Basic xtUML Modeling

Levels of Commitment

- Natural language and informal diagrams
  - Use cases
  - Activity diagrams
  - Sequence diagrams
- Structural models
  - Components & Interfaces
  - Class models
  - Data types
- Behavioral models
  - State models
  - Activities

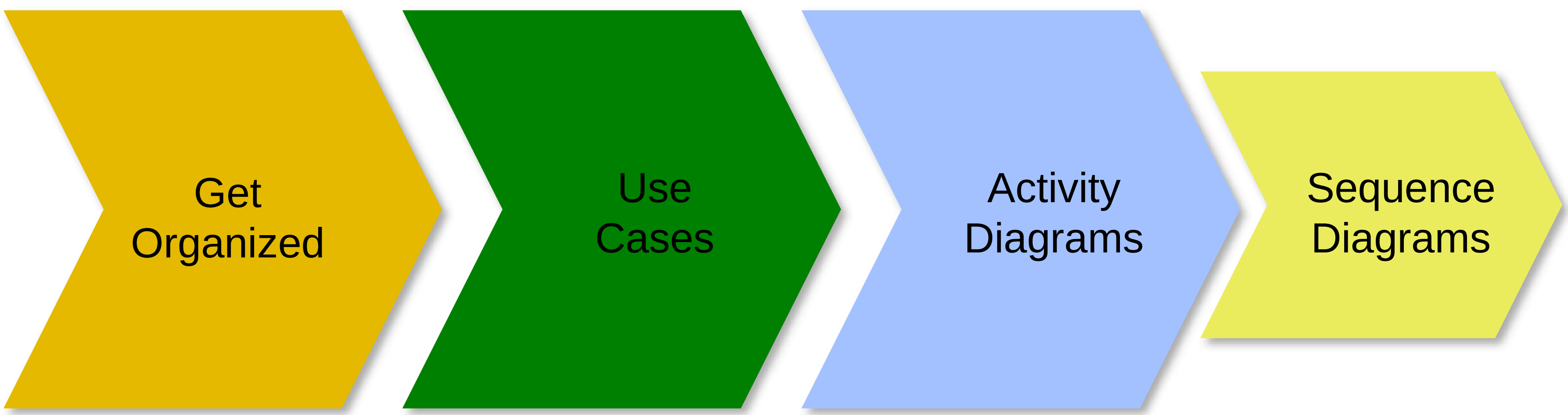
} Prior course

Requirements Clarification Process

The process was:

- Find all your people, resources, practices, etc.
- Find out what the system-as-a-whole does
- Determine the precise behavior of each use case
- And establish how it communicates with others

But it was really all about learning about the problem.



Abstraction

Now that everything is:

- reviewed,
- signed off, and
- it’s all in our heads,

it time to

THINK

And from that thinking, we create, and commit to, *abstractions*.

Levels of Commitment

We represent our abstractions in *models* of various types.

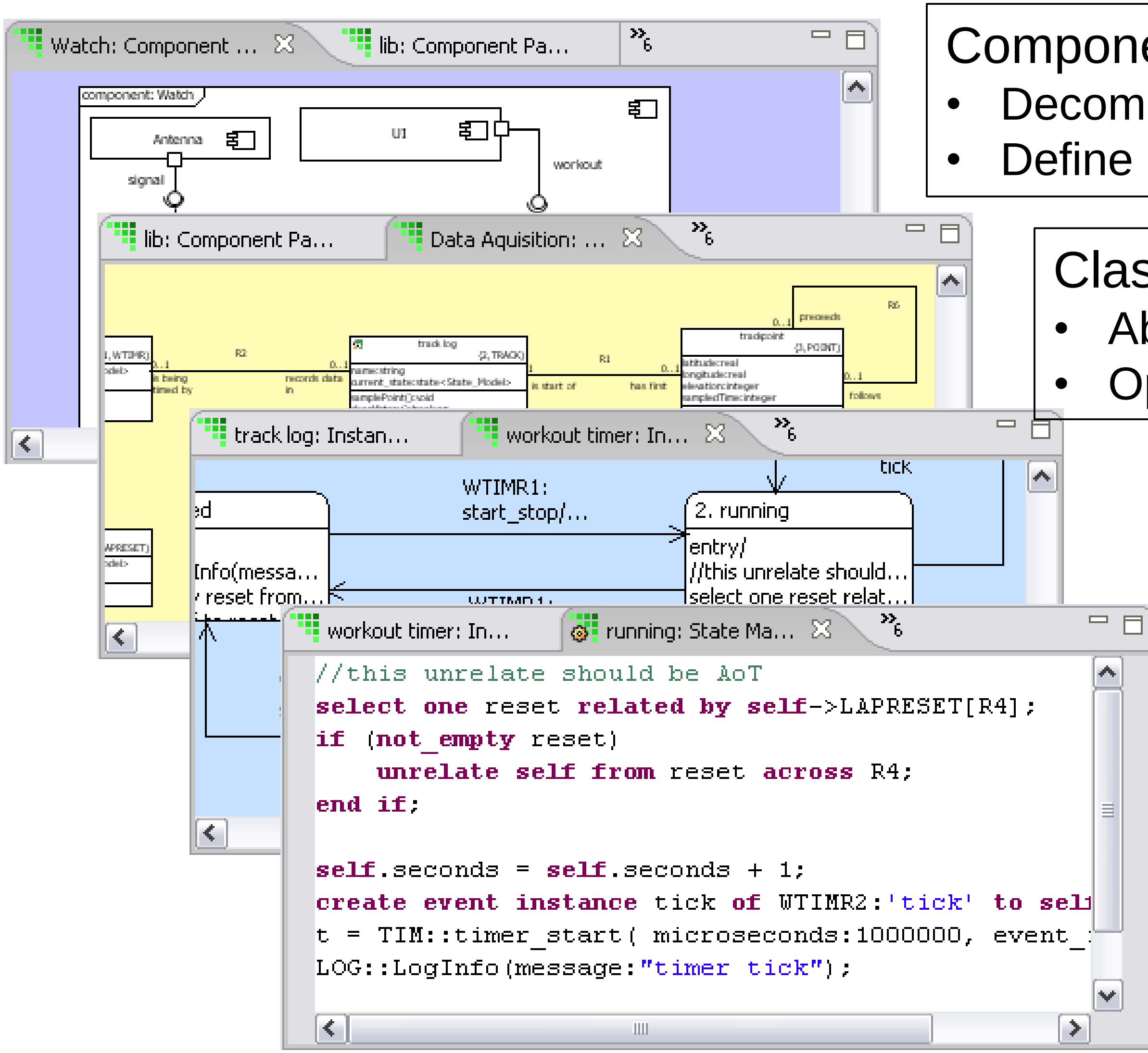
- Natural language and informal diagrams
  - Use cases
  - Activity diagrams
  - Sequence diagrams
- Structural models
  - Components & Interfaces
  - Class models
  - Data types
- Behavioral models
  - State models
  - Activities

This course

Executable Model Hierarchy

High level

Low level



Component Diagram
- Decompose the application
- Define Interfaces

Class Diagram
- Abstractions
- Operations

State Diagram
- Lifecycle
- Event handling

Activities
- Processing

Table of Contents

Component Diagram
- Decompose the application
- Define Interfaces

Class Diagram
- Abstractions
- Operations

State Diagram
- Lifecycle
- Event handling

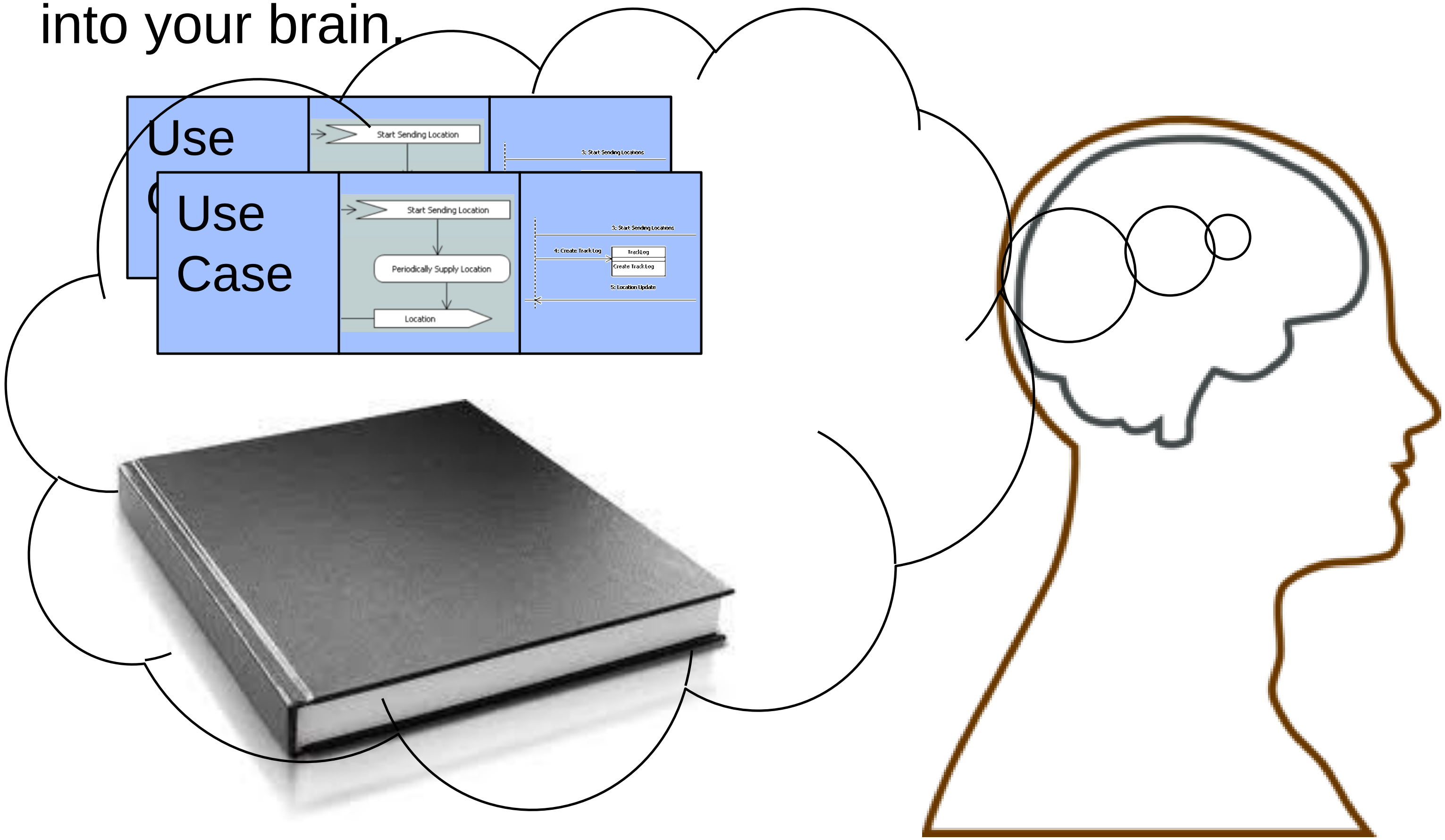
Activities
- Processing

1. Requirements Clarification

1

Building Executable Models

To begin to build executable models, you must first load the requirements clarification models and functional specification into your brain.

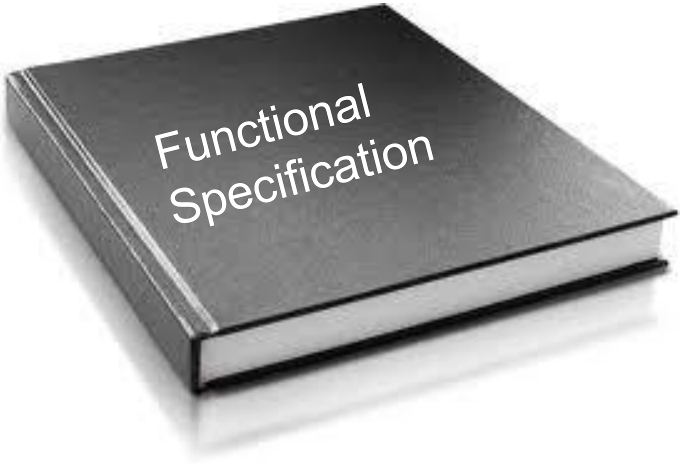


Read the Functional Specification

The functional specification contains a list of functional requirement describing what the system must do.

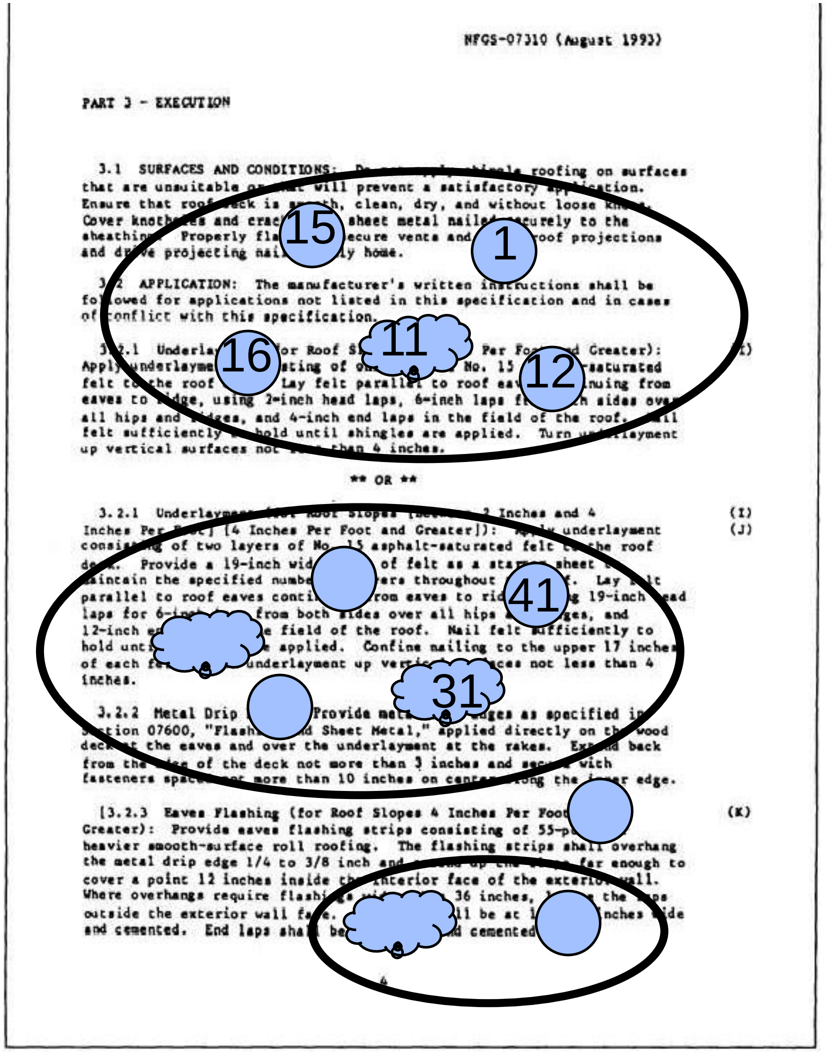
Read it.

It is expressed in natural language.



Relationship to Use Cases

Use cases, our basic unit of organization, cover multiple functional requirements.



NFGS-07310 (August 1993)

PART 3 - EXECUTION

3.1 SURFACES AND CONDITIONS: [illegible] roofing on surfaces that are unsuitable [illegible] will prevent a satisfactory [illegible]. Ensure that [illegible] is [illegible], clean, dry, and without loose [illegible]. Cover knot[illegible] and crac[illegible] sheet metal nailed [illegible] to the sheathing. Properly fla[illegible] secure vents and [illegible] roof projections and dr[illegible] projecting nai[illegible] home.

3.2 APPLICATION: The manufacturer's written instructions shall be followed for applications not listed in this specification and in cases of conflict with this specification.

3.2.1 Underla[illegible] for Roof S[illegible] Per Foo[illegible] Greater): (I) Apply underlayme[illegible] of [illegible] No. 15 [illegible]-saturated felt to the roof [illegible] Lay felt parallel to roof ea[illegible]nuing from eaves to ridge, using 2-inch head laps, 6-inch laps f[illegible] sides over all hips and [illegible], and 4-inch end laps in the field of the roof. [illegible] felt sufficiently [illegible] hold until shingles are applied. Turn u[illegible] up vertical surfaces not [illegible] than 4 inches.

** OR **

3.2.1 Underlay[illegible] Roof Slopes [illegible] 2 Inches and 4 (I) Inches Per [illegible] [4 Inches Per Foot and Greater]): [illegible] underlayment (J) consisting of two layers of No. 15 asphalt-saturated felt [illegible] the roof de[illegible]. Provide a 19-inch wid[illegible] of felt as a sta[illegible] sheet [illegible] maintain the specified numbe[illegible] throughout [illegible]. Lay [illegible] parallel to roof eaves conti[illegible] from eaves to rid[illegible] 19-inch head laps for 6-in[illegible] from both sides over all hips [illegible], and 12-inch e[illegible] field of the roof. Nail felt sufficiently to hold unti[illegible] applied. Confine nailing to the upper 17 inches of each f[illegible] underlayment up verti[illegible] not less than 4 inches.

3.2.2 Metal Drip [illegible] Provide met[illegible] as specified in [illegible]tion 07600, "Flash[illegible] Sheet Metal," applied directly on the wood deck [illegible] the eaves and over the underlayment at the rakes. Ex[illegible] back from the [illegible] of the deck not more than 3 inches and [illegible] with fasteners [illegible] more than 10 inches on cente[illegible] the [illegible] edge.

[3.2.3 Eaves Flashing (for Roof Slopes 4 Inches Per Foo[illegible] (K) Greater): Provide eaves flashing strips consisting of 55-p[illegible] heavier smooth-surface roll roofing. The flashing strips shall overhang the metal drip edge 1/4 to 3/8 inch and [illegible] far enough to cover a point 12 inches inside the interior face of the exterior wall. Where overhangs require flashi[illegible] 36 inches, [illegible] the [illegible] outside the exterior wall fa[illegible]. [illegible]ll be at [illegible] inches wide and cemented. End laps shal[illegible] be [illegible] cemented

4

Scope

The scope of the requirements-clarification use cases is *the system*.

The business and environment

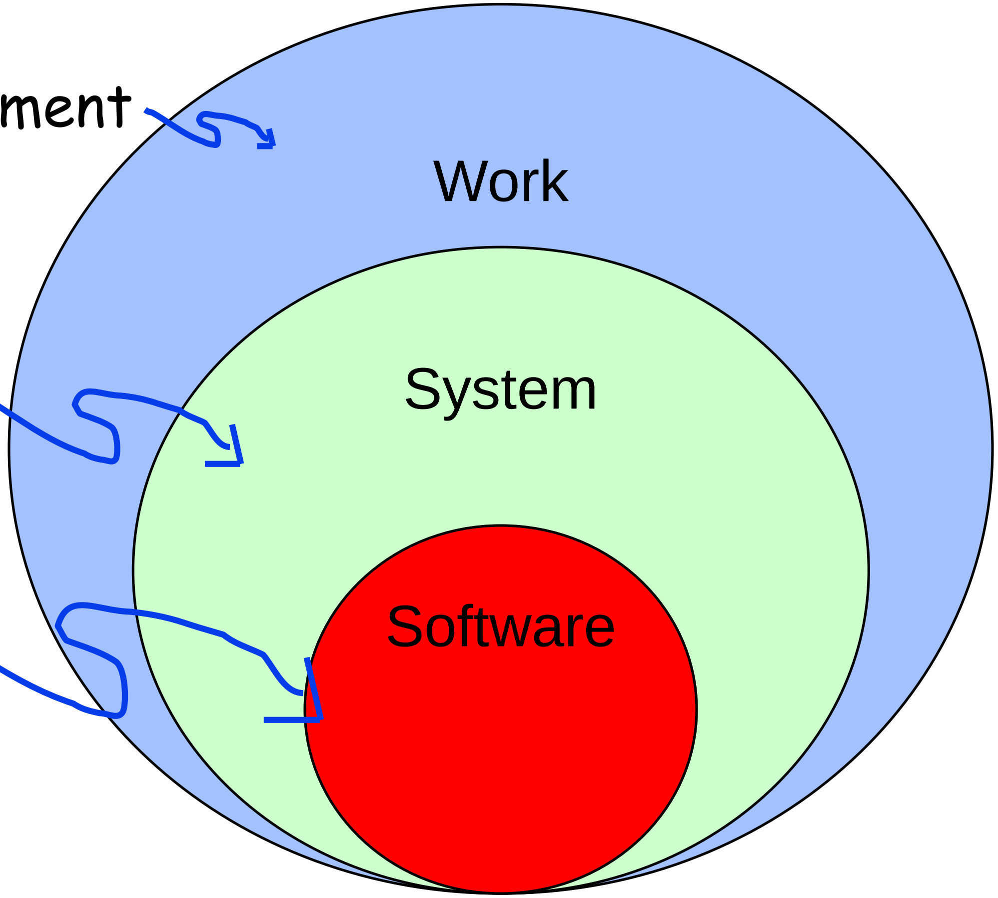
Requirements Gathering

What you are building

✓ Requirements Clarification

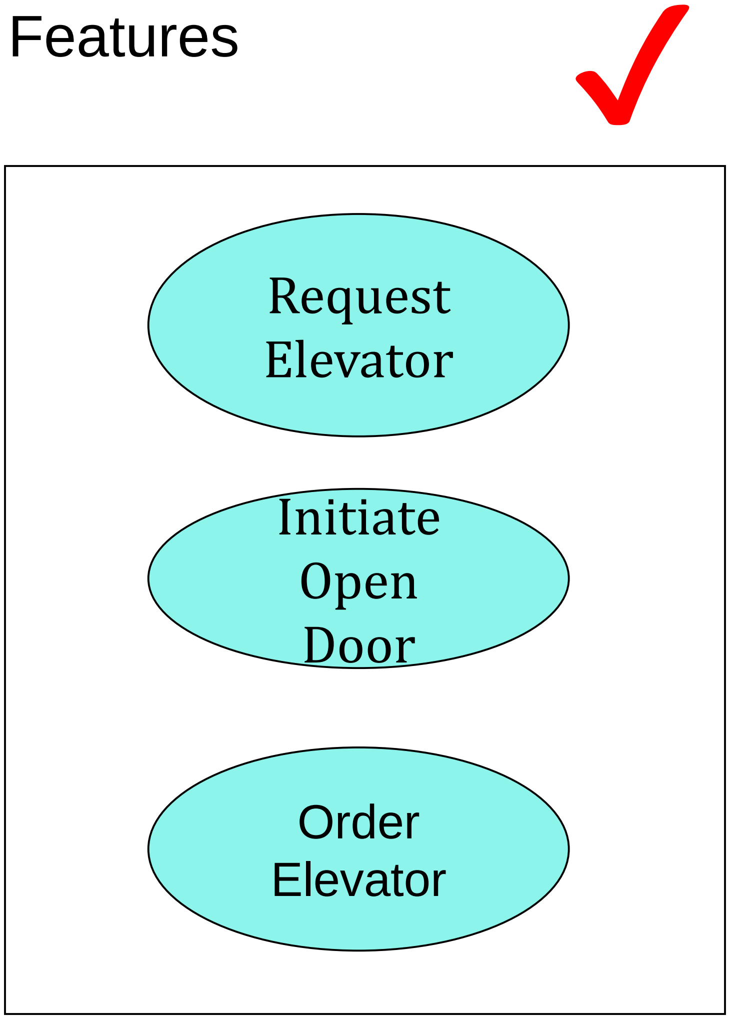
Just the software

Product Construction

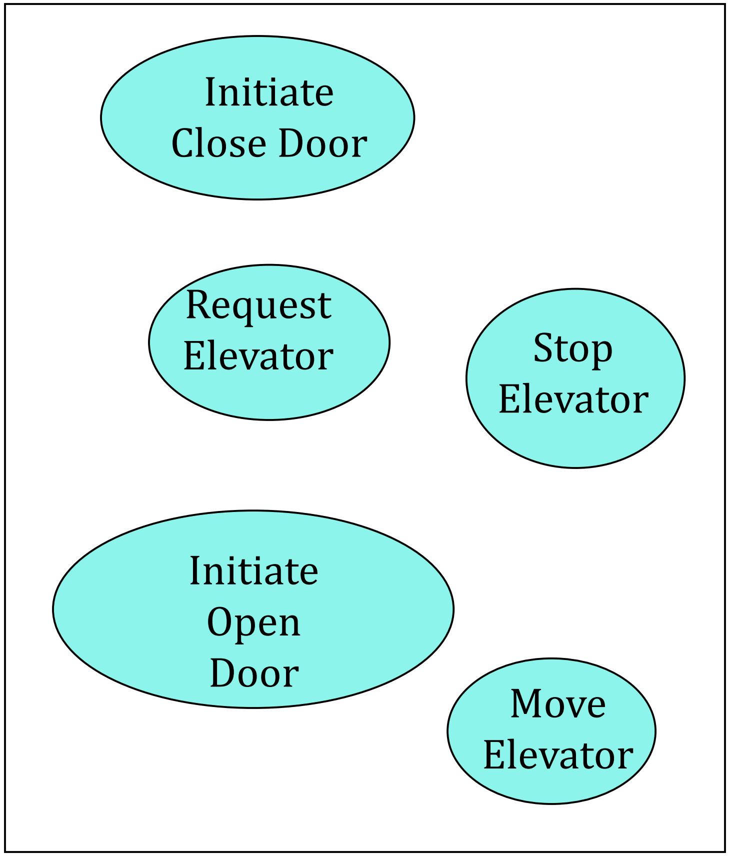


Each Use Case is a Feature

Features



Functions



Use Cases

Each use case shall contain:

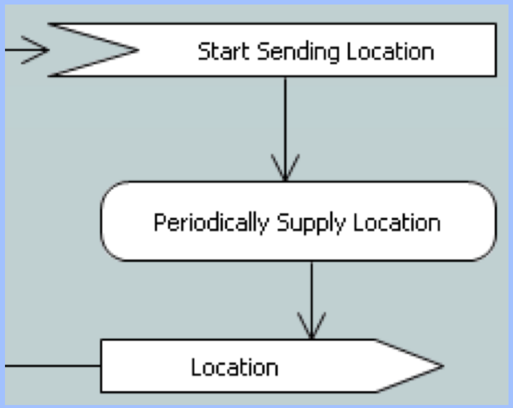
- a description
- an activity diagram, and optionally
- a sequence diagram

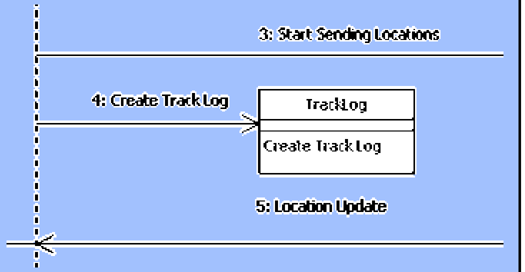
Use Case Name/Number

Pre-conditions:

Post-conditions:

Scenario:





Use Case Definition

Each use case follows this pattern:

<Use Case Number>: <Use Case Name>

Pre-conditions: What must be true before the use case can execute

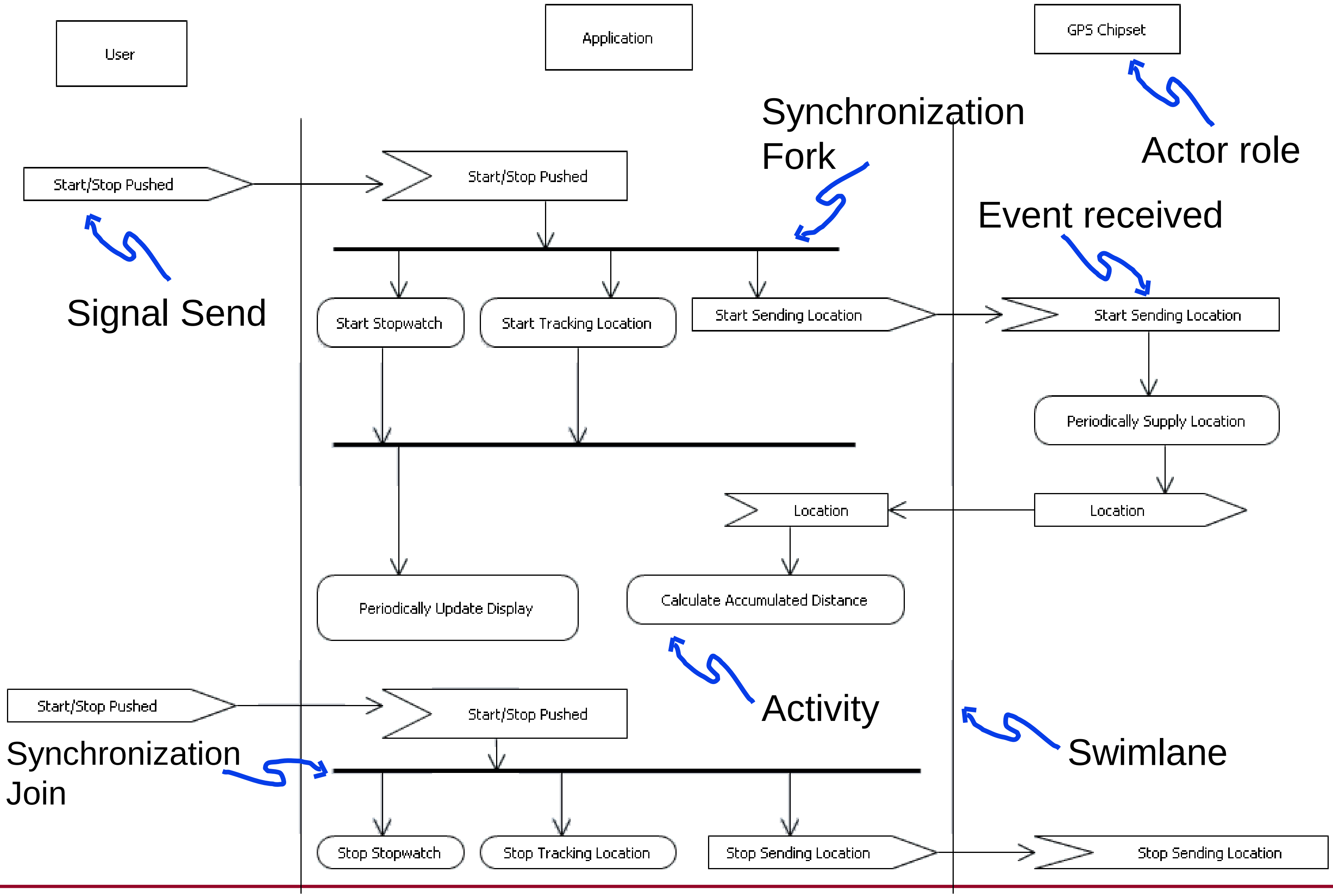
Post-conditions: What must be true after the use case has executed

Scenario: A description of just what happens

These are again in natural language.

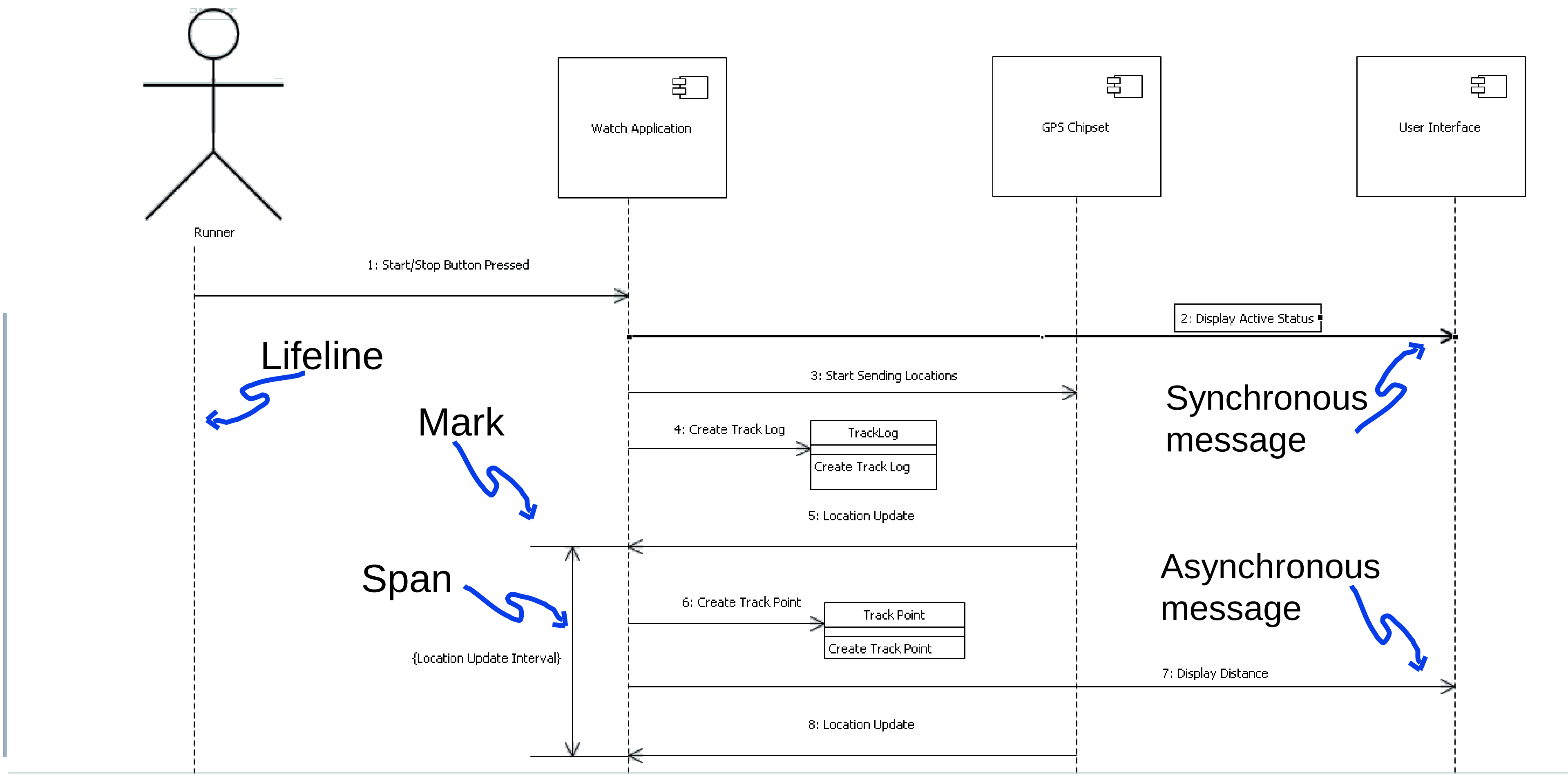
Read them.

Activity Diagram



Sequence Diagrams

Build a sequence diagram if it helps detail your understanding.



