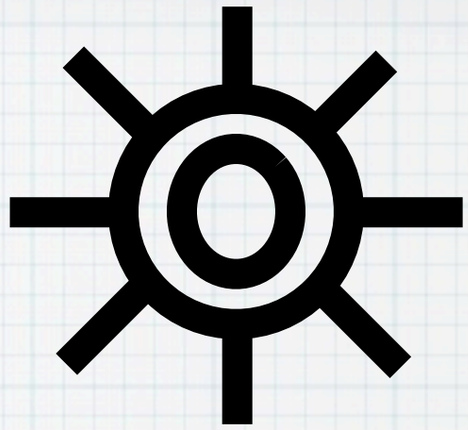
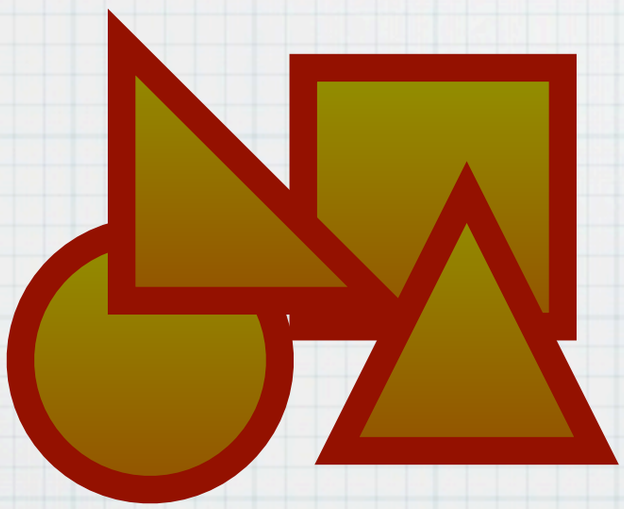


# Object Modeling with UML

## Fundamentals

Les Waguespack, Ph.D.



Slides Two

# Copyright and References




* UML 2 and the Unified Process 2nd Ed - Practical Object-Oriented Analysis and Design, Arlow & Neustadt, Addison-Wesley / Pearson Education, Inc., Boston, MA, ISBN 0-321-32127-8
* UML 2 Toolkit, Eriksson, Penker, Lyons & Fado, Wiley, Indianapolis, IN, ISBN 0-471-46361-2
* UML 2.0 - Superstructure, Object Management Group, http://www.omg.org/cgi-bin/doc?formal/05-07-04
* Object Oriented Analysis, 2nd Ed, Peter Coad and Edward Yourdan, Prentice-Hall, 1991.ISBN 978-0136299813
* Business Modeling With UML, Eriksson & Penker, Wiley, Indianapolis, IN, ISBN 0-471-29551-5
* Enterprise Modeling With UML Designing Successful Software Through Business Analysis, Addison-Wesley, Reading, MA, ISBN 0-201-43313-3
* Use Case Modeling, Bittner & Spence, Addison-Wesley / Pearson Education, Inc., Boston, MA, ISBN 0-201-70913-9
* Writing Effective Use Cases, Cockburn, Addison-Wesley, Boston, MA, ISBN 0-201-70225-8
* Object Oriented Systems Engineering, Waguespack, course notes CS390, CS460, CS630, CS771, Computer Information Systems Department, Bentley College, Waltham, MA.

# Outline

* 1. Overview
* 2. Diagramming in UML 2
  * 2.1 Class
  * 2.2 Use Case
  * 2.3 Sequence
  * 2.4 Activity
  * 2.5 Thumbnails of all the others

# 1. Overview

* Object-Oriented Modeling is based on a system of concepts that define the existence and relationships of facts within a defined system boundary.
* The system of concepts is called the Object-Oriented Paradigm.
* OOM is independent of UML or any other OO language: C++, Java, Smalltalk, C#, ...
* The Object-Oriented Paradigm is stable, well understood and documented.
* UML is an evolving, growing tool attempting to address a growing and evolving industry of system development.

http://www.omg.org/uml-certification/exam_info.htm

**OMG Certified**  **UML Professional**



# The Exams

There are three OCUP Exams - Fundamental, Intermediate and Advanced. Each Exam tests your knowledge of a different subset of the UML. Certification indicates the following abilities and qualifications.

Follow the links below for detailed information on each Exam.

| | |
|---|---|
| **Fundamental** | You can work with the most commonly encountered UML elements<br>You can create simple UML models<br>You are qualified to be a member of a UML Development Team. |
| **Intermediate** | You can work with a broad range of UML elements<br>You can create complex UML models<br>You are qualified to be a senior member of a UML Development Team. |
| **Advanced** | You can work with the full range of UML elements<br>You can create extremely large, complex UML models<br>You are qualified to manage a UML Development Team. |

# Coverage Map for the *OMG-Certified UML Professional Fundamental Exam*

***Topic Area Allocation***

Topic Area -->

| Topic Area | Percent of test this topic should represent |
|---|---|
| 1.0 Class Diagrams (Basic) | **30%** |
| 2.0 Activity Diagrams (Basic) | **20%** |
| 3.0 Interaction Diagrams (Basic) | **20%** |
| 4.0 Use Case Diagrams (Basic) | **20%** |
| 5.0 Miscellaneous basic notions | **10%** |
| **Total** | **100%** |

# UML Diagrams

- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams
    - **Class** - business objects and their structures
    - Composite Structure - nested contents of structured classifiers
    - Component - modules and replaceable parts of system
    - Deployment - maps software architecture to physical system architecture
    - Object - depicts the state of objects at a point in time
    - Package - a collection of classes forming a cohesive subsystem of concepts
  - Dynamic diagrams
    - **Activity** - object oriented "flowcharts"
    - Interaction - diagrams that depict the "active" relationship between objects
      - **Sequence** - time-ordered inter-object messages that complete a task
      - **Communication** - message traffic among classes in a class structure
      - Interaction Overview - show high level flow of control between interactions
      - Timing - real-time dependent object relationships
    - **Use Case** - user / system interactions / interfaces
    - State Machine - depicts stable points in process flow yielding predictable conditions

# 2. Diagramming in UML 2

- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams (static in UML 1)
    - **Class** - business objects and their structures
    - Composite Structure 2
    - Component
    - Deployment
    - Object
    - Package
  - Dynamic diagrams
    - **Activity**
    - Interaction
      - **Sequence** - system actions that complete a task
      - **Communication** (collaboration in UML 1)
      - Interaction Overview 2
      - Timing 2
    - **Use Case** - user / system interactions / interfaces
    - State Machine

2 new in UML2

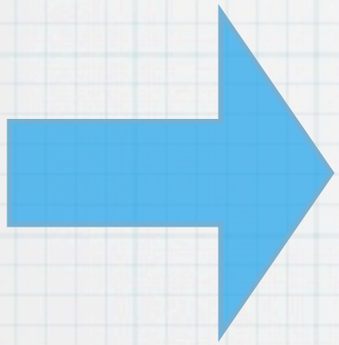
# Elements of UML 2

* Specifications
  * graphical - diagrams and icons to support visualization
  * textual - descriptions of the semantics of elements
* Adornments
  * additions to basic modeling elements that highlight important details
* Common Divisions
  * classifier and instance - categorization / realization of model elements
  * interface and implementation - separating the "how" from the "what"
* Extensibility mechanisms
  * constraints - allow adding new rules to modeling elements
  * stereotypes - allow adding new modeling elements beyond UML 2
  * tagged values - allow adding new properties to model elements
  * UML profiles - allow grouping the above as a "modeling template"

# "Ease of Use" vs. "Ease of Use"

* UML 1.x was developed primarily to support modeling in the analysis and design phases of software system development
* UML 2.0 is a refinement of UML consistent with OMG's Model Driven Architecture philosophy that allows UML models to be input, transformed, and reconfigured automatically by model compilers.
* UML 2.0 achieves this extended functionality by adding significant rigor, detail and complexity to the syntax and semantics of the modeling language.
* UML is a TOOL and as such every artisan, technician, and builder will need to assess the breadth and width of UML 2.0 that is appropriate to the task and normalize that "subset" among all the collaborators.

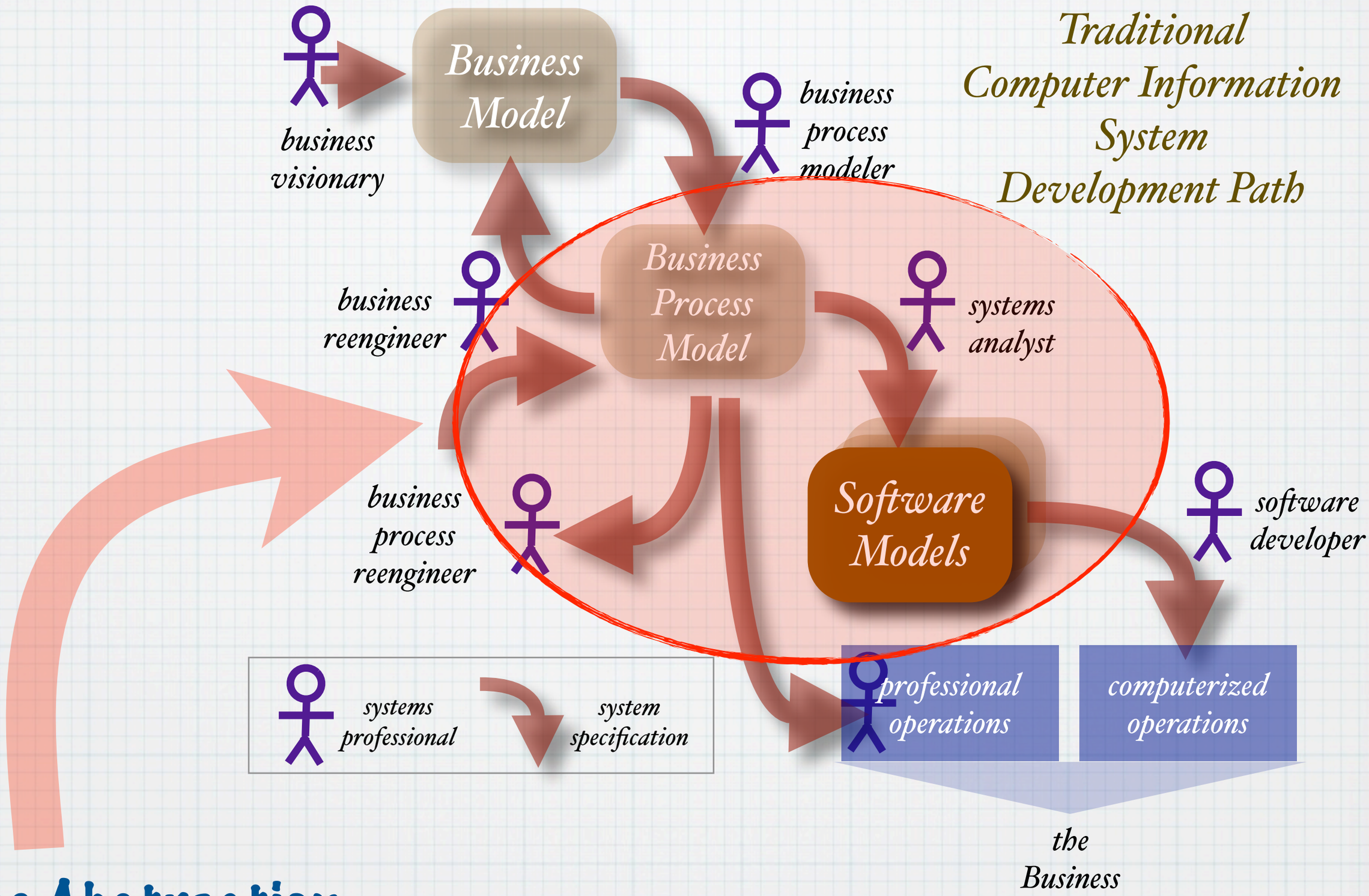
# 2.1 Class Diagramming in UML 2



- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams (static in UML 1)
    - **Class** - business objects and their structures
    - Composite Structure 2
    - Component
    - Deployment
    - Object
    - Package
  - Dynamic diagrams
    - **Activity**
    - Interaction
      - **Sequence** - system actions that complete a task
      - **Communication** (collaboration in UML 1)
      - Interaction Overview 2
      - Timing 2
    - **Use Case** - user / system interactions / interfaces
    - State Machine

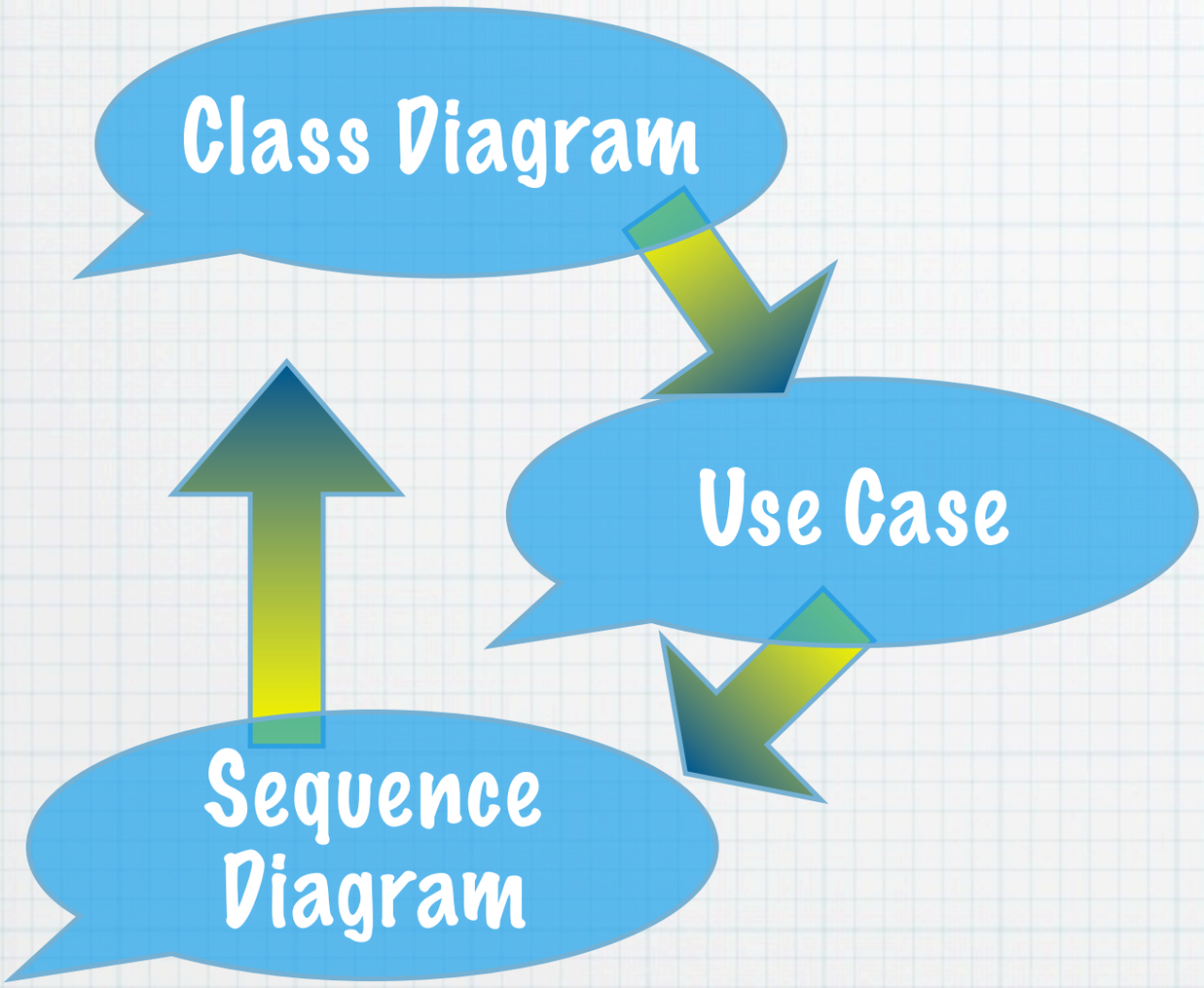
2 new in UML2

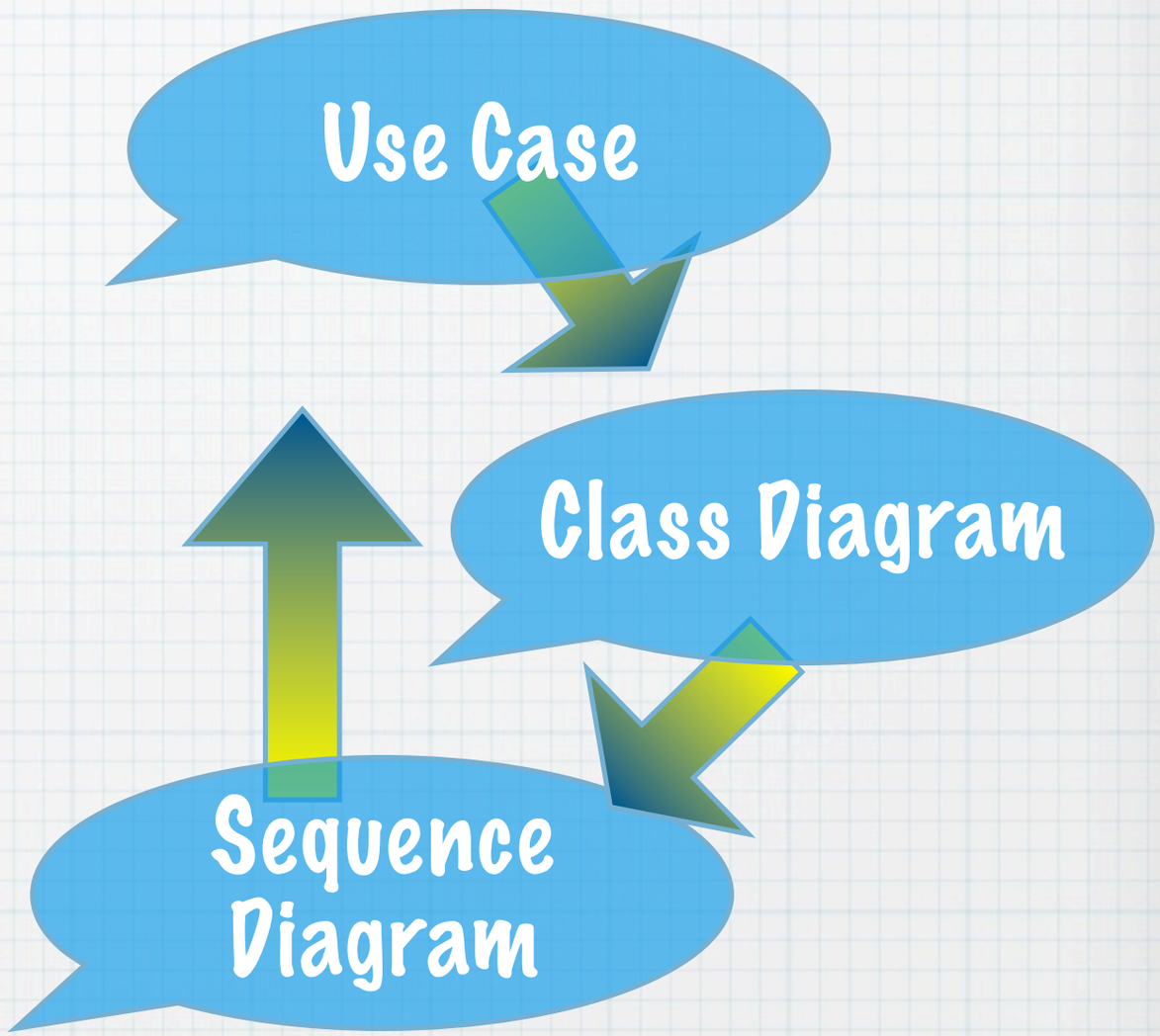
# Traditional Computer Information System Development Path



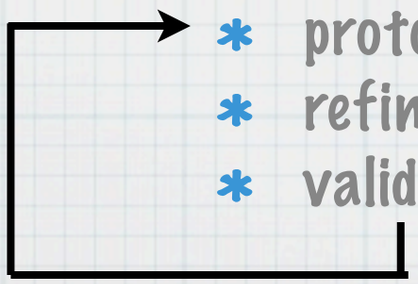

## The Abstraction Focus in this Course

# Model - Go - 'Round




* Modeling is an iterative process
  * prototype
  * refine
  * validate



# Class Diagramming
## A.K.A.
## "Domain Modeling"

* In the overall approach, class diagramming achieves these ends:
  * Identifying business objects and determining their sameness and difference
  * Identifying class structures that explain the sameness and difference of objects
  * Identifying association structures that define accessibility
  * Defining attributes that describe and identify distinct objects
  * Defining services / behaviors that describe the objects actions and responsibilities in the problem domain

# Class Diagramming

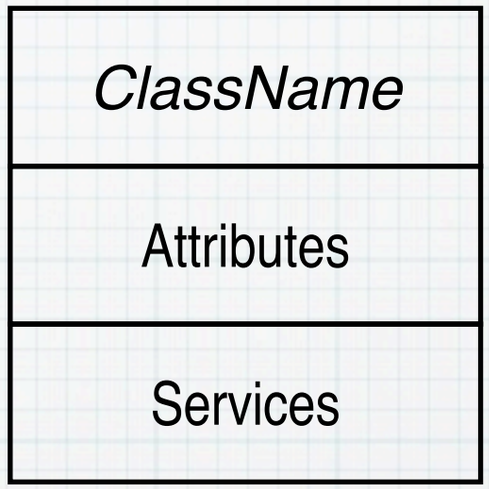
* Finding classes and objects
* Identifying class structures
* Identifying object structures
* Defining class and object attributes
* Defining class and object behaviors
  ( services / methods )

# Finding Classes and Objects

* **Object**. [something thrown in the way (Medieval Latin), a casting before (Latin)] A person or thing to which action, thought, or feeling is directed. Anything visible or tangible; a material product or substance.
  * a uniquely identifiable, attribute value bearing, "living instance"
* **Class**. [a division of the Roman people (Latin); a calling, summons (Greek)] A number of people or things grouped together because of certain likenesses or common traits. [ Webster's, 1977]
  * a template, a "cookie cutter", that defines the structure (memory and behavior) of objects derived from it

# Naming Classes and their Objects

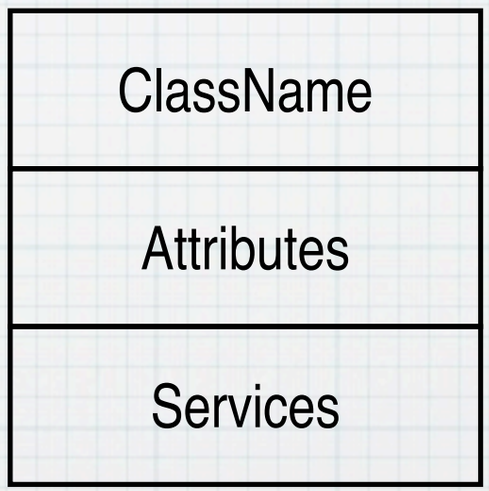
* Class symbol denotes a defined identical structure (attributes and services) that all instances of this class (objects) will share. Notice the **italic** class name!




An abstract description or template for objects of this class. No instances of this class are expected to be found in the problem domain.

Also called an "Abstract Class."

* Class and Object* symbol represents both the defined structure of the class but, also, represents any and all instances that may exist in the problem domain. Notice that the class name is **NOT italic**!



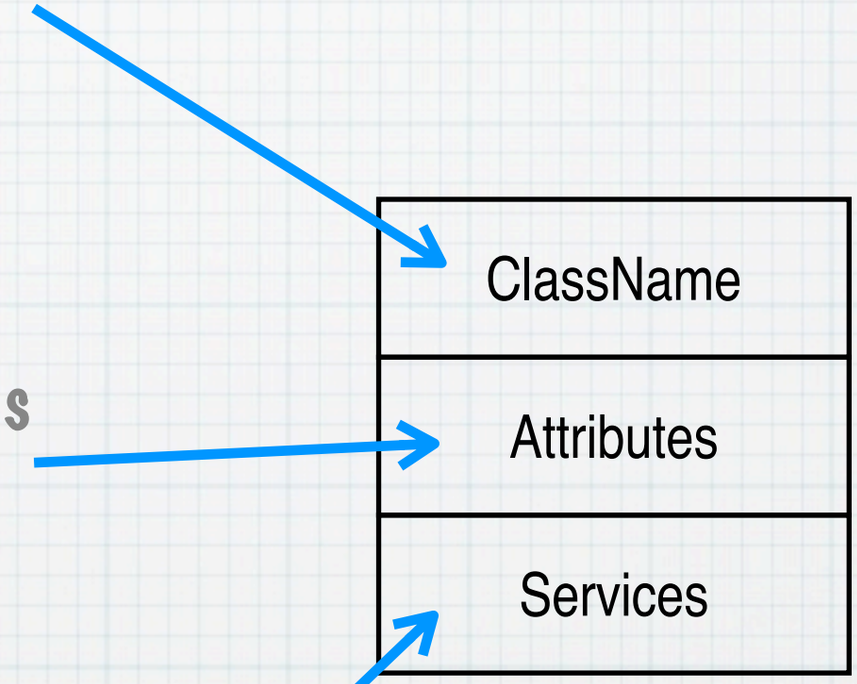

The abstraction and all the instances of this class that ARE expected to be found in the problem domain.

Also called a "Concrete Class."

*class and object is not a standard term in UML, but it better explains the element's purpose.

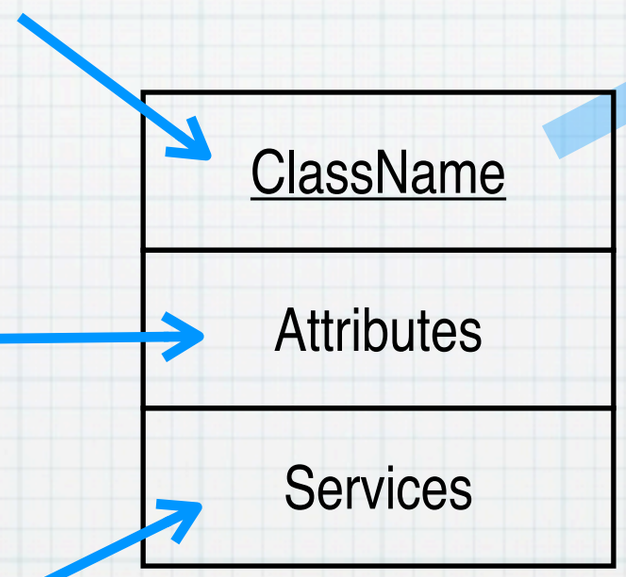
# "Class and Object" Drawn

* A name used in the standard vocabulary of the problem domain. A singular noun, or adjective and noun. Each instance is one item not a group. User client-familiar terms.
* The characteristics of this class and the specific values for an instance. These are defined in the attribute defining activity.
* These are the process functions performed by this object requested by other objects. They are defined in the service defining activity.



# Drawing an Object

* An object is an instance of a class that possesses the identical structure defined in the class, but retains its own values for attributes.
* The <name> is **underlined** to denote it is an object.
* An object may be anonymous (class name only), indeterminate (object name only), or specific (object name and class).




*Variations of <name>*

*Anonymous*:
:ClassName

*Indeterminate*:
myObject

*Specific*:
myObject: ClassName

As a *convention* in class names each and every "word" in the name begins with a capital letter because there are no "special characters" to separate them while in an object's name the very first letter is lower case.

Objects do not appear in class diagrams!

# Where to look for "class and objects"

* Observe first-hand: follow the client around performing normal domain activities, "walk a mile in the client's shoes".
* Seek out problem domain "experts" and have them describe the problem domain to you, what makes it interesting, what is most important (and why), what scenarios are most significant (and why)?
* Refer to previous specifications (hopefully OO), reuse Objects when relevant to this system.
* Seek out other systems with similar behavior or responsibilities.
* Read and re-Read the requesting document, identify the MISSION and the PURPOSE of the system.
* Prototype the object list and review it with the user and domain experts, refine, refine, refine!

# What to look for ...

- Structures: finding structures has its own activity in OOA, Generalization-Specialization and Whole-Part are very fruitful.
- Other Systems: are there "external-terminators" with which interaction is initiated or responded to, other persons, organizations, or systems?
- Devices: what devices will the system interact with?; not computer implementation specific devices like terminals and disk drives but, controls, sensors, monitors in a functional context.
- Things or events remembered: collect a list of all things or events that are "remembered" in the domain, identified by numbers or referred to in documents.
- Roles played: what roles do humans play in relationship to the system, does one individual play more than one role?
- Operational procedures: are there mechanical or clerical procedures that must be followed?
- Sites: are particular locations or contexts important to events?

# What to challenge . . .

* In order to pare down the list of potential objects apply the following tests:
  * Needed Remembrance
    * Are records of this object really used, is the record input to some defined function?
  * Needed Behavior
    * If the object is remembered it will at least have to service "create, connect, access, and release" messages, what else?
  * (Usually) Multiple Attributes
    * Objects are important because they are the "hubs" of function, one attribute objects should seem suspicious!
  * (Usually) More Than One Object in a Class
    * Objects with "proper" names (this object or her object) are probably instances but not classes in themselves!

# What (else) to challenge . . .

* In order to pare down the list of potential objects apply the
  * Always-Applicable Attributes
    * Do all instances of this class have a set of identical attributes? Differing sets of attributes indicate Gen-Spec. Structure!
  * Always-Applicable Services
    * Do all instances of this class have a set of identical services? Differing sets of services indicate Gen-Spec. Structure!
  * Domain-based Requirements
    * Requirements that will exist regardless of the design or implementation choices, (i.e. capacity, speed, precision, metrics. (Keep a file of design notes to assure these are heeded.))
  * Not Merely Derived Results
  * Avoid merely derived results, ("client's age" in a system that stores date of birth). Temporary files or results are design issues.

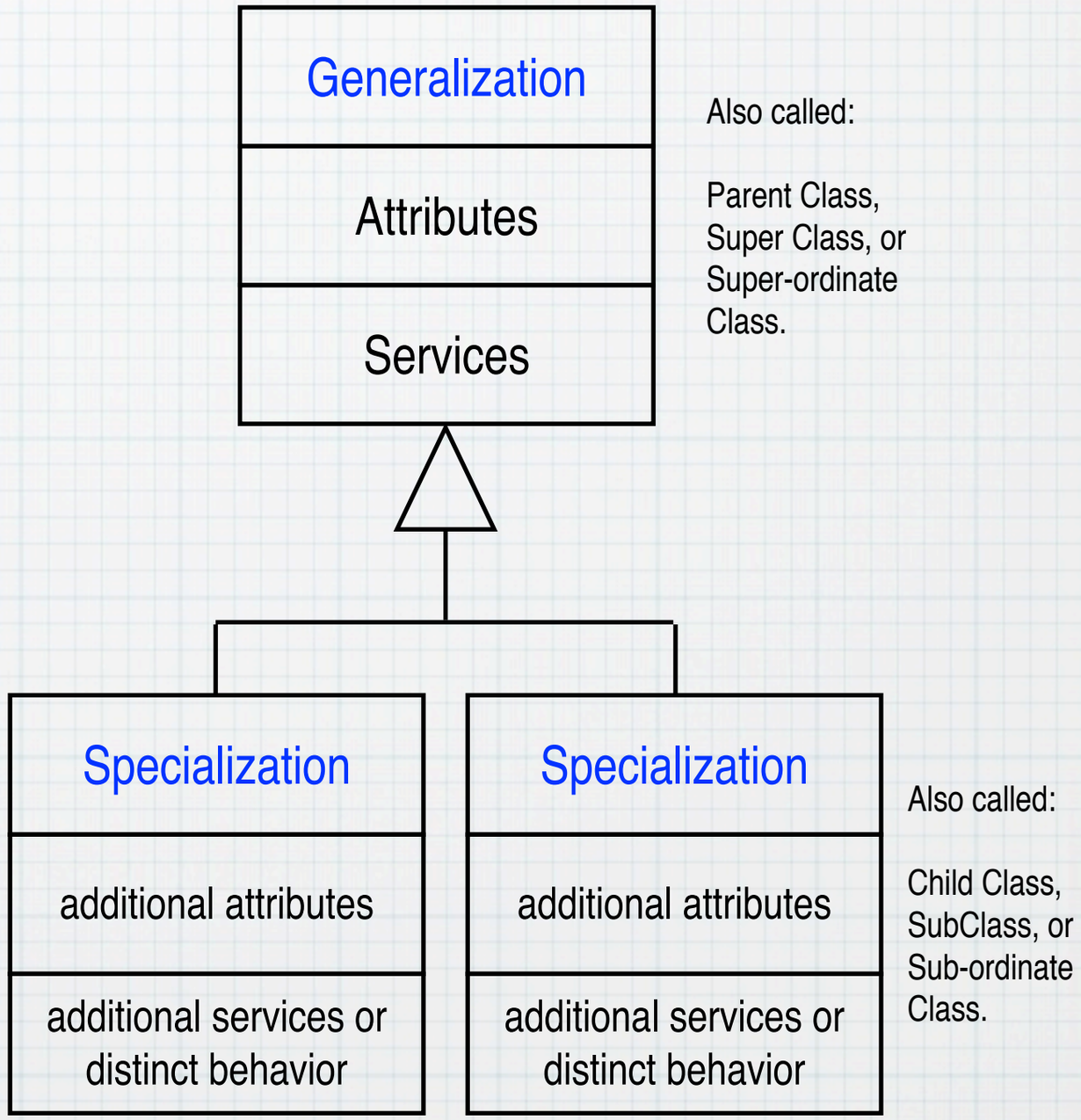
# Identifying Class Structures "Inheritance"

* Structure: A manner of organization. [Webster's 1977]
* Structure is an expression of problem-domain complexity, pertinent to the system responsibilities.
  * The term "structure" is used as an overall term, describing both **Generalization-Specialization** (Gen-Spec) Structure between classes and Whole-Part Structure between objects (instances of classes).
* **Gen-Spec** is a relationship between classes and therefore Inheritance only occurs between classes (**not objects***) !!

*objects derive their characteristics from the class to which they belong, but the attributes and behavior are expressed as a result of instantiation rather than inheritance!

# Drawing Generalization / Specialization

* Classes define the structure of the objects that will be instantiated from them, they are templates
* The sameness / difference that may exist between classes is drawn to explicitly define how two classes are the same and are different.
* Their sameness is defined by the structure of the generalization.
* Their difference is explicit in the distinctive structure of the specialization.

