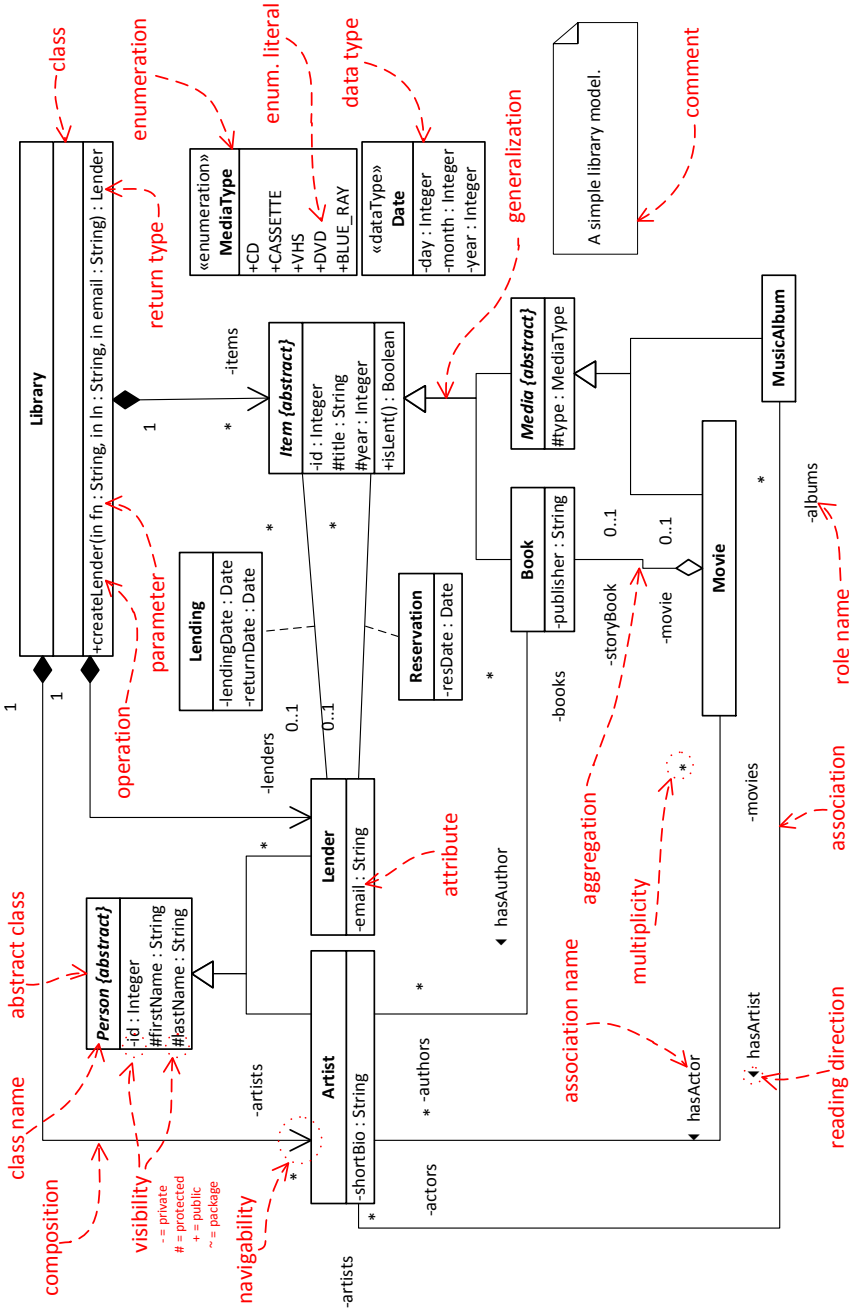
Class Diagrams



Naming Conventions

class, data type, and enumeration names Like Java classes: CamelCase, starting with a uppercase letter, no whitespace; singular nouns

attribute, parameter, role Like Java attribute/parameter names: camelCase, starting with a lowercase letter, no whitespace; nouns; multi-valued attribute, parameter and role names in plural form

operation names Like Java methods: camelCase, starting with a lowercase letter, no whitespace; operation names as verb-object phrase

association names camelCase, starting with a lowercase letter, no whitespace; verbs or verb-object phrases

enumeration literals All capitals, starting with a letter; compound words may be separated with underscores; no whitespace