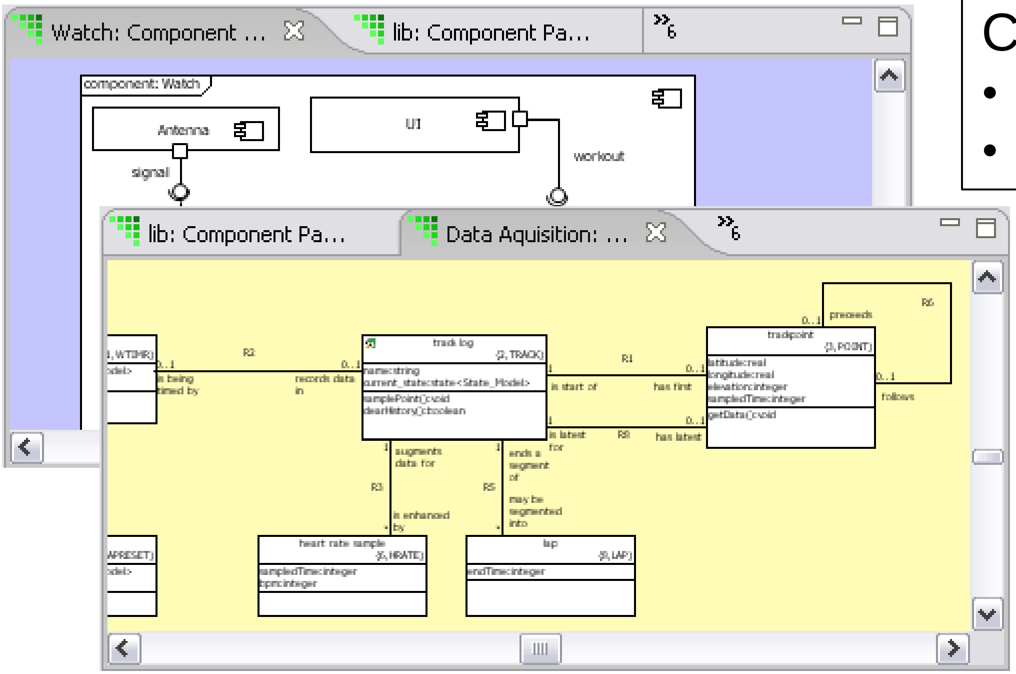
2. Classes

2

Executable Model Hierarchy

High level

Low level



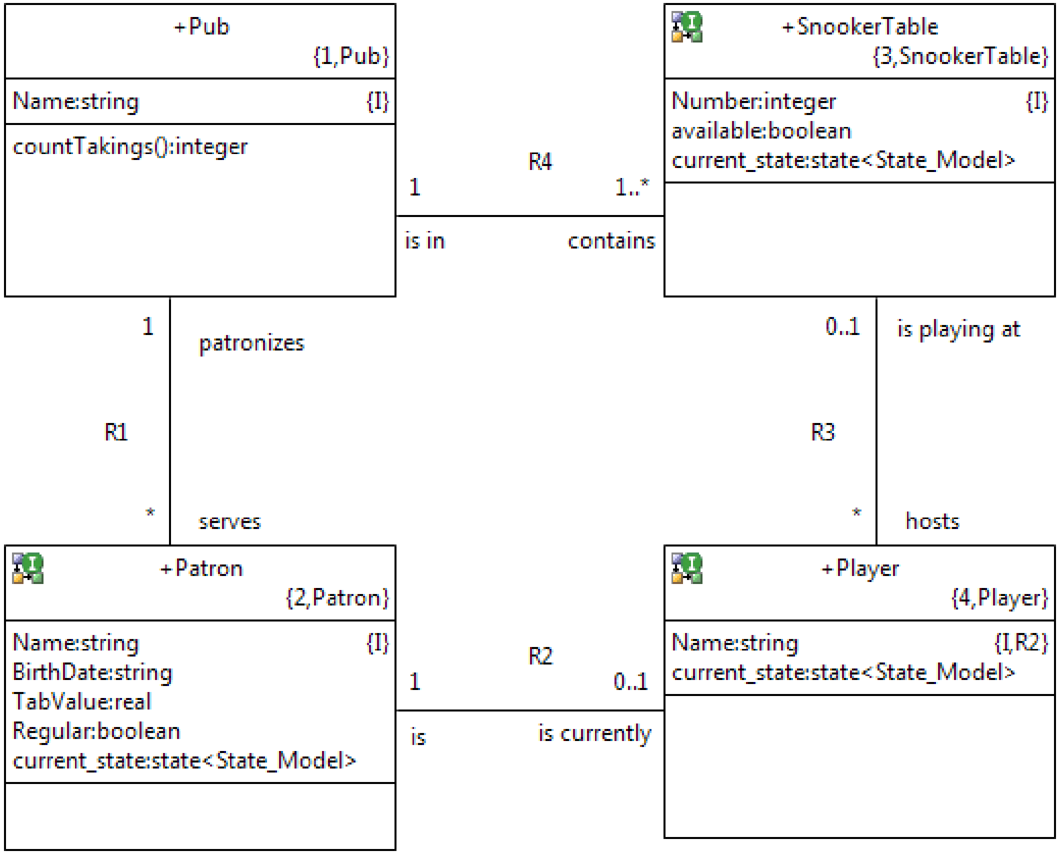
Component Diagram
- Decompose the application
- Define Interfaces

Class Diagram
- Abstractions
- Operations

Class Diagram

A class diagram consists of:

- classes
- attributes
- associations, and
- operations



We shall examine each in turn.

Class

A *class* is a *conceptual entity* within the *subject matter* at hand.

conceptual |kənˈsep ch oōəl|
adjective
of, relating to, or based on mental concepts

entity |ˈentitē|
noun (pl. **-ties)**
a thing with distinct and independent existence

subject matter |ˈsəbjəkt ˈmatər|
topic under consideration

Class

A *class* represents a *set* of instances that all:

- have the same behavior
- are described in the same way

"Set" means that each instance is unique.

"Same behavior" means that each instance behaves in the same way as the other instances.

"Described in the same way" means that any data describing the instance applies uniformly to each one.

Start with the Requirements!

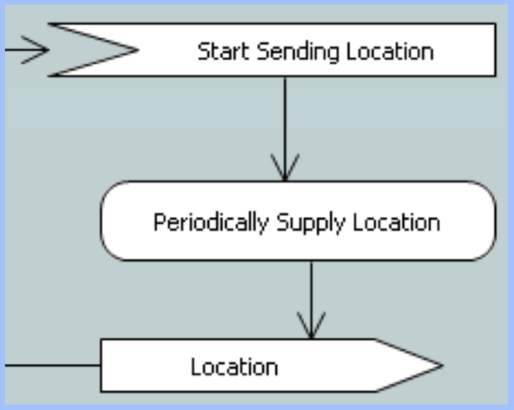
Re-read the requirements, as clarified.

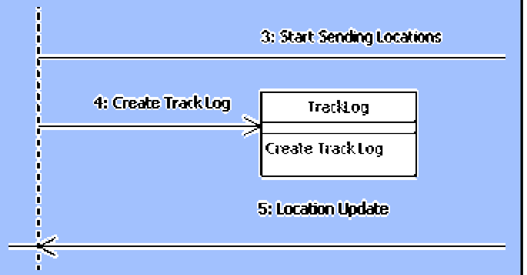
Use Case Name/Number

Pre-conditions:

Post-conditions:

Scenario:





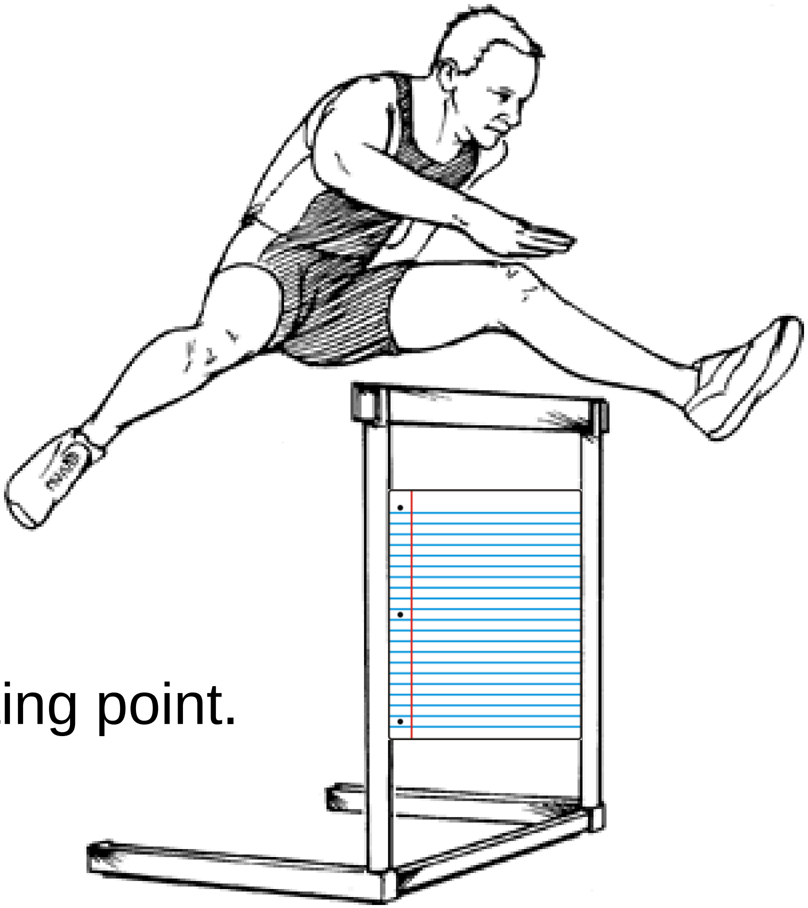
Blitz

A *blitz* is a technique for getting started.

There are no wrong answers.

- We don’t categorize
- We don’t organize
- We don’t evaluate

- We *just enumerate*



The purpose is to provide a starting point.

Class Blitz

Look at all the candidates and categorize them.

- Definitely a class
- Maybe a class
- Definitely not a class

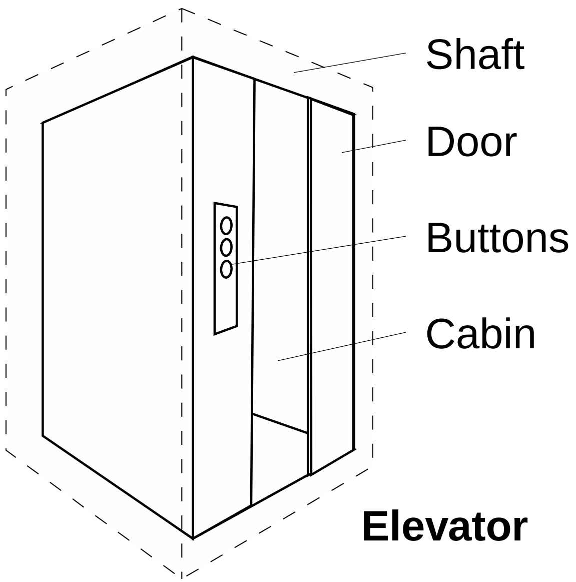
Finding Classes

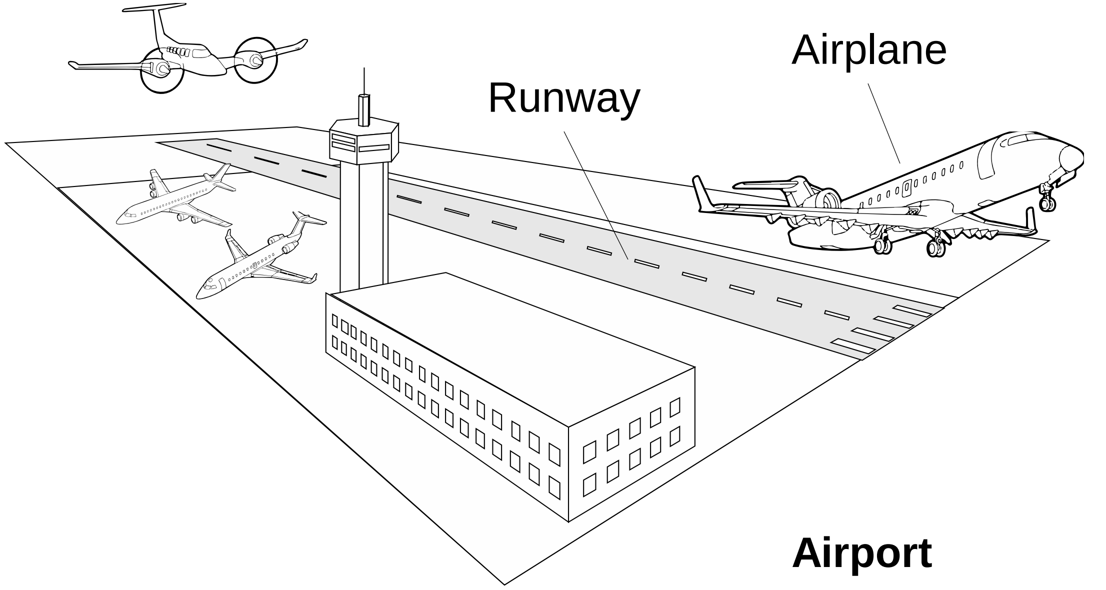
- Tangible things
- Roles
- Incident
- Interaction
- Specification

> Don’t classify classes.
>
> These are just guides for places to look.

Tangible Classes

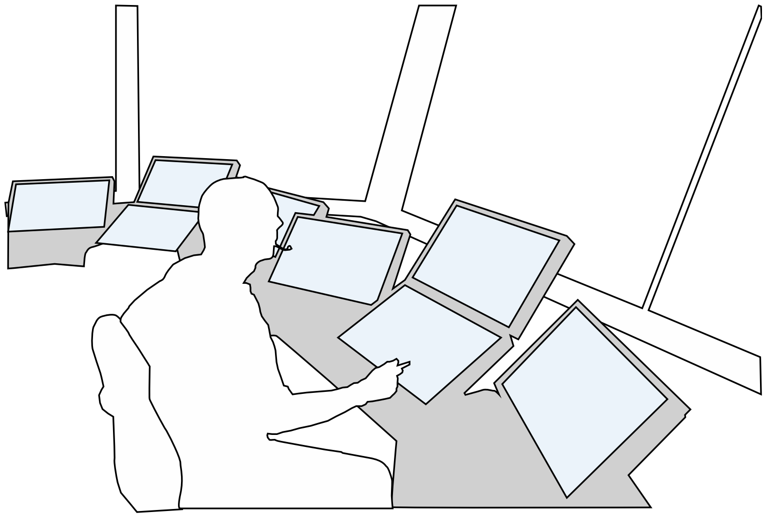
- airplane
- valve
- circuit breaker
- dog
- elevator
- message
- robot
- power supply
- dog owner
- cabin

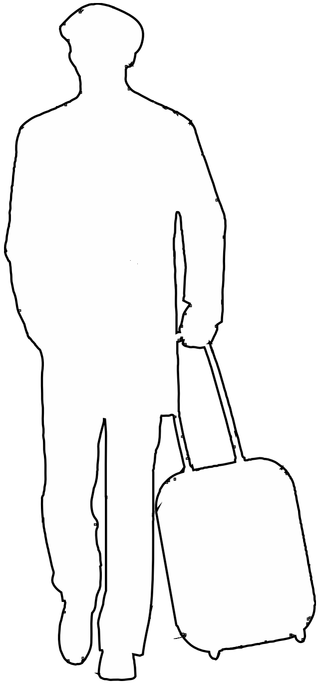




Roles as Classes

- broker
- landlord
- customer
- passenger
- pilot
- client
- tenant
- account holder
- administrator
- air traffic controller

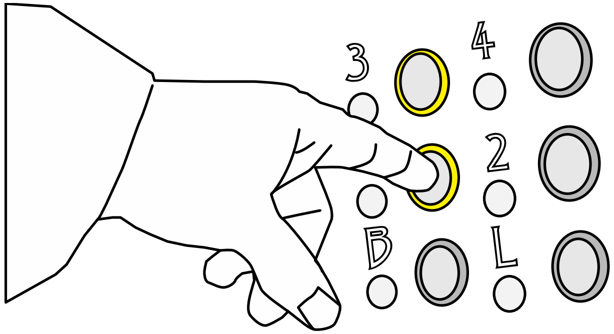


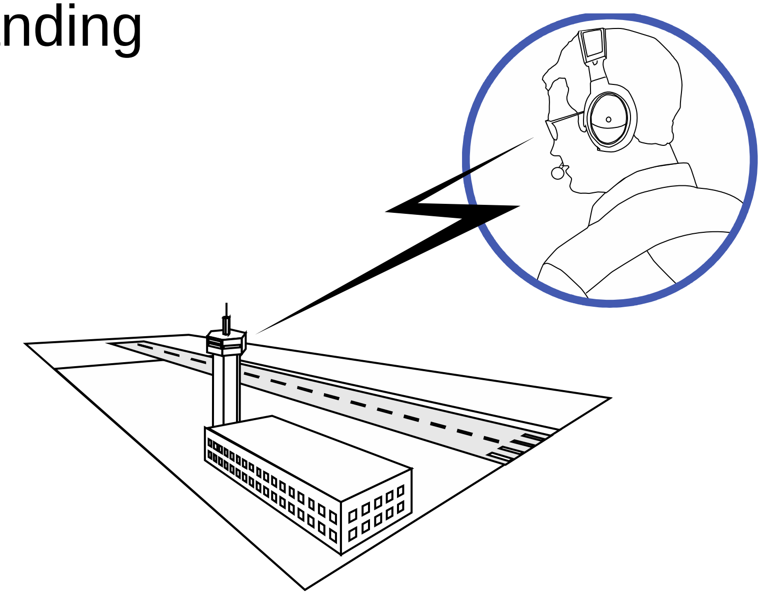


Incident Classes

- performance
- system crash
- breakdown
- request
- flight

- visit
- event
- service call
- order
- landing

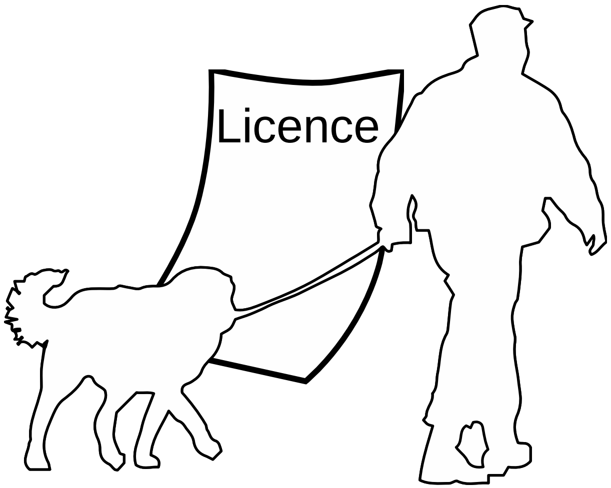


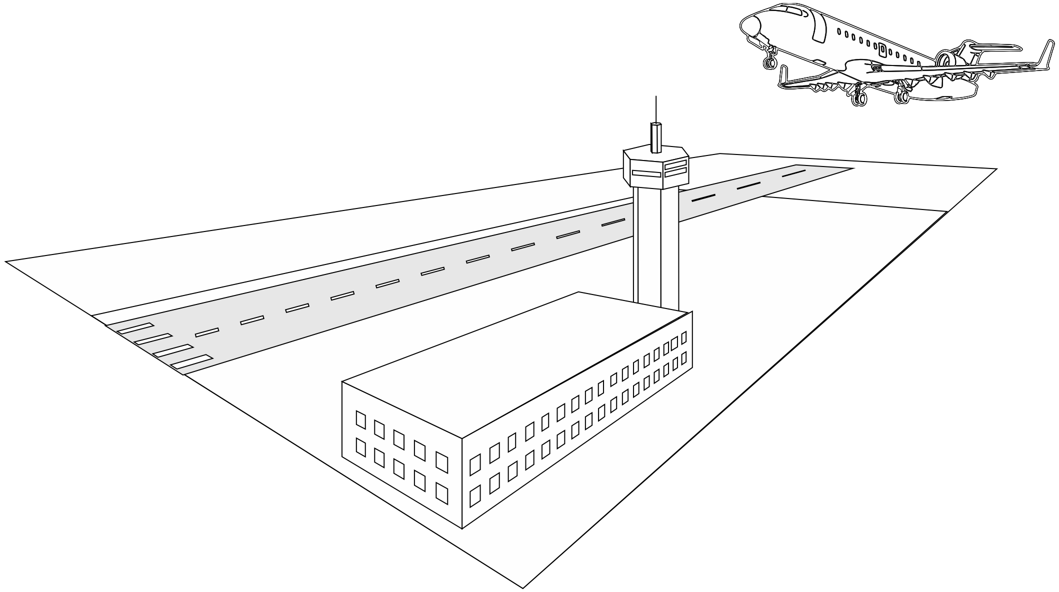


Interaction Classes

- cable
- birth
- purchase
- order
- command

- pipe connection
- link
- marriage
- sale
- landing





Specification Classes

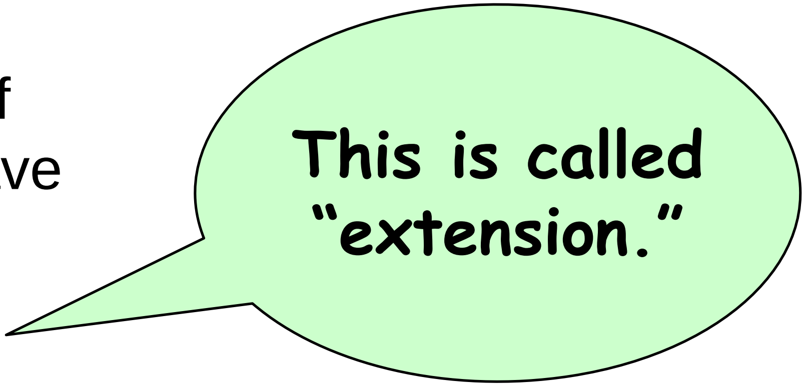
- policy type
- protocol
- phone spec.
- product spec.
- qualification

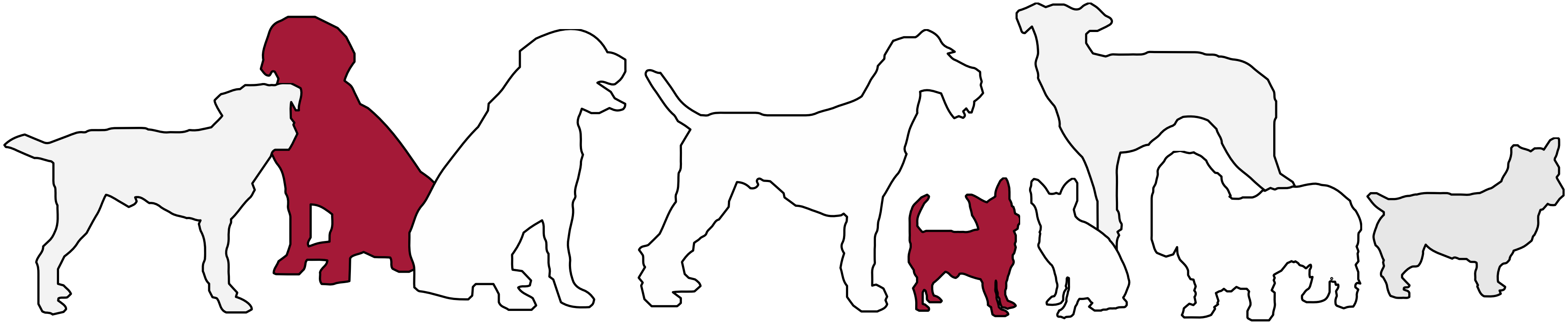
- goal spec'n
- configuration def'n
- account type
- vehicle model
- aircraft type

Finding Classes

We may observe that a number of instances in the subject matter have similar behavior and data.

We abstract from observed instances.





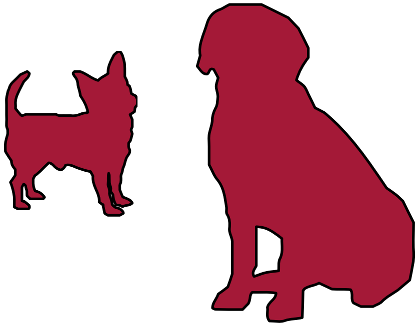
Finding Classes

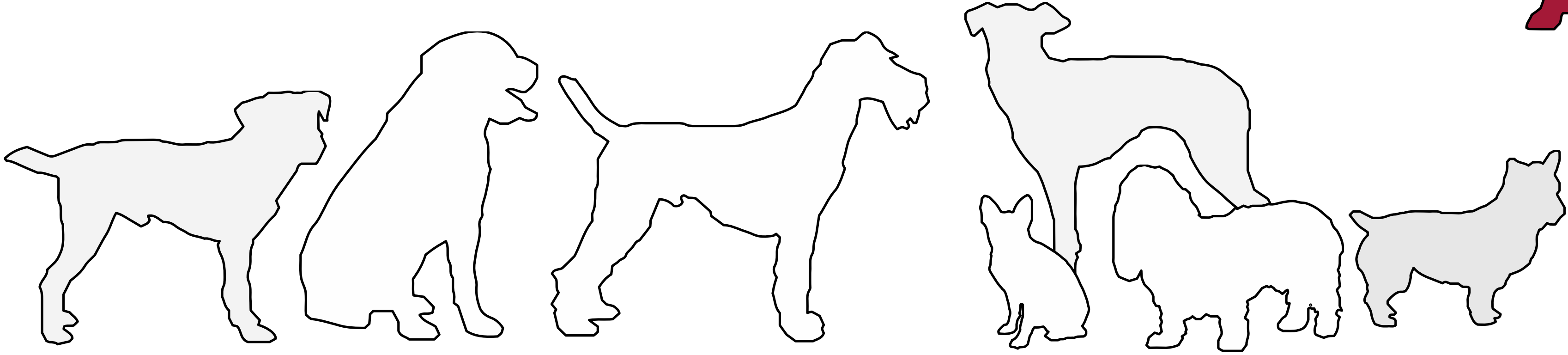
Or we may observe, identify or define a concept with specific qualification criteria.

We abstract based on our ideal.

This is called "intension."

THE FOLLOWING ARE THE QUALIFICATIONS A DOG MUST HAVE FOR ENTRY AT CRUFTS





Workshop

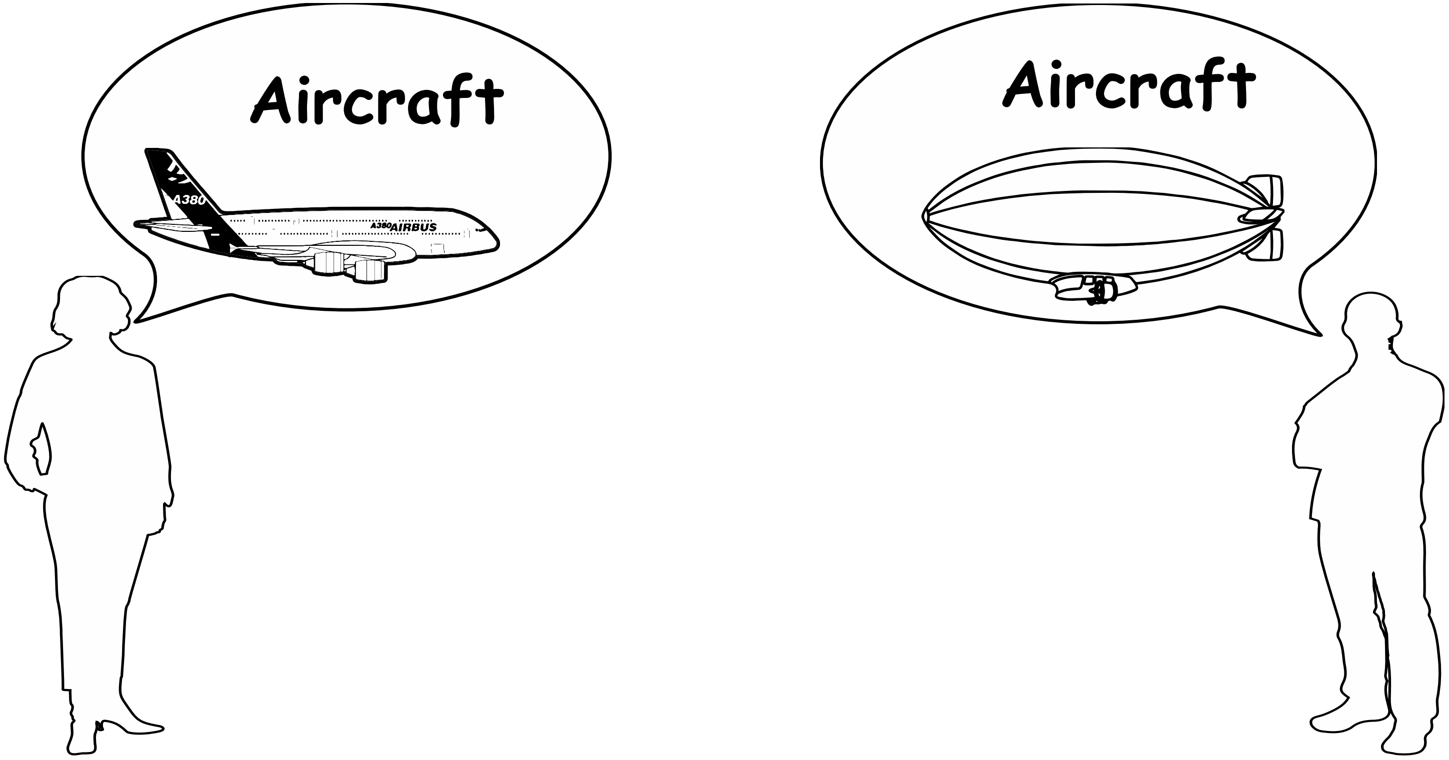
Blitz at least half-a-dozen classes.

Use the clarified requirements for input.

Be prepared to present your list to the class.

Class Definitions

Write a class definition that explains the *basis for abstraction* for each class.



Connect the model abstraction to the subject-matter thing.

Definitions

Write definitions for each thing you find.

> “An Aircraft is anything that flies that must be monitored by the air traffic control system.   The aircraft may carry anything: passengers, freight, nothing. The rules for what constitutes something that must be monitored are described in ….”

> “A Message is a single coherent piece of information sent between two applications. It consists of a header describing the sender and receiver, and a body that can be anything.

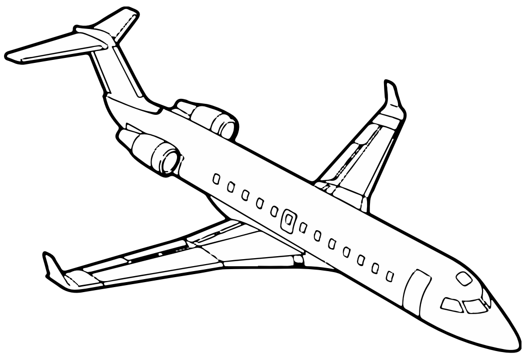
Definitions may incorporate inclusion or exclusion criteria.

Realms

Distinguish the thing in the system-under-study from the abstraction.

Subject Matter Under Study | Model

aircraft



Aircraft

Connect the model abstraction to the subject-matter thing.

Class Definitions

A good class definition:

- connects the subject-matter concept in the system-under-study to the model abstraction
- indicate creations, deletion and lifespan when appropriate (classes and associations)
- Can immediately be understood by non-experts

Workshop

Properly define two class descriptions from the classes you identified earlier.

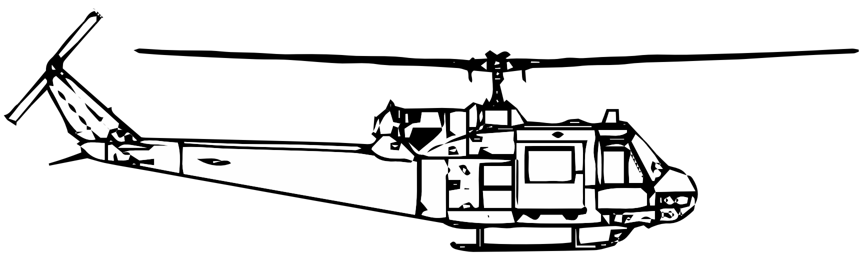
Testing Classes

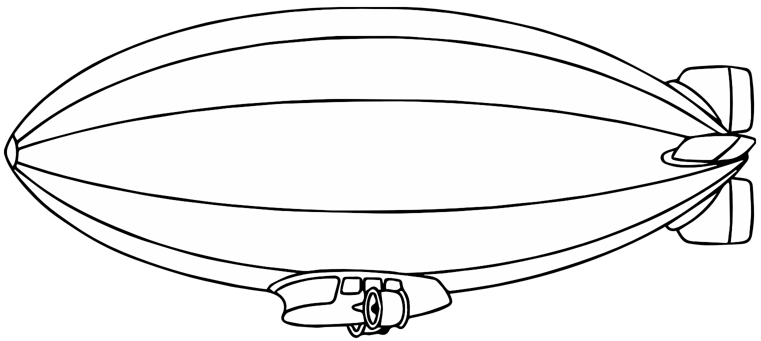
There are several tests you can apply to classes.

- The Uniformity Test
- The OR test
- The More-than-a-list test
- The Table test
- The –er test

The Uniformity Test

If instances of your classes have different data or different behavior, you probably have two classes.





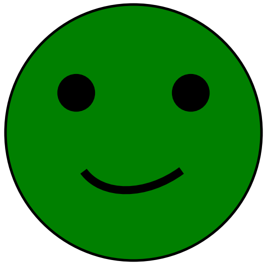
The Or Test

If your class description contains ‘or’ in a disjunctive way, you probably have two classes.

An airplane is an aircraft with a minimum take-off speed, or a helicopter.

An airplane is a passenger or cargo aircraft.

An airplane is run by a commercial airline, such as Laos Airways or LAN Ecuador.



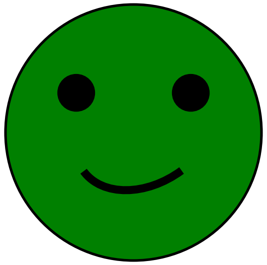
The More-Than-a-List Test

If your class description contains just a list, without any abstraction, you need to search for the basis of abstraction.

A commercial airline is Laos Airways or LAN Ecuador.

A regional airline is an airline that takes short trips, such as Laos Airways or LAN Ecuador.

A regional commercial airline is an airline that takes paying passengers for trips under 500km.



The Table Test

You should be able to fill in a table with candidate instances.

Flight/Passenger				
Number	**Start**	**End**	**Price**	**Paid**
XL1541	UIO	CUE	$44	$44
QF5	SYD	SIN	$814	$82
My Sunday Flight	Gloucester	Gloucester	$1.99/L	$1.99/L
XL516	UIO	MIA	$1095	

The –er Test

Classes should represent real “things” (concepts, rules, specifications, occurrences, incidents) in the physical, hypothetical or abstract world.

They should not be:

- implementation oriented
- vague –er names

Handler

Controller

Manager

These “classes” tend to be functionality wrappers.

Workshop

Test the classes you have defined up to this point.

Be prepared to share your results with the class, including the class candidates you discarded after testing.

3. ATTRIBUTES

3

Attributes

An *attribute* is an abstraction of a single, relevant characteristic that every instance of the class must have.

Each instance of the class may have a different value for the attribute.

"Relevant" depends on the requirements placed on the subject matter.

Aircraft.color may be relevant to fitting out, but not to air traffic control.

Flight				
Booking	**Start**	**End**	**Number**	**Price**
HPFGYI	UIO	MIA	XL516	$1095
JKLOIP	SYD	SIN	QF5	$814
HPFGYI	BKK	LPQ	QV633	$631
GHJKLP	EZE	IGR	AR1724	$244

Roles of Attributes

Attributes may take on one or more roles. They may be:

- descriptive: describes an instance of a class
  - eg latitude
- naming: names an instance of a class
  - eg, body number
- referential: refers to an instance of another class
  - eg myOwner

It's possible for an attribute to be all three (e.g. a role whose identifying attribute is a descriptive name, such as "Tiny.")

Descriptive Attributes

A *descriptive attribute* provides some information about an instance.

The attribute must be able to have a value at some point in the instance's lifecycle.

If the attribute value for an instance is "not applicable" you need to factor your class.

If it doesn't have a value *yet*, that's OK.

Naming Attributes

A *naming attribute* provides a label for an instance.

- License Number
- WayPoint Name
- Ticket Number

Highlighting these attributes helps understanding of the subject matter being modeled.

At implementation time, a handle performs the same function.

The label may also be descriptive.

- eg Control Station.North Station

(i.e. it is the Control Station that happens to be at the north end)

The label may be arbitrary.

- eg Employee.EmployeeNumber

(i.e. it's made up, possibly according to some policy)

Identifiers

An *identifier* is one or more attributes that, taken together uniquely identify an instance of a class.

It may comprise one or more attributes.

Each attribute that makes up an identifier is an "identifying attribute."

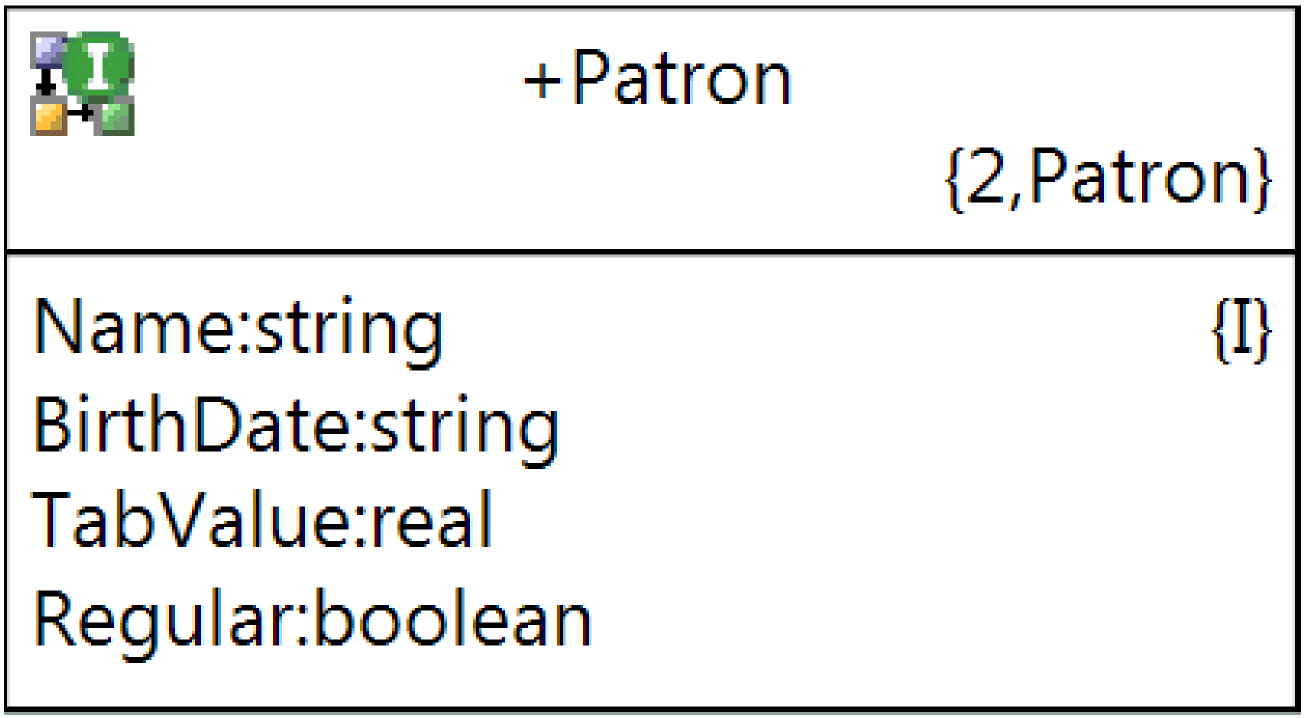
Referential Attributes

A *referential attribute* is an attribute that refers to an identifying attribute of another class.

