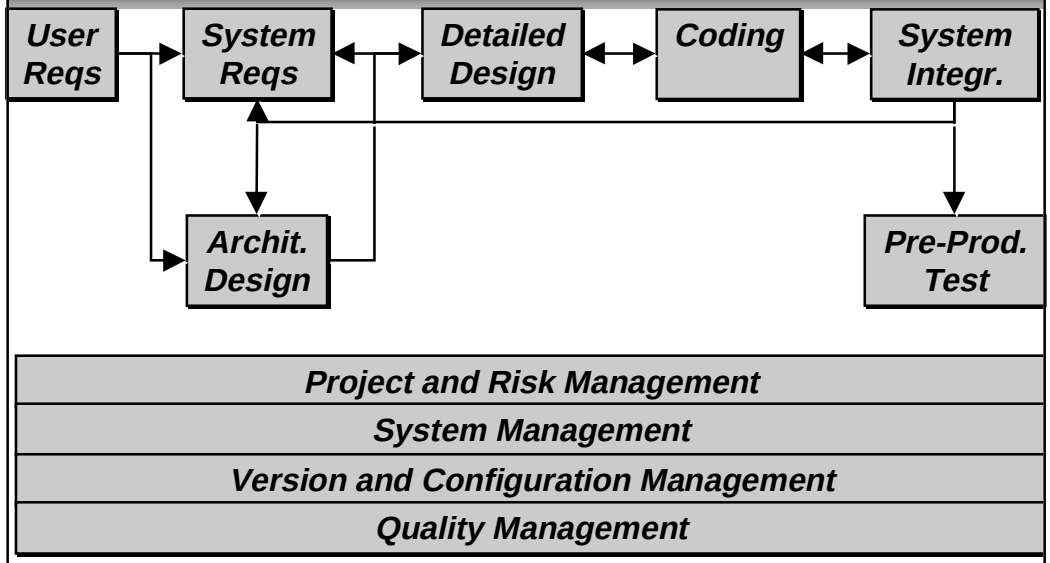
Communications Software Engineering Analysis Model

Wolfgang Emmerich

Lecture Overview

- *Aims of Analysis Model*
- *Building on outputs of Requirements Model*
- *Basic UML notations*
- *Introduction of new types of analysis objects*
- *Reuse of use cases*
- *Inputs for Design Model*

Stages of Development Process



System Reqs Document

Owner: Chief Business Analyst

1 UML Use Case Model (Rose) +

2 UML Interaction Diagrams (Rose) +

- *UML Sequence Diagrams*
- *UML Collaboration Diagrams*

3 UML Analysis Class Diagram (Rose) +

Analysis Model

*4 User Interface Prototypes (GUI Tools, Office) **

5 Specification of Report Format (Office)

6 User Manual (Office)

- *User Reference Manual*
- *User's Guide*

Aims of the Analysis

- *To provide a 'logical model' of the system, in terms of :*
  - *classes,*
  - *relationships*
- *"How to get the thing right, now and in the future "*

Producing an Analysis Model

- *Inputs*
- *Actions*
  - *10 Draft initial class diagram*
  - *11 Re-examine use case and object behaviour*
  - *12 Refine class diagram*
  - *13 Execute check*
  - *14 Revise class diagram*
  - *15 Group classes into packages*
- *Outputs*
- *Notations*

Analysis Input and Outputs

- *Inputs:*
  - *uses cases and use case model*
  - *problem domain object list*
- *Outputs:*
  - *class roles and responsibilities [text]*
  - *use case description in terms of classes and operations [text x use case]*
  - *completed analysis model [class and package diagrams]*

Notations

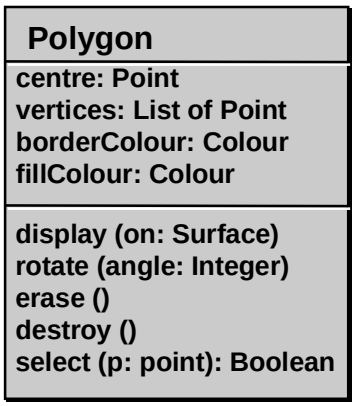
- *Class (rectangle containing name, attributes, operations)*
- *Object (rectangle plus obx:Cx)*
- *Association (by value/aggregation, cardinality/multiplicity)*
- *Generalisation (UML term replacing OOSE 'inheritance')*
- *Package*
- *Depends association*