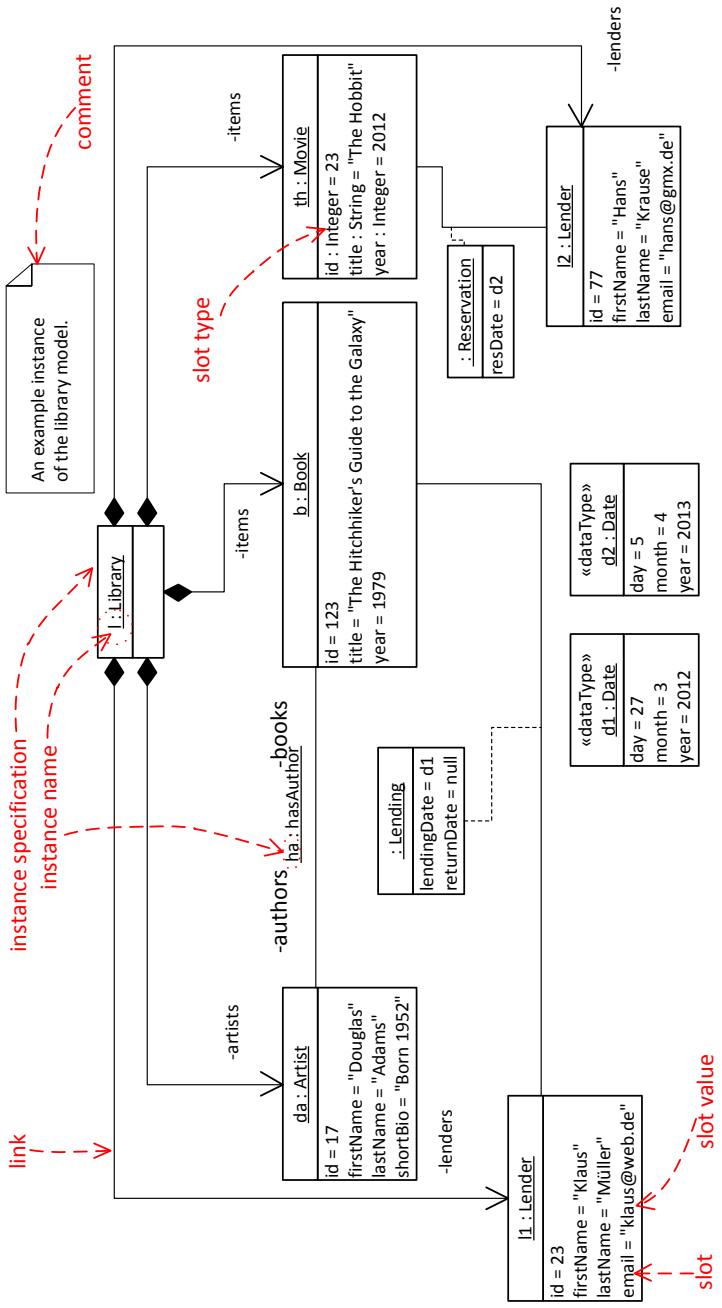
Rules

- **Class names must be unique**. If there a two classes of the same name in a diagram, those denote the very same class (and thus must not contradict each other).
- **Association names must be unique**.
- **For every class, all attribute names and far-end role names must be unique**. Inherited attribute names and role names have to be considered, too.
- **Every association must have a name or at least one role name.**
- **Use {`abstract`} constraint for abstract classes.** The UML allows for visualizing abstract classes by using an italic font for the class name, but this is harder to recognize.

Tool Compromises

- Few tools support annotating the reading direction of associations. Preferrably, use an association names that makes the direction clear. It is also allowed to misuse navigability arrows for the reading direction, but this has to be made explicit in a comment!
- Not all tools support the {`abstract`} constraint but only use an italic font for names of abstract classes. In that case, it is allowed to use the (outdated UML 1.x) stereotype `<<abstract>>` above the class name instead.