A referential attribute "formalizes an association."

Finding Descriptive Attributes

The terms you defined during requirements clarification are good candidates.



Finding Naming Attributes

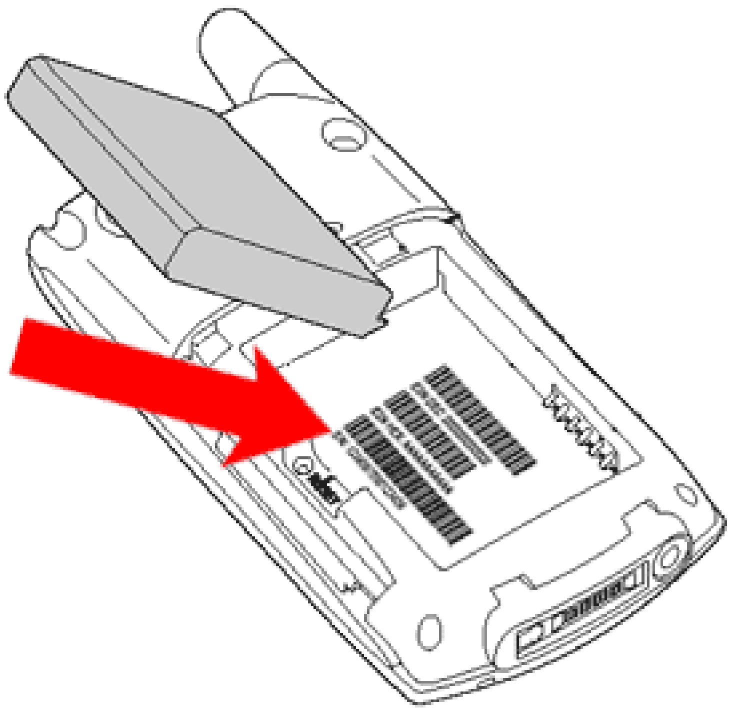
Don’t just slap down “ID”!

Ask if there’s anything that properly represents the abstraction.

What identifies a phone?

Finding Identifiers

Pick the identifier that *captures the abstraction* you intend.



Finding Identifiers

An identifier may comprise more than one attribute.

A “compound identifier” is an identifier comprising many (identifying) attributes.

Data Types

Every attribute has a *domain-specific data type*.

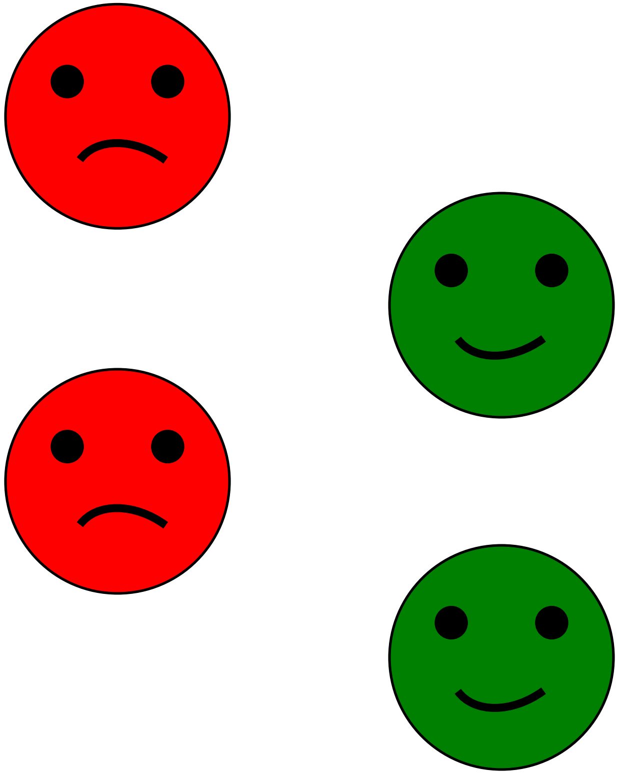
It has a value in the context of the subject matter.

Aircraft.Altitude: real

Aircraft.Altitude: height

Body.Length: char

Body.Length: number of bytes



Data Types

A type may have:

- units (e.g. , meters, feet, nautical miles)
- range (e.g. 10..260, natural, negative integer)
- initial value (e.g. temperature: 0)

System data types include:

- date
- time
- unique id

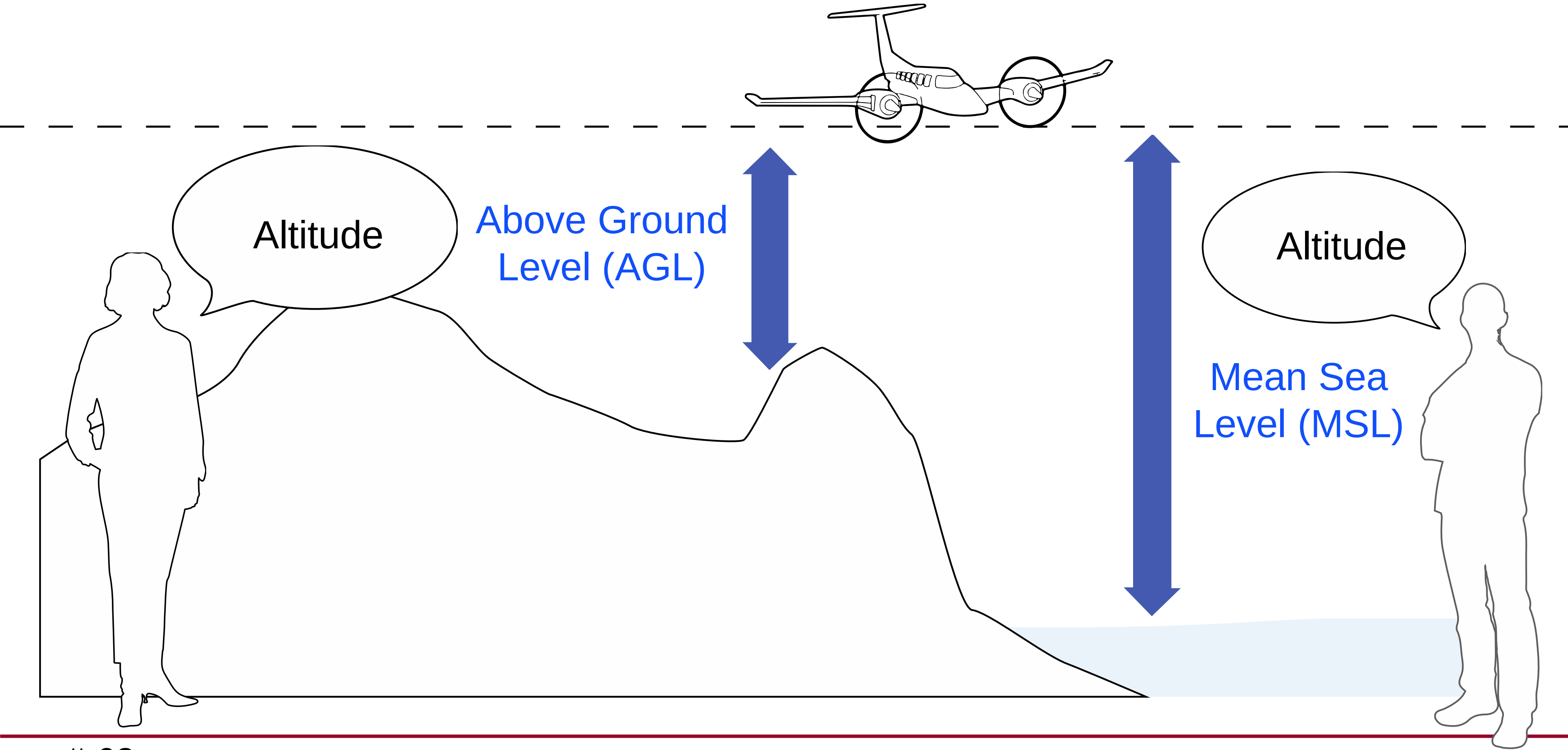
Workshop

For the classes for which you previously wrote descriptions, list the attributes, including the type of each.

Be prepared to share your attributes with the class, including any you eventually discarded.

Attribute Definitions

Write an attribute definition that explains the basis for abstraction for the attribute.



Examples

The speed of the aircraft is how fast it's going.

"The speed of the aircraft is relative to the air through which it travels, measured in knots, between zero and 700."

units

range

Examples

Lovely. It also excludes artichokes.

The length of the message excludes the header.

units

"The length of a message is measured in bytes. It may be between 0 to 256K-1. It is initially zero.

initial value

range

Attribute Definition Guidelines

- Connect subject matter concept or quantity to abstraction
- Readily understood by subject-matter experts
- For quantitative attributes:
  - Units (meters, yards, degrees Centigrade, milliparsecs)
  - Origin (above ground level, mean sea level)
- Initial value (false, 0 degrees Centigrade)

Testing Attributes

There are several tests you can apply to attributes.

- The Applies-to-All-Instances Test
- The Valid-Value Test
- The Multiple-Value Test
- The Compound-Value Test

They are all based on an attribute having a single potential value that has meaning in the subject matter.

Applies-to-All-Instances Test

Check that each attribute applies to all instances.

Aircraft Specification				
Model	**Weight**	**Range**	**Runway**	**Speed**
A380	276.8	15.7	Group V	900
B747	178.8	13.45	Group IV	830
Sikorsky H19	0.4795	0.652	N/A	163
Dash 8	14.7	1.889	Group II	500

Valid-Value Test

Check that each attribute has a valid value *at some point in its lifecycle.*

Flight				
Number	**Start**	**End**	**Price**	**Paid**
XL1541	UIO	CUE	$44	$44
QF5	SYD	SIN	$814	$82
My 10th	GLO	GLO	N/A	£150
XL516	UIO	MIA	$1095	

Multiple-Value Test

Check that each attribute has a single value.

Flight				
Booking	**Start**	**End**	**Number**	**Price**
HPFGYI	UIO	MIA	XL516	$1095
JKLOIP	SYD	SIN	QF5	$814
HPFGYI	BKK	LPQ	QV634/633	$631
GHJKLP	EZE	IGR	AR1724	$244

Compound-Value Test

Check that each attribute is treated as a single unit.

Aircraft.(latitude, longitude)

Operation “Proceed along the latitude line”

Operation “Move from (57ºN, 1ºE) to (57ºN, 10ºE)”

You (in your subject matter) cannot break it apart. Someone else might though.

Workshop

Write attribute descriptions for the attributes you identified earlier.

Apply the tests.

Be prepared to share your descriptions with the class. Highlight attributes that were discarded or changed.

Workshop

Compare your classes, attributes, and descriptions to those in the provided solution.

List issues that require discussion.

4. Associations

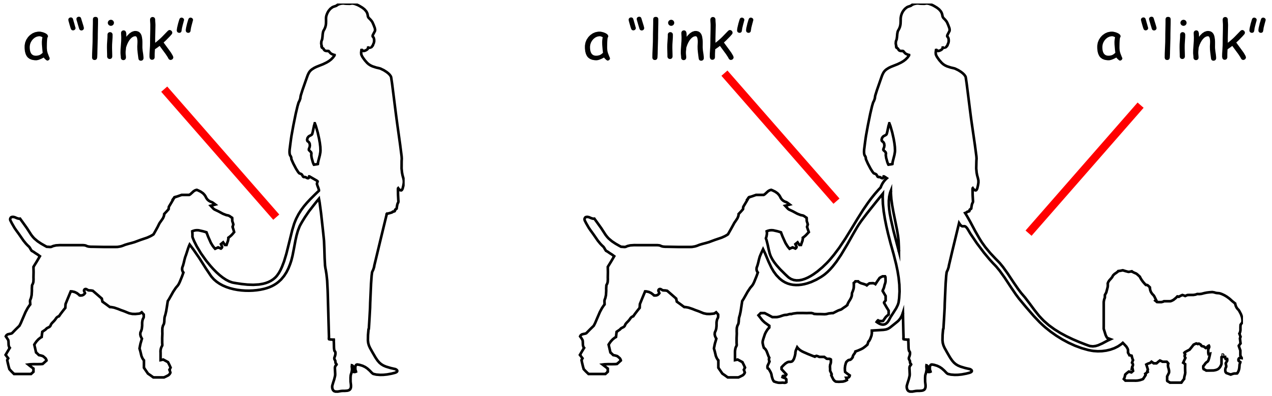
4

Binary Associations

A *binary association* is an abstraction of a relationship between two things that were abstracted as classes.

Each 'end' of the binary association has a:

- name that captures the meaning of the association
- multiplicity that captures the number of instances that participate



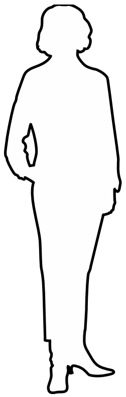
A "link" is an instance of an association.

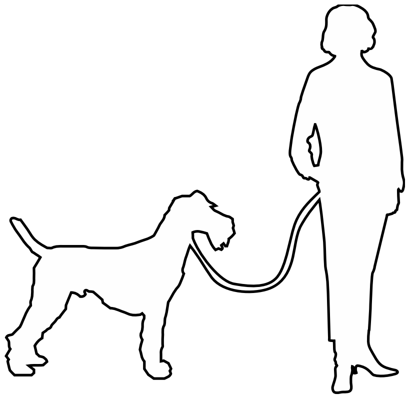
Binary Associations

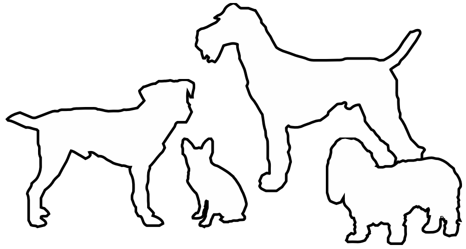
A *binary association* is an abstraction of a relationship between two things that were abstracted as classes.

Each ‘end’ of the binary association has a:

- name that captures the meaning of the association
- conditionality that captures whether the instances must participate in the association







Names

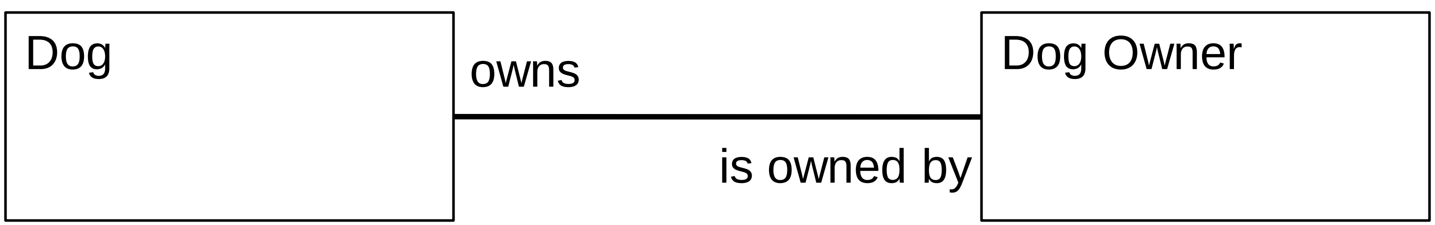
The name captures the role the “target” class plays with respect to the other end.

Many books use roles instead of verb phrases.

Ignore them.

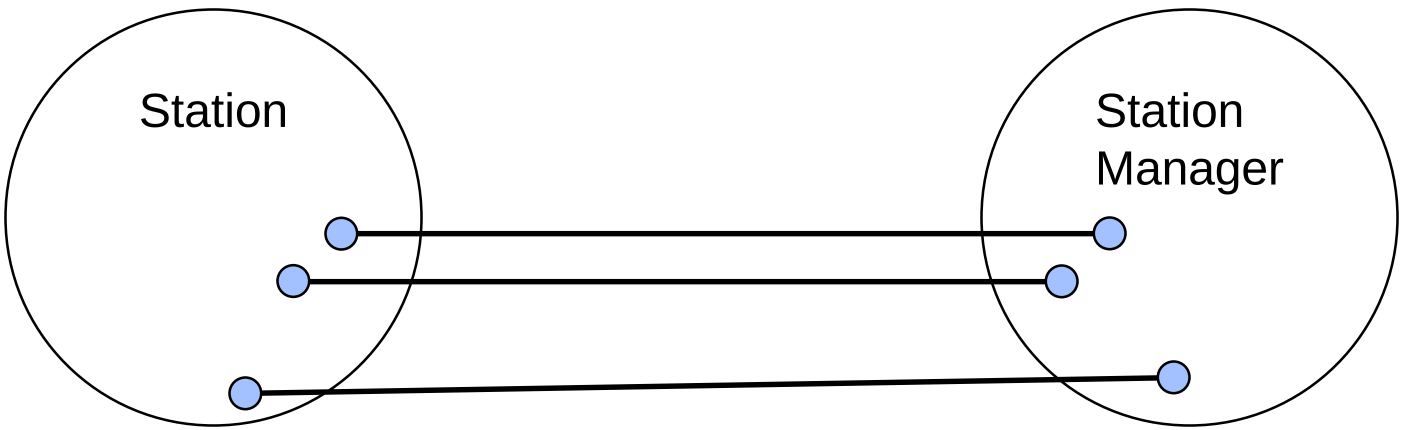
Roles won’t tell you what you need to know.

These are written:



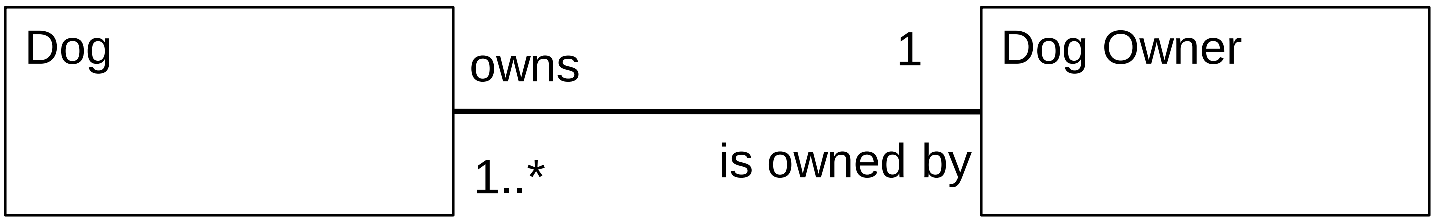
Multiplicity

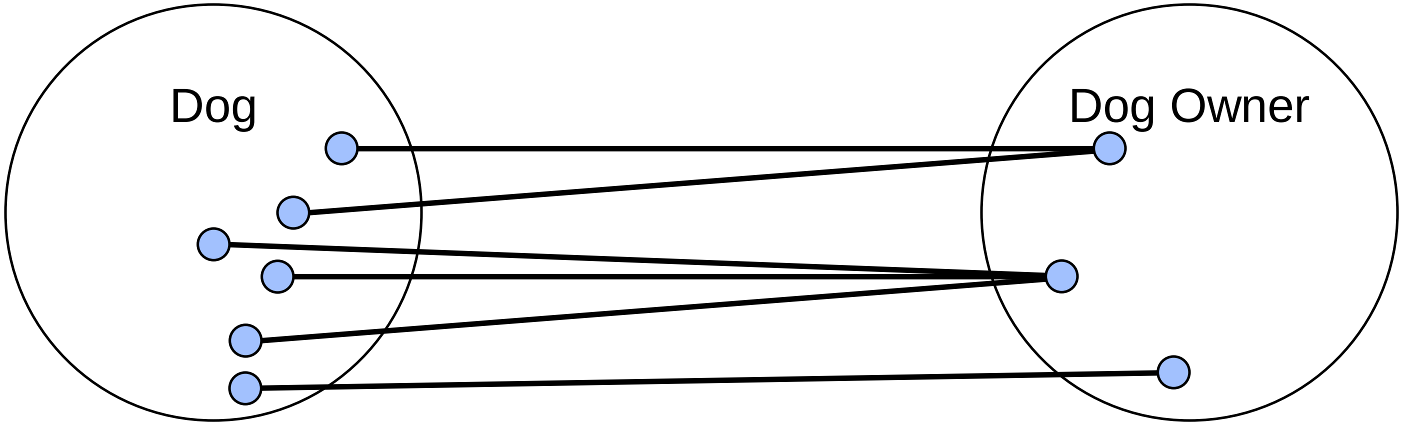
The *multiplicity* captures the number of instances that participate in the association.



Multiplicity

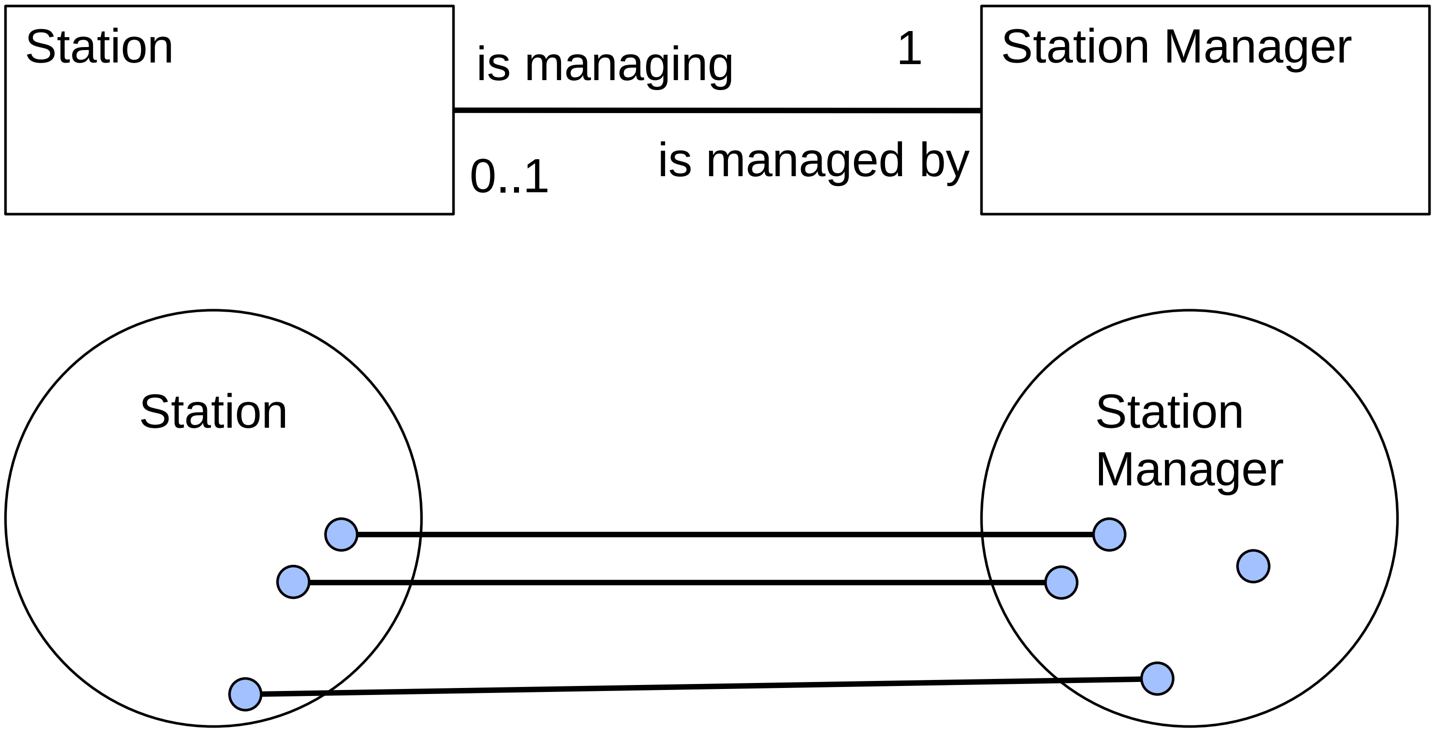
The *multiplicity* captures the number of instances that participate in the association.





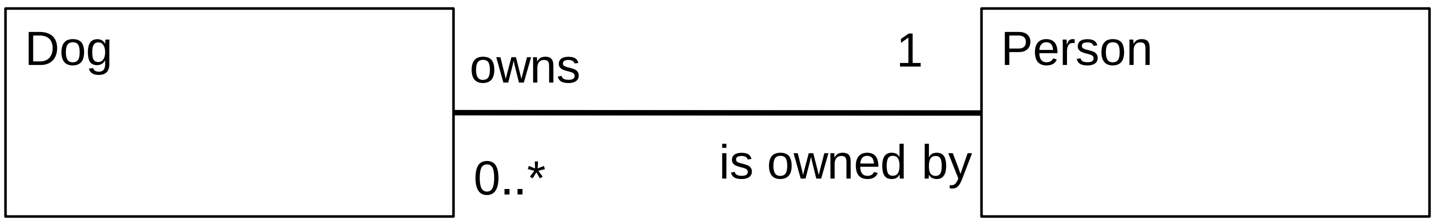
Conditionality

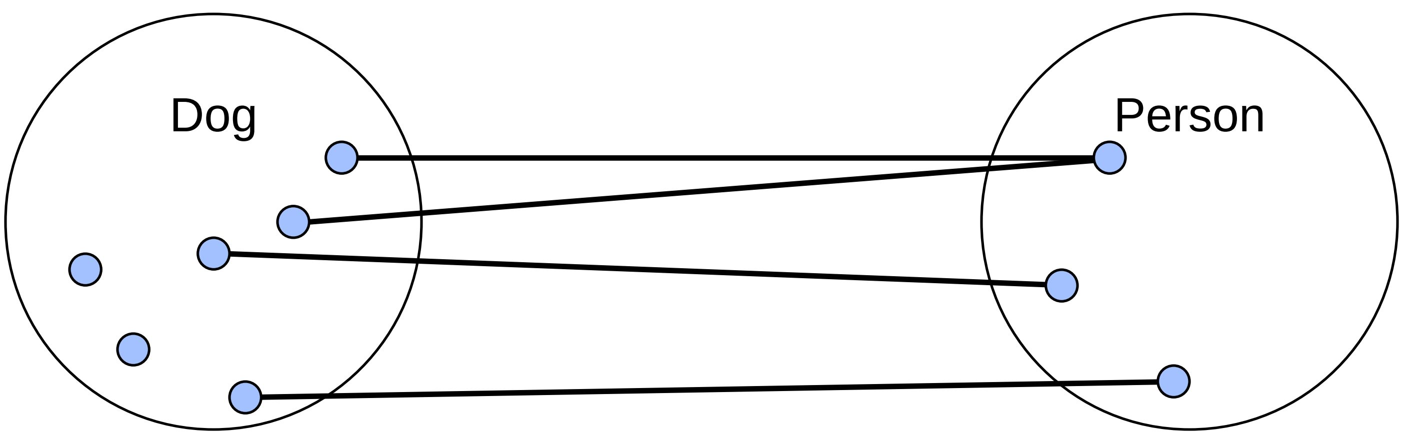
The *conditionality* captures whether an instance is required to participate in the association.



Conditionality

The *conditionality* captures whether an instance is required to participate in the association.

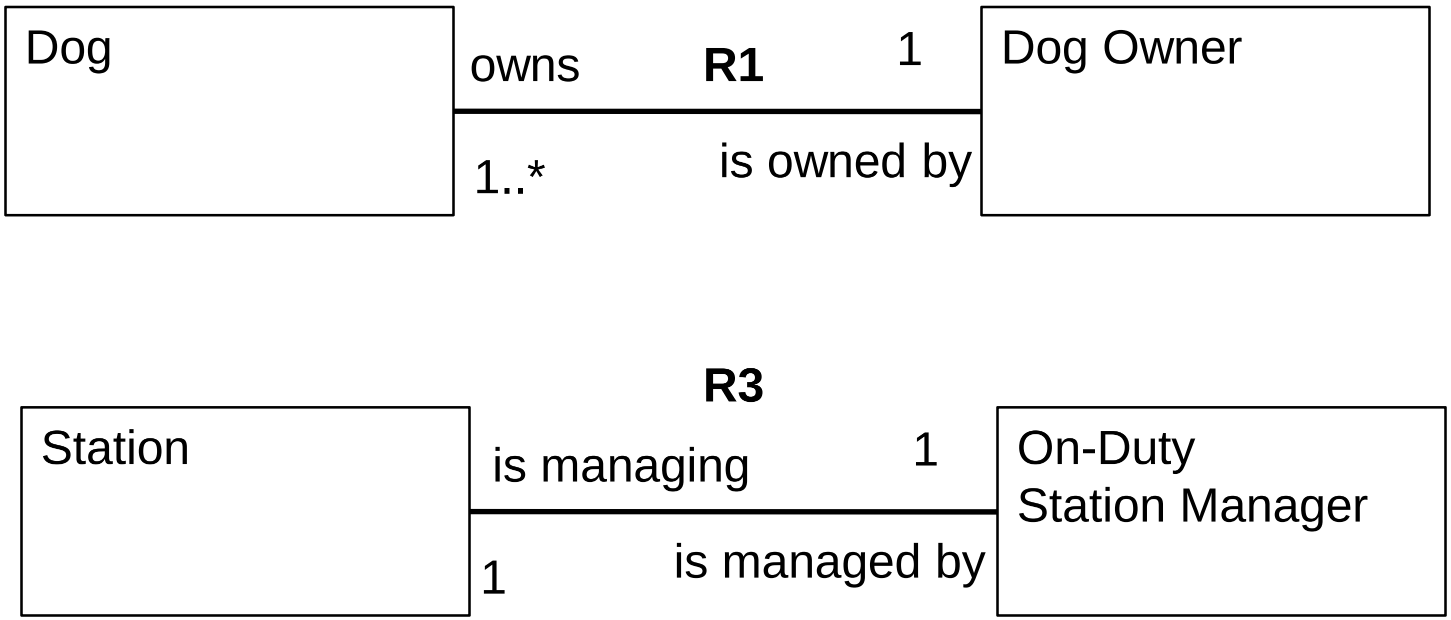




Association Identifiers

Names at the ends of associations may not be unique.

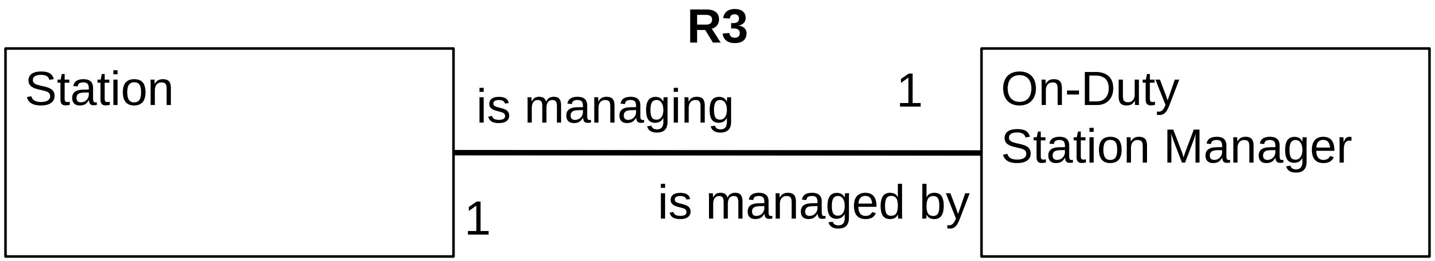
Therefore each association has a unique identifier.



Finding Associations

Capture the *meaning* of the association.

Be certain to name both ‘ends’ and check their multiplicity and conditionality.



Read this as:

- (One) Station is managed by one On-Duty Station Manager
- An On-Duty Station Manager is managing one Station

Association Descriptions

Every association must have a description that:

- connects the abstraction to the subject matter
- provides details about the semantics of the association or how it is used
- says when it is established and removed (*time scope*)

> Don't repeat what is on the diagram.
>
> One fact in one place!

R1: The association is created when the dog is acquired and deleted when the dog is given or sold to a new owner or when the dog or the owner cease to exist.

Multiplicity Test

Check that the class is defined in such a way that it justifies the multiplicity.

Can a Dog have multiple owners?

- At one time?
- Over time?

Dog

Do we need to know:

- What dogs were owned in the past?
- Is that a separate association?

Dog Owner

Your decisions must be based on the requirements!

These decisions define the *time scope* of the model.

Conditionality Test

Check that the class is defined so it justifies the conditionality.

When is a dog a Dog?

- At birth?
- When ‘owned’?
- When a license is issued?

Dog

When is a dog owner a Dog Owner?

- When he has a dog?
- When he has a license?
- Once a dog owner, always a dog owner?

Dog Owner

Your decisions must be based on the requirements!

> These decisions define the *time scope* of the model.

Time Scope

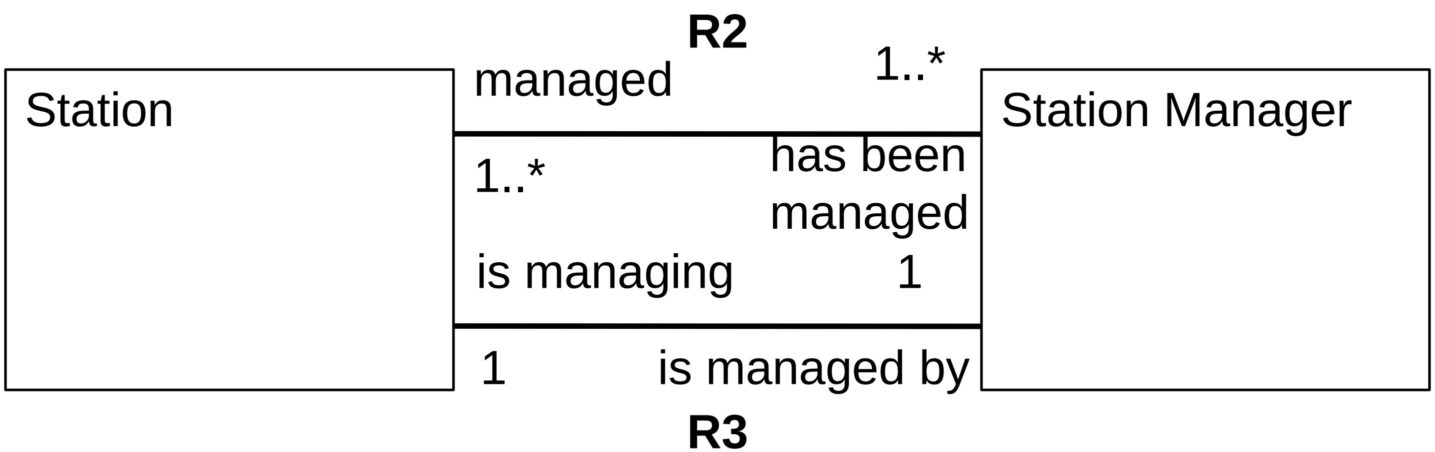
The model captures the instance population at any given instant in time.

Be sure:

- the conditionality and
- multiplicity

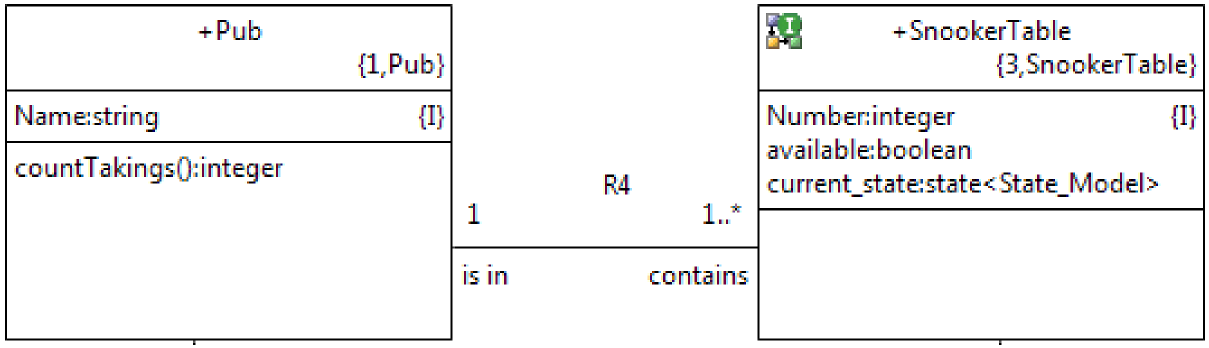
reflect that fact.

What does "instant in time" mean? Exactly?



Preexisting Instances

Some instances exist before the system starts running.



Pub
Name
The Kings Arms
The Queens Head

Table		
Number	**Available**	**State**
23456	Yes	...
12345	No	...
67890	No	...
13579	Yes	...

Workshop

Build associations between the classes you have so far.

Feel free to incorporate aspects of the provided solution.

Be sure to note:

- name
- multiplicity
- conditionality

for each ‘end’, and the

- association ID

for the association as a whole.

Remember to write association descriptions.

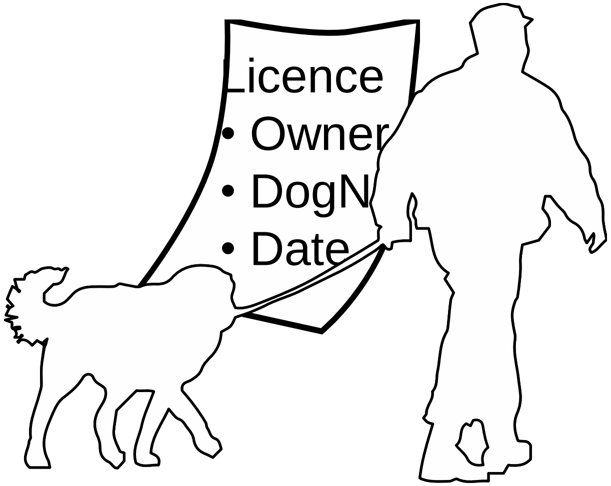
Apply the tests.