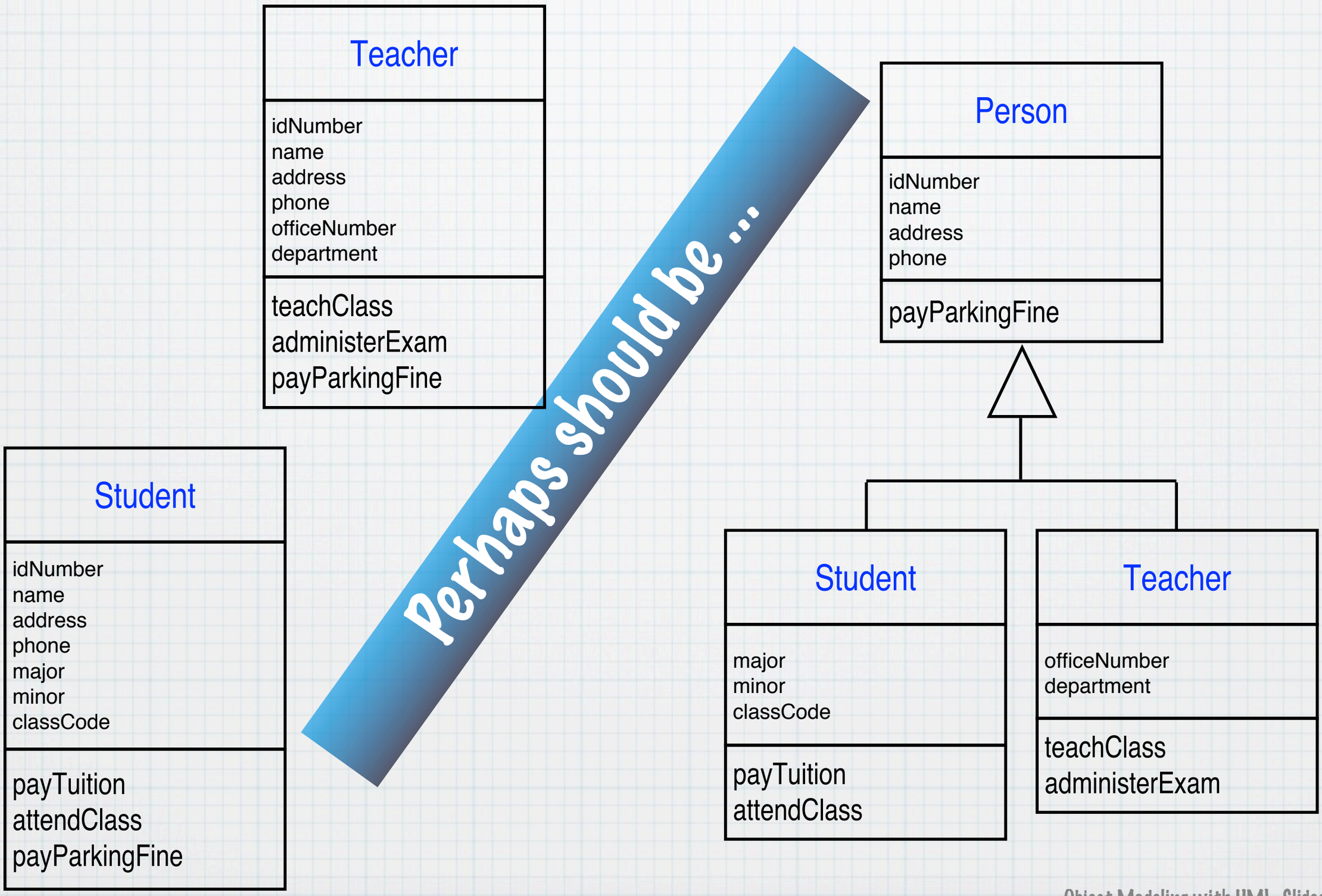


# Gen-Spec & Inheritance

- Gen-Spec is a structural relationship between CLASSES
- Gen-Spec defines the sameness of the child class with the parent class
  - everything the parent class can remember (attributes), so can the child
    - not the values of attributes - only the structure (values are in objects!)
  - every NAMED behavior of the parent class is available from the child
- Gen-Spec defines how the child is explicitly different
  - the child may have additional attributes not found in the parent class
  - the child may have additional behaviors (services) not in the parent
  - the child may implement a behavior NAMED in the parent differently
    - same Service Name ( same name and same parameters)
    - different WAY of implementing the behavior
      - also known as OVERRIDING or OVERLOADING a parent's service

We're talking class here not object !

# Gen-Spec Structure

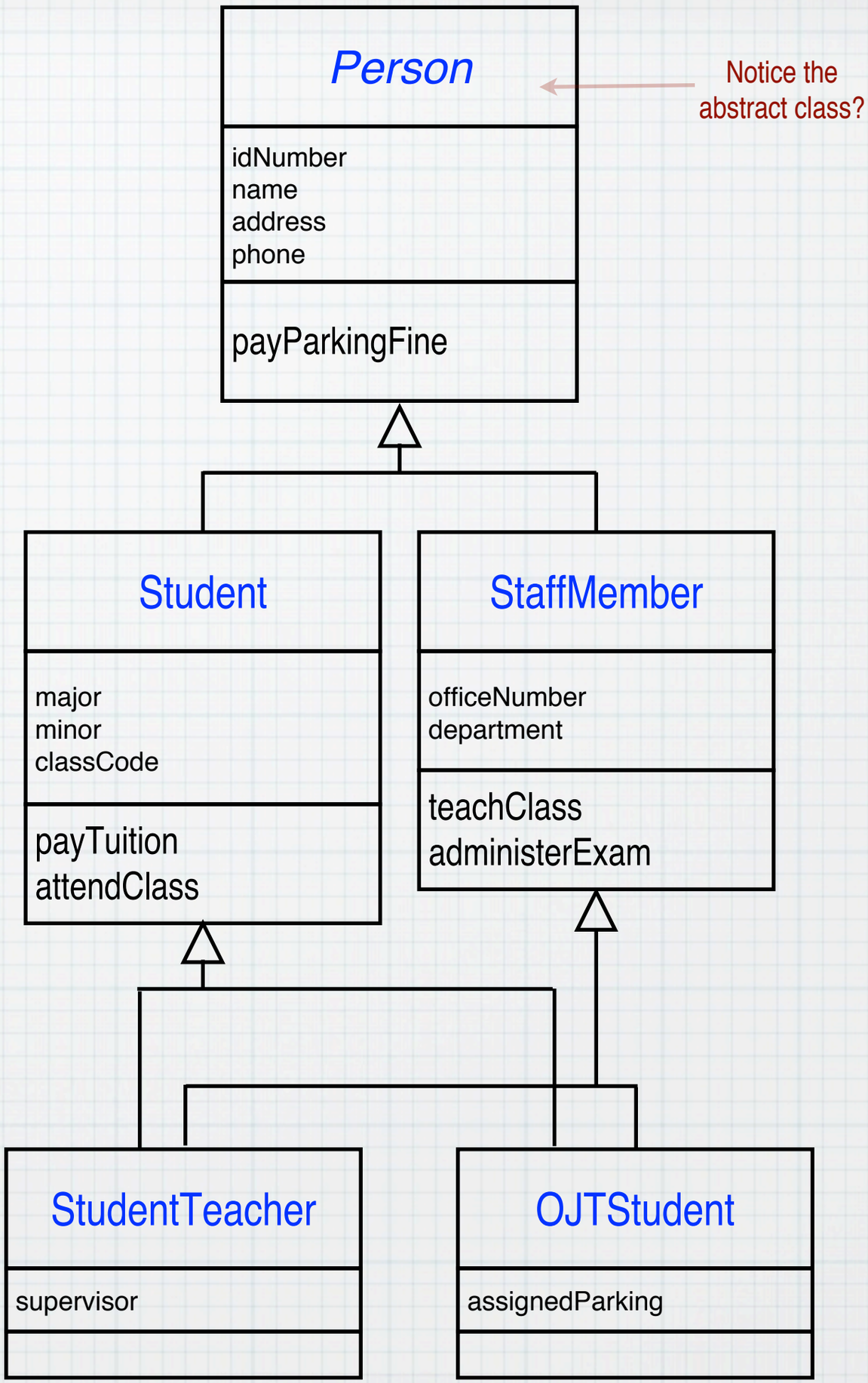


# Gen-Spec Strategies

* Consider each class as a generalization. For its potential specializations ask:
  * Is it in the problem domain?
  * Is it within the system 's responsibilities?
  * Will there be inheritance?
  * Will the specialization meet the "What to consider and challenge" criteria for Class and Objects?
* Consider each class as a specialization. For its potential generalizations ask the same questions!

# Hierarchy vs. Lattice

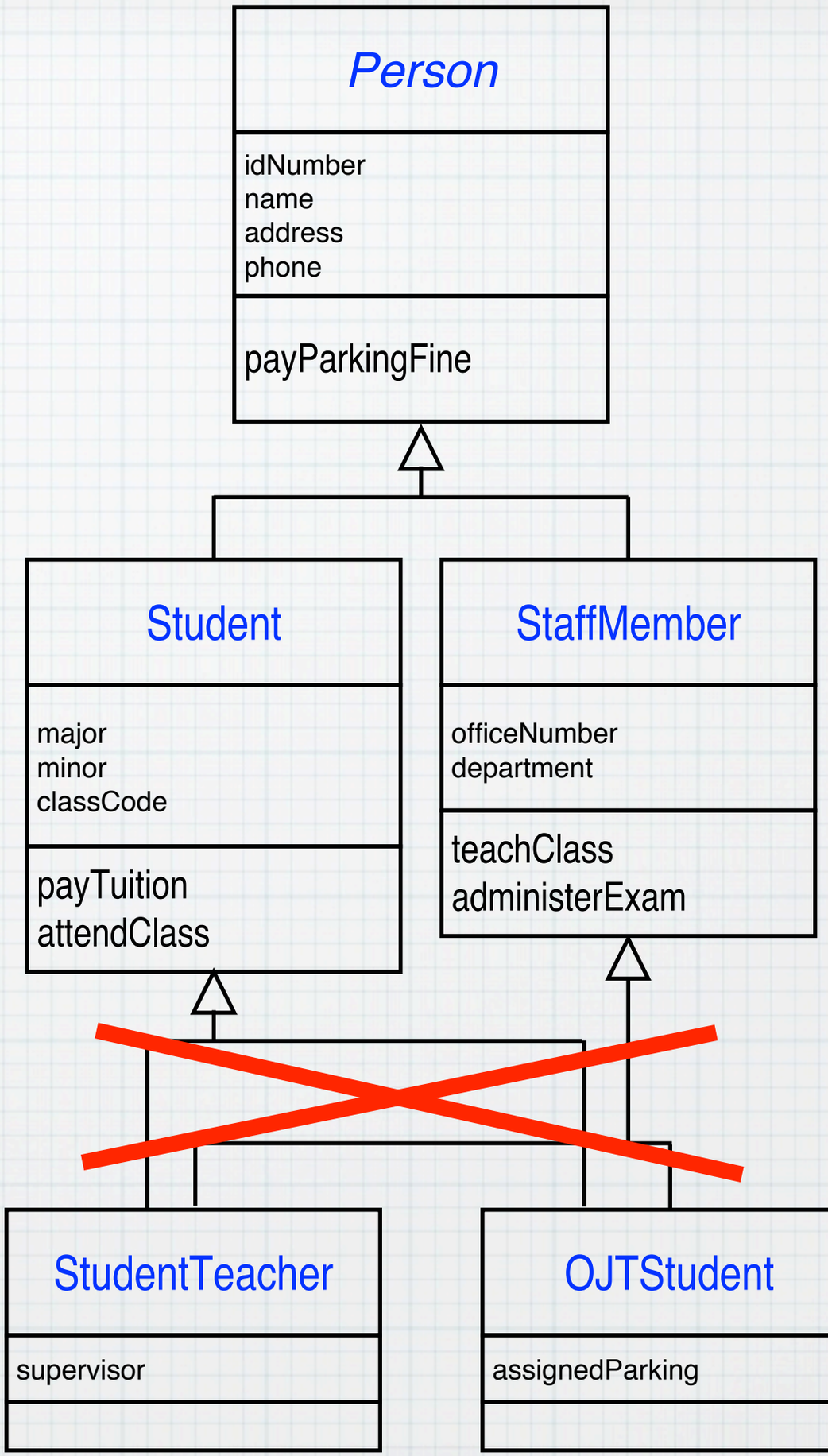
* The most common form of gen-spec is hierarchy.
* Lattice may be used to:
  * highlight additional specs
  * explicitly capture commonality
  * modestly increase model complexity



# Avoid Multiple Inheritance!

Waguespackism!

* Multiple inheritance makes further model evolution difficult
* Most apparent need for multiple inheritance is better handled using "role models" which are separate objects carrying the shared functionality
* Most programming languages handle multiple inheritance very awkwardly
* Avoid it at all costs !!

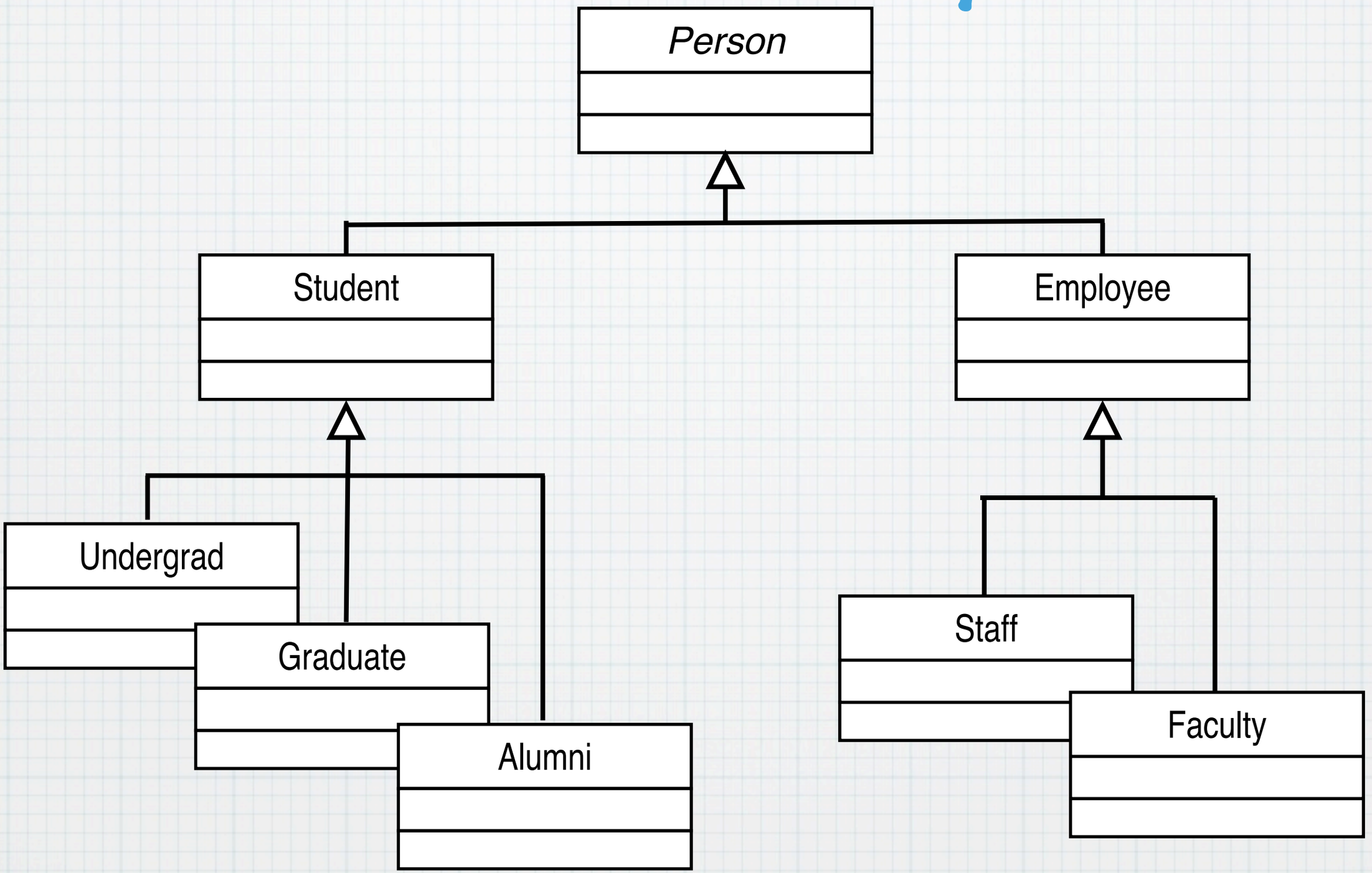


# Inheritance and Polymorphism

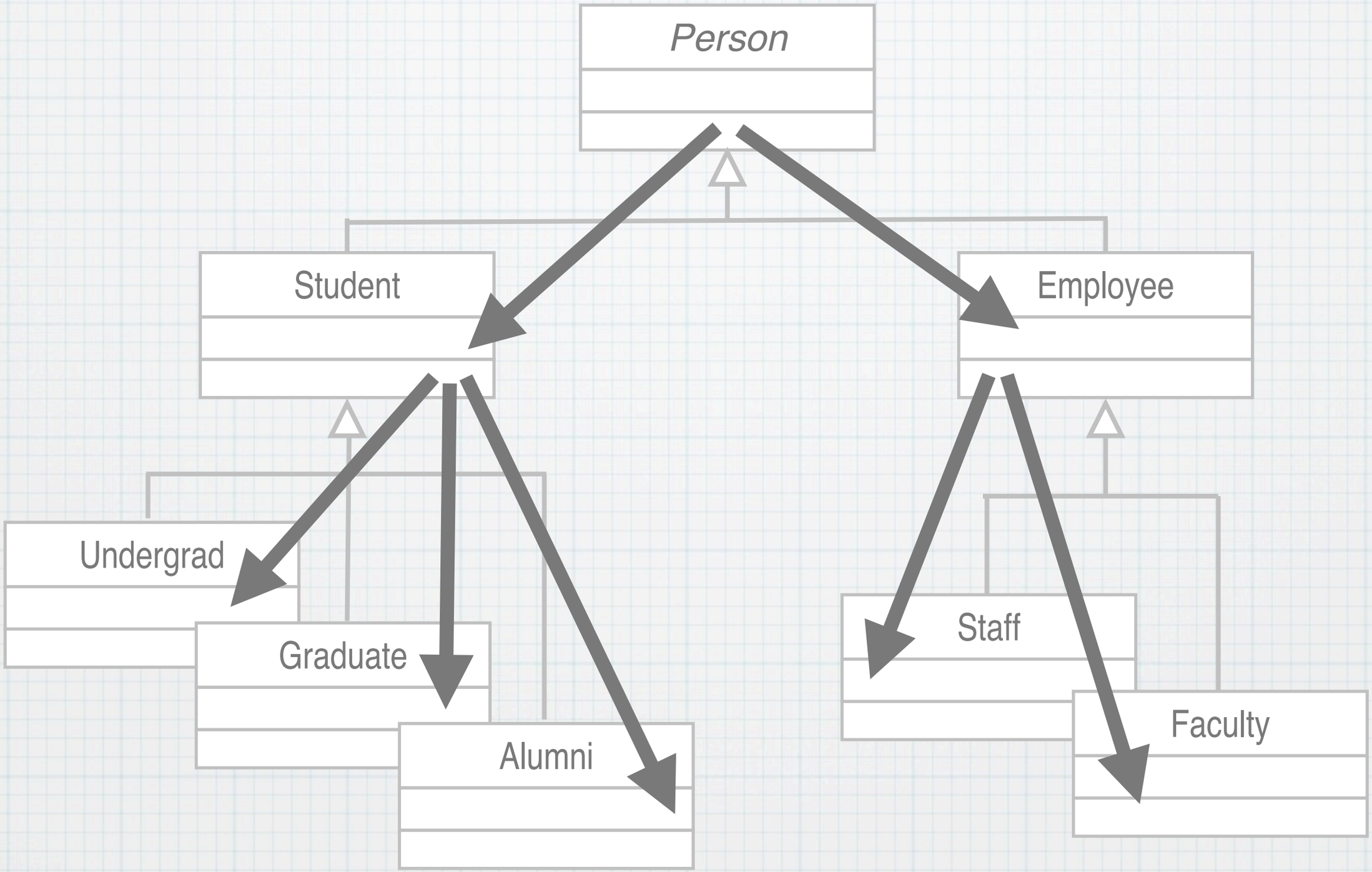
- "the child may implement a behavior NAMED in the parent differently"
  - same Service Name ( same name and same parameters)
  - different WAY of implementing the behavior
    - also known as OVERRIDING or OVERLOADING a parent's service
- Naming a service in a parent class sets a precedent
  - if the implementation is omitted we call this an "*abstract service*"
  - every child-class must (somehow) implement that NAMED behavior
  - each child-class may use a different implementation
  - the "abstract service" (method) leads to **POLYMORPHISM**
    - the same named service implemented differently in different classes
    - clients of this service use it in the abstract ignoring any difference in implementation

We're still talking class here not object !

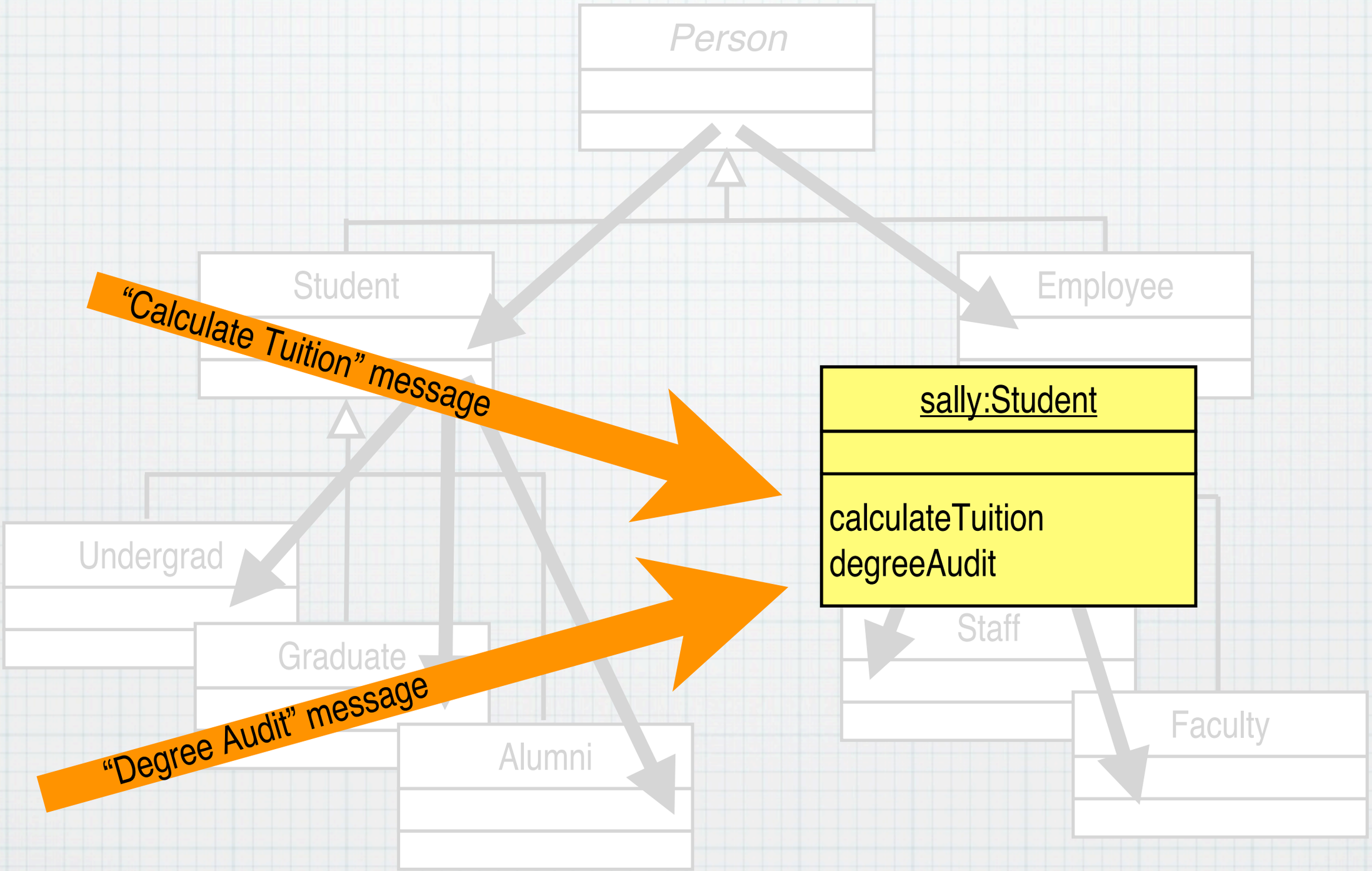
# Class Hierarchy



# Inheritance



# Polymorphism



# Identifying Object Structures

- **Associations** are relationships between OBJECTS
- Associations define the awareness that one object has for another
- Associations are defined by the strength of a relationship
  - **Composition** - the parts' existence depends on the whole
    - delete the whole and you must delete all the parts
  - **Aggregation** - the whole manages a collection of parts
    - the parts exist on their own without need for the whole
  - **Instance Connection*** (simple association) - one object knows the other
    - one object knows another exists and can send it messages
- Every association requires a defined cardinality
  - one to one!, one to many!, and many to many ?!

*Instance Connection is another very useful term but not formally part of UML.

Now we're talking about objects!

# Drawing Associations

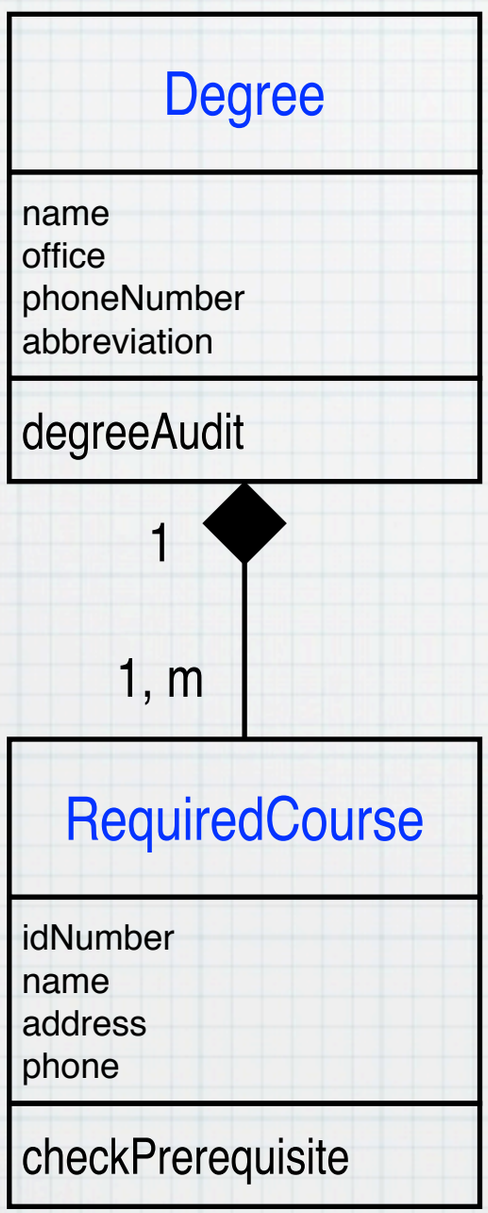
* Association is a basic method of organization in human thinking. It is helpful in identifying objects at the edge of the problem domain, and at the edges of system responsibilities. It can group together Class and Objects based upon whole-part meaning.

* The notations are directional, so that the Structure could be drawn at any angle; however, consistently placing the whole higher and the parts lower produces an easier to understand model. Note that not only may there be several parts but, they may be of different kinds as well!

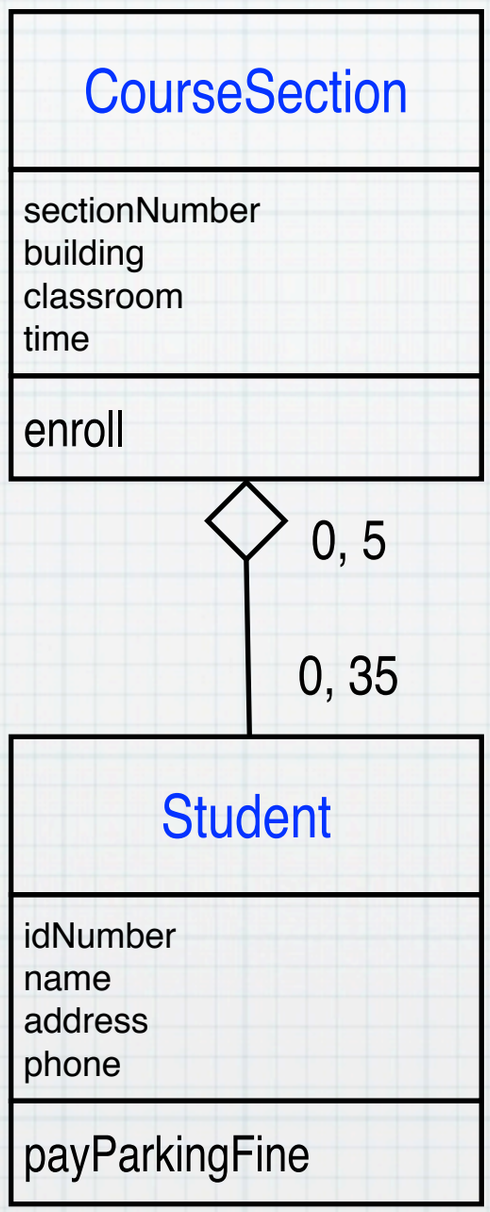
* Note that if a parent class is a whole, a part, or has an instance connection then any of its child classes are equally capable.

# Drawing Associations

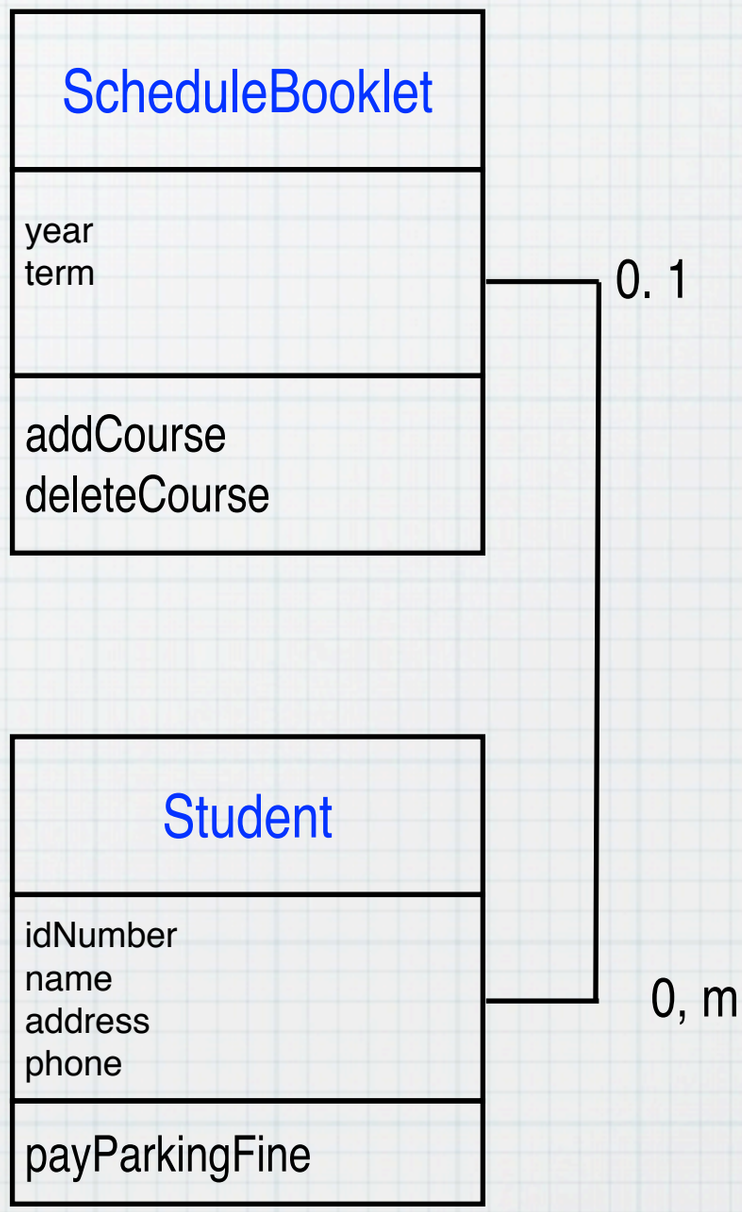
## Composition




## Aggregation

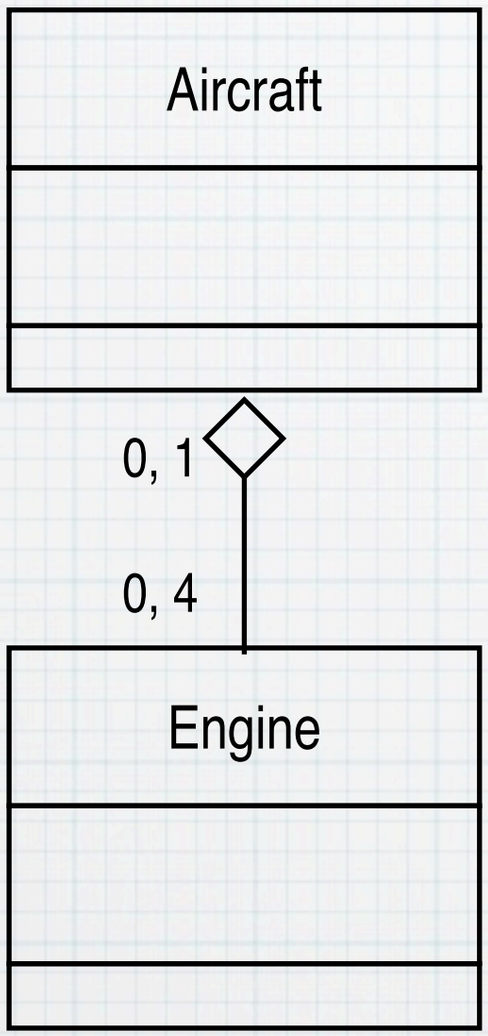



## "Instance Connection"

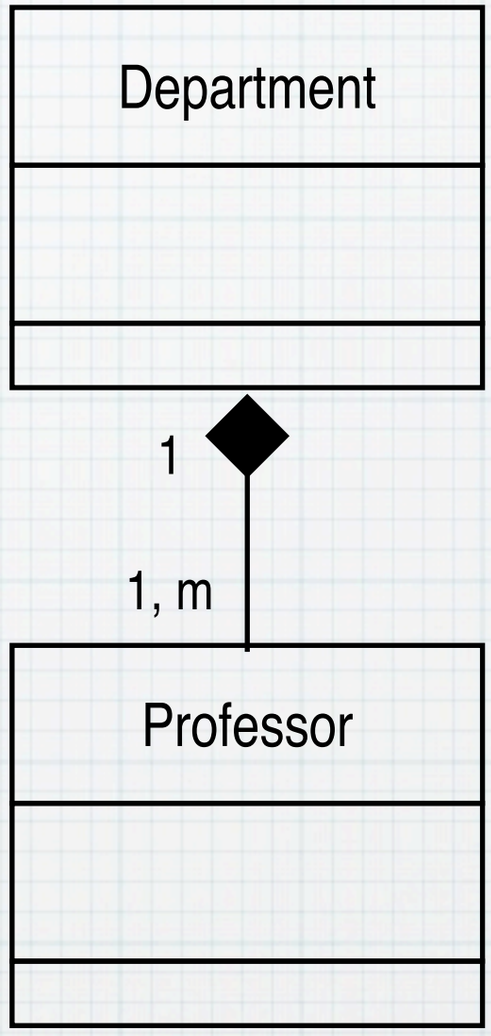



Now we're talking about objects derived from these classes!

