Software Engineering 2

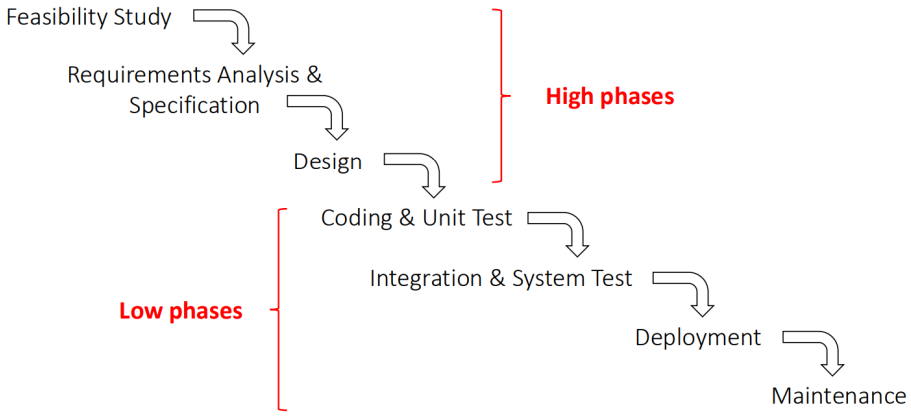
Course notes

Andrea Bellani

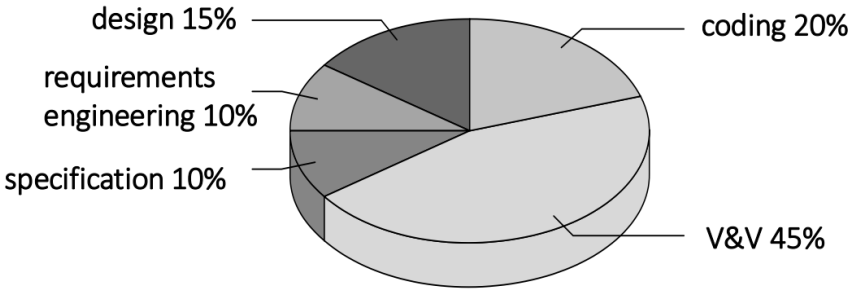
Introduction Without a doubt, the development of complex software systems requires proper skills in identify, organize and document each aspect of the systems: a correct and complete documentation is essential for having a correct, complete and maintainable large-scale system. Thus, in this course we will focus ourselves on understanding how to deal with the problems related to the software development, from the customer requirements to the final implementation and maintenance.

However, bear in mind that most of the time the effort does not end up with the release, because a software have to stay on the market and cope to its future needs, so even the time the software takes to response to the change requests is also a concern (*timeliness*).

We will follow the "basic" workflow of the software development however, most of the times the phases that we see in sequence are overlapped (e.g. the implementation starts even when the software design is not finished):



An estimation that might surprise someone is:



which is the estimated effort distribution in each macro-phase of the development of a complex software.

Contents

I Software development process

Software development process

1. The requirement analysis

1.1 Software properties

Accordingly to ISO/IEC 25010:2011, in a software product, *quality* is made up of:

and we always should take care of them all, but of course we have to understand the level which the customer wants them (e.g. usability often changes if user categorizes of our software change).

1.2 The requirements engineering

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended.

Basically, the purpose of the requirements engineering the purpose for which the system was intended. What generally happens is that our customers gives us a set of more or less detailed requirements concerning the aspect of the systems he wants. We can divide them into three macro-categories:

- *functional requirements* : describe the interactions between the system and its environment (the main goals the software has to realize);

- *non-functional requirements* : further characterization of user-visible aspects not directly related to the functions (e.g. availability, security);
- *constraints* (or *technical requirements*): requirements imposed regarding the ways of implementing the system.