Association Classes

An *association class* is a class that comes about as a result of an association.

The association may have:

- attributes that do not describe either participating class
- behavior of its own



Association Class

An association class is a class like any other.

And an association like any other.

Class

A *class* is a *conceptual entity* within the *subject matter* at hand.

conceptual |kənˈsep ch oōəl|
adjective
of, relating to, or based on mental concepts

entity |ˈentitē|
noun (pl. **-ties)**
a thing with distinct and independent existence

subject matter |ˈsəbjəkt ˈmatər|
topic under consideration

Binary Associations

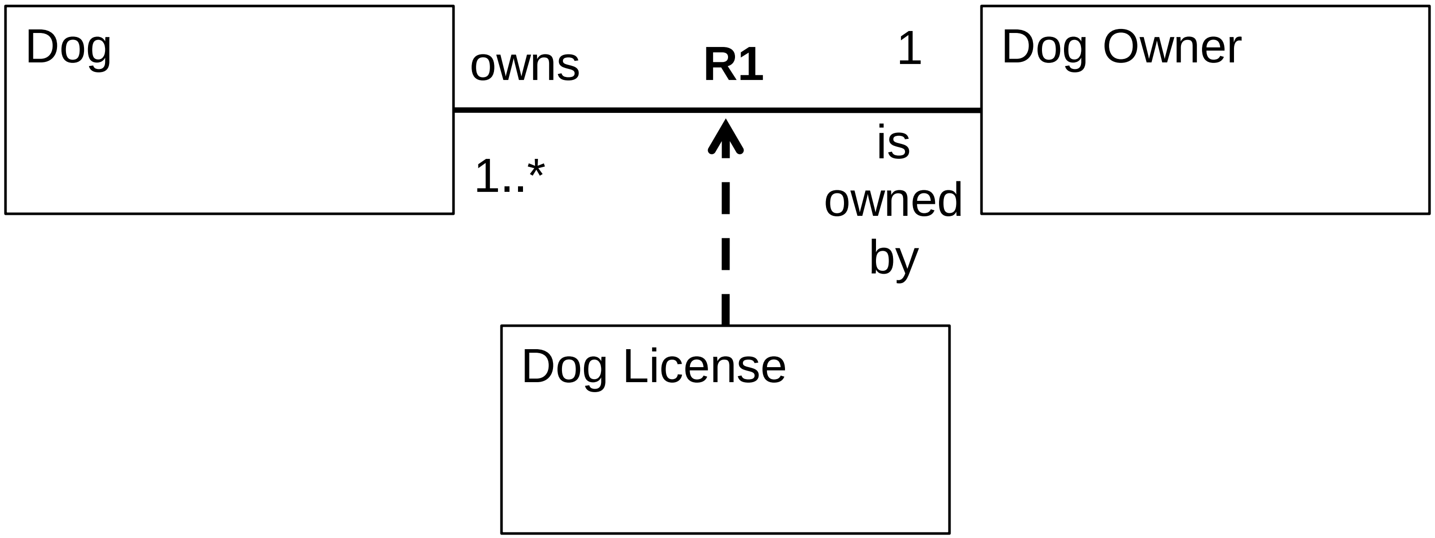
A *binary association* is an abstraction of a relationship between two things that were abstracted as classes.

Each 'end' of the binary association has a:

- name that captures the meaning of the association
- multiplicity that captures the number of instances that participate

Association Class

An association class is an association like any other.

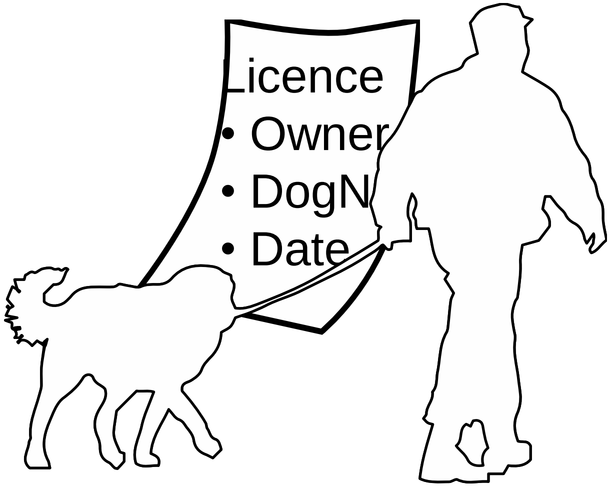


Hence the name!

Finding Association Classes

Association classes come about when:

- the association has data that describes it
- the association has behavior of some sort



Defining and Testing Association Classes

Define and test the ‘class’ bit as you would any class:

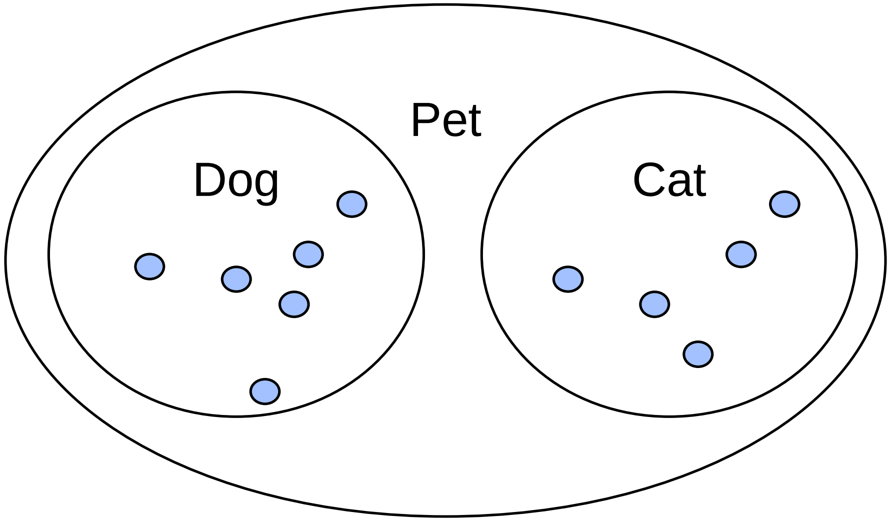
- The Uniformity test
- The OR test
- The More-than-a-list test
- The Table test
- The –er test

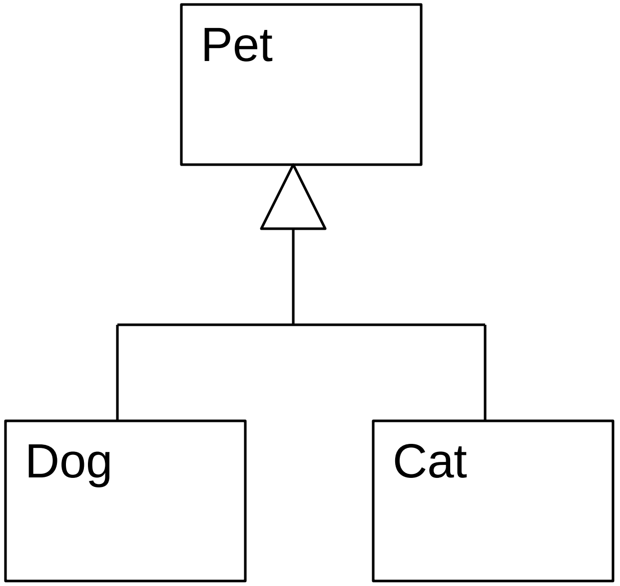
Define and test the ‘association’ bit as part of the association descriptions.

- The Multiplicity test
- The Conditionality test

Generalization

Generalization partitions a set into subsets.





It is not the same as inheritance

5. Class Modeling

5

Class Modeling

Class modeling is rarely about the classes.

It's easy to find:

- tangible classes
- classes derived from terms during clarification
- components of various sorts

But they often hide behavior of other classes.

Consider this example:

A Phone Class

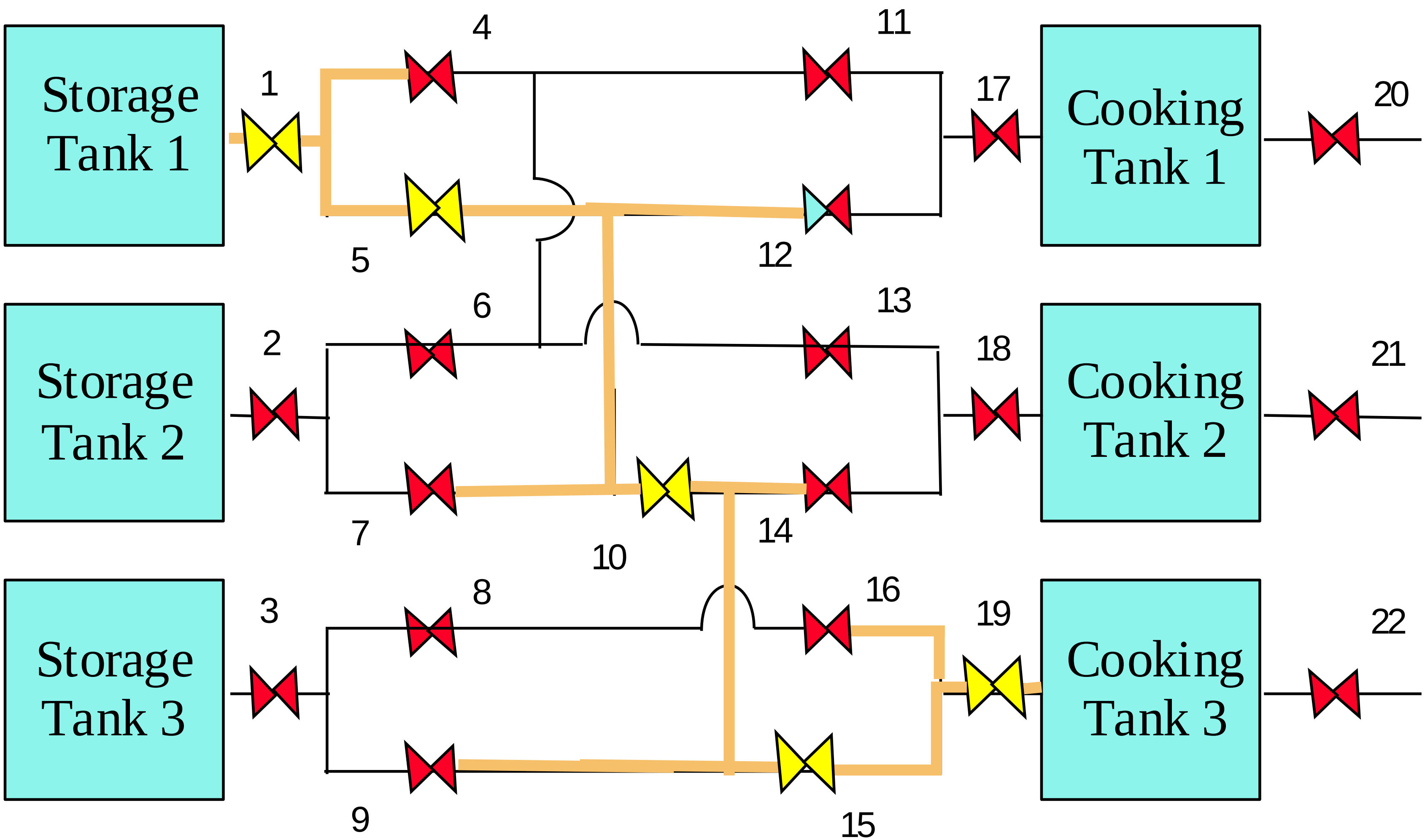
A Phone class could hide the behavior of :

- the phone itself (on/off hook)
- the making of a call
- managing call-waiting
- creating a conference call
- etc
- etc
- etc

Better to split it up into multiple classes.

another example….

What are the Classes?



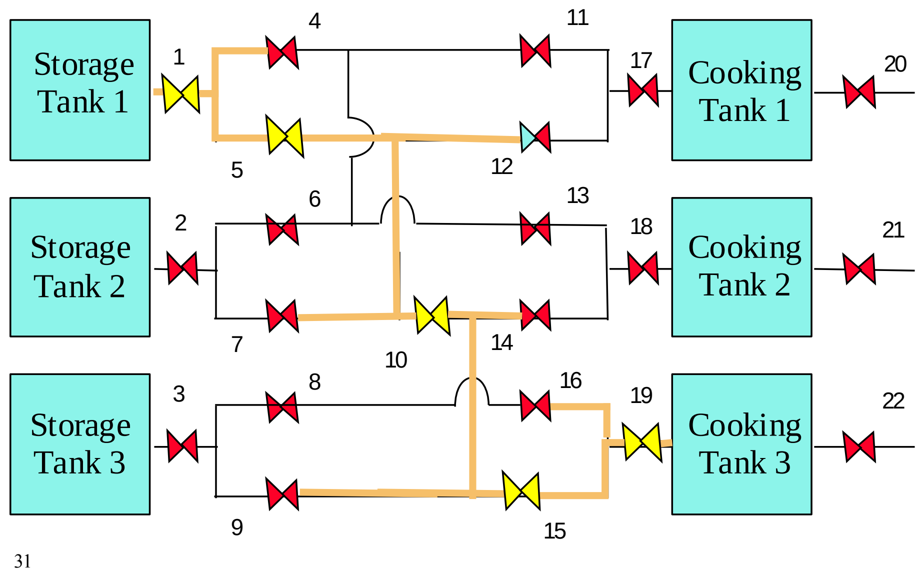
31

Workshop

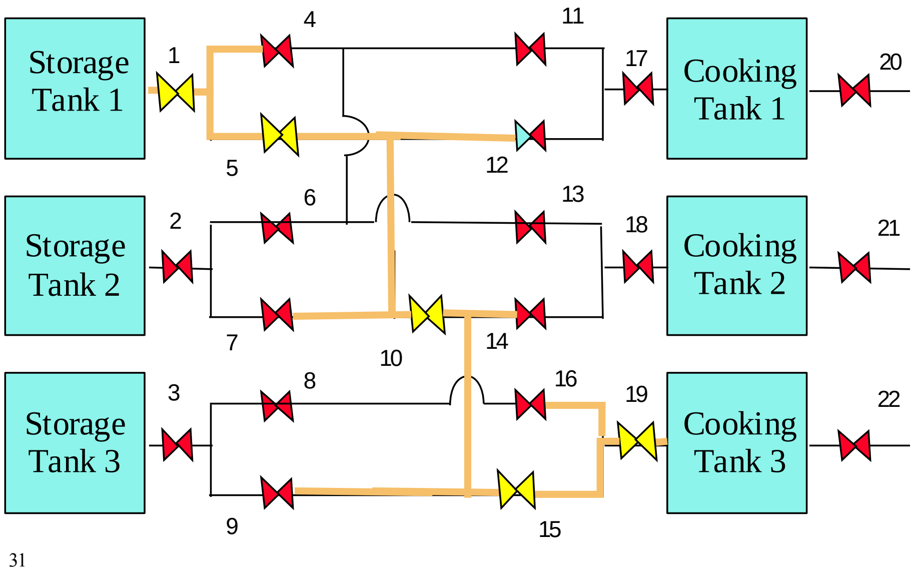
Your job is to move fluid between storage tanks and cooking tanks.

What are the classes?

How are they associated?



Simplistic Solution




```
Function OpenReservedPath(
    StorageTank,
    CookingTank);
OpenValve( StorageTank.Outlet);
If ( StorageTank.ID = 1 and
     CookingTank.ID = 3 ) then
   OpenValve( Middle );
OpenValve( CookingTank.Inlet);
EndFunction;

   OpenReservedPath( Storage1,
      Cooking3 );
```

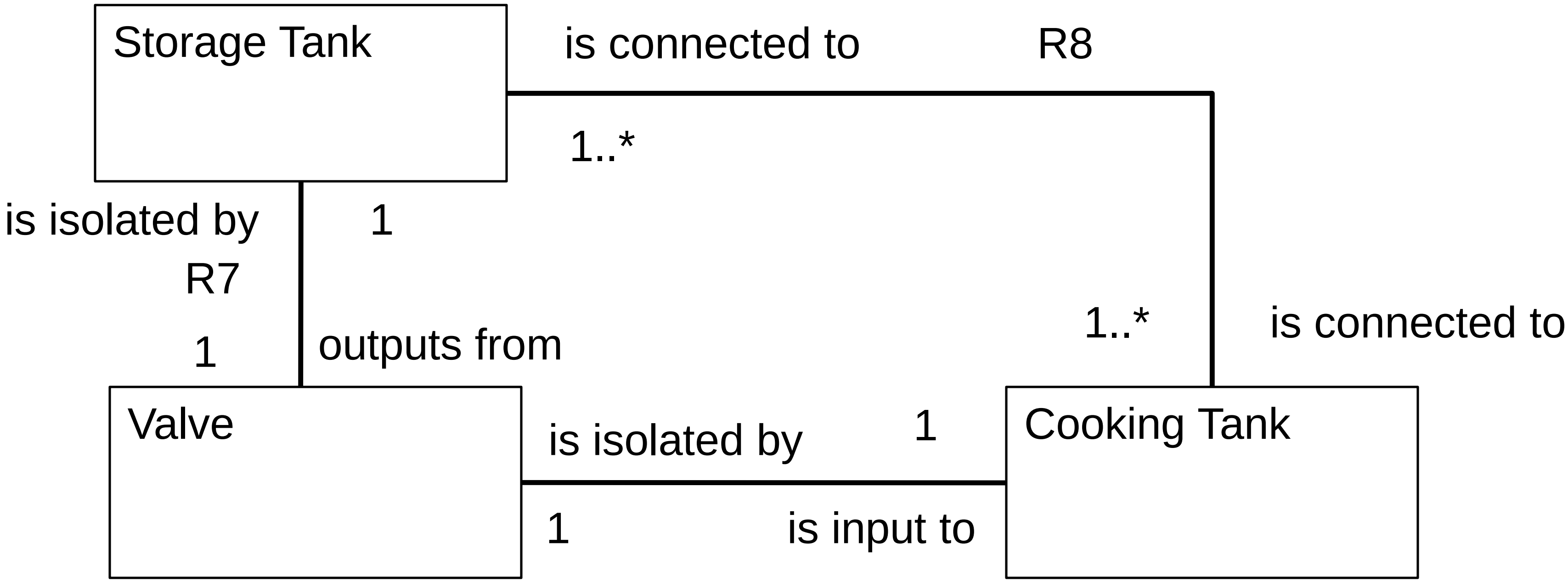
Hmm, all storage tanks have an outlet valve, and an upper or lower manifold.

And all cooking tanks have an inlet valve and two manifolds. If we know which....

Then there's valve 10-the middle one-that connects the top and bottom tanks.

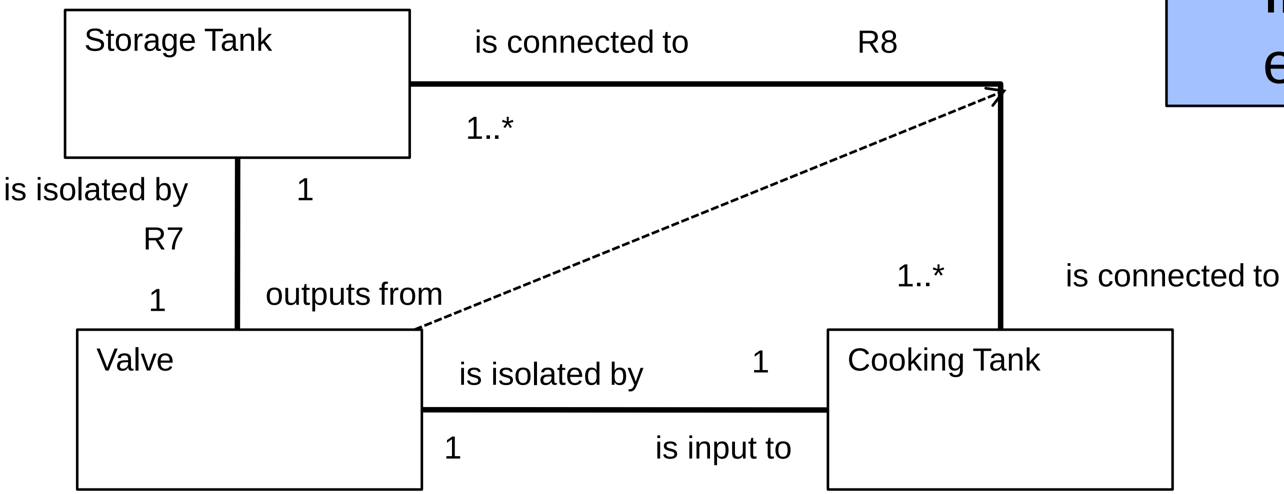
Simplistic Class Model

What’s wrong with this picture?



What's Wrong With That?

- The classes (tanks, valve etc) have complex behavior
- It's not clear where the behavior belongs
  - Should the Storage Tank empty itself?
  - Or the Cooking Tank fill itself?



In short, it's extremely brittle

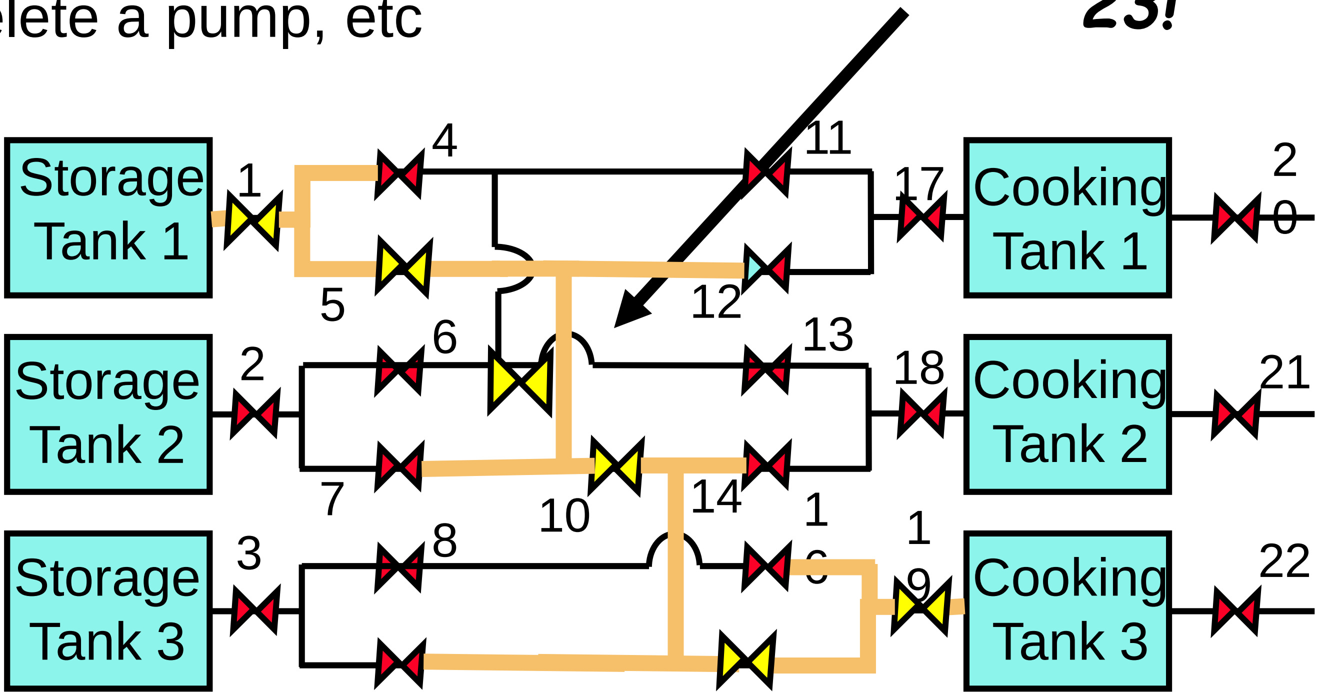
What about the intermediate valves?

Possible Changes

… we make changes:

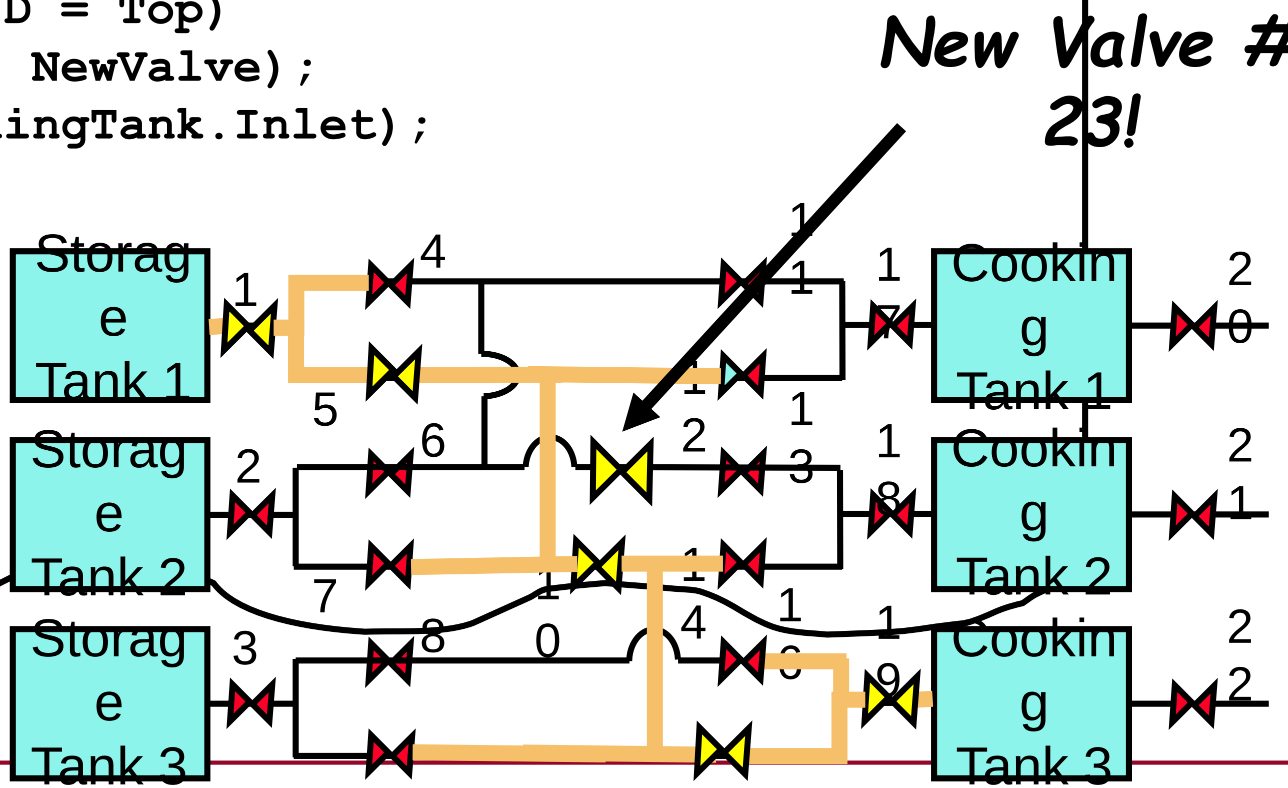
- Add a valve in the middle of a pipe
- Change target of product
- Add a new tank
- Delete a pump, etc

New Valve # 23!



Simplistic Solution

```
Function OpenReservedPath(StorageTank,CookingTank);
OpenValve( StorageTank.Outlet);
If ( StorageTank.ID = 1 and CookingTank.ID = 3 )
then OpenValve( Middle );
If ( StorageTank.ID = 2 and CookingTank.ID = 2 and
     Manifold.ID = Top)
then OpenValve( NewValve);
OpenValve( CookingTank.Inlet);
EndFunction;
```

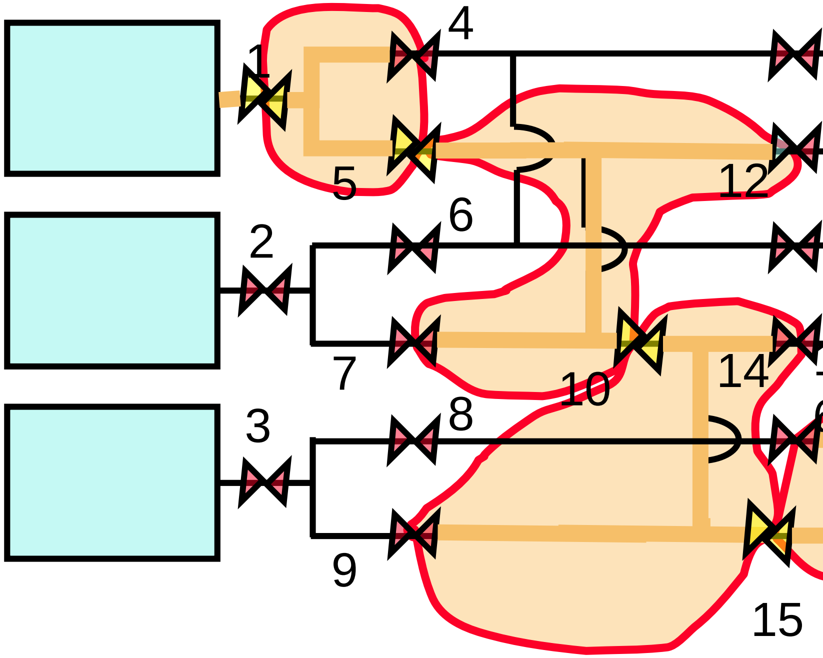


Invariants

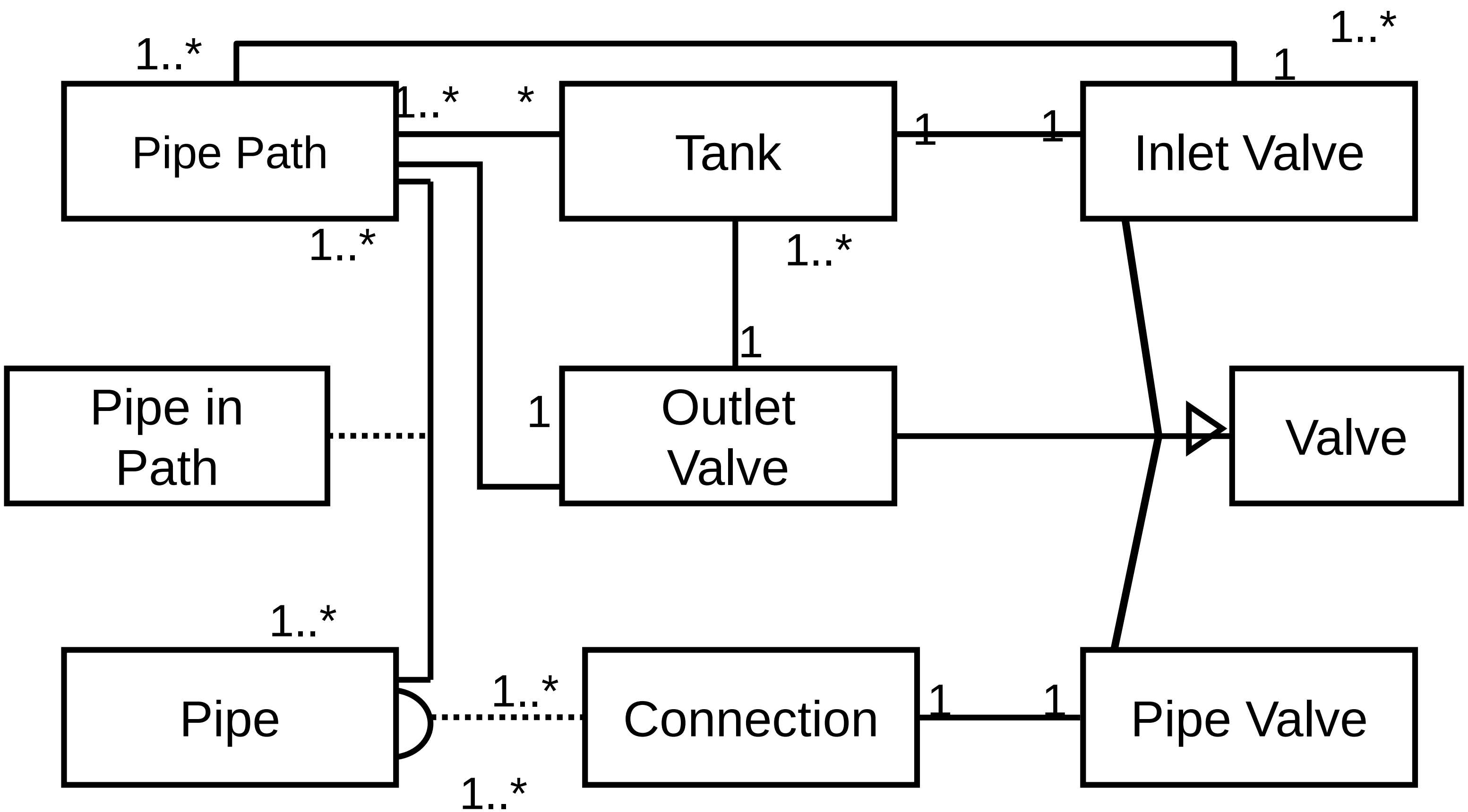
Look for the invariants:

- The facts of valve, pumps, tanks etc.
- A closed pipe will always contain the same fluid
- A pump can move fluid from one pipe to another
- If a valve is open between two pipes, they behave like a single pipe
- Closure

That is, the physics of fluids.



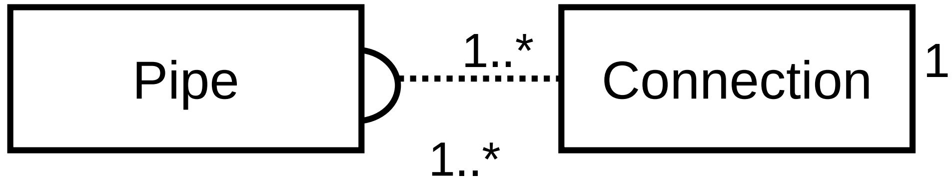
The Abstractions



The Behavior

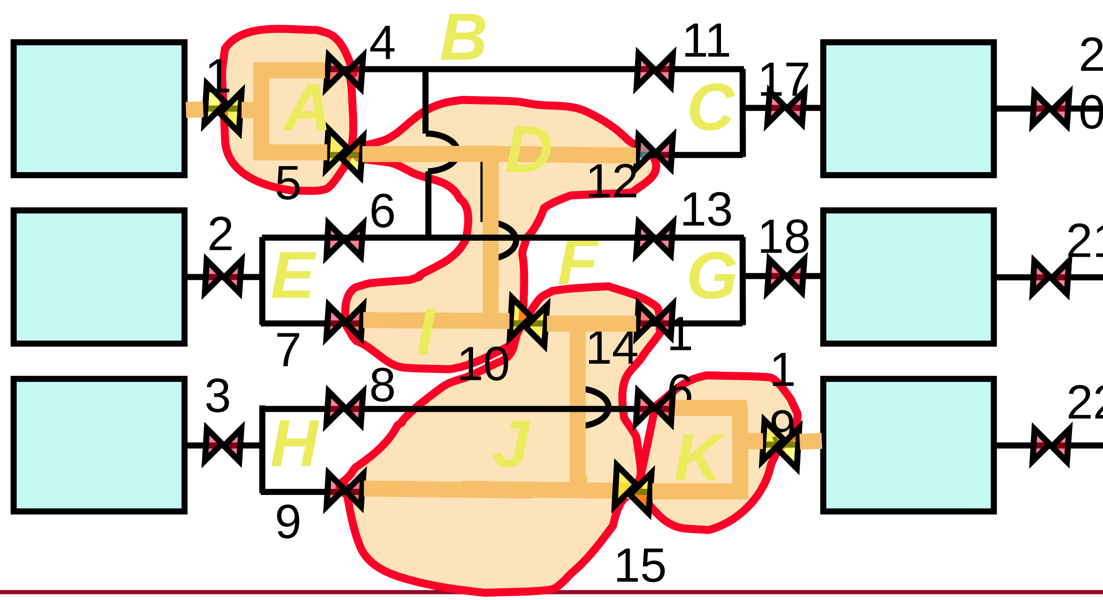
- Each Pipe shares a Connection with an adjoining Pipe.

- Each Connection has a Pipe Valve.



Connection

Pipe	Pipe	Valve
A	*D*	5
D	*J*	10
J	*K*	15



Logic

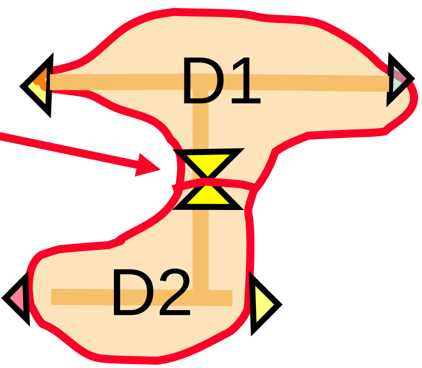
```
To open a path from a storage tank to
a cooking tank:
     Select a PipePath between the two tanks;
     Find all the Pipes in the PipePath;
     Find all the Connections between the
          Pipes in each PipePath;
     Find the PipeValve for each Connection;
     Open each PipeValve;
     Open the InletValve for the CookingTank;
     Open the OutletValve for the StorageTank;
```

Resilient to Change?

