Introduction to the Unified Modeling Language

Perdita Stevens, University of Edinburgh

May 2010

Agenda

- Use cases
  - textual representation
  - basic use case diagram as summary of use cases
- Class diagrams
- Sequence diagrams
  - realising use cases
  - illustrating protocols etc.
- State diagrams, activity diagrams
- *And the rest... deployment diagrams, component diagrams, object diagrams, timing diagrams, etc.*
- *OCL and alternatives (programming language, English, informal mathematics, formal specification lang.)*
- *Ways of modelling: agile modelling, executable modelling, model-driven development, DSMLs*

Approach

UML is a large language...

Today we'll cover the absolute essentials.

Next time, we'll add some of the "nice to know" stuff.

General warning: it is as hard to write correct UML as correct Java, but there's no compiler to tell you you've made a mistake. *Many* books and websites contain incorrect UML – does it matter? Sometimes...

Resources for learning more

Online resources are a mixed bunch.

Lots of good books...

The OMG standard itself `http://www.uml.org`

There is still a lot of material out there relating to UML1.x, and UML2 is different in many ways.

Use cases

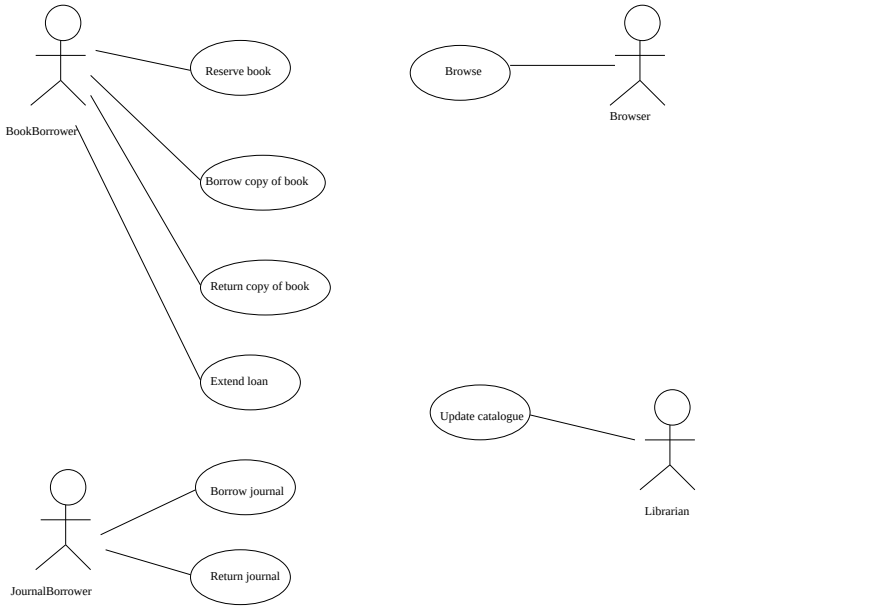
document the behaviour of the system *from the users' points of view*. They help with three of the most difficult aspects of development:

- capturing requirements
- planning iterations of development which are good for users
- meaningful system testing

First introduced by Ivar Jacobson (early 90s), developing from *scenarios*. Independent of OO – strength or weakness??

Simple use case diagrams are easy to understand: can be useful for communication between customers and developers.

A simple use case diagram



Textual use cases

The diagram is just a summary. Usually behind each use case is a structured textual description e.g. with sections:

- Use Case Name – short, descriptive
- Goal in Context – sentence
- Preconditions
- Success End Condition – to hold normally
- Failed End Condition – to hold even on failure
- Primary Actor(s)
- Secondary Actor(s), if any –
- Trigger – could be a time event
- Main success scenario – sequence of numbered steps
- Extensions – exceptional and failure cases

Recommendation: *Writing Effective Use Cases*, Alistair Cockburn.

Actors

An **actor** – shown as a stick figure – can be:

- a human user of the system *in a particular rôle*
- an external system, which *in some rôle* interacts with the system.

More specifically, a particular *kind* of user. E.g, bank has many customers, but we only show one Customer actor on the diagram.

The same human user or external system may interact with the system in more than one rôle: he/she/it will be (partly) represented by more than one actor. (e.g., a bank teller may happen also to be a customer of the bank).

What is a use case?

A **coherent work unit** of the system which has value for an actor, e.g. `Borrow copy of book`.

Shown on diagram as named oval.