1.2.1 What a "bad requirement" is

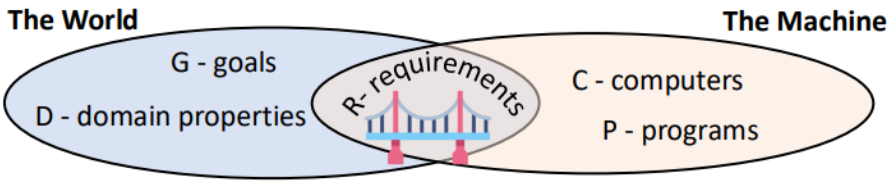
Formulating a requirement in a precise way is a concern of our customers, but understanding them is our concern! So, it is important to have clear in mind (as much as we can) whether a requirement gives all the information we need on the regarding aspect. So, "bad requirements" are all the requirements that:

- are vague or incomplete;
- are conflicting each other;
- can't be tested properly: this is probably our biggest concern, we always need a way to test the completeness of our solution;
- mix up different aspects;
- are related with the user responsibility: once the user has been warned of what he should not do with our system, what he does, as long as it does not affects the security of our system, is not a concern for us;
- are not reasonable (also considering the available resources).

In addition, we have also to take to consideration that our customer could emphasize some properties of the already existing system while minimizing others (there are also other more specific problems, e.g. the *Probe Effect*). At the end, we should always combine multiple strategies when it comes to study the already existing system.

1.3 Our system in the "universe"

Our system has of course to interact with other entities that from its point of view are nothing more that black-boxes (even users):



- *goals*: what our system has to provide to the *World* formulated in terms of world phenomena (e.g. *process a public call for an ambulance*);
- *domain assumptions*: what we take for grounded about the *World*;
- *requirements*: what our system has to provide to the *World* formulated in terms of shared phenomena (*process a call encoding*).

This to highlight the fact what is around our system is even more complex than the system itself, so it is foremost to identify precisely the boundaries of our system and the relationships with the World (with each identity of the World).

This complexity is also the reason why the requirements engineering workflow is a long phase:

1. discovery: we listen each stakeholder but also study the existing systems (this includes also studying their documentations);
2. modeling: we model our assumptions on the given requirements;
3. documenting: we produce a requirements document complete of all goals, domain assumptions and requirements interest the system (*RASD: Requirement Analysis and Specification Document*).

The identification of goals, domain assumptions and requirements is the one AND ONLY matter of the RASD, implementation choices, considerations on architectures and any other matter is not part of requirements engineering (unless there are some precise constraints, for example, *the application must be a web app*).

1.4 Completeness of requirements

Given:

- the set of requirements: R;
- the set of domain assumptions: D;
- the set of goals: G;

$$\text{R is } \textit{complete} \Leftrightarrow R \wedge D \models G.$$

So, this means that if their something that not satisfies G, it must not satisfy $R \wedge D$.

1.5 From concrete scenarios to use-cases

1.5.1 A scenario

A *scenario* is a narrative description of what people see and experience. They are basically concrete examples of a situation that may happen in our system, [METTERE DOMANDE DA FARE AL CLIENTE PER DEFINIRE GLI SCENARI]

1.5.2 Use-case

1.5.2.1 General structure of a use-case

Given a scenario (or a set of scenarios), an *use-case* is a formalized and generalized description of a set of them. It does include:

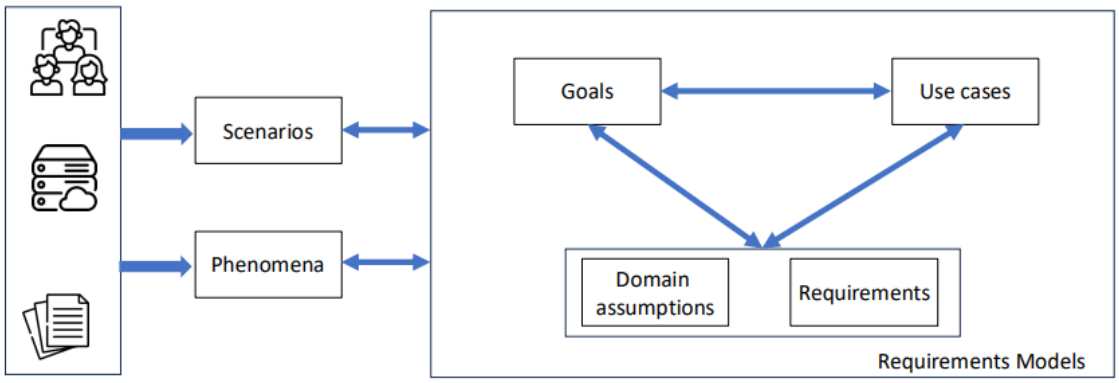
- actors: which are the generalized entities that participate in the use-case (e.g. *the lawyer, the accountant, the plumber...*);
- entry conditions: what makes the use-case happen (e.g. it's the day to draw the budget up);
- flow of events: the sequence of steps to perform once the entry condition is satisfied;
- exit conditions: what makes the use-case finish (e.g. *the chief accountant sends the budget to the director*);
- exceptions: exceptional cases that can happen during the regular flow of events;
- any other special requirement: non-functional requirements and technological constraints.