- In the world, the addition of new valve #23 is small.
- In the old abstraction, the resulting change is huge.
- In the new abstraction, only the data changes.

Pipe	Pipe	Valve
A	D1	5
D1	D2	23
D2	J	10
J	K	15

The dreaded new valve 23.



- The change in the logic is none, absolutely none.

What Did We Learn?

Putting behavior in the tangible classes makes them

- large, and
- hard to understand

Phone
* Number
* On/Off hook
* Dialing
* Number being dialed
* Call Waiting
* Conference call number
* etc

And leads to complex state models with duplicated behavior.

What Did We Learn?

To avoid that, we:

- focus on associations
- find the invariant
- build classes that control dumb devices

Here's another example:

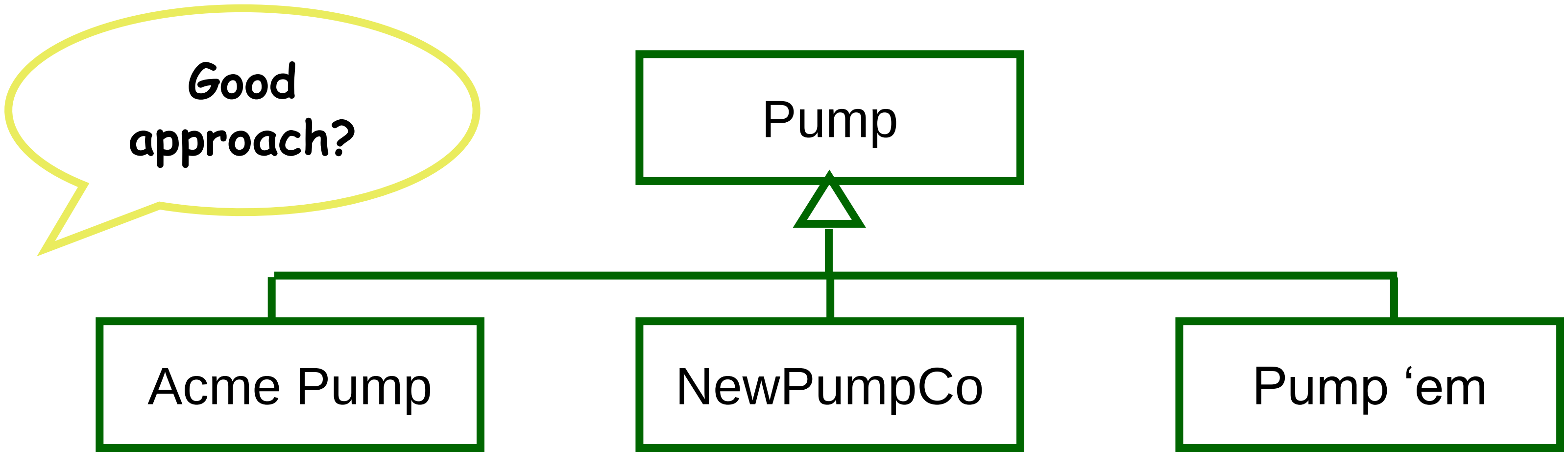
Context

The motors in some pumps require a “gearing factor” depending on the desired final speed.

Type	Speed	Value
Acme 101	0-10	speed * 1.0
	10-30	speed * 1.75
	25-50	speed * 2
NewPumpCo		No gearing
Pump ‘em	0-20	speed * 1
	20-up	speed * 3

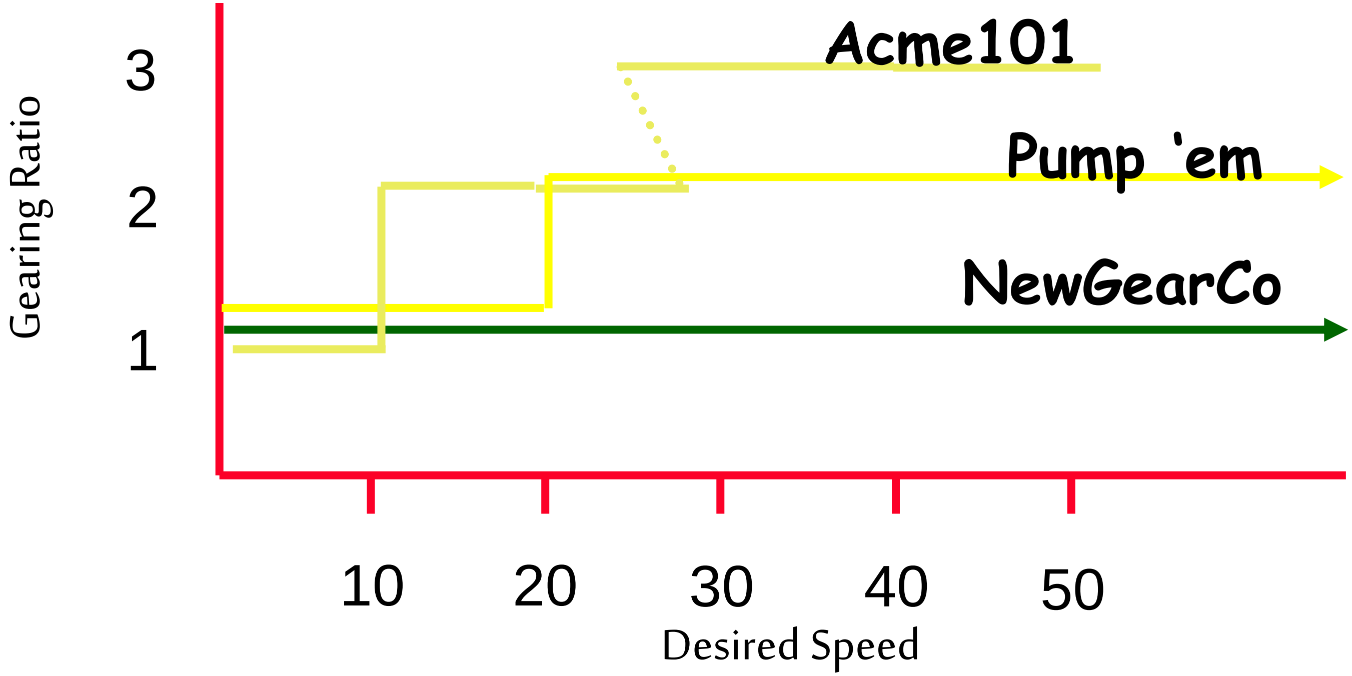
Changes

- The number of tiers
- The start and stop of each of the set points
- The variety and number of devices.
- Further idiosyncrasies of certain devices



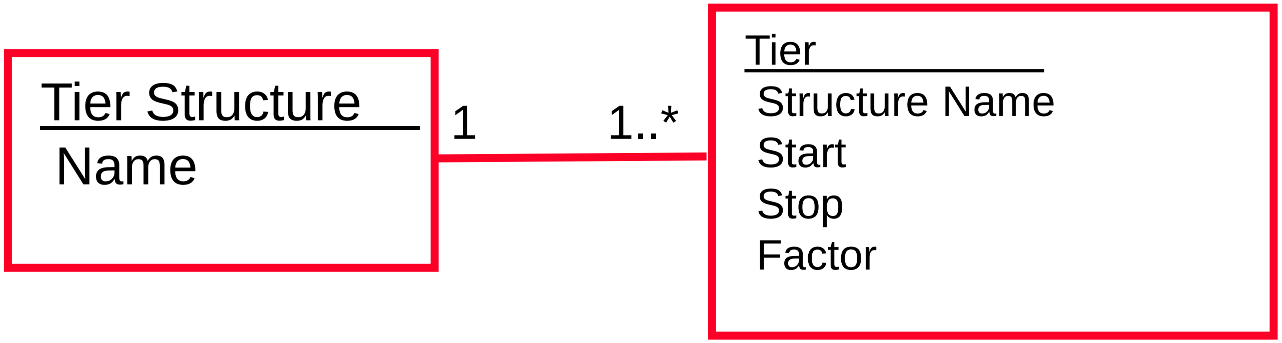
Invariants

- There are tiers
- Each tier is a line (or a ray)
- No sense in overlap



Abstractions

Abstract the tiers thus:



Tier Structure

Name
Acme 101
NewGearCo
Pump'em

Tier

Structure	Start	Stop	Factor
Acme101	0	10	1
Acme101	10	27	1.75
Acme101	27	50	2
Acme101	50	None	0
NewGearCo	0	None	1
Pump'em	0	….	…..
……...			

The Behavior

- Find the tier structure matching the pump type.
- Find the tier that contains the desired speed
- Compute desired speed * selected factor
- Send to the device

Hmmm….wouldn't tiers
work for other
devices?
Other computations?

Impact of Requirements Changes

Tier Structure

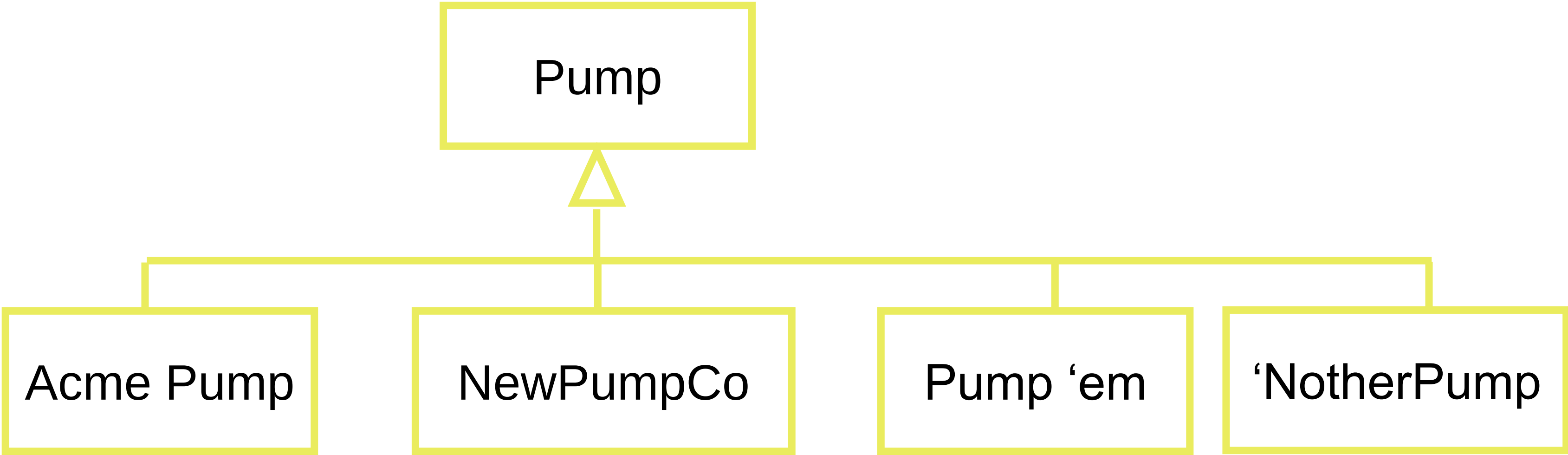
Name
Acme 101
NewGearCo
Pump'em
Nother Pump

Tier

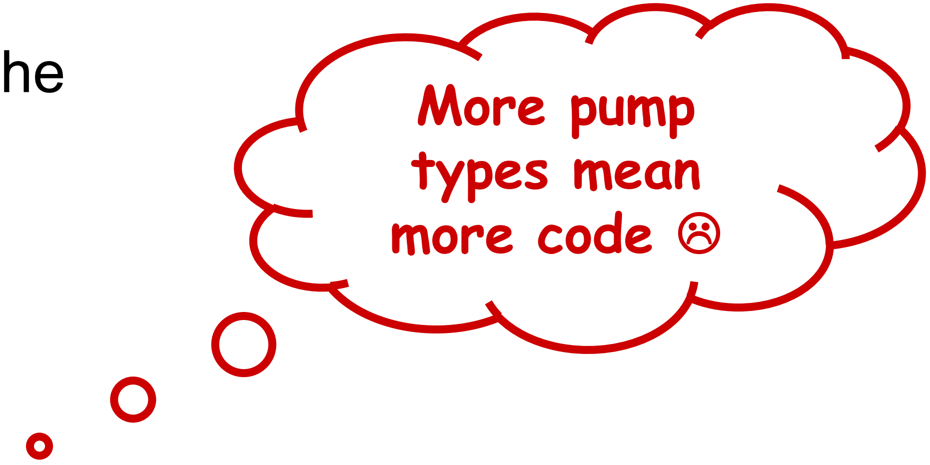
Structure	Start	Stop	Factor
Acme101	0	10	1
Acme101	10	27	1.75
Acme101	27	50	2
Acme101	50	None	0
NewGearCo	0	None	1
Pump'em	0	….	…..
Nother Pump	10	20	0.75
Nother Pump	20	45	1.75

Only if the tier concept fails do we need more code ☺

Impact of Requirements Changes



The impact is dependent on the abstractions *you* select.



Remember!

- Beware of "-er" classes (e.g. Handler, Manager…)
- A good class model is simple and easily understandable even to the subject matter newbie
- A more elaborate class diagram usually results in simpler state models

Class Models

Class models capture:

- abstractions of the things in and around the system.
  - Ideally the invariants, the things and concepts inherent in the subject matter regardless of the current requirements.
- associations between the instances of these abstractions.
- rules about the instance population at any point in time.
- a single subject matter.
- The granularity of reuse is the entire class diagram for a subject matter, not individual classes.

Good Class Models

Good class models:

- Are easily understood by:
  - Experts in the subject matter so they can verify/dispute its correctness
  - Those new to the subject matter so they can learn the domain
- Expose more information on the class diagram to lead to simpler state models.
- Expose information, not hide (“encapsulate”) it

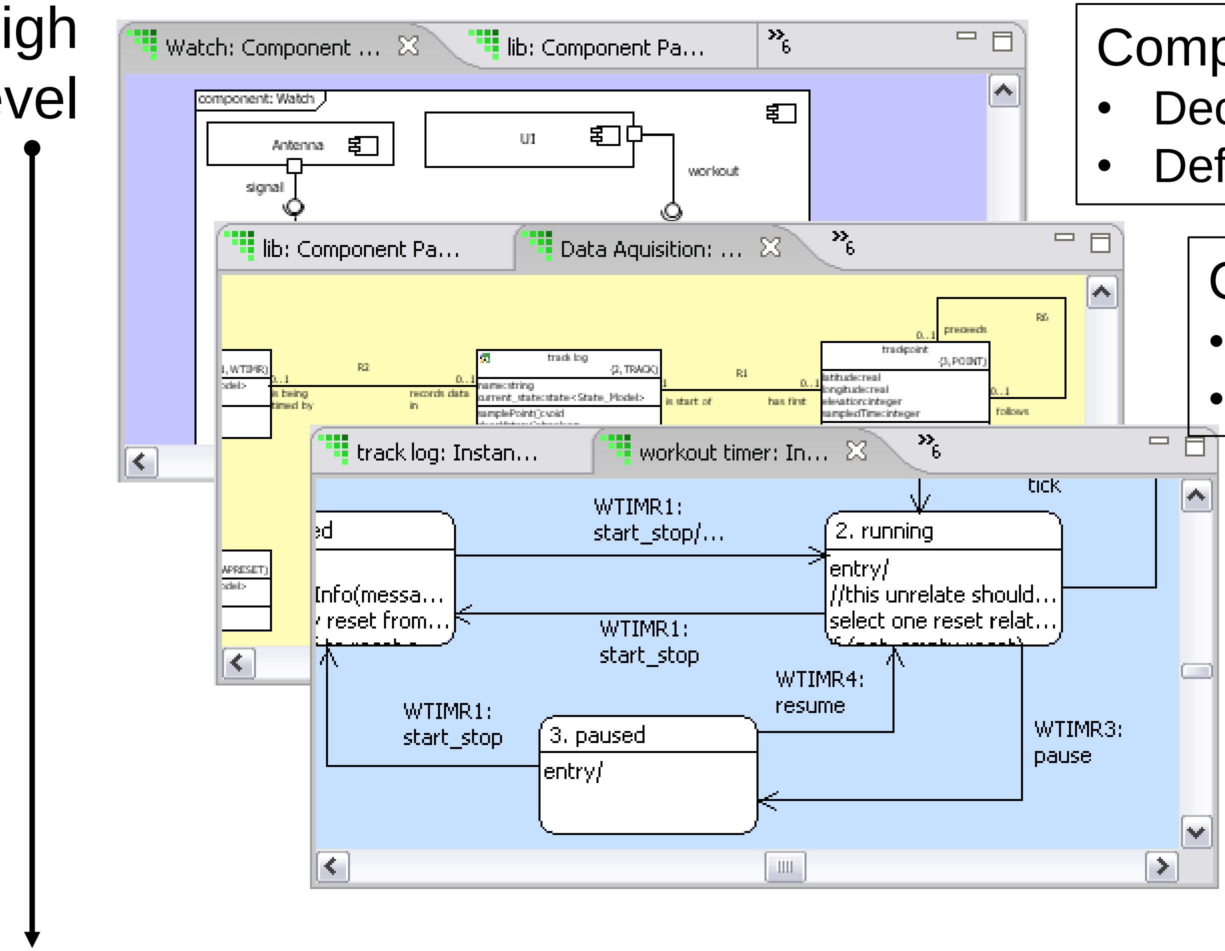
6. State Models

6

Executable Model Hierarchy

High level

Low level



Component Diagram

- Decompose the application
- Define Interfaces

Class Diagram

- Abstractions
- Operations

State Diagram

- Lifecycle
- Event handling

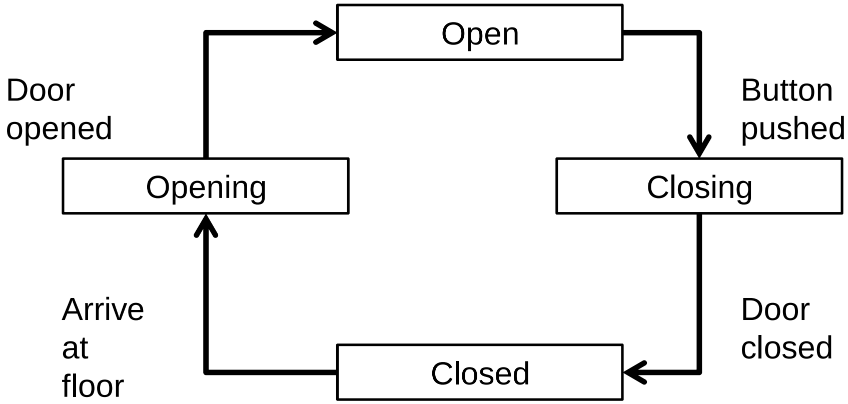
State Models

Some instances progress through stages during their lifetime.

The collection of stages and the order of progression constitutes its *lifecycle*.

It is represented as a *state model*, which may be captured as:

- a state diagram
- a state-event matrix

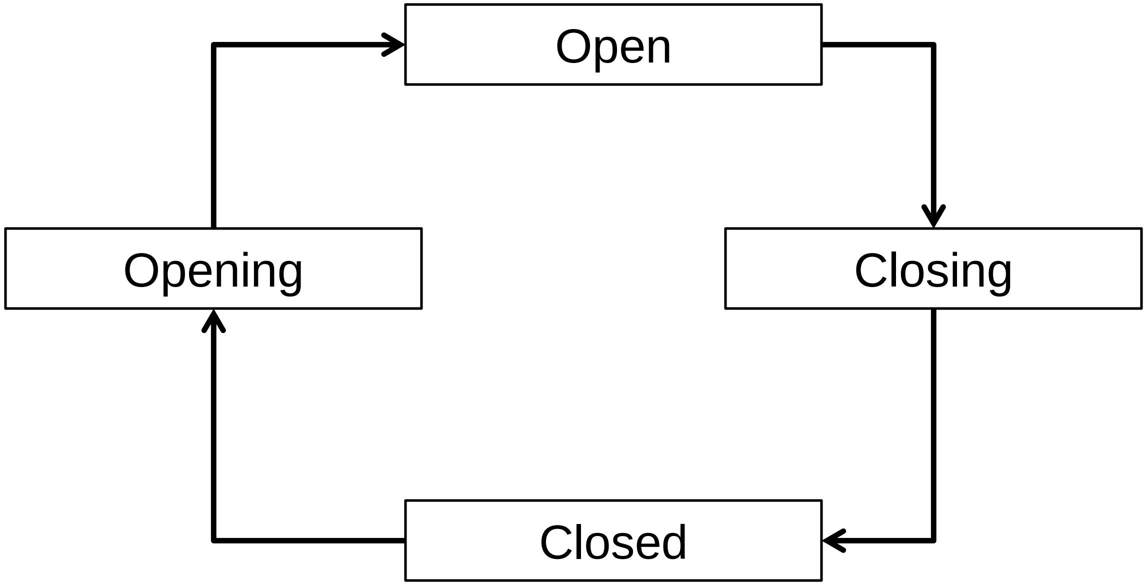


	Button pushed	Door closed	Arrive at floor	Door opened
Open	Closing			
Closing		Closed		
Closed			Opening	
Opening				Open

State Models

A state diagram comprises:

- States
- Transitions
- Events
- Activities



We'll talk about each in turn.

States

A *state* is an abstraction of a stage in an instance's lifecycle.

Open

Closed

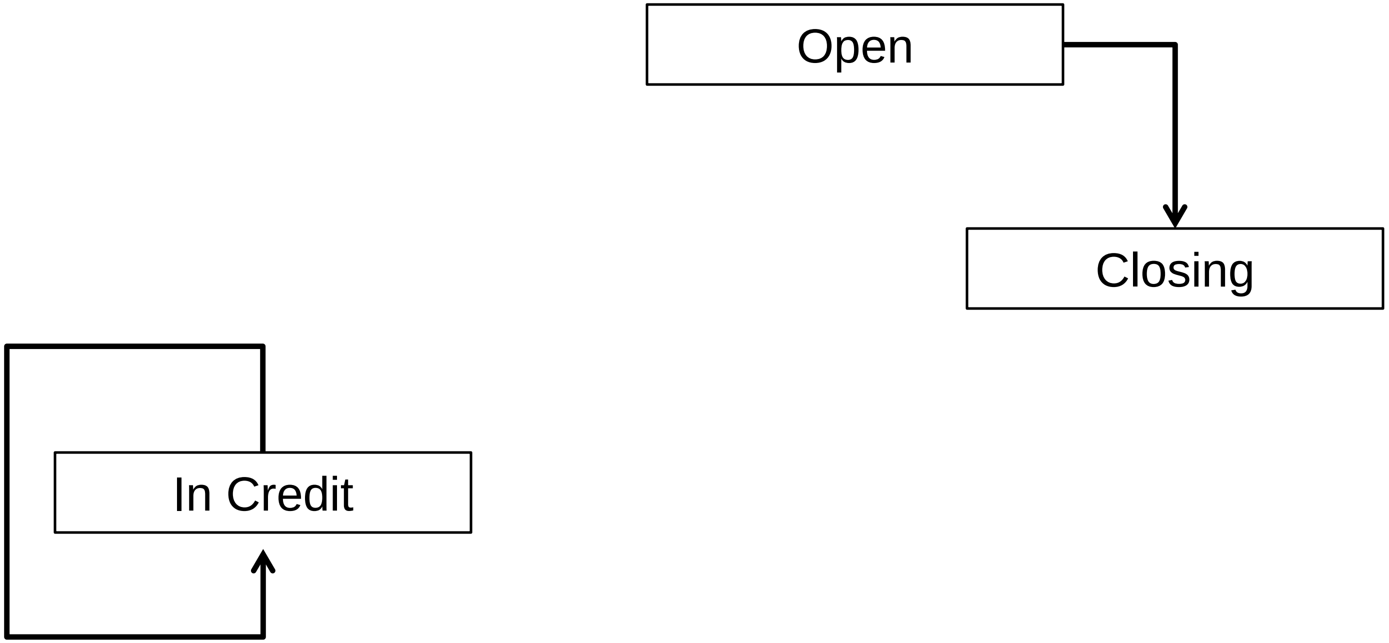
Opening

Closing

WARNING: Many mathematical and object-oriented texts use "state" to mean the values of *all* the attributes.

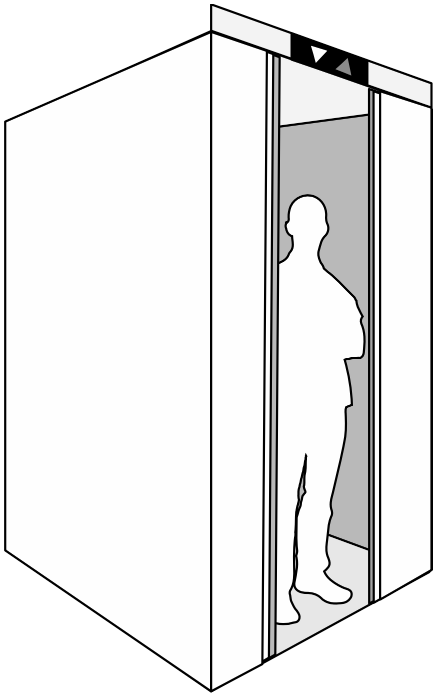
Transitions

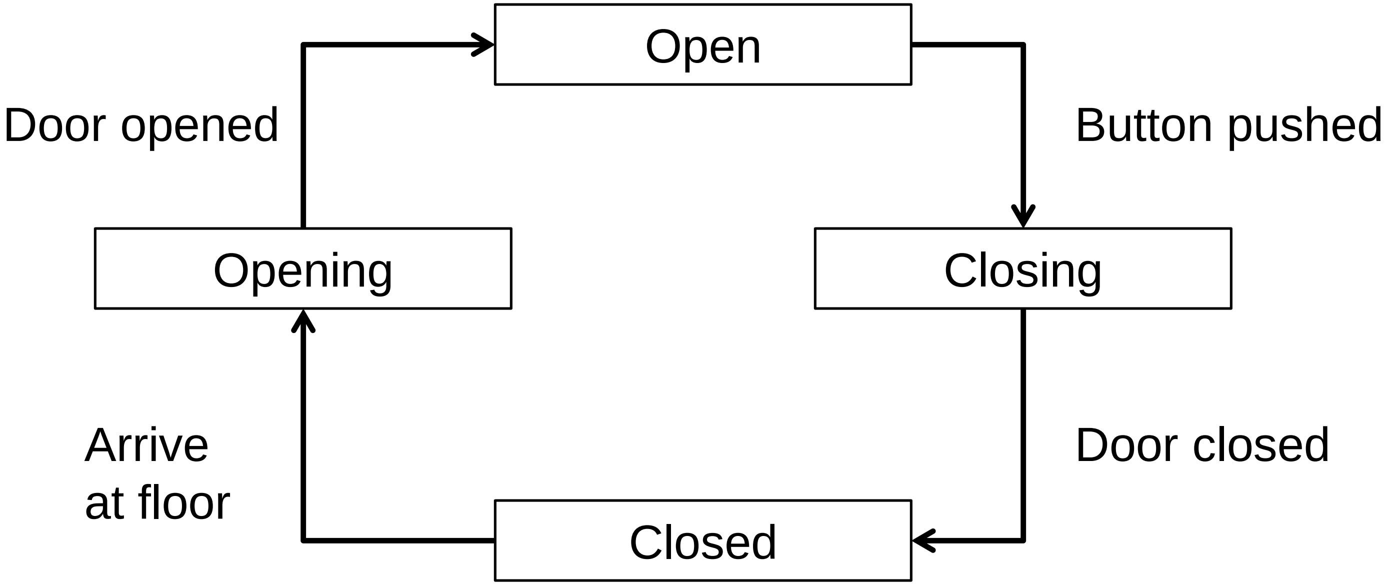
A *transition* is a change from one state to another (possibly the same) state.



Events

An *event* is an abstraction of a real-world incident that causes the instance to move from one state to another.

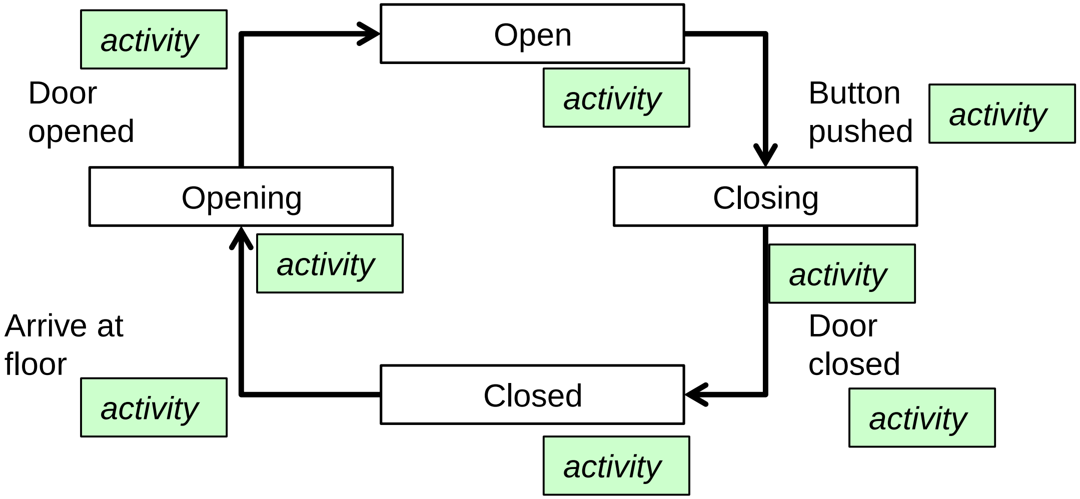




Activities

An *activity* comprises a collection of actions that *do* something:

- Create and delete instances
- Read and write attributes
- Create and delete links
- Perform logic and arithmetic
- Send events to other state machines

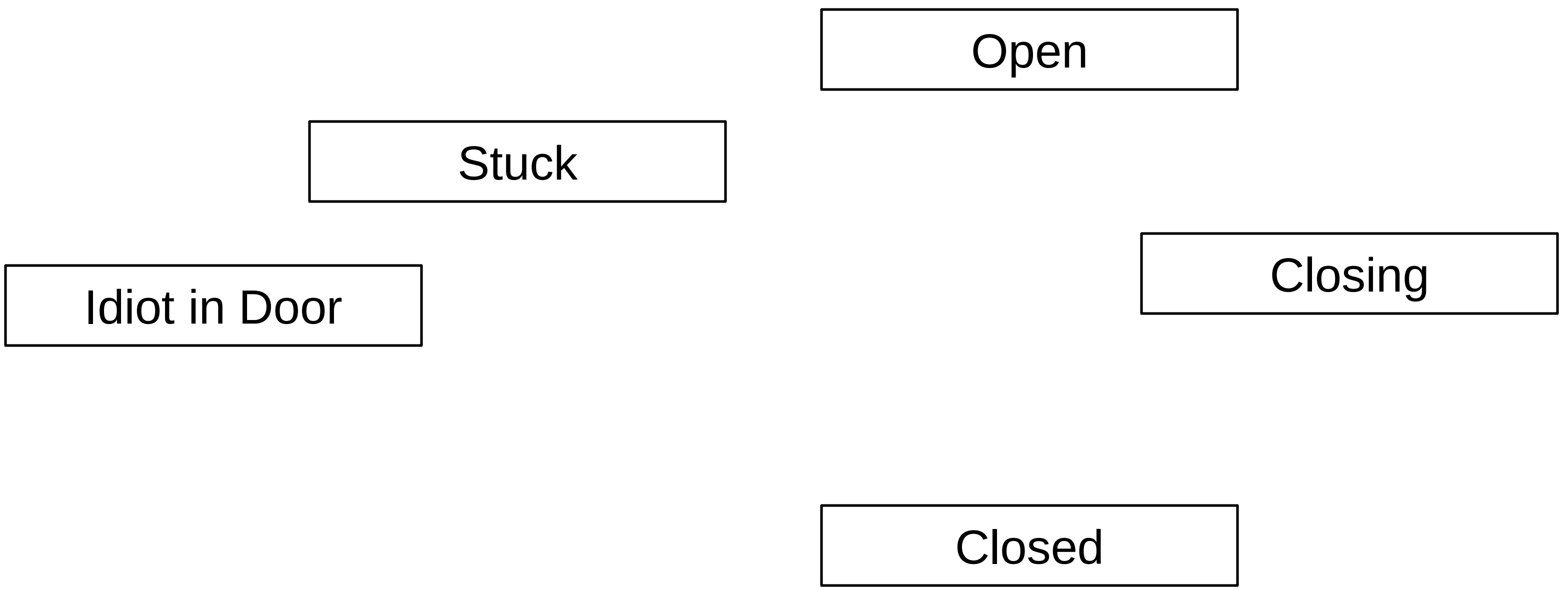


Activities may execute on the transition or on entry to the state.

Finding States

Enumerate the states you know.

If necessary, write a comment to describe the state further.



Blitz

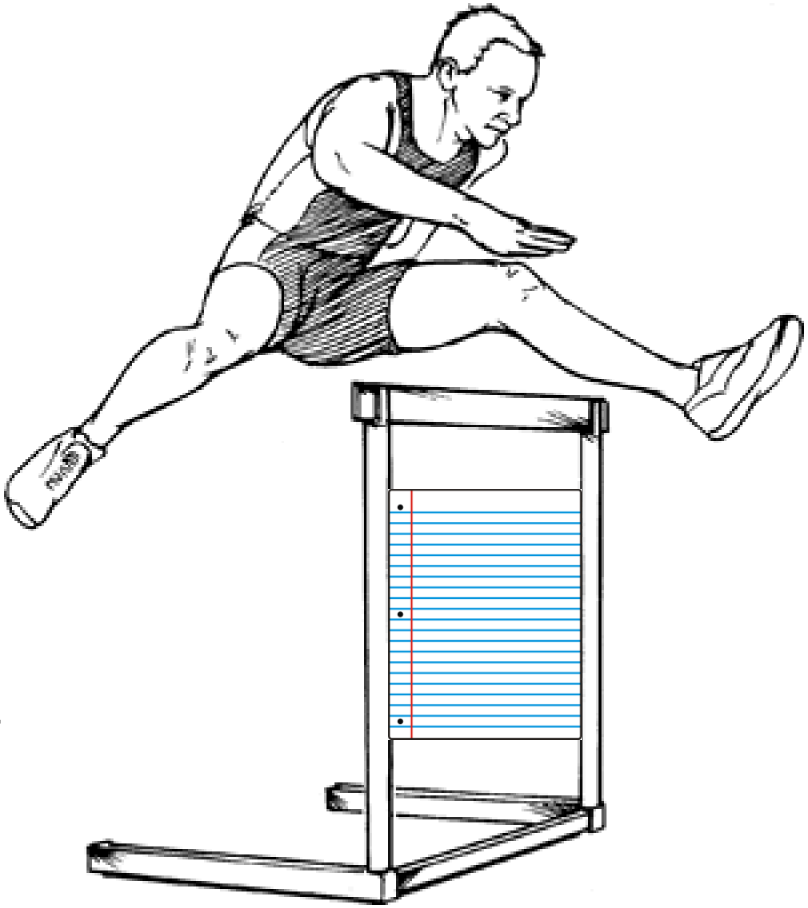
A *blitz* is a technique for getting started.

There are no wrong answers.

- We don't categorize
- We don't organize
- We don't evaluate

- We *just enumerate*

The purpose is to provide a start



State Blitz

Look at all the candidates and categorize them.

- Definitely a state
- Maybe a state
- Definitely not a state

Open

Stuck

Closing

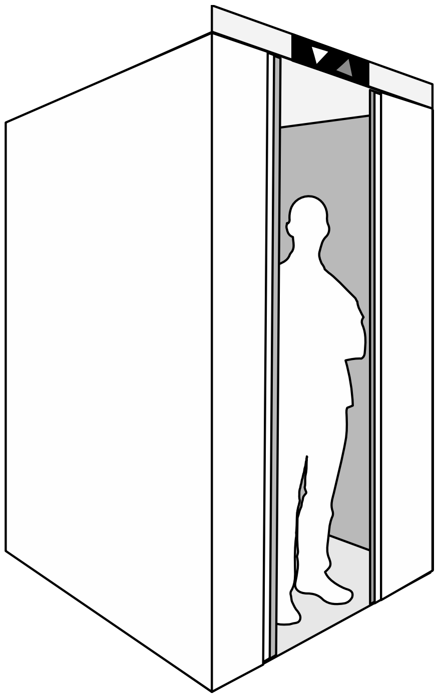
Idiot in Door

Closed

One State at a Time

An instance is in exactly one state at a time.

Choose states so that the instance is *always* in one state.