# Whole-Part Examples




An aircraft has 0 (glider) to 4 engines. An engine belongs to zero or one aircraft (an engine cannot be shared!).



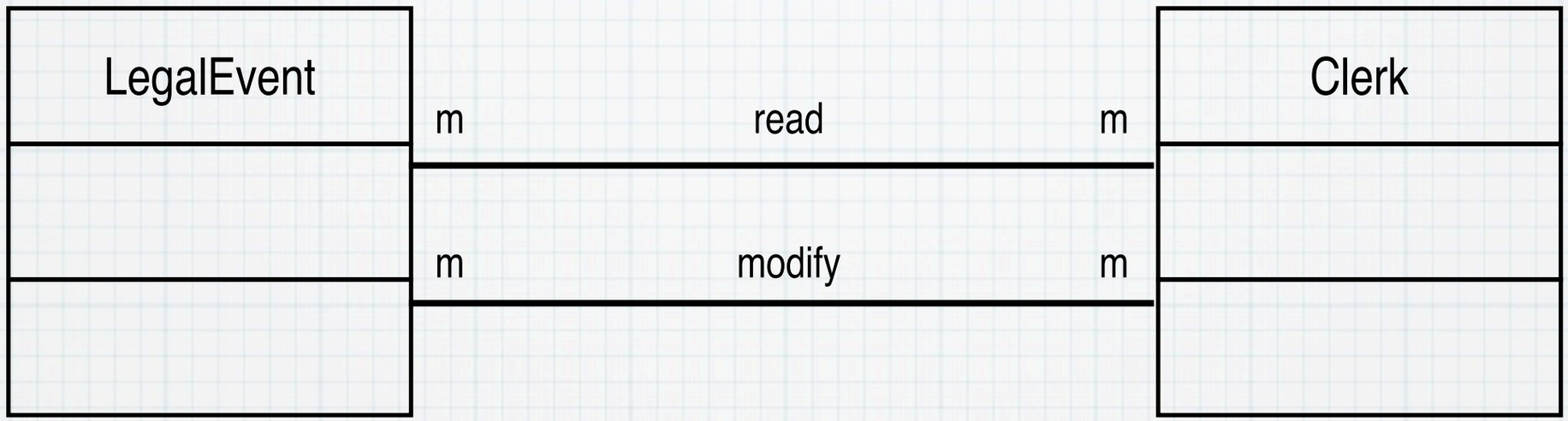

A department has 1 or more professors (must have one) and each professor belongs to a department!

Now we're talking about objects derived from these classes!

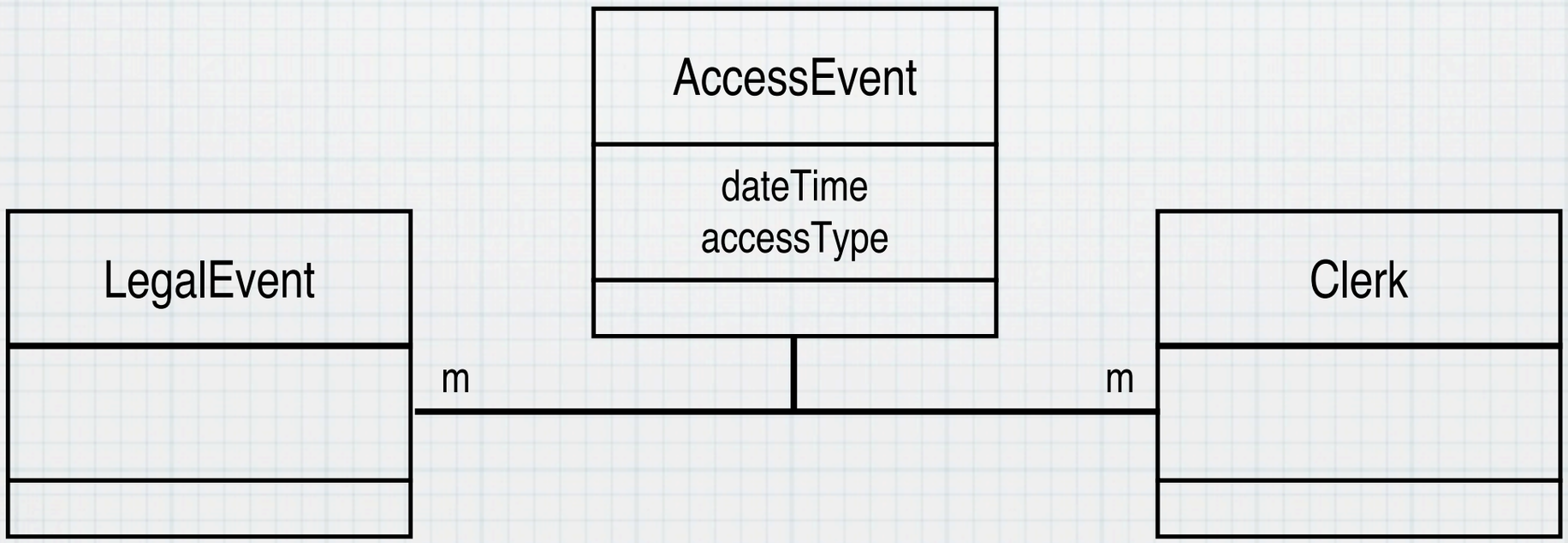
# Associations and Cardinality

* Cardinality in associations is a critical aspect of defining the business rules
* Cardinality is critically important when defining how information will be stored and later retrieved as in a database
* "**One to one**" and "**One to Many**" relationships reflect a clear and complete understanding of the business rules
* "**Many to Many**" relationships will eventually need further explanation
* In general, any "many to many" relationship will need to be converted to one or more "one to many" relationships before a model can actually be implemented in programming!

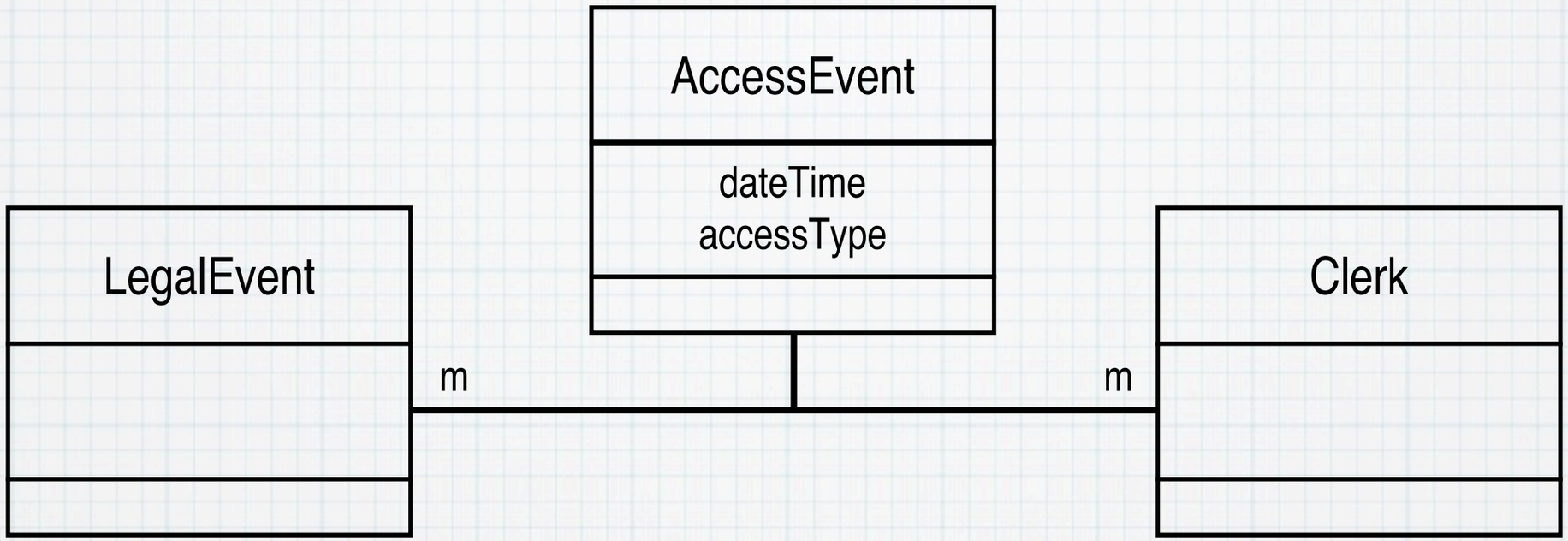
# Instance Connection Example



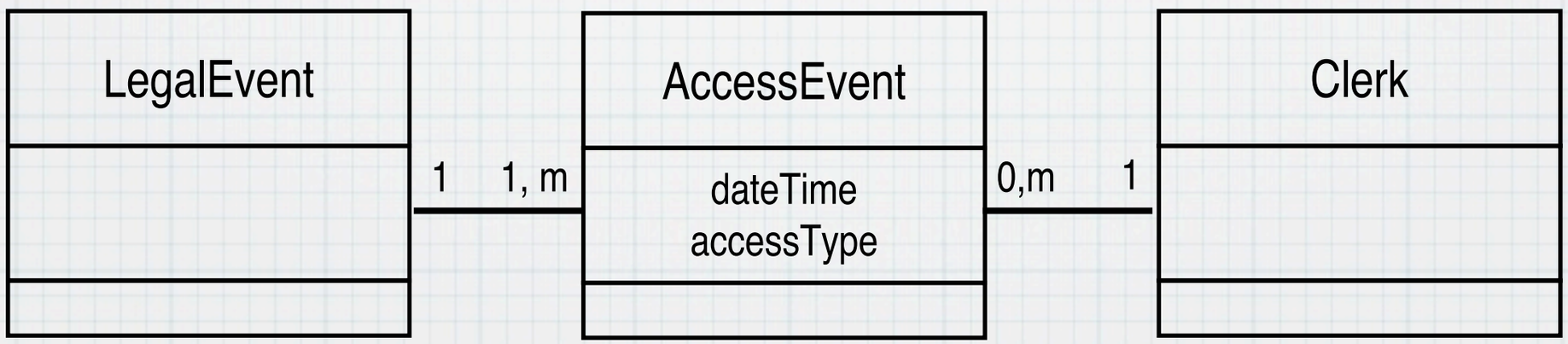

# Perhaps should be ...



# Example continued . . .




# Perhaps should be ...



# Whole-Part Strategies

- Investigating **whole-part** may point out the need for a Class and Object, perhaps one not even mentioned in the "requesting document" from the client.
- What to Look for:
  - Assembly-Parts
    - (e.g. aircraft/engines; bicycle/[handle bars, wheels, pedals], building/rooms)
  - Container-Contents
    - (e.g. aircraft/[pilot, cargo item, fuel, passenger]; safety kit/[flare, bandage, medicine]
  - Collection-Members (an varieties)
    - (e.g. class/[teacher, student]; bus route/bus stop; project plan/phase) {additional constraint: ordered collection}

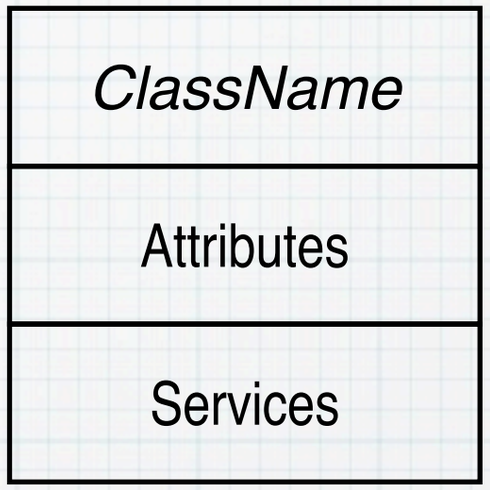
Now we're talking about composition and aggregation of objects!

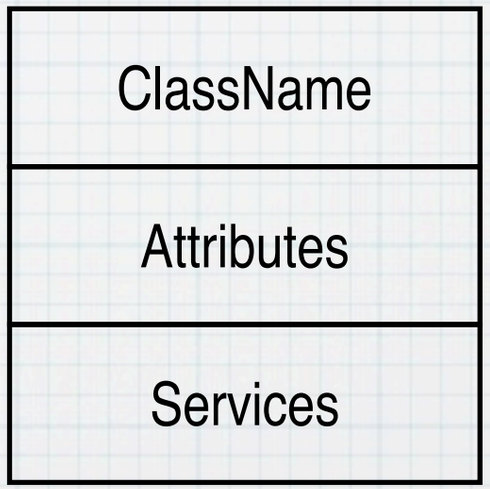
# What to consider / challenge...

* Consider each Object in the class as a whole. For its potential part(s), ask:
  * Is it in the problem domain?
  * Is it within the system's responsibilities?
  * Does it capture more than just status value?
  * If not, then just add an attribute!
  * Does it provide a useful abstraction in dealing with the problem domain?
* Consider each Object in the class as a part. For its potential whole, ask the same questions!

# Defining Attributes

* **Attribute**: any property, quality, or characteristic that can be ascribed to a person or thing. [Webster's 1977]
* An attribute is some data (state information) for which each Object in a Class has its own value.
* Attributes describe values (state) kept within an Object, to be exclusively manipulated by the Services of that Object.
* The Attributes and Services are treated as an intrinsic whole.



# Identifying Attributes

* What is the Object in a Class responsible for knowing?
* For each Object ask:
  * How am I described in general?
  * How am I described in this problem domain?
  * How am I described in the context of this system 's responsibilities?
  * What do I need to know?
  * What state information do I need to remember over time?
  * What states can I be in?

# More about Attributes

- What characteristics should attributes themselves have?
  - Attributes capture "atomic concepts."
    - The motivation for expressing an "atomic concept" is to produce a simpler model for human review with fewer attribute names, and natural data groupings for easier assimilation.
  - Defer to design - Normalization
    - Defer compromises between introducing new tables to eliminate data redundancy (normalization) and achieving acceptable performance.
  - Defer to design - Identification mechanisms
    - Defer coding schemes and artificial key design. Capture mandatory coding scheme if present.
  - Defer to design - Holding a re-calculable Attribute over time
    - Just specify the recalculation Service and decide later if the value should be stored.

# More Attribute Strategies

* Attributes (with Services) guide the definition of Classes:
  * "Not applicable?", then revisit Object's Gen-Spec.
  * Recheck each Object with only one attribute.
  * Check each attribute for repeating values.
* Instance Connections behave much like attributes
  * Do not model foreign keys needed for connections as attributes!
* Treat Instance Connections as 1-1, 1-m and m-m relations:
  * Check each many to many Instance Connection asking what Attributes might describe the connection.
  * Check each Instance Connection between Objects in the same Class.
  * Check multiple Instance Connections between Objects.
  * Check for additional needed Instance Connections.

# Defining Services

- **Service***: an activity carried on to provide people with the use of something. [Webster's 1977]
- Service: a specific behavior that an Object is responsible for exhibiting.
- Services and Attributes combine to abstract the principle of "change over time".
- The fact that Services reside in Objects abstracts the principle of "similarity of function" and "immediate causation."
- Services also provide necessary communication between Objects.
- Every "data processing " system has some PROCESSING.
- Define Services:
  - identify Object states
  - identify required Services
  - identify Message Connections
  - specify the Services

*Service is another term commonly used not formally in UML. UML would call this an "operation" sometimes a "method."

# Standard Required Services

- Algorithmically Simple Services:
  - Create an Object:
    - This Service checks the values against the constraints; then if AOK, create the new instance of the Object; then returns a result
  - Connect an Object:
    - This Service connects (disconnects) and Object with another. It establishes or breaks a mapping between Objects.
  - Access an Object:
    - This Service gets or sets the value of an Object's Attribute(s).
  - Release an Object:
    - This Service releases (disconnects and deletes) an Object.
- Algorithmically Complex Services:
  - Calculate:
    - This Service calculates the results from Attributes of the Object. Access to other Objects may be needed to complete the Service.
  - Monitor:
    - This Service monitors an external system or device; it may have asynchronous signaling responsibilities.

# Describing Services

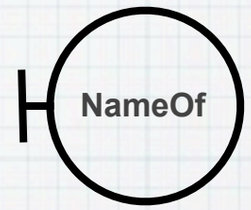
* The description of the Service may take on a variety of forms: prose, pseudo-code, flow-diagrams, decision logic, state transition, programming language syntax (C++, Java, C#, SmallTalk, etc.).
* Except where definite prescribed procedures are known focus on the "**What**" rather than on the "**How**"!
* Use a consistent verb tense and mood (present imperative).
* "Future" references are not descriptions of Service responsibility as much as extended specification of requirements not yet addressed.
* State dependent actions should fully express the state context: "Precondition", "trigger," and "terminate."

# Documentation vs. Diagrams

* The class diagram is a very useful modeling tool
  * It can be a white board "mock-up"
  * It can be the back of a bar napkin "pipe dream"
  * It can be the back of an envelope "notion to be completed later"
* A diagram is complete documentation -- NOT!!
  * Each element of a diagram requires a prose description
    * class - abstract or concrete
    * generalization - specialization
    * attribute - valid values, range constraints
    * service - prose, pseudo-code, Java, Smalltalk, C#
    * association - composition (whole-part), aggregation, instance connection
    * cardinality - required versus optional relationships
  * The prose explains how the diagram element accurately reflects the " real world " business rule that being documented

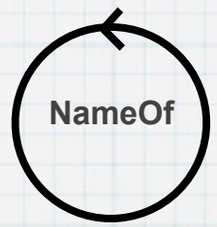
# Commonly Used Adornments

- Class symbols are often replaced by adornments that represent some aspect of their role in the model
  - **boundary** (interface)
    - provides communication with elements outside the structure being modeled



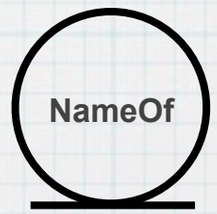
Boundary Class/Object

  - control
    - describes a class/object that implement policy applying business rules and controlling execution



Control Class/Object

  - entity
    - represents a class/object whose "remembered contents" must persist beyond a single "execution" of this structure



Entity Class/Object

  - Stereotypes
    - model elements that obey extended assumptions defined by team or problem profile can "follow a stereotype!"

**<<StereotypeName>>**

# You Need to be able to Explain:

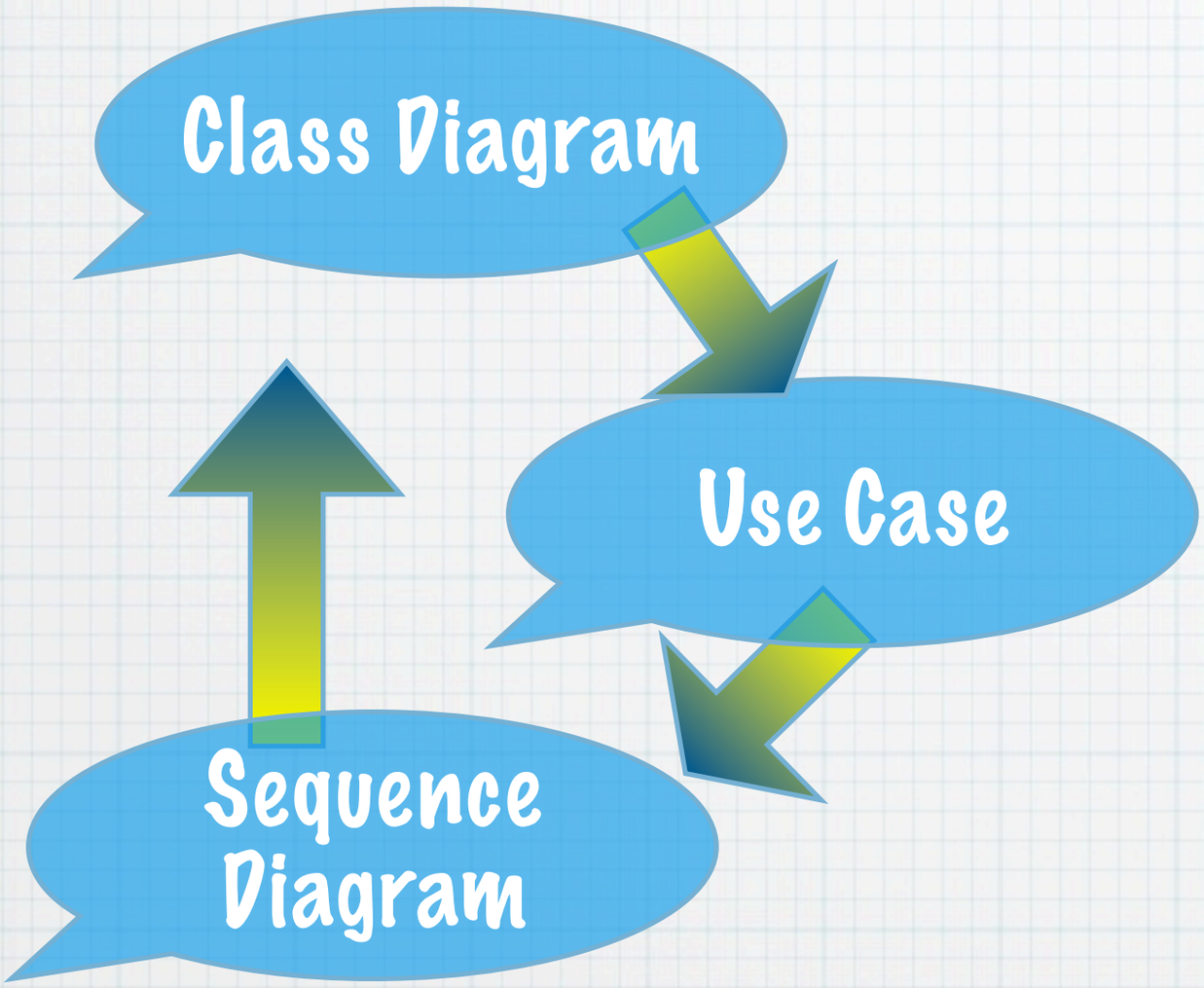
* Class
  * parent, child, super, sub, super-ordinate, subordinate
  * abstract Class
* Object
* Attribute
  * atomicity, re-calculable results
* Service
  * operation
  * method
  * abstract service
* Generalization-Specialization
  * class hierarchy
  * inheritance
  * multiple inheritance
* Polymorphism
* Association
  * whole-part
    * composition
    * aggregation
  * instance connection
  * cardinality

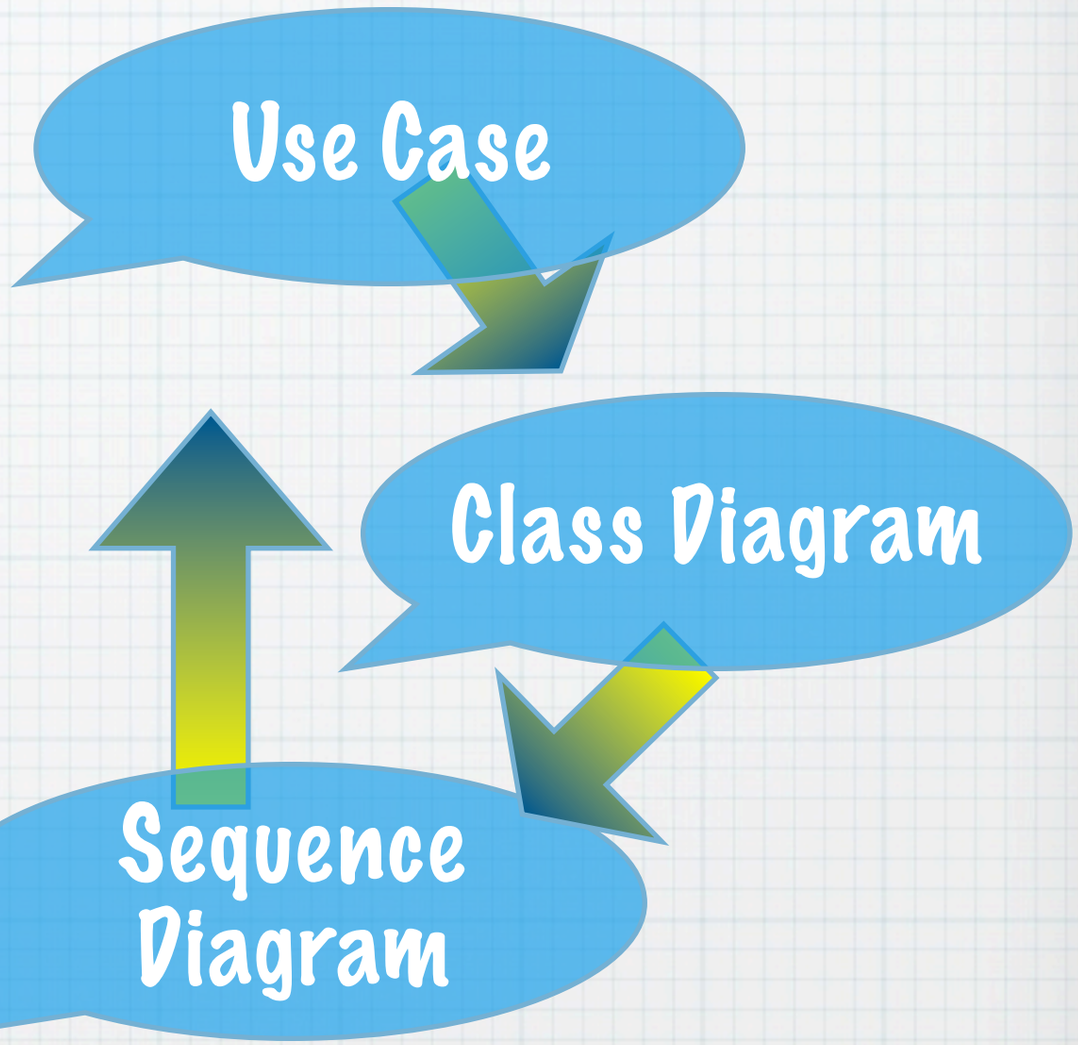
# 2.2 Use Case Diagramming in UML 2

- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams (static in UML 1)
    - **Class** - business objects and their structures
    - Composite Structure 2
    - Component
    - Deployment
    - Object
    - Package
  - Dynamic diagrams
    - **Activity**
    - Interaction
      - **Sequence** - system actions that complete a task
      - Communication (collaboration in UML 1)
      - Interaction Overview 2
      - Timing 2
    - **Use Case** - user / system interactions / interfaces
    - State Machine

2 new in UML2

# Model - Go - 'Round




* Modeling is an iterative process
  * prototype
  * refine
  * validate

# Requirements Engineering

* "Requirements tell us WHAT is happening in the problem space, but not necessarily HOW it is happening!"
  * Functional Requirements
    * ::= what behavior the system should demonstrate
      * e.g. accept payment, issue receipts, record inventory changes
  * Non-Functional Requirements
    * ::= a specific property or constraint on the system
      * e.g. web-based interface, email receipts, handle at least 5,000 catalog entries
* Requirements Activities
  * Eliciting
  * Documenting
  * Maintaining

# Requirements Elicitation

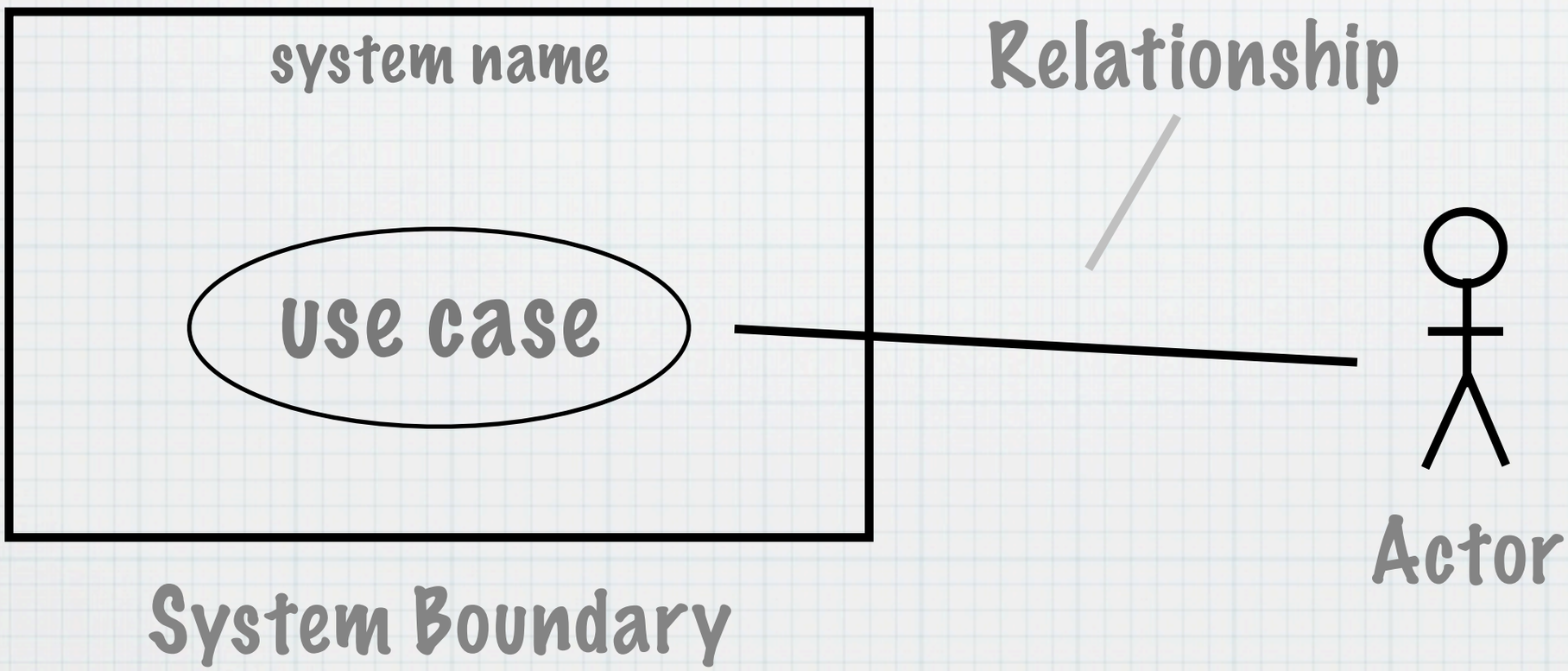
* A complete description of a system is no less complex than the real system itself
* Our task is to construct useful MODELS of real systems from which we can build computer systems that support the real system' s functions
* Our MODELS must necessarily
  * filter out some details that are not computer system relevant
  * describe the system behavior as the user experiences it
  * try to identify the rules that define what system behavior is possible and what is not ( all , everyone, always, never, nobody, none )
* Our Models must be recognizable and understandable by the users whose goals and objectives we are supporting