Also includes (textual) description of the (a?) sequence of messages exchanged between the system and any actors, and actions performed by the system, in order to realise the functionality.

Connection between use case descriptions and other forms of requirements documentation is rather controversial.

Use cases: scope and connections

A use case:

- may include logic to handle unusual or alternative courses, e.g. "if the `BookBorrower` has the maximum number of books on loan already, refuse this loan" *even though these may result in the actor being unsatisfied.*
- may be associated with other UML models which show how it is realised.

A use case diagram summarises all the tasks performed by the system (or subsystem, etc.)

Requirements capture

Use cases can help with requirements capture by providing a structured way to go about it:

1. identify the actors
2. for each actor, find out
   - what they need from the system
   - any other interactions they expect to have with the system
   - which use cases have what priority for them

There may be aspects of system behaviour that don't show easily show up as use cases for actors.

Analysis vs design

Some actors are part of the requirements: usually the ones who derive benefit from a use case.

Others are part of the (business process) design: the ones who interact with the computer system to provide the benefit.

For example, consider a `FindBook` use case of a library, in which the user enters details of a book and wants to end up with a copy of it. Maybe the system will give the user directions to where the book is on the shelf. Maybe it will alert a librarian to go and fetch it. In the latter case, should the librarian be shown as actor? In some sense, the choice is a design decision.

Using use cases in development

Use cases are a good source of system tests: requirements documented as desired interactions, which translate easily into tests.

They can also help to validate a design. You can walk through how a design realises a use case, checking that the set of classes provides the needed functionality and that the interactions are as expected.

Use cases are not limited to documenting the whole system: they may describe any classifier, e.g. subsystem, class, COMPONENT.

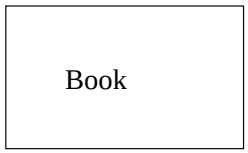
What use cases are not

Use cases document the requirements of a system: not the whole business process into which the system fits.

For example, UML does not permit associations between actors: you cannot legally use a use case diagram to show an interaction between two humans followed by one of them using a system. (E.g. can't legally show librarian and library member as separate actors in Borrow Book, if only the librarian interacts directly with the system.)

There are extensions to UML to allow business process modelling, not considered here.

A class



A class as design entity is an example of a **model element**: the rectangle and text form an example of a corresponding **presentation element**.

UML explicitly separates concerns of actual symbols used vs meaning.

An object

This pattern generalises: always show an instance of a classifier using the same symbol as for the classifier, labelled instanceName : classifierName.

Classifiers and instances

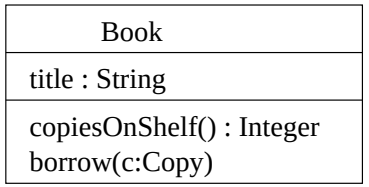
An aspect of the UML metamodel that it's helpful to understand up front.

An **instance** is to a **classifier** as an object is to a class: instance and classifier are more general terms.

(In the metamodel, Class inherits from Classifier, Object inherits from Instance.)

We'll see many other examples of classifiers.

Showing attributes and operations



Syntax for signature of operations (argument and return types) adaptable for different PLs. May be omitted (together) – but the formal parameter name is compulsory when the argument list is given(!)

Compartments

We saw the standard:

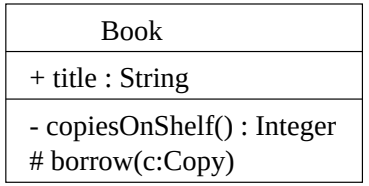
- a compartment for attributes
- a compartment for operations, below it

They can be suppressed in diagrams.

They are omitted if empty.

You can have extra compartments labelled for other purposes, e.g., responsibilities.

Visibility

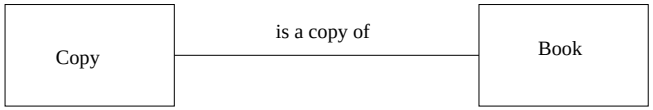


Can show whether an attribute or operation is

- public (visible from everywhere) with $+$
- private (visible only from inside objects of this class) with $-$

(Or protected ($\#$), package ($\sim$) or other language dependent visibility.)

Association between classes

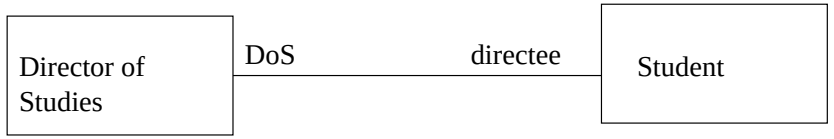


This generalises: association between classifiers is always shown using a plain line.