CLASSES IN UML

className

attribute name: type

Op name (parameter: type): result type



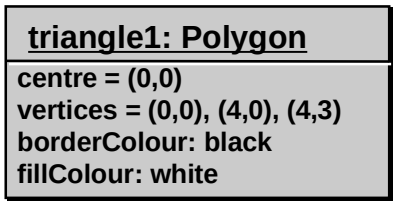
className

Objects in UML

objectName: className

attribute name: type = value

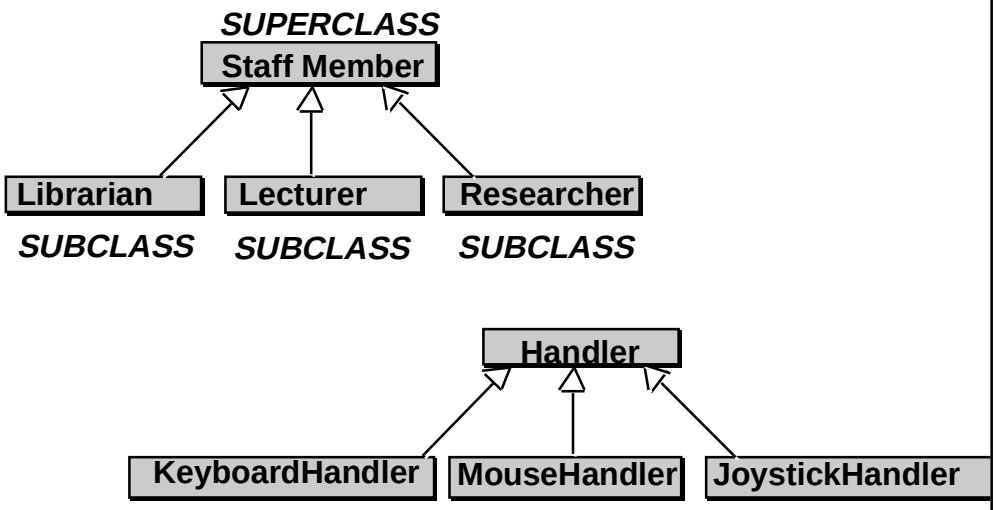
(same operations for all instances of a class)



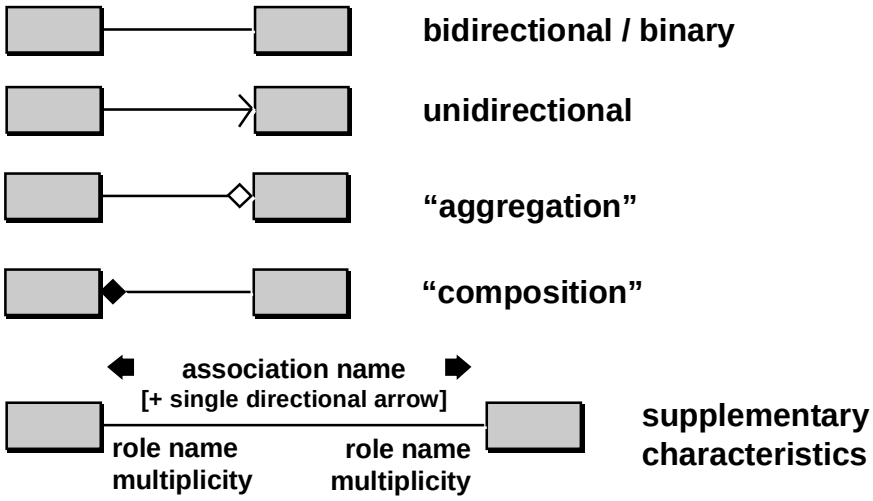
display (on: Surface)
rotate (angle: Integer)
erase ()
destroy ()
select (p: point): Boolean

objectName: className

UML Generalization



Associations in UML



Composition vs Aggregation

■ Composition

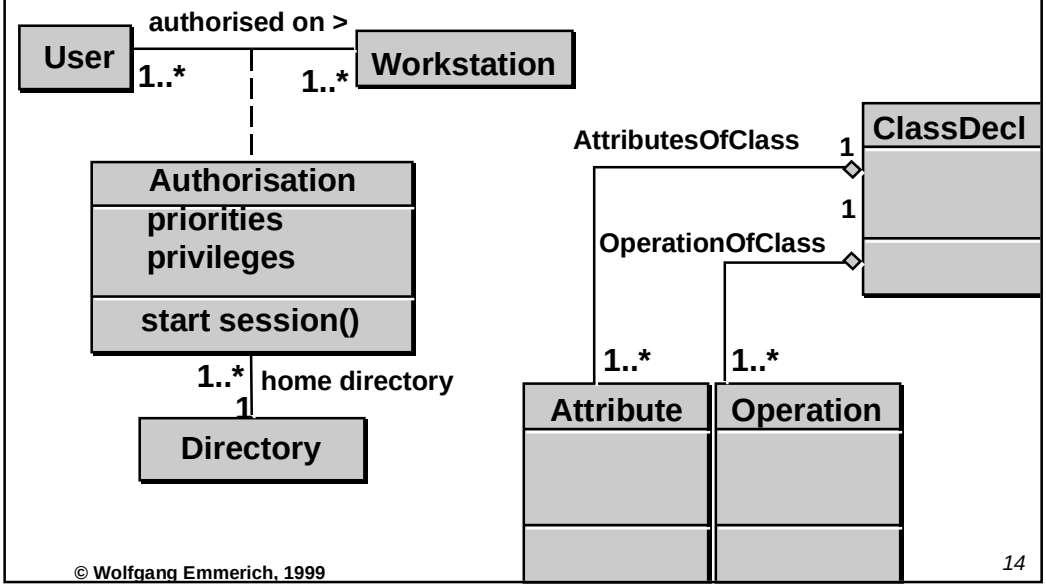
```
class Address {
  char *street;
  char * plz;
}
class Person {
  char * name;
  Address adr;
}

Person p = new Person()
```

■ Aggregation

```
class Address {
  char *street;
  char * plz;
}
class Person {
  char * name;
  Address *adr;
  Person(){
    adr=new Address();
}
Person p=new Person;
```