# Use Case Model

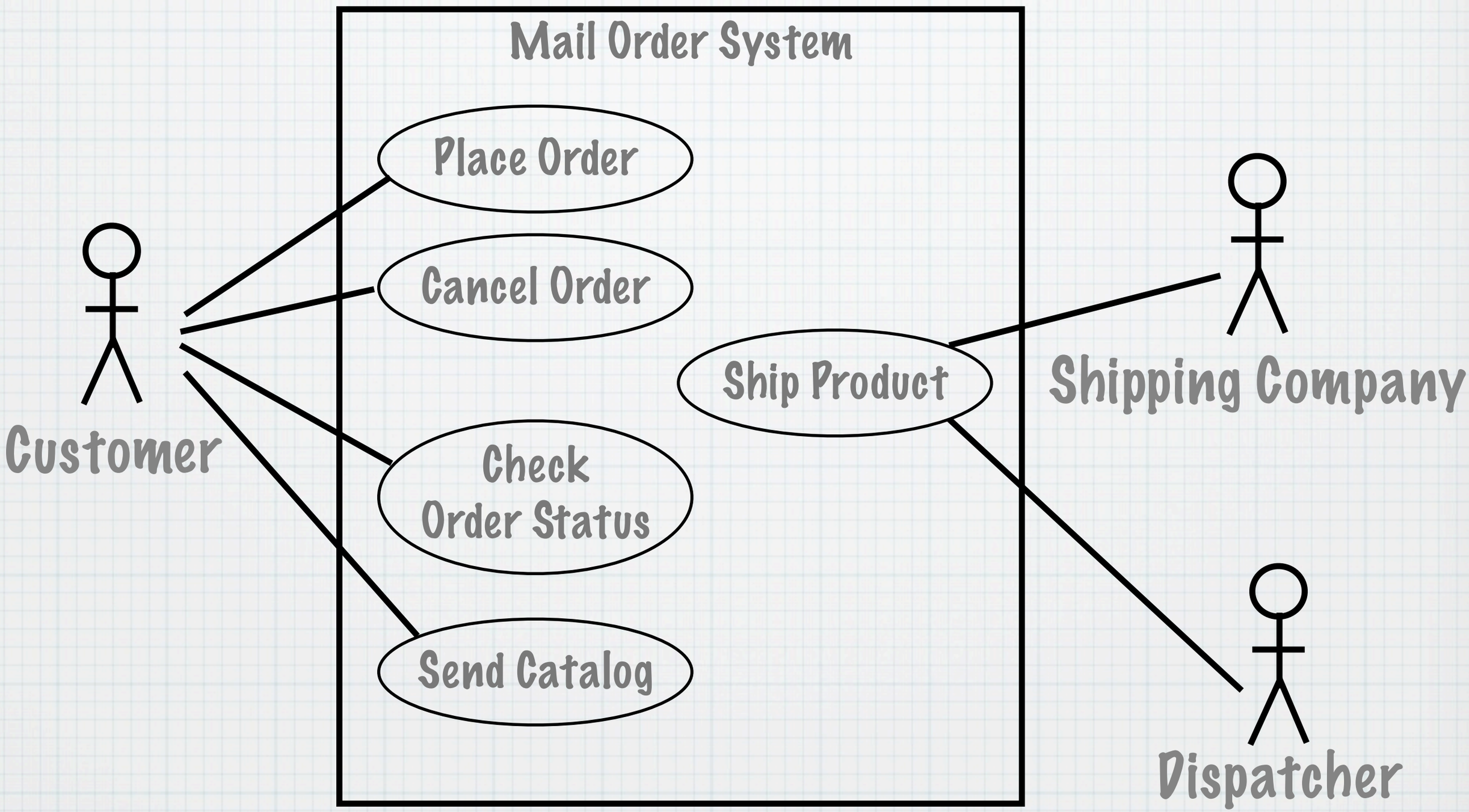
- A Use Case Model is a way of capturing requirements.
  - "A Use Case describes the interaction of some actor with the functional capabilities of the system"
- Use cases depend on three concepts -
  - System Boundary - what behavior is relevant to the modeling
  - Actors - who initiates, participates in, and/or receives the result of system behavior
  - A description of system behavior to achieve the desired results . . .
    - actions
    - their ordering
    - which are required or optional
    - what constitutes success
    - what constitutes failure
- Documenting a Use Case requires a diagram **and** specification

# Use Case Diagram

* A Use Case diagram is a short-hand depiction of the actors, system behavior, and system boundaries involved in a use case
  * Actors
  * Use Cases
  * Relationships
  * System Boundary



# Use Case Diagram Example



# Project Glossary

* Use Case activities are the front line in gathering the user terms and jargon
* Resist using terms other than those familiar to the user since users will be the final judge of model accuracy
* Capture terms that seem to be synonyms or homonyms as these will cause confusion as modeling progresses
* Creating a project glossary to collect these terms and their definitions serves not only this use case, but the entire modeling effort

# Use Case Specification

- Use Case Diagrams are good for discussion sessions and "hand waving"
- Use Case Specifications are the meat of documenting requirements - ( clear, concise, complete )
- Use Case Specification is a prose (text) description
  - use case name - a means of uniquely naming the use case
  - actor list - identifying all the persons or other systems involved
  - preconditions - what must be in place for the use case to be possible
  - flow of events - what actions occurs for the use case to succeed (or fail)
  - postconditions - what must be in place after this use case completes

# Use Case Specification Example

| Use case: Manage Basket |
| --- |
| actors: Customer |
| Preconditions:<br>1. The shopping basket is not empty<br>2. The shopping basket contents are visible |
| Flow of events:<br>1. the customer selects an item in the basket<br>2. the customer selects an action:<br>2.1 "delete item:" the system removes the item<br>2.2 "change quantity:" the system updates the quantity<br>3. the customer confirms the changes |
| Postconditions:<br>1. The basket contents have been visibly updated |

# Describing Ordered Actions

- Alternative action flows
  - "IF" and "SELECT" can be used to make choices of actions
  - "FOR each" and "WHILE" are also options for lists
- "Alternate Flow" may be more convenient than complex or nested branching in the Flow of events
  - Notice that each alternate flow may require its own postconditions

| Use case: Manage Basket |
|---|
| actors: Customer |
| Preconditions:<br>1. The shopping basket is not empty<br>2. The shopping basket contents are visible |
| Flow of events:<br>1. the customer selects an item in the basket<br>2. the customer selects an action:<br>2.1 "delete item:" the system removes the item<br>2.2 "change quantity:" the system updates the quantity<br>3. the customer confirms the changes |
| Postconditions:<br>1. The basket contents have been updated |
| Alternate Flow of events:<br>1. the customer selects "Log out" |
| Postconditions:<br>1. The basket is empty<br>2. The customer is not logged in |

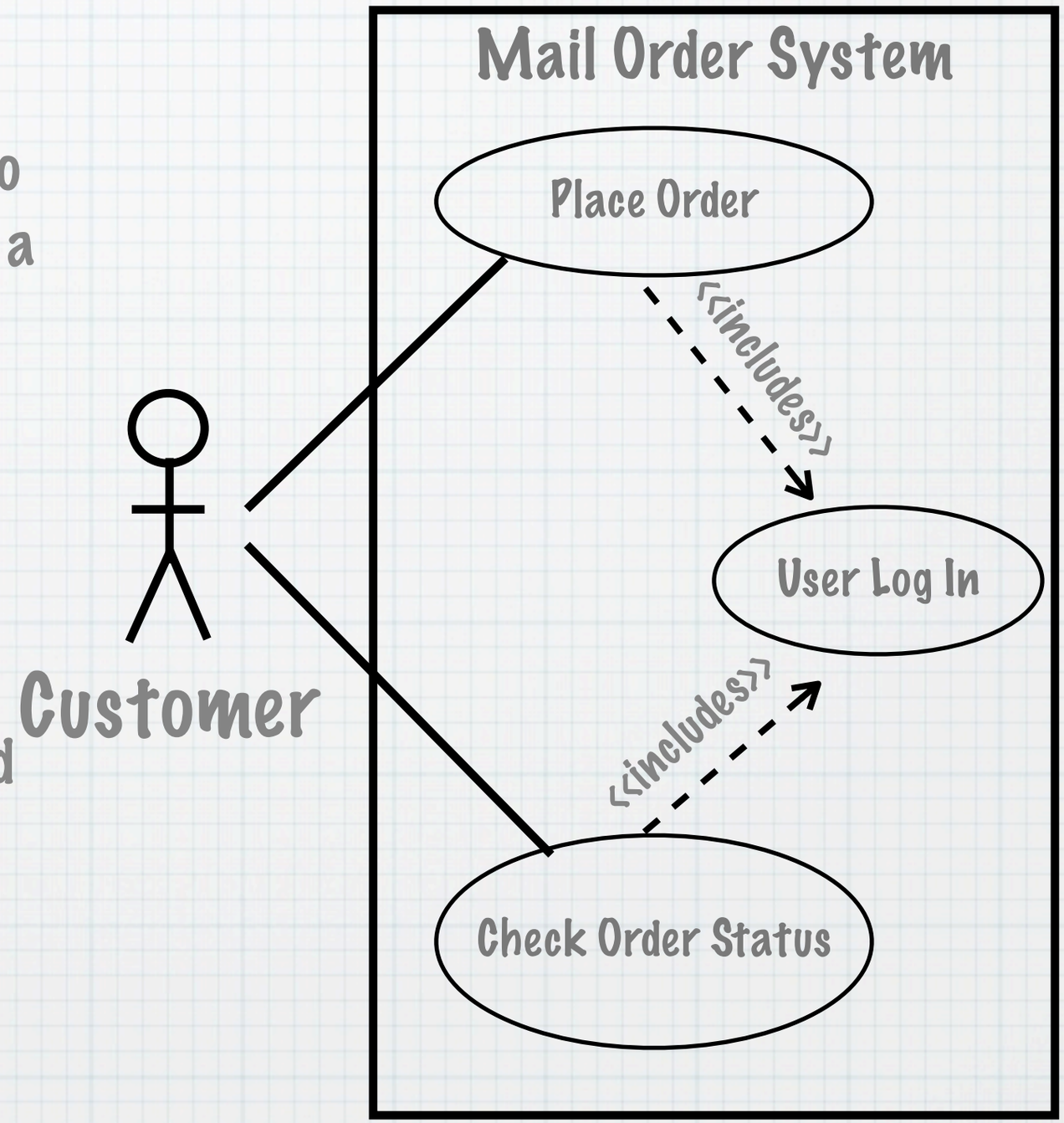
# Complex Use Cases w/ Scenarios

* A scenario is one path through a use case flow of events
* The primary scenario ("Happy Path") is the most common or usual of behaviors expected
* Secondary scenarios depict paths with the same preconditions but result of departures from the primary scenario
* If a scenario is possible, it is not an exception it is normal!

| Use case: Checkout |
| --- |
| actors: Customer |
| Preconditions: . . |
| Primary Scenario:<br>1. the customer selects "go to checkout"<br>2. system displays customer order<br>3. the customer is asked to log in with their customer id<br>4. The system displays customer details<br>5. The customer is asked to verify credit card information<br>6. The customer confirms order and payment |
| Secondary Scenarios:<br>1. The basket is empty<br>2. The customer id is not found<br>3. The credit card has expired<br>4. The customer chooses not to confirm the order and payment |
| Postconditions: . . . |

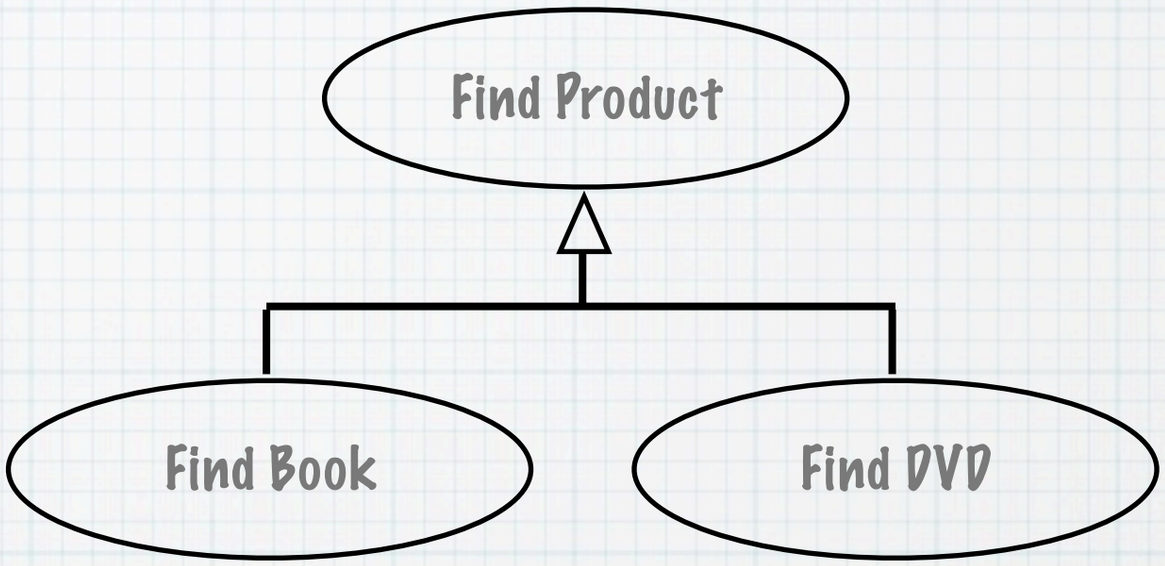
# Use Case Modularization

* A use case may be needed to accomplish the function of a more complex business activity
  * validating a user's id
  * computing sales/tax totals
* A use case may be employed like a "subroutine" within another use case with the "includes" stereotype



# Use Case Generalization

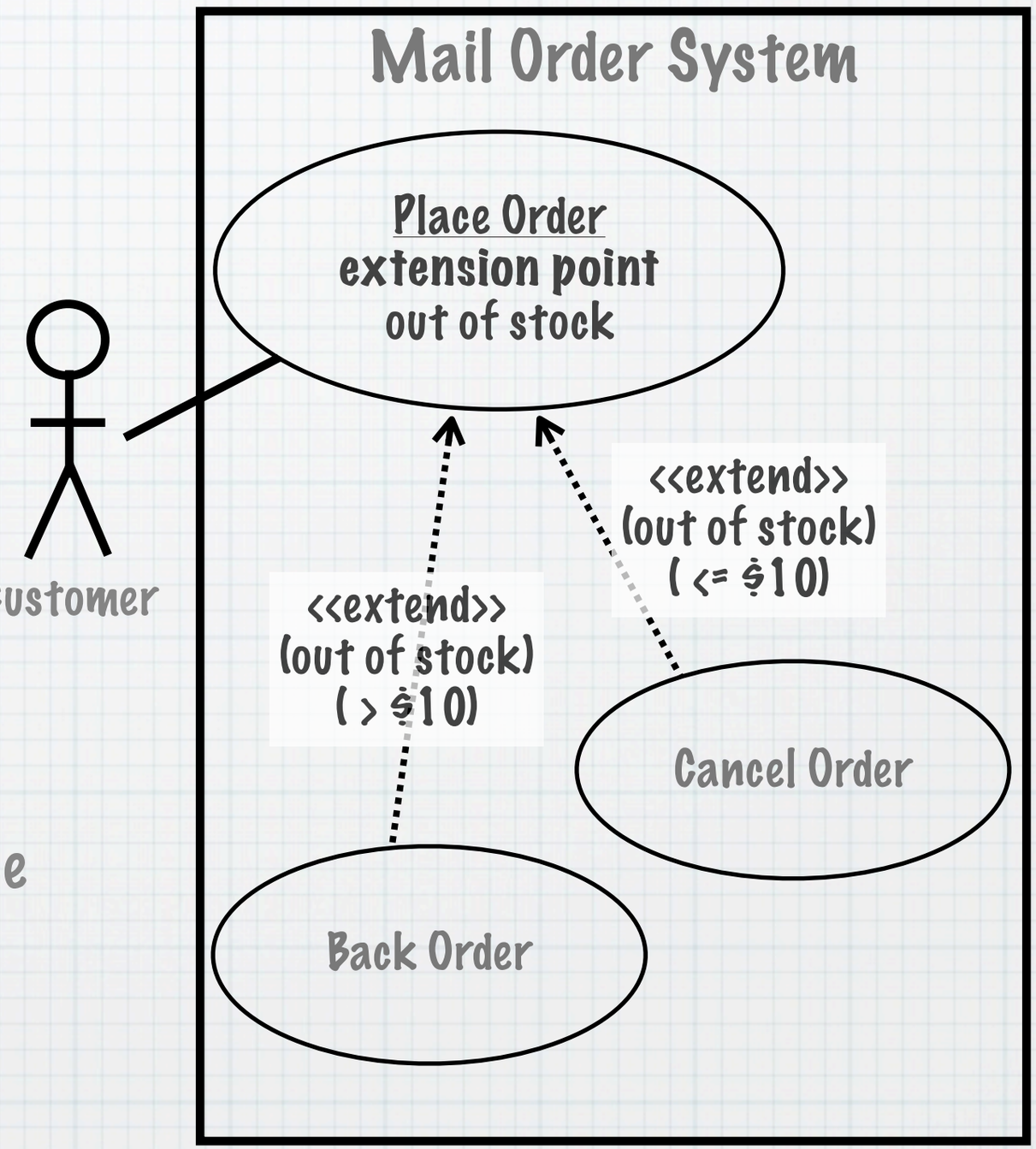
* Use Cases can be the same or different from other use cases
* Use Case Gen-Spec is a useful modeling tool
* Parent/Child use cases capture the sameness and difference that is needed in the class diagram




| Child Use Case Element | Inherits | can Add | can Override |
|---|---|---|---|
| Relationship | yes | yes | no |
| Precondition | yes | yes | yes |
| Postcondition | yes | yes | yes |
| Step in main flow | yes | yes | yes |
| Alternative Flow | yes | yes | yes |
| Attribute | yes | yes | no |
| Operation | yes | yes | yes |

# Extension Points

* An alternative to Gen-Spec for use cases is the <<extend>> stereotype
* The "Extension Use Case" describes the "added behavior" that the extension point condition requires
* A choice of extension may be denoted by "conditions" on the "extends arrow"



- Alistair Cockburn's use case template
- Variations are used to highlight particular issues in modeling this project
- Whichever template you use, clarity of expression is the key to success!

| | | |
|---|---|---|
| **USE CASE #** | < the name is the goal as a short active verb phrase> | |
| **Goal in Context** | <a longer statement<br>of the goal in context<br>if needed> | |
| **Scope & Level** | <what system is being considered black box under design><br><one of : Summary, Primary Task, Subfunction> | |
| **Preconditions** | <what we expect is already the state of the world> | |
| **Success End Condition** | <the state of the world upon successful completion> | |
| **Failed End Condition** | <the state of the world if goal abandoned> | |
| **Primary, Secondary Actors** | <a role name or description for the primary actor>.<br><other systems relied upon to accomplish use case> | |
| **Trigger** | <the action upon the system that starts the use case> | |
| **DESCRIPTION** | **Step** | **Action** |
| | 1 | <put here the steps of the scenario<br>from trigger to goal delivery,and any cleanup afte> |
| | 2 | <...> |
| | 3 | |
| **EXTENSIONS** | **Step** | **Branching Action** |
| | 1a | <condition causing branching> :<br><action or name of sub.use case> |
| **SUB-VARIATIONS** | | **Branching Action** |
| | 1 | <list of variation s> |

| | |
|---|---|
| **RELATED INFORMATION** | <Use case name> |
| **Priority:** | <how critical to your system / organization> |
| **Performance** | <the amount of time this use case should take> |
| **Frequency** | <how often it is expected to happen> |
| **Channels to actors** | <e.g. interactive, static files, database, timeouts> |
| **OPEN ISSUES** | <list of issues<br>awaiting decision<br>affecting this use case > |
| **Due Date** | <date or release needed> |
| **...any other management information...** | <...as needed> |
| **Superordinates** | <optional, name of use case(s) that includes this one> |
| **Subordinates** | <optional, depending on tools,<br>links to sub.use cases> |

# You need to be able to Explain:

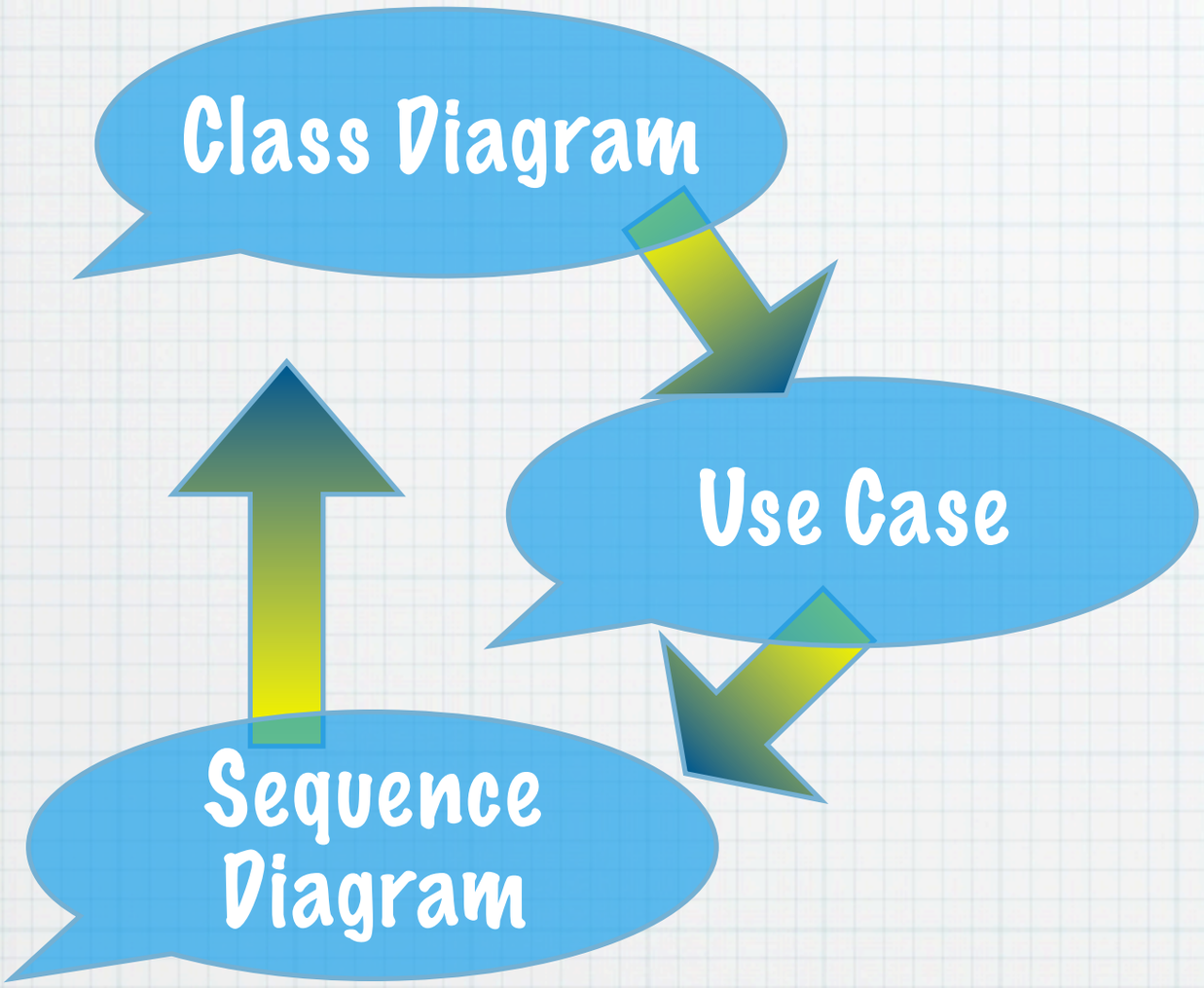
* Requirements Engineering
* Use Case
  * actor
  * system boundary
  * actor / system relationship
* Project Glossary
* Use Case Diagram
* Use Case Specification
  * preconditions
  * postconditions
  * flow of events
  * alternate flow
  * scenario
* Use Case Extension Points
  * <<includes>>
  * <<extends>>

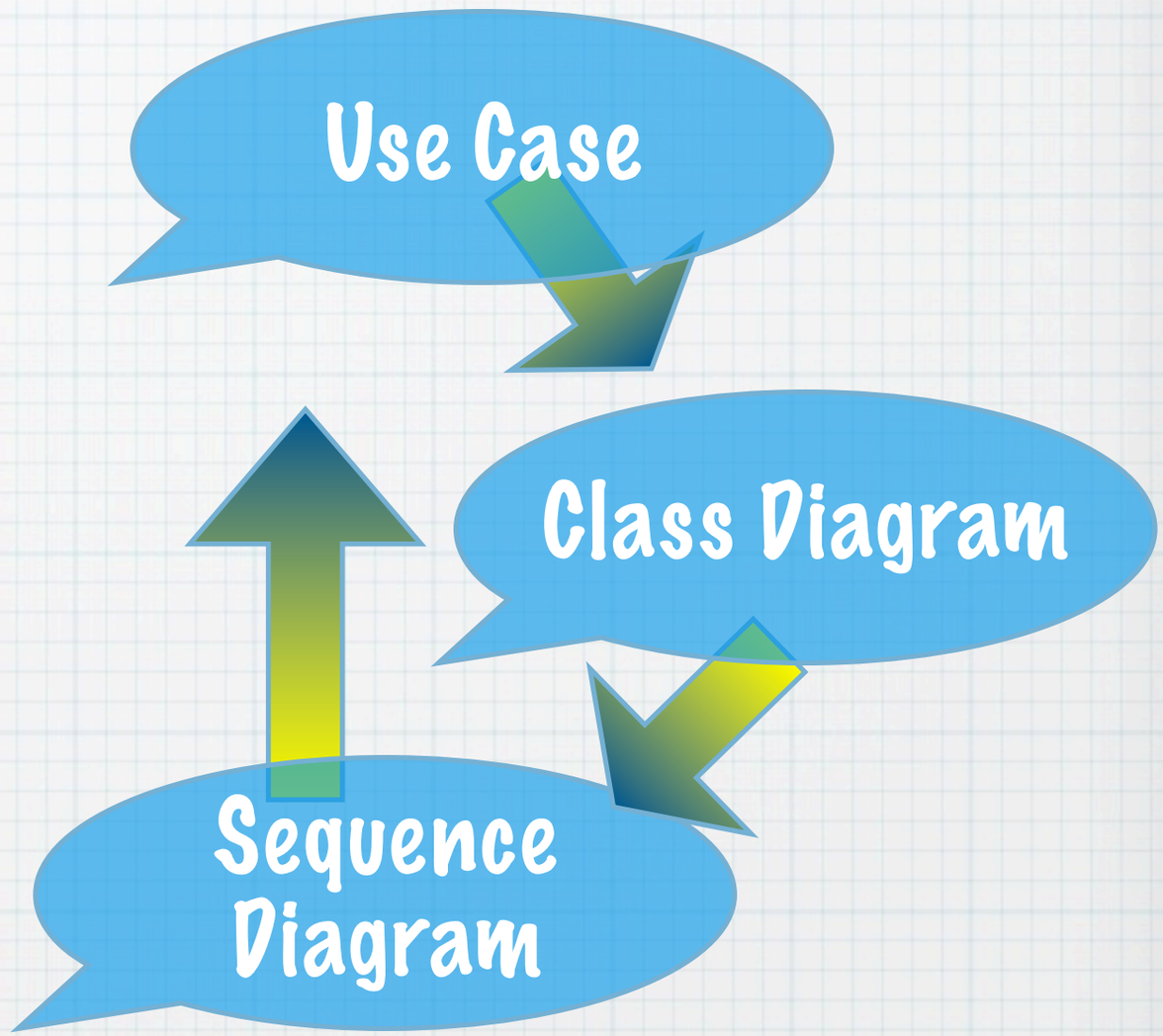
# 2.3 Sequence Diagramming in UML 2

- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams (static in UML 1)
    - **Class** - business objects and their structures
    - Composite Structure 2
    - Component
    - Deployment
    - Object
    - Package
  - Dynamic diagrams
    - **Activity**
    - Interaction
      - **Sequence** - system actions that complete a task
      - Communication (collaboration in UML 1)
      - Interaction Overview 2
      - Timing 2
    - **Use Case** - user / system interactions / interfaces
    - State Machine

2 new in UML2

# Model - Go - 'Round



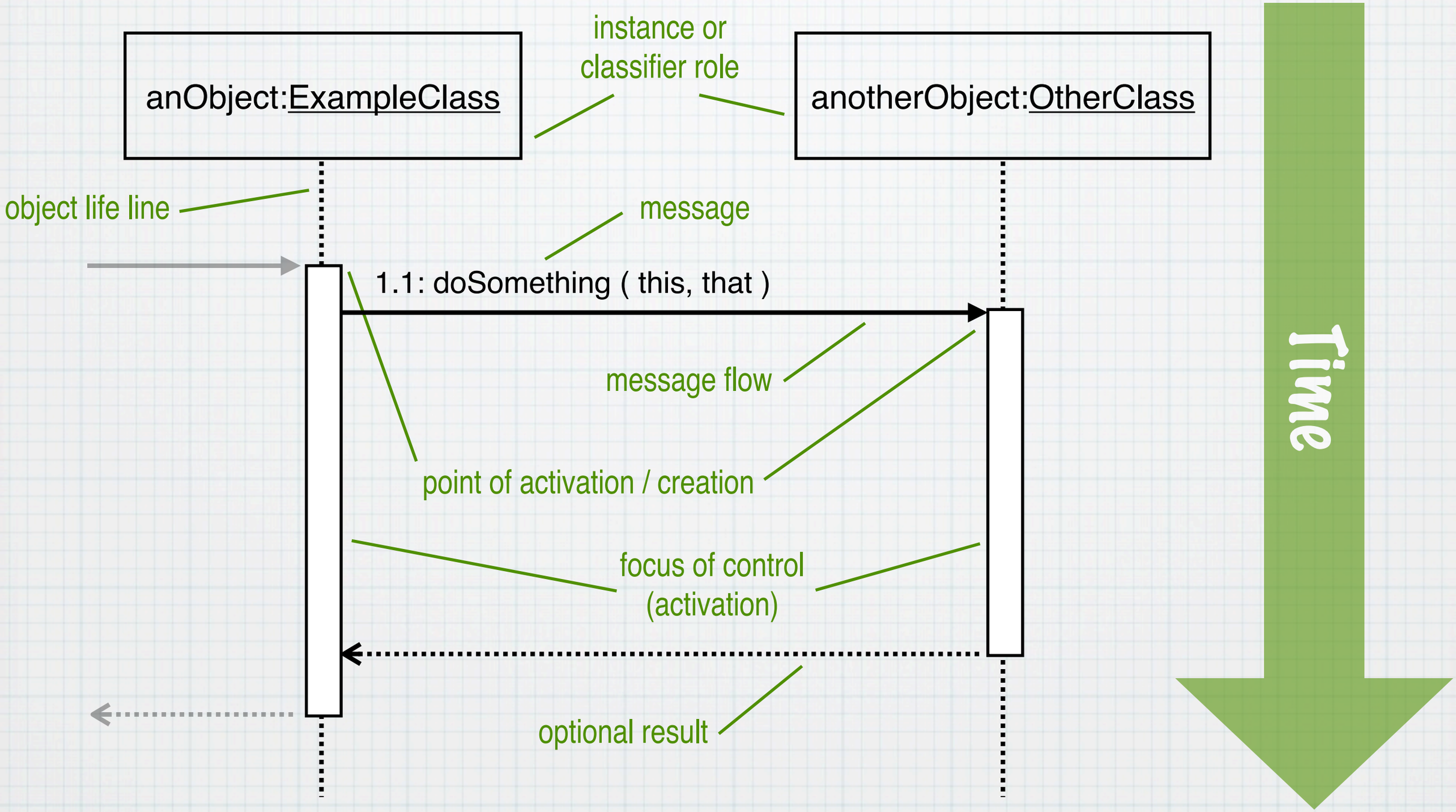




* Modeling is an iterative process
  * prototype
  * refine
  * validate

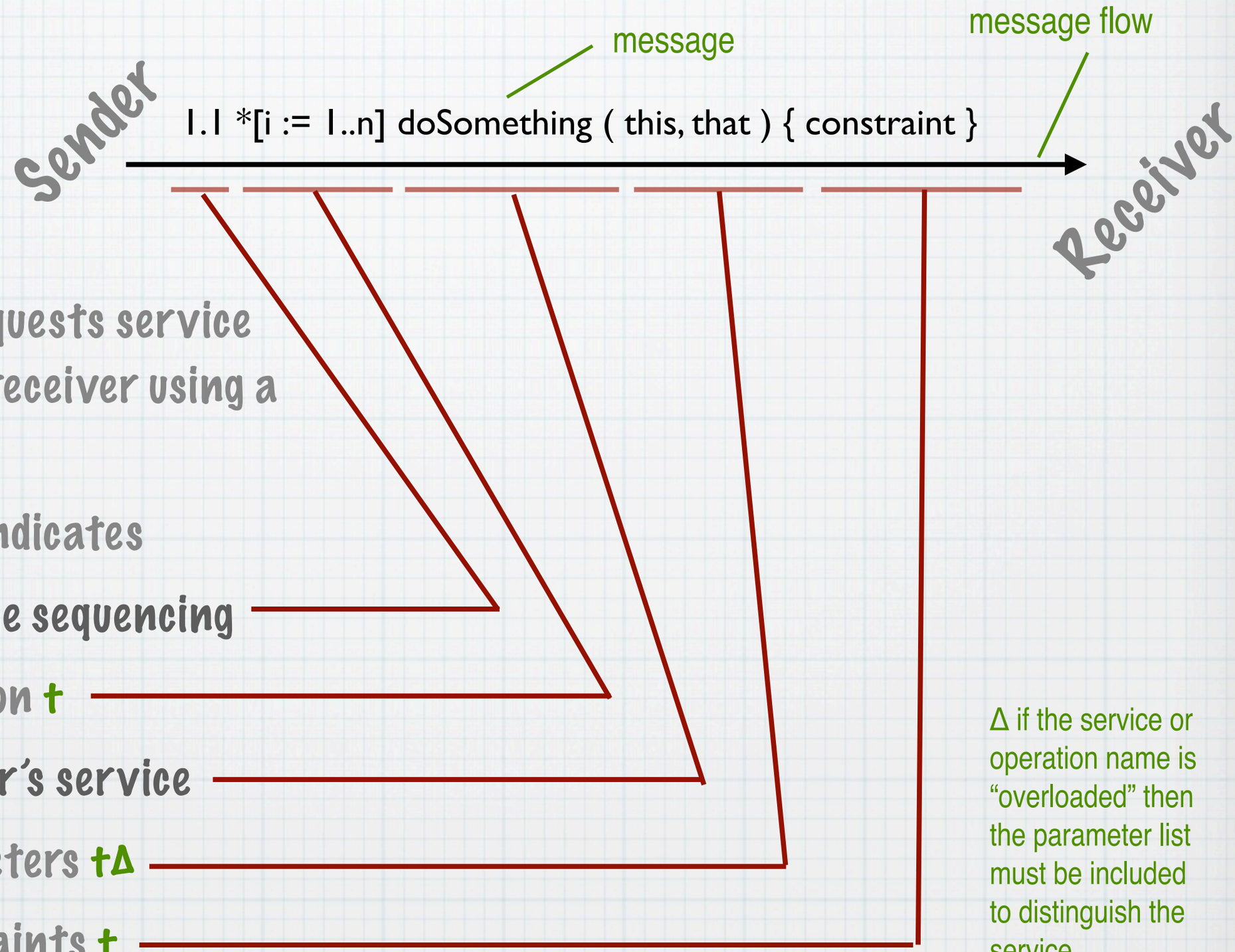
# Sequence Diagram

* Sequence Diagrams are an extended form of Communication Diagram
  * Communication Diagrams show actual objects and their relationships
  * Sequence Diagrams show the sequence of messages and events that are permitted between objects of specific classes
* Sequence Diagrams use the classes and their services defined in the Class Diagrams
* Sequence Diagrams explain the actions required to accomplish the specific responsibility of a class service including the help from objects of other classes

# Object Interaction w/ Messages



# Message Syntax




* Sender requests service from the receiver using a message
* message indicates
  * message sequencing
  * iteration †
  * receiver's service
  * parameters †Δ
  * constraints †

† optional

Δ if the service or operation name is "overloaded" then the parameter list must be included to distinguish the service addressed.