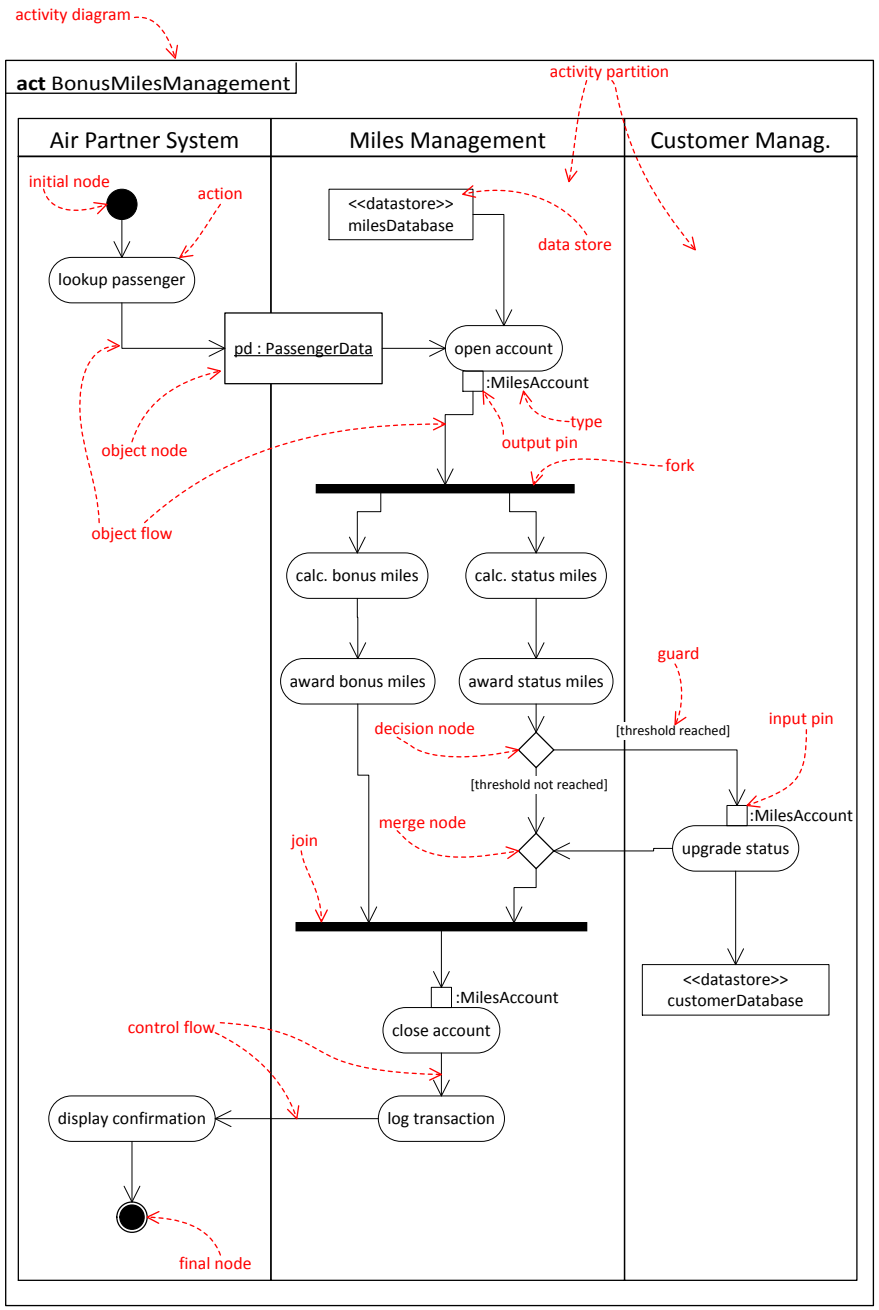
Object Diagrams



Rules

- **There may only be a slot if the instance's class (or a superclass thereof) declares such an attribute.**
- **There may only be a link if there is an association between the classes (or superclasses) of the instances.**
- The instance name is optional and may be omitted. However, **to be able to use a data type instance as slot value, the instance name has to be provided and unique**. (See the `Date` instances `d1` and `d2`, and their usage as slot value in the `Lending` and `Reservation` instances.)
- Showing the slot types is optional. (Compare instance `b` and instance `th` for example.)