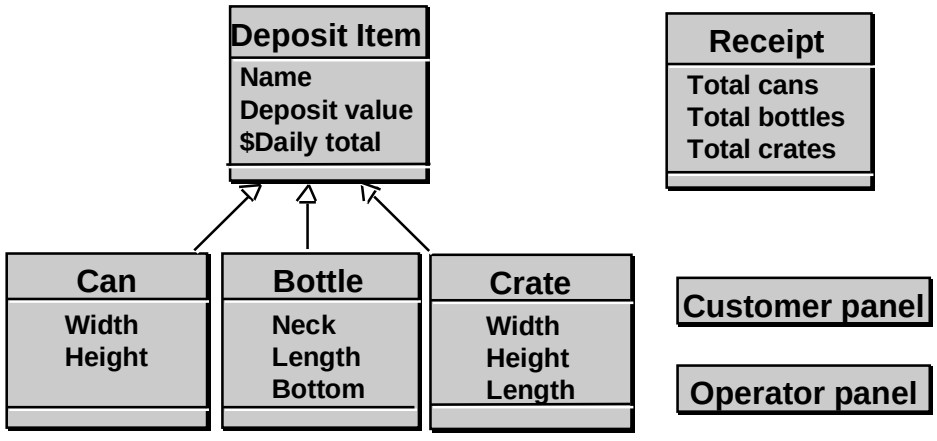

Association Examples in UML



Class Diagrams in UML

- *Class diagrams :*
  - *show logical, static structure of system*
  - *provide core of 'unified model'*
- *Generation of initial class diagram from problem domain object list*
  - *classes of objects*
  - *associations / attributes*
  - *generalization relationships*

A 'Recycling Machine' Class Diagram

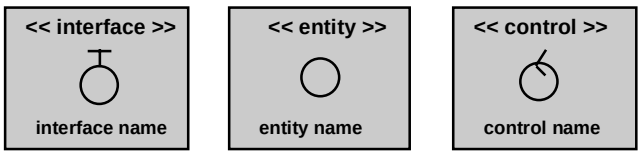


Exploiting Use Cases

- *Employ classes and use cases, one by one,*
  - *to describe roles and responsibilities of each class*
  - *to distribute behaviour specified in use cases*
  - *to ensure that there is a class for every behaviour*

Roles of Classes in OOSE

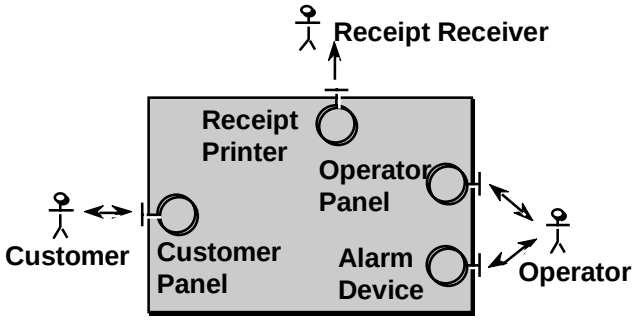
- *Interface classes - for everything concerned with system interfaces*
- *Entity classes - for persistent information and behaviour coupled to it*
- *Control classes - for functionality not normally tied to other classes*



- *Integrated into UML as stereotypes:*

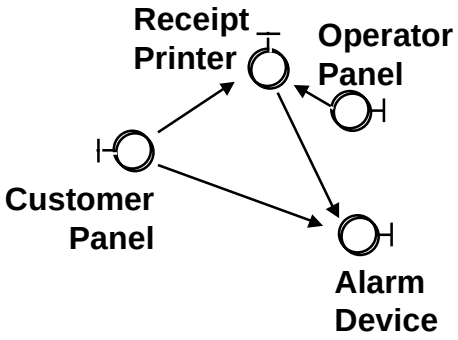
Interface Classes

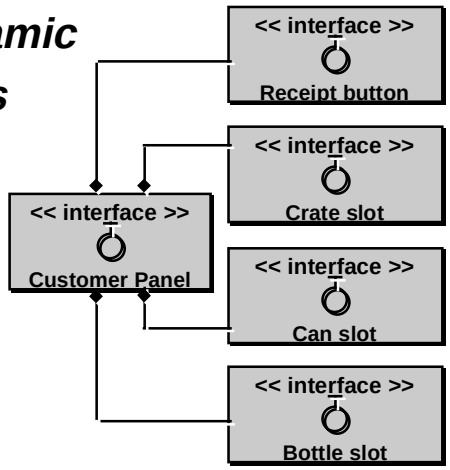
- *Contain functionality directly dependant on system environment*
- *Definitions focus on interaction between actors and use cases*



Associations between Interface Classes

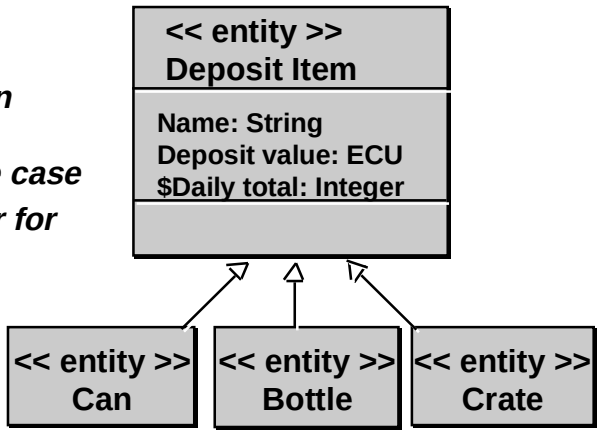
- *Definition of both dynamic*
- *and static associations*