Of course, use-cases can also mention each other (e.g. *if the accountant discovers an errors in the budget it falls into the scenario "Discovering a budget mistake"*). Often, we define a use-case for each *user-transaction* (a an atomic sequence from the triggering of an action to the formulation of an answer).

It seems intuitive but having both comprehensible and complete use-cases can be harder in complex situations, here we put some frequent mistakes in the design of use-cases:

- using name instead of verbs: use-case are basically relationships between actions and events;
- forgetting the mention secondary actors;
- mentioning the system as an actor: system is always an implicit actor;
- making too long use-cases instead of splitting them into smaller ones;
- designing use-cases that not lead to any requirement;
- not mentioning the actor that performs an action;
- not making clear the casual relationship between steps.

But at the end, the requirements engineering is a more circular activity:



1.6 Modeling requirements

How should we formulate our requirements? Often we choose between:

- a semi-formal description: by using semi-formal defining languages such as UML accompanied with textual specifications whether necessary;
- a formal-description (which can be verified and validated automatically): by using formal definition languages such as Alloy (preferable for the most critical requirements).

1.6.1 Formal modeling with Alloy

Alloy is a declarative model checker tool that gives us the possibility to automatically check the correctness of our model. By using first order logic and relational calculus we can express the properties and the relationships that interest our model. Note that since it is "declarative" we do not specify properties by properties for the initial and final state of our model. In this sense, the ability is to designing a model which correctness is subordinate to that of the system and so we can check it before implementing the system.

1.6.1.1 Entities and relationships

Let's start with the basics:

```
sig Name, Addr
sig Book
  {
    addr:  Name→ lone Addr
  }
```

We defined three *entities* (or also *types*): `Name`, `Addr` and `Book`. The first two have no properties while the third one has a property `addr` which is a relationship (→) between a `Name` and `Addr` with the cardinality `lone`. In Alloy there are four types of cardinalities:

- `set` : any number;
- `one` : one and only one;
- `lone` : one or zero;
- `some` : one or more than one.

To be precise, `addr:  Name→ lone Addr` models a ternary relationship because also `Book` is included.

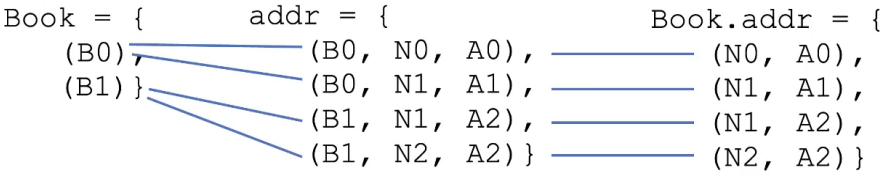
We can also put the keyword `one` before `sig` to specify that we want the only one instance of that entity can exists (we will see later the model instantiation).

1.6.1.2 Join relationships

As we can imagine, since there are relationship there must be a way of using them! For example, let's consider these relationships:

```
Book = {        addr = {
    (B0),           (B0, N0, A0),
    (B1)}           (B0, N1, A1),
                    (B1, N1, A2),
                    (B1, N2, A2)}
```

To "join" them we use the . operator, for example `Book.addr` is a new relationship that we can see as the result of an equi join:



1.6.1.3 Alloy possible worlds

Once our model is defined, we can use Alloy to instantiate a "world" that satisfies each property we have stated. Since they may be a lot, Alloy shows us them one by one, to the simplest to the more complex ones.