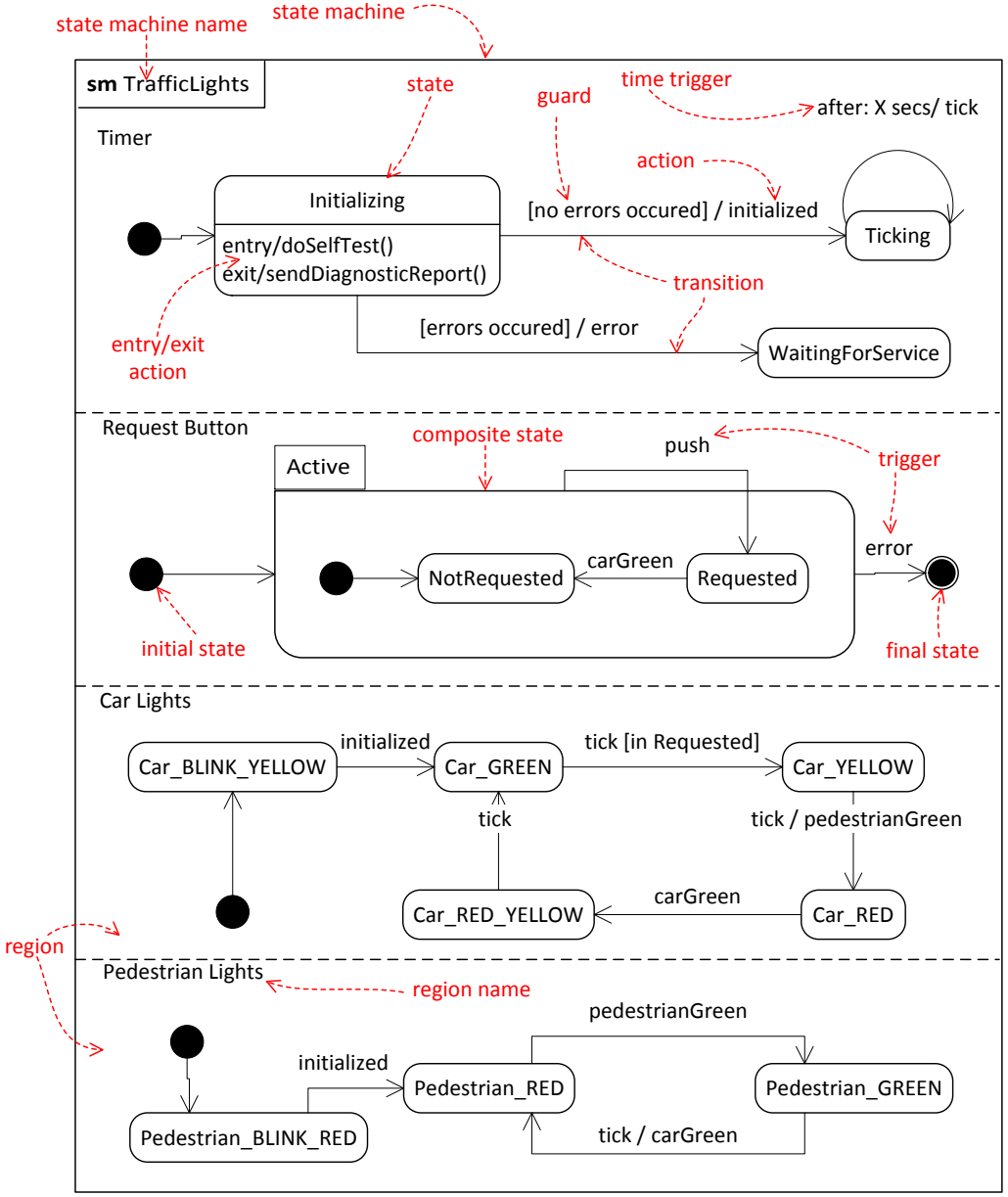
Activity Diagram



Rules

- Object flows between two actions with an intermediate object node, and object flows between pins of two actions are semantically equivalent.
- An object flow between two actions implies control flow.
- All control flows starting at some fork need to be joined again, except for those finishing in a *flow final node* (drawn as a crossed circle).
- If two or more flows start at an action, this is an implicit fork.
- Every flow starting at a decision node must have a guard, all guards must be exclusive/non-overlapping, and the set of guards should cover the complete range of values relevant for the decision. That means, there's always exactly one outgoing flow whose guard matches.

