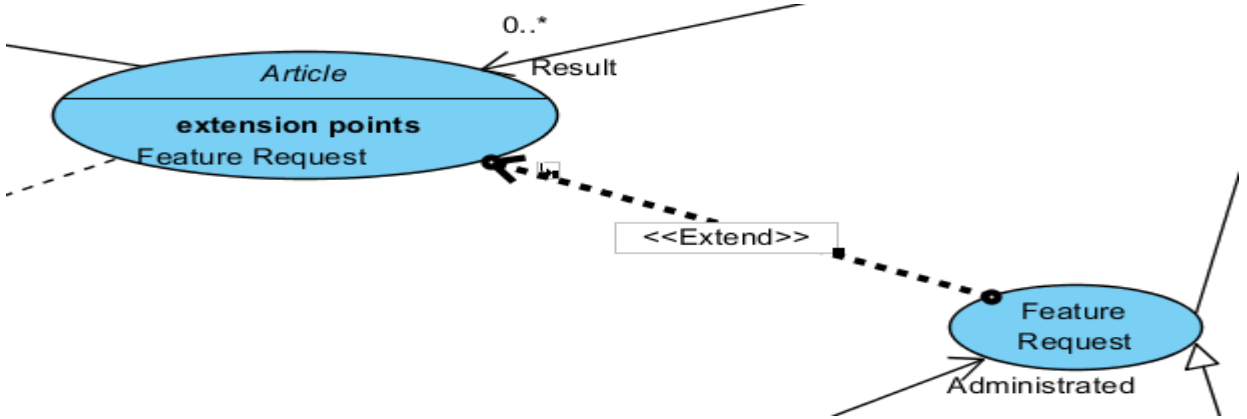
More about “Extend”

- This relationship specifies that the behavior of a use case may be extended by the behavior of another (supplementary) use case.
- The extended use case is defined independently of the extending use case and is meaningful independently of the extending use case.
- On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case **under specific conditions**.

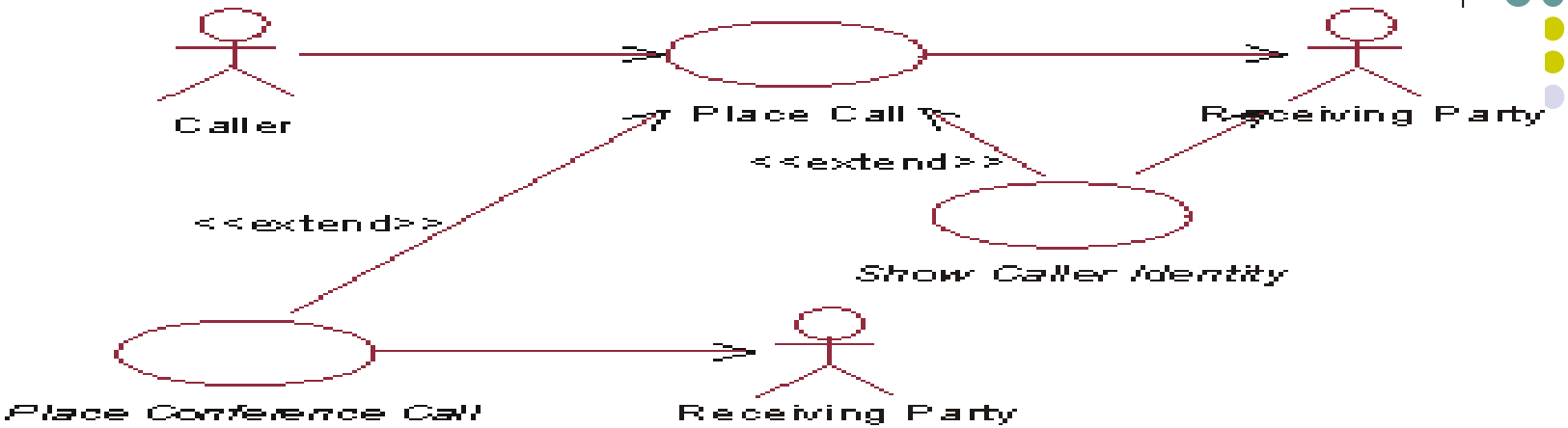


UML 2.0 New Terms

- ***Extension Points*** (UML 2.0) – they show the actual logic necessary for one use case to extend another.
- An extension point identifies ***the point in the base use case*** where the behavior of an extension use case ***can be*** inserted.
- The extension point is specified for a base use case and is referenced by an extend relationship between the extension use case ***extending*** the base use case (i.ee., the ***extended*** use case).



Example of Extensions



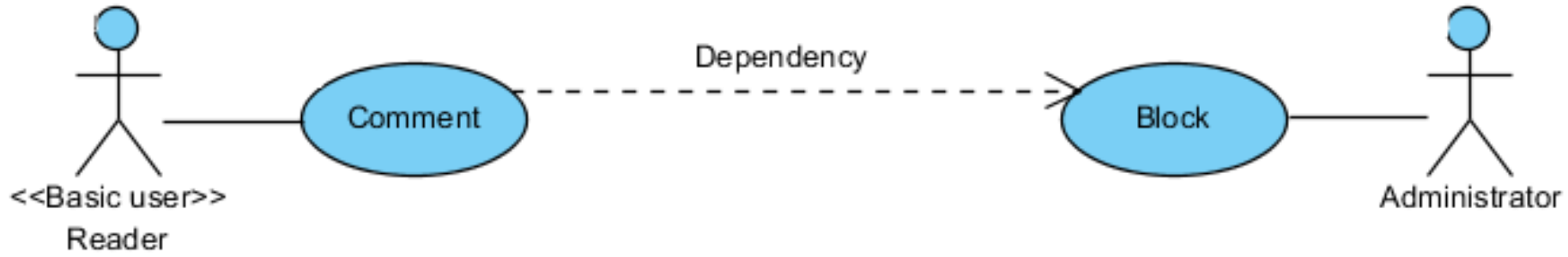
In a phone system, the primary user service is represented by the use case Place Call. Examples of optional services (extensions) are:

- To be able to add a third party to a call (Place Conference Call).
- To allow the receiving party to see the identity of the caller (Show Caller Identity).

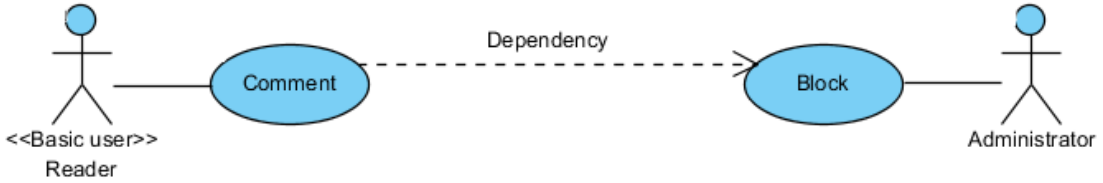
The extension is *conditional* - it is dependent on what has happened while *executing the base use case*. The base use case does not control the conditions for executing the extension – *the conditions are described within the extend-relationship*.

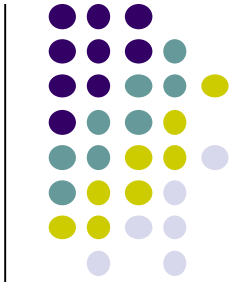
Dependency

- A dependency is a relationship that signifies that a single element or a set of model elements requires other model elements *for their specification or implementation*
- Changes in “Block” will affect “Comment”
- The complete semantics of the depending element (the **client**) is either **semantically or structurally dependent** on the definition of the **supplier** element(s)