Note that Alloy is not a testing tool but a tool for performing *bounded verifications*:

testing:
a few cases of arbitrary size

bounded verification:
all cases within a certain bound

because we can't generate all possible worlds, we will see that we always choose the amount of instances of our possible worlds and we restrict our verification to that number of instances (and we obviously could not do otherwise). The picture above clarify that a bounded verification is limited but more complete in its bounds than a normal testing, which can surely cover less cases but in a "greater universe". In addition, this is also a reminder of the importance of doing both.

1.6.1.4 Predicates

Once I have defined the entities, I should define a set of *predicates* that represents the "rules of my model" and each "world" that will be instantiated will have to respect them all. A *predicate* can be defined with the following notation:

```
pred my_pred {}
```

This predicate is actually empty, so it has no constraint inside.

Run commands To apply a defined predicate to the worlds we are going to generate, we have to userun:

```
pred show {}
run show for 3 but 1 Book
```

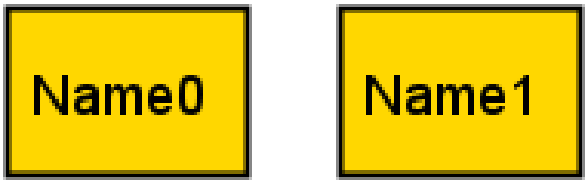
In this example we are defining an empty predicate `show` and a configuration that applies it on each world that contains:

- maximum 3 instances for each identity;
- maximum 1 instance of `Book`.

So, if we "run" `show` (we can select which configuration run) in Alloy it will generate only worlds with these properties that respect the predicate `show` (it is empty in this case).

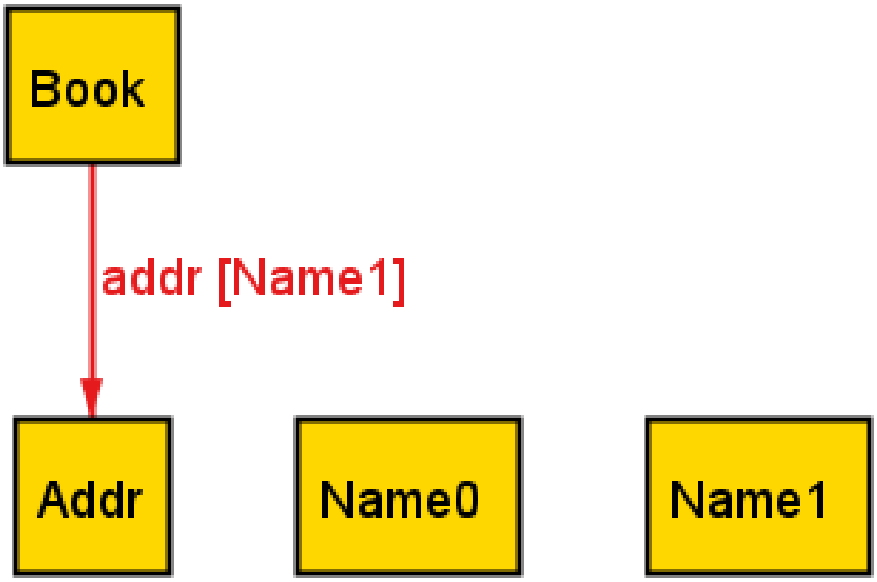
Note that, since we have not specified any constraint for the relationship `addr` contained in `Book`, all these worlds can be generated possibly:

- neither `Addr` instances nor `Book` instances but two `Name` instances:



- one `Book` instance, one `Addr` instance and two `Name` instances but the `Book` instance contains no addresses:

- same set of instances as before but now the `Addr` instance is "included" in the one of `Book`:



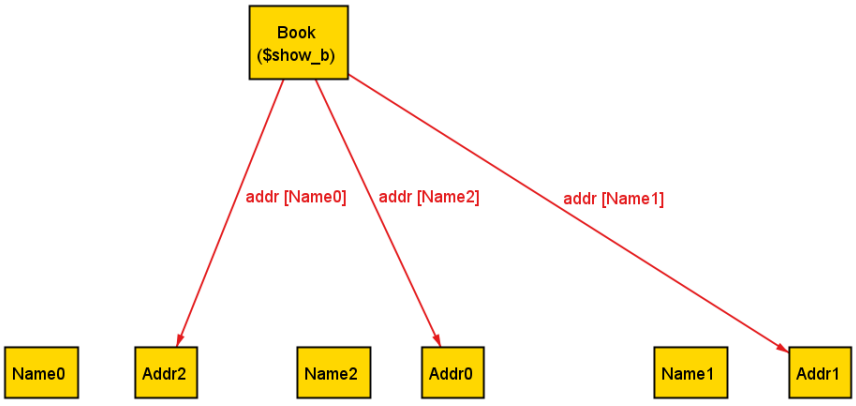
So now we should have got how Alloy works. We define some entities, relationships between them, predicates (for now, only the empty one) and it (Alloy) generates some worlds where:

- entities are instantiated the number of times specified by the relative `run`;
- instances may or may not take part in the defined relationships (unless we force it with predicates);
- predicates are always satisfied (if it is impossible to satisfy them all, no world will be generated).

Logical predicates We can't do much with the empty predicate, let's start with some logical predicates:

```
pred more_than_one_addr_per_book [b:  Book]
  {
    #b.addr>1
  }
```

Intuitively, we are specifying that each instance of `Book` must have at least 2 addresses. So, a possible world is:



We can even add multiple constraints to the same predicate, for example:

```
pred two_or_more_different_addresses [b:  Book]
```

```
  {
    #b.addr>1
    #Name.(b.addr)>1
  }
```

`#Name.(b.addr)>1` implies that in our book there must be at least two different addresses (see how join works).

As we mentioned before, we can ever create first order logic constraints, for example:

```
pred at_least_two_addr_per_name [b:  Book]
  {
    #b.addr>1
    some n:Name|#n.(b.addr)>1
  }
```

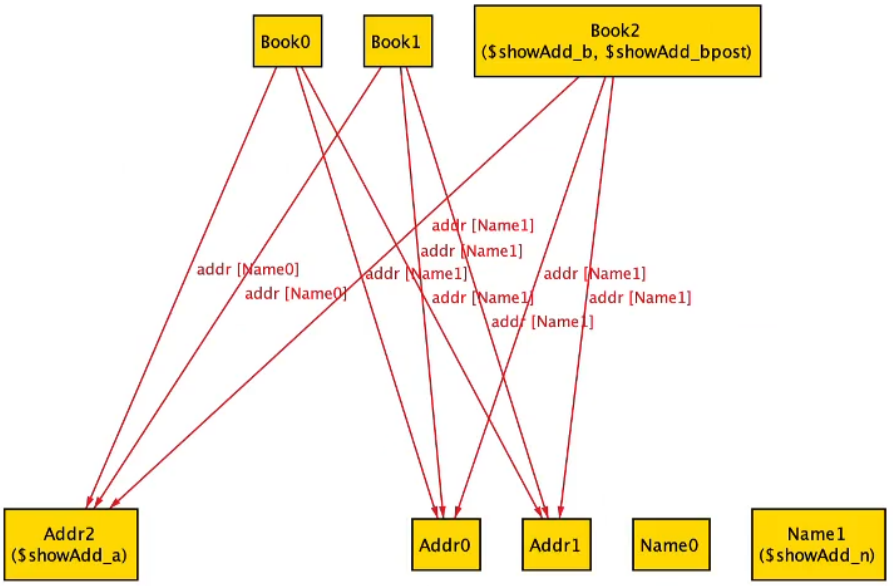
`some n:Name|` can be read as $\exists n \in Name$,, so "exists at least one name with two or more addresses associated". We note that this makes a contradiction if we use `lone` so, no worlds will be generated.