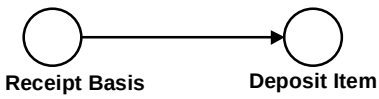
Attributes of Entity Classes

- ***Purposes of entity classes :***
  - ***To store information persisting after completion of a use case***
  - ***To define behaviour for manipulating this information***



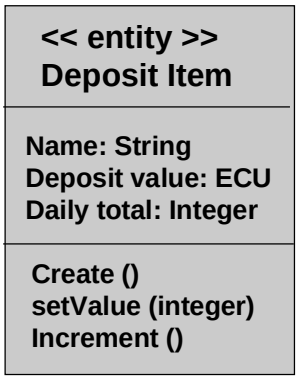
Entity Communication

- ***A primary task to identify associations involving communication***
  - ***modelling of communication between objects***
  - ***shows the sending and receiving of messages as stimuli***
  - ***starts from object initiating communication***
  - ***directed to object where reply generated or operation executed***



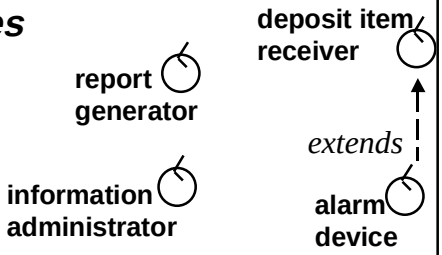
Entity Operations

- ***Defining entity operations for:***
  - ***storing and fetching information***
  - ***creating and removing object***
  - ***behaviour that must change if entity object is changed***



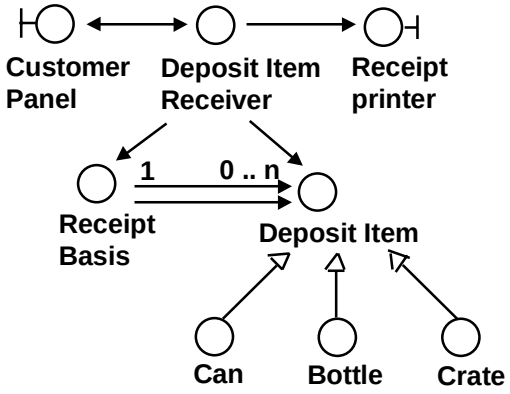
Control Classes

- ***Control classes needed to provide for:***
  - ***behaviour not natural in interface and entity classes***
  - ***'glue' between other classes in use case***
  - ***typical control behaviours***
  - ***improved maintainability***



Use Case View

- *Model each use case*
- *Describe use case in terms of classes*



An elaborated Use Case

When the customer returns a deposit item the Customer Panel's sensors measure its dimensions. These measurements are sent to the control object Deposit Item Receiver which checks via Deposit Item whether it is acceptable. If so, Receipt Basis increments the customer total and the daily total is also incremented. If it is not accepted, Deposit Item Receiver signals this back to Customer Panel which signals NOT VALID.

When the Customer presses the receipt button, Customer Panel detects this and sends this message to Deposit Item Receiver. Deposit Item Receiver first prints the date via Receipt Printer and then asks Receipt Basis to go through the customer's returned items and sum them. This information is sent back to Deposit Item Receiver which asks Receipt Printer to print it out.

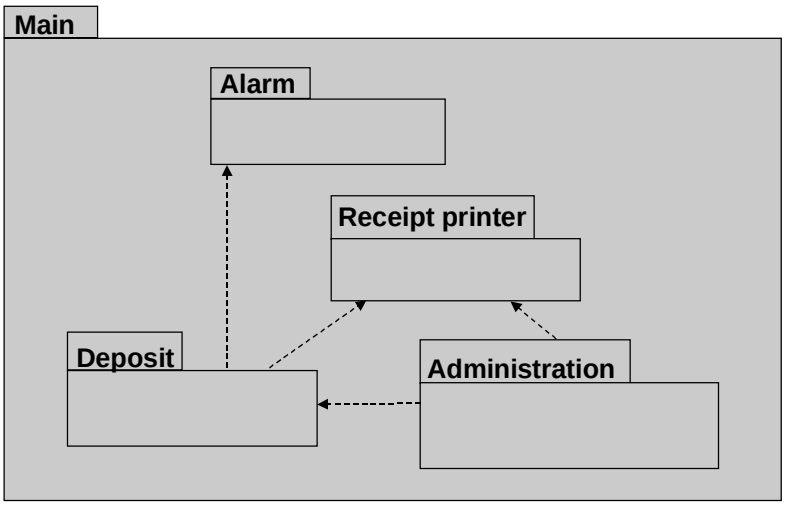
Packages

- *Packages are a way to decompose class diagrams into manageable units*
- *Packages are necessary:*
  - *because of large numbers of classes*
  - *to provide optional functionality*
  - *to minimise effect of change*