More about dependency

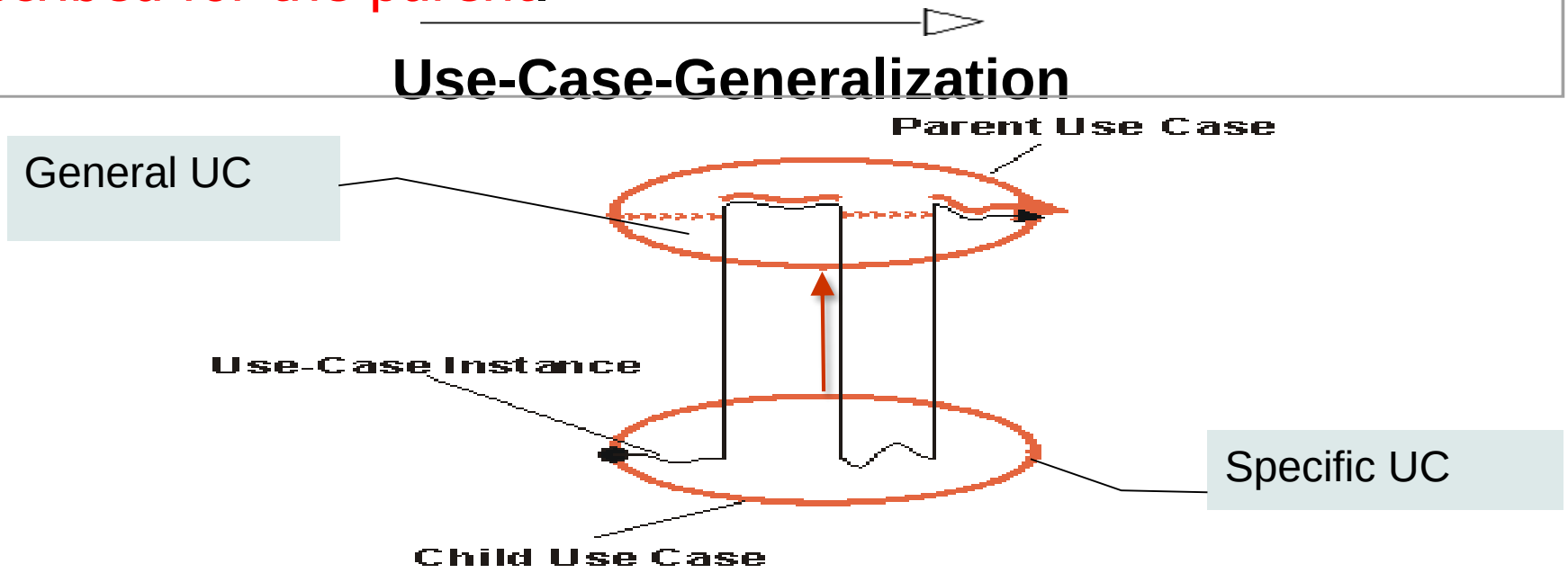




Supplier	The element(s) independent of the client element(s), in the same respect and the same dependency relationship. In some directed dependency relationships (such as Refinement Abstractions), a common convention is to put the more abstract element in this role. However, we can make a more abstract element dependent on that which is more specific.
Client	The element(s) dependent on the supplier element(s). In some cases (such as a Trace Abstraction) the assignment of direction (that is, the designation of the client element) is at the discretion of the modeler, and is a stipulation.

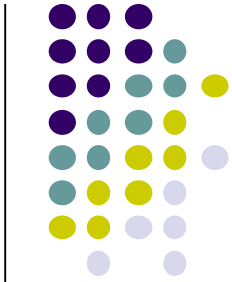
Use-Case-Generalization

A **use-case-generalization** is a taxonomic relationship from a child use case to a more general, parent use case, specifying how a child can specialize all behavior and characteristics described for the parent.



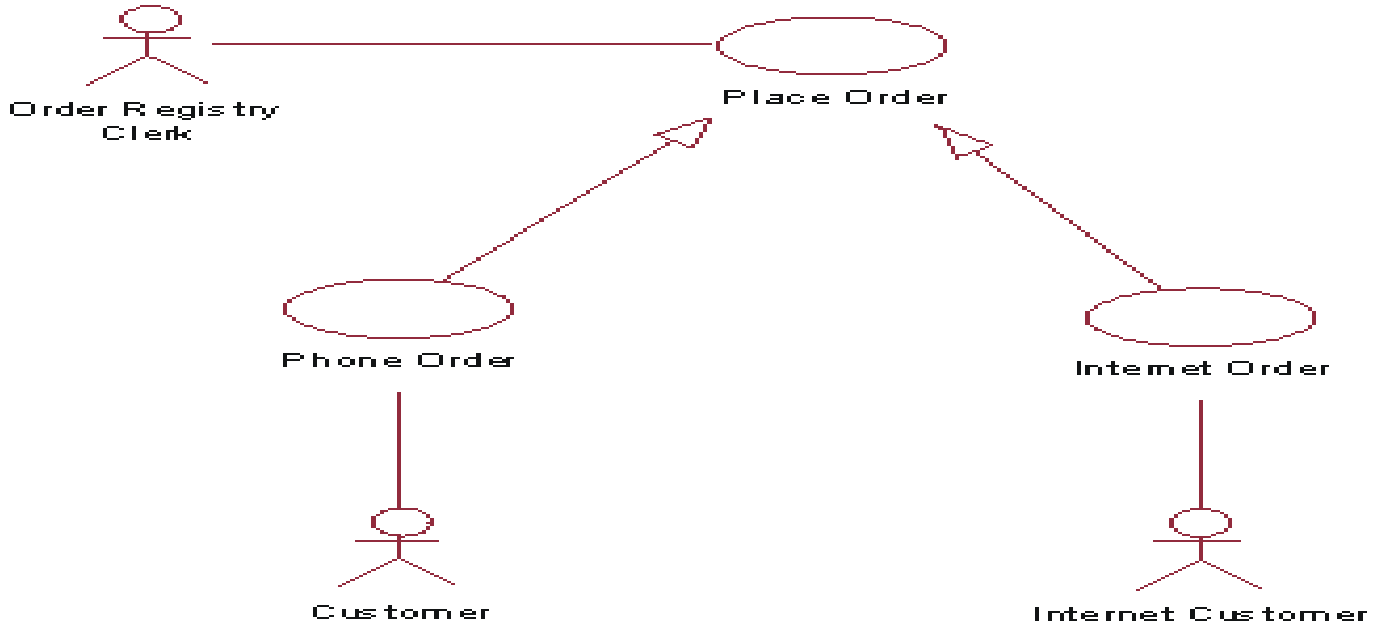
Execution: the use-case instance follows the parent use case, with behavior inserted or modified as described in the child use case.

More about Generalization



General	References the general classifier in the Generalization relationship.
Specific	References the specializing classifier in the Generalization relationship.
Substitu-table	Indicates whether the specific classifier can be used wherever the general classifier can be used. If true, the execution traces of the specific classifier will be a superset of the execution traces of the general classifier.

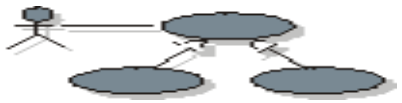
Example of Use-Case-Generalization



The actor Order Registry Clerk can instantiate **the general use case Place Order**. Place Order can also **be specialized** by the use cases Phone Order or Internet Order.