State Machine



Rules

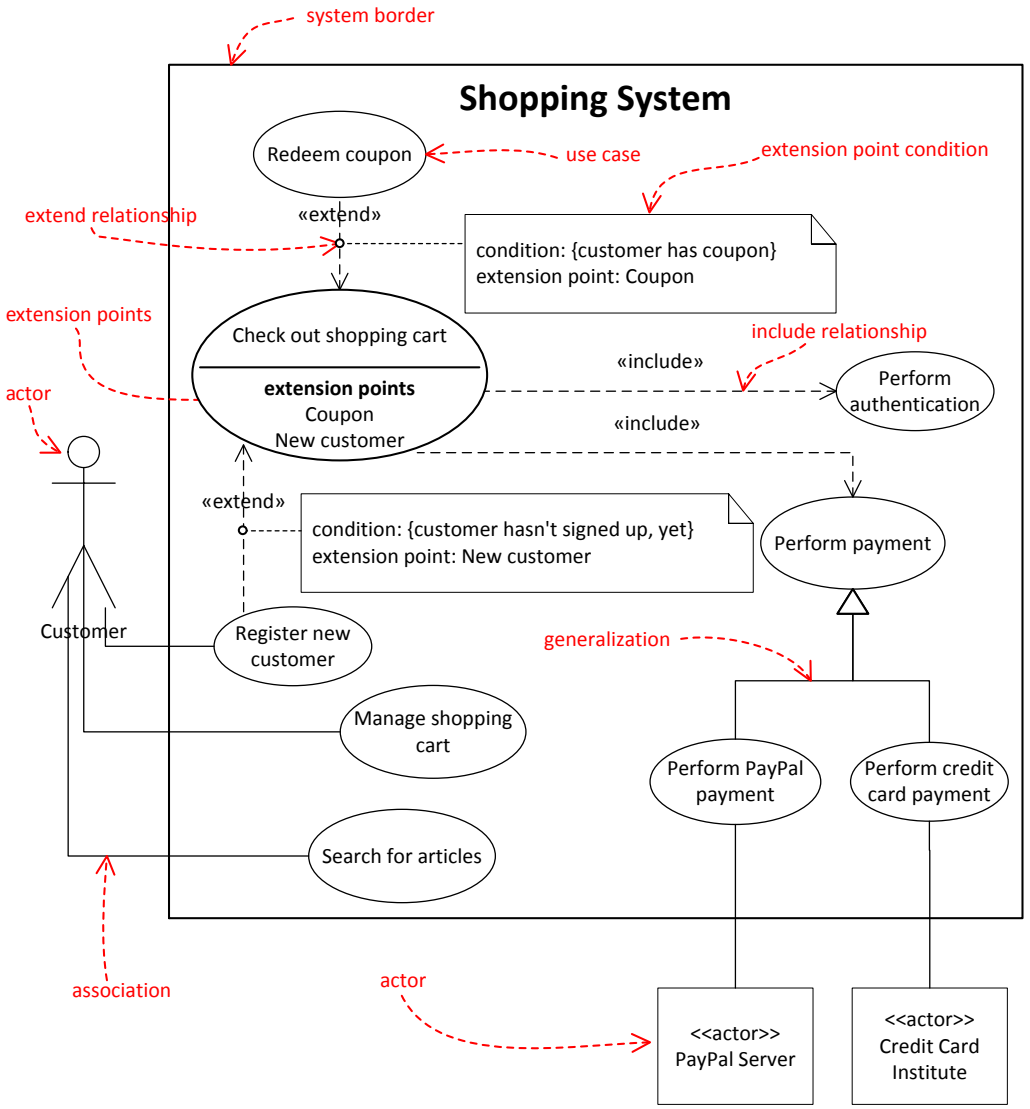
- There mustn't be triggers or guards at transitions starting from an initial state.
- The syntax of transitions is: *trigger [guard] / action*. All three are optional. Multiple comma separated triggers may be given. The transition fires on any of the specified triggers.
- States may have *entry*, *do*, and *exit* actions.
- Types of triggers:
  time trigger: Fires after a specified amount of time. Examples: *after(3 secs)* or *after: 3 secs*
  call trigger: Fires when a method is called. Example: *doStuff()*
  signal trigger: Fires when a signal is received: Example: *push*
- Types of actions:
  call action: A method is being called. Example: */ doStuff()*
  send signal action: A signal is being sent. Example: */ tick*
- Types of guards:
  opaque expression: An arbitrary condition in (pseudo-) code. Example: *[no errors occured]*
  orthogonal state test: It is tested if some other state in a orthogonal region is active. Example: *[**in** Requested]*