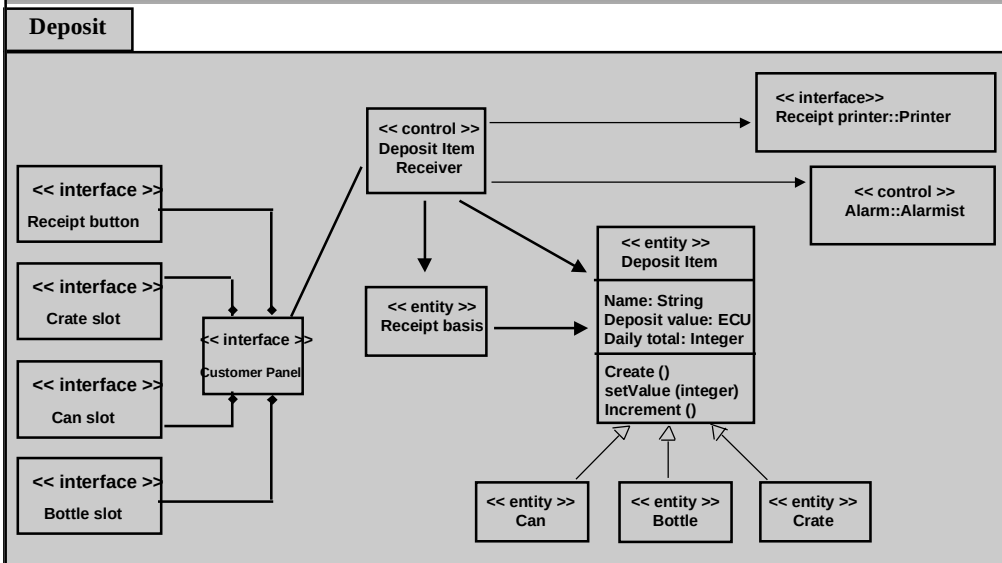
Packages (Cont'd)

- *Packages should have a:*
  - *tight functional coupling inside*
  - *weak coupling outside indicated by 'dependency associations' between packages*
- *Packages may:*
  - *'contain' nested packages with 'service packages' as atomic parts*
  - *have individual classes outside*
  - *be result of organisational or managerial pressures*

Recycling Machine Packages



'Deposit' Package in UML



Analysis Model

- *Outputs:*
  - *class roles [text]*
  - *use case description in terms of classes and operations [text x use case]*
  - *completed analysis model classes [diagram]*
  - *sub-system diagrams [package diagram]*
- *Notations introduced:*
  - *class, object, associations*
  - *(binary, unidirectional, aggregation, generalisation)*
  - *stereotypes (classes, associations)*
  - *package (+ dependency association)*

Analysis in OOSE - Summary

- *USER PERSECTIVE*
  - *Actors and Use Cases*
  - *Strong emphasis on requirements modelling*
  - *Resistence to effects of change*
- *ADVANTAGES OVER OTHER METHODS*
  - *Ways to identify and define classes and objects*
  - *Effective and useful identification of roles of classes*
  - *Recognition of user role (and interface)*
  - *Refined with practical use*

Modelling Processes

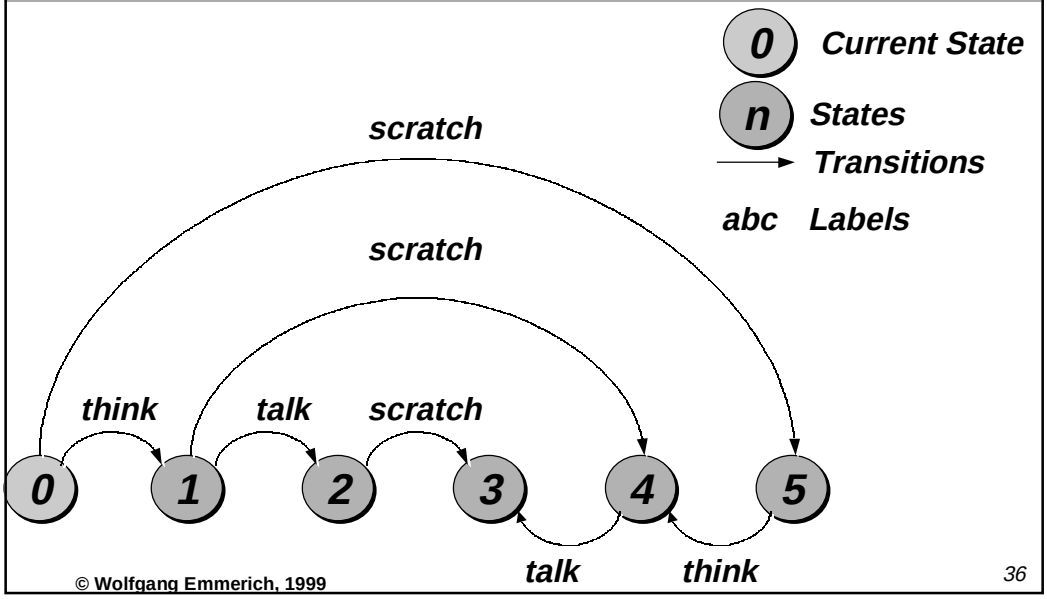
Processes and Threads

- *Execution of a program is a process*
- *Concurrent programs consist of multiple processes*
- *Threads are lightweight processes*
- *Both threads and processes can be modelled in the same way*
- *We use finite state machines for that*