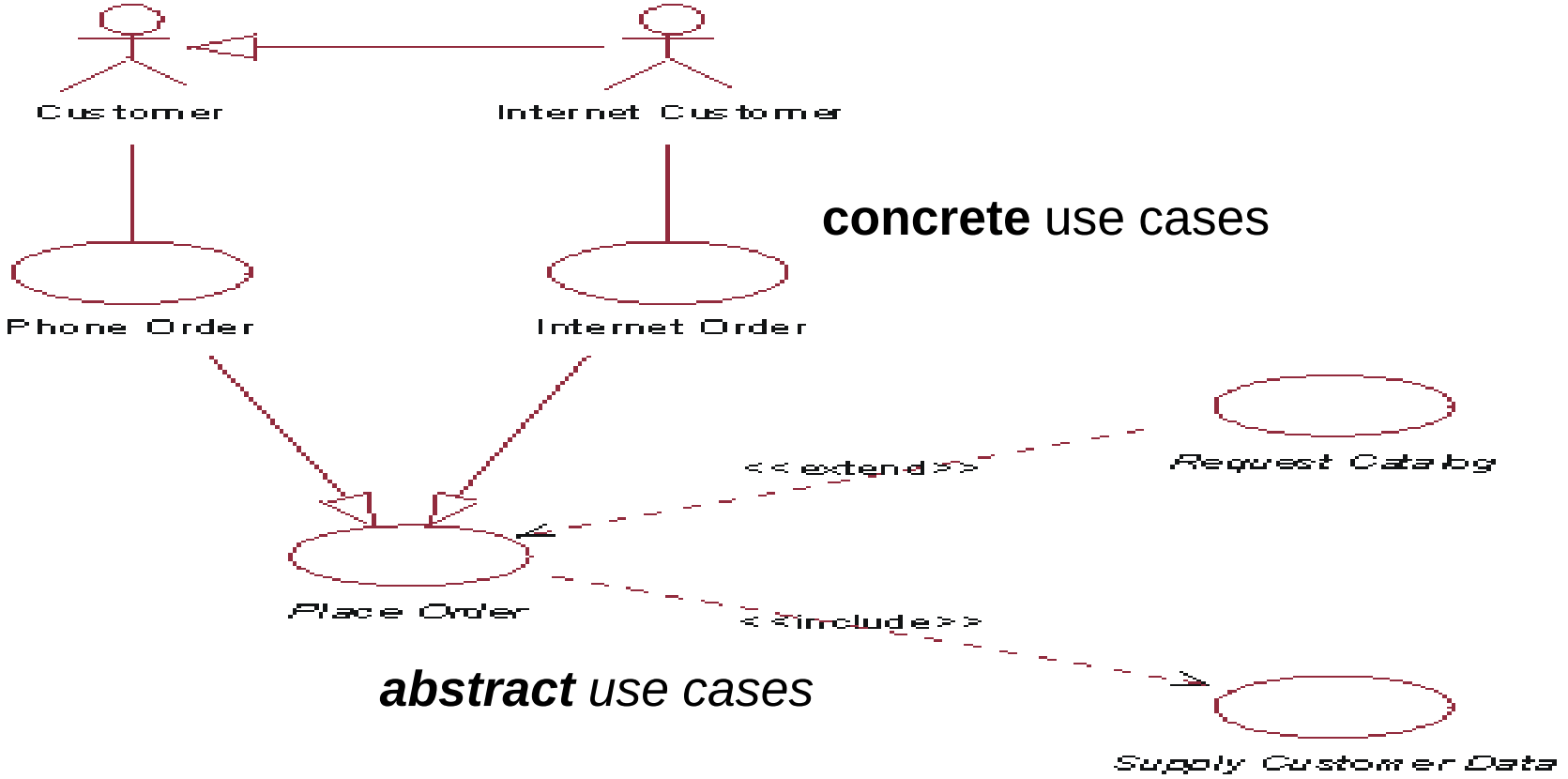
The child may modify behavior segments inherited from the parent. The structure of the parent use case is preserved by the child. Both use-case-generalization and include can be used to reuse behavior among use cases.

Use-Case Model of an Order Management System



Use-Case Model

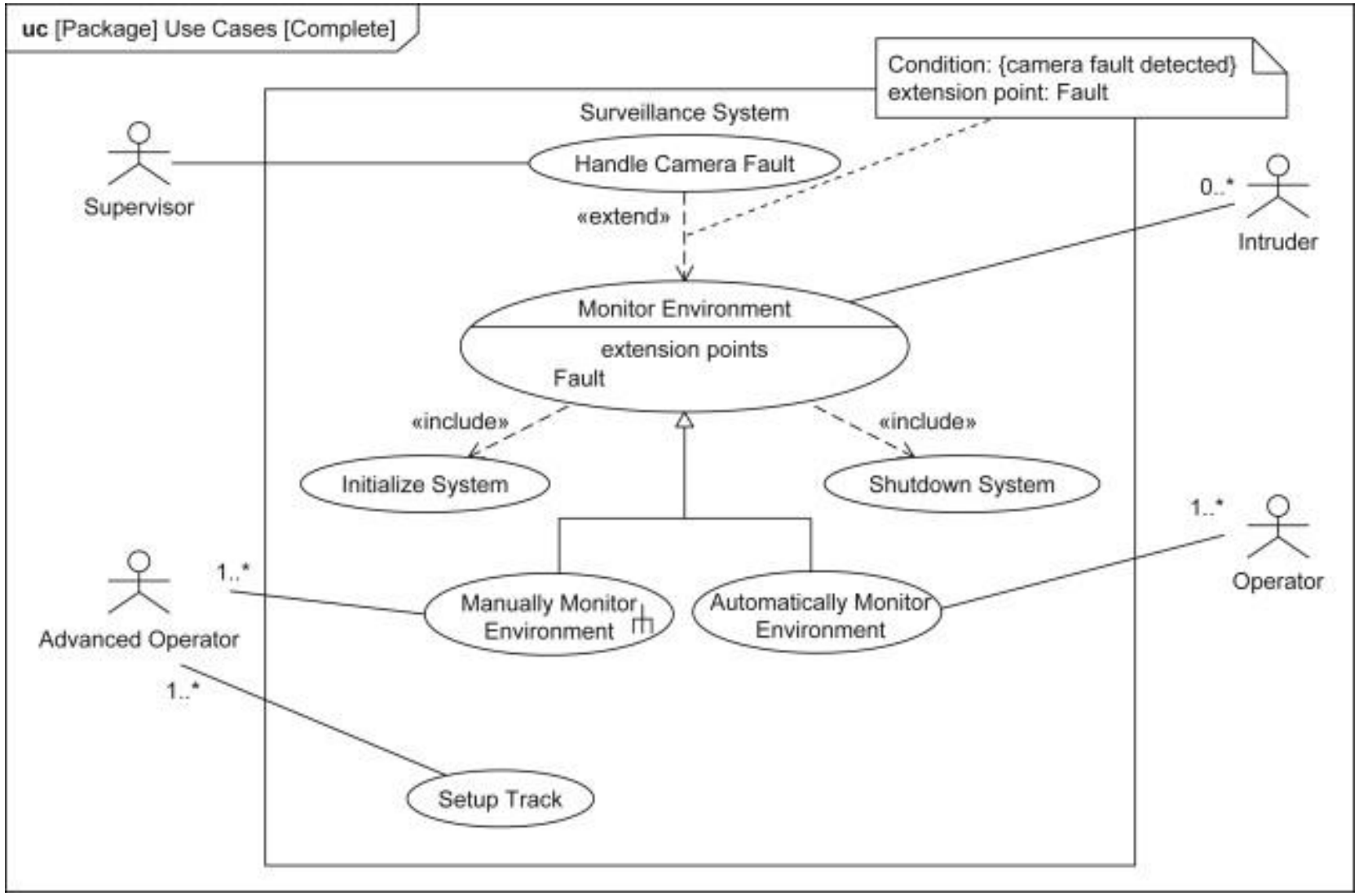
The **use-case model** is a model that describes a system's requirements in terms of use cases.