An instance of an association connects objects (e.g. Copy 3 of War and Peace with War and Peace).

An **object diagram** contains objects and links: occasionally useful.

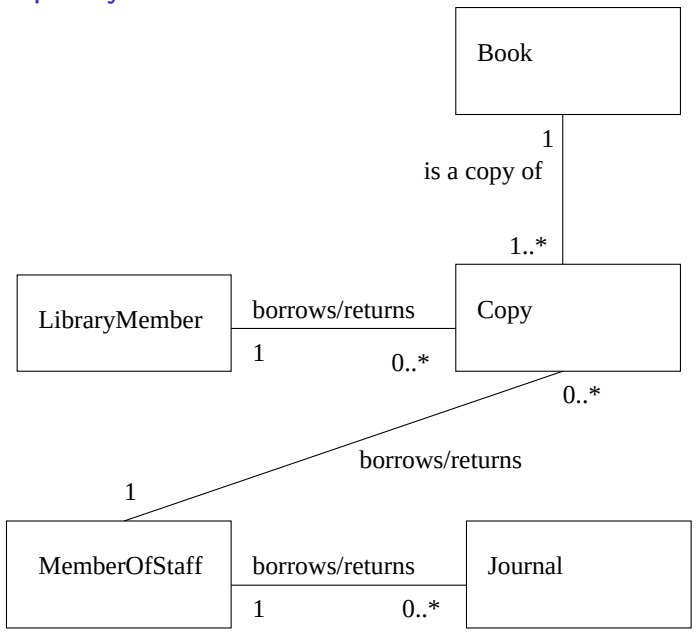
Rolenames on associations



Can show the role that one object plays to the other.

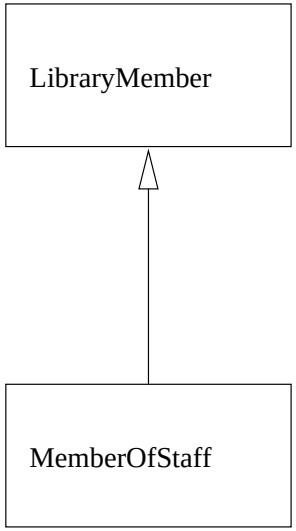
Useful when documenting the class: e.g. a *class invariant* for DirectorOfStudies could refer to the associated Student objects as self.directee (a set, if there can be more than one).

Multiplicity of association



Commas for alternatives, *two* dots for ranges, * for unknown

Generalisation



This generalises: generalisation between classifiers is always shown using this arrow.

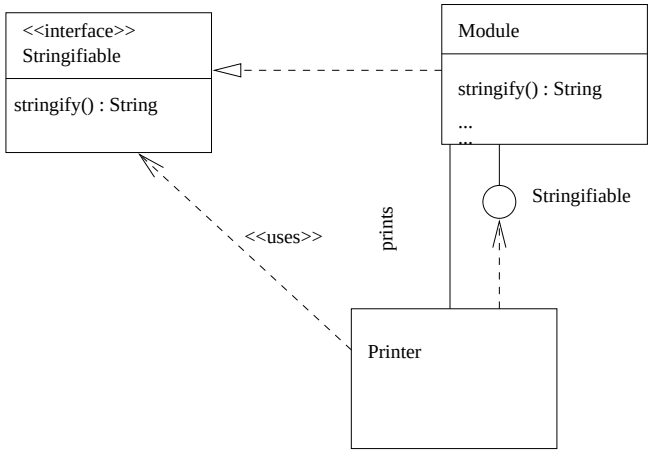
Abstract operations and classes

An operation of a class is abstract if the class provides no implementation for it: thus, it is only useful if a subclass provides the implementation.

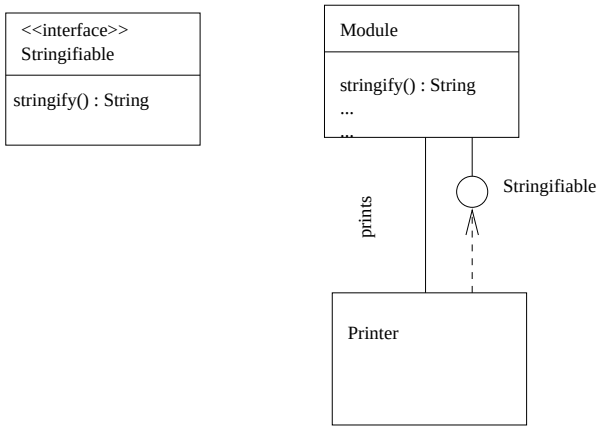
A class which cannot be instantiated directly – for example, because it has at least one abstract operation – is also called abstract.

