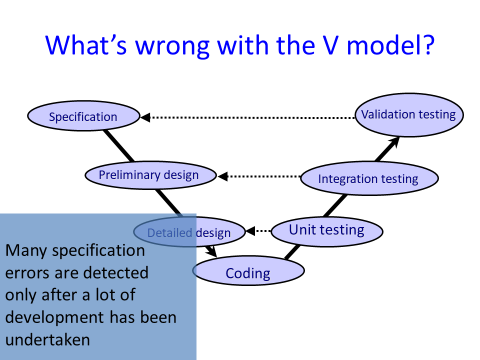
*Lecture 1:*

**Software engineering:** The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

**Software process models:**

* Determine the order of the stages of software development.
* Establish the transition criteria to proceed from one stage to the next.

****

*Lecture 2:*

**What are requirements?**

Requirement: description of something a product must do or a quality it must have.

**Functional requirements** are things the product must do or provide of it is to be useful to its users (e.g. allow user to check-in). *(USES)*

**Nonfunctional requirements** are the qualities the product must have (e.g. shall respond in 2 seconds, can handle 5000 simultaneous requests). *(QUALITIES)*

**Constraints** are global issues that shape the requirements (e.g. can run on a 3G mobile telephone).

**Functional requirements**

Functional requirements should

* **Focus** on the intended purpose of the system.
* **Ignore** details of how that purpose is achieved.

If the purpose is to provide some **service**, then:

* Model **what** a system does from the perspective of the service users.
* ‘Users’ might be computing agents as well as humans.

If the purpose is to **control**, **monitor** or **protect** some phenomenon, then:

* The assumptions should focus on those **phenomenon**.
* In **what** way should they be controlled or protected?

**Should design concerns be considered as requirements?**

Design Concerns or choices describe **how** the desired system will be designed and implemented (e.g. the system shall use a database to store customer information).

The requirements phase should **avoid** identifying any design requirements and should instead focus on functional requirements (specifying design requirements too early may overly constrain the design).

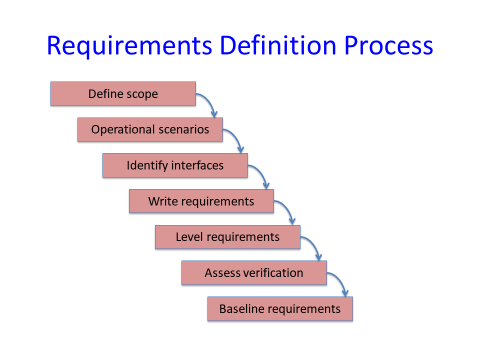
**Distinguish different kinds of functional requirements.**

Normal – users can send messages to other users.

Error – in case of failure, notify the sender.

Safety – the elevator must not move while the doors are open.

Security – a user must provide a password in order to read their email.



**Assumptions.**

Assumptions: describe properties of the **environment** of a software product. The software product cannot control these properties directly, but they are necessary for overall correct functioning of the system.

* Examples:
  + Users keep passwords secret.
  + If the brakes are applied to a train system, the train will stop within 3 minutes.

**Defining scope.**

The scope provides a high level view of the planned product on terms of:

* **Need:** what business/mission need is it intended to meet?
* **Goals:** what is its intended purpose? What functionality will it provide?
* **Case:** what is the business/mission case for the product?
* **High-level operational concepts:** high-level view of how the product will be used.
* **Stakeholders:** customers, users, administrators, maintainers, developers, testers.

The purpose of the scope is to help keep the bigger picture in view. Ideally the scope should remain fixed during the entire development process.



**Use Case**

Definition – in software and systems engineering, a use case is a list of actions or event steps typically defining the interactions between a role (known in UML as an actor) and a system to achieve a goal. The actor can be a human or other external system.

Use cases provide descriptions of scenarios for usage of the system typically describing series of interaction between users and the system to be developed.

These help people to quickly gain an understanding of what is intended.

**Interfaces**

Points where the system interacts with the environment. (environment: users, physical devices such as sensors…).

Defining the interfaces helps to define the boundary between what is the responsibility of the product development team and what is outside their responsibility.

**Interfaces are important for defining ways of testing the implemented system.**

E.g. interfaces for an online bank:

Customers:

* PC, smartphone, tablet…

Bank staff:

* Branch IT access system...

…

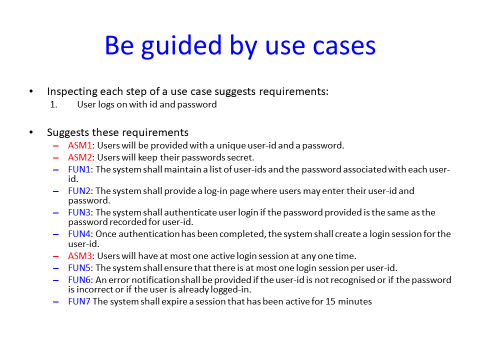
**Writing Requirements**

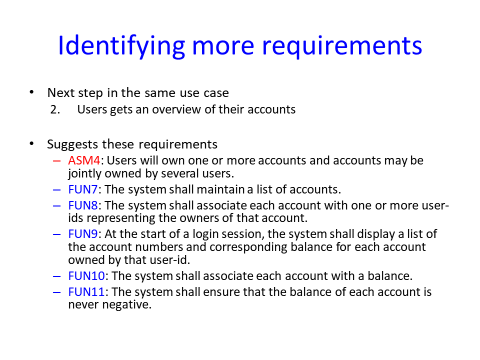
Requirements definition: documentary record of the system requirements expressed in the customers’ terms.

Two types of **functional requirements:**

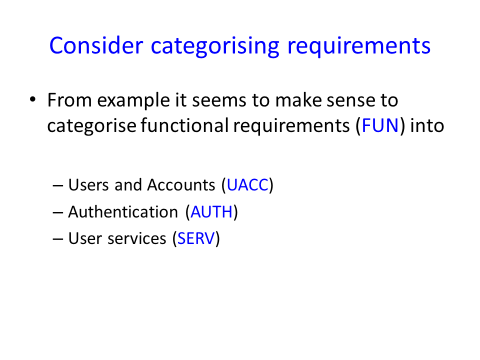
* **Assumptions** about the environment. (The user will…)
* **Requirements** on the desired behavior. (The system shall…)

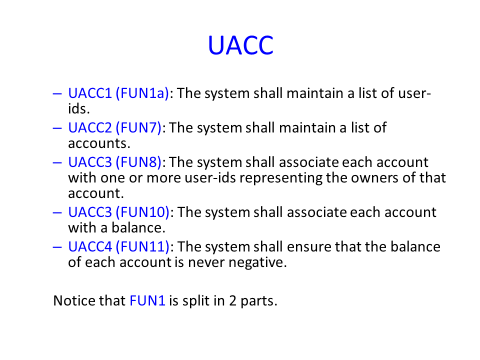
Use simple sentences to write requirements.

****

****

**Categorize Requirements**

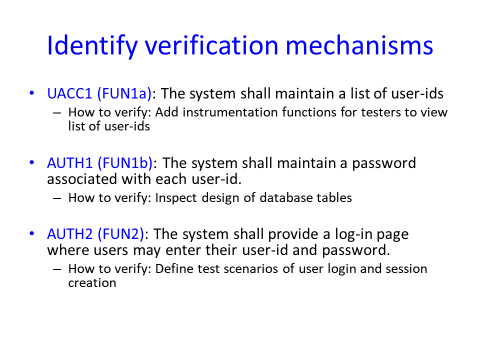
****



And so on… Look at slides for continued example.

**Access verification**

Once a requirement has been identified, think about a mean of verifying that the requirement has been satisfied by the implementation.



**Baseline the requirements**

Definitions formally reviewed and agreed on.

Serves as basis for further development of the software system.

Only change baselined requirements through formal change process, including version numbers.

*Lecture 3:*

**Use cases (again)**

A structured textual description.

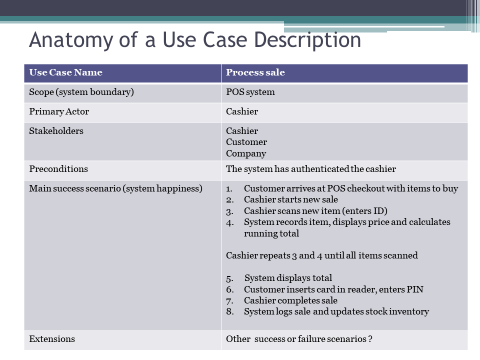
Captures a more formal view of a user’s interaction with a system.

Use cases model:

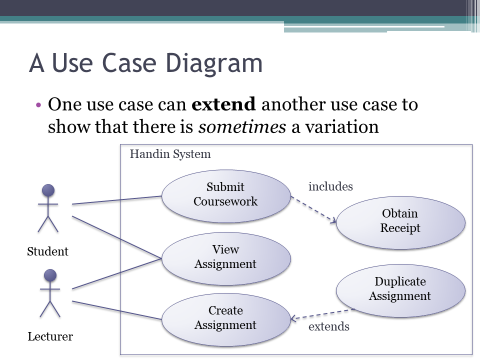
* A collection of use cases.
* Define what the system should do.
* And shows who uses the system.

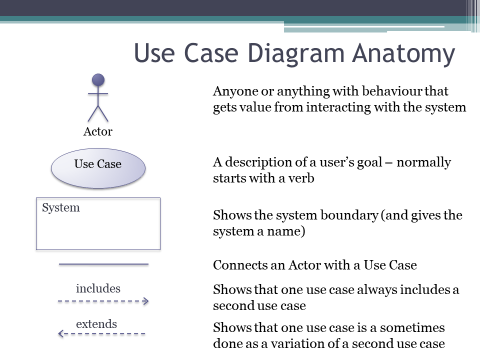
They are useful because they:

* Capture fuzzy requirements in a more precise way.
* Provide a basis for testing.



**Diagrams:**





*Lecture 4:*

**What is UML?**

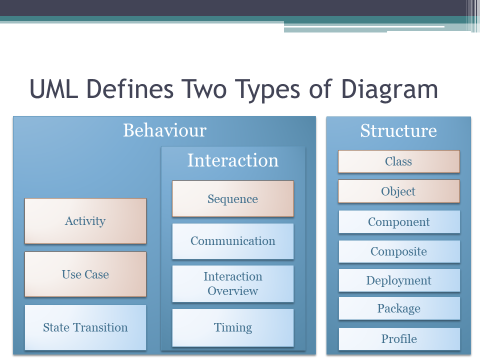
Standard language for **specifying, visualizing, constructing** and **documenting** the artifacts of software systems, business modeling and other non-software programs.

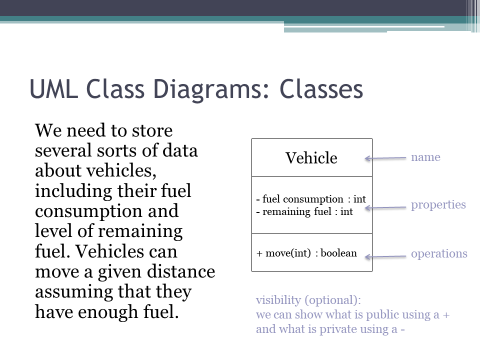
The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

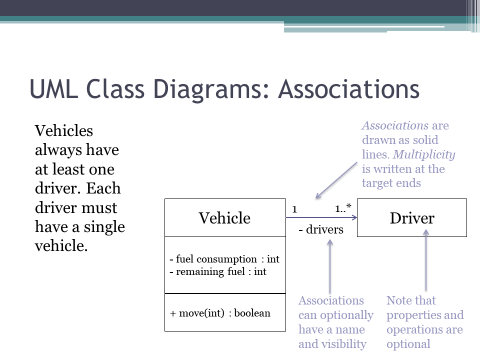
The UML is mostly used in developing object orientated software and the software development process.

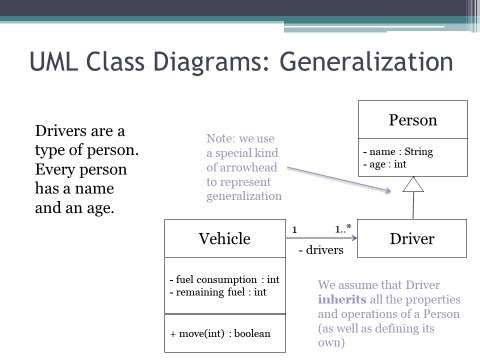
The IML uses mostly **graphical notations** to express the design of software projects.