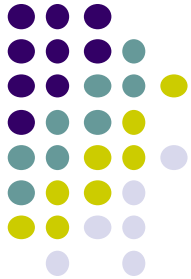
concrete use cases

***abstract** use cases*

Other example



Collaboration

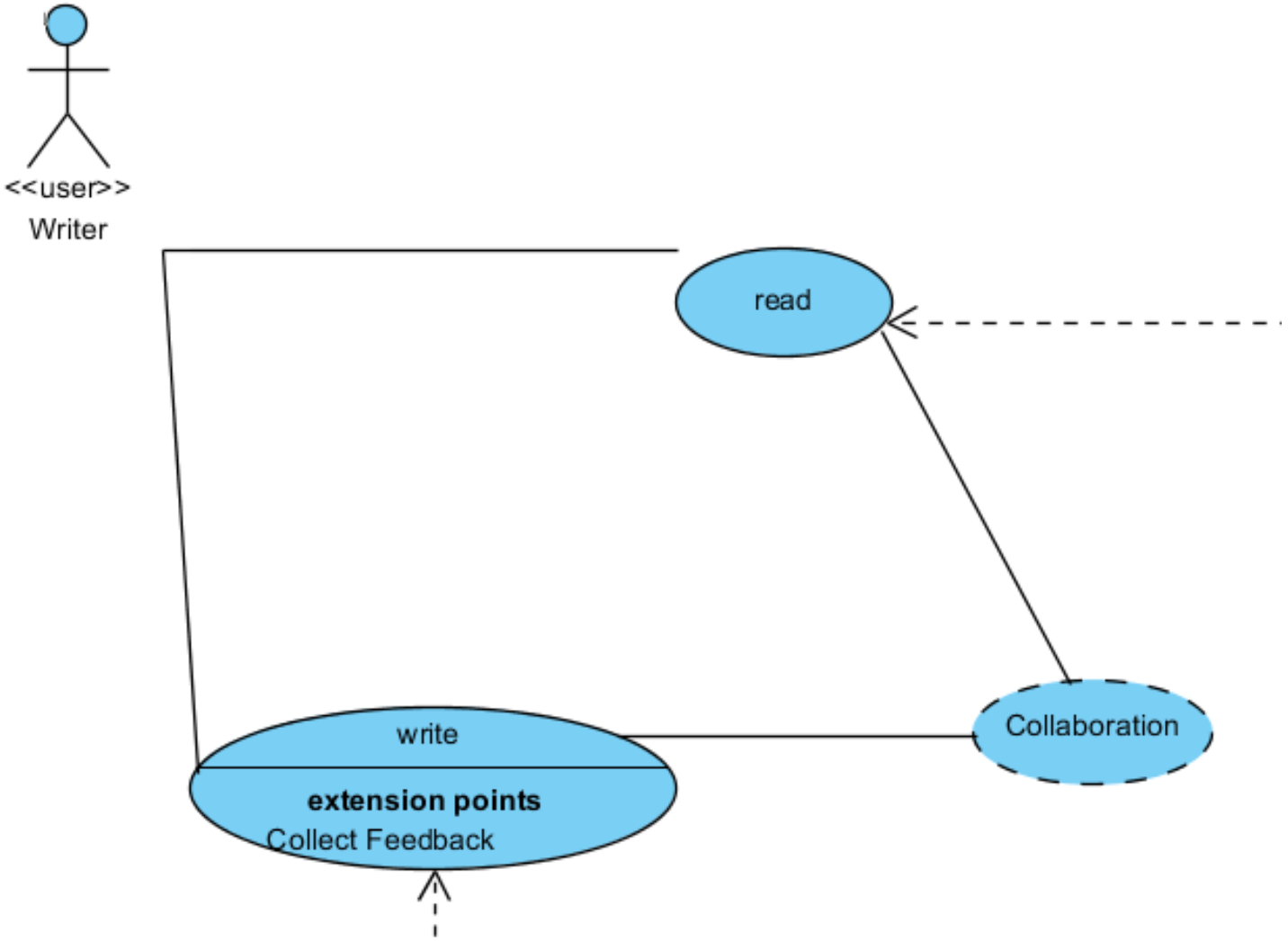


- A collaboration describes a container of collaborating elements, each performing a specialized function, which collectively accomplish some desired functionality.
- Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity or precise class of the actual participating instances are suppressed.

Source: UML Superstructure Specification version 2.4.1, page 174

UML collaboration

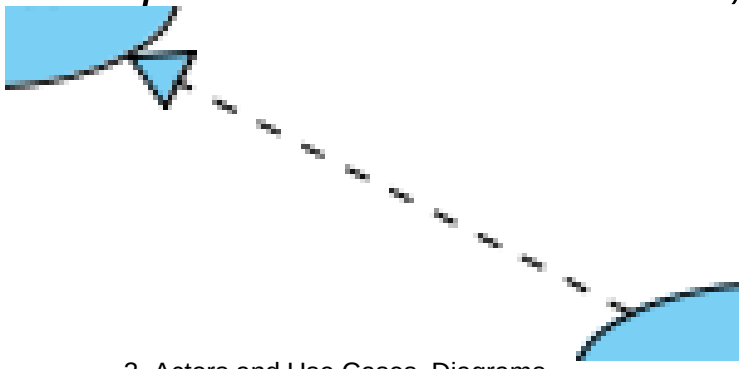
Collaboration example



Use case realization

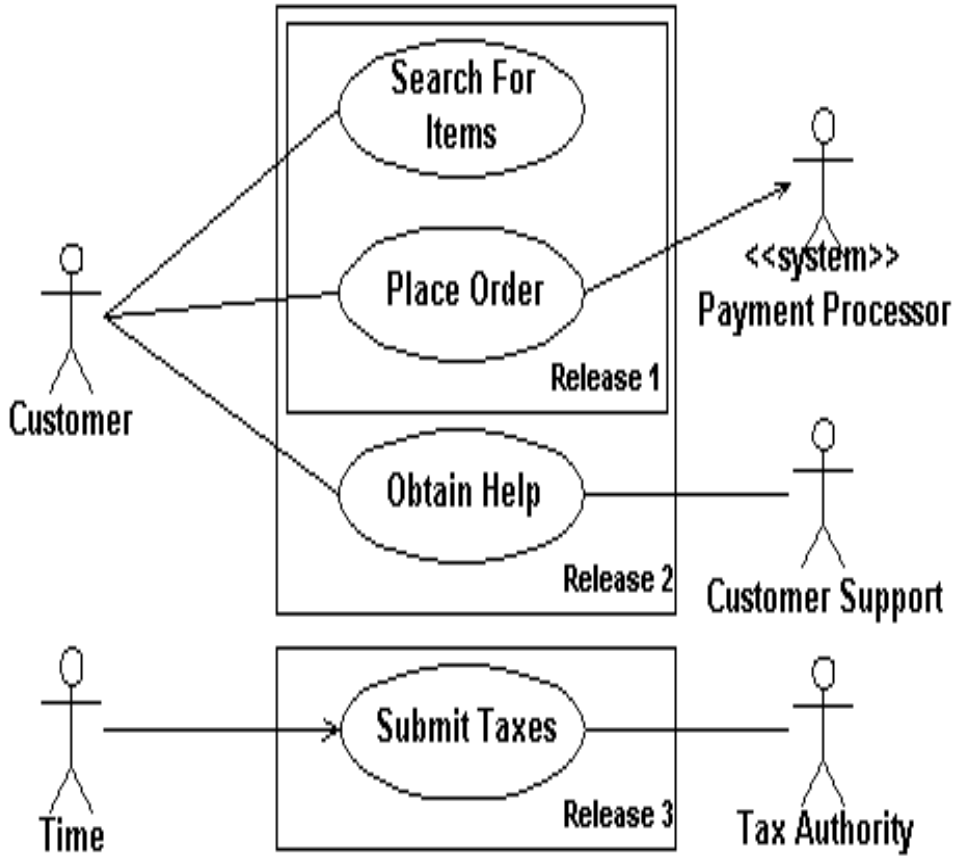
- A realization is a relationship between a specification and its implementation
- Realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other represents its implementation (the client).
- Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.