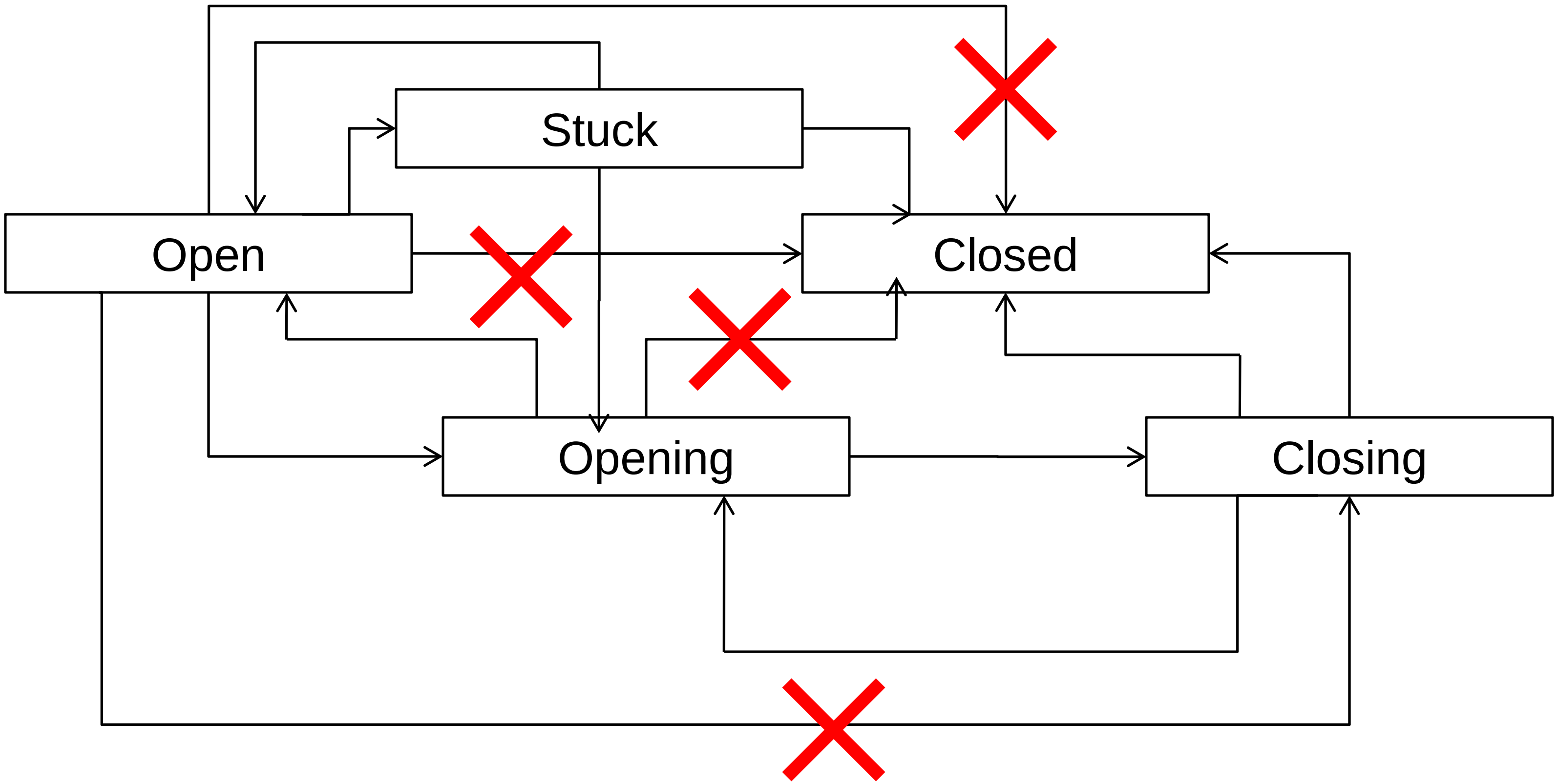
Open

Closing

Closed

Finding Transitions

Show the possible transitions from one state to another.



Finding Patterns

Cyclic

- Reusable resource such as equipment, link etc.
- Usually returns to a state in which nothing is happening, named according to the subject matter

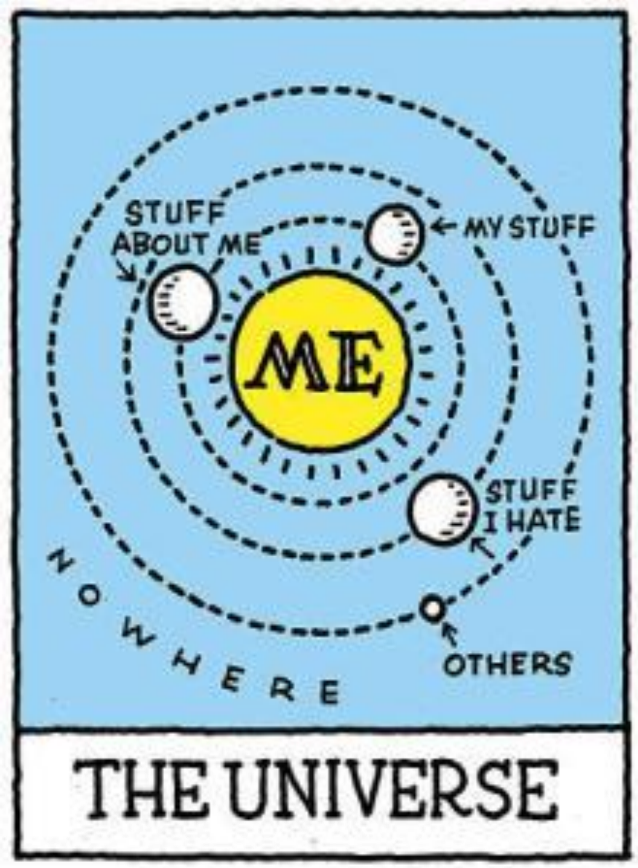
One shot

- Manage an action that takes time to complete
- No record of action is required (Born and Die)
- Record of action is required (Born and Quiescent)

Anthropomorphize

Take the perspective of an instance:

- How do *I* come into existence?
- What happens to *me* to cause *me* to change state?
- Where do *I* go from here?



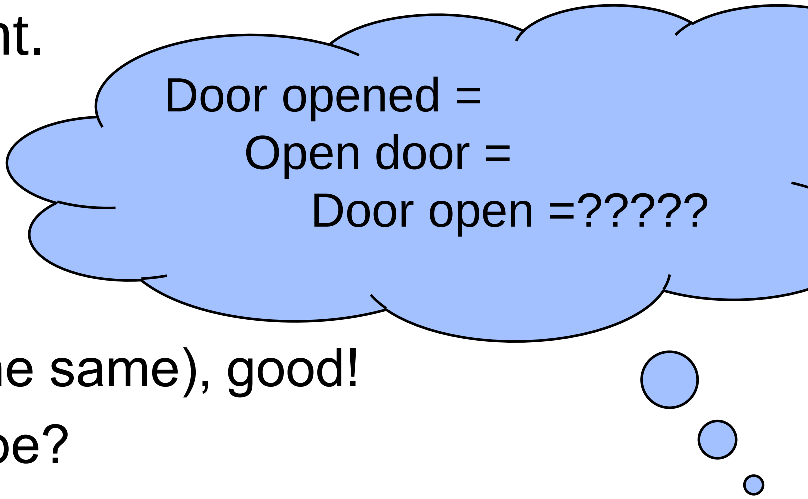
Workshop

Find and name the states for the Pub system shown at the beginning of the section on classes.

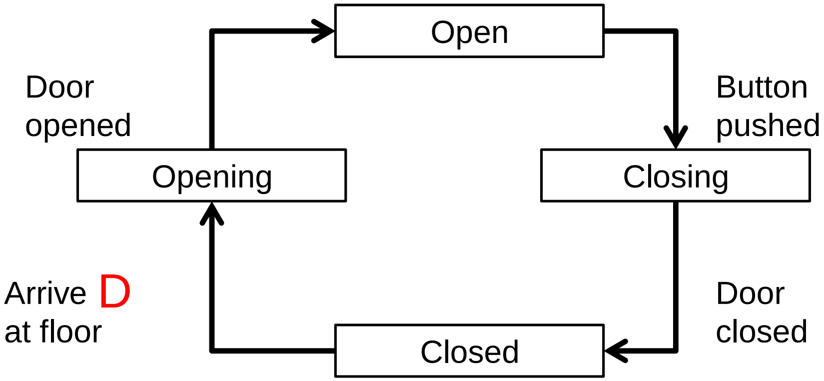
Find and draw all the legal transitions.

Identify Events

For each transition, identify the event.

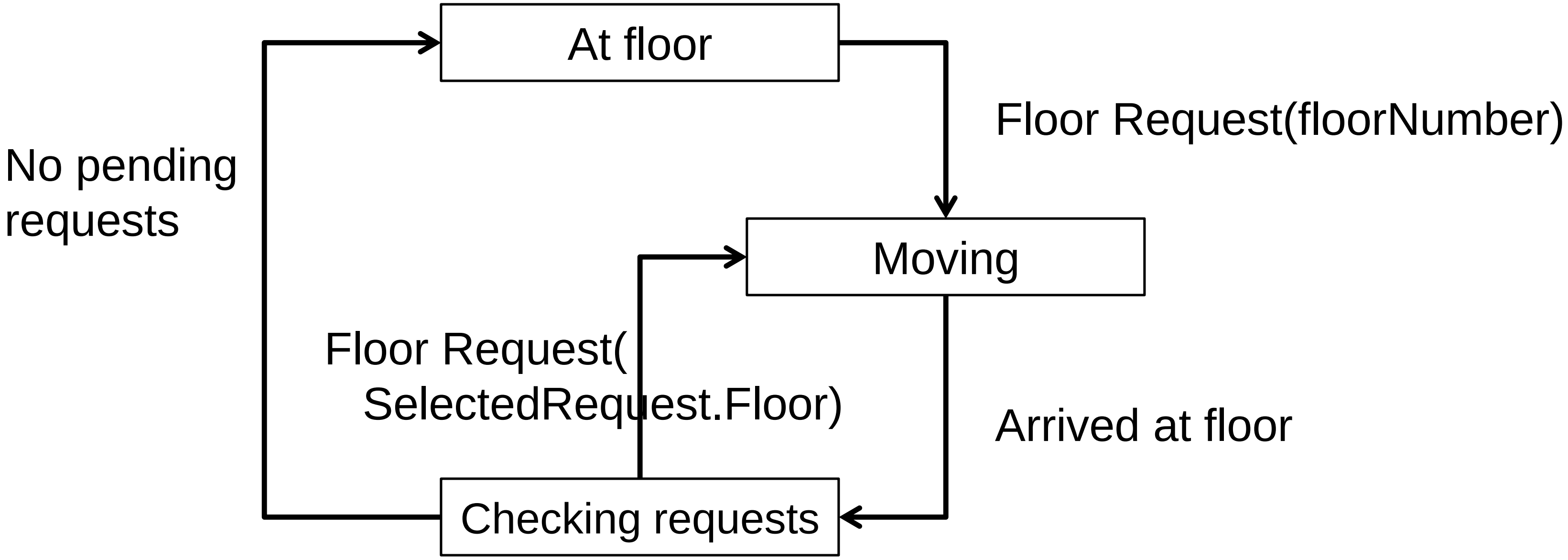


- Propose a name for the event
- Check all other event names
  - If it’s the same (and means the same), good!
  - If it’s not the same, should it be?
- Make event names consistent in structure



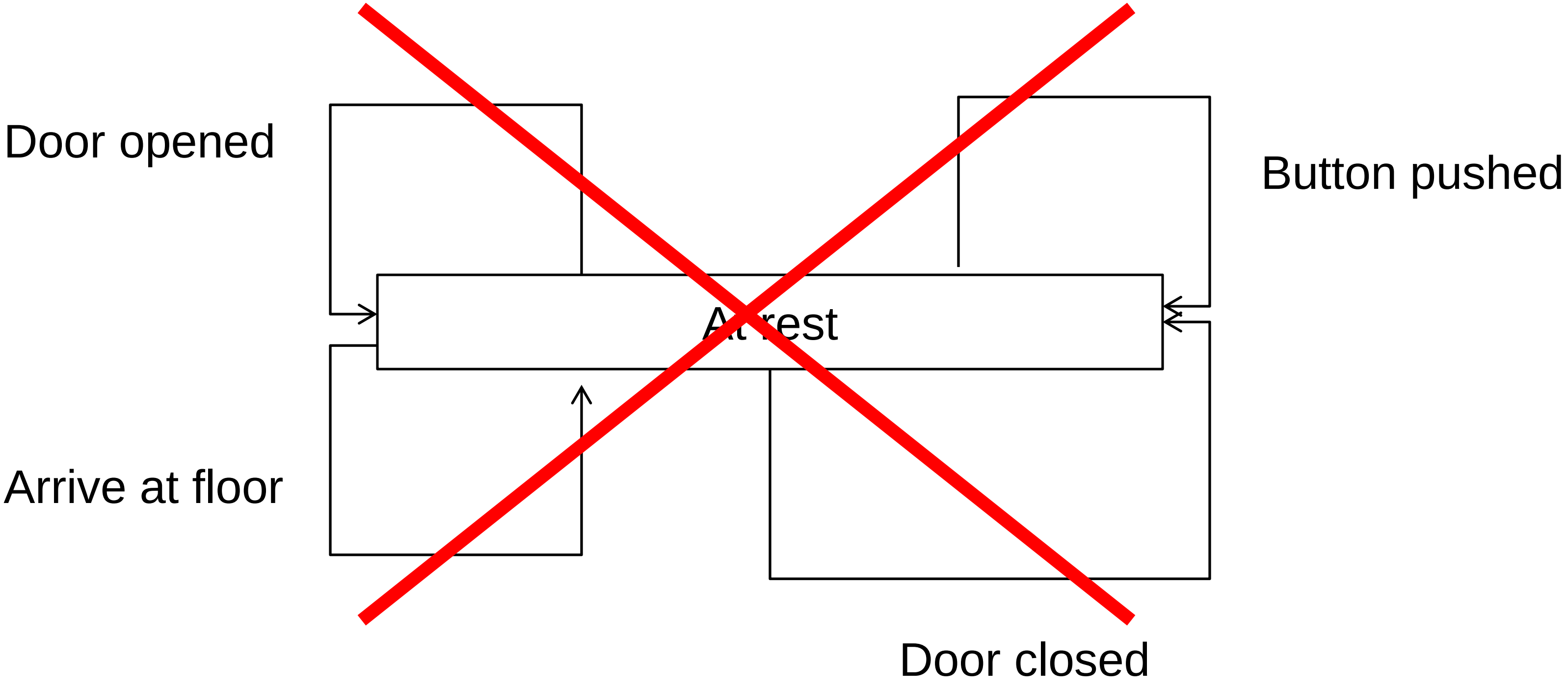
Event Data

Events may carry data with them.

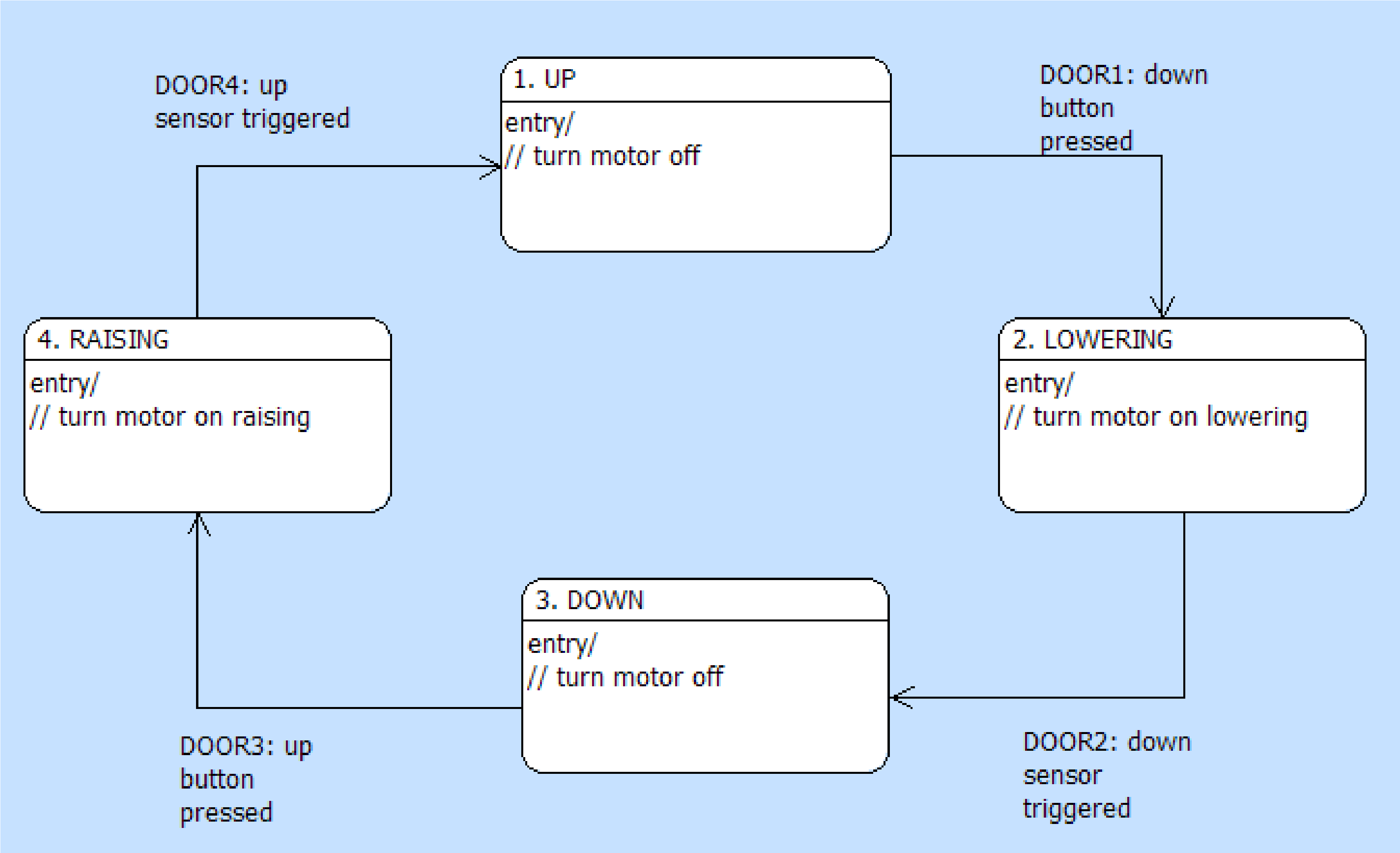


Anti-Pattern

The state diagram should reflect a lifecycle, not a set of things to do.



Garage Door State Model



State Models

The process for building state models:

- Enumerate, name, and describe as necessary the states.
- Define the legal transitions through exhaustive enumeration.
- Define an event for each transition, reusing existing events as appropriate.
- Use comments to describe activities.

Workshop

For each transition in the Pub system:

- pick a name for the event that drives the transition
- add it to the diagram

Don’t forget to check the list of existing events.

Testing the State Model

A *state-event matrix* is used to check for completeness of a state diagram.

It has:

- columns for events
- rows for states

	Down button	Down sensed	Up button	Up sensed
UP	LOWERING	Can't happen	Event Ignored	Can't happen
LOWERING	Event Ignored	DOWN	RAISING	Can't happen
DOWN	Event Ignored	Can't happen	RAISING	Can't happen
RAISING	LOWERING	Can't happen	Event Ignored	UP

Each cell contains:

- the name of the new state
- links to the activities

Testing the State Model

Establish whether there is a transition from each state to *every other state*.

Events

	Down button	Down sensed	Up button	Up sensed
UP	LOWERING			
LOWERING		DOWN		
DOWN			RAISING	
RAISING				UP

States

Fill in the State-Event Matrix

Examine each cell and fill in the destination state.

	Down button	Down sensed	Up button	Up sensed
UP				
LOWERI				
DOWN				
RAISING				UP

WARNING: Do *NOT* look at the diagram

What about the empty cells?

Empty Cells

The empty cells can be:

- A transition you forgot ➔
  Fill in the destination state
  Go back the diagram and fix it too
- An event that occurs, but you don't care ➔
  Ignore it ("Event Ignored")
- A logical impossibility ➔
  Something has gone horribly wrong ("Can't Happen")

Event Ignored

An event can occur that you simply ignore.

Can’t Happen

An unexpected event can occur that likely indicates a:

- a software fault
- a hardware fault

You can’t do anything about it.

Use “Can’t Happen” for situations from that cannot be recovered or handled by the application models.

> WARNING: “Can’t Happen” ≠ “Shouldn’t Happen”

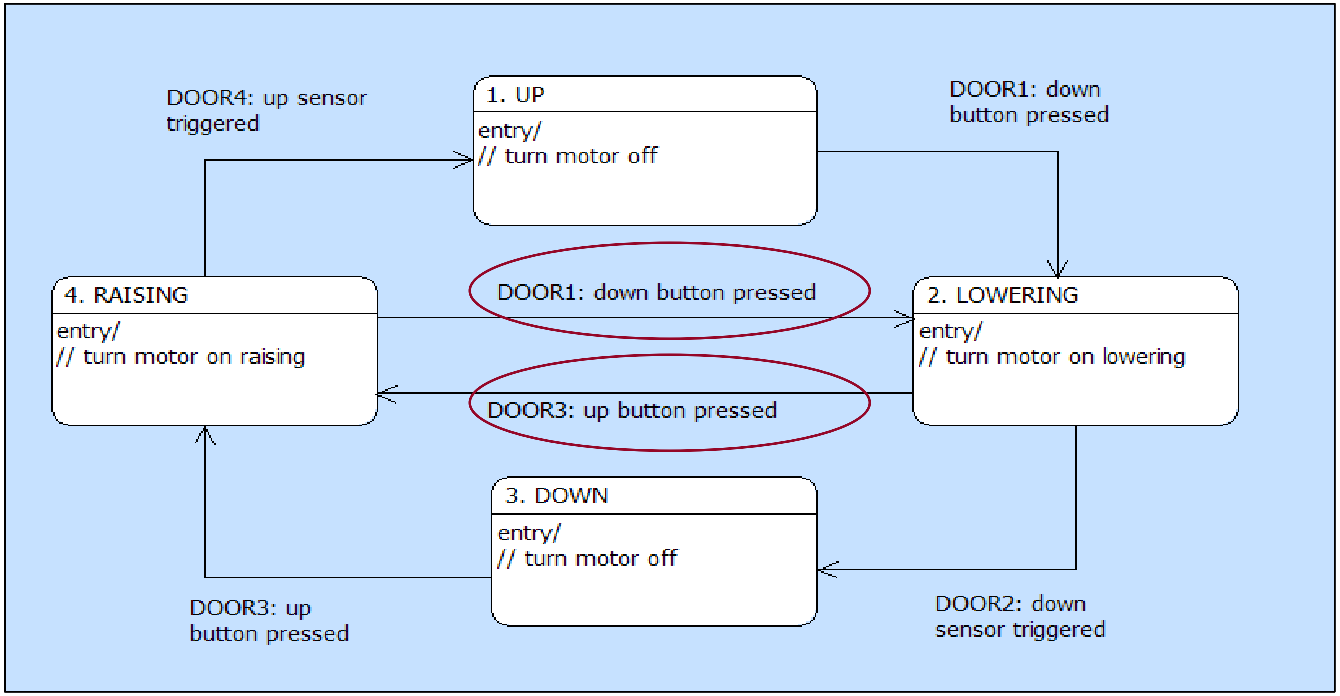
If an undesirable event occurs, you have to handle it.

Filling the State Event Matrix

	Down button	Down sensed	Up button	Up sensed
UP	LOWERING	Can't happen	Event Ignored	Can't happen
LOWERING	Event Ignored	DOWN	RAISING	Can't happen
DOWN	Event Ignored	Can't happen	RAISING	Can't happen
RAISING	LOWERING	Can't happen	Event Ignored	UP

Completed Diagram

You may add descriptions to any model element that relate the model element to the subject matter under study.



Workshop

Fill in the state-event matrix for a state model that you constructed.

For each cell (state **x** event), indicate whether it's a:

- Transition to another (named) state,
- Ignore,
- can't happen or
- shouldn't happen

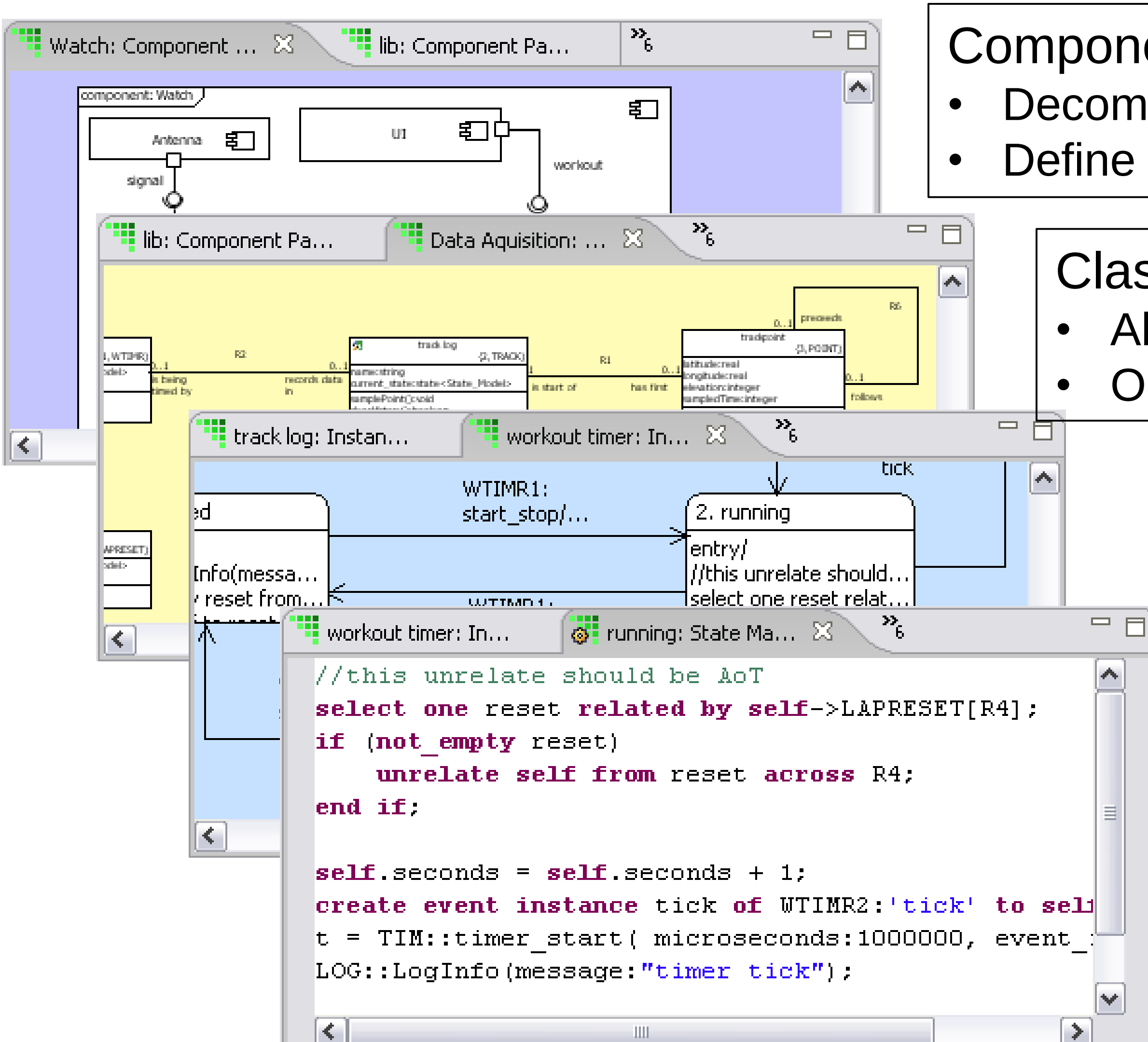
7. Activities

7

Executable Model Hierarchy

High level

Low level



Component Diagram
- Decompose the application
- Define Interfaces

Class Diagram
- Abstractions
- Operations

State Diagram
- Lifecycle
- Event handling

Activities
- Processing

Activities

An activity is a block of *model-level logic* comprising a collection of actions that can:

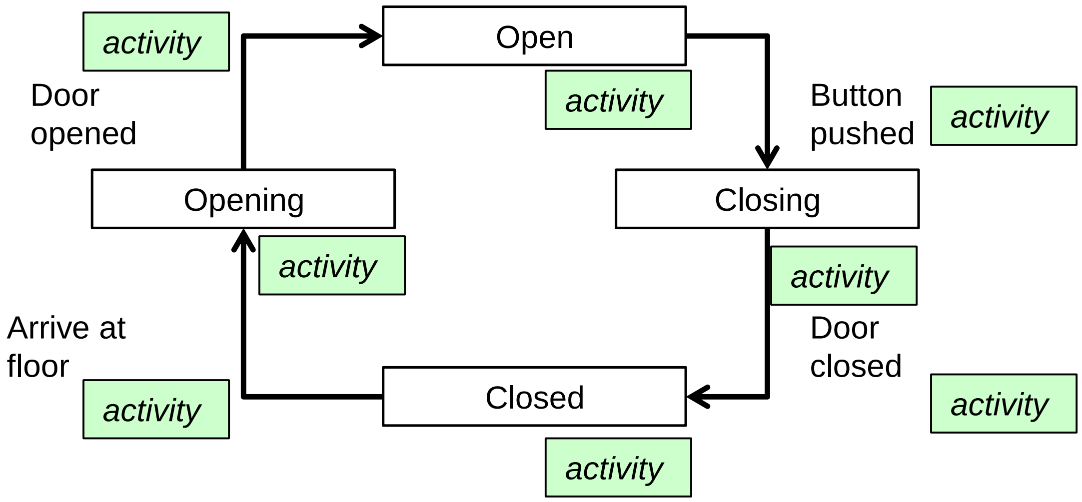
- Create and delete instances
- Read and write attribute values
- Compute new values
- Generate events
- Link and unlink associations between instances
- Select instances across associations
- Find instances based on attribute values
- Communicate with the outside world

Activities

You can place an activity pretty much anywhere.

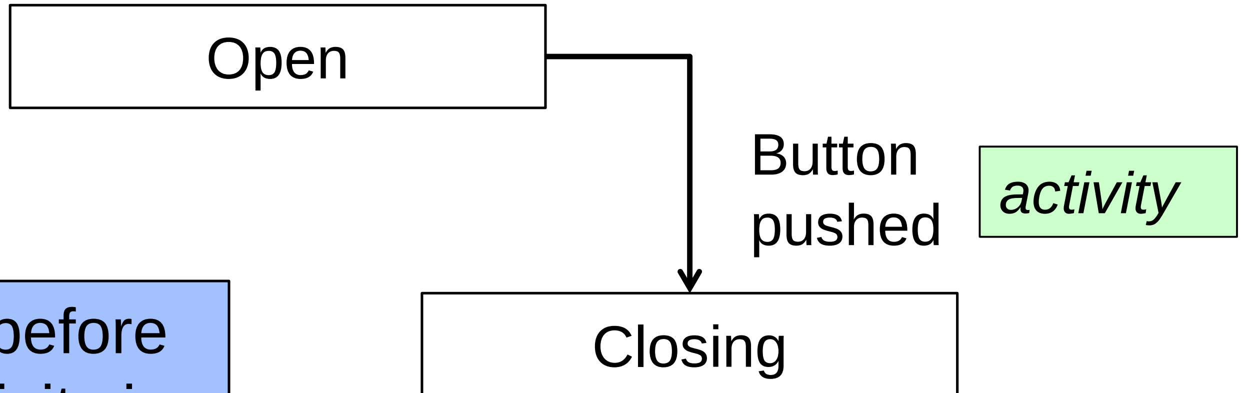
In the context of a state model, that means

- on a transition
- on entry to a state



Activities on Transitions

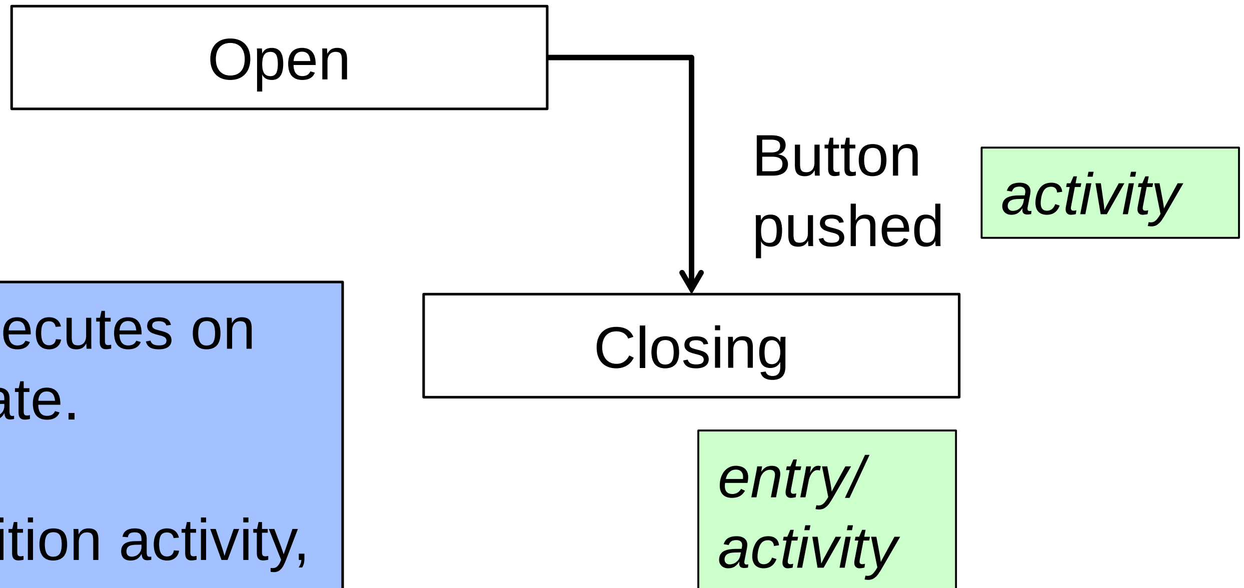
You can associate an activity with a transition.



It completes before any other activity is executed for this state machine.

Activities on Entry

You can associate an activity with entry to a state.

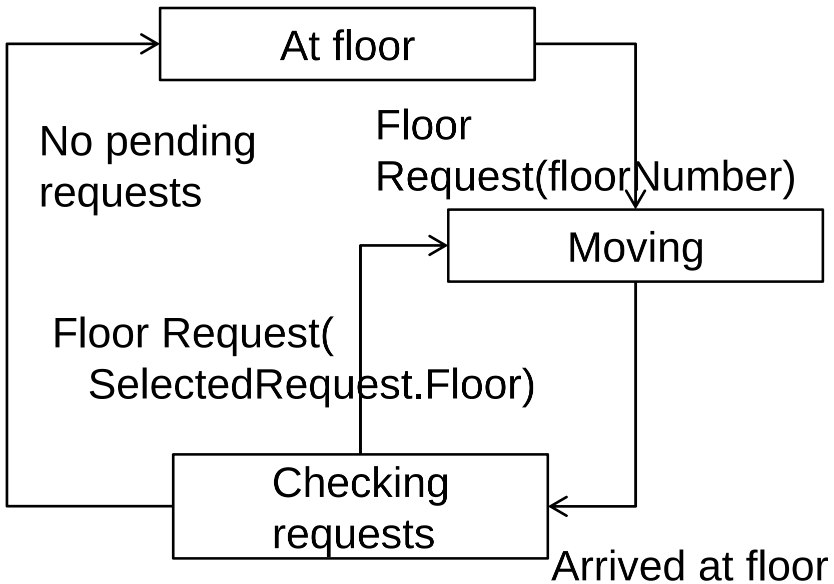


The activity executes on entry to the state.

After the transition activity, if any.

Event Data

Activities are handed *parameters* with the event.



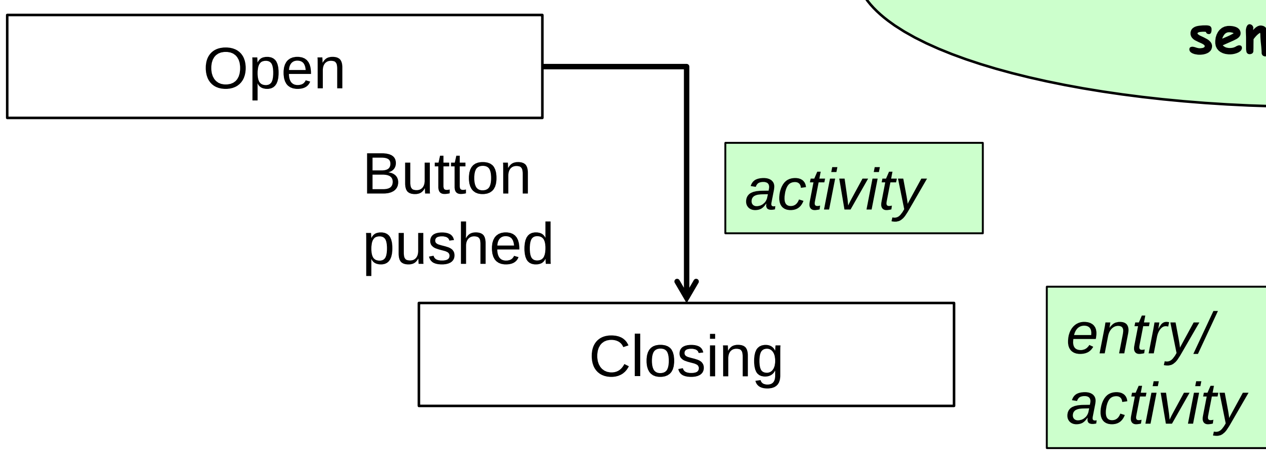
You can think of an activity as a routine with input parameters and side-effects *only*

When the activity terminates, only object data remains.