Operation predicates Performing an operation, means that we have take something and turn it into another thing, which seems quite impossible in "static" worlds, we had to generate a world an then generate another which is the result of the operation (e.g. in the first world I have two records in the book and in the second one I have one of them removed). However, this is not the logic adopted by Alloy. Let's see some quick examples:

```
pred add_an_addr [b,bpost:  Book,n:  Name, a:  Addr]
  {
    bpost.addr=b.addr+n→a
  }
show_add [b,bpost:  Book,n:  Name, a:  Addr]
  {
    add[b,bpost,n,a]
  }
```

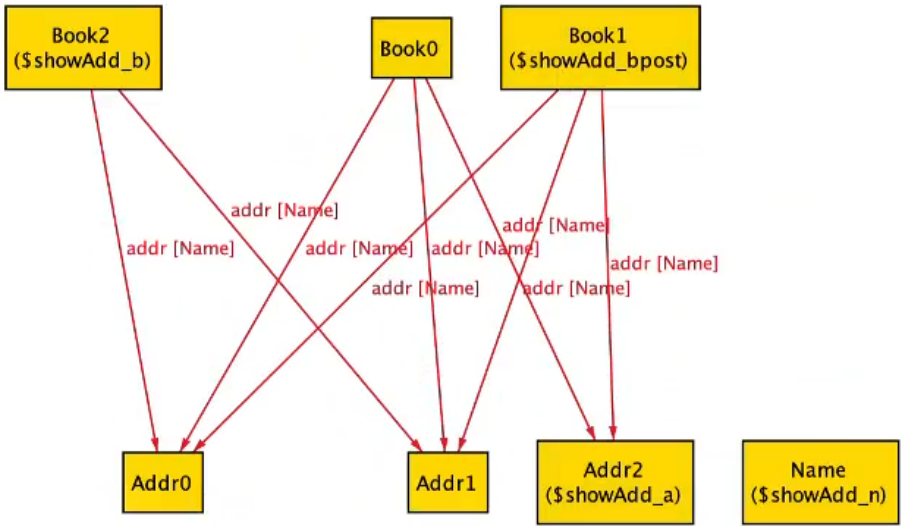
Basically, what happens is that in each world Alloy (as for what happens for each predicate: AFTER having generated each instance), chooses the instances that have to satisfy the predicate, and so it will choose one instance for `b`, one for `bpost`, one for `n` and one for `a` to apply the predicate. The predicate basically impose that `bpost` must have all records of `b` plus `n→a` (+ the set-union). Note that these worlds can also be generated:

- worlds where the instance chosen for `b` is the same of the one chosen for `bpost`:



- worlds where the instances chosen for `n` and `a` are already present in the one chosen for `bpost`.

So, a situation like this:



which is the most expressive for us, it is not the only correct one.

Note that `n→a` is a single record but we can specify any cartesian product, for example `n→Addr` (it will add to `bpost` each possible record where `n` is the number).

Along with the set-union we have also the set-minus with `-`.

1.6.1.5 Functions

We can define functions for code reusing, the syntax is pretty basic:

```
fun <function-name> [<function-parameters>] :  <function-return-type>
  {
    <function-body>
  }
```

and we can call it with the already seen notation `<function-name>[<call-parameters>]`.

For example (return the set of addresses that in `b` are associated with the name `n`):

```
fun getNameAddresses [b:  Book, n :  Name] :  set Addr
  {
    n.(b.addr)
  }
```

1.6.1.6 Assertions

An *assertion* is a condition that should never occur in any possible world. Define an assertion, Alloy can check if the set of possible worlds contains at least one world that violates it and we can also see it. Let's see a quick example (`some` has the same meaning of $\forall$ and `implies` the same meaning of $\Rightarrow$):

```
assert delUndoesAdd
  {
    all b,bpost,bppost:  Book, n:  Name, a:Addr|
      add_an_addr[b,bpost,n,a] and remove_an_addr[bpost,bppost,n,a]
implies b.addr = bppost.addr    }
check delUndoesAdd for 3
```

`check` is the analogue of `run` for the predicates.

This assertion asserts that each instance of `Book` on which the `add_an_addr` is applied is equal to the one which is the result of `remove_an_addr` (note that obviously this is satisfied only in worlds where the added record is not present in the initial book, otherwise it would not be added to the book but only removed and so the assert would not be satisfied). Anyway, it only needs a quick adjust:

```
assert delUndoesAdd
  {
    all b,bpost,bppost:  Book, n:  Name, a:Addr|
      no n.(b.addr) and add_an_addr[b,bpost,n,a] and
remove_an_addr[bpost,bppost,n,a]          implies b.addr = bppost.addr
}
check delUndoesAdd for 3
```

Assertion can be used also to verify the relative strength of a predicate. Let's consider two generic predicates `p1` and `p2`:

```
assert p1StrongerThanP2    {
    p1 implies p2
  }

assert p2StrongerThanP1    {
    p2 implies p1
  }
```

"Strength" in the sense that one predicate is more restrictive than other. Note that if the two assertion assertions are true, it means that the two predicates are equivalent.