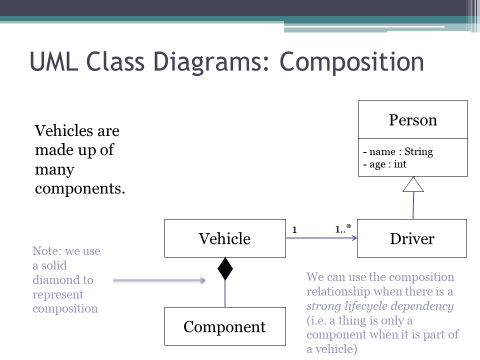
Using the UML helps project teams **communicate**, explore potential designs, and validate the architectural design of the software.

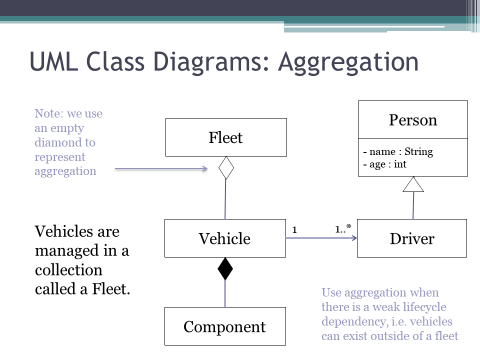


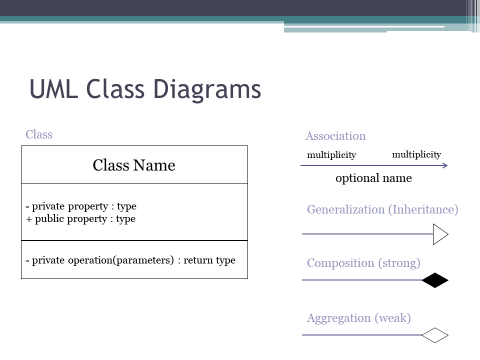


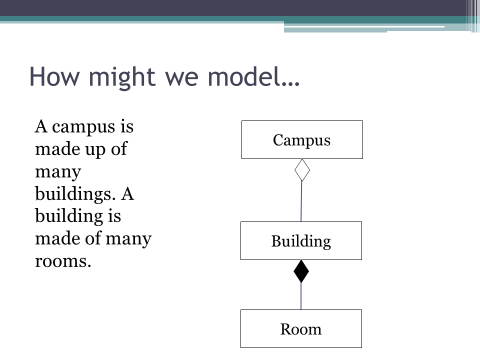












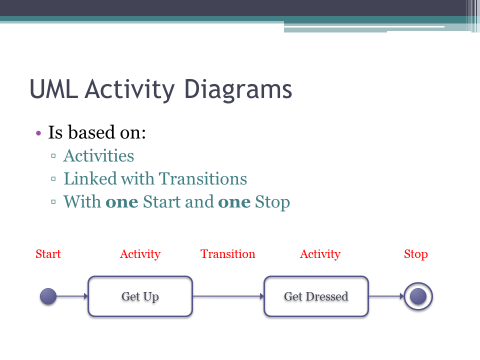
**When to use Class Diagrams**

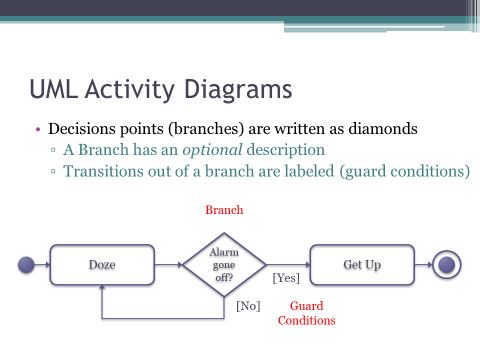
Whenever you want to model the types of things in a system, and how they relate to one another (sometimes called data modelling).

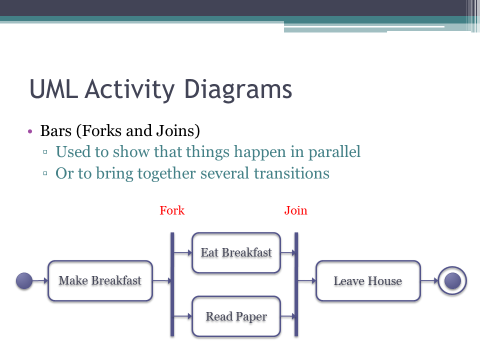
*Lecture 5:*

**Activity diagrams**

UML Equivalent of a flow chart.









Swinlanes (upstairs and downstairs)

Swimlanes partition a diagram (used to show different logical areas).

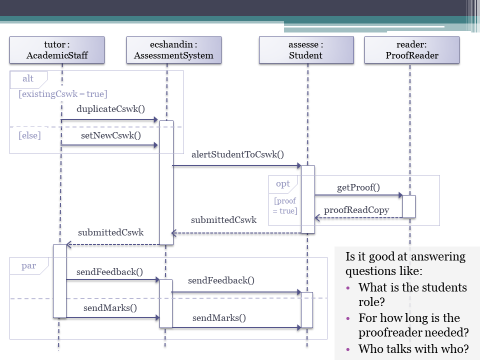
Diagrams can be partitioned in different ways. There is no right way.

When to use activity diagrams?

* When analyzing a use case.
  + What actions are there and when do they happen?
  + This is sometimes called the Flow of Control.
* Useful for communication order and dependency.

**Sequence diagrams**

Look presentation for more explanation on the slide.

****

**Which diagrams to use?**

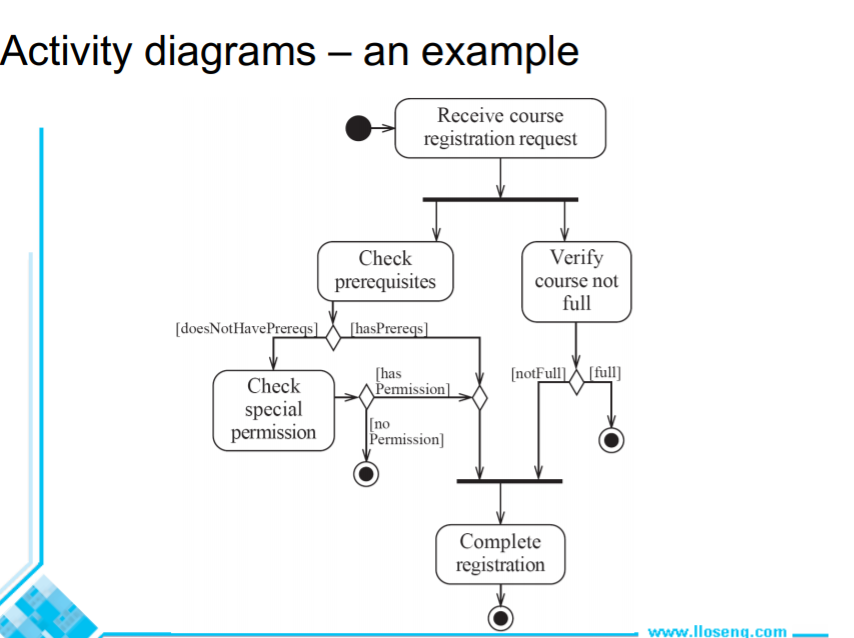
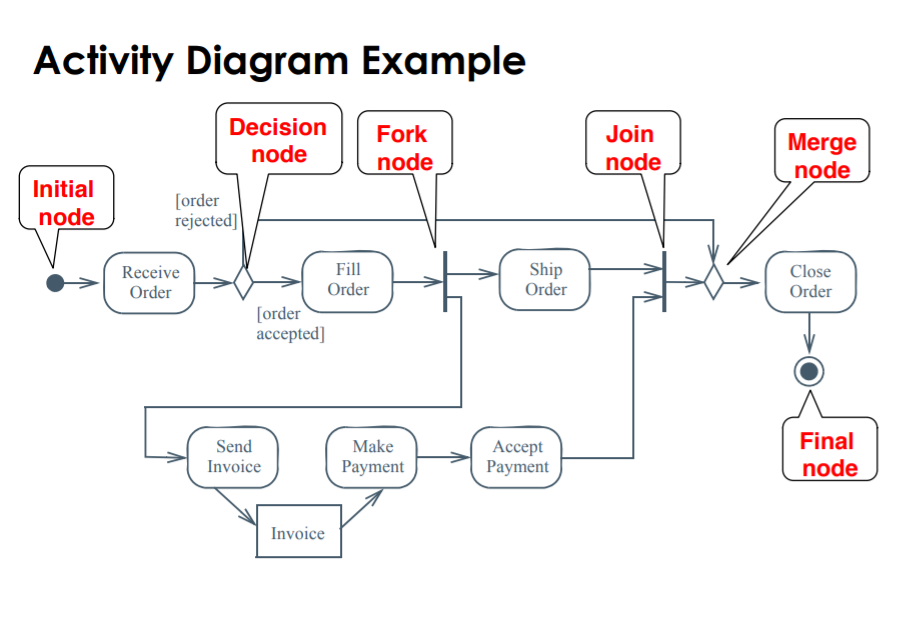
**Activity diagrams:**

* What actions are there and when do they happen?
* Useful for communication order and dependency.

**Sequence diagrams:**

* What are the key roles, how do they interact in what order?
* Shows the sequence of interaction.
* Useful for communication lifespans, and showing interaction over time.

*Lecture 6:*



**Swimlanes** are used to denote the object or subsystem that implements the activities.

Activity diagrams are most often associated with several classes. The partition of activities among the existing classes can be explicitly shown using swimalanes.

**There are different types of objects.**

* Entity objects:
  + Represent the persistent information tracked by the system (application domain objects, also called “business objects”).
* Boundary objects:
  + Represent the interaction between the user and the system.
* Control objects:
  + Represent the control tasks performed by the system.

**Heuristics for Sequence Diagrams:**

Layout:

* 1st column should be the **actor** of the use case.
* 2nd column should be a **boundary** **object**.
* 3rd column should be the **control** **object** that manages the rest of the use case.

Creation of objects:

* Create control objects at begging of event flow.
* The control objects create the boundary objects.

Access of objects:

* Entity objects can be accessed by control and boundary objects.
* Entity objects should not access boundary or control objects.

**What else can we get out of Sequence Diagrams?**

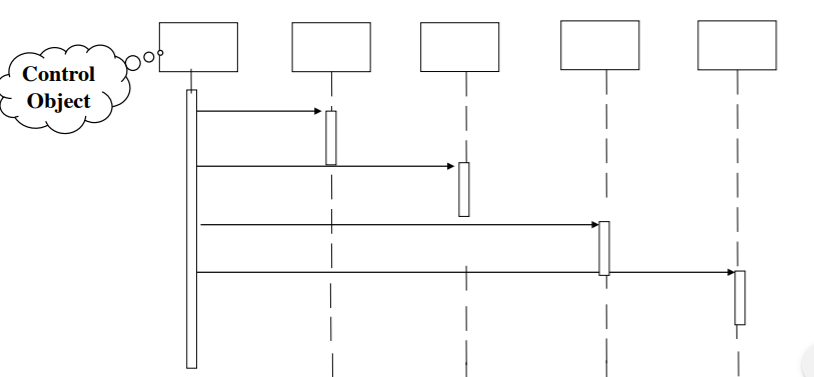
Sequence diagrams are derived from use cases.

The structure of the sequence diagram helps us to determine how decentralized the system is.

We distinguish two structures for sequence diagrams – **Fork Diagrams** & **Stair Diagrams**.

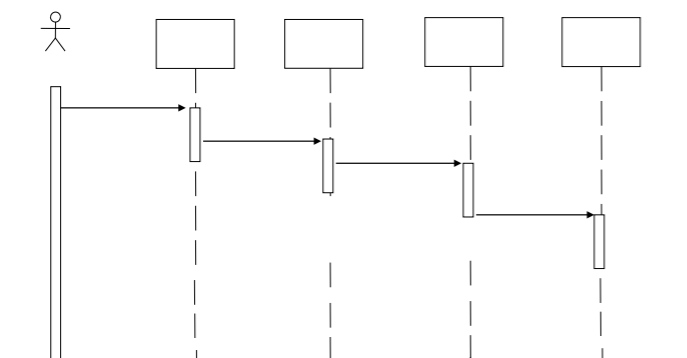
**Fork diagrams**

The dynamic behavior is placed in a single object, usually a control object. (The control object knows all the other objects and often uses them for direct questions and commands).



**Stair diagram**

The dynamic behavior is distributed. Each object delegates responsibility to other objects. (Each object knows only a few of the other objects and knows which object can help with a specific behavior).