History Pseudostates

History pseudostates are missing in the example, so here's a brief explanation.

- A *shallow history pseudostate* (H) may be added to some composite state (such as *Active* in the example). Whenever a transition leads to this state, the last recently active state (or the last recently active states in each region) inside it becomes active again. If there are composite states contained in the state with the deep history, their active states are determined by the transitions from their initial states.
- A *deep history pseudostate* (H*) may be added to some composite state (such as *Active* in the example). Whenever a transition leads to this state, the last recently active state (or the last recently active states in each region) inside it becomes active again. If there are composite states contained in the state with the deep history, their last recently active states are activated recursively.
- There must be at most one history pseudostate in any composite state.
- There may be a default transition from a history pseudostate to some state. This transition is triggered only in case the surrounding composite state containing the history pseudostate has never been active before.

Use Case Diagrams



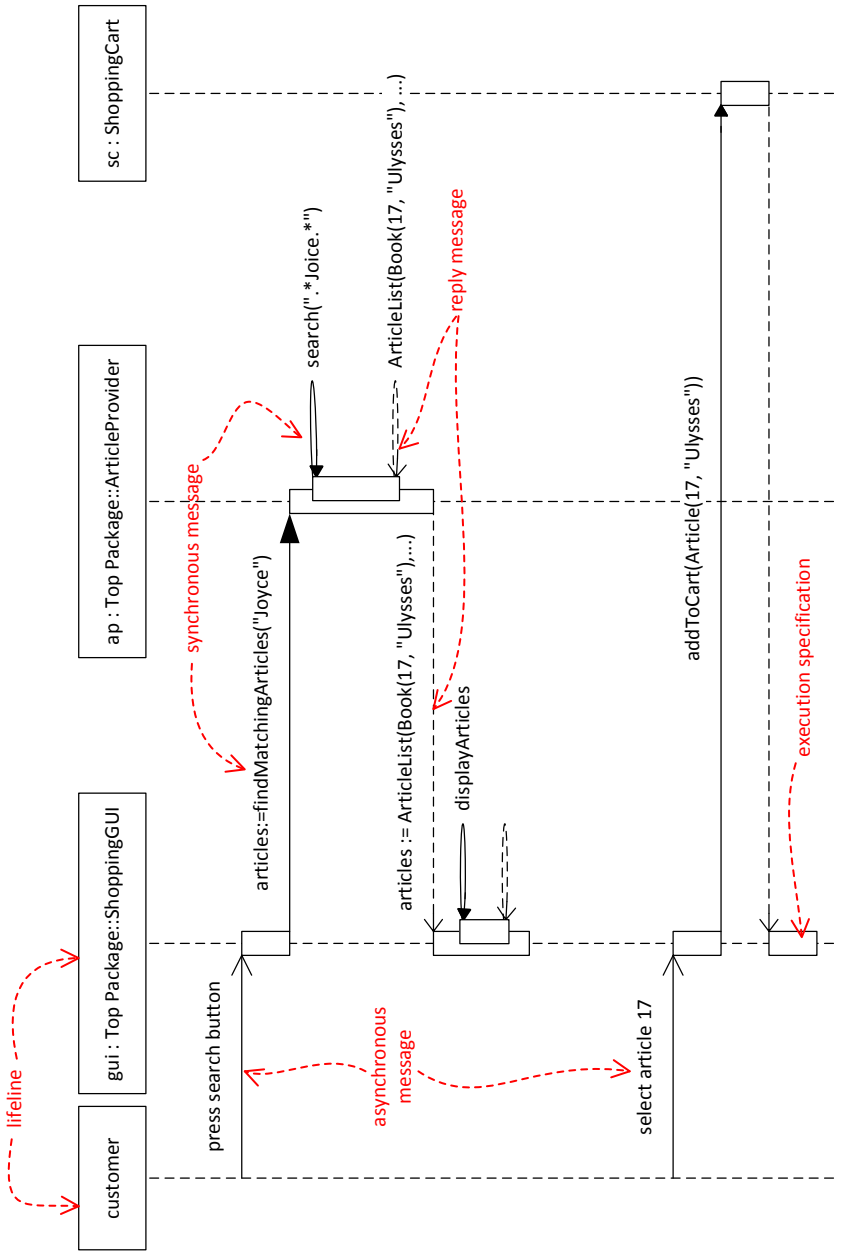
Naming Conventions

- Use cases should be named with verb-object phrases.

Rules

- Actors are only outside the system border, use cases are only inside.
- Associations can only connect actors to use cases, but they cannot connect two actors or two use cases.
- Use case names may alternatively be written below the use cases instead of in the use cases.
- Use case diagrams shouldn't be overly detailed. Usually, there shouldn't be more than 9 use cases.
- Generalizations are supported for both use cases and actors.
- Associations always connect an actor to a use case.
- Associations may also have a name, multiplicities, and role names, although those are usually omitted.
- Non-human actors may also be notated with a computer icon instead of the stick-man or rectange with `<<actor>>` stereotype.