Can show *abstract* operation or class using italics for the name, and/or using the *property* {*abstract*}.

Interfaces

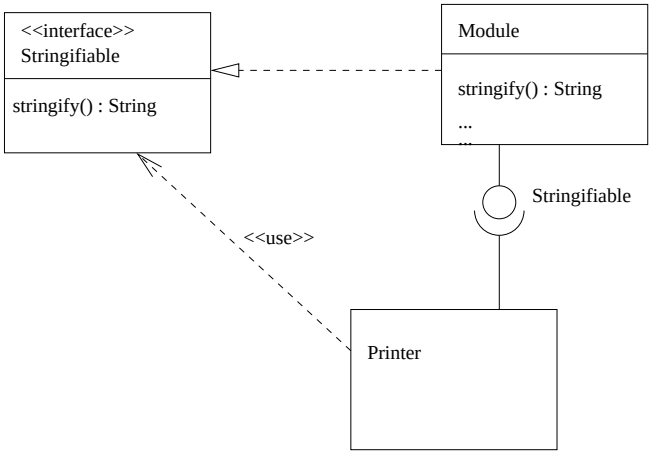


Simpler diagram: WRITE ONCE



Many things other than classes can realise interfaces: can use the lollipop symbol on e.g. components, actors.

Interfaces, newer notation



Designed relationships between classes

So far we've dealt with what Fowler calls the *conceptual model*. We've identified the key domain abstractions and the conceptual relationships between them.

Here we look at more advanced features of class models, especially at how to record information pertaining to the design.

Dependencies

To make the system maintainable we want to minimise the dependencies between parts of the system. Not all "real world" connections are reflected in the system.

A is dependent on B if a change to B may force a change to A.

What counts as a change is context-dependent.

Aim to avoid complex dependencies especially circular ones.

Dependencies in UML

Dependencies are shown using a dotted arrow:

We've seen them between use cases and between a class and an interface: used generally for "relationship not otherwise specified".

Note that some dependencies are implied, and need not be repeated: for example any class depends on its superclasses.

Circular dependencies

going through more than one "sensible reuse unit".

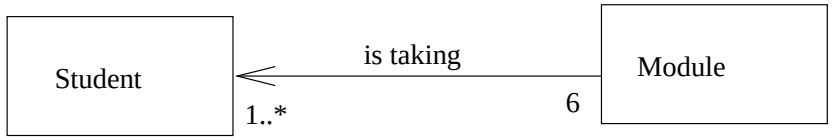
E.g. (from Webster p228):

> *graphical rendering subsystem uses calls from text rendering subsystem uses calls from line layout subsystem uses calls from graphical rendering subsystem.*

It's quite easy to get this by accident: one reason why we delay deciding in which direction our associations are navigable.

Problem: if this happens you can say goodbye to reuse. It's also very hard to understand.

Navigability



When should the navigability of an association be decided?

Some experts believe vehemently that you should never identify an association without deciding its navigability. Others disagree.

"As early as possible, but no earlier."

An aggregation relationship

Non-exclusive part relationship.

A common fault is identifying too many aggregations. If in doubt use plain association.

An composition relationship

Exclusive part relationship.

Interaction diagrams

describe the *dynamic* interactions between objects in the system, i.e. the pattern of message-passing.

Two main uses:

- Showing how the system realises [part of] a use case
- Showing how an object reacts to some message