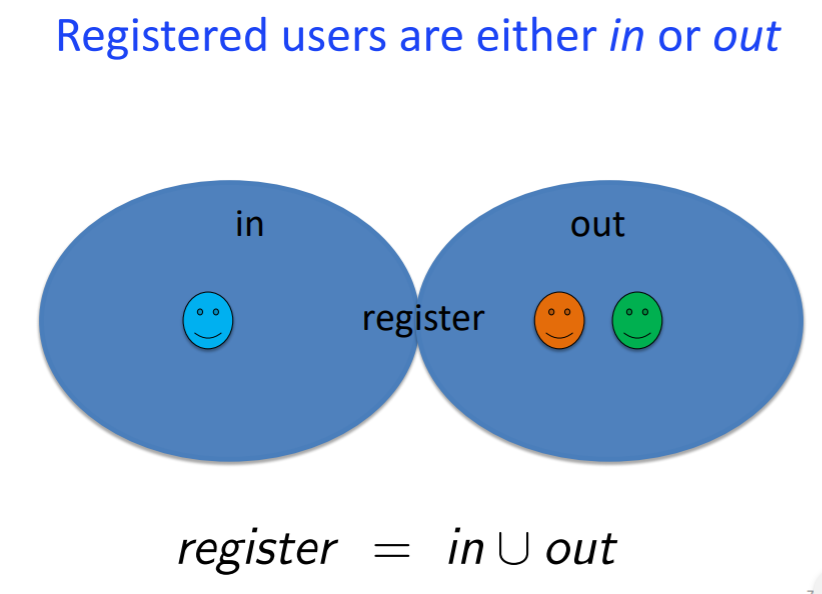
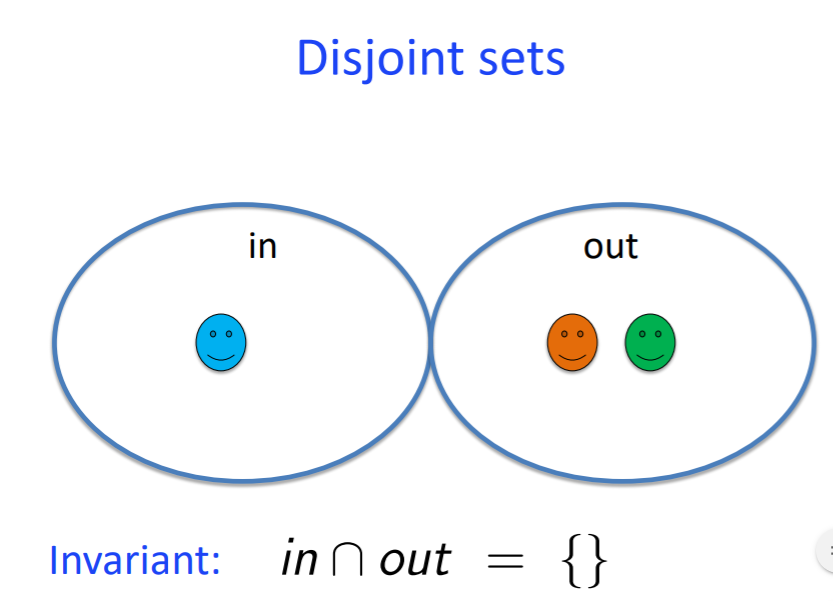
**Fork or Stair?**

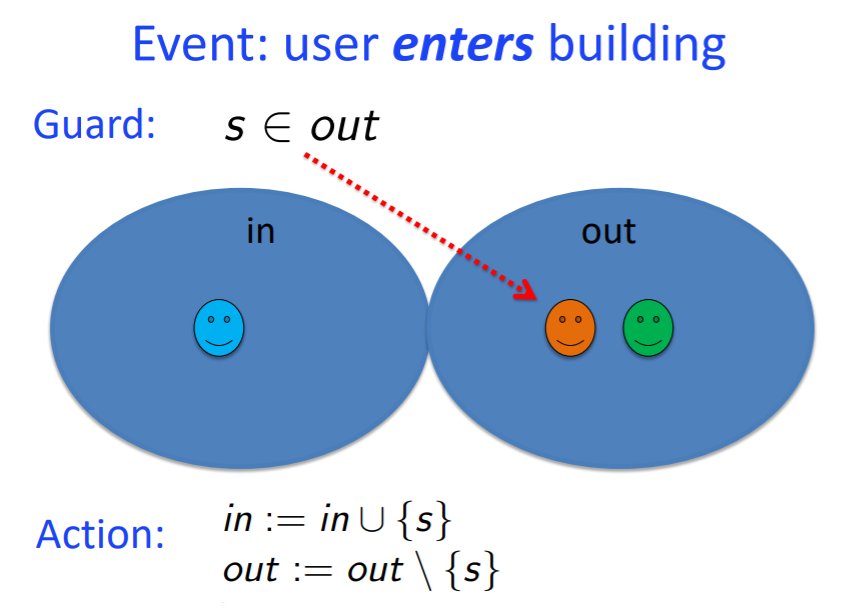
Object-orientated supporters claim that the stair structure is better.

Modelling advice:

* **Choose the stair** – a decentralized control structure – if:
  + The operations have a strong connection.
  + The operations will always be performed in the same order.
* **Choose the fork** – a centralized control structure – if:
  + The operations can change order.
  + New operations are expected to be added as a result of new requirements.

*Lecture 7:*





**Basic set theory:**

A **set** is a collection of **elements**.

Elements of a set are **not** **ordered**.

Elements of a set may be numbers, names, ids, etc.

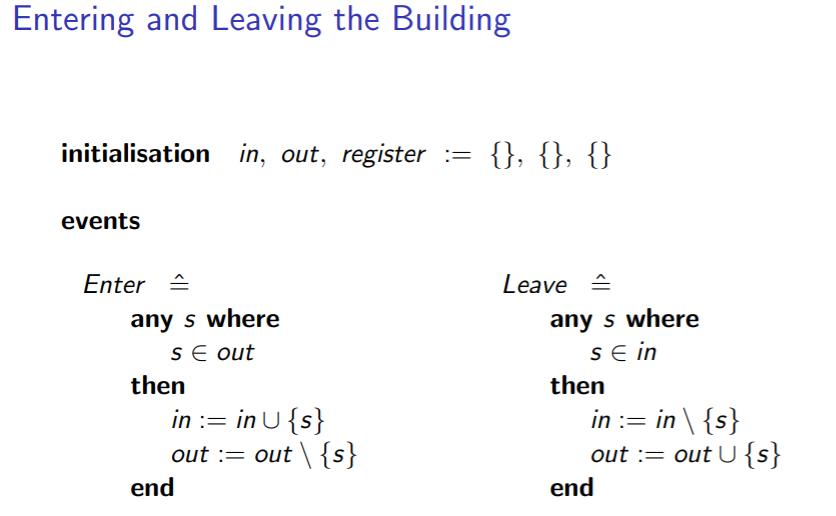
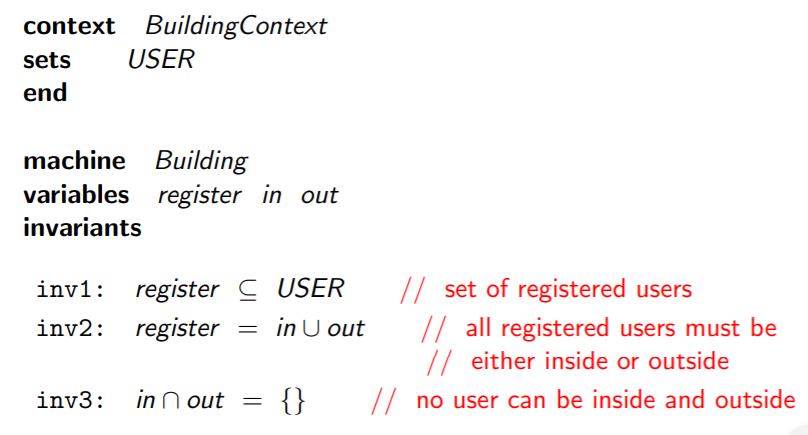
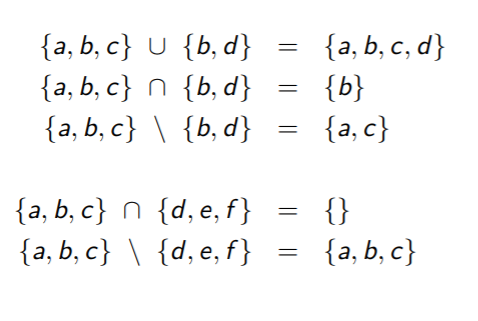
Sets may be **finite** or **infinite**.

Relationship between an element and a set: the element is a **member** of the set.

A set S is said to be a **subset** of set T when every element of S is also an element of T. This is written as: **S ⊆ T**.

A set S is said to be **equal** to set T when S ⊆ T and T ⊆ S. (e.g. {5, 8, 3} = {3, 5, 5, 8})

**Difference** of S and T: set of elements **in** S **but not in** T: S / T.



**Event-B context:**

**Carrier sets**: abstract types used in specification.

**Constants**: logical variables whose value remain constant.

**Axioms**: constraints on the constants. An axiom is a logical predicate.

**Event-B machine:**

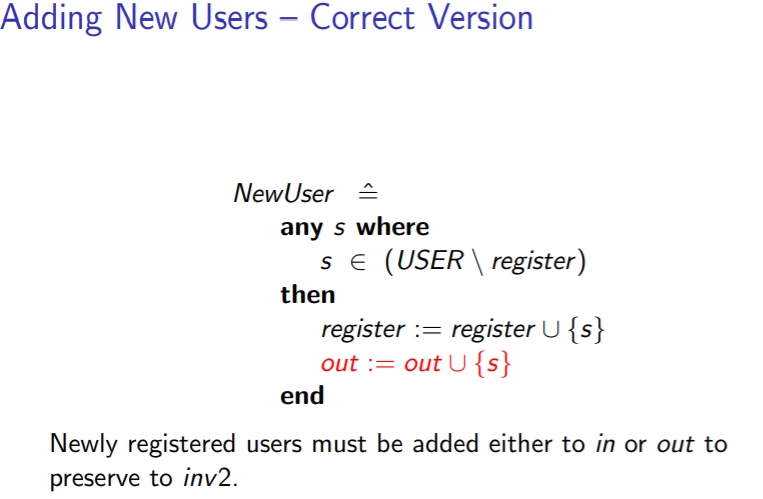
**Sees:** one or more contexts.

**Variables:** state variables whose values can change.

**Invariants:** constraints on the variables that should always hold true. An invariant is a logical predicate. (basically the requirements)

**Initialization:** initial values for the abstract variables.

**Events:** guarded actions specifying ways in which the variables can change. Events may have parameters.



**Event-B in software development**

Event-B is intended for modelling and reasoning about system behavior.

System specifications are derived from **requirements**.