# Sequence Expression

- The sequence expression defines the ordering of messages depicted in a sequence diagram
- A series of integers separated by periods " ."
  - Each integer represents the order of messages sent from a particular activation in an object life line
  - Each time a service invokes a new activation in the receiver's life line a new integer is added to the list for any messages that that activation sends
  - The number of integers indicates the "nesting" of activations
  - The value of the integer indicates the order of the messages sent from a single activation
- Example: 1.3.2: " 2nd message of the activation caused by the 3rd message of the activation caused by the first message"

  *be patient!!!*

# Iteration Expression

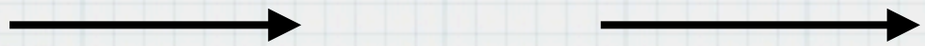
* This is an optional part of a message designation
* There is no formal syntax for iteration, but this works well for those familiar with programming language syntax
  * [ i := 1..n ] - - - iterate the message "n " times
  * [ i := 1..8 ] - - - iterate the message exactly 8 times
  * [ while ( some boolean expression ) ] - - - " do while "
  * [ until ( some boolean expression ) ] - - - " do until "
  * [ for each ( collection of objects ) ] - - - send one message for each object found in the collection

# Receiver Service / Parameters

* The service name found in the message is the name of a service provided by the receiving object
  * message takes on the "present imperative" tense
  * the sender is "commanding" the receiver to perform a service
  * Note that the message is being sent from a service currently "executing" in the activation of the sender
* Parameters are defined in the class diagram when the service is specified for a class
  * parameters may be typed or un-typed
  * parameters may be simple values derived from attributes
  * parameters may be object references derived from associations
* As a model matures parameters, typing and object references clarify and bring into focus the actual implementation details needed in design

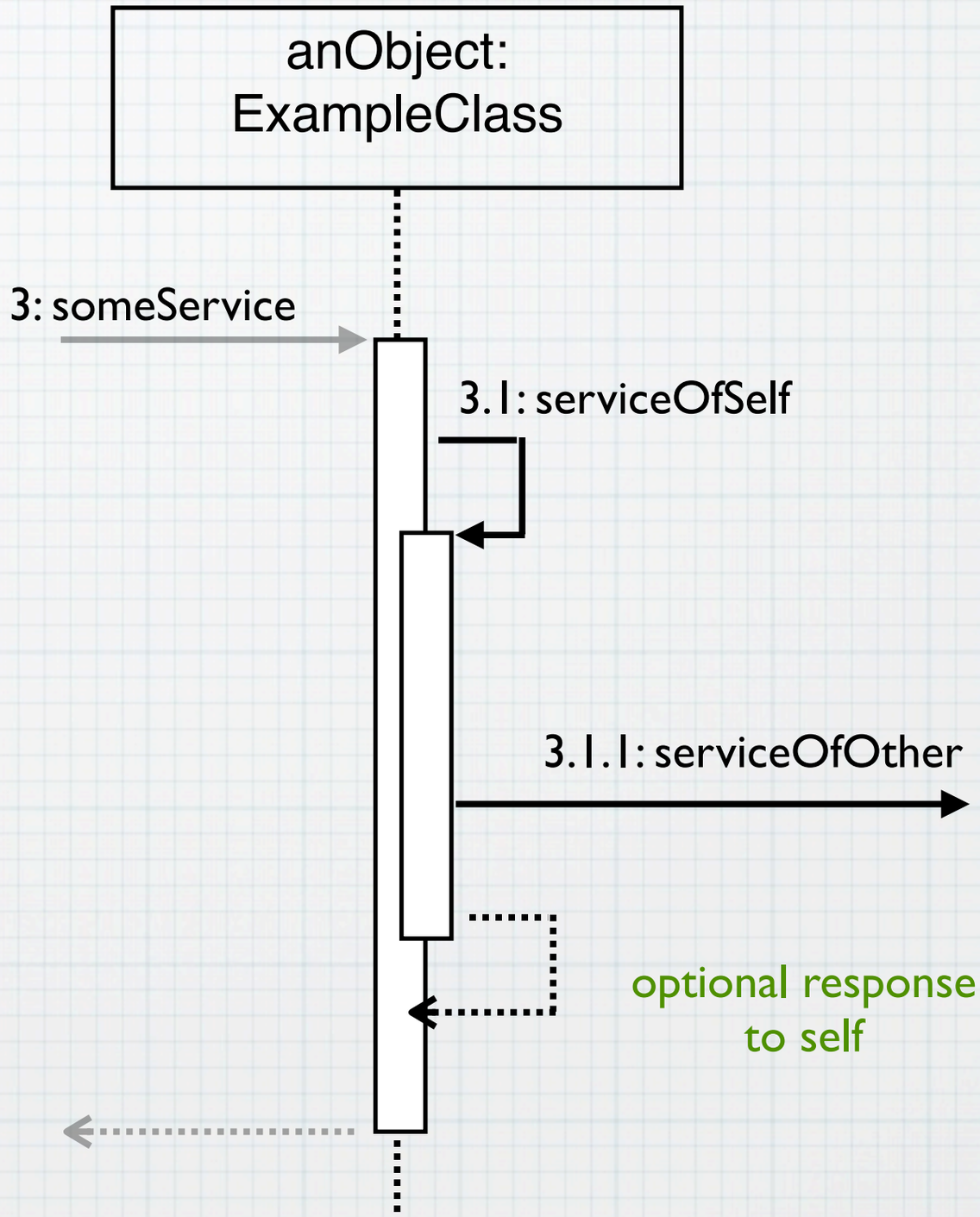
# Constraint

- This is an optional part of a message designation
- Message constraints indicate a condition or state that the receiving object must have for the message to "make sense"
- Some standard constraint values are used in UML for common situations of object creation, deletion and temporary use
  - {new} - - - an instance is created by the message interaction
  - {destroyed} - - - an instance is destroyed by the interaction
  - {transient} - - - an instance is created, but is destroyed once the activation of its service is completed
- In some instances only the constraint (as a stereotype) may be used to indicate the simple purpose of a message
  - <<create>> or <<destroy>>



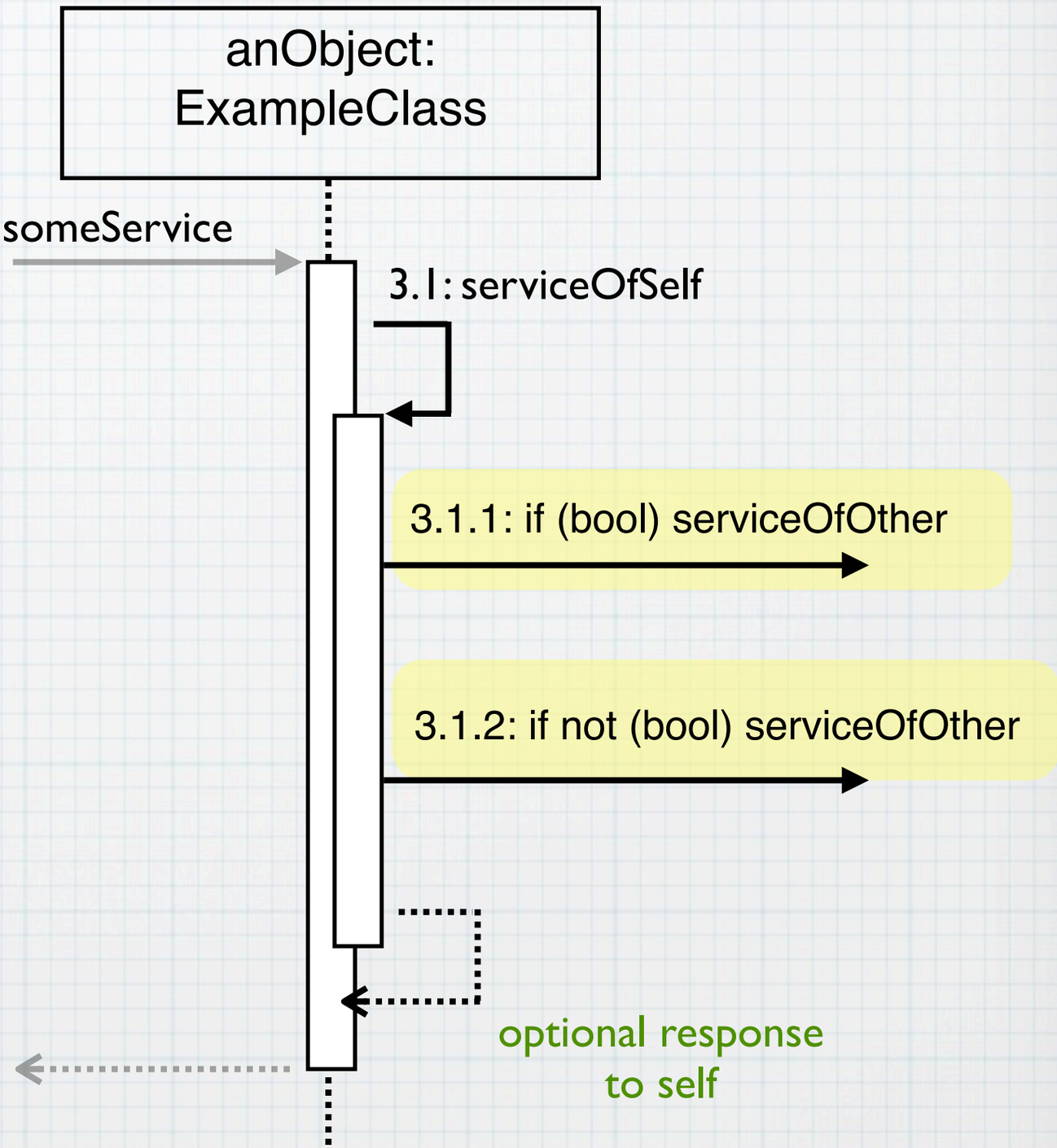
# Self-Delegaton

* Self-Delegation is the situation that the receiver of a message is also the sender of a message
* Since the message causes a new activation, the new activation symbol is "nested" on top of the sender's activation
* The response is optional
* If the new activation sends messages: add an sequence segment to the sequence expression



# Conditional Messages

* Messages may be sent on condition
  * " if (bool) send message "
* guard condition
  * [bool] send message
  * the bool must be true to allow the message to be sent

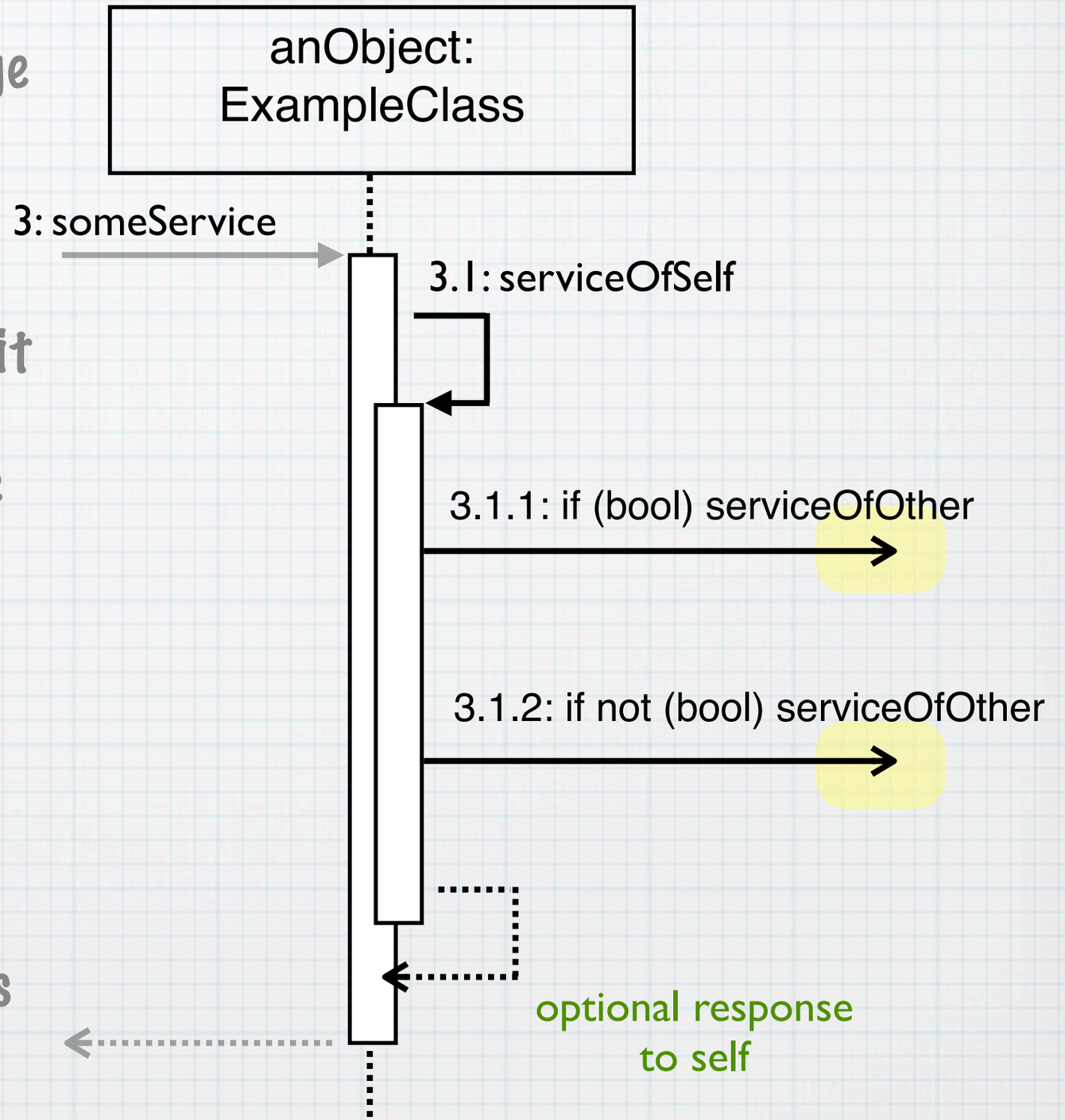


# Concurrency

* The arrow heads on message flows indicate synchronous or asynchronous messages
  * synchronous messages require the sender to wait (do nothing) until the activation caused by the message is completed
  * asynchronous messages cause an activation in the receiver but do not require the send to wait
    * the sender and receiver's activations execute concurrently

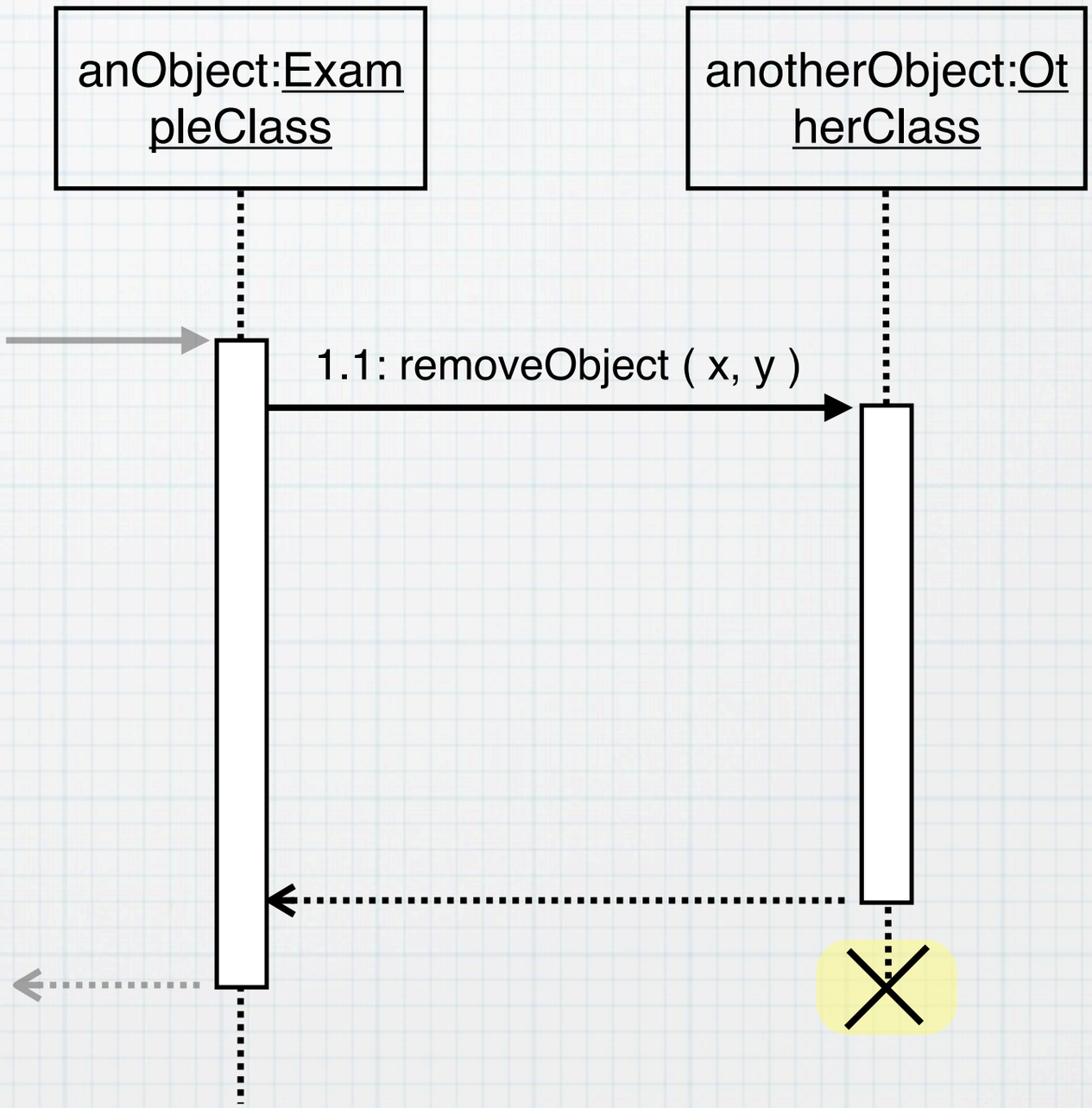
synchronous

asynchronous

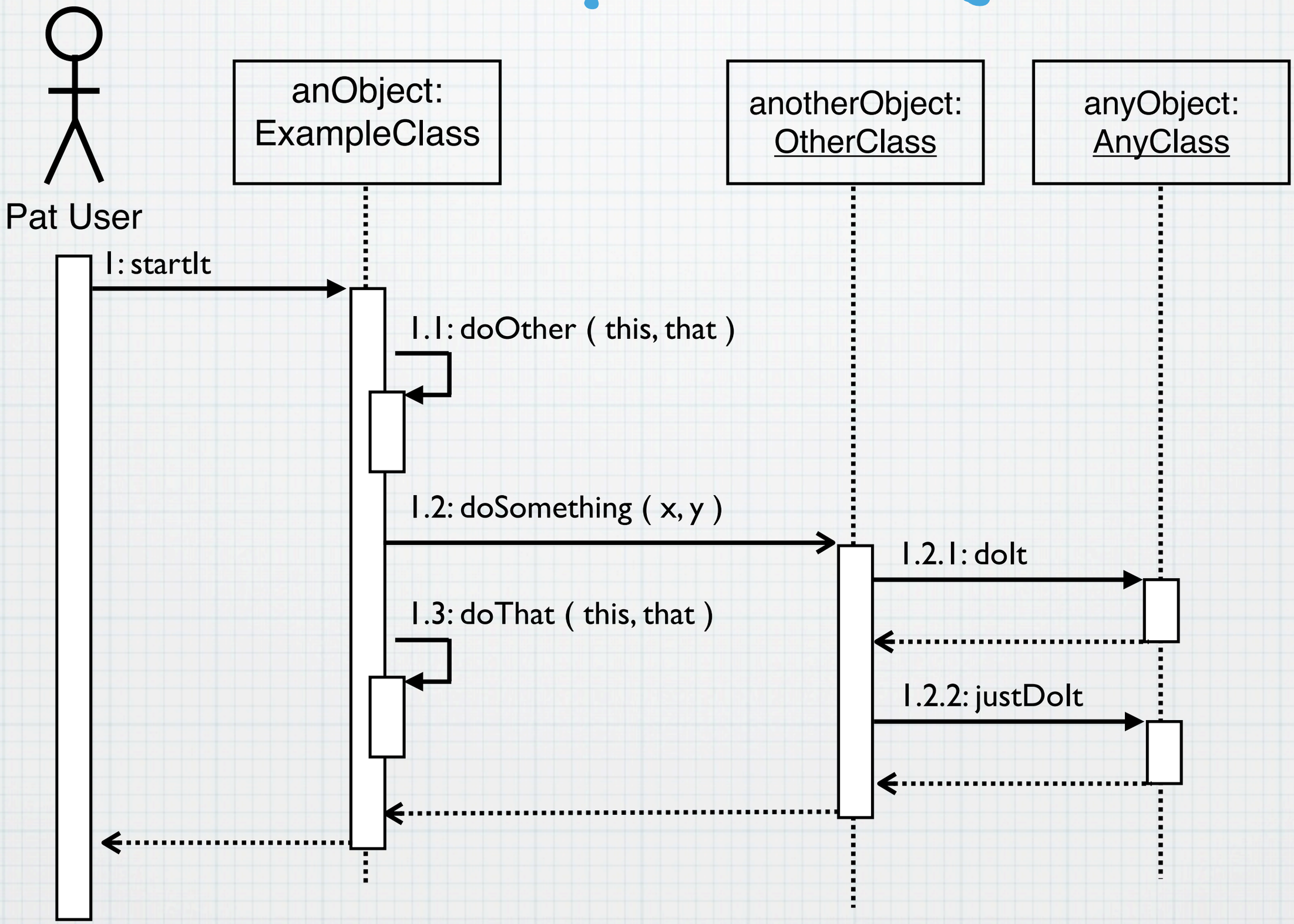


# Instance Deletion

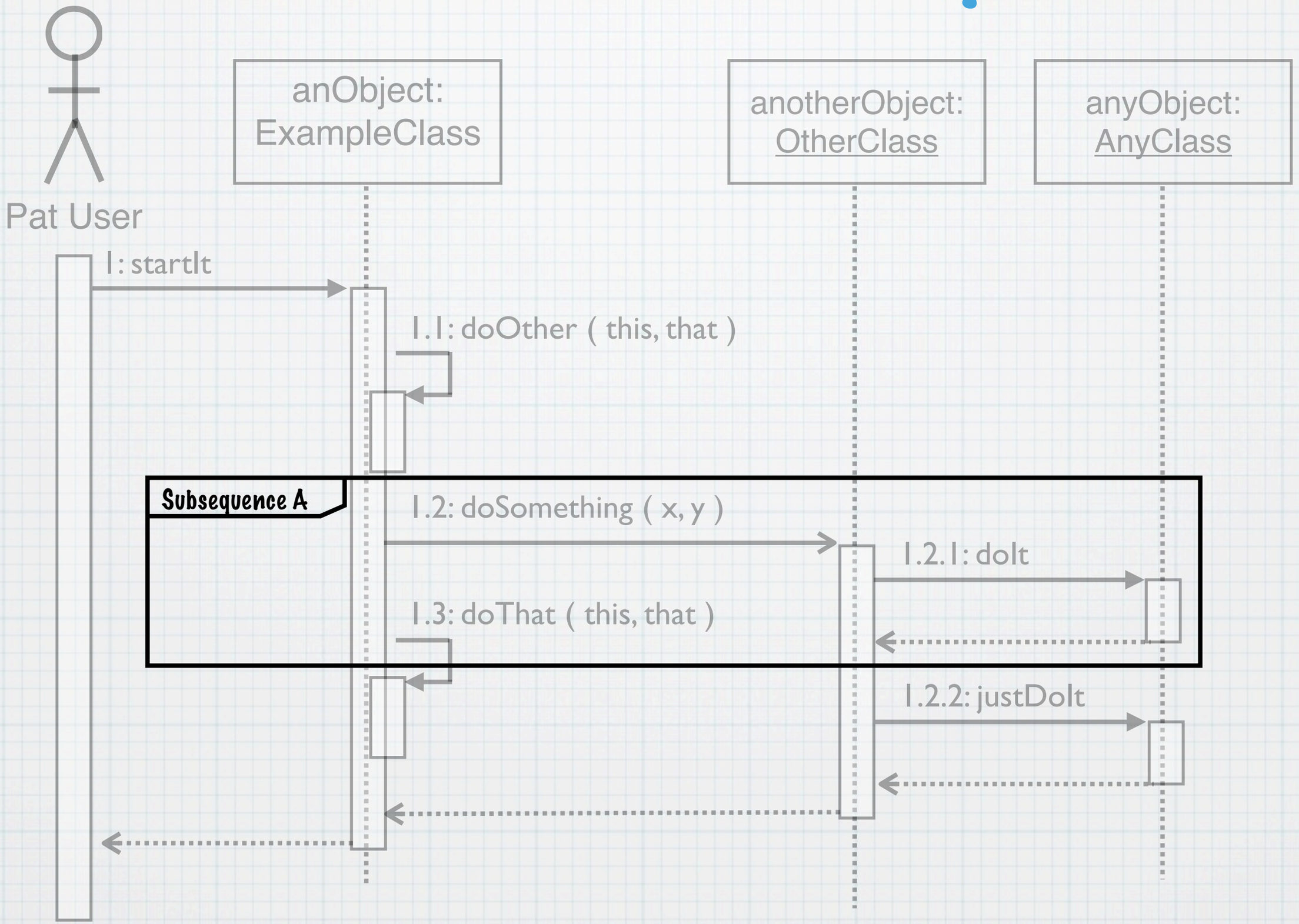
- At some point an object's usefulness may end
- The object deletion symbol indicates that not only has the activation ended, but the object no longer exists
  - Notice the life line ends!