Source: UML Superstructure Specification version 2.4.1, page 131

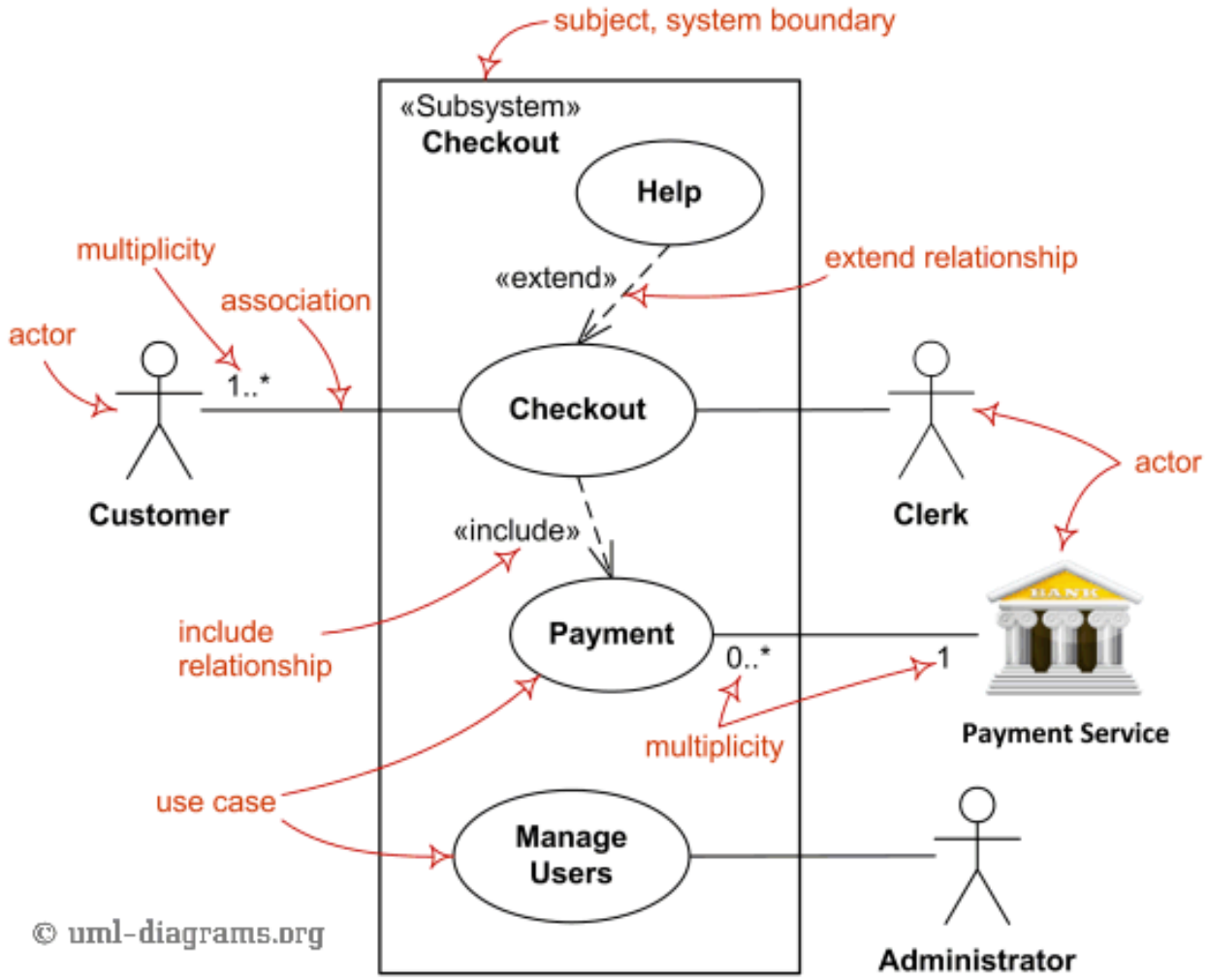


System boundary boxes

- **System boundary box (optional)** - a rectangle around the use cases to indicates the scope of your sub-system
- Anything within the box represents functionality that is in scope and anything outside the box is not
- Rarely used – i.e., to identify which use cases will be delivered in each major release of a system

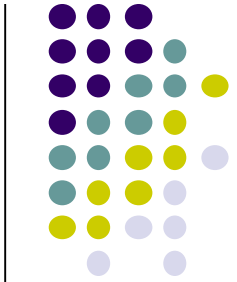


System use case diagram example



Source: http://www.uml-diagrams.org/use-case-diagrams.html

Business use case diagrams

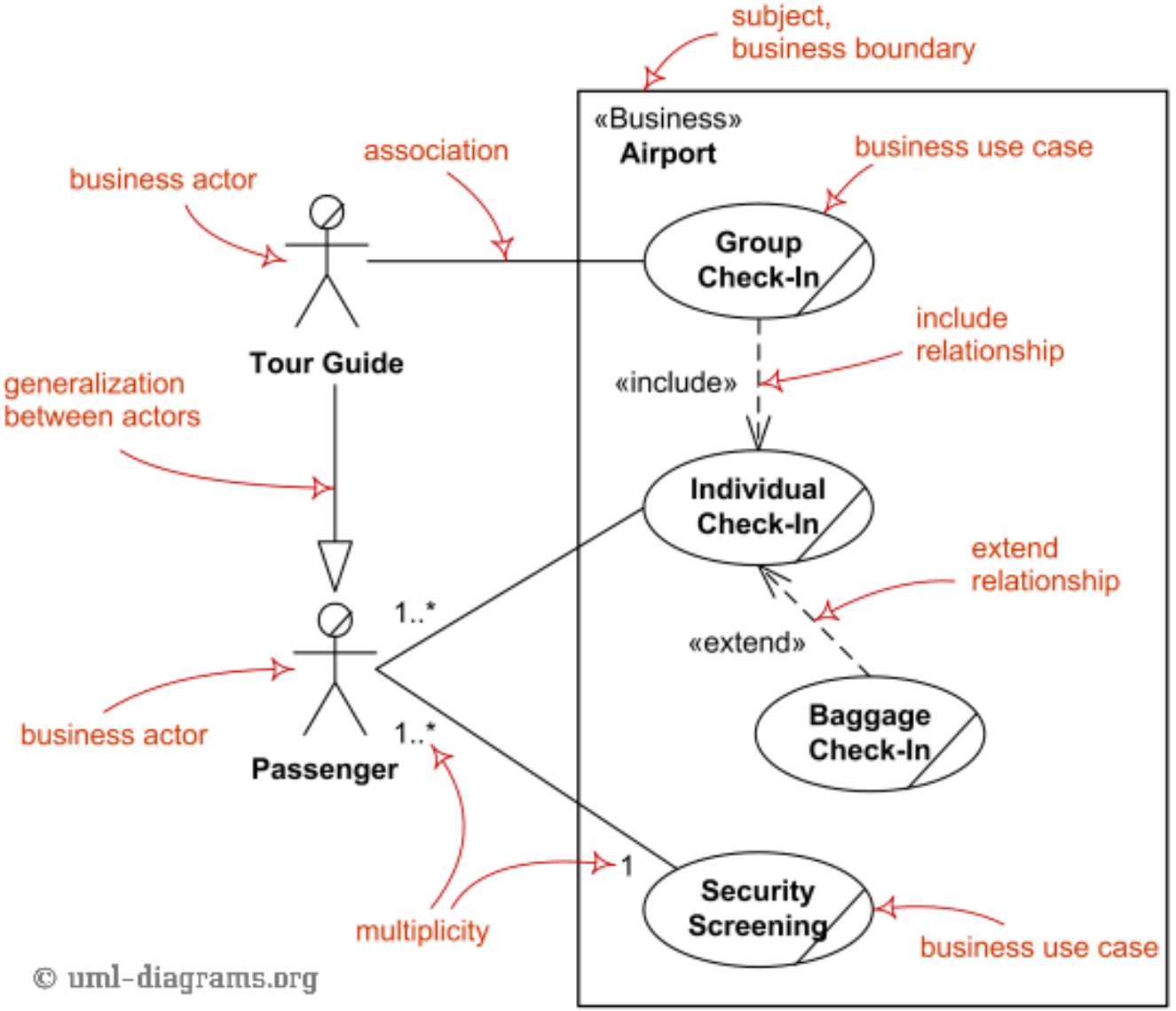


- While support for business modeling was declared as one of the goals of the UML, UML specification provides no notation specific to business needs.
- Business use cases were introduced in Rational Unified Process (RUP) to represent business function, process, or activity performed in the modeled business.
- A ***business actor*** represents a role played by some person or system external to the modeled business, and interacting with the business.
- A ***business use case*** should produce a result of observable value to a business actor.

Source: http://www.uml-diagrams.org/use-case-diagrams.html

Source: http://www.uml-diagrams.org/use-case-diagrams.html

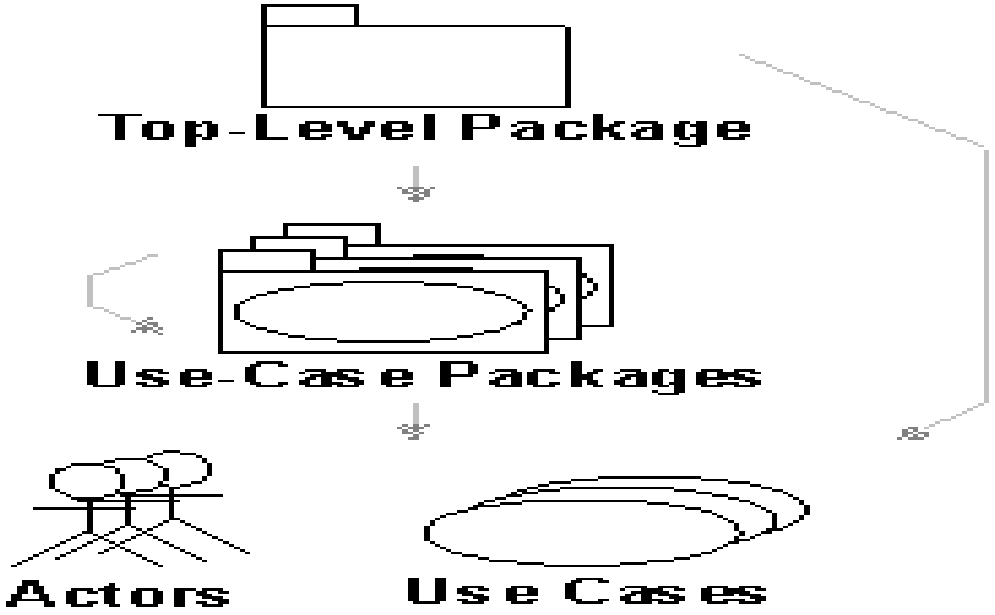
Business use case diagram example



Use-Case Packages

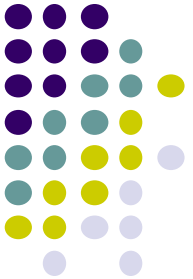
Use-Case Package

A **use-case package** is a collection of use cases, actors, relationships, diagrams, and other packages; it is used to structure the use-case model by dividing it into smaller parts.