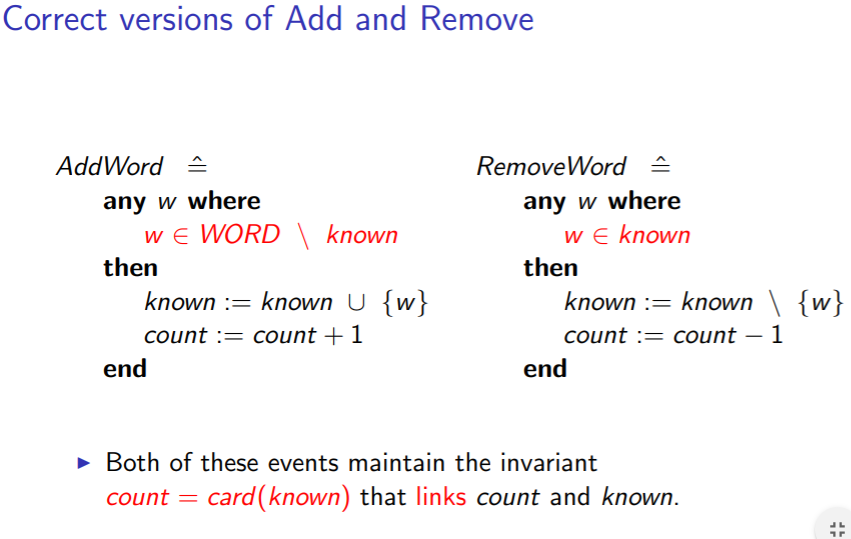
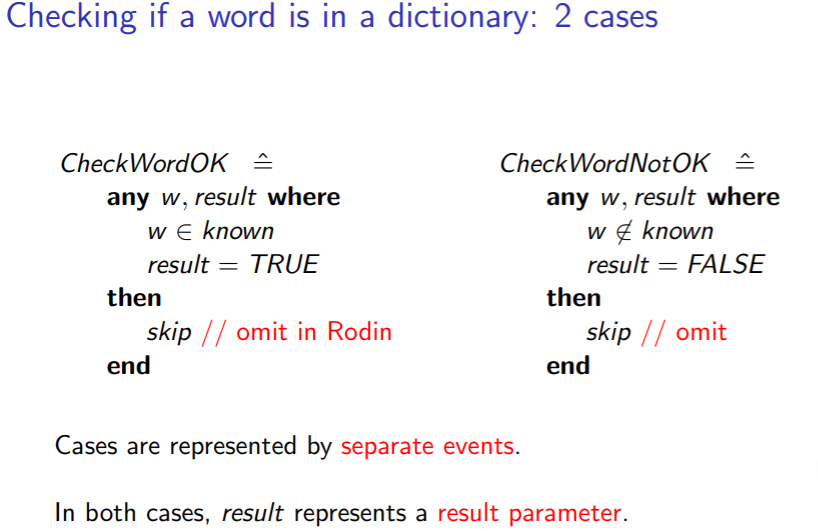
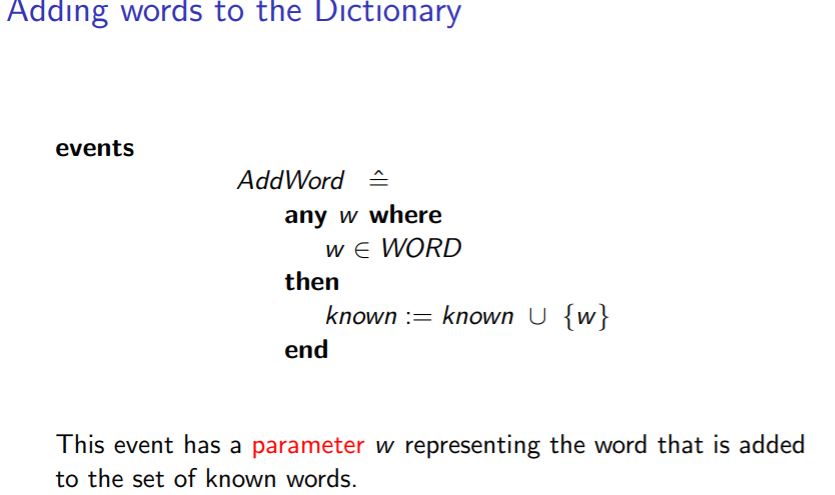
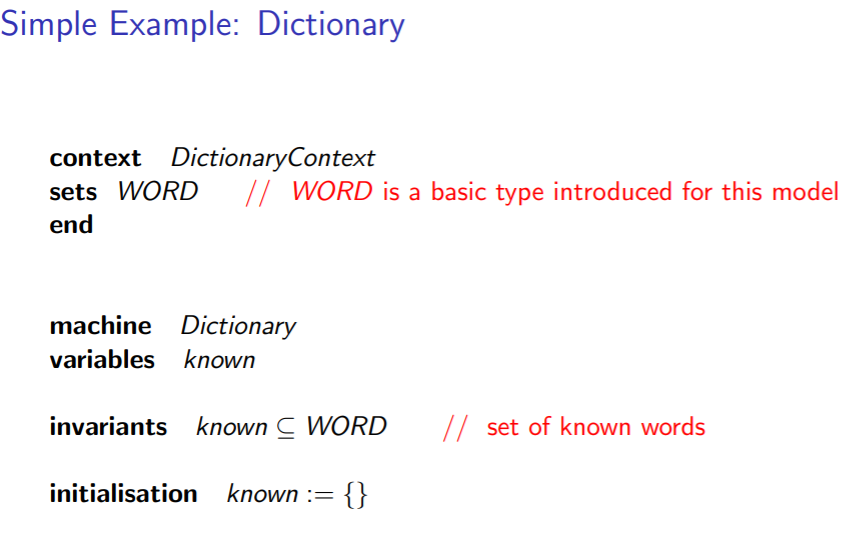
System specification in an important **precursor** to programming and testing.

Event-B: formal language for writing **high-level specifications** of computer systems.

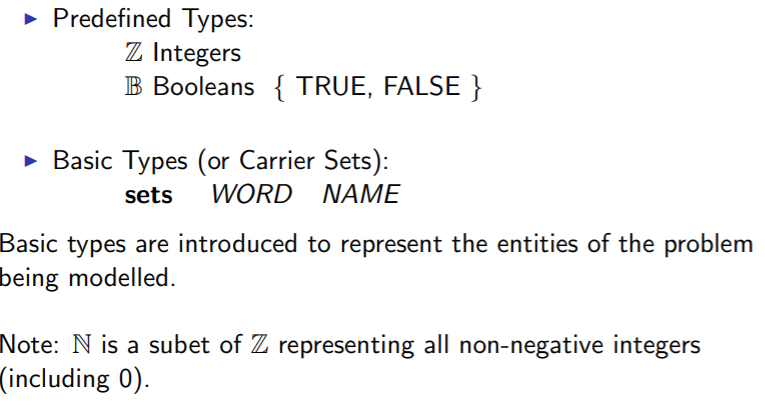
Event-B language includes **logic** and **set theory**.

Formal specification is more **precise** and **consistent** than an informal (natural language) specification.

Event-B typically used in **safety-critical** or **mission-critical** applications.



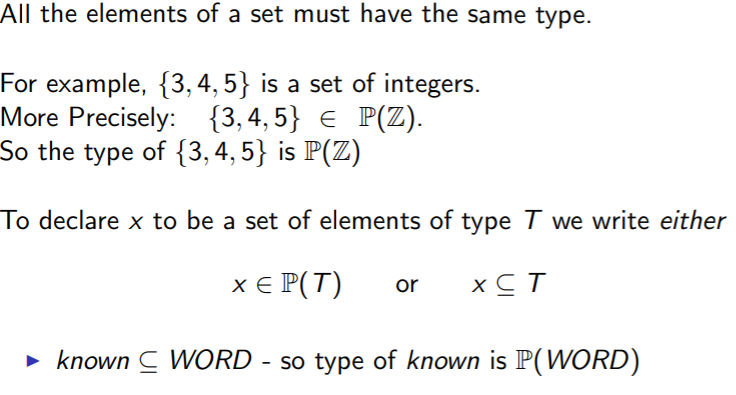
**Types in Event-B**



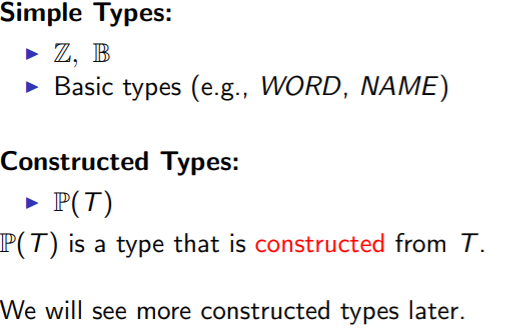
**Powersets**



**Types of Sets**



**Classification of Types**



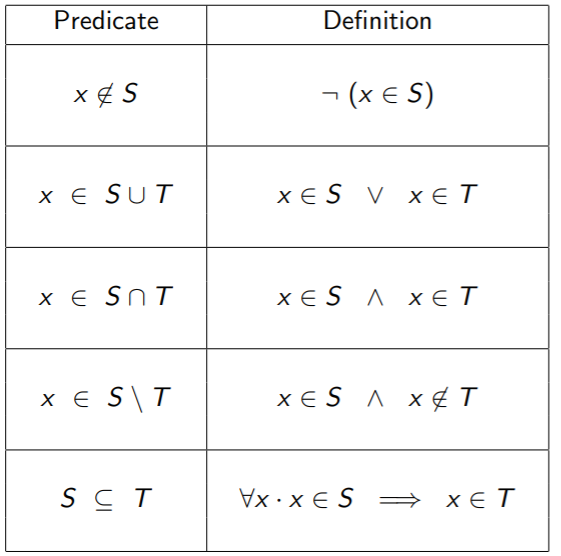
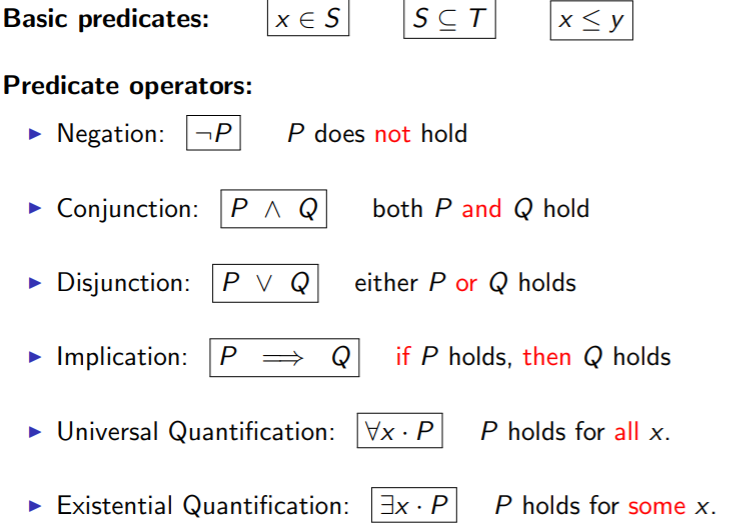
**Why types?**

Types help to structure specifications by differentiating objects.

Types help to prevent errors by not allowing us to write meaningless things.

Types can be checked by a computer.

**Predicate logic:**



*Lecture 8:*

**Relations and Functions**

**Requirements for building an access system:**

Specify a system that control access to a **collection** **of** **buildings**.

Registered users will have access **permission** to enter certain buildings.

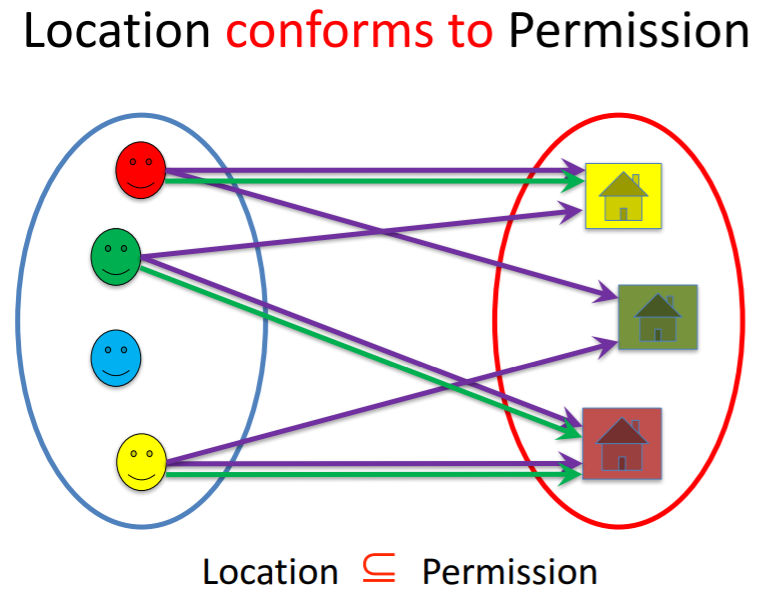
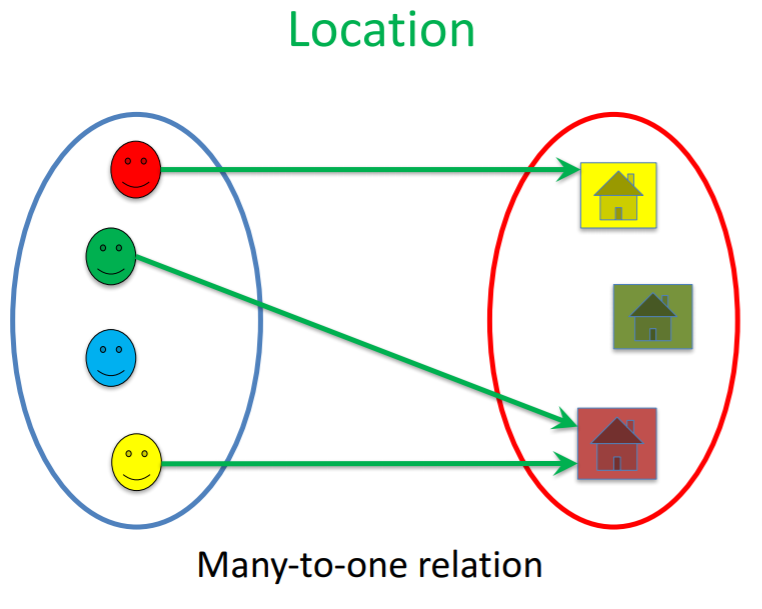
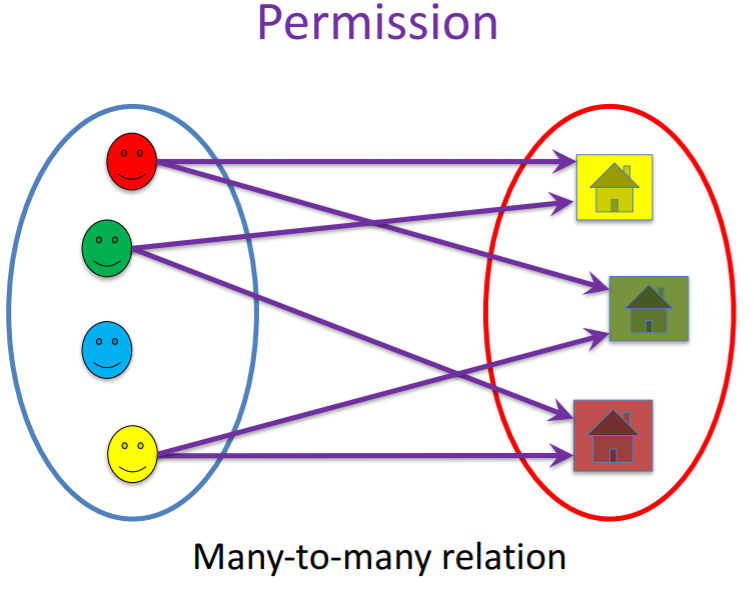
A user can only enter buildings that they have access permission for.