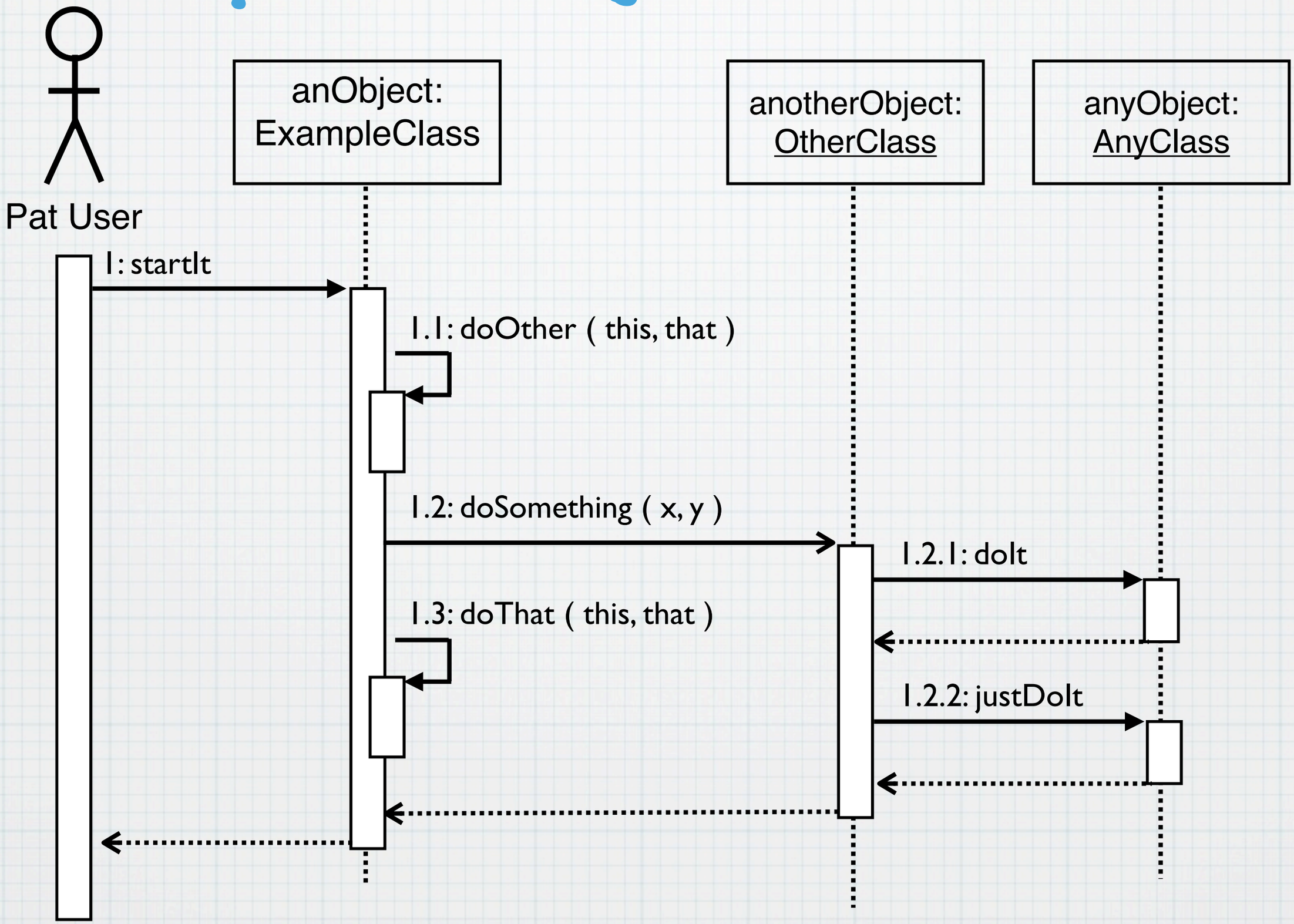
# Actors in Sequence Diagrams



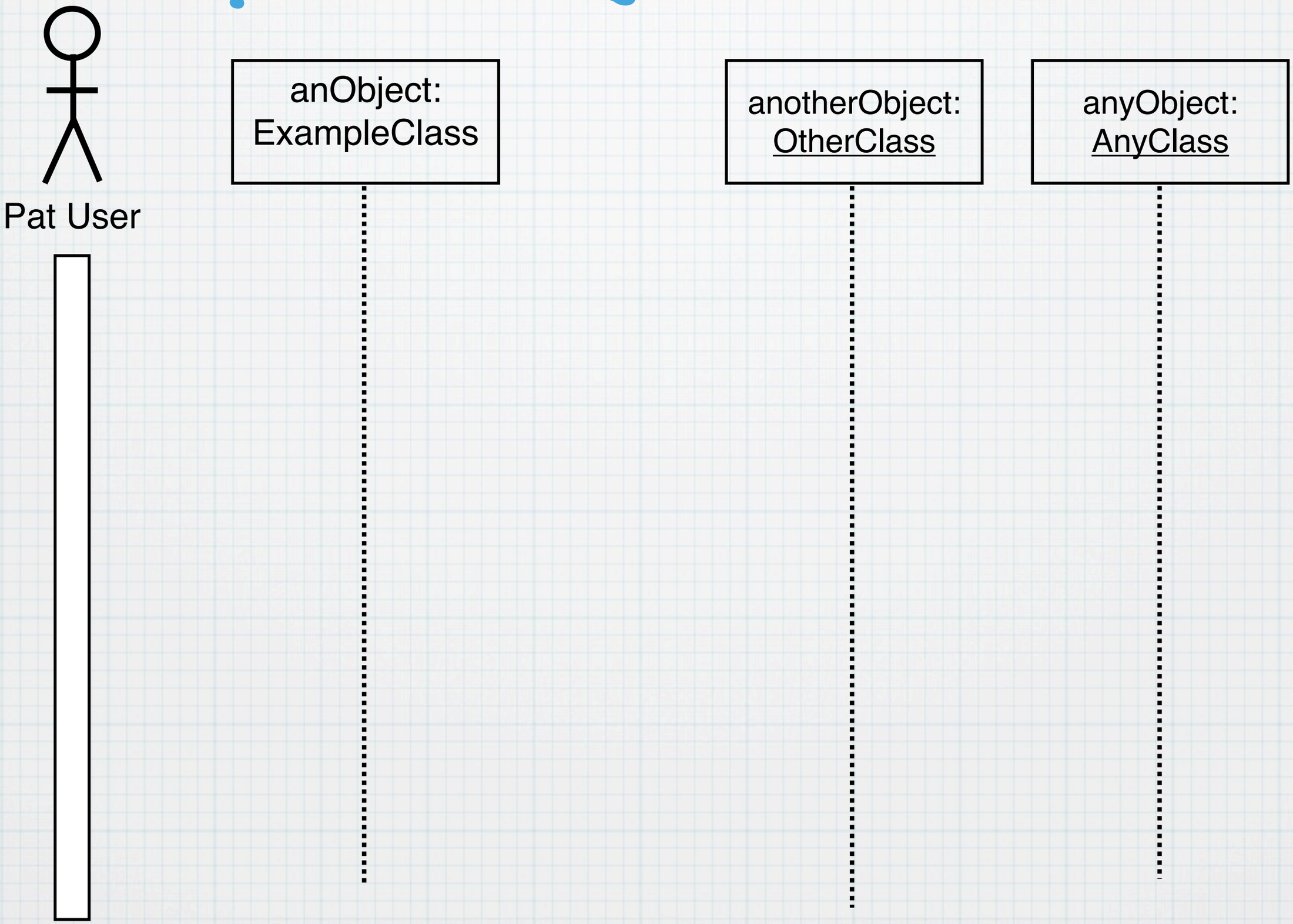
# Frames to Denote Subsequences



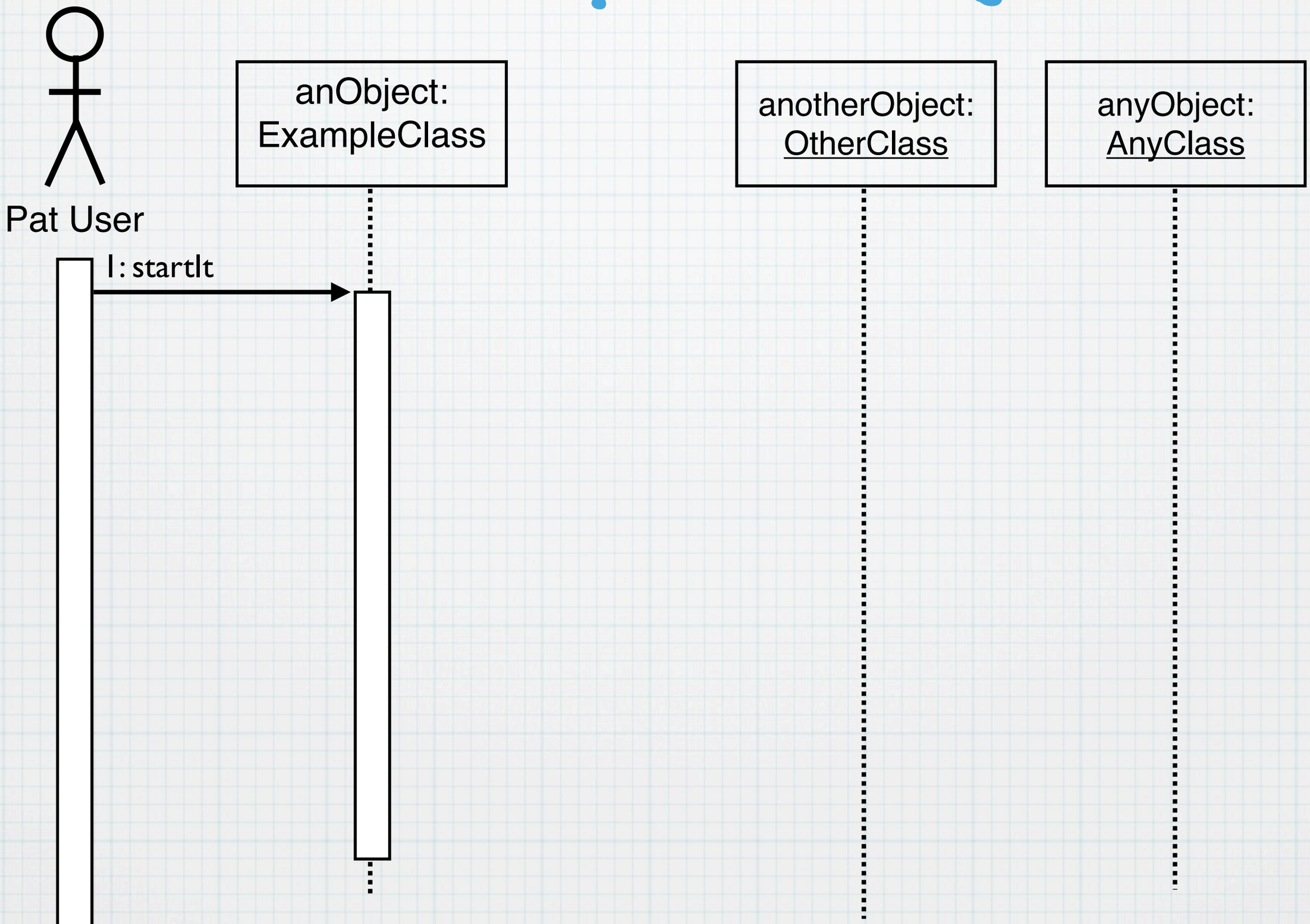
# Sequence Diagram in Action



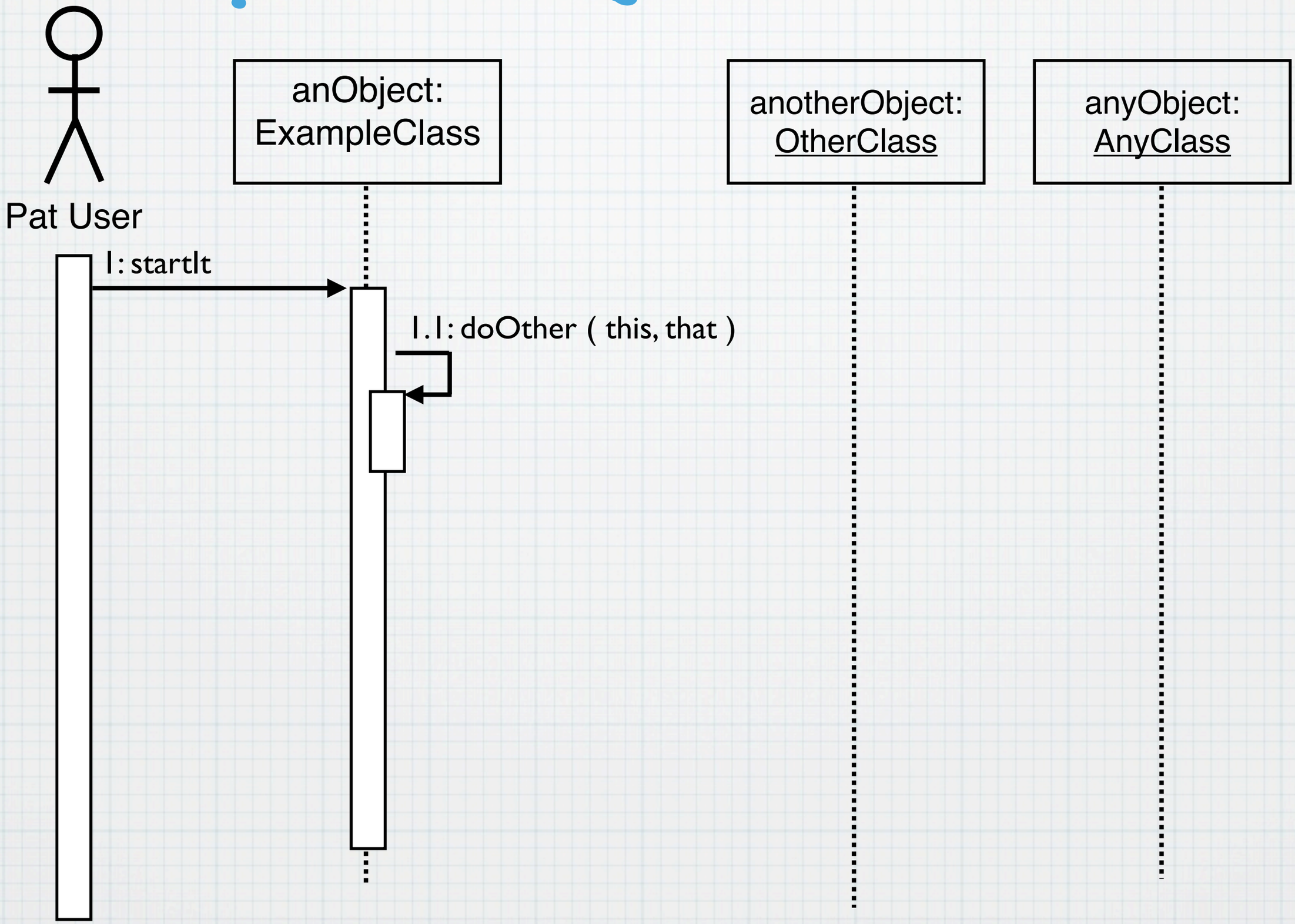
# Sequence Diagram in Action



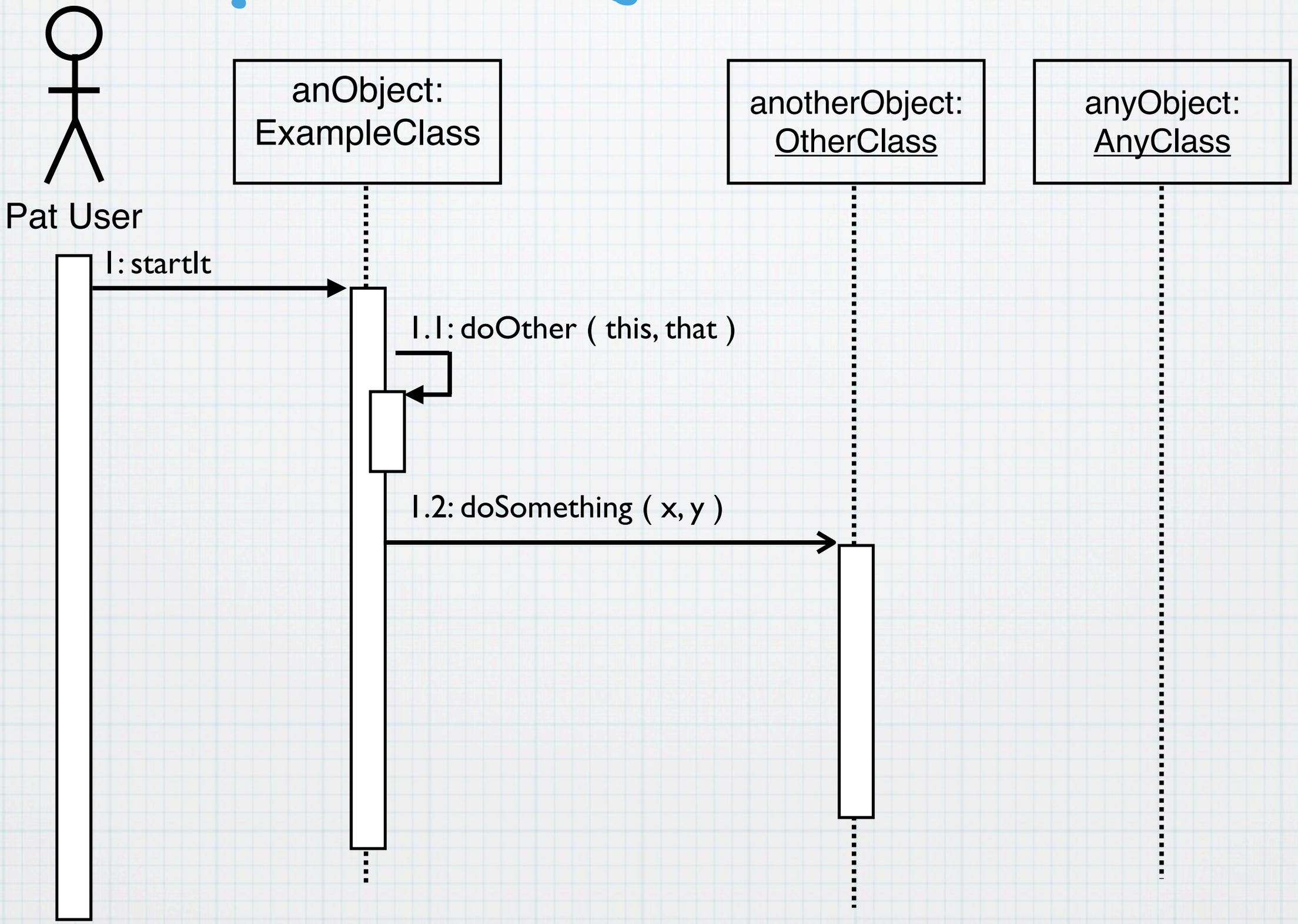
# Actors in Sequence Diagrams



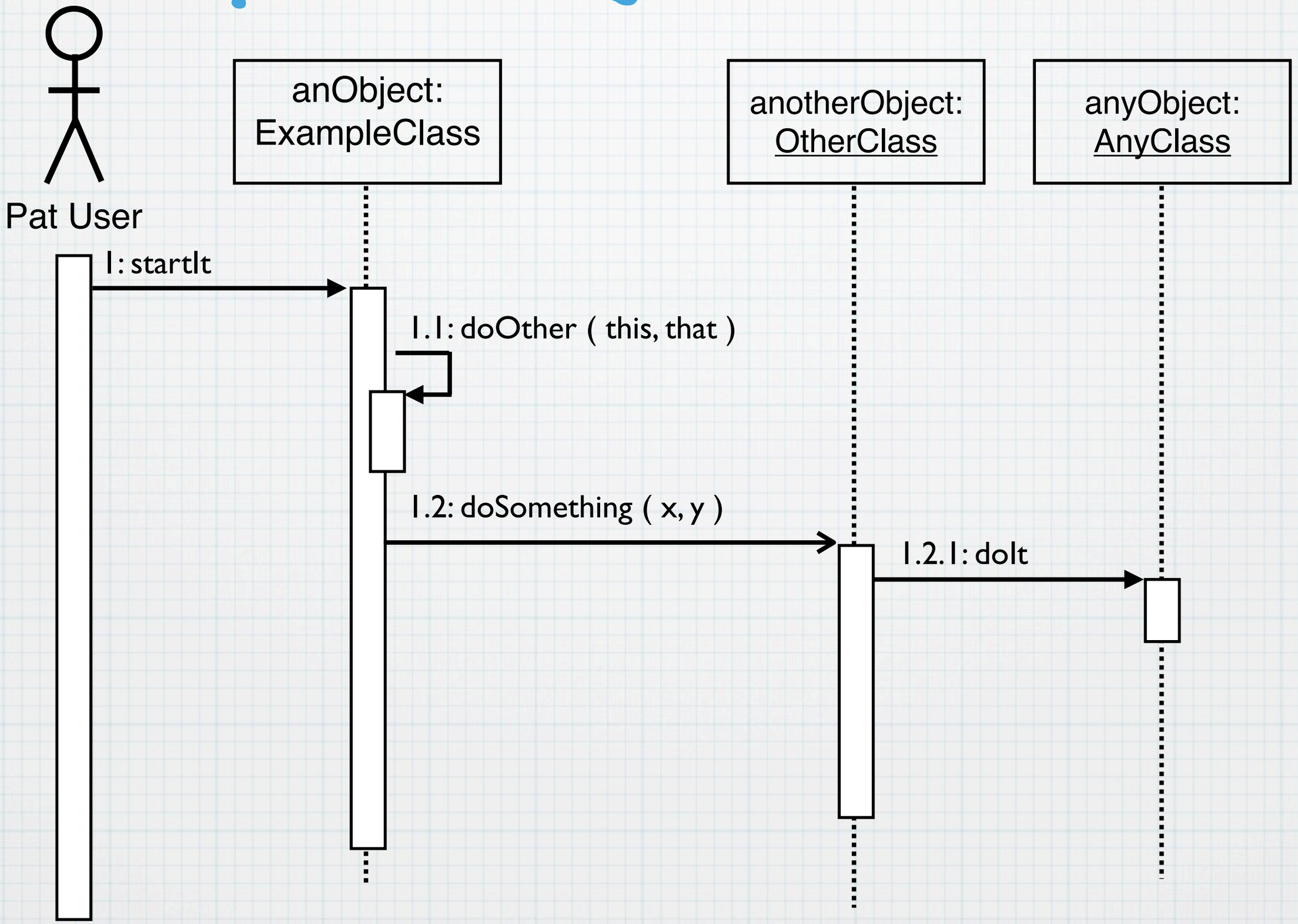
# Sequence Diagram in Action



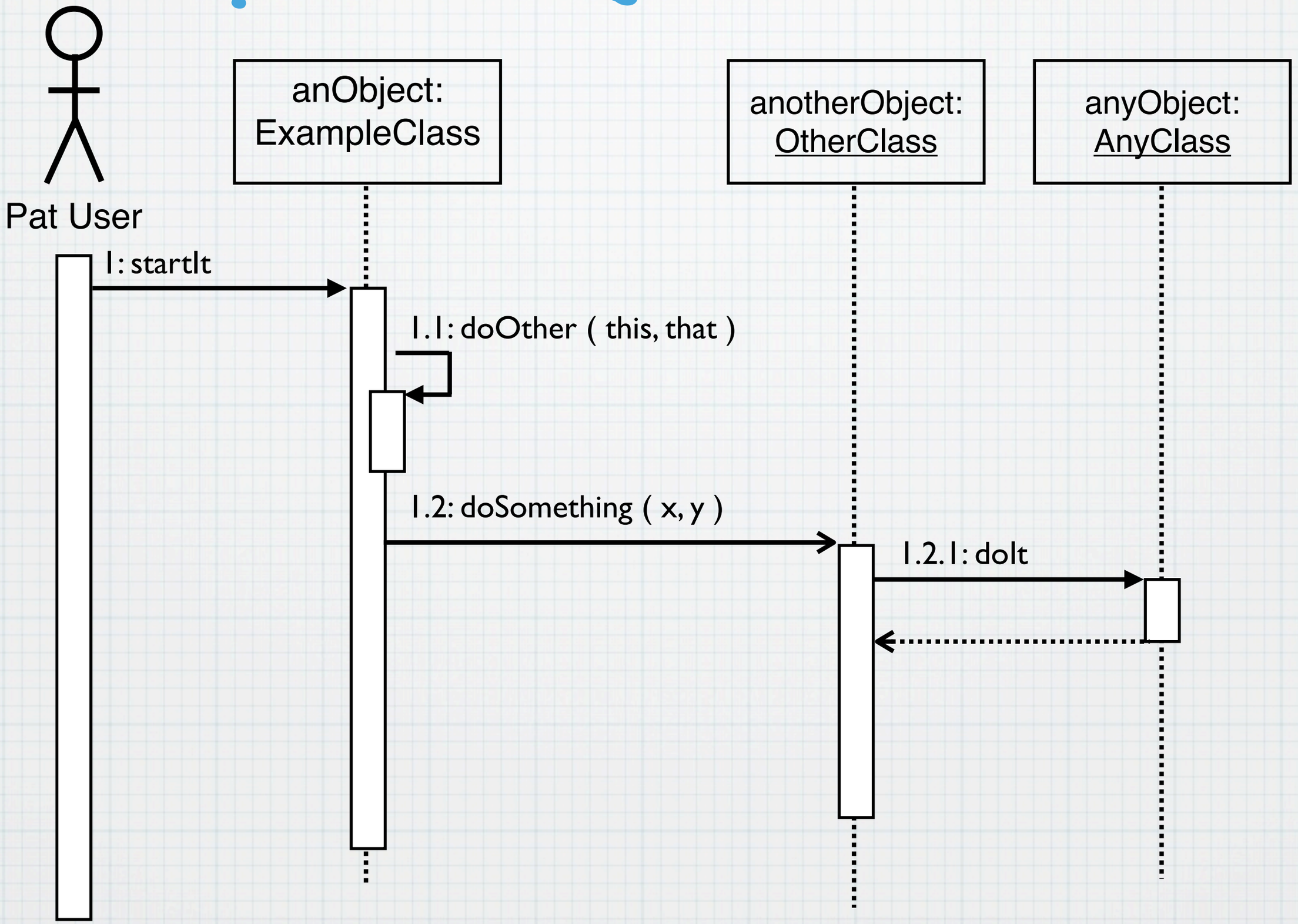
# Sequence Diagram in Action



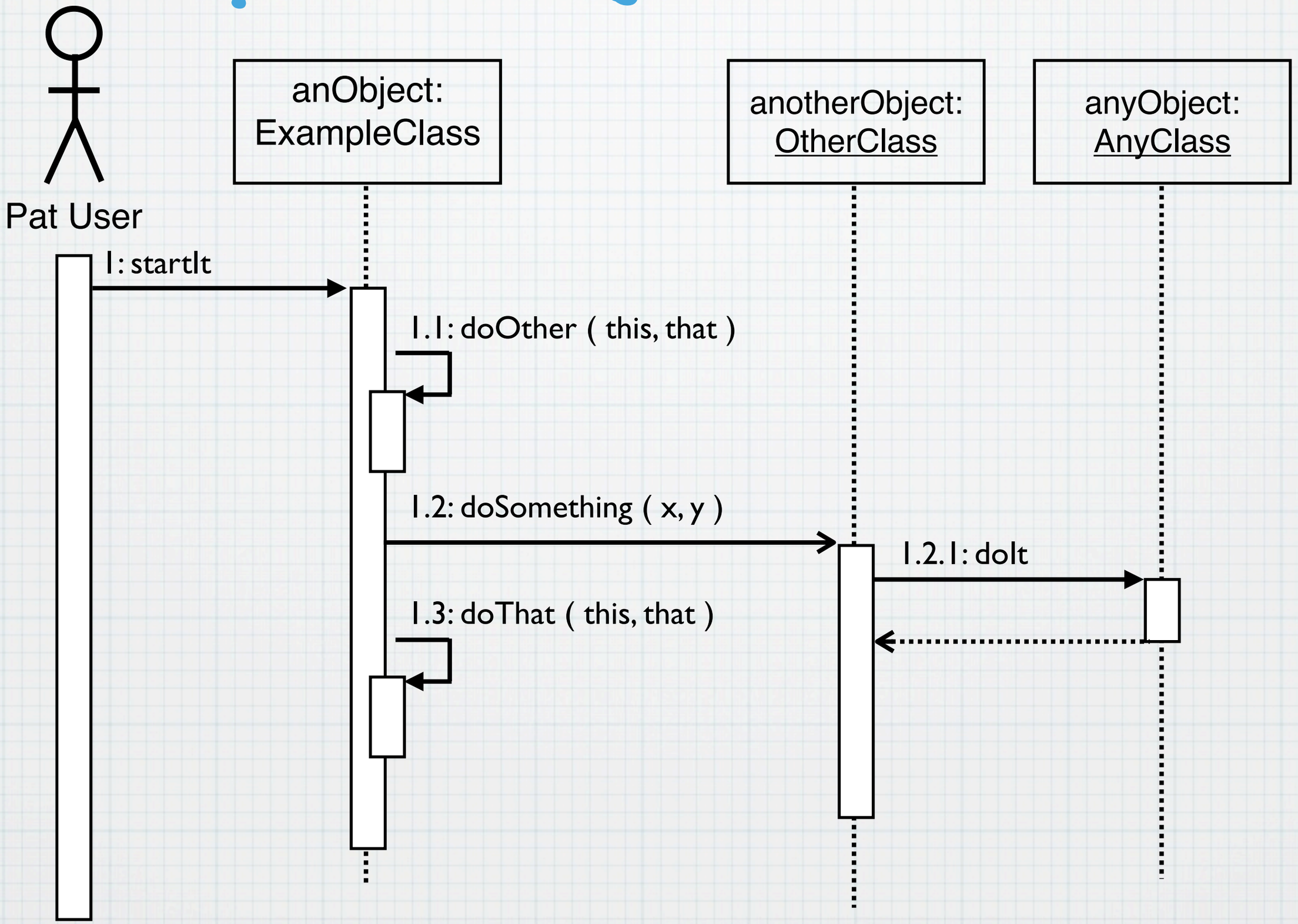
# Sequence Diagram in Action



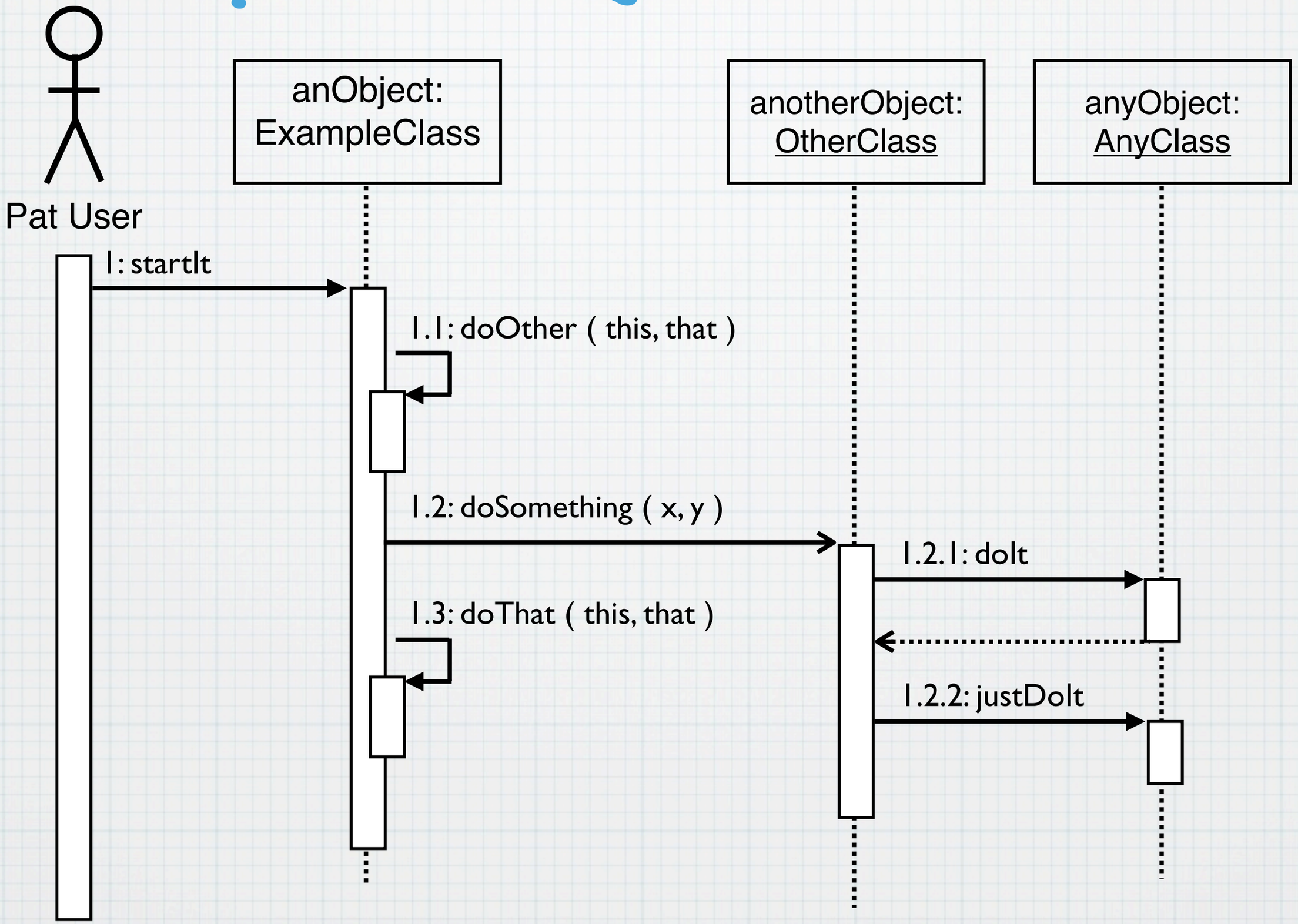
# Sequence Diagram in Action



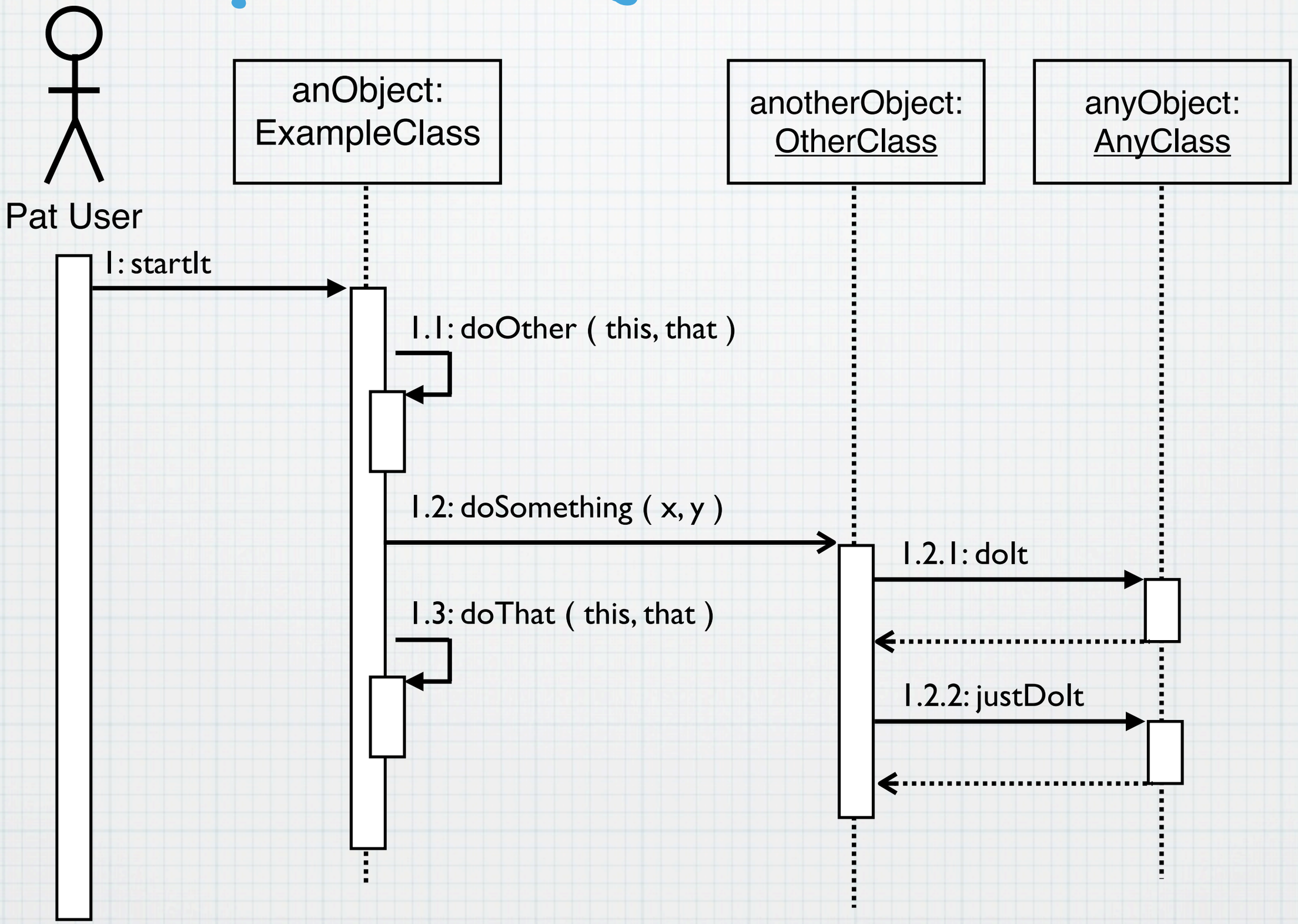
# Sequence Diagram in Action



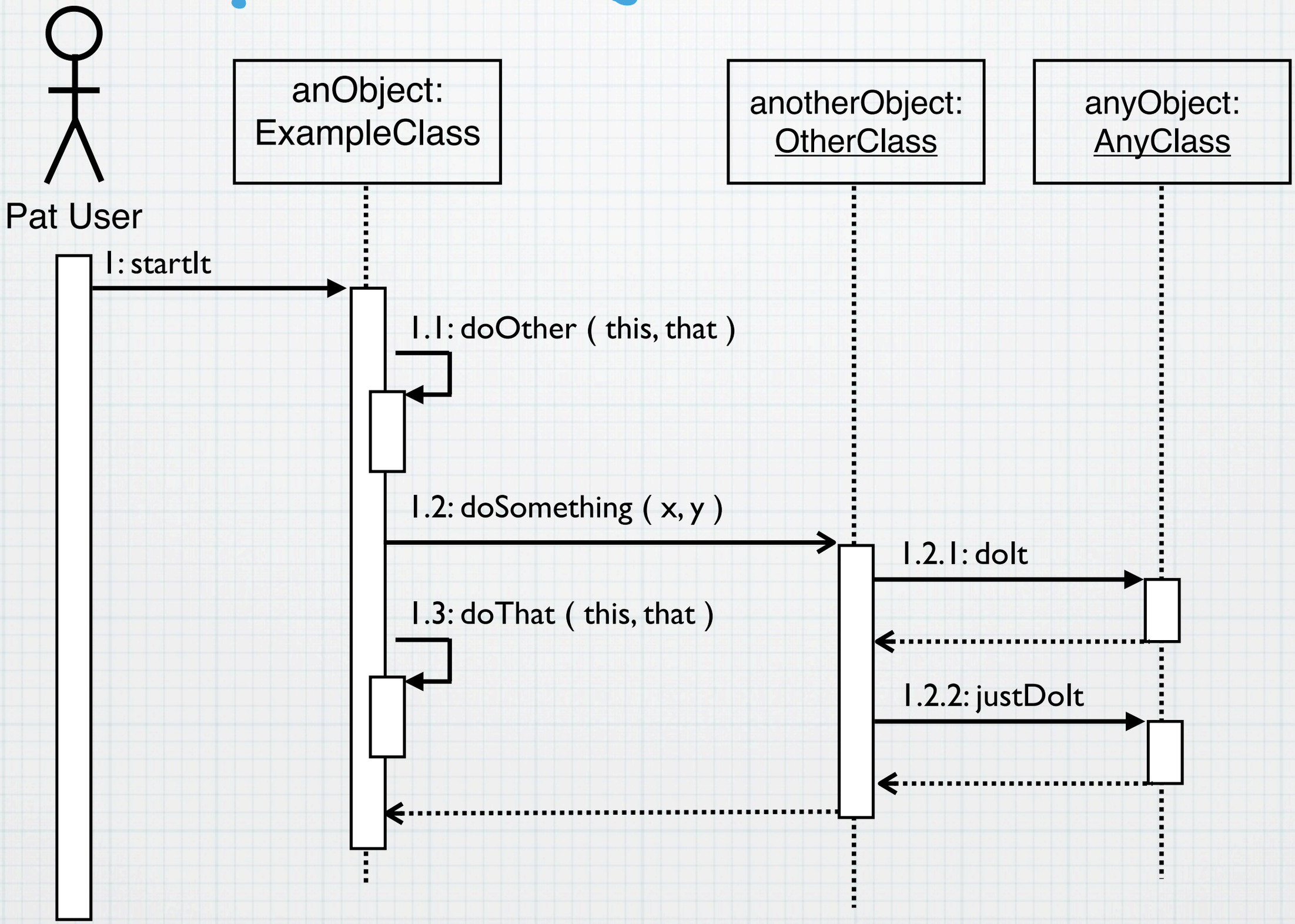
# Sequence Diagram in Action



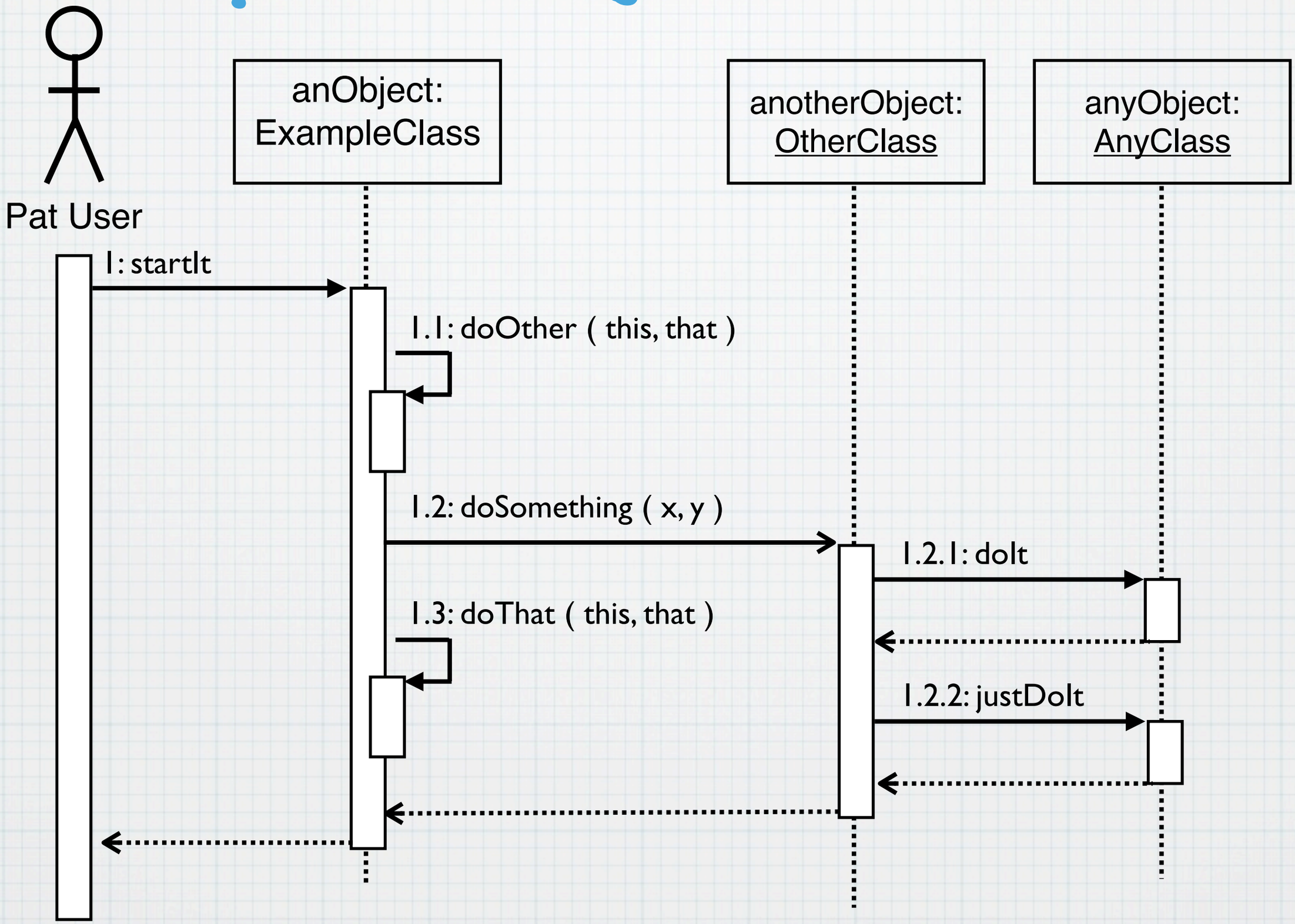
# Sequence Diagram in Action



# Sequence Diagram in Action



# Sequence Diagram in Action



# You Need to be able to Explain:

* Sequence Diagram
* Message
* Message flow
* Sequence Expression
* Iteration Expression
* Parameters
* Constraint
* Self-Delegation
* Conditional Message
* Concurrency
* Instance Deletion