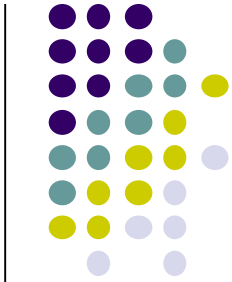
A graph showing the use-case model hierarchy. Arrows indicate possible ownership.

Use cases and requirements capturing – concepts [Bruegge&Duttoit, 2004]



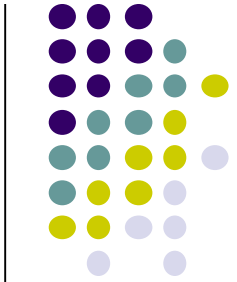
- Req. capturing is focused on the purpose of the system
- Req. spec serves as a contract b/w clients and developers
- Req. capture and analyses are concentrated on the user's view of the system
- Functional and non-functional req's (usability, reliability, adaptability, maintainability, performance, scalability…)
- Completeness, consistency, unambiguity and correctness – definitions?

Use cases and requirements capturing – concepts (2)



- Realism – the system req. lead to realization under constraints
- Traceability – each system function can be traced back to its corresponding set of req's
- Verifiability and validability – after construction phase, repeatable tests can be built for proving the system fulfils the req. spec. Examples for non-verifiable req's:
  - Our product shall have a good GUI.
  - The system should be error free.
  - The response time will be less than 5 sec's for most of the cases providing there is assured high bandwidth.
  - ….

Requirements capturing activities



- Identifying Actors
- Identifying Scenarios
- Identifying Use Cases
- Refining Use Cases
- Identifying Relationships among Actors and Use Cases
- Identifying Initial Analyses Objects
- Identifying Non-functional Req's

Identifying actors for FRIEND

[Bruegge&Dutoit, 2004]

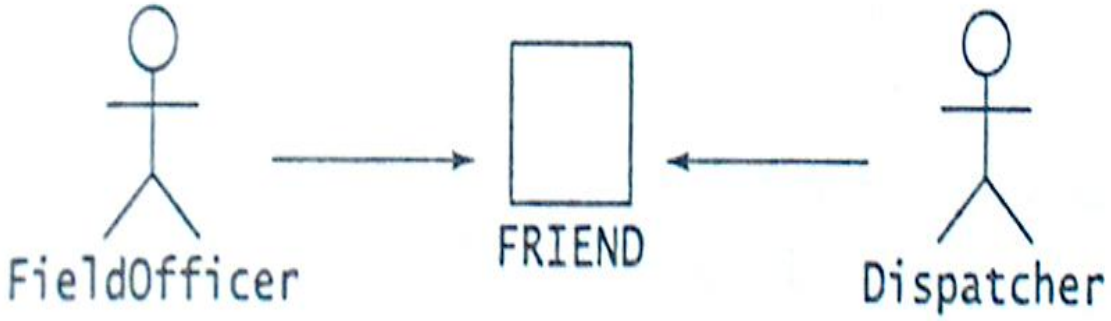


Figure 4-5 Actors of the FRIEND system. FieldOfficers not only have access to different functionality, they use different computers to access the system.

FRIEND = First Responder Interactive Emergency Navigational Database

Identifying use cases in FRIEND

[Bruegge&Dutoit, 2004]

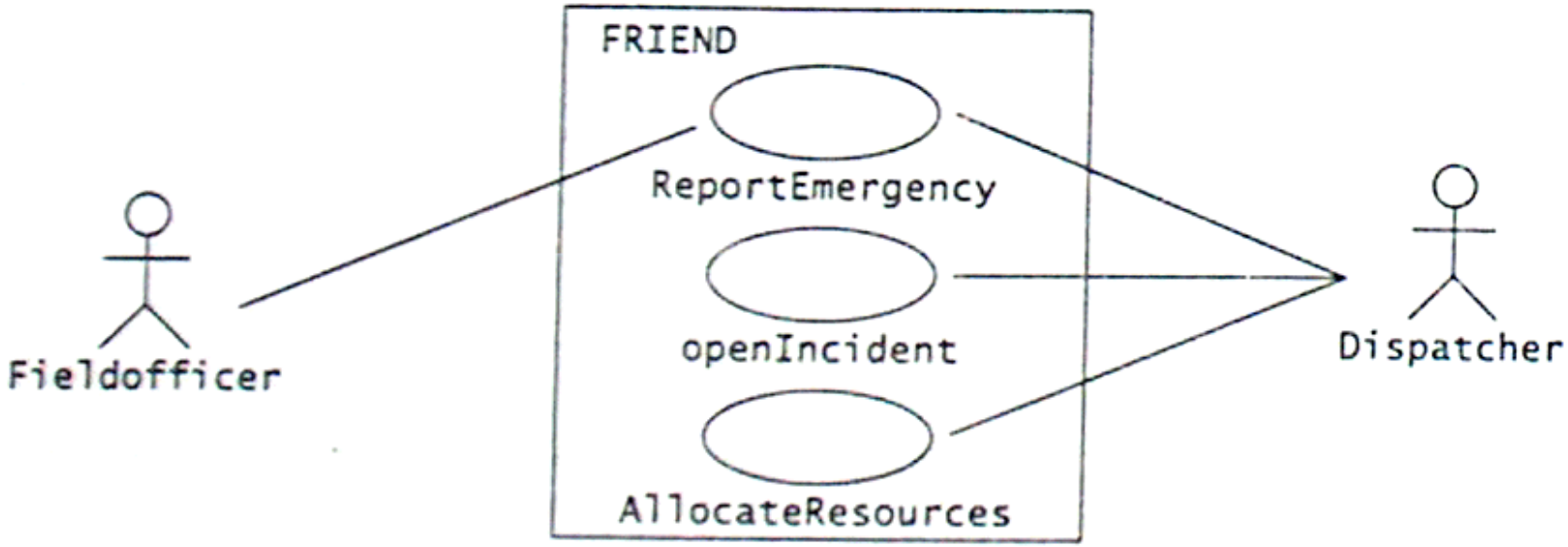
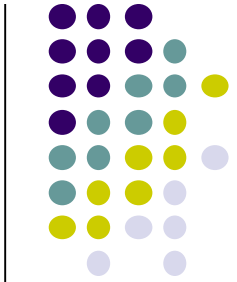


Figure 2-13 An example of a UML use case diagram for First Responder Interactive Emergency Navigational Database (FRIEND), an accident management system. Associations between actors and use cases denote information flows. These associations are bidirectional: they can represent the actor initiating a use case (FieldOfficer initiates ReportEmergency) or a use case providing information to an actor (ReportEmergency notifies Dispatcher). The box around the use cases represents the system boundary.

Refining use cases in FRIEND - a bad and a good use case example

[Bruegge&Dutoit, 2004]



Use case name	`Accident`	*Bad name: What the user is trying to accomplish?*
Initiating actor	Initiated by `FieldOfficer`	
Flow of events	1. The `FieldOfficer` reports the accident.	
	2. An ambulance is dispatched.	*Causality: Which action caused the `FieldOfficer` to receive an acknowledgment?* *Passive voice: Who dispatches the ambulance?*
	3. The `Dispatcher` is notified when the ambulance arrives on site.	*Incomplete transaction: What does the `FieldOfficer` do after the ambulance is dispatched?*