The system should keep track of the **location** of users.

The system should manage **registration** and access permission for users.



--



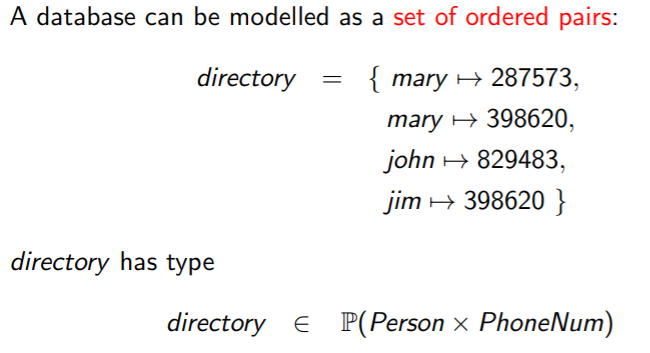
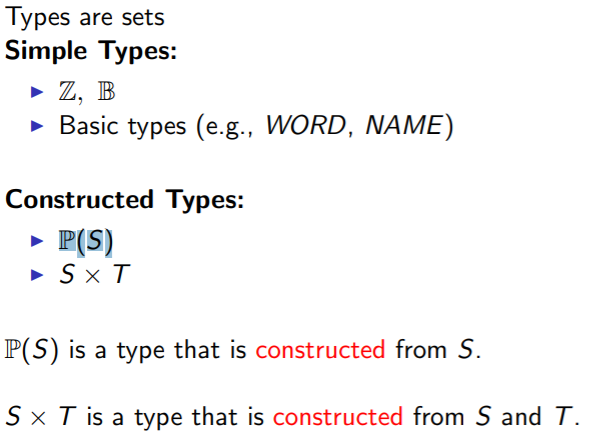
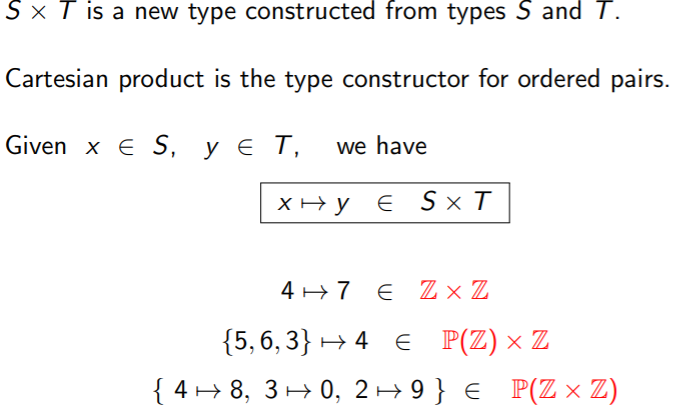
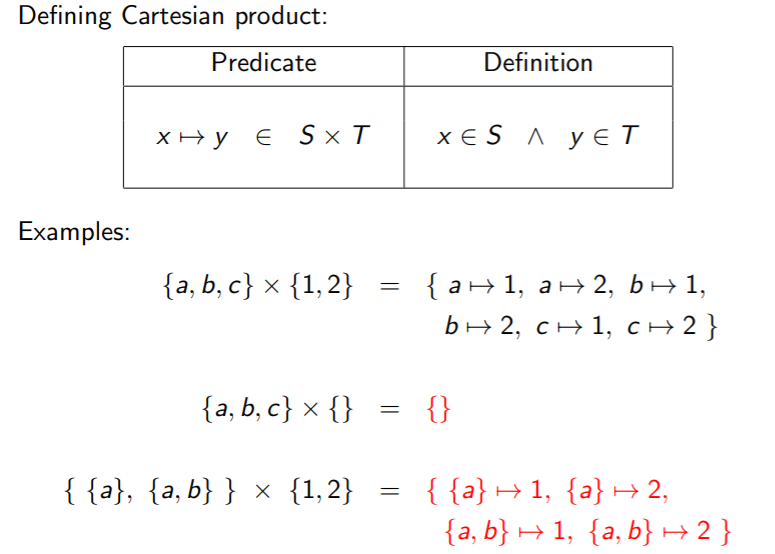
**Ordered pairs and Cartesian products**

An **ordered pair** is an elements consisting of two parts.

An ordered pair with first part X and second part Y is written:



The **Cartesian product** of two sets is the **set of pairs** whose first part is in S and second part is in T.



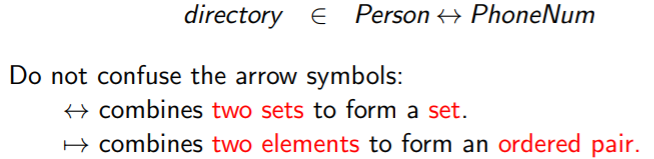
**Relations**

A relation is a set of ordered pairs.

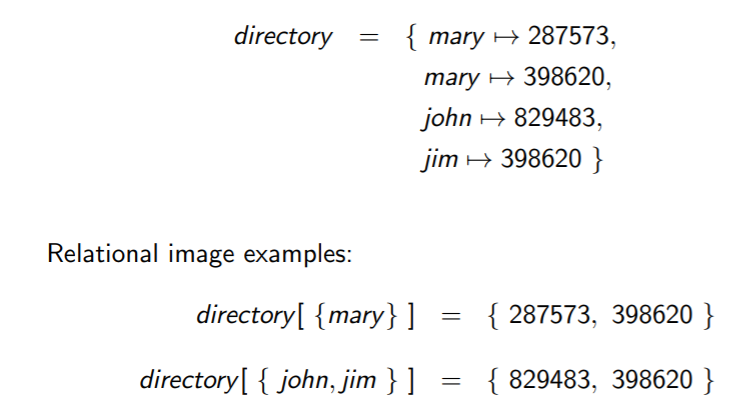
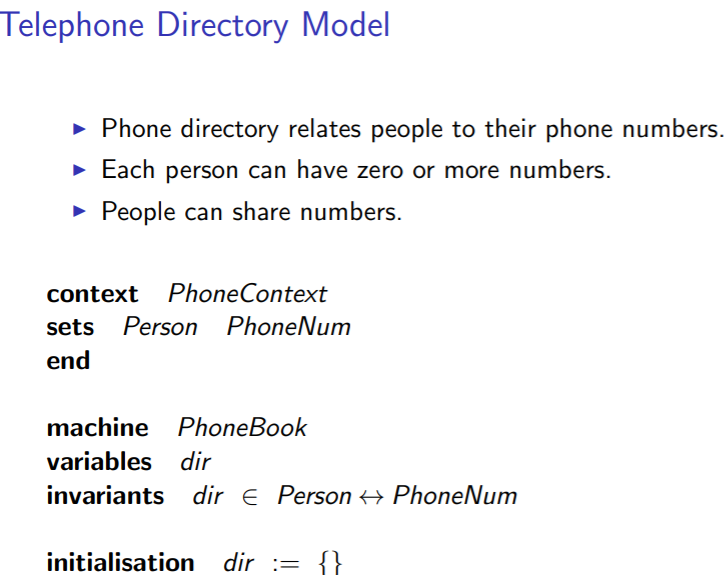
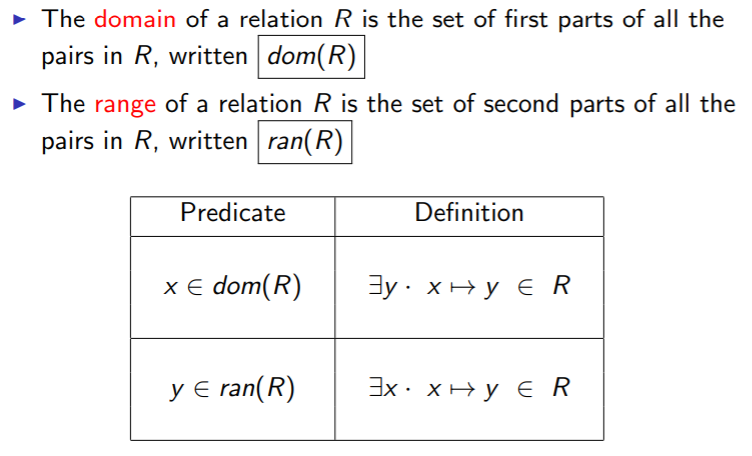
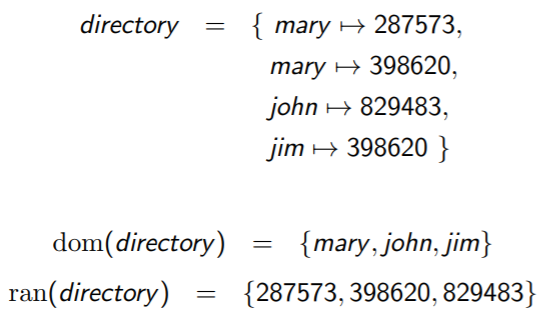
A relation is a common modelling structure so Event-B has a special notation for it:



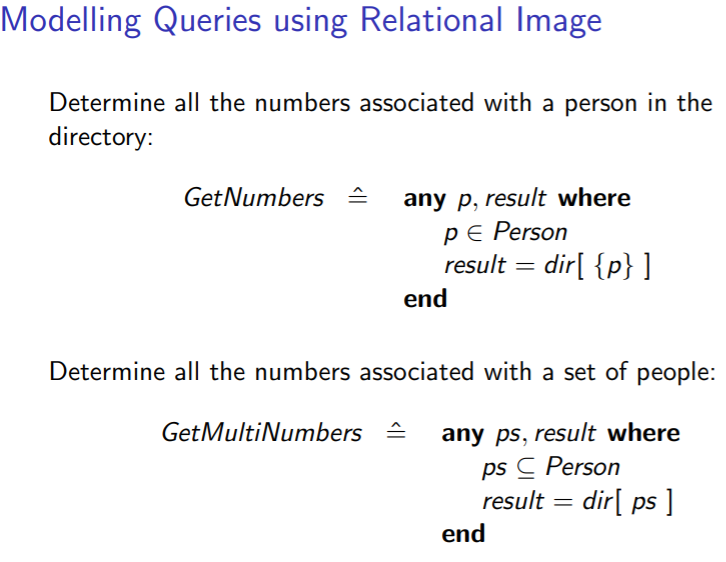
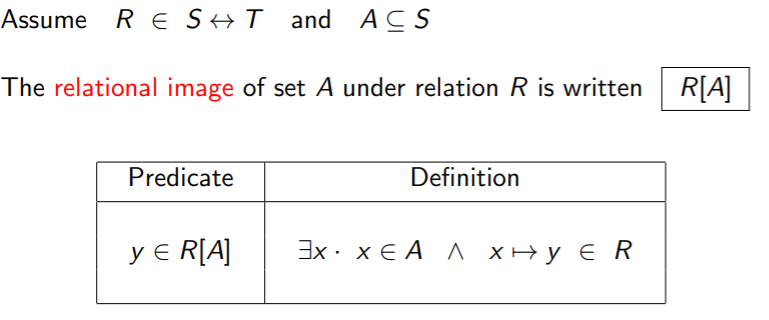
So we can write:



**Domain and range**



**Relational image definition**



*Lecture 9:*

**Powerset** is the type constructor for sets of elements.

**Cartesian product** is the type constructor for pairs of elements.

A **relation** is a set of pairs:

* Domain and range of a relation.
* Relational image.
* Restriction and subtraction.