# 2.4 Activity Diagramming in UML 2

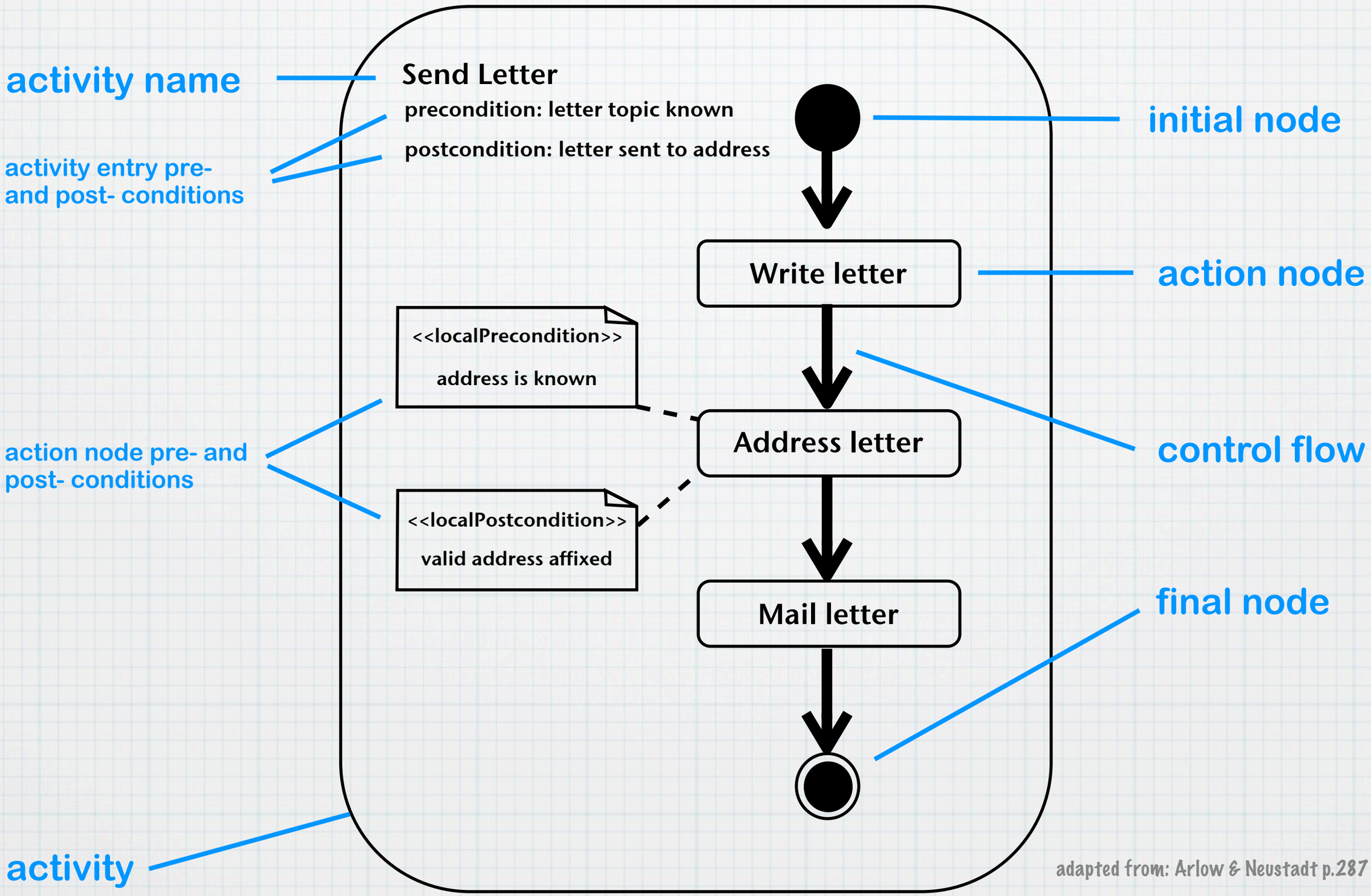
- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams (static in UML 1)
    - **Class** - business objects and their structures
    - Composite Structure 2
    - Component
    - Deployment
    - Object
    - Package
  - Dynamic diagrams
    - **Activity**
    - Interaction
      - **Sequence** - system actions that complete a task
      - Communication (collaboration in UML 1)
      - Interaction Overview 2
      - Timing 2
    - **Use Case** - user / system interactions / interfaces
    - State Machine

2 new in UML2

# Activity Diagram

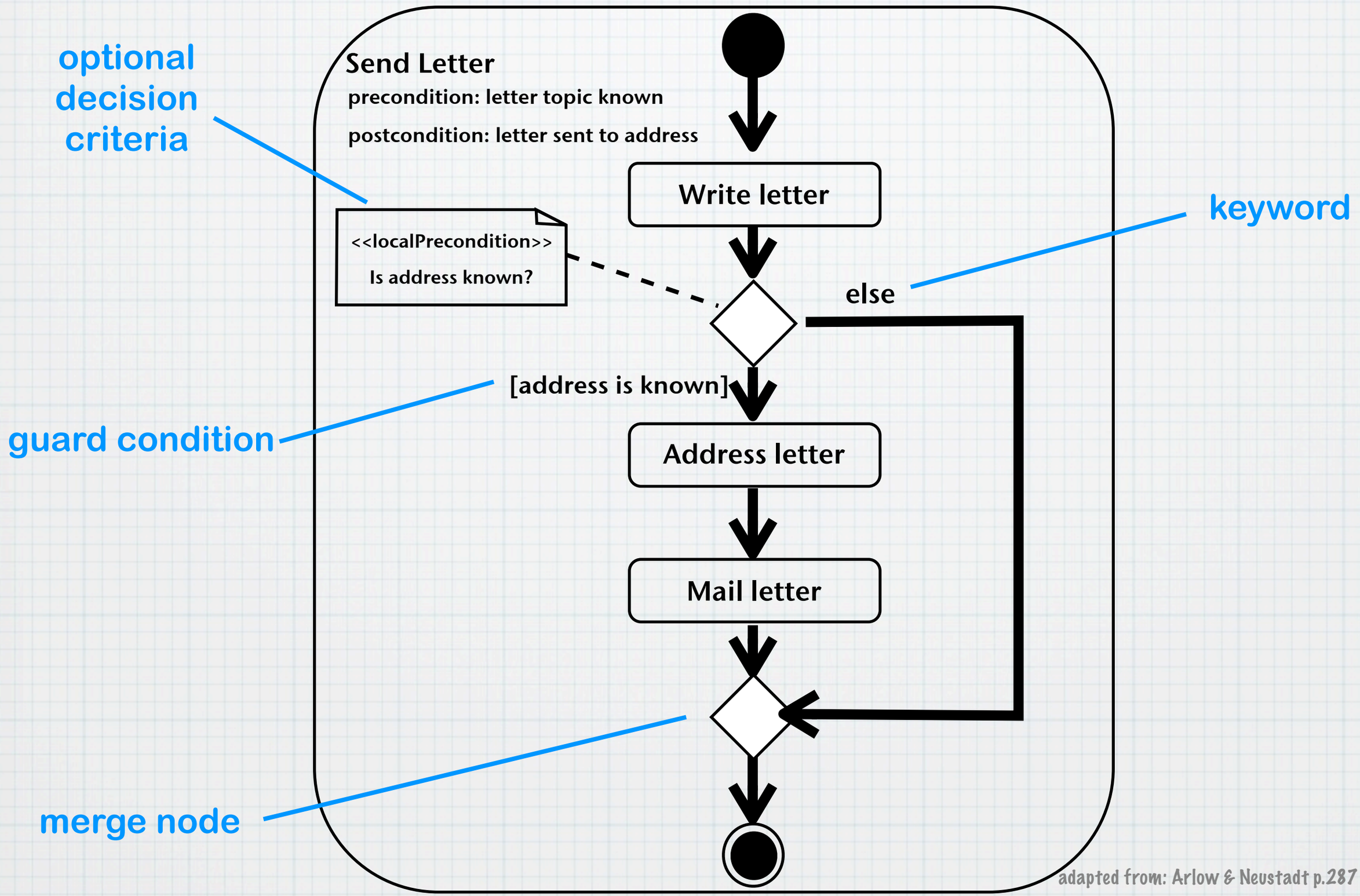
- Activity Diagrams are "OO Flowcharts"
- Activities are **nodes** connected by **edges**.
  - nodes - points where activity or decision occurs
    - **Action** - discrete units of work atomic within the activity
      - all node activity is completed before control passes through them
      - if a node has multiple input edges ALL precedent actions must complete first
      - action types
        - call action - invokes an activity, behavior or operation
        - send signal - sends an asynchronous signal to a "receiving" action node
        - accept event action - waits for events detected by its owning object of containing activity
    - **Control** - control the flow of control through an activity
    - **Object** - objects used in an activity
  - edges - paths of flow through the diagram
    - **control** flow - represent the flow of control through an activity
    - **object** flow - represent the flow of objects through an activity

# Activity Diagram Example



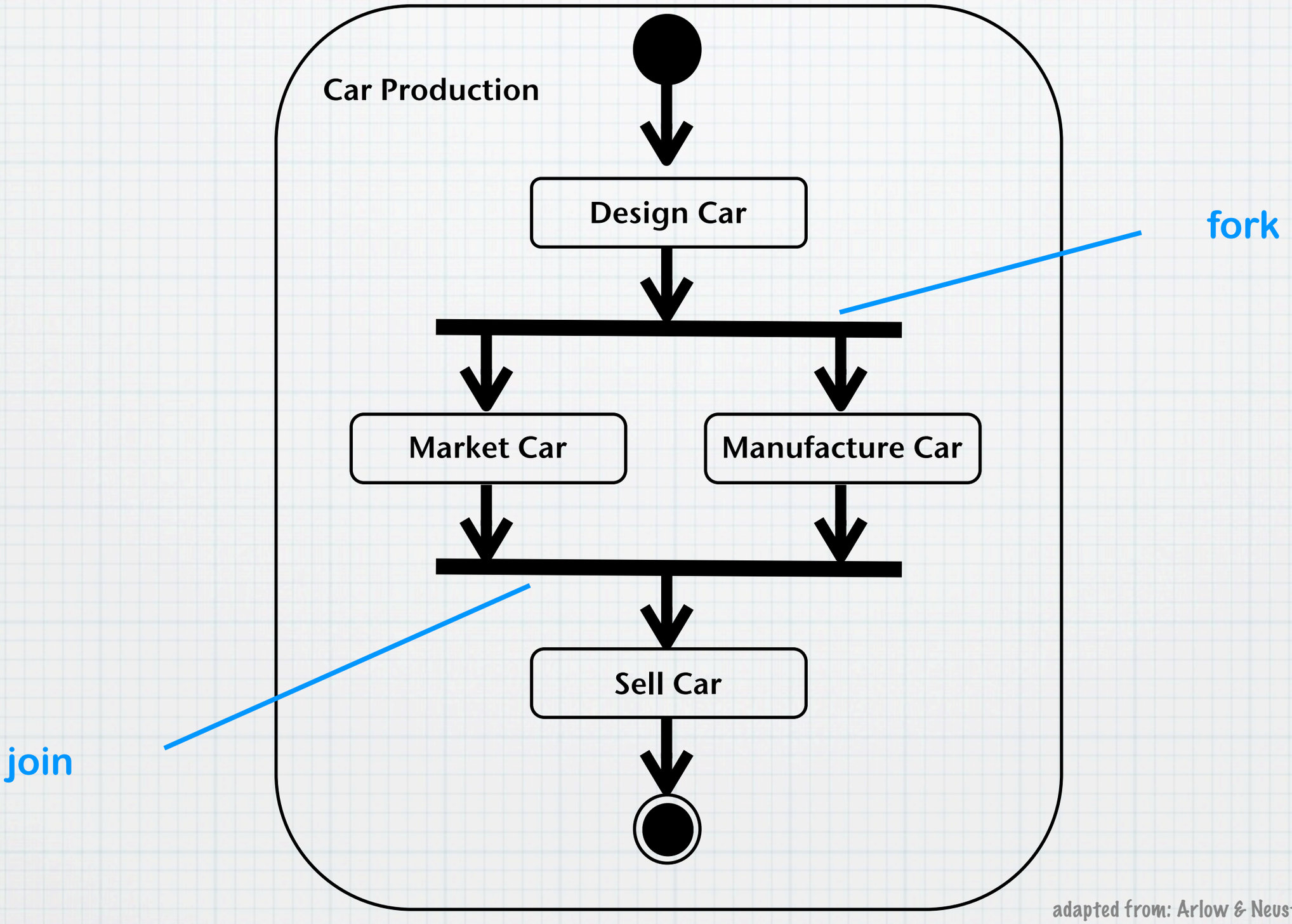

adapted from: Arlow & Neustadt p.287

# Path Control



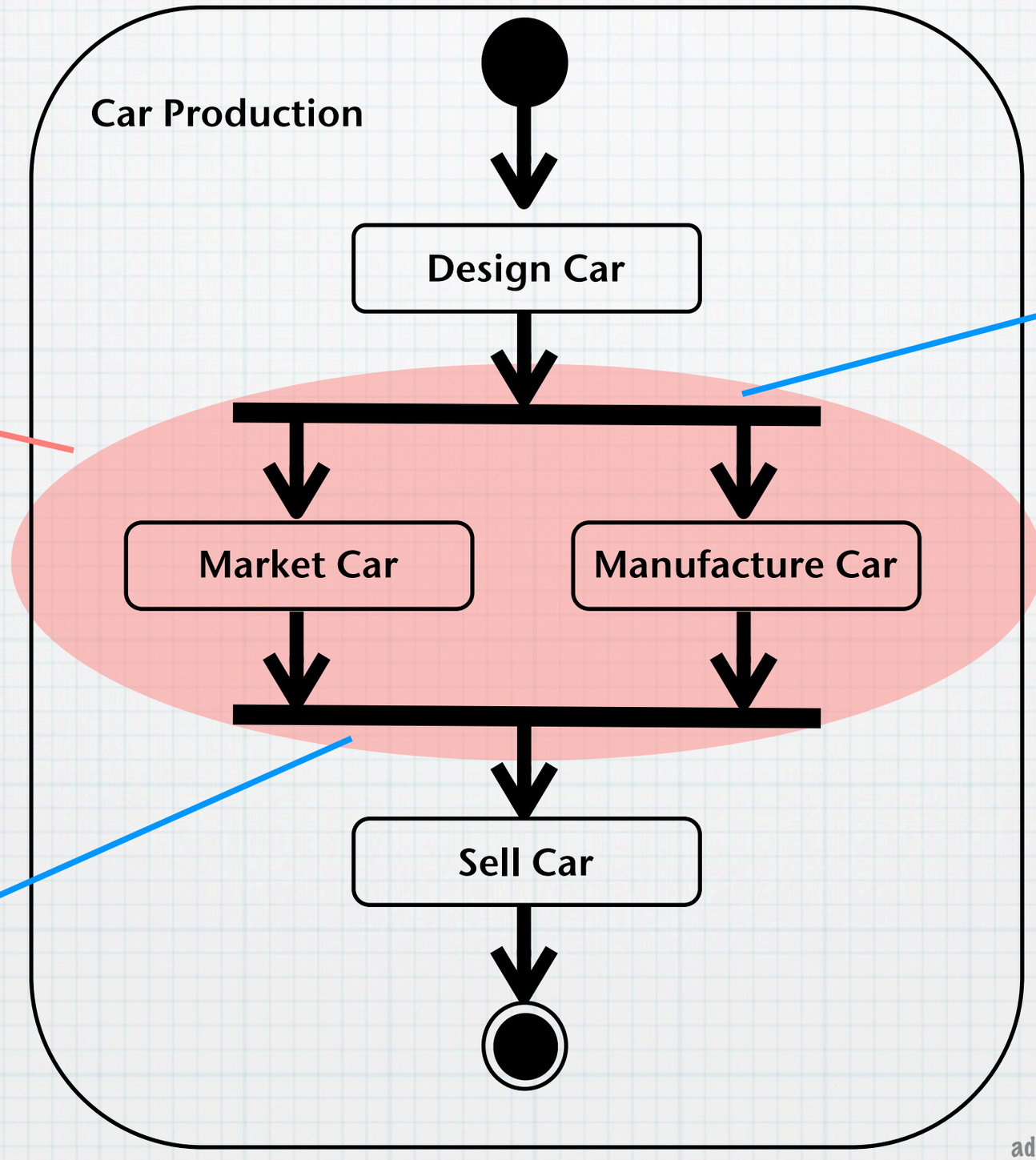

adapted from: Arlow & Neustadt p.287

# Concurrency Control




adapted from: Arlow & Neustadt p.300

# Concurrency Control




**concurrent activities**

**fork**

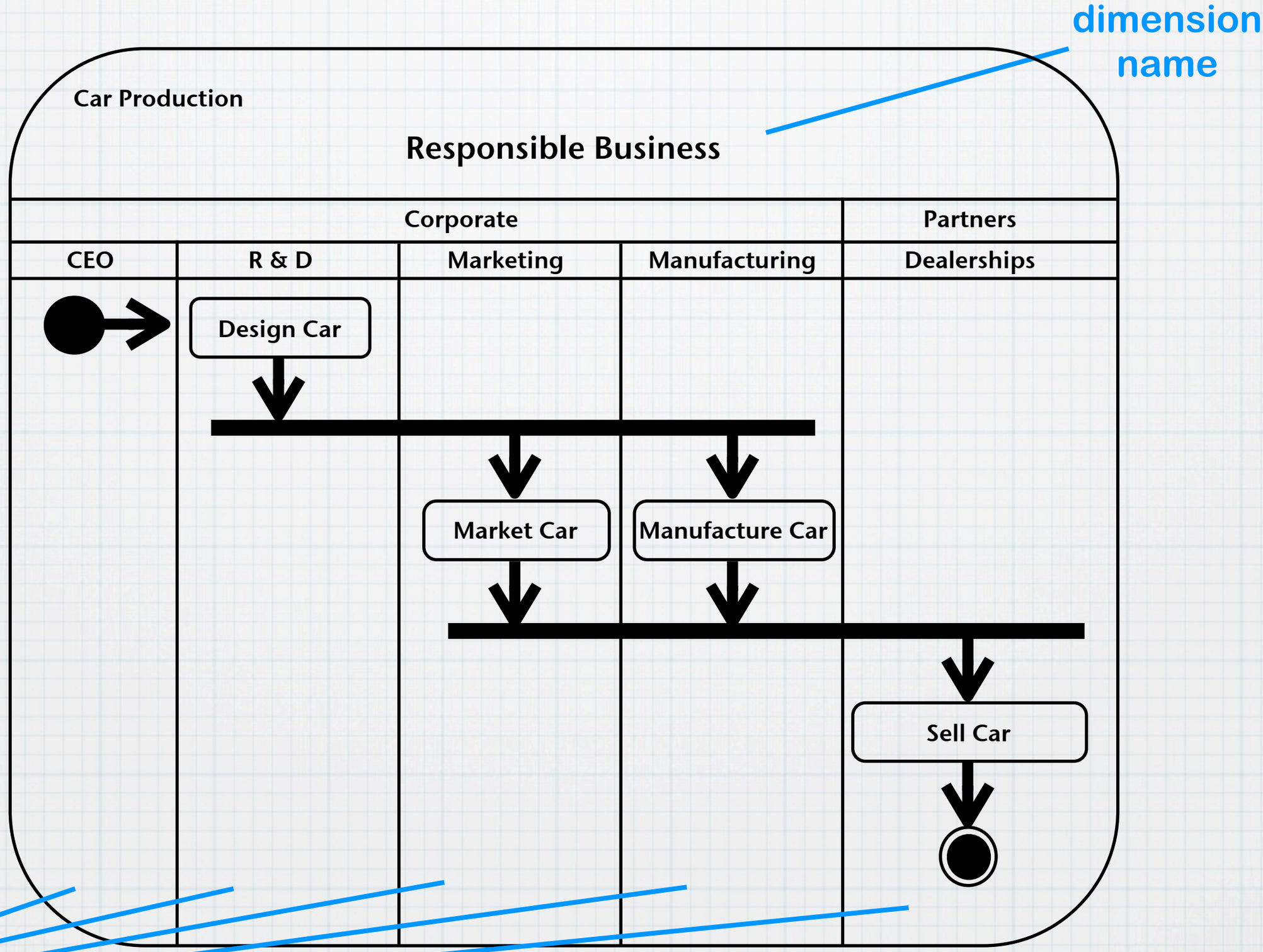
**Multiple paths of activity begin and proceed (by default) unaware of each other!**

**join**

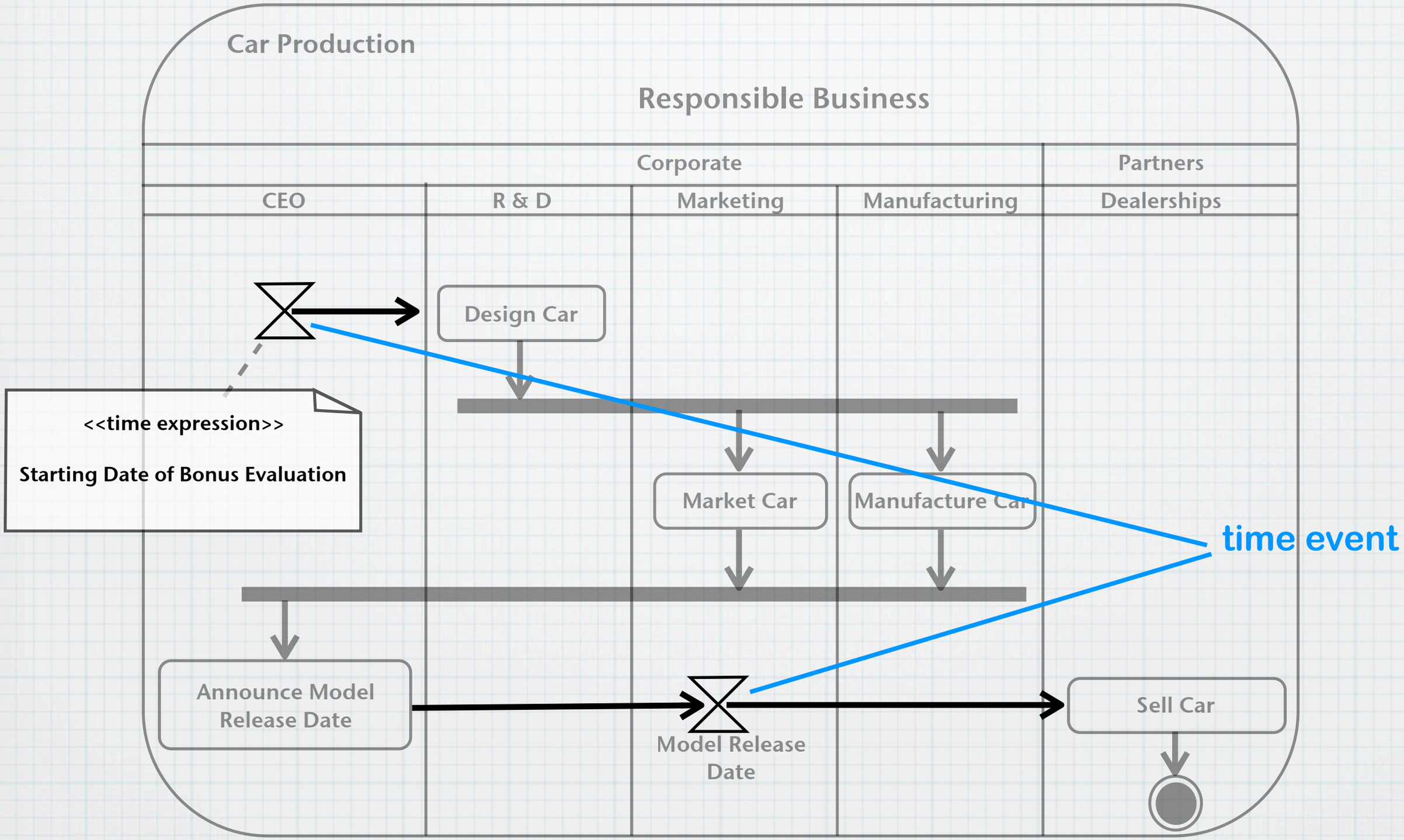
**All incoming paths must reach the join before the outgoing path begins!**

adapted from: Arlow & Neustadt p.300

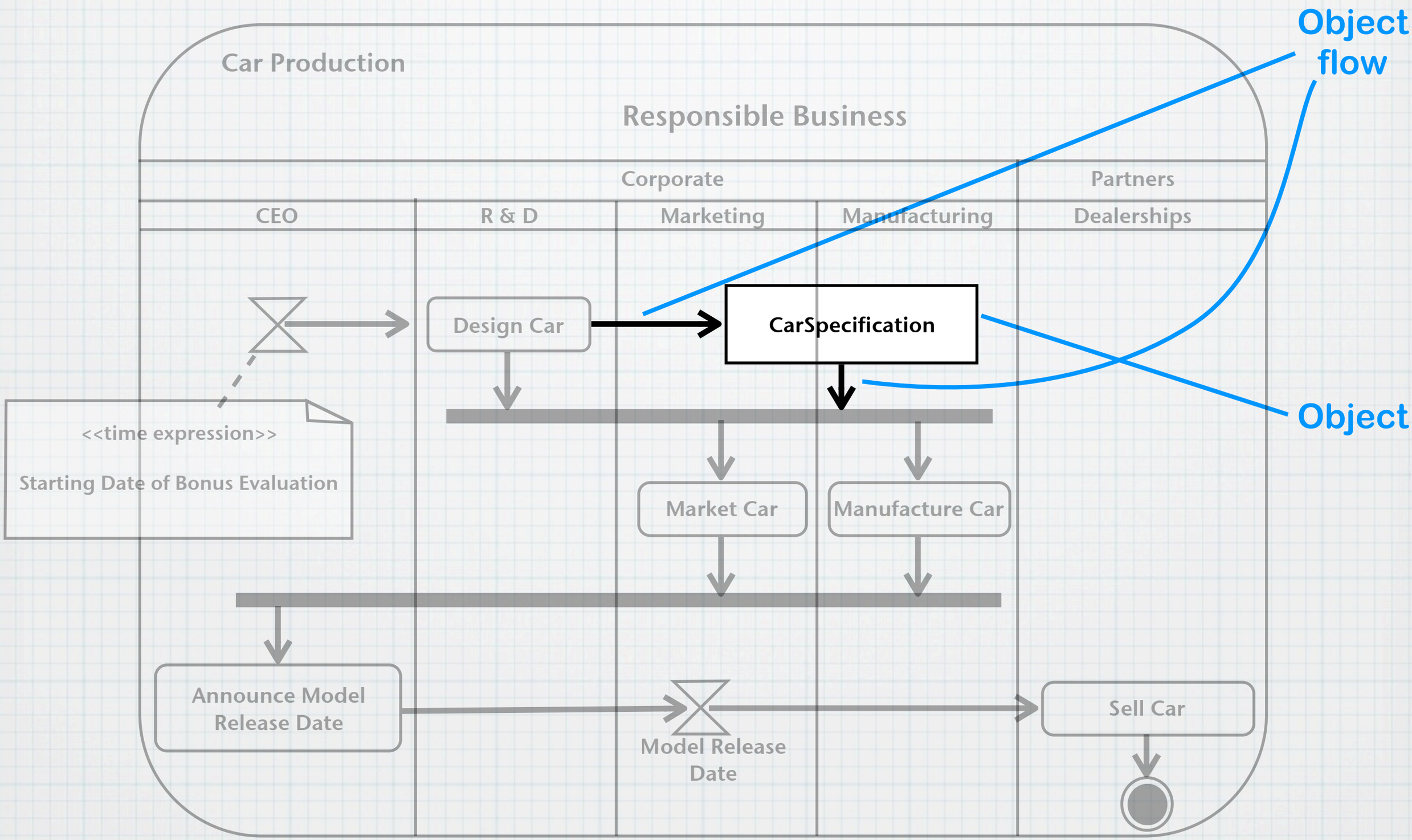
# Activity Partitions



# Time Events


Car Production
Responsible Business
Corporate
Partners
CEO
R & D
Marketing
Manufacturing
Dealerships
Design Car
<<time expression>>
Starting Date of Bonus Evaluation
Market Car
Manufacture Car
time event
Announce Model Release Date
Model Release Date
Sell Car

# Object Flow


Car Production
Responsible Business
Corporate
Partners
CEO
R & D
Marketing
Manufacturing
Dealerships
Design Car
CarSpecification
<<time expression>>
Starting Date of Bonus Evaluation
Market Car
Manufacture Car
Announce Model Release Date
Model Release Date
Sell Car
Object flow
Object

# You Need to be able to Explain:

* Activity Diagram
  * nodes, edges, partitions
* Action Node
  * pre-, post- conditions, call action (activity, behavior, operation), send signal, accept event
* Control Node
  * decision criteria, guard conditions, merge node, time event
* Control Flow
  * sequential, synchronous, asynchronous
* Object Flow
  * object node

# 2.5 Other Diagrams in UML 2

- UML is a collection of diagramming disciplines that define the static and dynamic characteristics of a problem or system
  - Structure diagrams (static in UML 1)
    - **Class** - business objects and their structures
    - Composite Structure 2
    - Component
    - Deployment
    - Object
    - Package
  - Dynamic diagrams
    - **Activity**
    - Interaction
      - **Sequence** - system actions that complete a task
      - Communication (collaboration in UML 1)
      - Interaction Overview 2
      - Timing 2
    - **Use Case** - user / system interactions / interfaces
    - State Machine

2 new in UML2

# Composite Structure Diagram

A composite structure diagram depicts the internal structure of a classifier, as well as the use of a collaboration in a collaboration use.