Execution Sequence

- An activity is executed on the transition
- Another activity is executed on entry to the state

This is commonly called *run-to-completion* semantics



- Both activities must complete before accepting another event
- Both activities must complete before the instance may be considered to be in the next state

Event Dispatch

Event delivery causes one of:

- Transition
- Ignore
- Can't Happen

Transition:

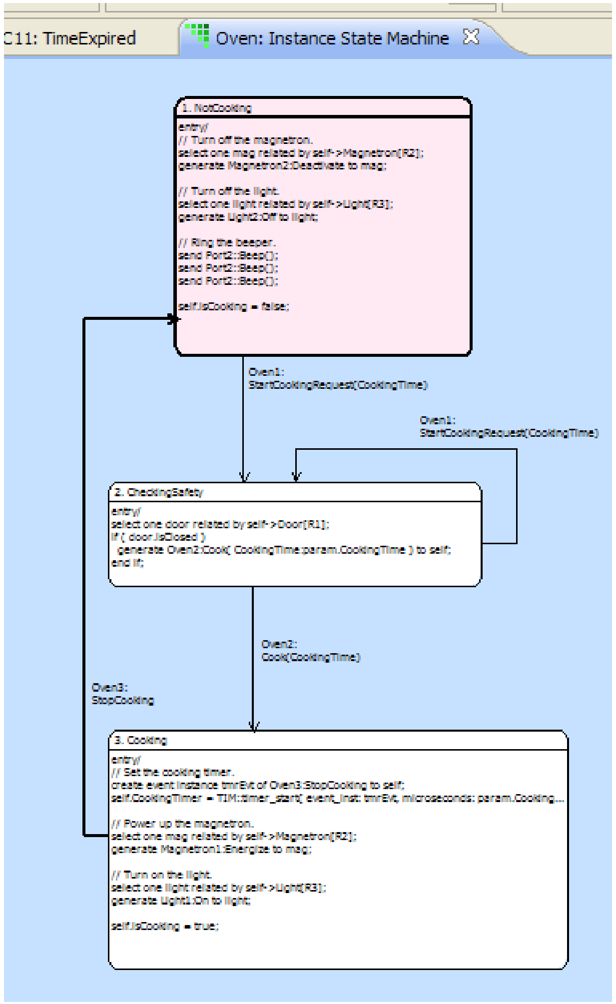
- Execute activity on transition
- Execute activity within state
- Change current state

Ignore:

- Event is discarded,
  no state change, no actions

Can't Happen:

- System-level recovery invoked

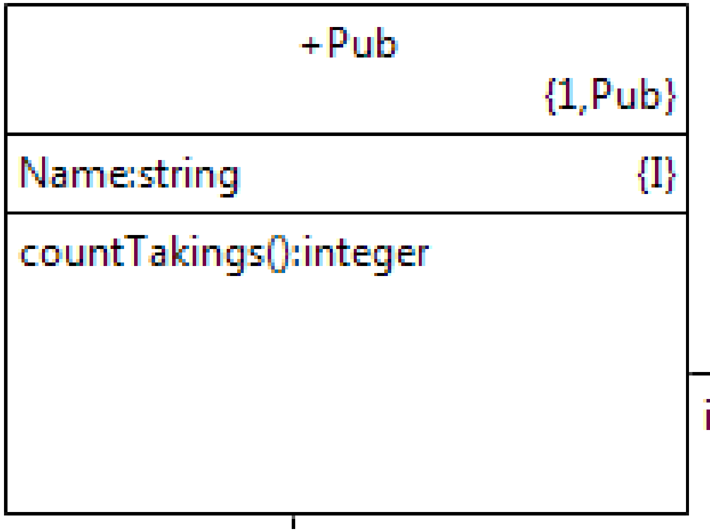


Activities on SEMs

Each cell may contain a reference to the activity to be executed on the transition.

	Player1: LookForTable	Player2: FoundTable	Player3: GameOver
Drinking	LookingForTable	Can't Happen	Can't Happen
LookingForTable	Event Ignored	Playing	Can't Happen
Playing	Can't Happen	Can't Happen	Drinking

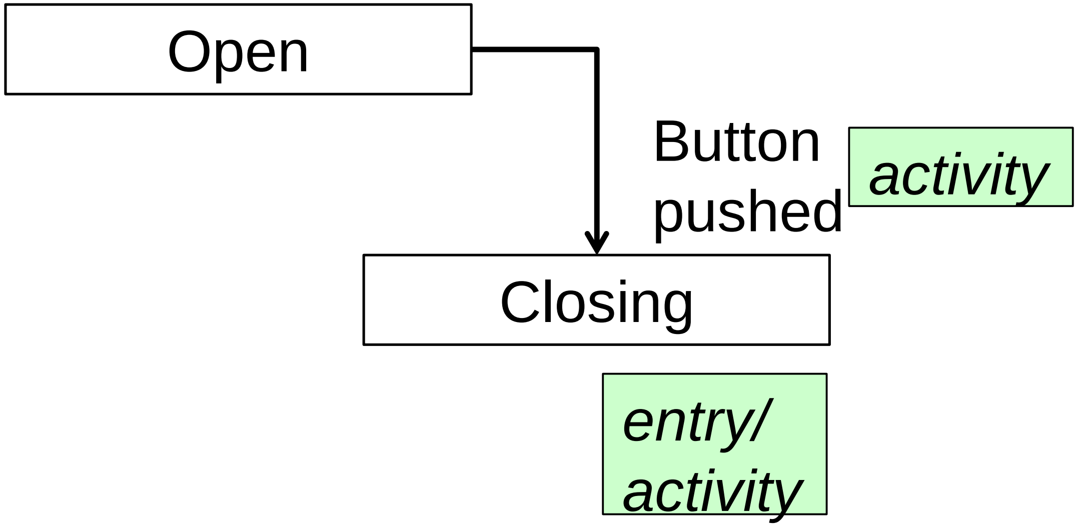
Activities



Activities can be placed anywhere:

- on transitions
- on (entry to) states
- on operations of classes

We describe the activity using an *action language*.



Workshop

Construct a state model for each class in the GPS Watch that has a lifecycle.

Indicate the location of activities.

Describe each activity in natural language.

8. Action Language

8

Object Action Language [OAL]

Object Action Language is a concrete syntax that implements the UML standard.

OAL is complete enough to be executable, but abstract enough that it does not prescribe implementation specifics.

```
create object instance request of REQ;

select one channel related by device->CHAN[R100];

device.priority = lastpriority + 1;

generate CHAN11:'host relinquish' to channel;
```

OAL Does…

Object Action Language can:

- Create and delete instances
- Read and write attribute values
- Compute new values
- Generate events
- Link and unlink associations between instances
- Select instances across associations
- Find instances based on attribute values
- Communicate with the outside world

And control when these actions take place.

Data Types

Core primitive data types

- boolean
- integer
- real
- string
- unique_id

> All data items are implicitly typed by the value assigned to them on their first use within an action.

Reference data types

- instance handle
- instance handle set
- event instance
- component instance handle

Built-in user-defined types

- date
- timestamp
- timer handle

Operators

In addition to the usual operators:

- empty [<instance handle> | <instance handle set>]
- cardinality [<instance handle> | <instance handle set>]
- not <boolean>

Loops

Use foreach to iterate over a collection.

while loops

- can be nested.
- define a local scope.

```
for each mobile in mobiles
      // do something
end for;
```

```
i = 0;
while (i < 4)
      // do something
      i = i + 1;
end while;
```

Parameters

- Event and operations carry parameters
- Parameters are tagged, not positional.
  - param is a pre-pended keyword to access arguments

```
select any probe from instances of SP where
     selected.probe_ID == param.probe_id;
trackPoint.latitude  = param.latitude;
```

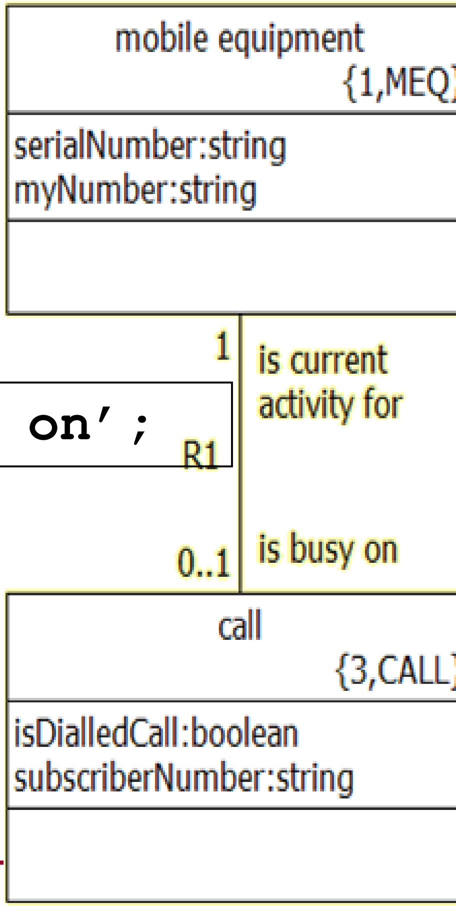
Relate / Unrelate Statement

Link specific instances of classes using `relate`.

```
relate mobile to call across R1.'is busy on';
```

Local instance reference variable

Association ID




```
unrelate mobile from call across R1.'is busy on';
```

Local instance reference variable

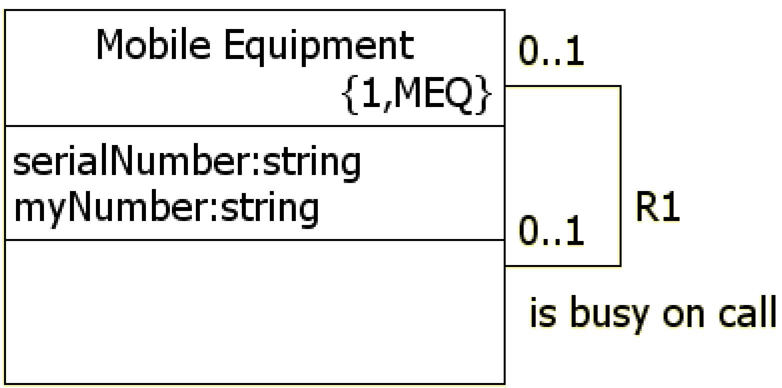
Select Any / Many

```
select any mobile from instances of MEQ;
```

Local instance reference variable — Key letters

```
select many mobiles from instances of MEQ
      where selected.serialNumber > 10000;
```

Where clause



Select One / Many … Related By

- Select one requires the use of the related by clause
- `Self` is the instance of the class that originates an action

```
select one timer related by self->
WorkoutTimer[R4.is timed by'];
```

Local instance reference variable: `timer`

Originating class instance: `self`

Key letters: `WorkoutTimer`

Association phrase: `[R4.is timed by']`

Workshop

Write OAL for the activity that completes the goal, specifically:

- Move the just-completed goal from ‘Currently executing’ (R11) to ‘Executed’ (R12)
- Create a new goal based on the next one in sequence
- Associate the newly created goal with the currently executing goal

9. Distribution of Intelligence

9

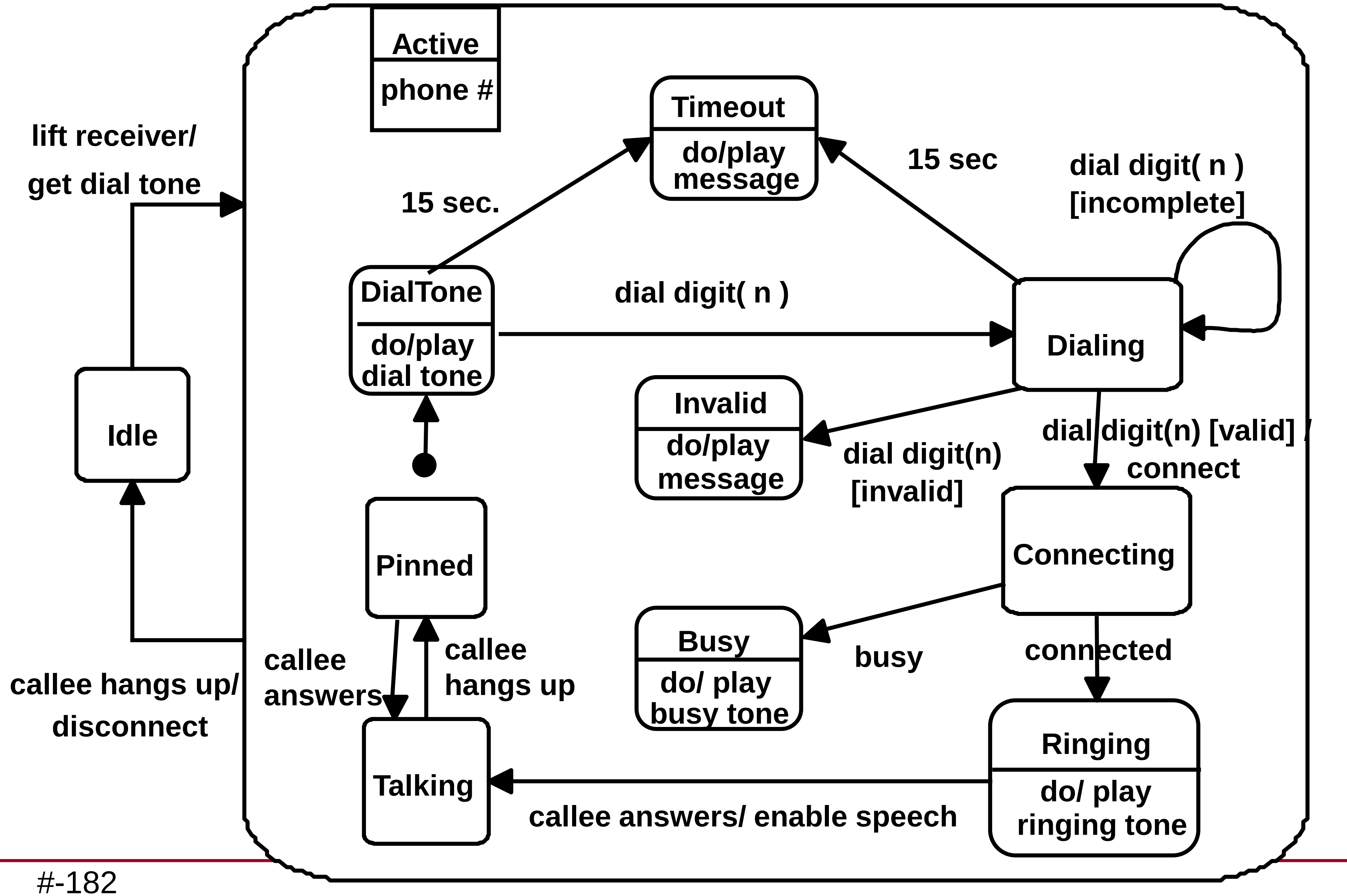
Tangible Things

Tangible things rarely have interesting lifecycles.

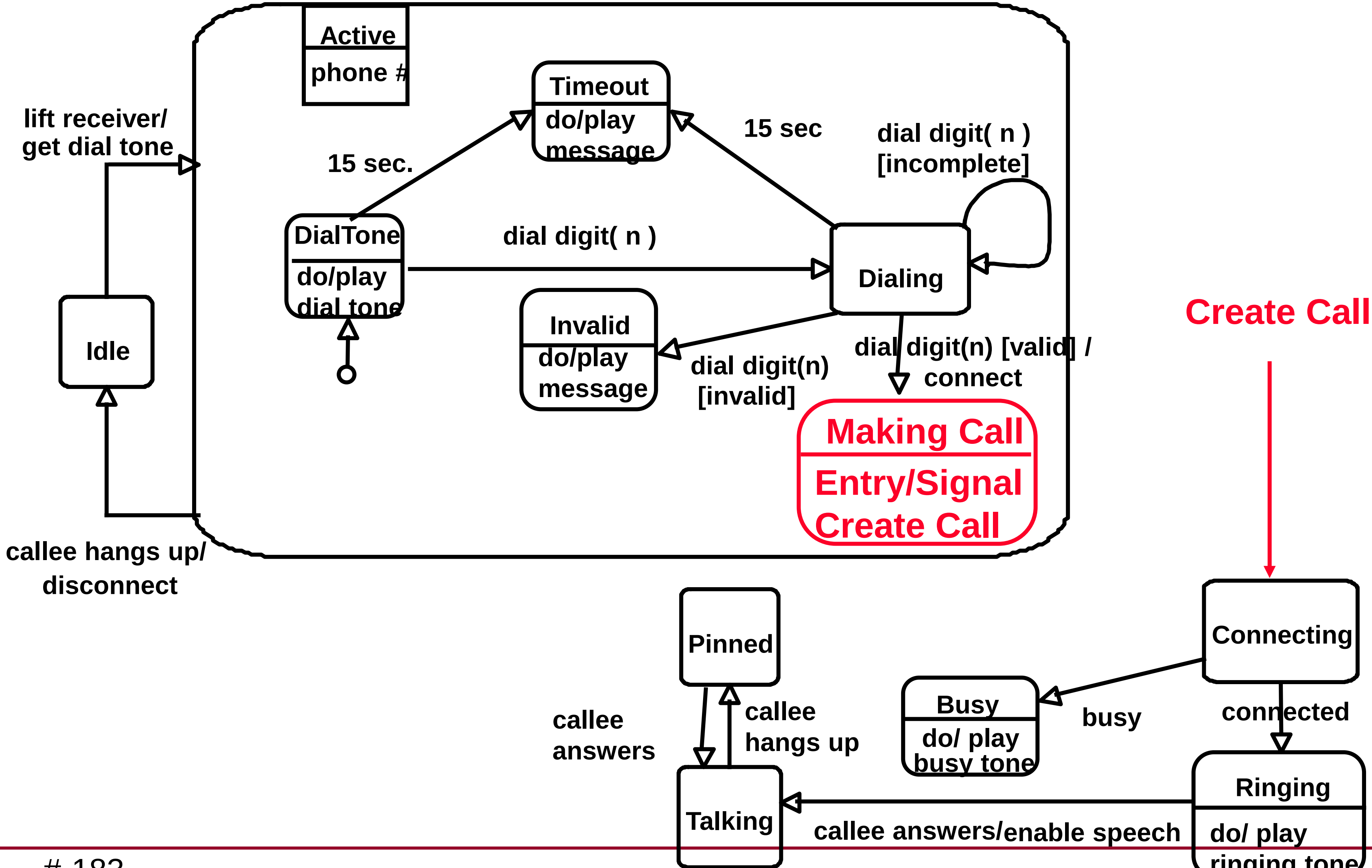
They are *driven* by classes that capture behavior.

You must distribute intelligence among the classes.

Complex State Model



Simpler Communicating State Models



Patterns

There are two control patterns that occur frequently:

- Top-driven: where a user/operator drives behavior
- Bottom-driven: where a device/hardware drives behavior

And two patterns based on factoring data:

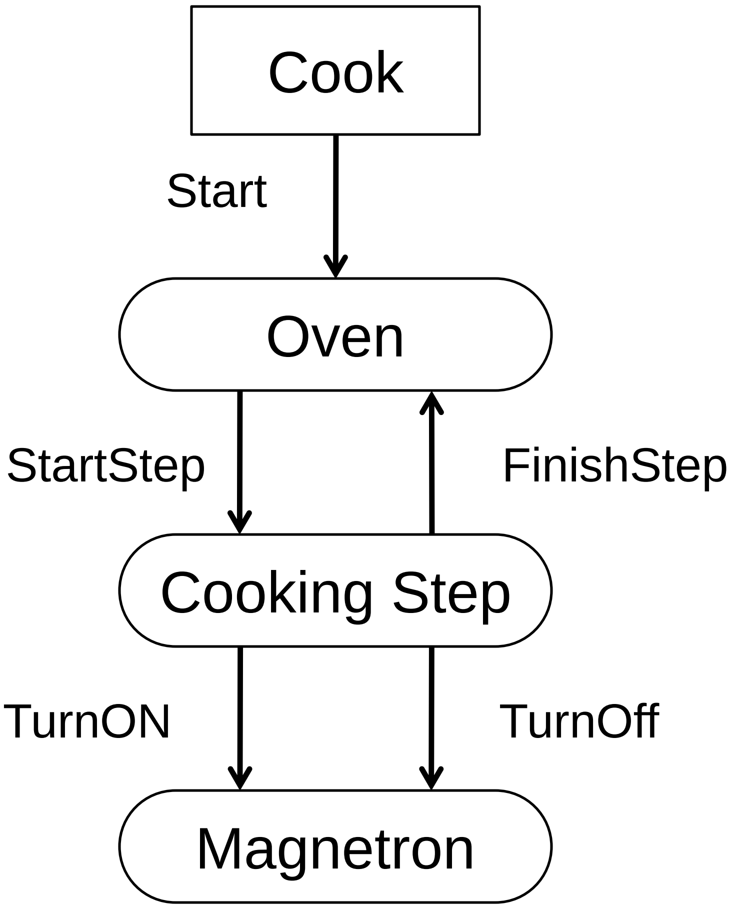
- Push-and-pull: Data is pushed in, and pulled out
- Pivot: The pivot is the place where
  the data comes “to rest”

Top-Driven

In a top-driven pattern, a user/operator drives behavior.

Examples:

- Microwave oven
- Chemical plant operations
- Phone calls

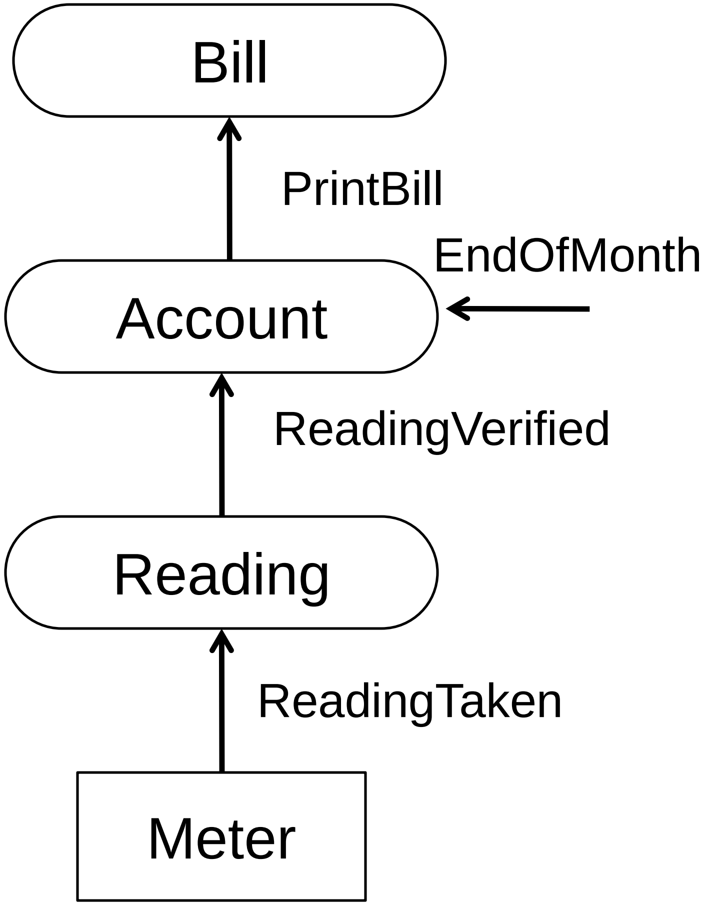


Bottom-Driven

In a bottom-driven pattern, a device/hardware drives behavior.

Examples:

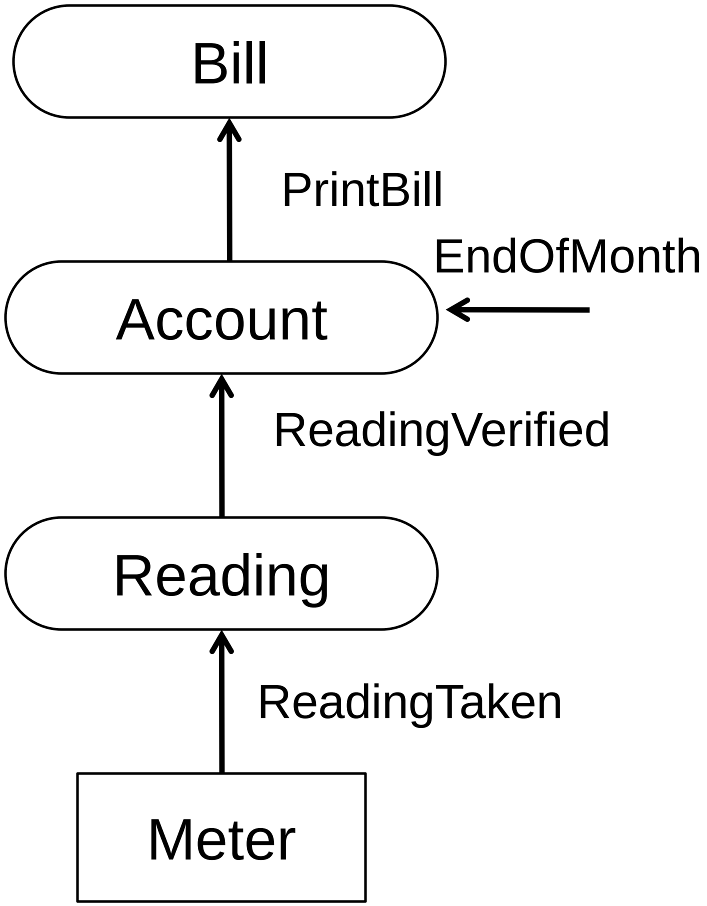
- Meter Reading
- Alarm System
- Satellite



Push-and-Pull

In push-and-pull, data is

- pushed so far, then
- rests, then is
- pulled the rest of the way

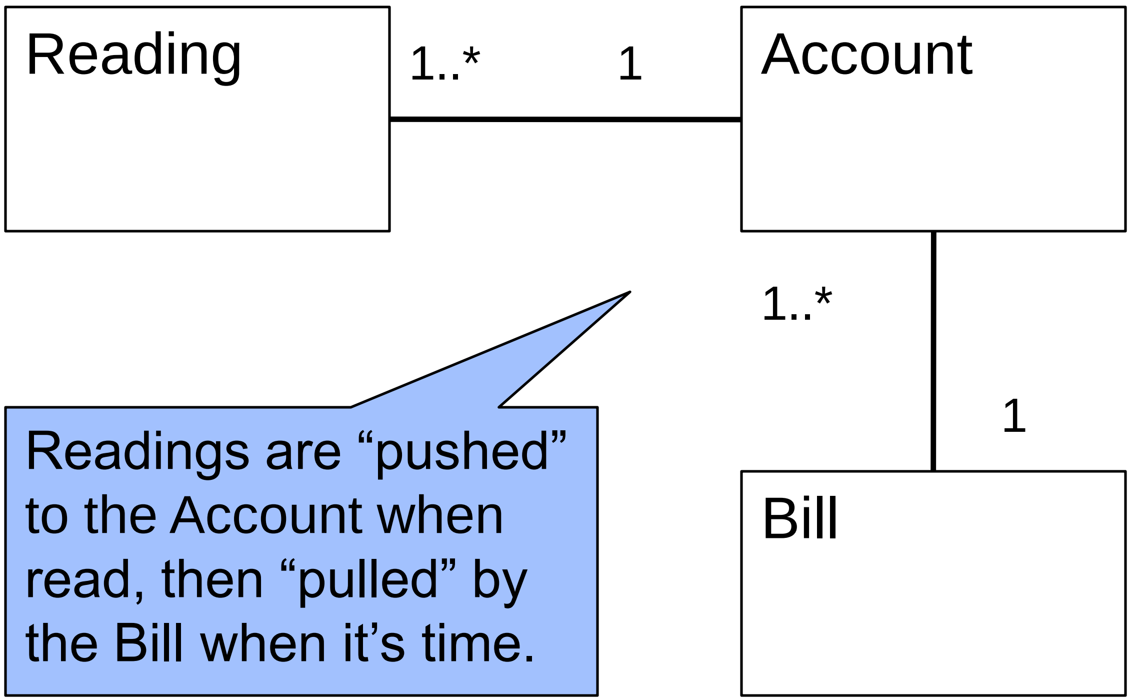


Examples:

- Meter Reading
- Order fulfillment
- Message accumulation

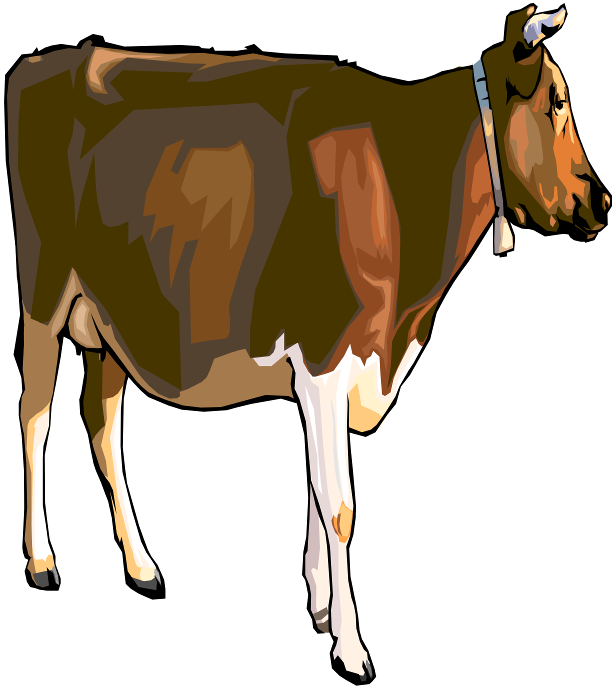
Pivot

The trick with push-and-pull is to find the pivot.



Associations

Associations often carry interesting behavior.



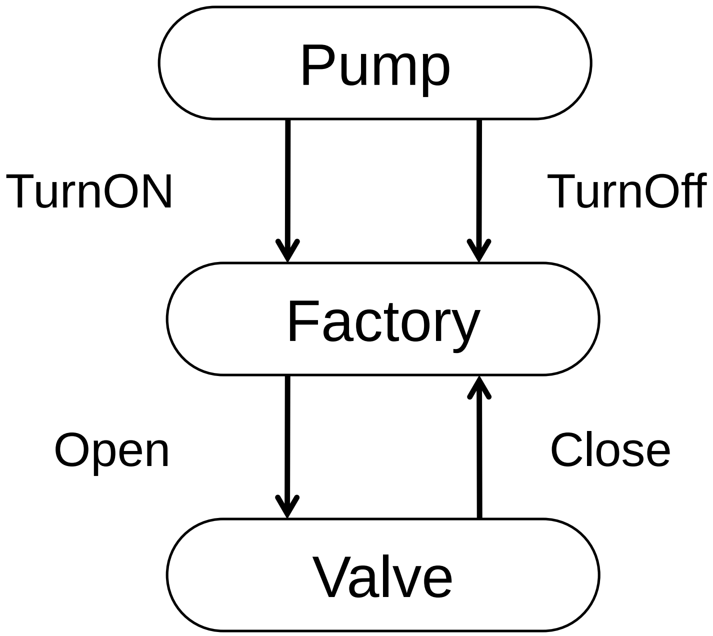
<u>Milking</u>

- Cow ID
- Urn ID
- Time

Does the cow demilk itself? Or the milk uncow itself? Neither!

Anti-pattern

Avoid controller/manager state models that control everything.



Completeness

If you followed the process, your state models are complete.

Check the model for completeness anyway.

- Does every event have (a) source(s)?
- Does every event have (a) destination(s)?
- Does each state model have all the events it needs?

Workshop

Based on what you just learned, review and revise your state models to improve the distribution of intelligence.

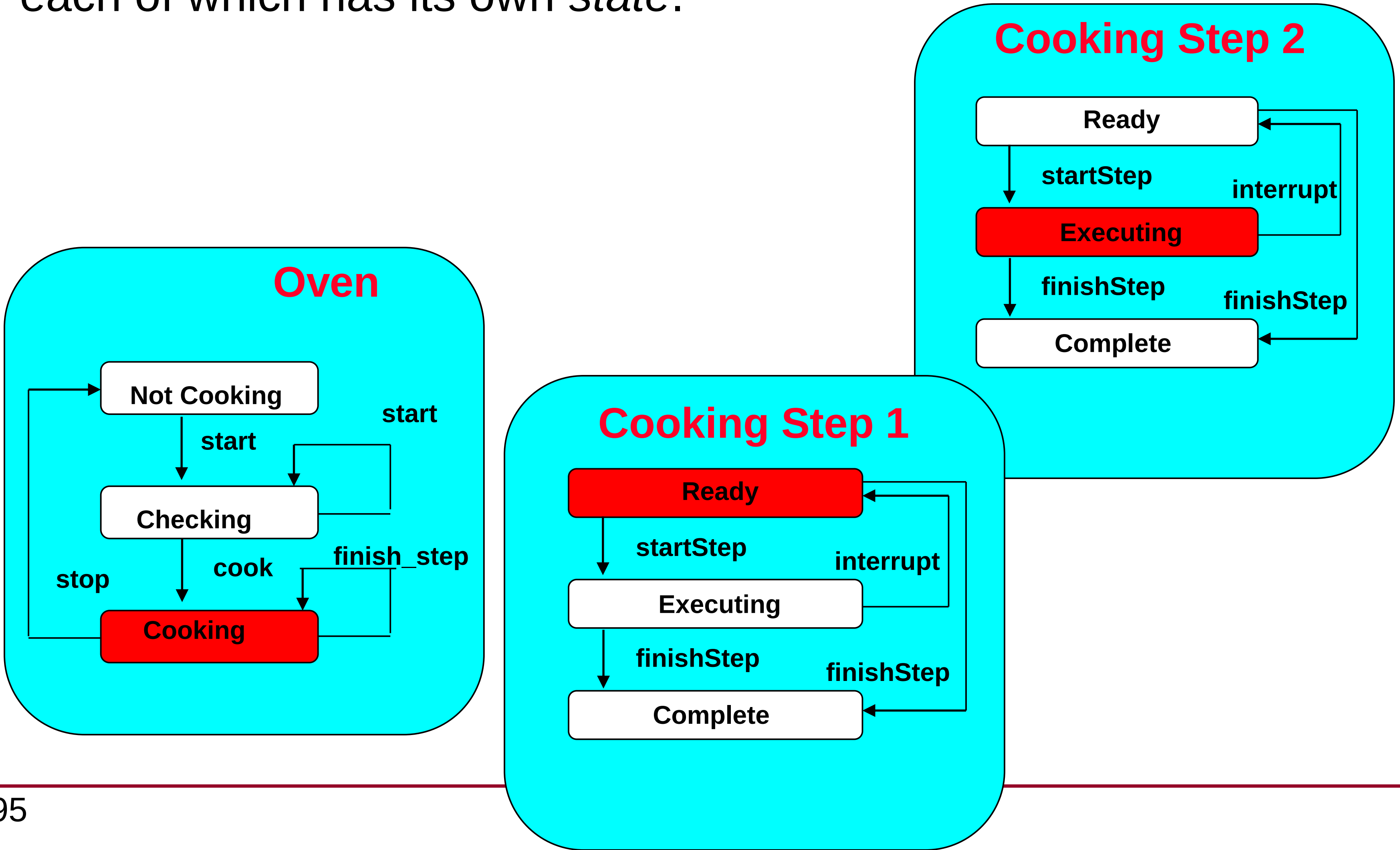
Be prepared to describe your approach for distribution intelligence to the class.

10. Model Execution

10

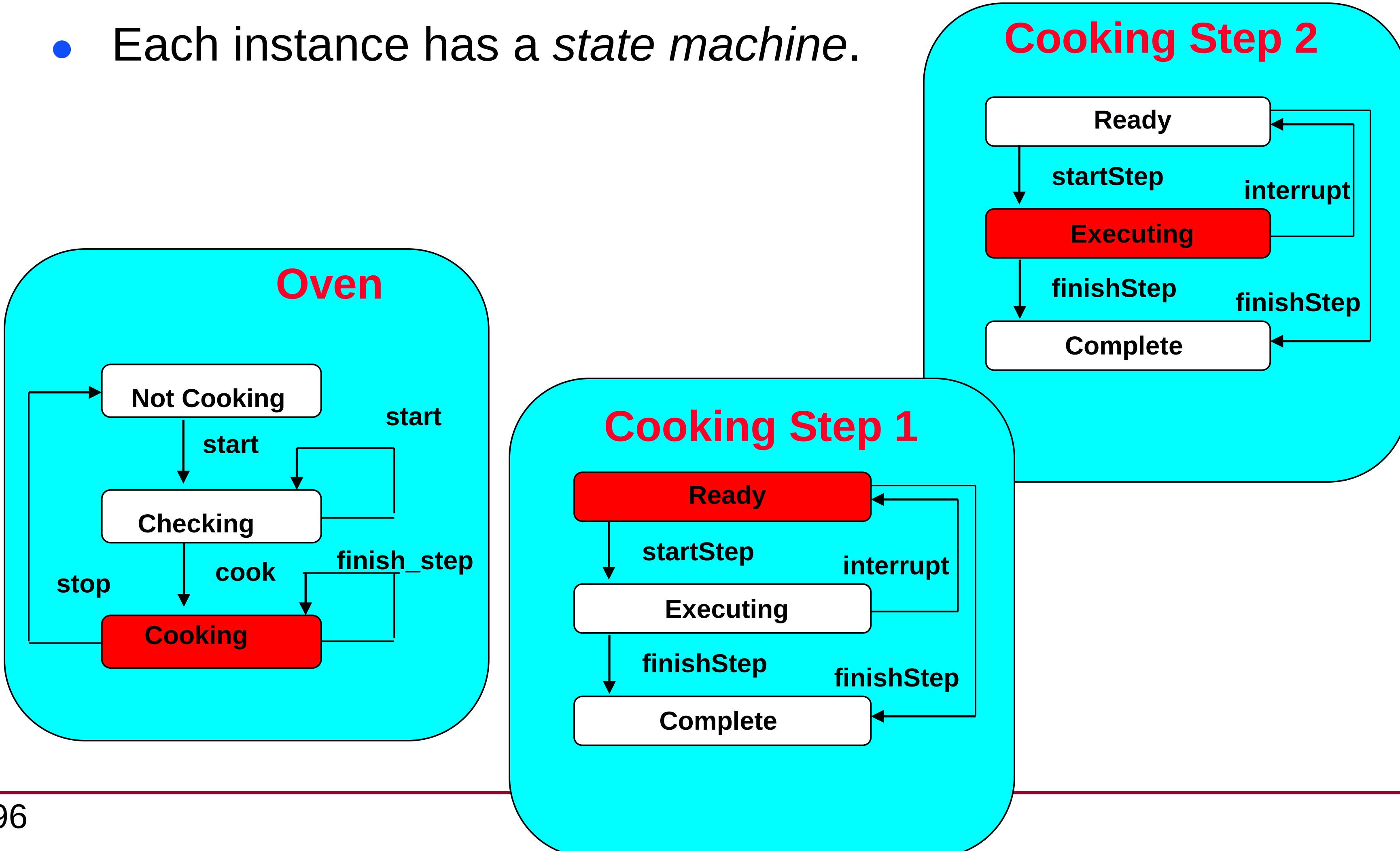
State Machines

A *state machine* is a copy of a *state model* for each instance, each of which has its own *state*.