A function is a special case of a relation:

* Many-to-one: each domain element mapped to a unique range element.
* Partial function, function application.
* Function override.
* Total functions.

**Partial functions**

Special kind of relation: each domain element has **at most one range element** associated with it.

To declare f as a partial function:

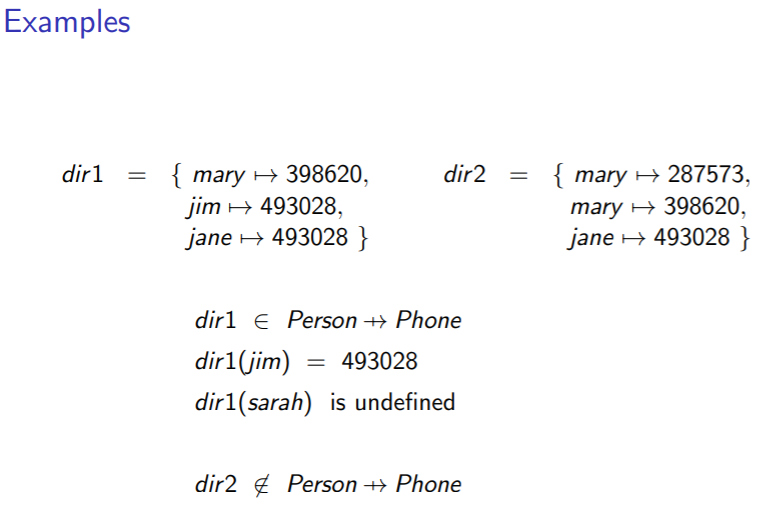
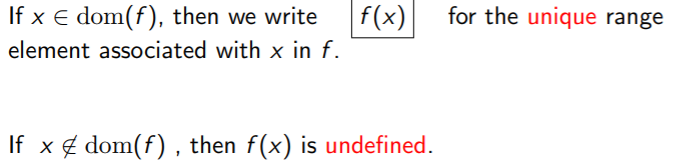


This says that f is a **many-to-one** relation.

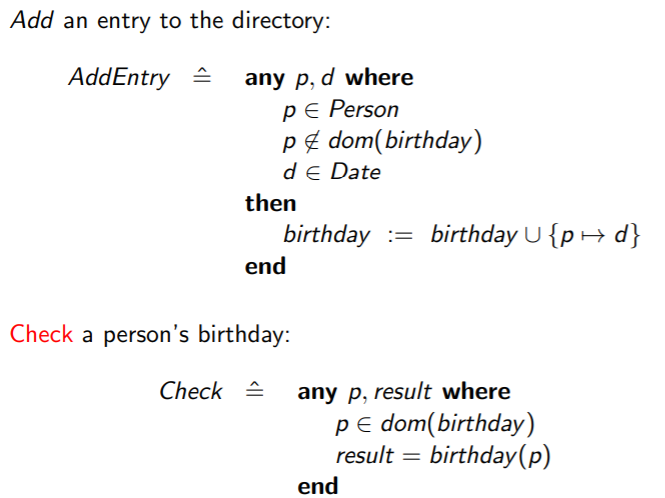
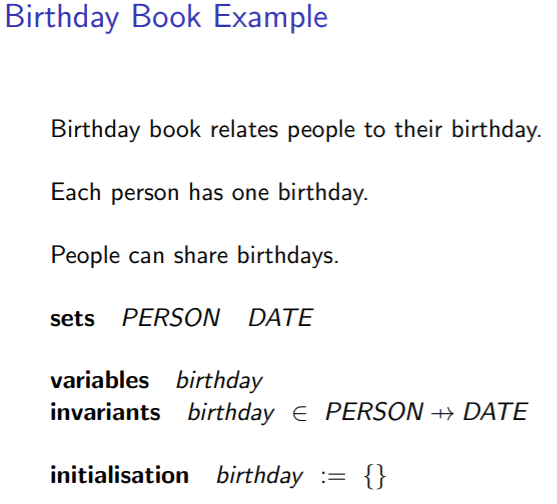
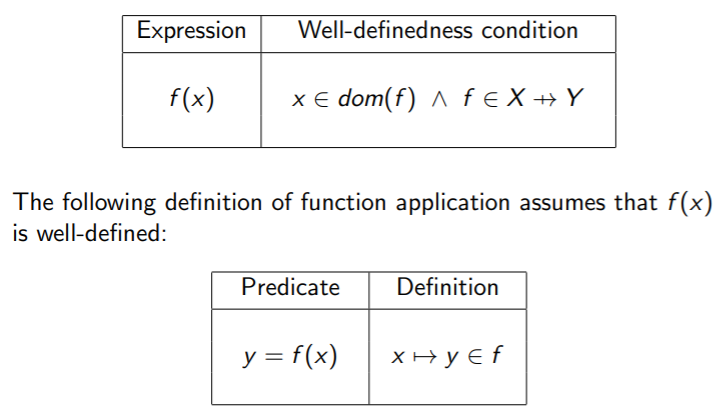
Each domain element is mapped to exactly one range element.

**Function application**

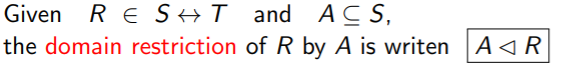
We can use **function application** for partial functions.



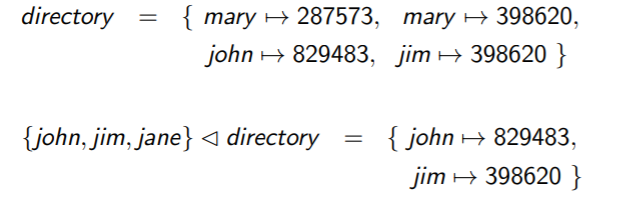
**Well-definedness and application definitions**



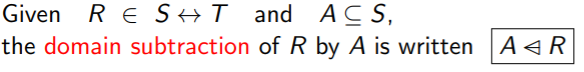
**Domain restriction**



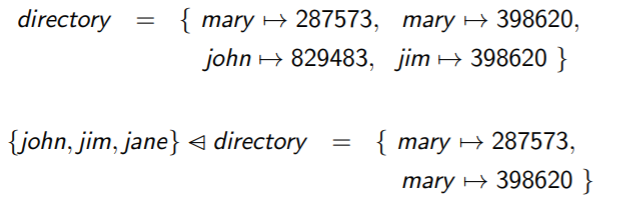
Restrict relation R so that it only contains pairs whose first part is in the set A.



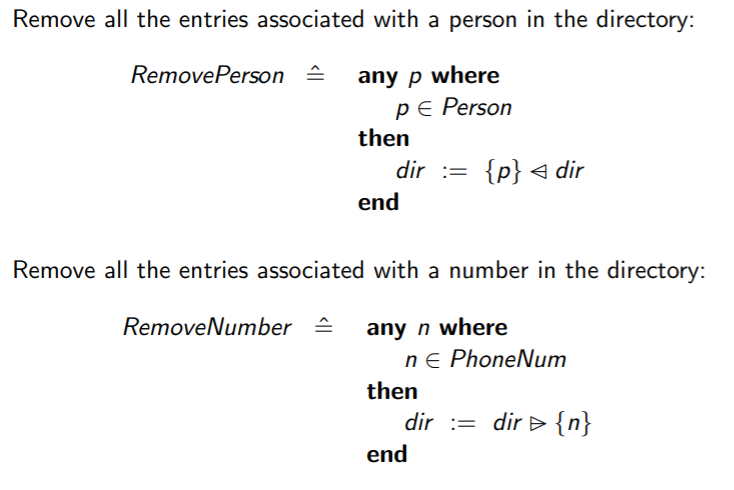
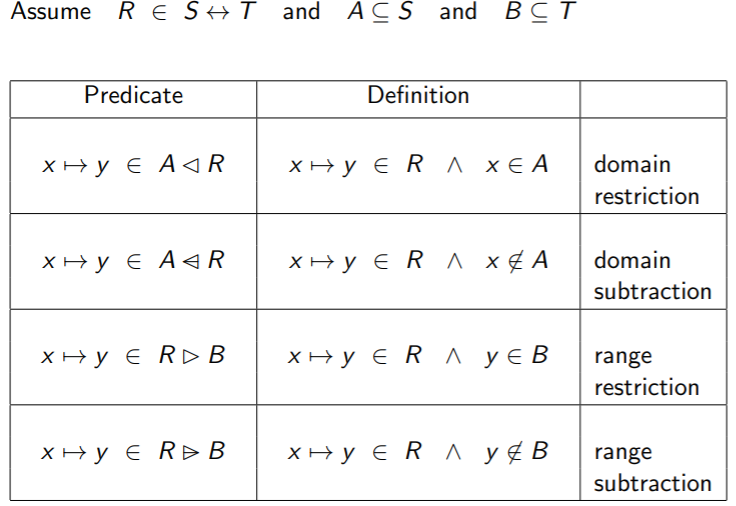
**Domain subtraction**



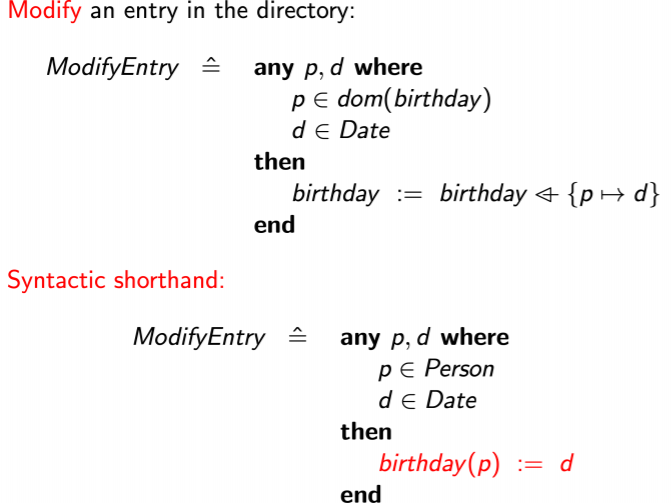
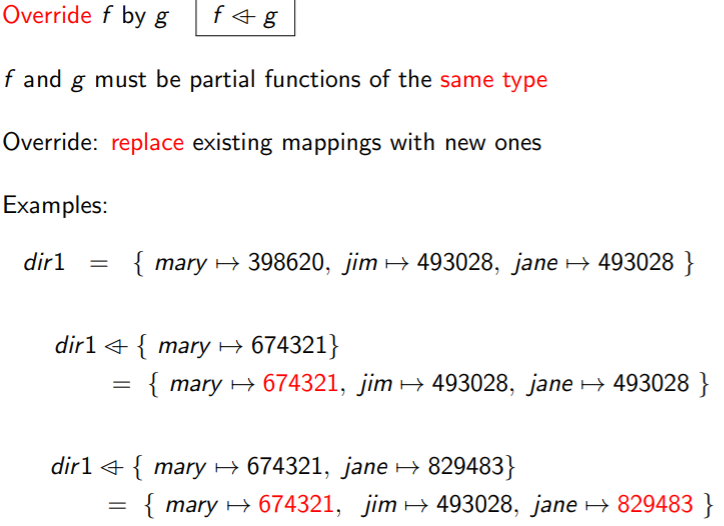
Remove those pairs form R whose first part is in A.



**Domain and Range, Restriction and Subtraction**



**Function overriding**



*Lecture 10:*

**Powerset** is the type constructor for sets of elements.

**Cartesian product** is the type constructor for pairs of elements.

A **relation** is a set of pairs:

* Domain and range of a relation.
* Relational image.
* Restriction and subtraction.

A function is a special case of a relation:

* Many-to-one: each domain element mapped to a unique range element.
* Partial function, function application.
* Function override.
* Total functions.

Relational inverse.

Relation composition.

Relation operators apply to functions (with caution).