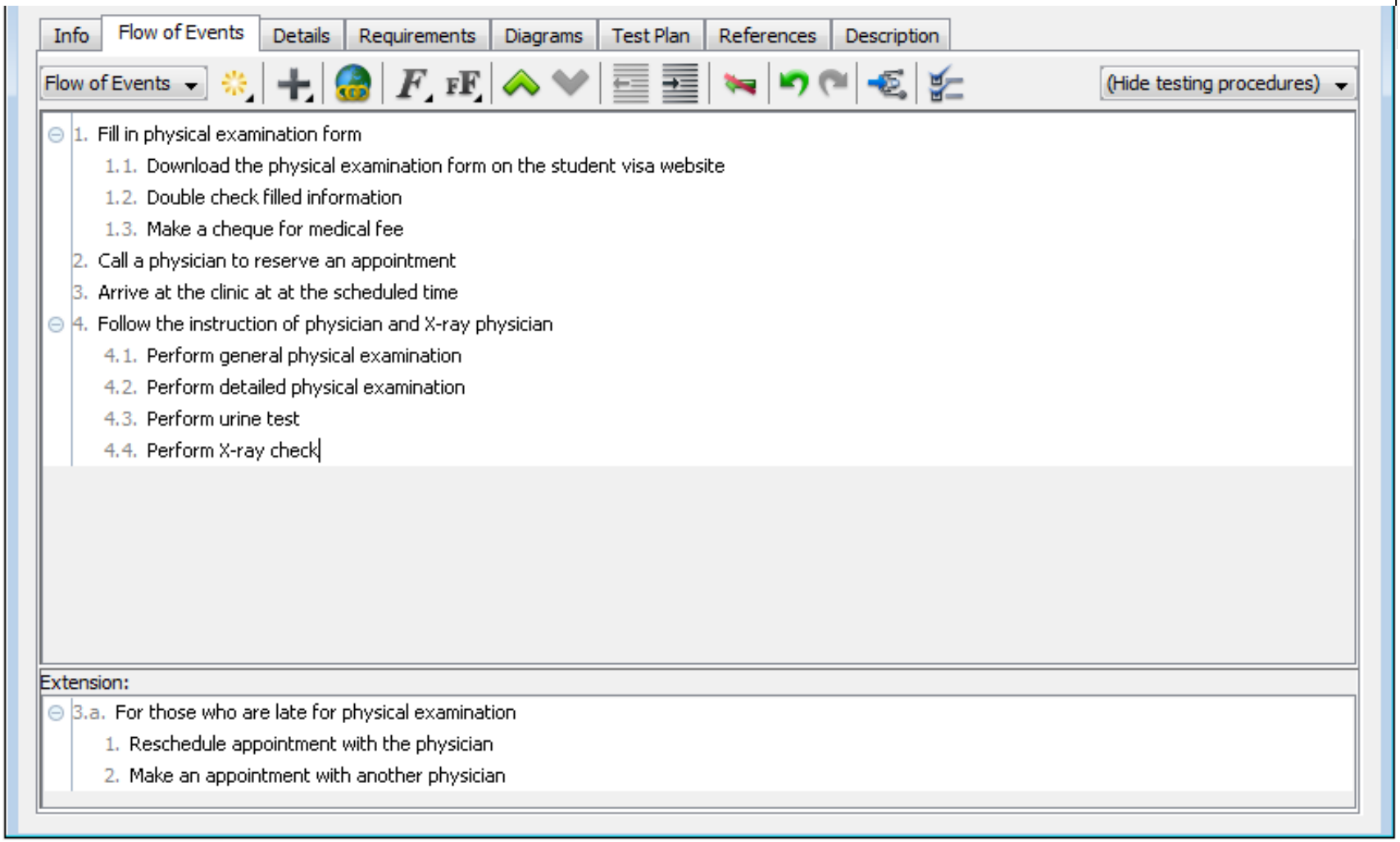
Figure 4-9 An example of a poor use case. Violations of the writing guide are indicated in *italics* in the right column.

FRIEND = First Responder Interactive Emergency Navigational Database

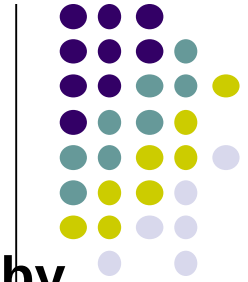
Use case name	ReportEmergency
Participating actors	Initiated by FieldOfficer Communicates with Dispatcher
Flow of events	1. The FieldOfficer activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the FieldOfficer. 3. The FieldOfficer fills out the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form. 4. FRIEND receives the form and notifies the Dispatcher. 5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report. 6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.
Entry condition	• The FieldOfficer is logged into FRIEND.
Exit condition	• The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR • The FieldOfficer has received an explanation indicating why the transaction could not be processed.
Quality requirements	• The FieldOfficer's report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Figure 2-14 An example of a use case, ReportEmergency.

Documenting Flow of Events in Visual Paradigm



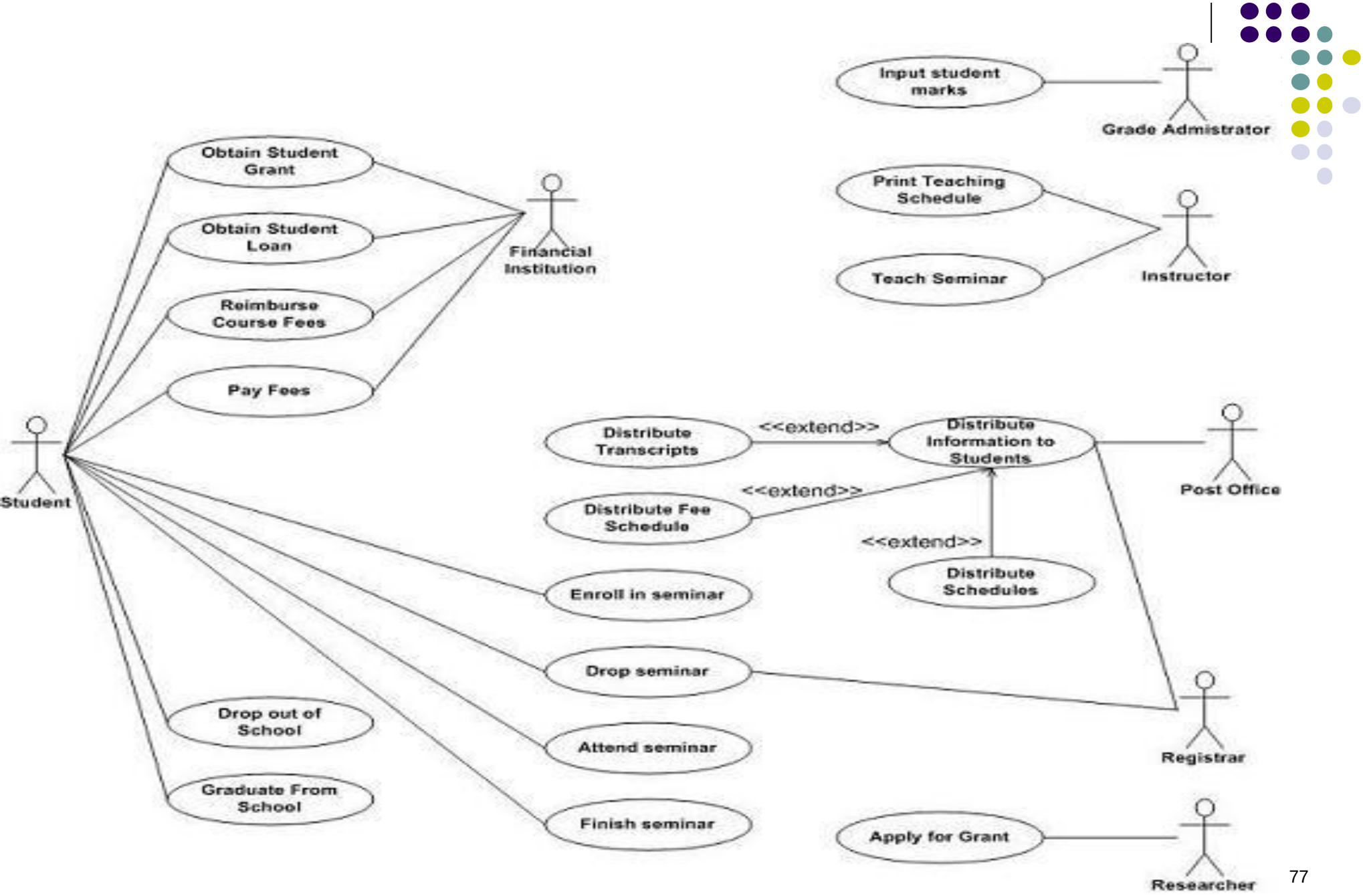
Example: The University Course Registration (UCR) Case Study