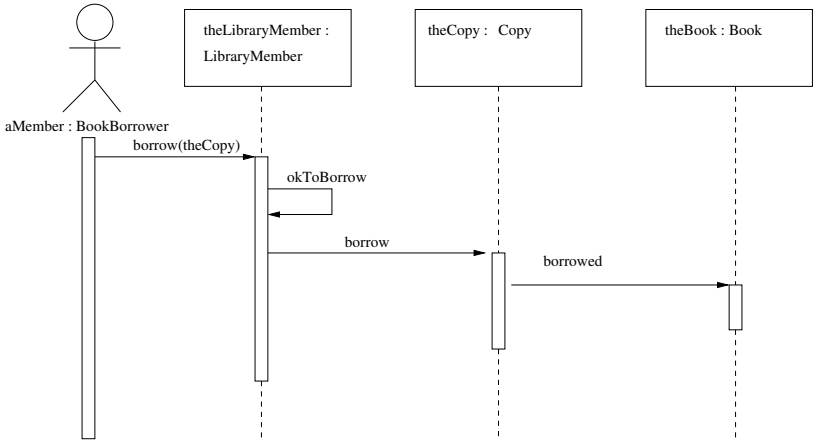
Particularly useful where the flow of control is complicated, since this can't be deduced from the class model, which is static.

UML has two sorts, *sequence* and *communication* diagrams – the differences are syntactic.

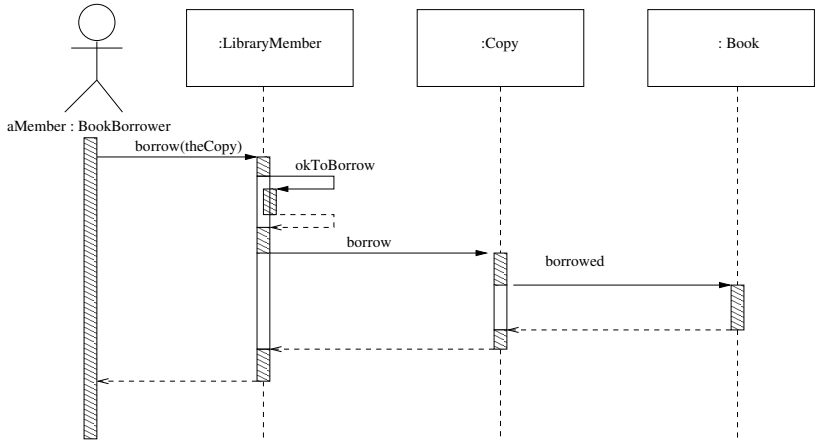
Developing an interaction diagram

1. Decide exactly what behaviour to model.
2. Check that you know how the system provides the behaviour: are all the necessary classes and relationships in the class model?
3. Name the objects which are involved.
4. Identify the sequence of messages which the objects send to one another.
5. Record this in the syntax of a sequence or communication diagram.