**Total functions**

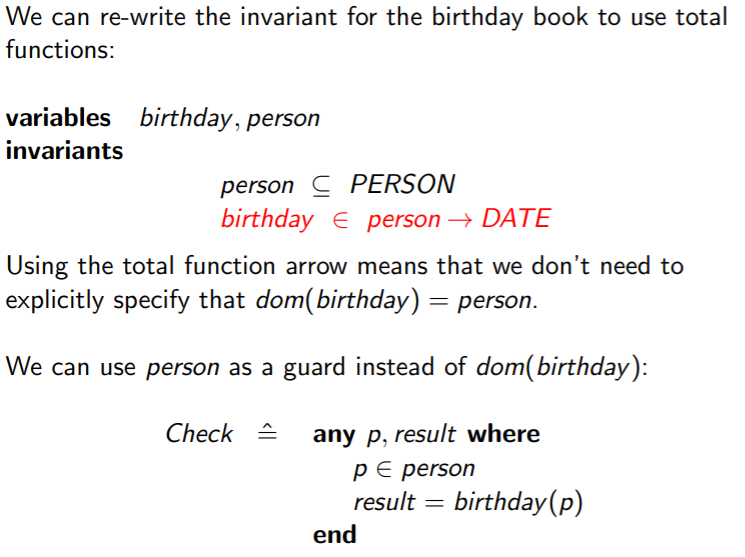
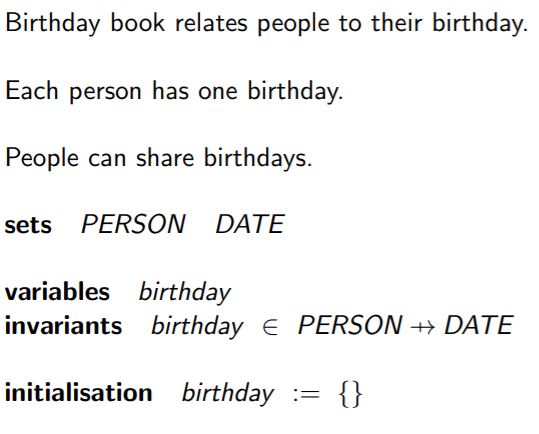
A total function is a special kind of partial function. To declare f as a total function:



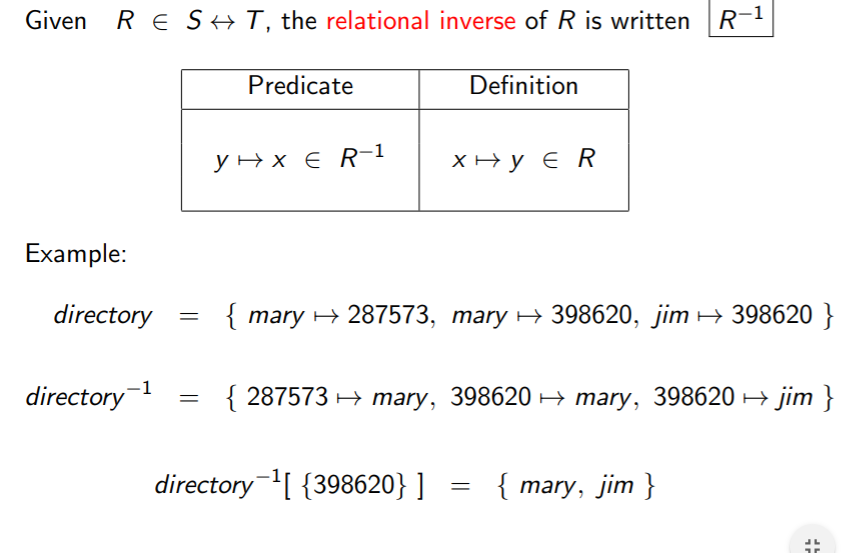
**This means that f is well-defined for every element in X**.

(A well-defined function means that it is mapped to another element)

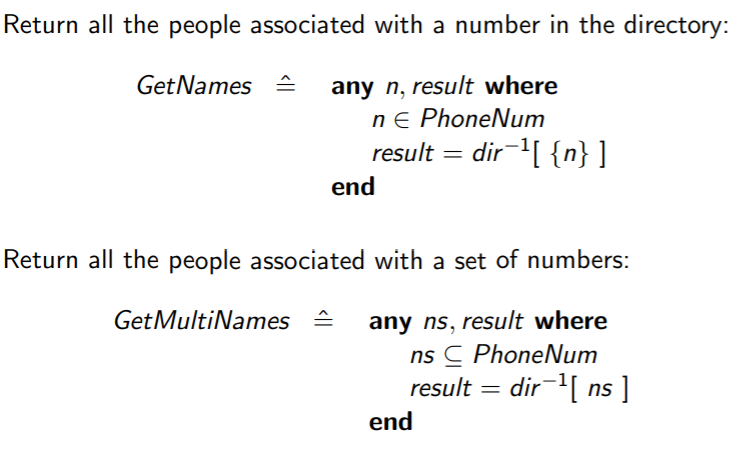
(Total functions mean that every element in X is well-defined, i.e. every element is mapped)



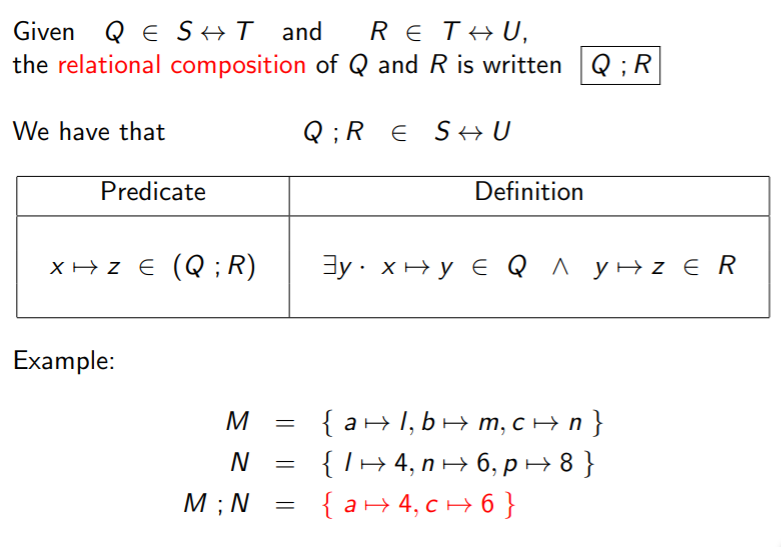
**Relational inverse**



**Inverse Queries**

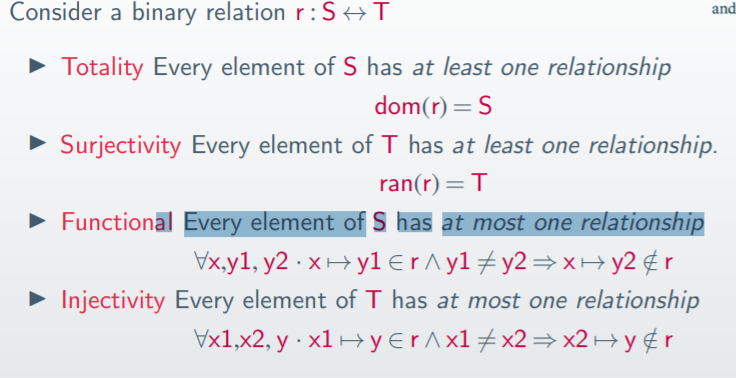


**Relational composition and Image**



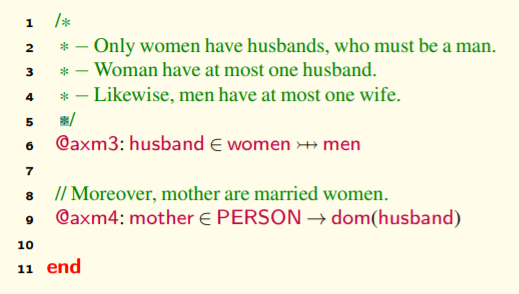
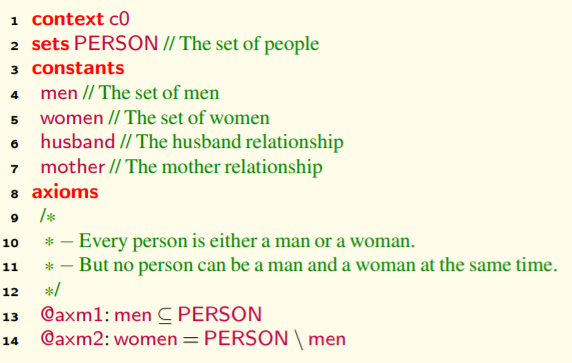
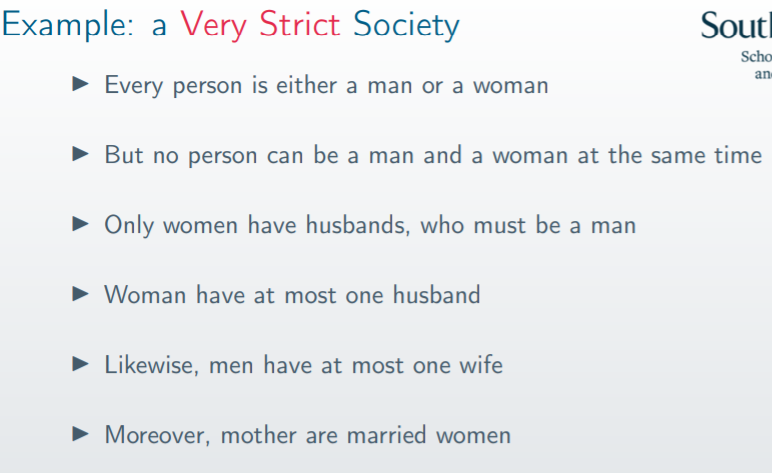
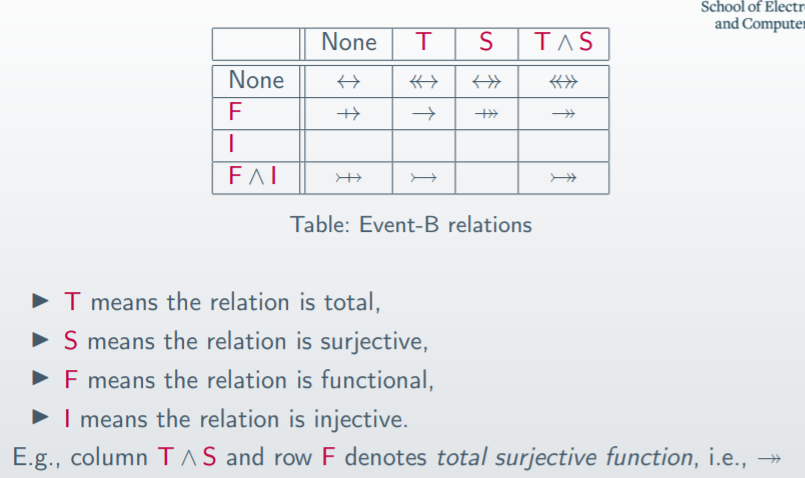
*Lecture 11:*

**Different properties of relations**



Note how totality vs. surjective, functional vs. injectivity are symmetric properties.