1.6.1.7 Facts

While a predicate once run is applied to a certain subset of the instances generated, a *fact* must be always respected by every instance. The instance is almost identical to the predicates one, we only use `fact` instead of `pred`.

1.6.1.8 Entity hierarchies and some advanced algebraic and logical operators

Let's see now how we can define hierarchies of entities. As example, we will define a simple prototype of a family tree:

```
abstract sig Person
  {
    father:  one Man
    mother:  one Woman
  }
sig Man extends Person
  {
    wife:  lone Woman
  }
sig Woman extends Person
  {
    husband:  lone Man
  }
```

Basically, we say that a `Person` `Man` or `Woman` and then:

- a `Person` has a `father`, that is a `Man`, and a `mother`, that is a `Woman` (let's simplify a bit from the reality);
- a `Man` can have zero or one `Woman` associated that is his `wife`;
- a `Woman` can have zero or one `Man` associated that is her `husband`.

Note that this is not a proper family tree! Theoretically it could be, but in practice there are several possibilities that for common-sense or for social conventions are not proper of a family tree:

1. a person can be an ancestor of him/herself;
2. a person can be an ancestor of him/her parent;
3. a person can have one parent that is both the father and the mother;
4. a person can only marry someone who married him/her;
5. a person can marry a relative (including him/herself).

Let's now see how we can ensure these basic properties. Note that nothing changes if we are note considering a hierarchy, this example is only an intuitive one to do some practice with facts/predicates formulations.

For the first requirement, we need to use a kind of "endless" operator because we want to state something "for any hierarchic level": for parents, for grandparents, for grandgrandparents, for grandgrandgrandparentents and so on. Given a generic relation `r`, in Alloy we denote with:

- `^r` : the transitive closure of that relation;
- `*r` : the reflexive and transitive closure of that relation.

Basically, `^r` is a compact form for the expression `r+r.r+r.r.r+...` that is exactly the transitive closure of the relation `r` if we see the join operation (`.`) as the relational product from the relational algebra.

So, this is the fact to fulfill the first requirement (`no` works as $\nexists$):

```
fact
  {
    no p:Person | p in p.^(mother+father)
  }
```

`mother`, for example, is the relationship that contains each pair "son-mother", transitively closed it becomes the relationship that contains each pair "son-(grand)mother" at any level and so `p.^(mother)` returns the set of the mother and all grandmothers (at any level) of `p`.

The third requirement is pretty basic:

```
fact
  {
    no p:Person | p.father = p.mother
  }
```

The fourth requirement is pretty simple to express by means of the operator $\sim$:

```
fact
  {
    wife = ~husband
  }
```

which is the transpose operator (given a generic relation `r`, $\sim$`r` is its inverse relation). Note that if we want to write the equivalence closure of a generic relation `r` we can simply write `r+*r`.

The last requirement can be expressed as follows (`&` is the intersection symbol and `none` is the empty set):

```
fact
  {
    no p:Person|p.wife=p or p.husband
    no ((wife+husband) & ^(mother + father))
    all p1:Man,p2:Woman | p1->p2 in wife implies (p1.^(mother+father))&
p2.^(mother+father)=none    }
```

Let's explain each one:

- `no p:Person|p.wife=p or p.husband`: a person can't merry him/herself;
- `no ((wife+husband) & ^(mother + father))`: a person can't merry one of his/her ancestors; `all p1:Man,p2:Woman | p1->p2 in wife implies (p1.^(mother+father))& p2.^(mother+father)=none`: a person has not the other one in his/her relatives.

1.6.1.9 Mutable relations and timing constraints

A recent Alloy feature is the real possibility of modeling world evolutions.