Labelled Transition Systems

- *Special form of finite state machines*
- *Used to model states of concurrent programs and transitions between them*
- *LTS:=(S,T,A,δ,c) where*
  - *S (a finite set of states)*
  - *$T \subseteq S \times S$ (a finite set of transitions)*
  - *A (an alphabet of atomic actions)*
  - *$\delta: T \rightarrow A$ (a transition labelling)*
  - *$c \in S$ (the current state)*

Graphic LTS Notation



LTS Semantics

- *All actions that are annotations of transitions starting from the current state are enabled*
- *If process engages in enabled action target of transition becomes current state*

Demo

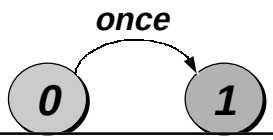
- *In this way LTS determines all possible traces of process*

Finite State Processes (FSP)

- *LTS become unmanageable for large number of states and transitions*
- *Process algebras determine LTSs in a more concise way*
- *Finite State Processes (FSP): machine readable notation for a process algebra*
- *For each FSP model an equivalent LTS can be constructed automatically*

FSP Intro: Action Prefix

- *Let* `x` *be an action and* `P` *a process. The action prefix* `(x->P)` *is process that initially engages in action* `x` *and then behaves in the same way as process* `P`
- *Used to model atomic actions*
- *Actions have lower case identifiers, states have upper case identifiers*
- *Example:* `ONESHOT=(once->STOP).`
- *Equivalent LTS:*

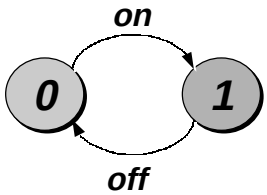


FSP Intro: Recursion

- *Let* `P` *be a process. Then* `P` *may be used in action prefixes in a recursive way.*
- *Used to model repetitive behaviour*
- *Example:*

```
SWITCH=OFF.
OFF    =(on->ON).
ON     =(off->OFF).
```

- *Equivalent LTS:*



- *Note: Processes are equivalent to states*

FSP Intro: Local Processes

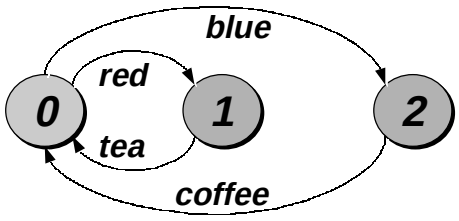
- *It is not necessary for all states/processes to be globally visible.*
- *Restricting states/processes by use of ‘,’*
- *Example:*
  `SWITCH=OFF,`
  `OFF=(on->ON),`
  `ON=(off->OFF).`
- `OFF` *and* `ON` *are not visible outside* `SWITCH`
- *Equivalent to:*
  `SWITCH=(on->off->SWITCH).`

FSP Intro: Choice

- `(x->P|y->Q)` *describes a* <u>*choice*</u> *that engages either in* `x` *or* `y`*. After* `x` *it contin-ues with* `P`*, after* `y` *it continues with* `Q`
- *Example:* `DRINKS=(`

```
          red->tea->DRINKS
        | blue->coffee->DRINKS
        ).
```

- *Equivalent LTS:*



FSP Intro: Indexes

- *A range type is a finite and scalar type:*
- *Example:* `range T=0..3`
- *If T is a range type then x[i:T] is the declaration of an action index and P[i:T] is declares an indexed process.*
- *A process index variable is valid within the process, an indexed action is valid within the scope of the choice.*

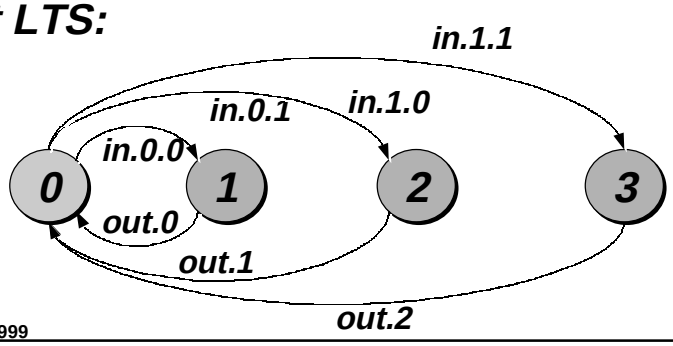
range T=0..5

X=X[0],
X[i:T]=(inc->X[i+1]).

FSP Intro: Index Example

```
const N =1
range T =0..N
range R =0..2*N
SUM     =(in[a:T][b:T]->OUT[a+b],
OUT[s:R]=(out[s]->SUM).
```

- *Equivalent LTS:*

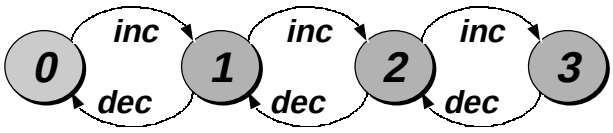


FSP Intro: Guarded Actions

- *The guarded action* `when B x->P` *means that when the guard* `B` *is true action* `x` *is enabled and the process proceeds as* `P`*.*
- *Example:*

```
COUNT(N=3) =COUNT[0],
COUNT[i:0..N]=(when(i<N) inc->COUNT[i+1]
              |when(i>0) dec->COUNT[i-1]
              ).
```

- *Equivalent LTS:*



Summary

- *Formal Definition of LTS*
- *Algebraic notation in FSP*
- *Equivalence between LTS and FSP*
- *FSP and LTS concepts introduced so far are sufficient for sequential programs*
- *Next session: FSP constructs for modelling concurrent programs*

Modelling Concurrency in FSP

What do we have to model?