**Types of relations in Event-B**

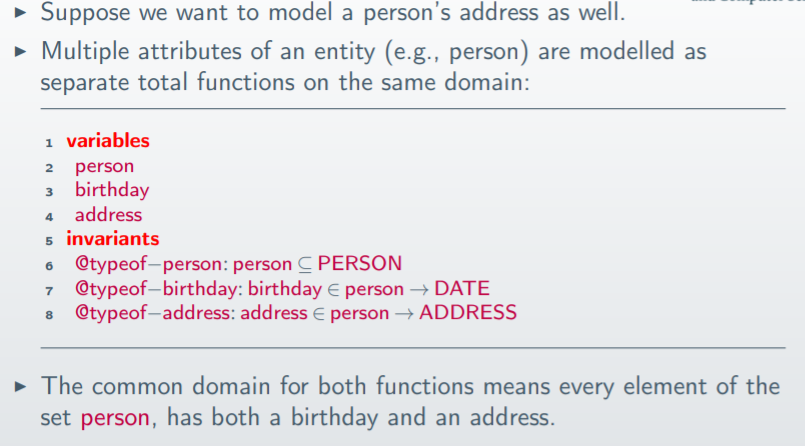
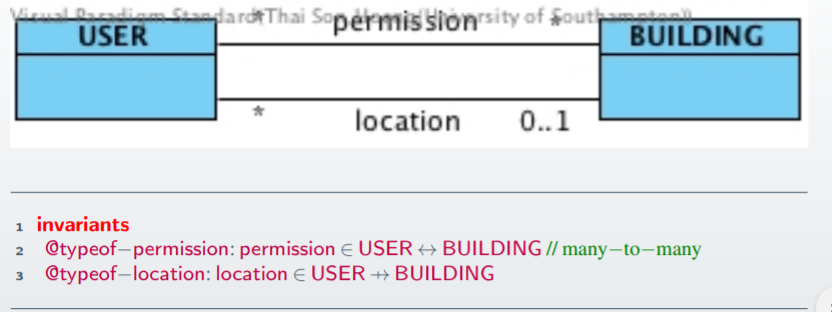
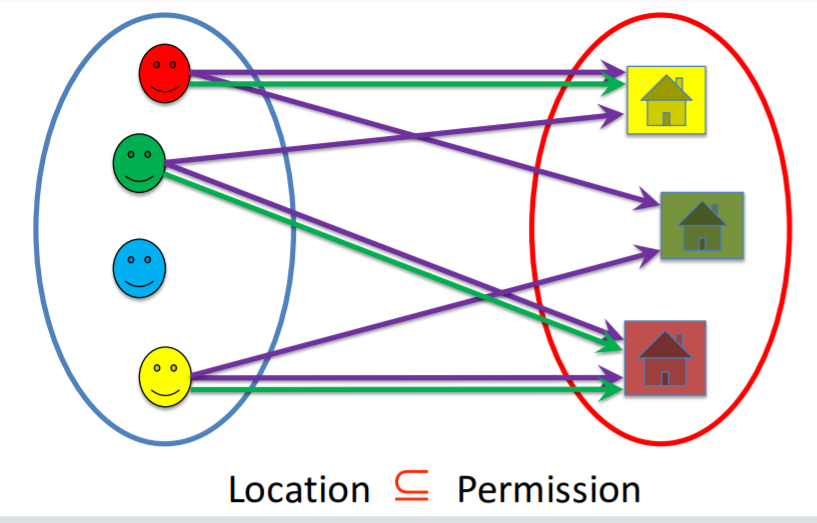
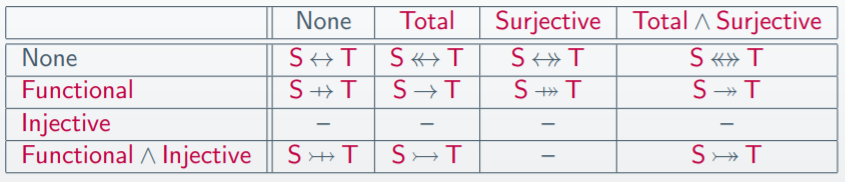


More examples in presentation

*Lecture 12:*

**Relations again**

* **Totality:** Every element of **S** has at least one relationship.
* **Surjectivity:** Every element of **T** has at least one relationship.
* **Functional:** Every element of **S** has at most one relationship.
* **Injectivity:** Every element of **T** has at most one relationship.



**LOOK AT SLIDES FOR PROPER EXAMPLES**

*Lecture 13:*

**Abstraction** – Abstraction can be viewed as a process of **simplifying our understanding** of a system.

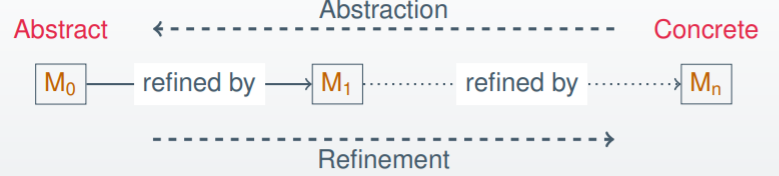
The simplification should:

* Focus on the **intended** **purpose** of the system.
* **Ignore** **details** **of** **how** that purpose is achieved.

The modeler needs to make judgements about what they believe to be the **key** **features** of the system.

**Refinement** – Refinement is a process of enriching or modifying a model to:

* **Augment** the functionality being modelled, or
* **Explain** **how** some purpose is achieved.



*M1 is a refinement of M0 & M0 is an abstraction of M1*

**Facilities abstraction** – We can postpone treatment of some system features to later refinement steps.

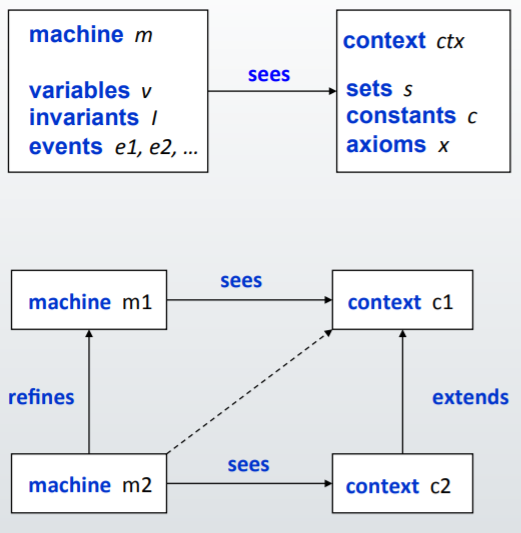
**Coping with System Complexity** – Abstraction and refinement together should allow us to **manage system complexity** in the design process.

**Properties preservation** – Properties are **preserved** during refinement.



**Refinement**

* Preserve **safety** (e.g. invariants) properties.
* We use proofs to **verify** the consistency of a refinement step.
* Failing proofs help **identify inconsistencies** in a refinement step.



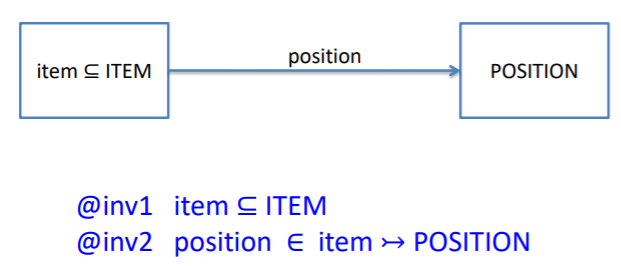
**AGAIN LOOK AT PRESENTATION FOR MORE EXAMPLES**

*Lecture 14:*

Set (unordered collection) and Lists (ordered collections).

In a list, the first item has lowest position number and each item has a single position.

That is why, to define a List we use **Injective mapping (one to one).**



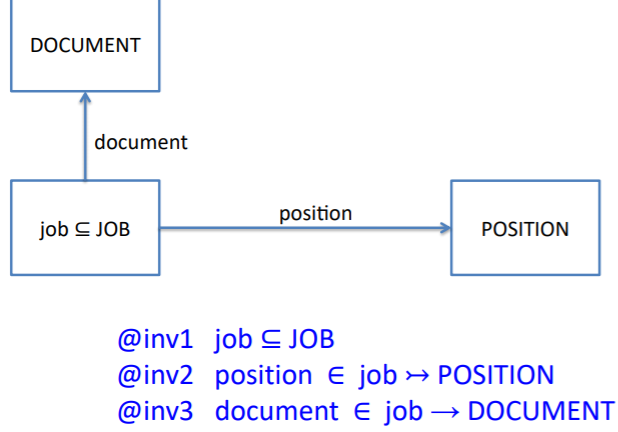
**Queues**

Queues are useful for **managing** **access** **to** **shared resources** in a fair way.

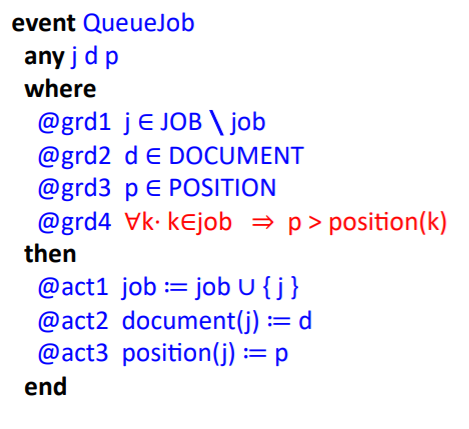
A **queue** can be viewed as a **list**.

Modelling a Printer Queue:

Each job has a single position, but jobs can have the same documents (functional mapping, each document has a single position).

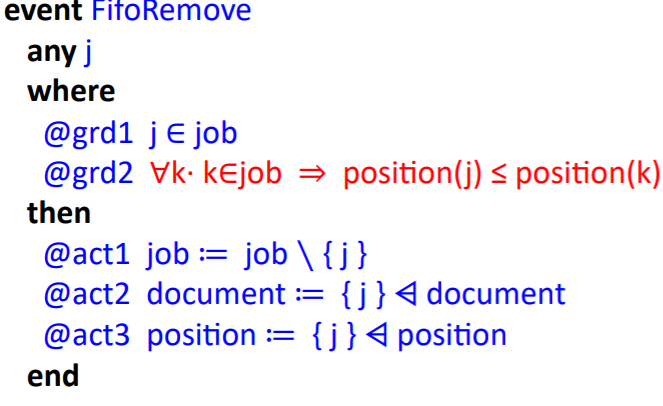


Adding a job to the queue



(Red: p is greater than all current positions on the queue)

FIFO removal of job from queue.



RED: j is at the lowest position of the queue

Same for LIFO, just have >= instead of <=.

RECAP:

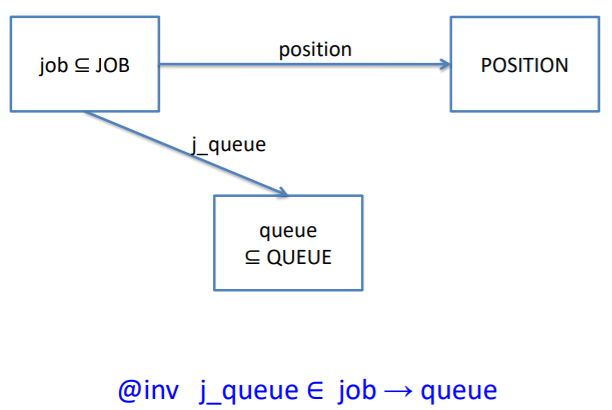
Lists can be modelled as an injective function between items and integers.

Position can be used to select the highest or lowest position.

*Lecture 15:*

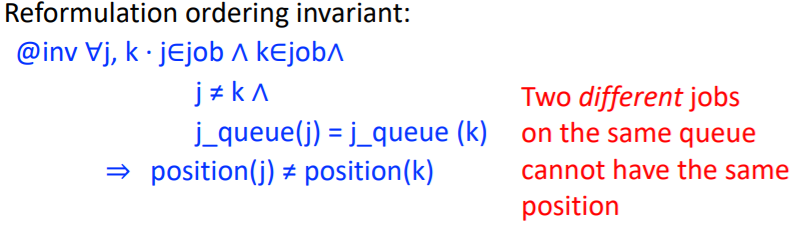
**Collections of lists**

Rather that managing a single print queue, we want to model a system that manages **collection** of queues. (That is done by introducing a carrier set QUEUE to distinguish between different queues.) (We also associate each job with a queue as well as a position).

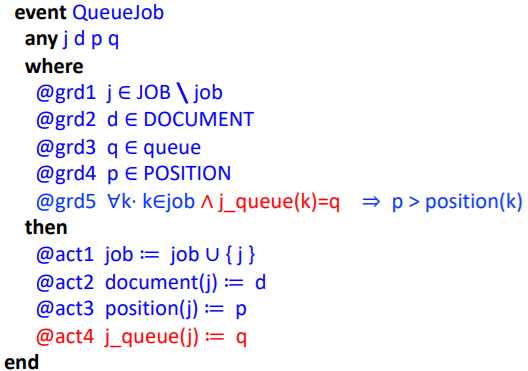


Ordering within a queue -> for a single queue we had injectivity, so no two jobs had the same position.

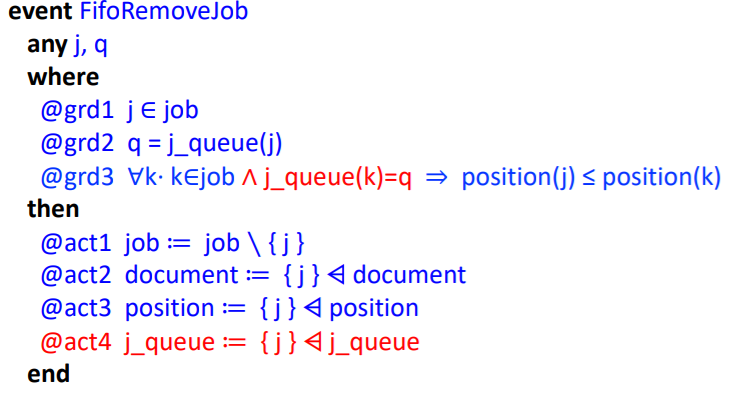
For managing multiple queues, this injectivity is too strong. We are required to use it only within each queue.



Adding a job to a queue:



Removing a job from a queue:



Recap: A collection of queues modelled by introducing explicit queue identifier. (position is injective within each queue).