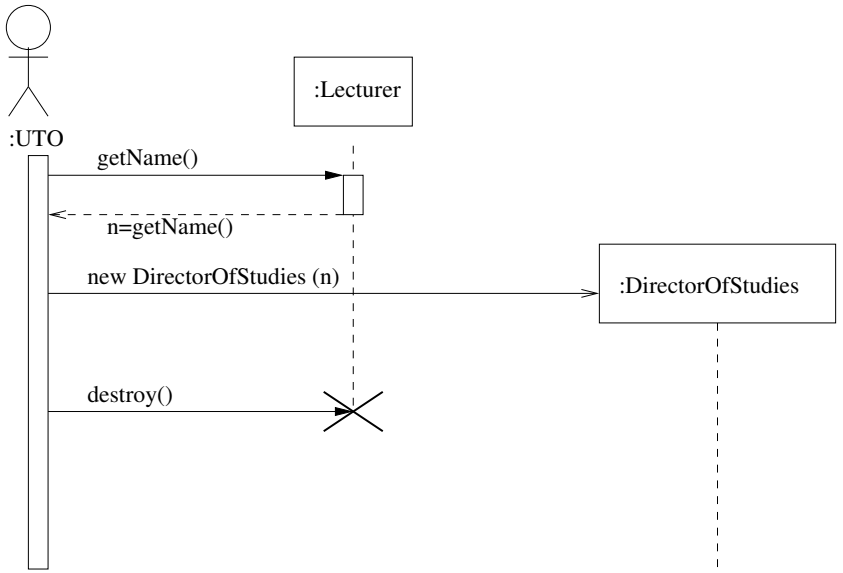
Sequence diagram



Showing more detail



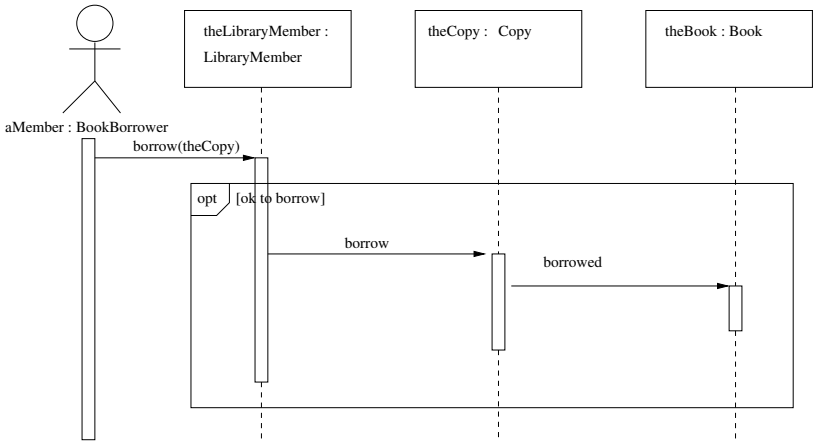
Creation/deletion in sequence diagram



Modelling asynchronous communication

So far we've only shown single-threaded synchronous behaviour – messages are sent and replied to. Can also use sequence diagrams to show asynchronous communication, using different arrowheads.

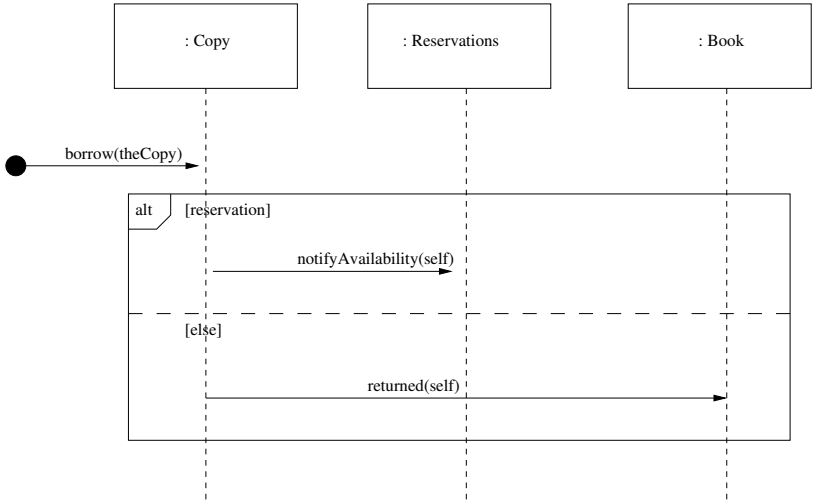
Interaction type	Symbol	Meaning
Synchronous or call	–▶	The 'normal' procedural situation. The sender loses control until the receiver finishes handling the message, then gets control back, which can optionally be shown as a return arrow.
Return	<– –	Not a message, but a return from an earlier message. Unblocks a synchronous send.
Asynchronous	→	The sender does not lose control; it sends the message and may continue immediately. The recipient of the message may also become active, if it wasn't already.

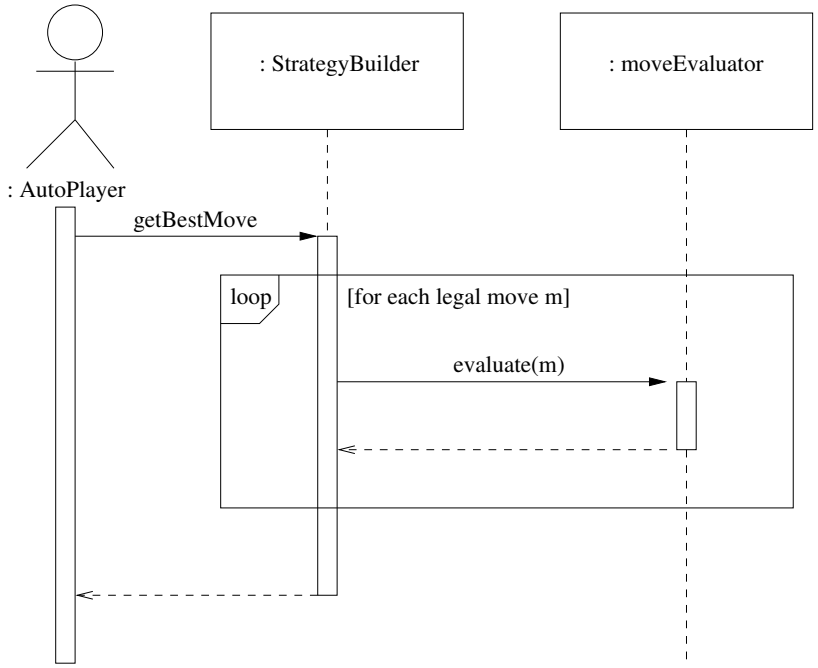


Showing more than one scenario

Sequence diagrams are most useful for showing just one, linear, sequence of events – an example of what may happen, not a full description of everything that could happen.