- *Relative or absolute speed?*
  - *Neither!*
- *Concurrency or parallelism?*
  - *Interleaved model of concurrency!*
- *Relative order of actions?*
  - *Arbitrary interleaving!*
- *We use an asynchronous model of execution!*

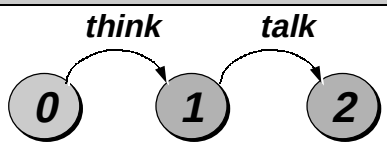
FSP: Parallel Composition

- *If* `P` *and* `Q` *are processes then* `(P||Q)` *denotes the parallel execution of* `P` *and* `Q`
- `||` *is used to model parallel composition of processes*
- *Names of concurrent processes are preceeded by* `||`
- *Example:*

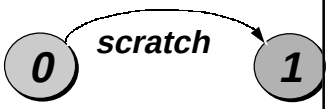
```
CONVERSE =(think->talk->STOP).
ITCH =(scratch->STOP).
||CONVERSE_ITCH =(ITCH || CONVERSE).
```

Equivalent LTSs

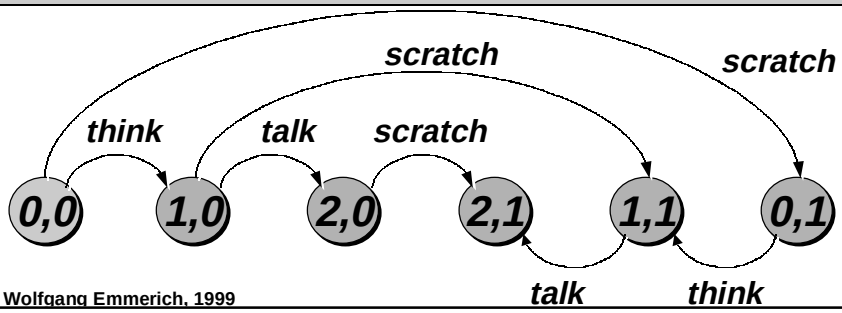
```
CONVERSE =(think->talk->STOP).
```




```
ITCH=(scratch->STOP).
```




```
||CONVERSE_ITCH =(ITCH || CONVERSE).
```



Properties of Parallel Composition

- *Parallel composition operator has two important algebraic properties*
- *Commutativeness*
  - `(P||Q)=(Q||P)`
  - *ordering is not important!*
- *Associativeness*
  - `((P||Q)||R)=(P||(Q||R))=(P||Q||R)`
  - *brackets can be omitted!*

FSP: Process Interactions

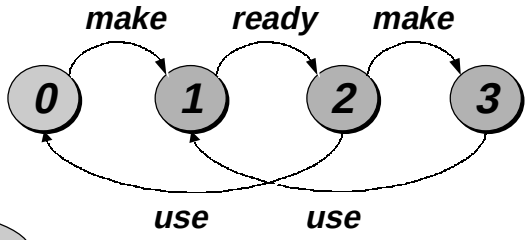
- *Concurrent processes that share actions <u>interact</u> with each other*
- *Used to model synchronisation*
- *Example:*

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).
||MAKER_USER = (MAKER || USER).
```

- *Product has to be ready before it can be used.*

Equivalent LTS

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).
||MAKER_USER = (MAKER || USER).
```



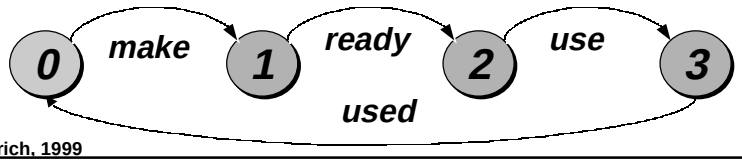
Demo

Handshake

- *An action that is acknowledged by another action is referred to as handshake*
- *Widely used to structure process interactions*
- *Example:*

```
MAKERv2=(make->ready->used->MAKERv2).
USERv2 =(ready->use->used ->USERv2).
||MAKER_USERv2 = (MAKERv2 || USERv2).
```

- *LTS:*



FSP: Process Labelling

- *The process label* `a:P` *prefixes each label in the alphabet of* `P` *with* `a`
- *Avoids name clashes in different instantiations of processes and enables reuse.*
- *Example:*

```
SWITCH = (on->off->SWITCH).
||TWOSWITCH=(a:SWITCH||b:SWITCH).
```