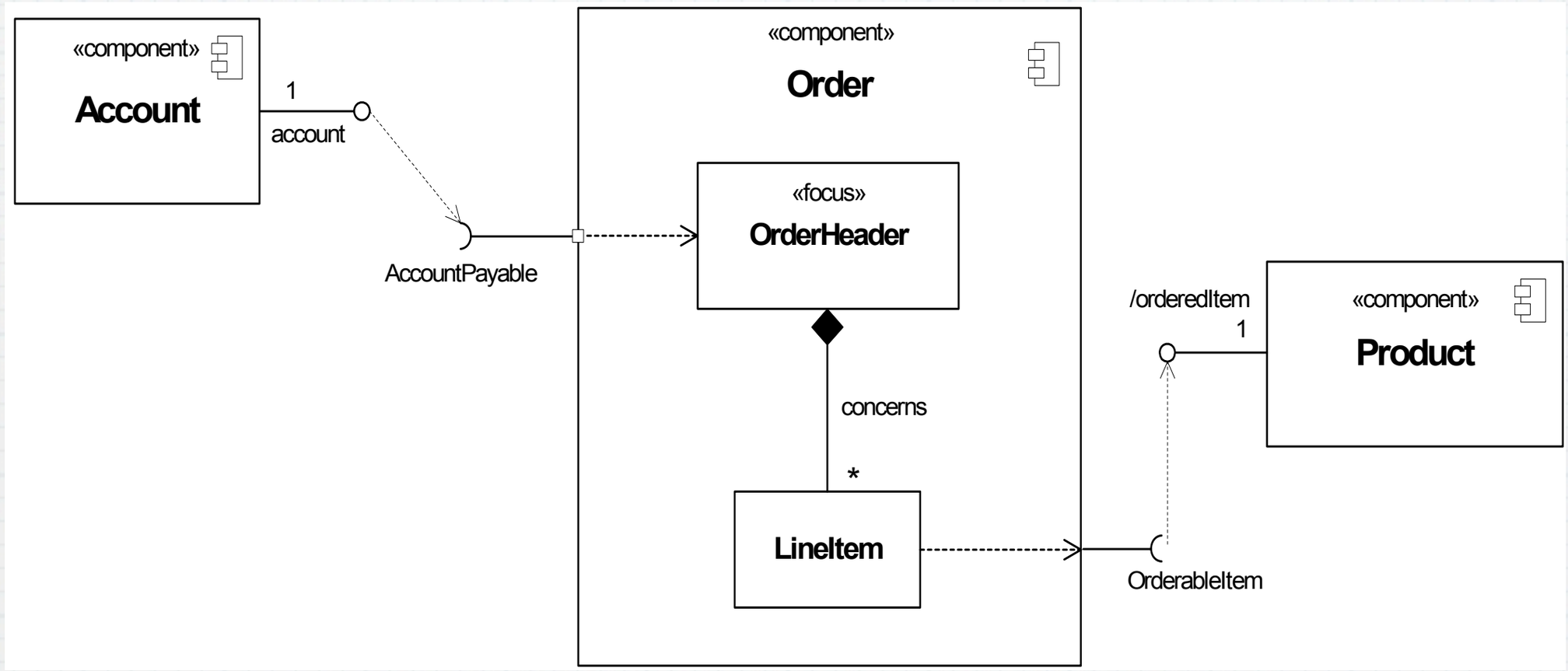



**Figure 8.14 - Example of a platform independent model of a component, its provided and required interfaces, and wiring through dependencies on a structure diagram.**

**"Unified Modeling Language: Superstructure," Version 2.0, formal/05/07/04, http://www.omg.org**

# Component Diagram

A component is shown as a Classifier rectangle with the keyword «component». Optionally, in the right hand corner a component icon can be displayed. This is a classifier rectangle with two smaller rectangles protruding from its left hand side.

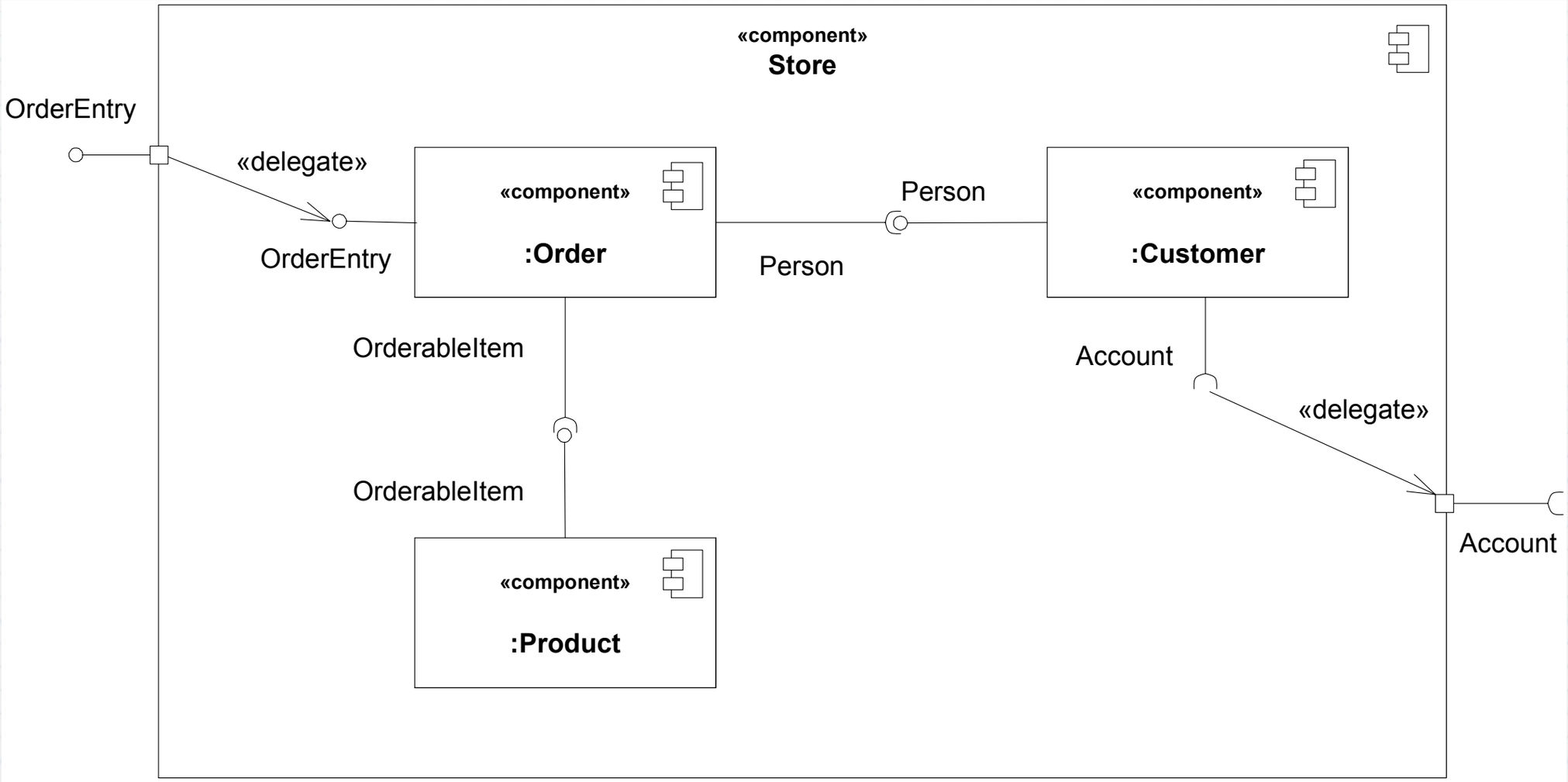
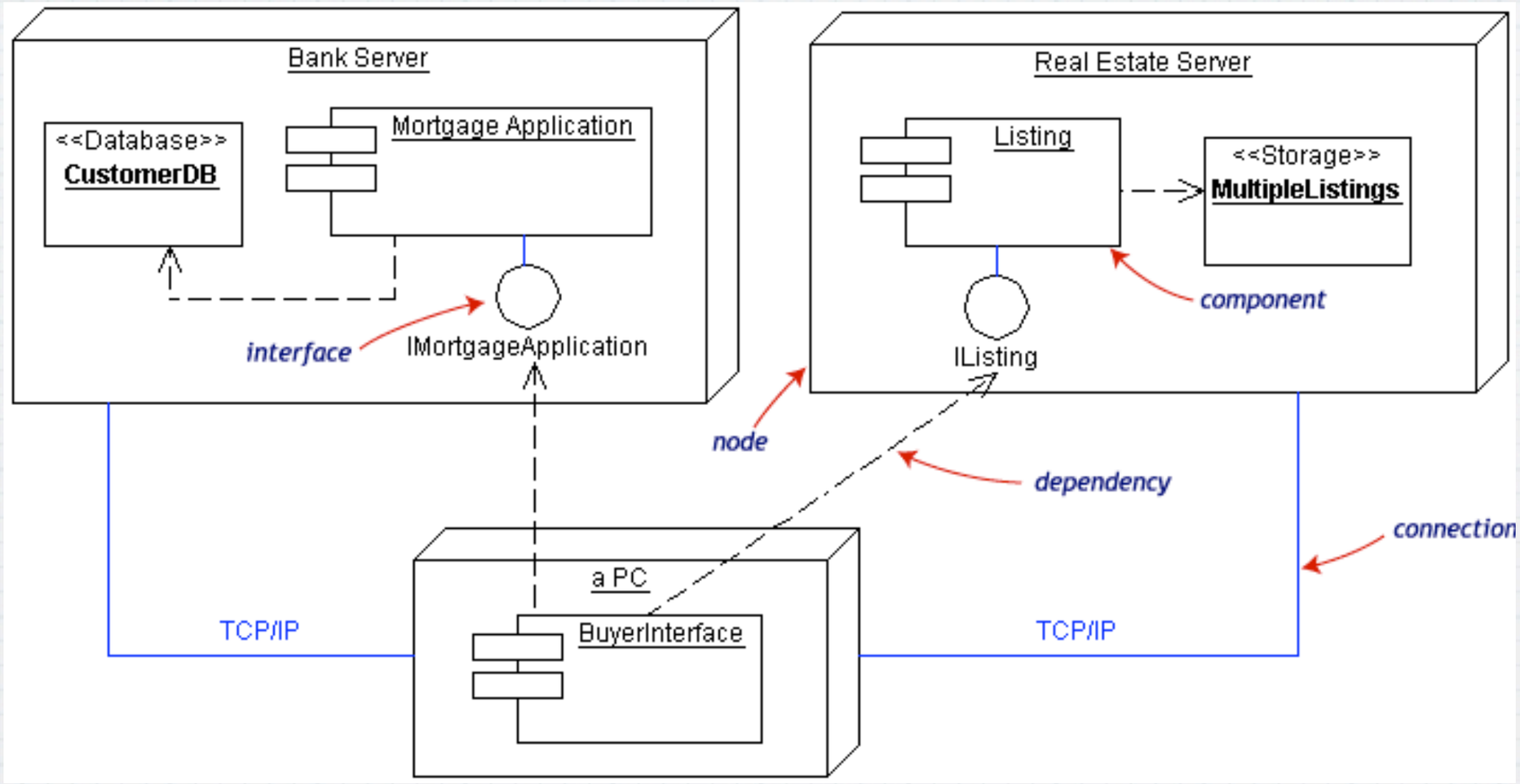



**Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components as parts of its internal assembly.**

**"Unified Modeling Language: Superstructure," Version 2.0, formal/05/07/04, http://www.omg.org**

# Deployment Diagram

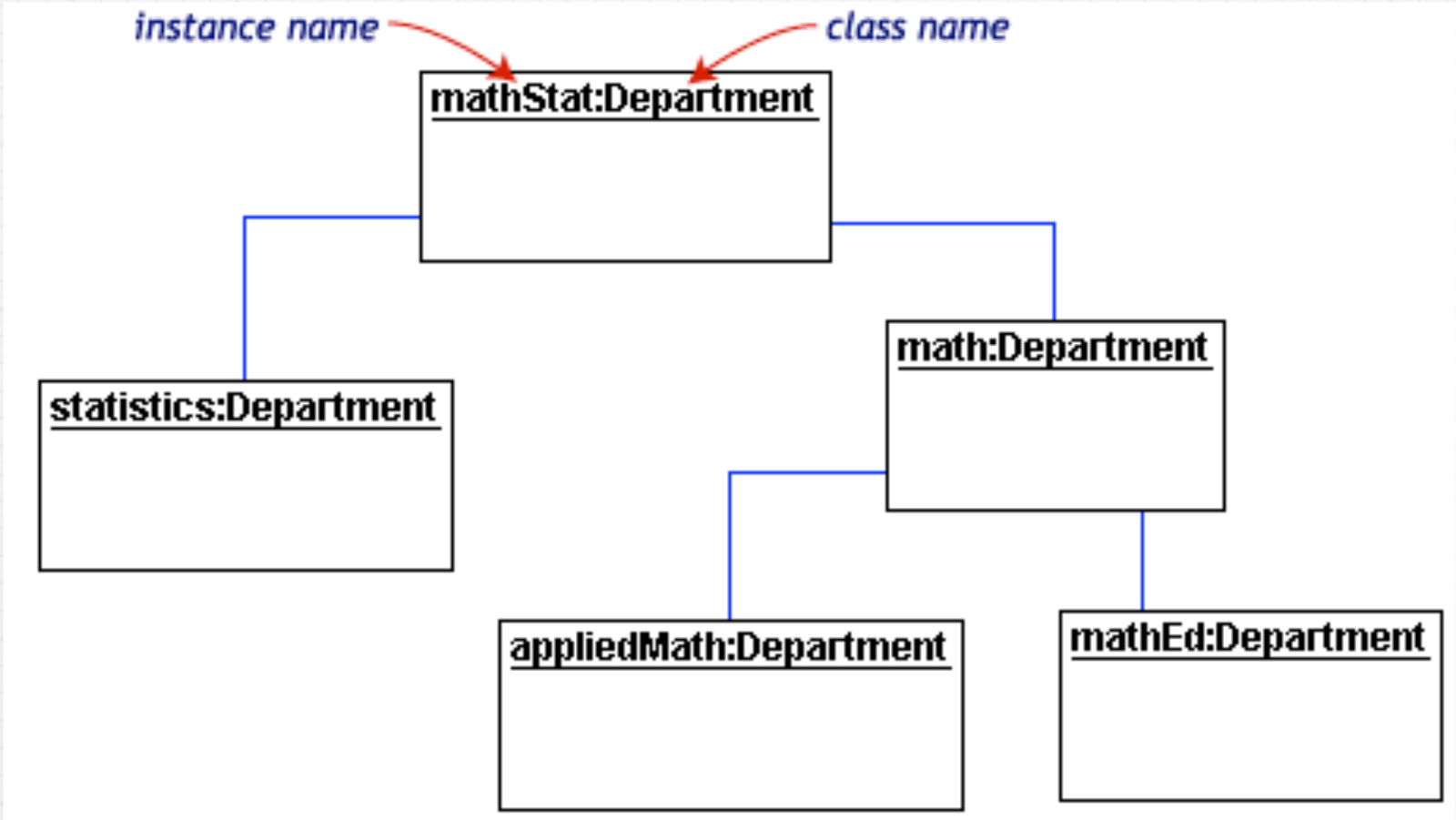
The physical hardware is made up of nodes. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.




**"Practical UML™: A Hands-On Introduction for Developers," http://dn.codegear.com/article/31863**

# Object Diagram

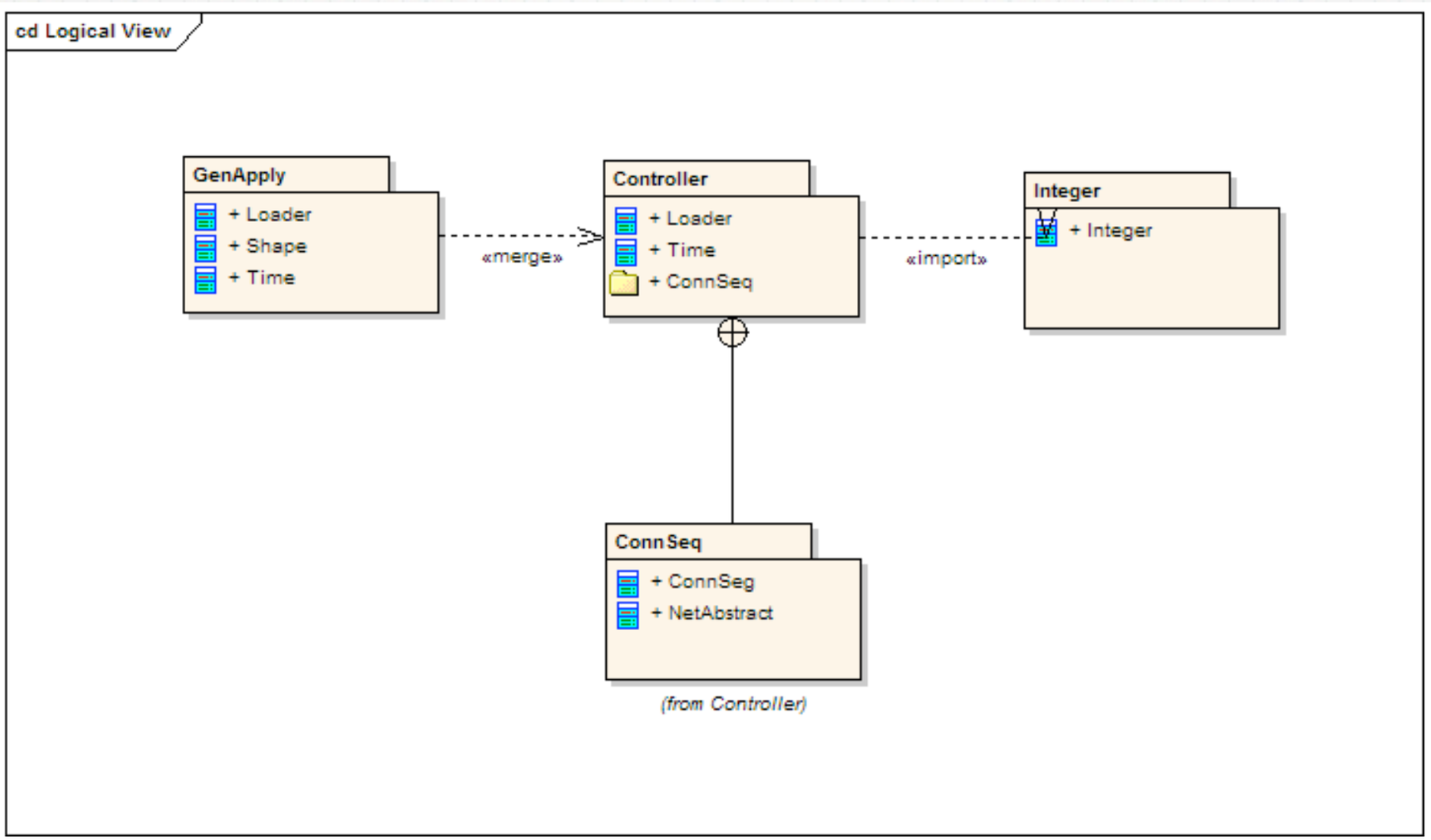
Each rectangle in the object diagram corresponds to a single instance. Instance names are underlined in UML diagrams. Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear.




"Practical UML™: A Hands-On Introduction for Developers," http://dn.codegear.com/article/31863

# Package Diagram

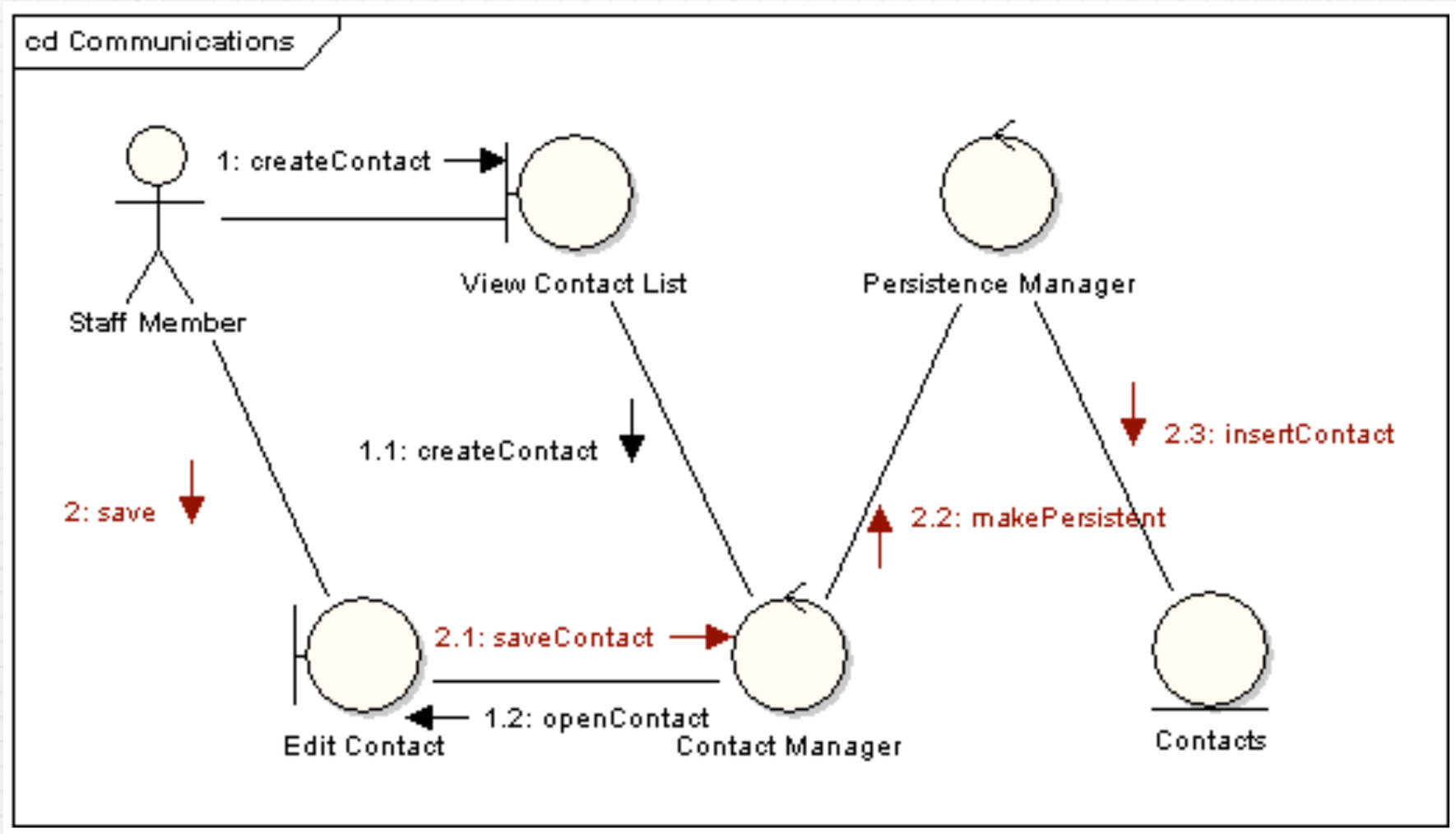
Package Diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, Package Diagrams are used to provide a visualization of the namespaces. The most common use for Package Diagrams is to organize Use-Case Diagrams and Class Diagrams, although the use of Package Diagrams is not limited to these UML elements.




http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_packagediagram.html

# Communication Diagram

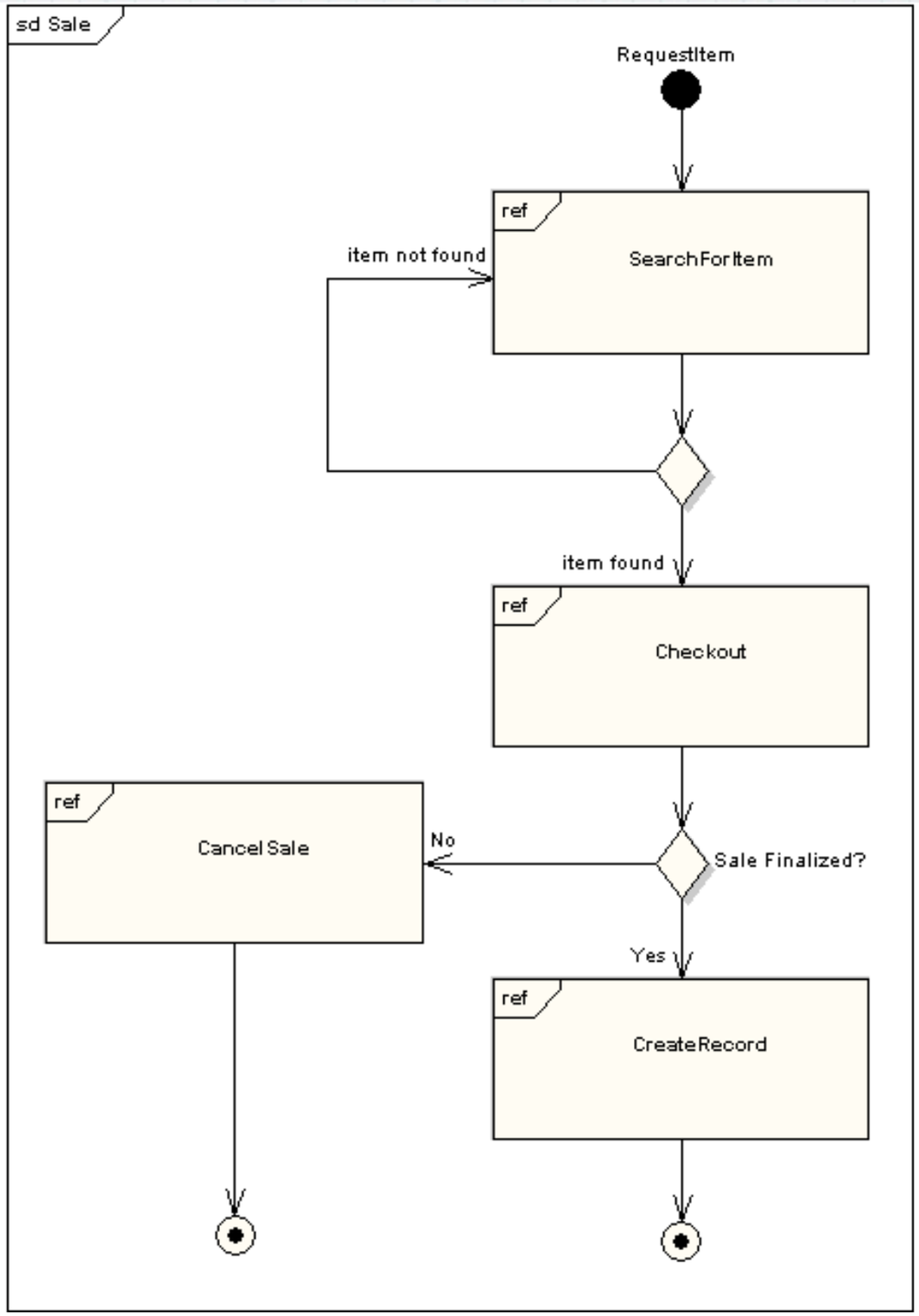
A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but its primary focus in on object relationships. On communication diagrams, objects are shown with association connectors between them. Messages are added to the associations and show as short arrows pointing in the direction of the message flow. The sequence of messages is shown through a numbering scheme.




http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_packagediagram.html

# Interactive Overview Diagram

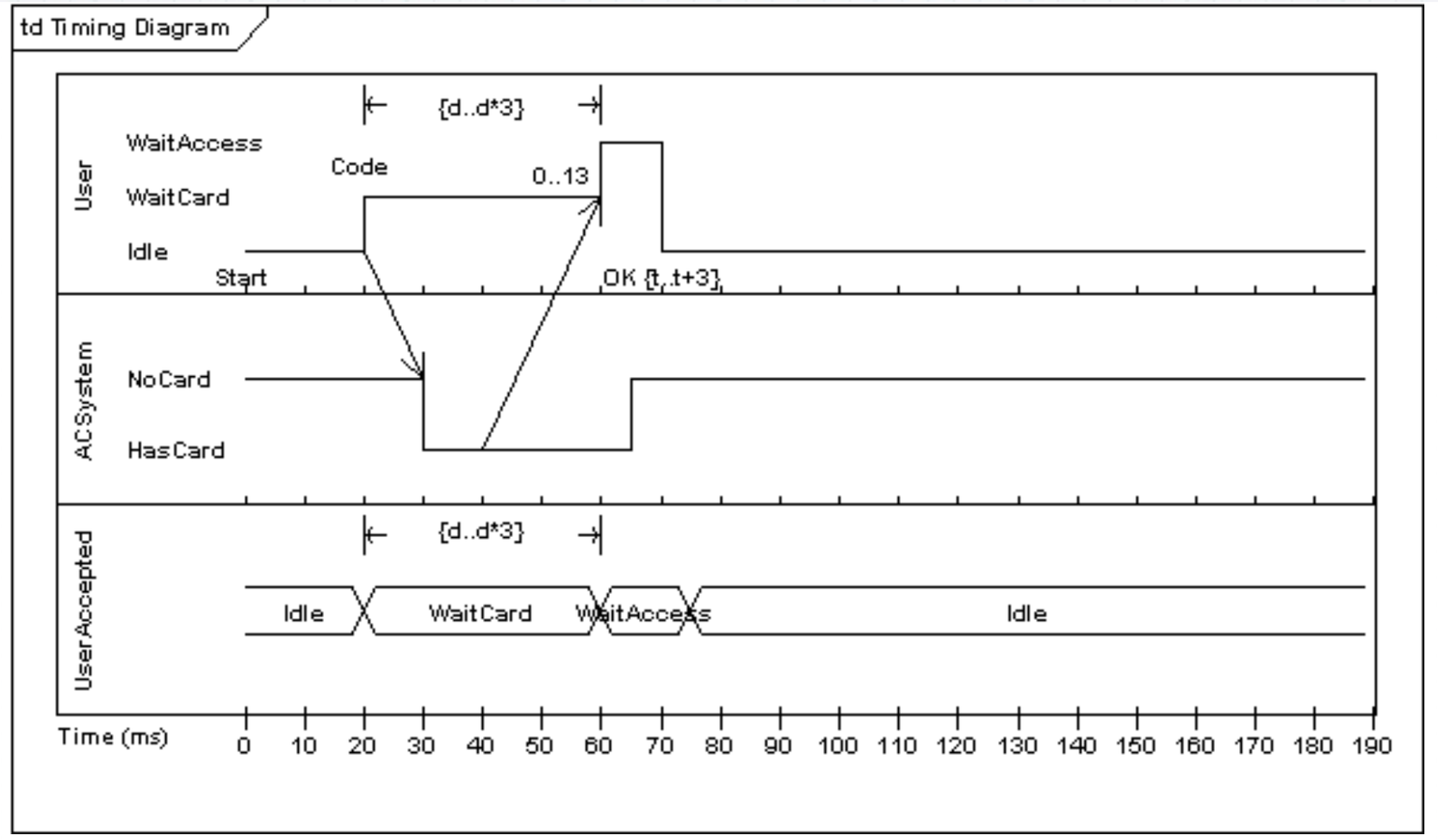
An Interaction Overview Diagram is a form of activity diagram in which the nodes represent interaction diagrams. Interaction diagrams can include sequence, communication, interaction overview and timing diagrams. Most of the notation for interaction overview diagrams is the same as for activity diagrams, for example initial, final, decision, merge, fork and join nodes are all the same. However, interaction overview diagrams introduce two new elements, interaction occurrences and interaction elements.




http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_packagediagram.html

# Timing Diagram

UML timing diagrams are used to display the change in state or value of one or more elements over time. It can also show the interaction between timed events and the time and duration constraints that govern them.




http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_packagediagram.html

# State Machine Diagram

A state machine diagram is a graph that represents a state machine. States are rendered by appropriate state symbols, while transitions are generally rendered by directed arcs that connect them or by control icons representing the actions of the behavior on the transition.

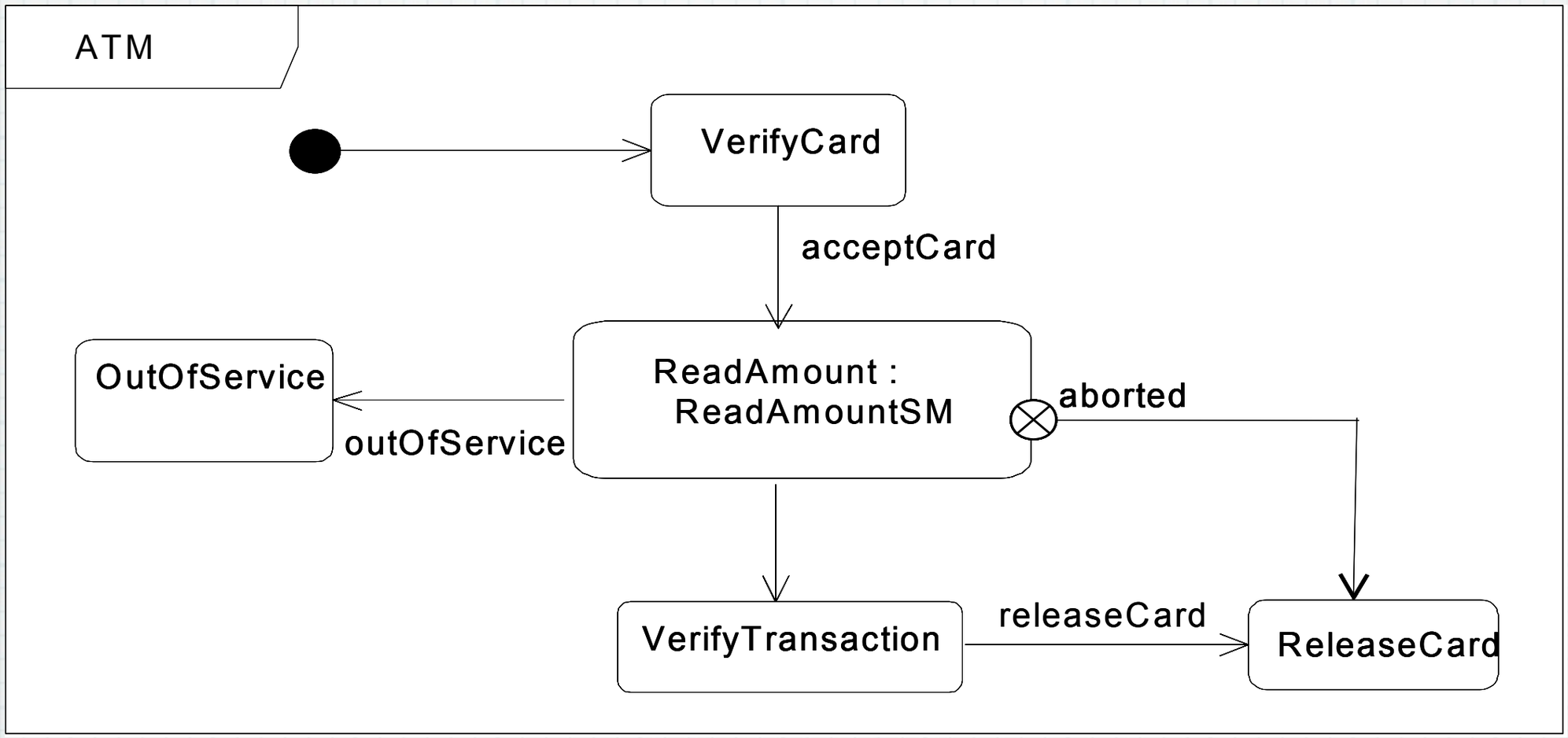



**Figure 15.48 - SubmachineState with usage of exit point**

**"Unified Modeling Language: Superstructure," Version 2.0, formal/05/07/04, http://www.omg.org**

# UML 2 Syntax Wrap-Up . . .

* This slide set is a good INTRODUCTION and OVERVIEW of the syntax and semantics of UML 2 diagrams

* To truly understand and master the UML 2 syntax requires extensive practice in using the language to document and analyze REAL projects with REAL teams

* Like any other LANGUAGE UML 2 is quickly and easily forgotten without some reinforcing practice