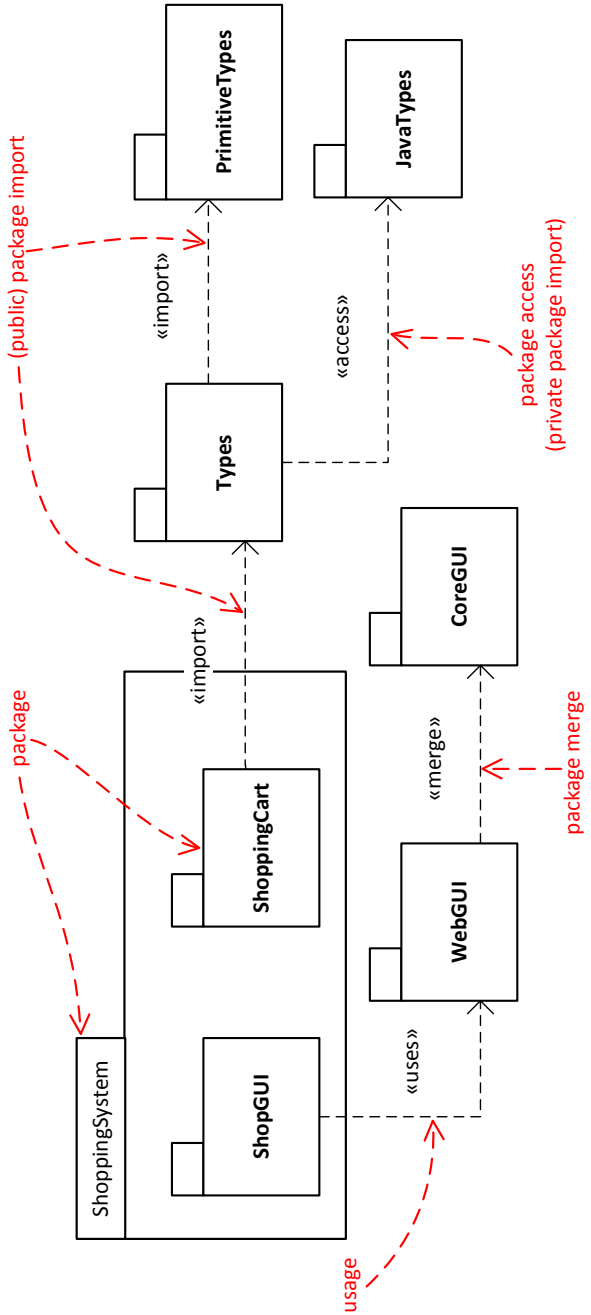
Sequence Diagrams



Rules

- **Every synchronous message needs to have a reply message.**
- **Synchronous messages usually correspond to operation calls.** Because we usually model a concrete scenario without alternatives using sequence diagrams, synchronous messages should be annotated using a Java-like method call notation with concrete values as parameters. Conversely, a reply message should be annotated with a concrete return value.
  The `articles:=` and `results:=` in the example diagram is optional and not needed, but it's only an artifact of the tool used to model the diagram. Similarly, the RSA defaults to include the operation name also in the reply message which is verbose but ok.
- There must not be any intermediate messages between a synchronous message and its reply message. (When a method is being called, the caller blocks execution until the called method has returned.)