Sometimes you want to go further. UML1.x had some ad hoc notation for it involving splitting object lifelines etc. UML2 does this more systematically with *fragments*. We'll just see a few examples...





Effects of interactions on objects

So far we've seen how to model:

- the requirements of the system with use cases
- the structure of the system with a class model
- the interactions between objects with interaction diagrams

Interactions describe how an object reacts to an event that forms part of that particular interaction. ("What happens next?")

But what determines this? In particular, the same object may react to the same event in different ways, depending on its internal state.

We model this using state diagrams.

State diagrams

Useful for showing the way that an object of a given class changes state, if it has qualitatively different internal states. May include:

- states
- events that cause transitions between states
- guards that must be true for a transition to take place
- actions that are caused by a given transition
- activities that take place when in a certain state
- start and end markers

States *may* be nested - but most classes will not even need statecharts.

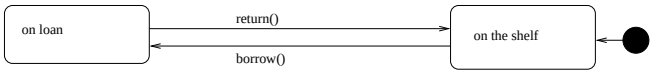
Protocol state machines

The simplest – and most useful – state machines are called protocol state machines because they document a protocol for interacting with an object.

E.g. what sequences of messages make sense?

They show how an object's state changes on receipt of messages, but not what the object does to the rest of the system, e.g. what messages it sends.

A simple state diagram



What are the states?

If we could draw infinite[1] diagrams, we could represent each set of values for an object's attributes and links as a separate state and show exactly what happens when the object receives each message. We could include as much detail as the code.

In practice, a state represents an equivalence class of attribute (and link) values: objects which behave *qualitatively* the same way are in the same state.

That is, a real state diagram represents an abstraction of the "ideal" (and useless!) state diagram.

[1] OK, computers are finite...

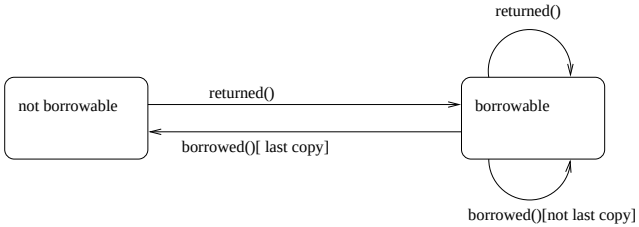
Transitions in more detail

When an object passes from one state to another it does so as a result of an event, e.g. receiving a message. In addition to changing state, the object may react in some way e.g. by sending a message. Such (re)actions are shown after the slash: event/action.

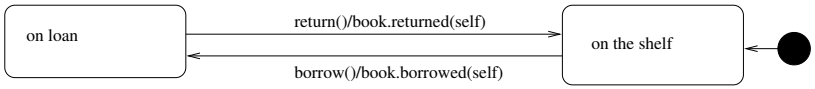
Sometimes an event causes a state change only if a guard is satisfied. The guard is shown: event[guard] / action.

> An event is something done to the object:
> an action is something the object does.

State diagram for Book, with guards



State diagram for Copy, showing actions



Cautionary note

It is possible to use state diagrams with actions very heavily to define the behaviour of whole systems, by having a state diagram for each component.

UML's semantics of state diagrams is *run-to-completion* which roughly means that when an event is processed, all the transitions caused directly or indirectly by that event must happen, before the next event is processed.

However, looked at in detail the UML semantics do not actually make much sense where these actions are themselves synchronous messages...

Advice: either stick to asynchronous actions, or treat actions as comments.

Activity diagrams

Useful as an alternative to interaction (sequence or communication) diagrams for:

- detailing a use case
- explaining an an object's reaction to a message

Also useful for showing the dependencies between use cases: e.g. *workflow* of an organisation.

Pros and cons of activity diagrams

Advantages:

- Can show parallel activities, so make dependencies and non-dependencies explicit: avoid premature design
- Much the best way to document dependencies between use cases.

Disadvantages:

- Not automatically clear who/what carries out an activity; can be hard to make the connection with underlying objects...
- ... swimlanes attempt to help

Activity diagram - BOX SHAPE WRONG