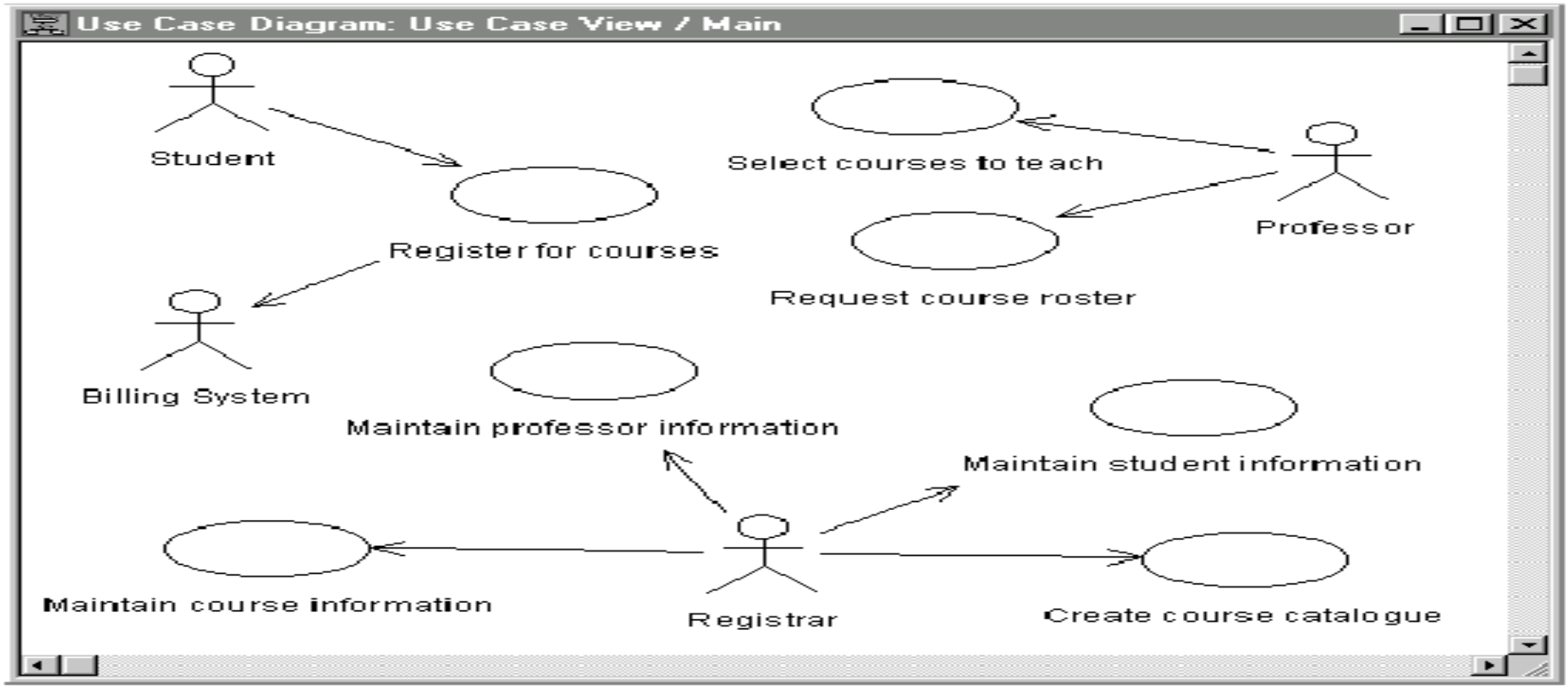
Goal: **to assign automatically students to the courses teach by University professors.**

Current process:

1) Batch report printed with all the courses teach by the professors
2) Students filled out course registration form (no course offering for more than 20 students and less than five; four courses for each student)
3) Registrar's office processes the forms
4) Processing the conflicts – first choice usually is OK but in case of conflicts Registrar's officers talk to students to get additional choices
5) After successful assignment to courses, hard copy of the curriculum is sent to the students
6) Professors receive a student roster for each course they are scheduled to teach

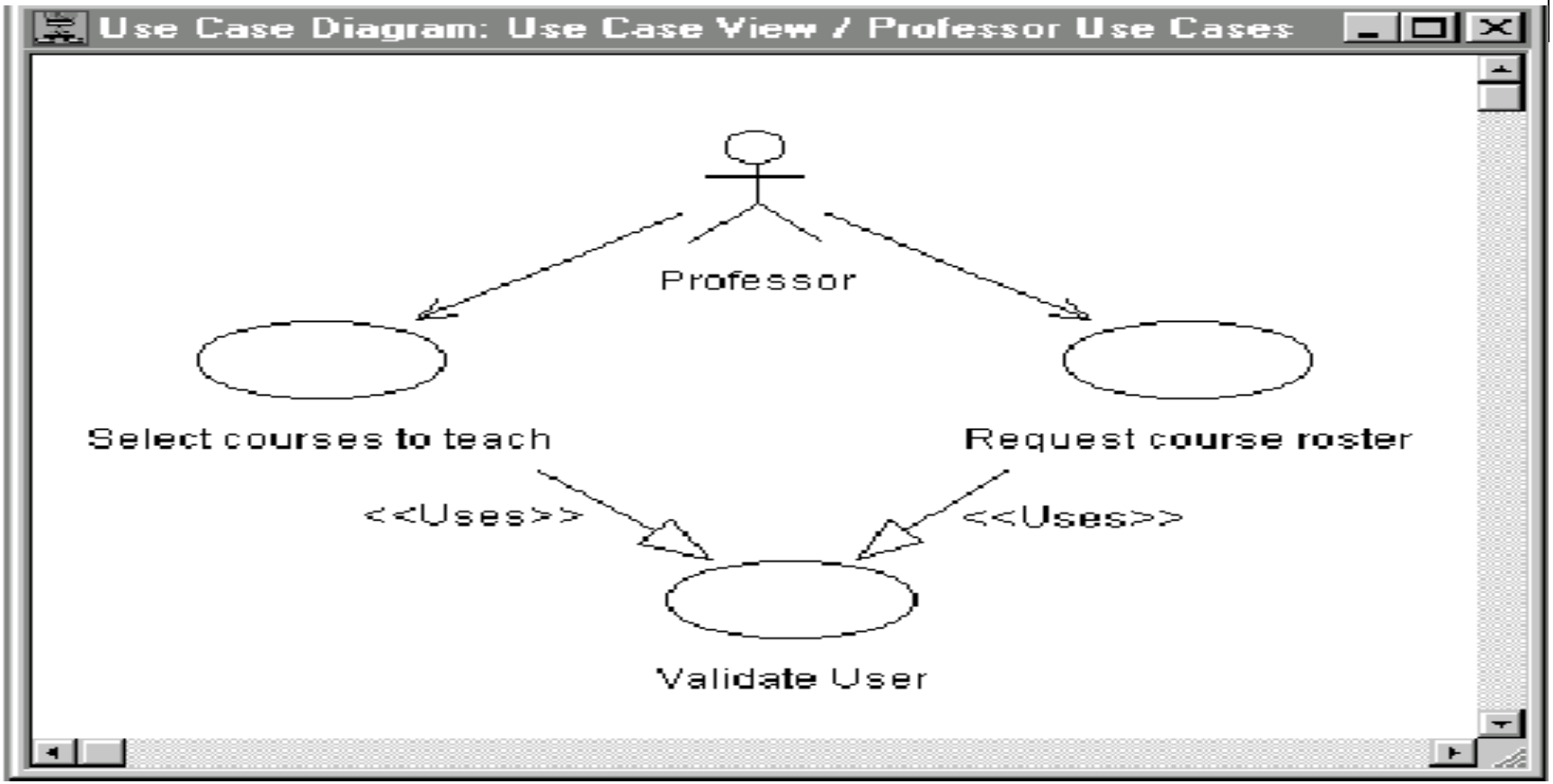

Input student marks
Grade Administrator
Obtain Student Grant
Obtain Student Loan
Reimburse Course Fees
Pay Fees
Financial Institution
Print Teaching Schedule
Teach Seminar
Instructor
Student
Distribute Transcripts
<<extend>>
Distribute Information to Students
Post Office
Distribute Fee Schedule
<<extend>>
<<extend>>
Distribute Schedules
Enroll in seminar
Drop seminar
Registrar
Drop out of School
Attend seminar
Graduate From School
Finish seminar
Apply for Grant
Researcher

URC Use Cases



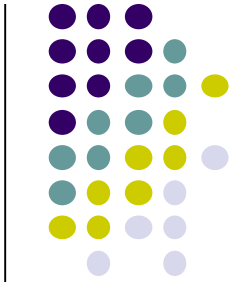
Main use case diagram

URC Use Cases - 2



An additional use case diagram

For Homework



Read the VP tutorials:

- How to Generate Use Case Scenario from Notes? - https://www.visual-paradigm.com/support/documents/vpuserguide/94/2575/83684_produceuseca.html

- Documenting use case details - https://www.visual-paradigm.com/support/documents/vpuserguide/94/2575/21179_documentingu.html

Q & A