- **Asynchronous messages don't have reply messages.**
- Execution specifications are optional.

Package Diagrams



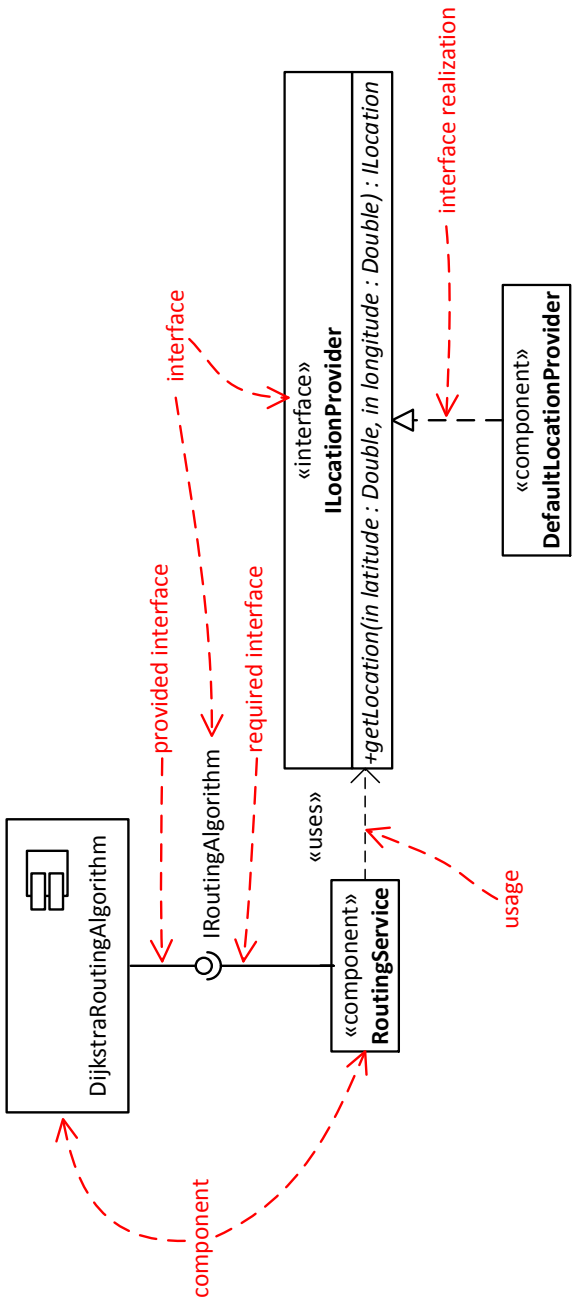
Rules

- There may be no cycles in package imports (no matter if public or private) nor in package merges.

Tool Compromises

- Standard UML shows package containments by embedding the subpackages inside the containing package. Many tools don't allow this and provide a visual connector ⊕—— that is used to connect a package with its subpackages.

Component Diagrams



Rules

- The `<<component>>` stereotype and the component icon are equivalent.
- In essence, the lollypop notation (using ball/socket) is equivalent to the more detailed notation that visually represents interfaces – similar to classes – as rectangles with the `<<interface>>` stereotype. Components can realize interfaces and use other interfaces.