- *Alphabet of* `||TWOSWITCH`*:*

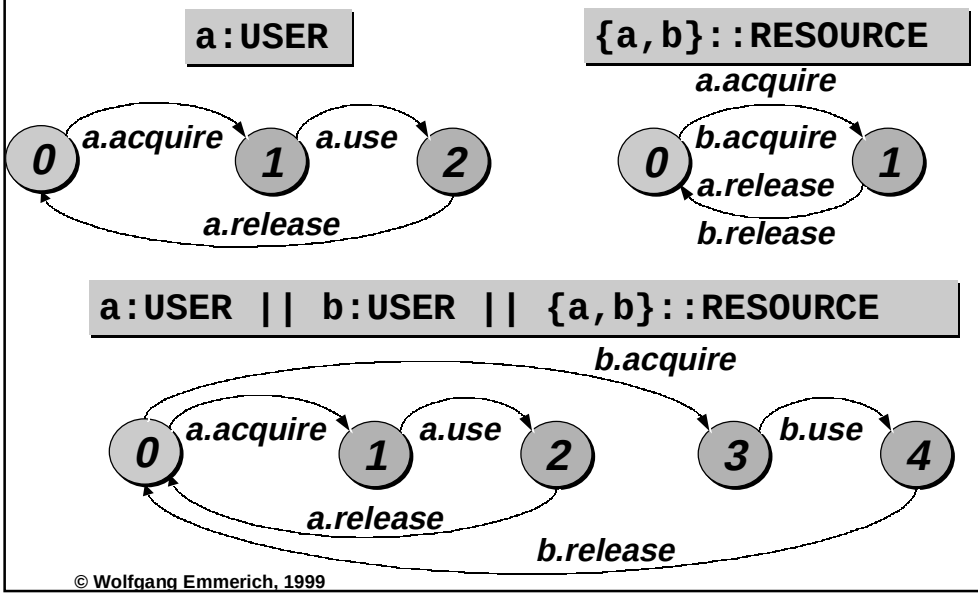
```
{a.on, a.off, b.on, b.off}
```

FSP: Process Labelling (cont'd).

- *The process label* `{a1,..,ax}::P` *replaces every label* `n` *in the alphabet of* `P` *with label* `a1.n,…,ax.n`.
- *Example:*

```
RESOURCE=(acquire->release->RESOURCE).
USER = (acquire->use->release->USER).
||RESOURCE_SHARE =
 (a:USER || b:USER || {a,b}::RESOURCE).
```


Equivalent LTSs



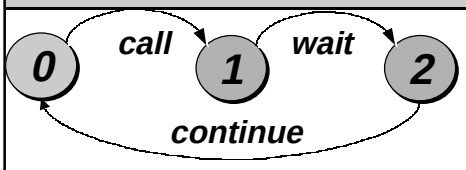
FSP: Relabelling

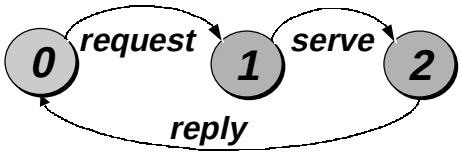
- ***Relabelling functions change names of action labels. The relabelling function is:***
  `/{new1/old1,… newn/oldn}.`
- ***Used to synchronise actions with different names in composite processes.***
- ***Example:***

```
CLIENT=(call->wait->continue->CLIENT).
SERVER=(request->serve->reply->SERVER).
||CLIENT_SERVER = (CLIENT || SERVER)
   /{call/request, reply/wait}.
```


Equivalent LTSs

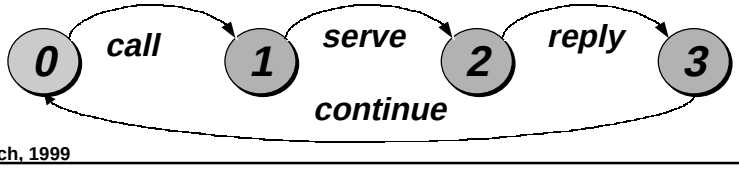
```
CLIENT=(call->wait->
continue->CLIENT).
```






```
SERVER=(request->serve
->reply->SERVER).
```

```
||CLIENT_SERVER = (CLIENT || SERVER)
        /{call/request, reply/wait}.
```



FSP: Hiding

- *The hiding operator* `\{a1..ax}` *removes action labels* `a1..ax` *from alphabet of* `P` *and hides them*
- *Hidden actions are labelled* `tau`
- *Hidden actions in different processes are not shared*
- *Example:*

```
USER=(acquire->use->release->
      USER)\{use}.
```

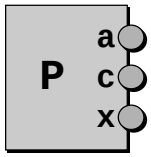

FSP: Interfaces

- *The interface operator* `@{a1..ax}` *hides all actions in the alphabet of* `P` *that do not occur in the set* `a1..ax`*.*
- *Complementary to hiding*
- *Like hiding used to reduce complexity of resulting LTS.*
- *Example:*

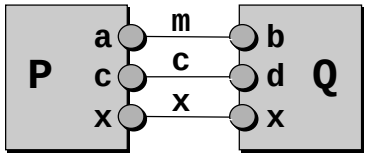
```
USER = (acquire->use->release->
        USER)@{acquire,release}.
```

FSP: Structure Diagrams

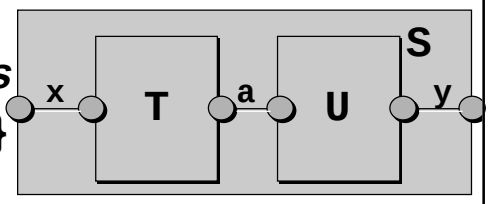
Process P with alphabet `{a,c,x}`



Parallel Composition `(P||Q)/{m/a,m/b,c/d}`



Composite Process `||S = (T||U)@{x,y}`



Summary

- ***Parallel Composition***
- ***Process Interactions***
- ***Process Labelling***
- ***Process Relabelling***
- ***Hiding / Interfaces***
- ***Structure Diagrams***
- ***Solve Exercises of tutorial sheet***