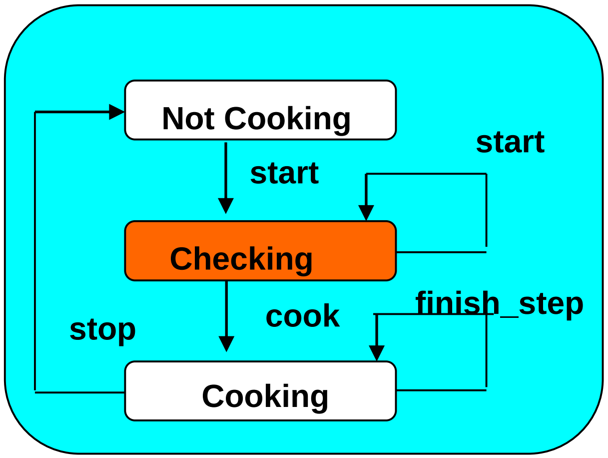
State Machines

- Each class has a *state model*.
- Each instance has a *state machine*.



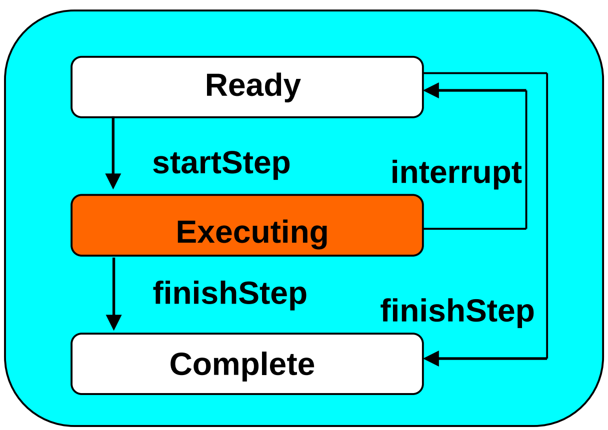
Concurrent Execution

Oven

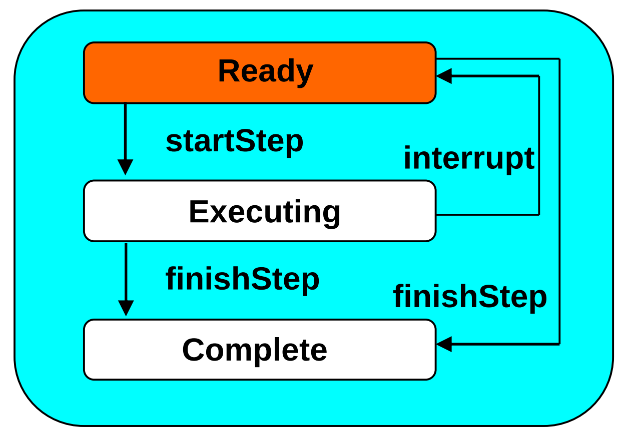


- All instances execute concurrently.

Cooking Step 1



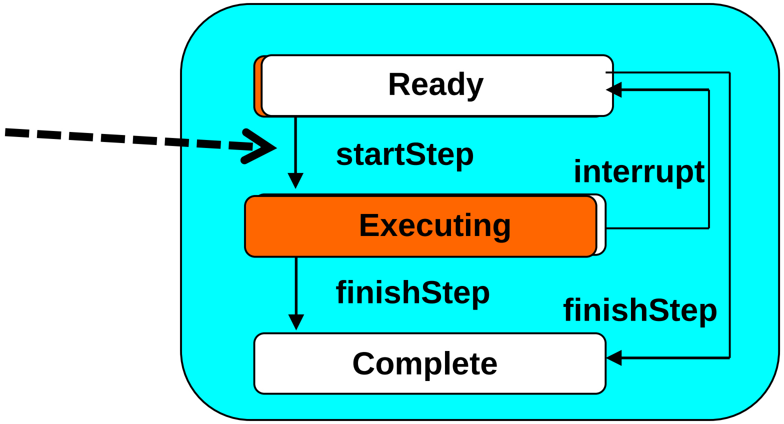
Cooking Step 2



Executing the Model

The model executes in response to events from:

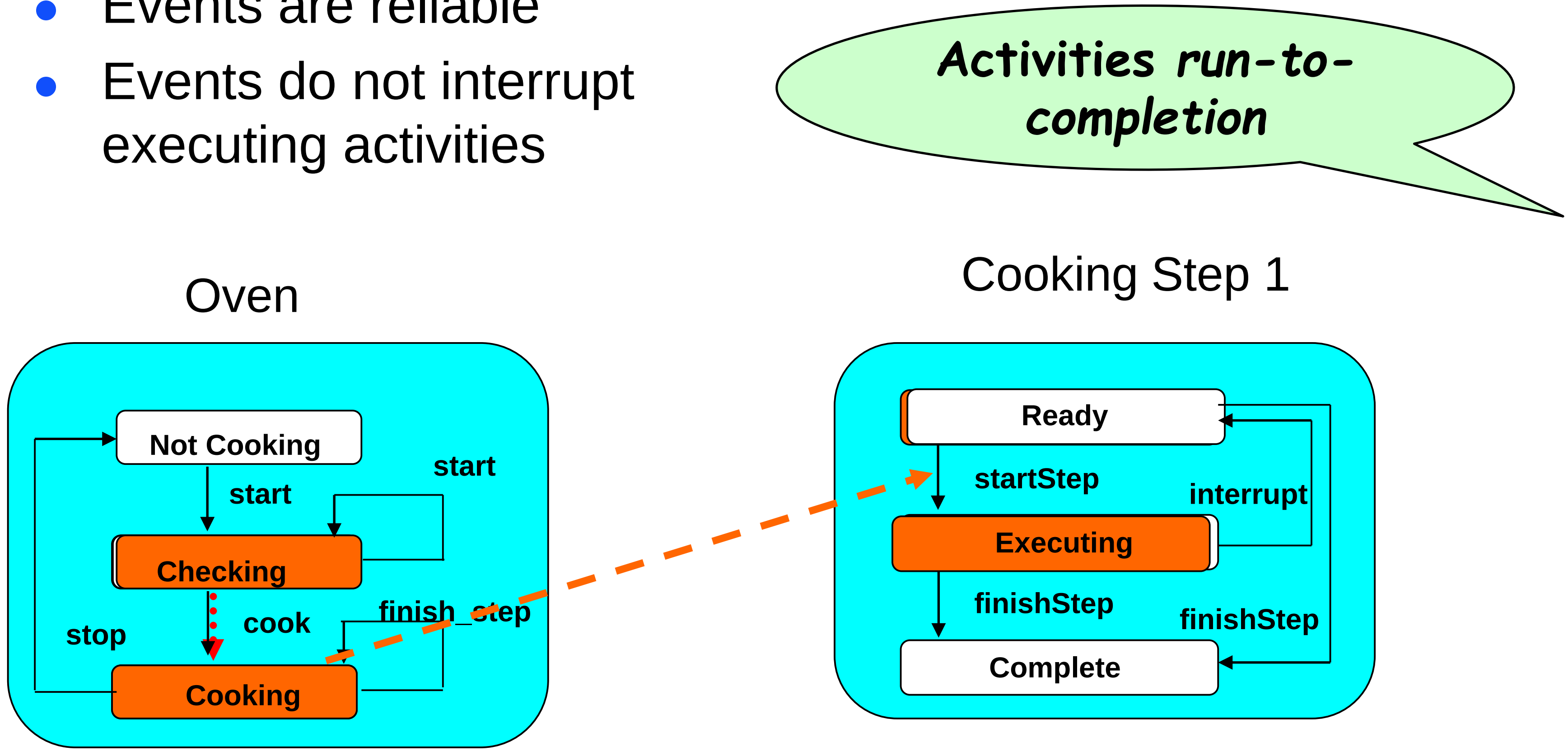
- the outside,
- timers
- other instances as they execute



Communication

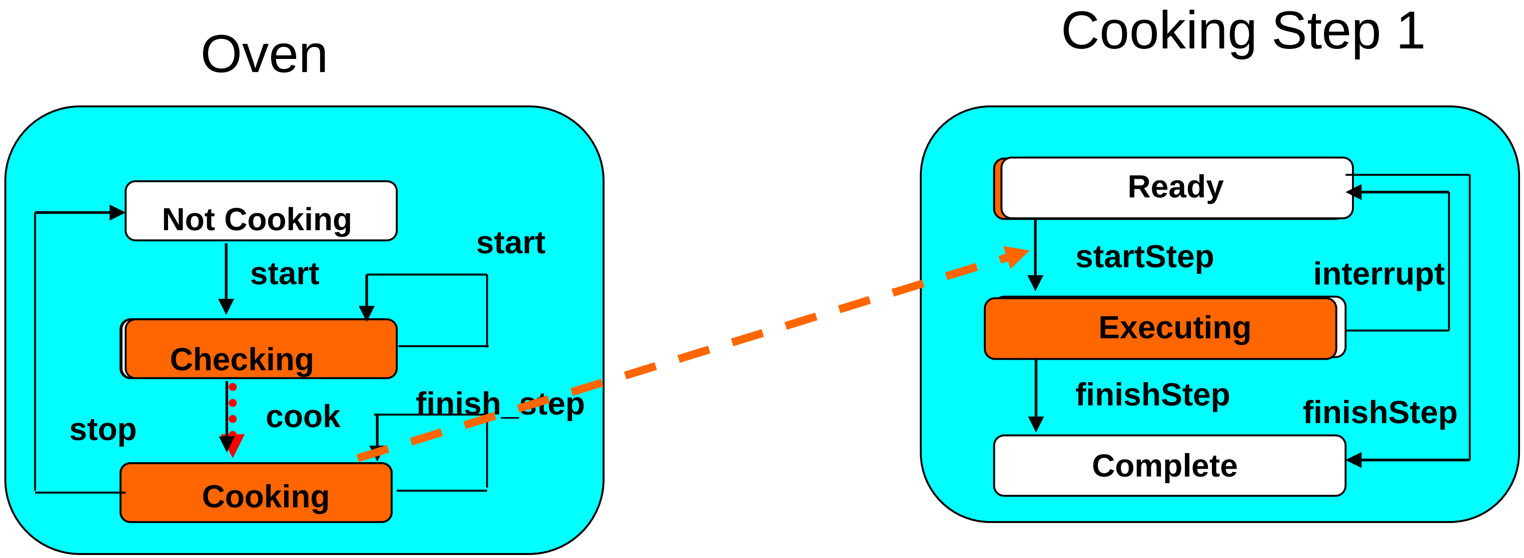
State machines drive each other through their lifecycles by sending each other events.

- Events are reliable
- Events do not interrupt executing activities



Communication

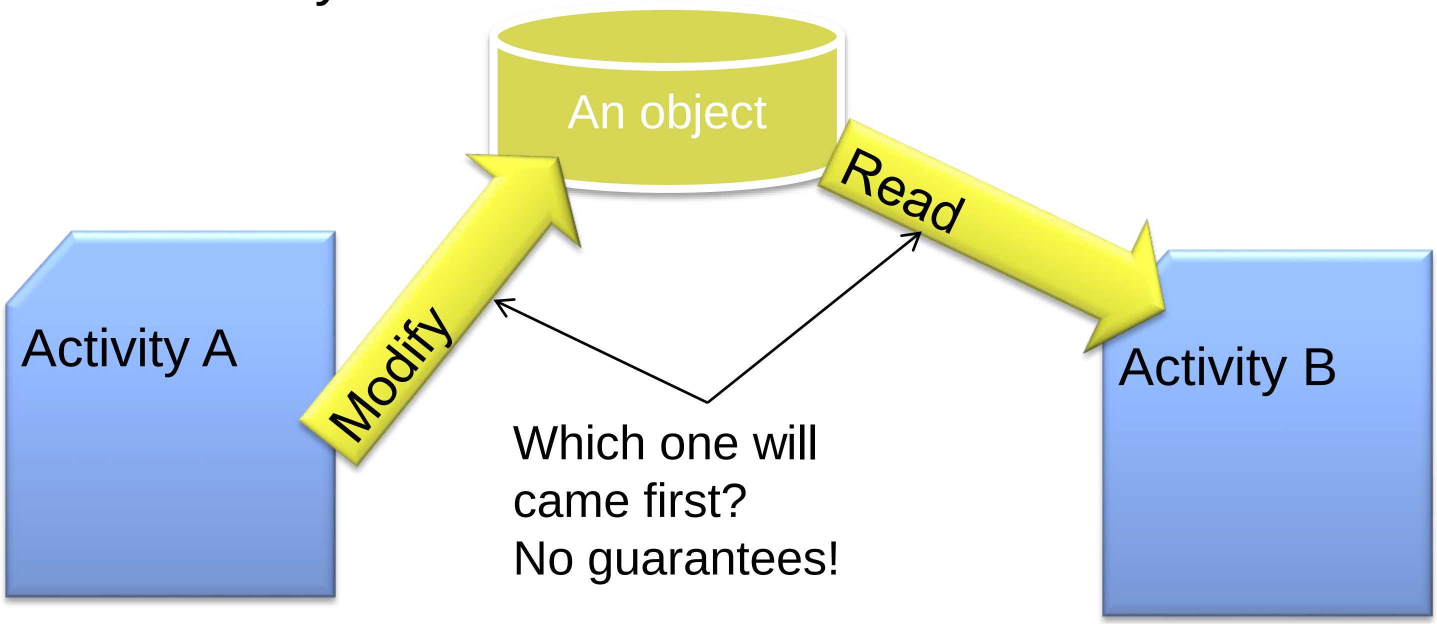
1. Start in Checking and Ready states
2. Accept event ‘Cook’
3. Change to Cooking State
4. Generate ‘startStep’ signal
5. Change to Executing state



Run-to-Completion ≠ Atomic!

Other state machines (and their activities) run concurrently

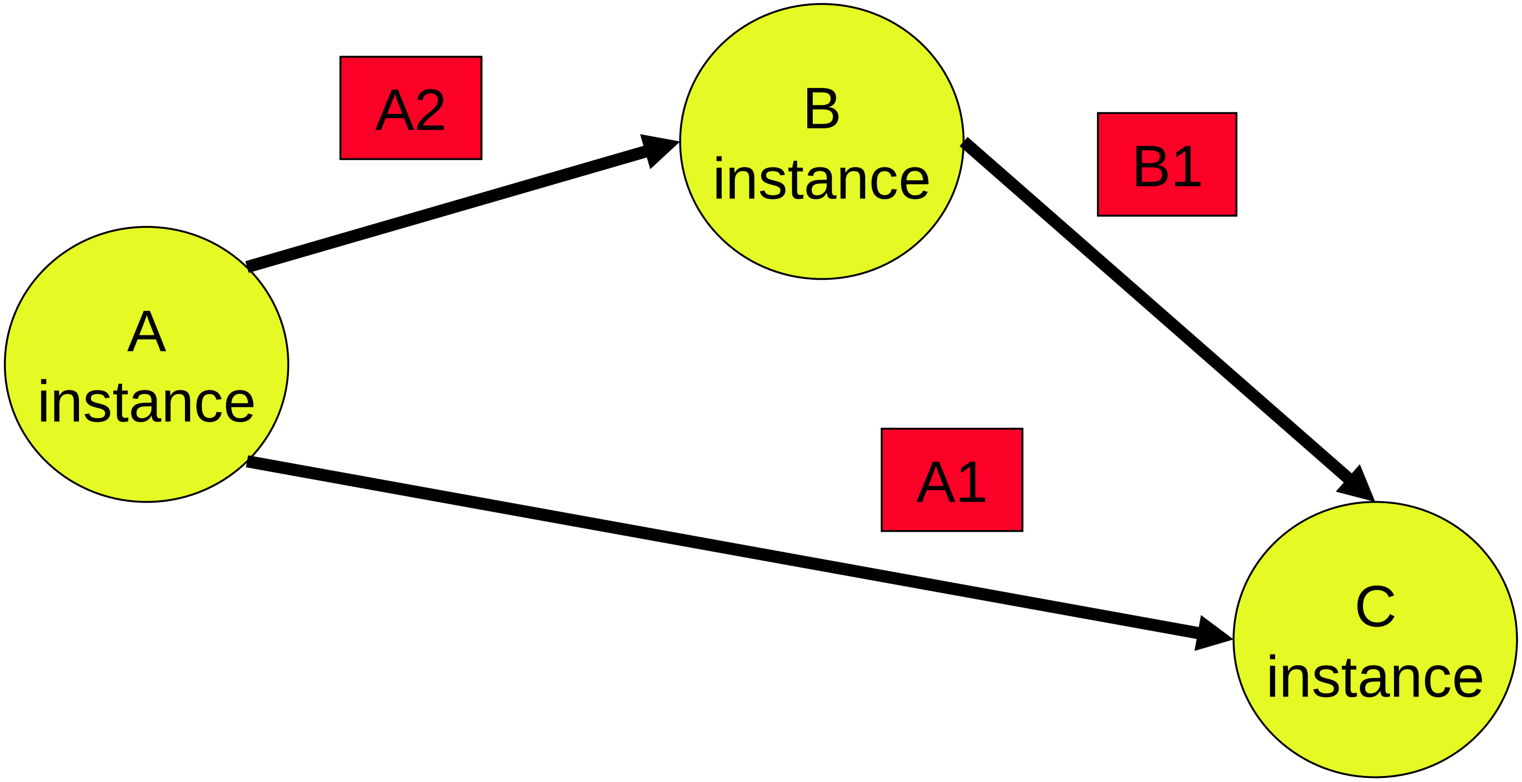
- An activity can be pre-empted during execution
- One state machine may change the data accessed by another



Or, you can set a global switch to prevent activities from pre-empting one another.

Synchronization

State machine instances are coordinated by sending signals.



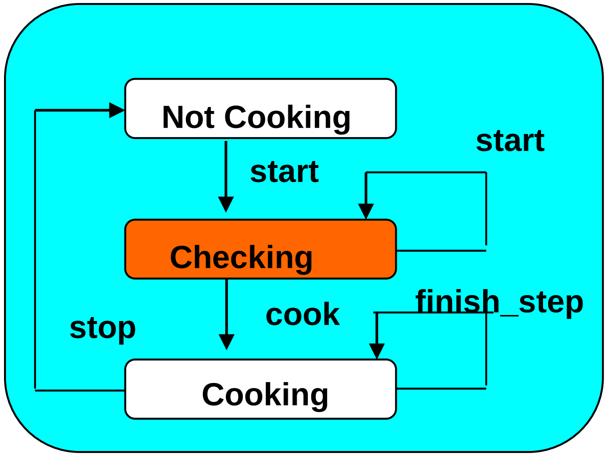
The order of arrival of A1 and B1 at C is indeterminate, even if A1 was generated first.

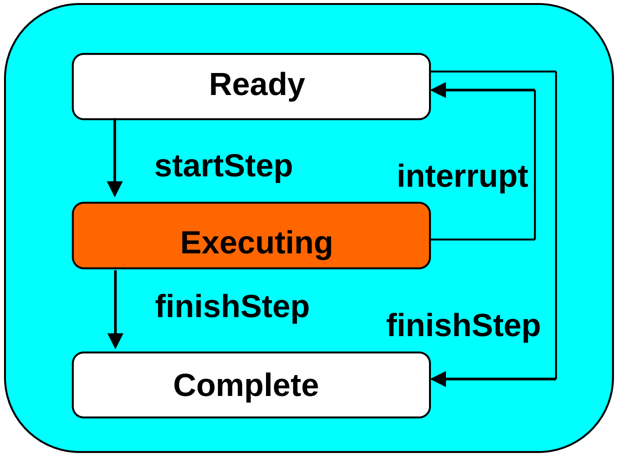
Time

- Time is relative to each observer

Oven

Cooking Step 1





Timers

A *timer* can generate an event.

- With a delay (e.g. in 10 seconds)
  - The delay specified is a minimum
- You may cancel a timer
  - But the event may already be "in flight"
  - You have to account for that case.

Summary

- Build models relying only on the execution rules of xtUML
- Build (or buy) a model compiler that implements these rules for your target
- Understand the trade-off between model portability and exploitation of platform-specific characteristics
  - Make deliberate, explicit decisions and document them clearly

Workshop

Get Pub state machines from your instructors.

Label instances of Patron, “Hugo”, “Debbie”, “Tiny”, “Zoltan”.

Place coins on the initial states of each state machine:

- Patron Hugo: Outside
- Patron Debbie: Drinking
- Patron Tiny: Drinking
- Patron Zoltan: Needs Drink

- Snooker Table 1: Available

and walk through their lifecycles as shown on the next page.

Workshop

Inject the following events and change states appropriately:

Patron 1: Thirsty to Patron Tiny

Patron 1: Thirsty to Patron Hugo

Patron 2: Served to Patron Zoltan

Patron 3: Bored to Patron Zoltan

Patron 2: Served to Patron Tiny

Player 1: Look for Table to Player Zoltan

Patron 2: Served to Patron Hugo

Patron 3: Bored to Patron Debbie

Patron 1: Thirsty to Patron Tiny

Player 1: Look for Table to Patron Debbie

Player 2: Found Table to Patron Zoltan

Player 2: Found Table to Patron Debbie

Patron 2: Served to Patron Tiny

Patron 1: Thirsty to Patron Tiny

Patron 2: Served to Patron Tiny

SnookerTable2:LastBallPlayed to Table 1

Patron 1: Thirsty to Patron Zoltan

Patron 5: Sated to Patron Debbie

Patron 1: Thirsty to Patron Tiny

Workshop

Expected Post-conditions:

- Patron Hugo: Drinking
- Patron Debbie: Outside
- Patron Tiny: NeedsDrink
- Patron Zoltan: NeedsDrink
- Snooker Table 1: Available

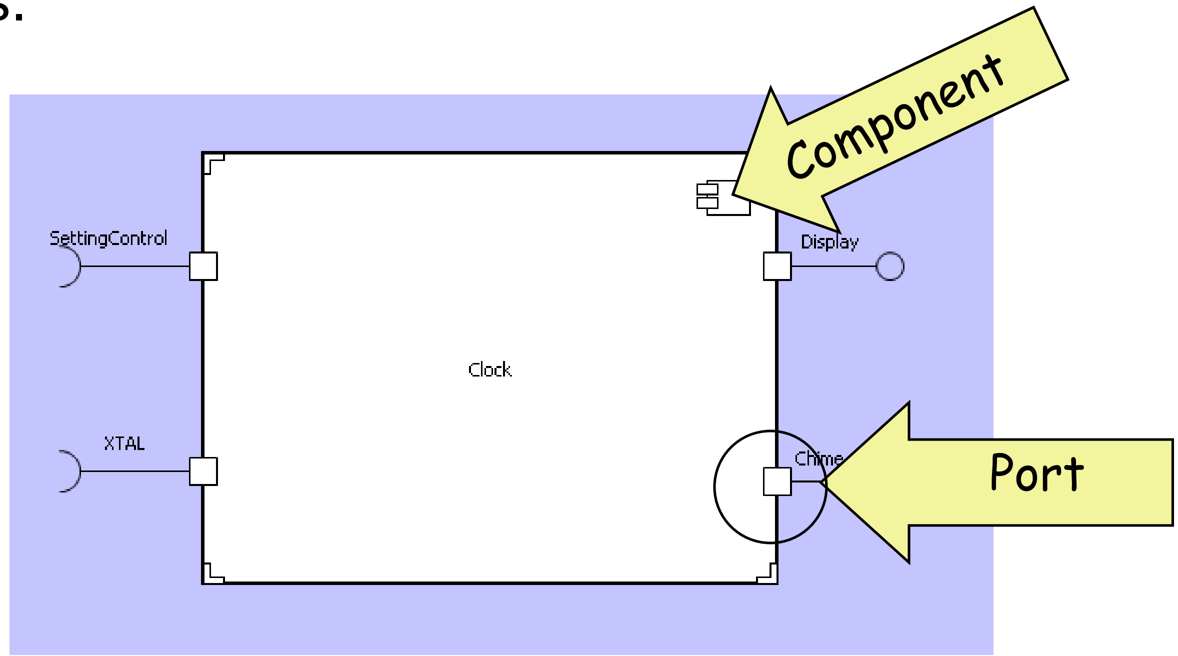
11. Components and Interfaces

11

Components

A *component* is a part of a system that hides its implementation behind ports.

The inside of a component can “see” the outside only through its ports.



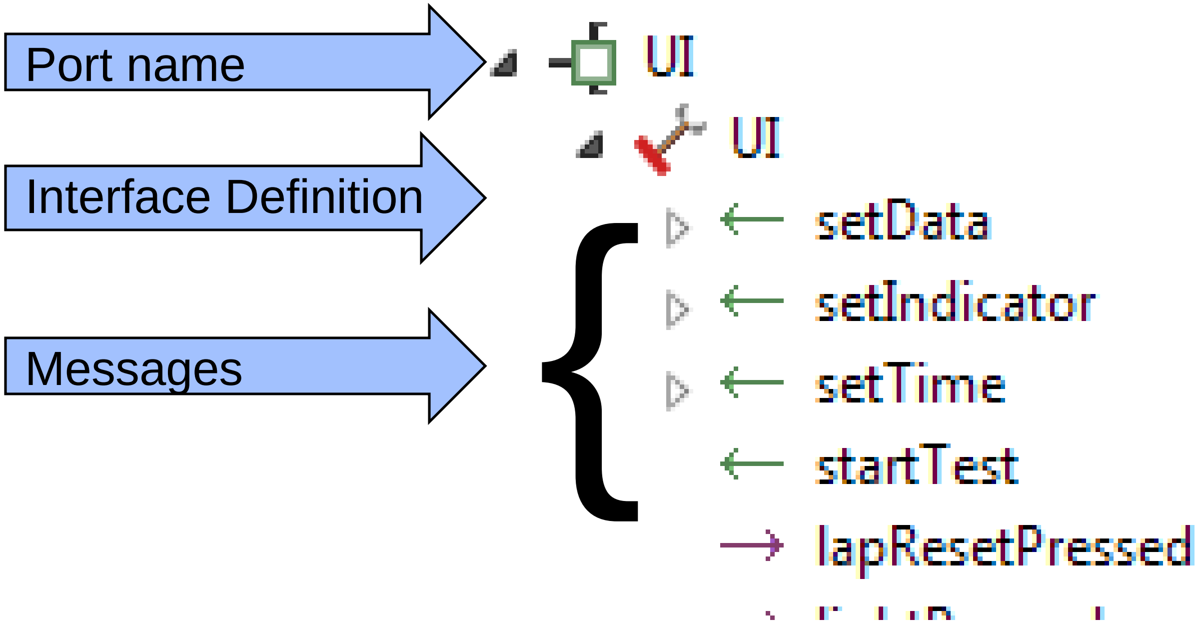
Components and ports are named.

Ports

Each port …

… surfaces an interface …

… by referencing an interface definition.



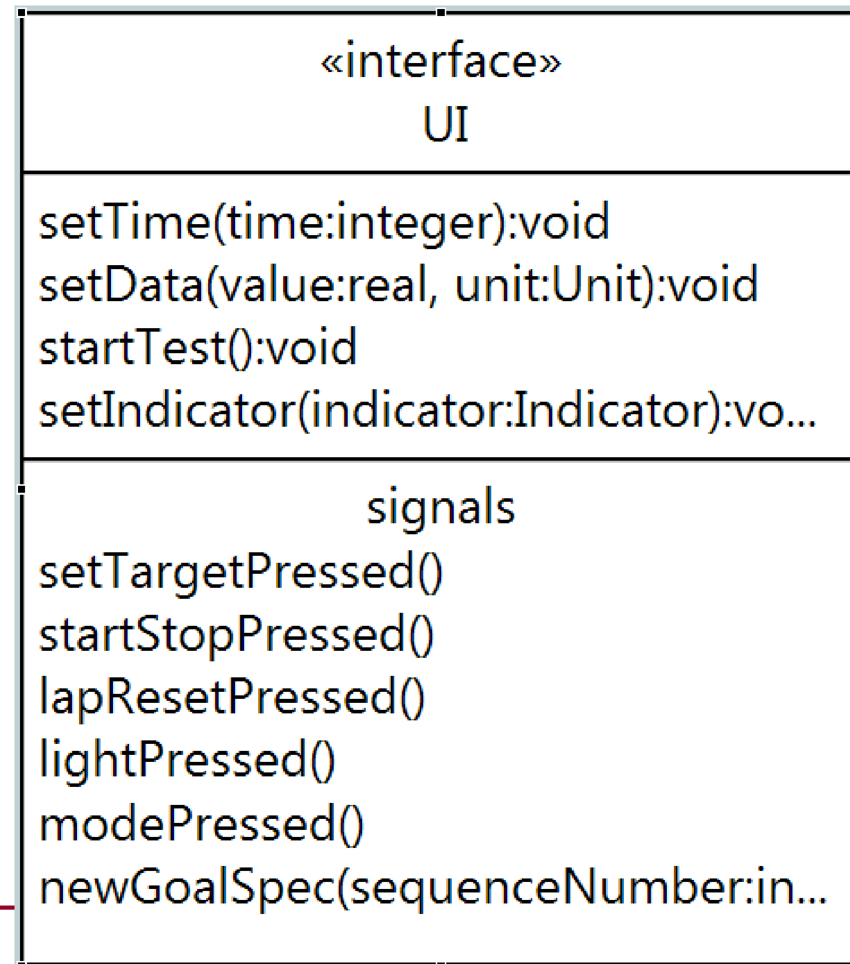
Actions within a component specify through which port any outgoing messages should be sent.

Interfaces

An interface defines of collection of messages, similar to a declaration in a programming language.

Each interface is defined once, and may be used multiple times.

Each message can carry typed parameters.



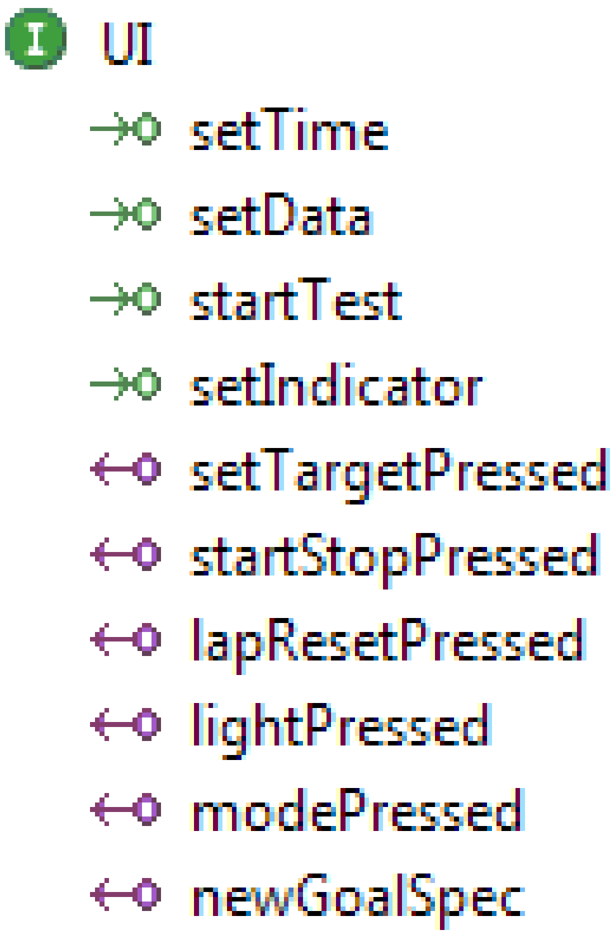
Messages

Messages have a direction

(relative to/from the provider).

The ball indicates the provider.

The arrow indicates the direction of the message in an interface definition.



Port Activity

Each message (in the interface, connected to a port) may have a *port activity* executed when the message is received.

It may contain any actions valid within the context of the receiving component. Best to keep port activities simple:

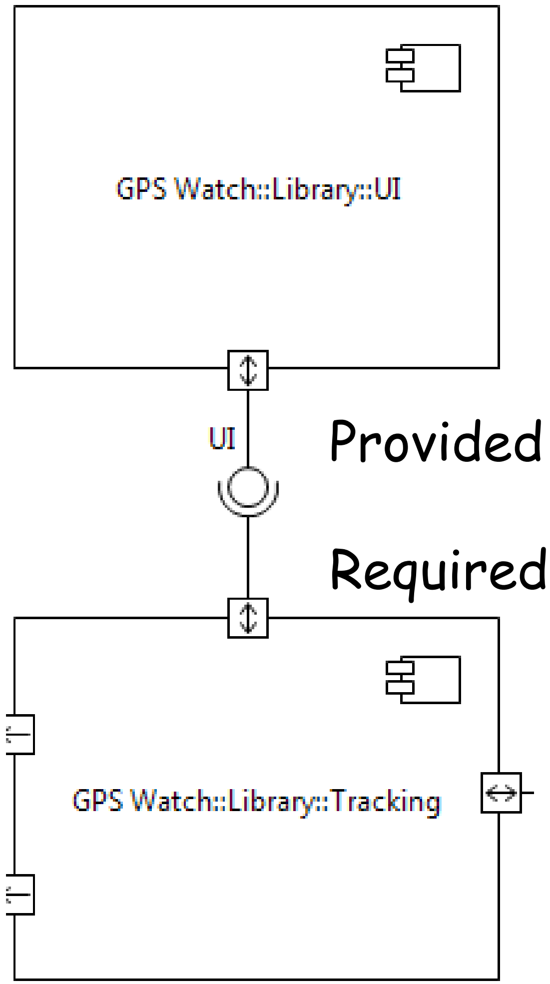
- invoke an operation
- generate an event to an instance

```
// If necessary, create a workout session and everything required to
// support it.  Then, forward this signal to the workout timer.

WorkoutSession::create();

// Forward this signal, as an event, to the singleton instance of WorkoutTimer.
select any workoutTimer from instances of WorkoutTimer;
generate WorkoutTimer1:startStopPressed() to workoutTimer;
```

Provided vs. Required Interfaces

A provided Interface allows a component to provide services to other components.

A required Interface allows a component to demand services from another component.



Workshop

Draw the Component Diagram for this system.

Define all the interfaces.

Write at least one port activity.

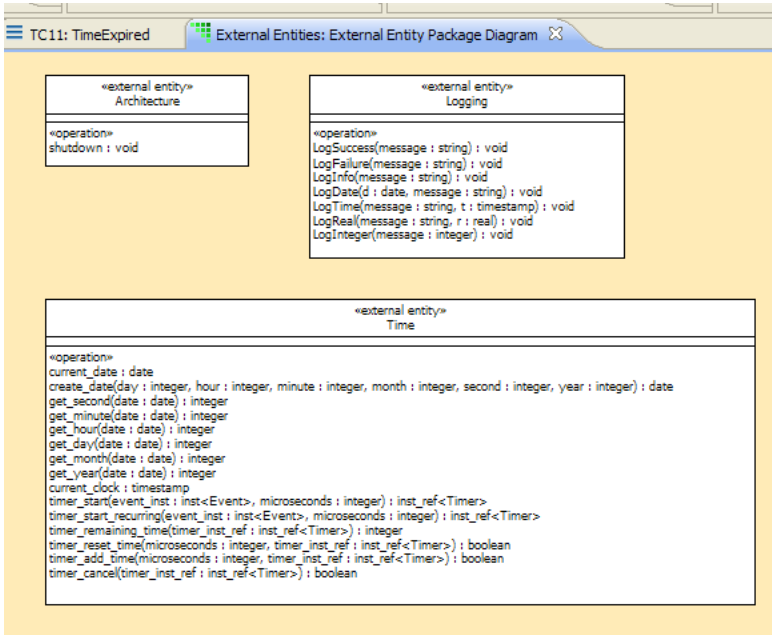
Bridges

Another form of synchronous operation is a *bridge*. It:

- Takes parameters
- Can be wired to external code or defined with OAL

It is used for library functions

- Time
- Logging
- Math



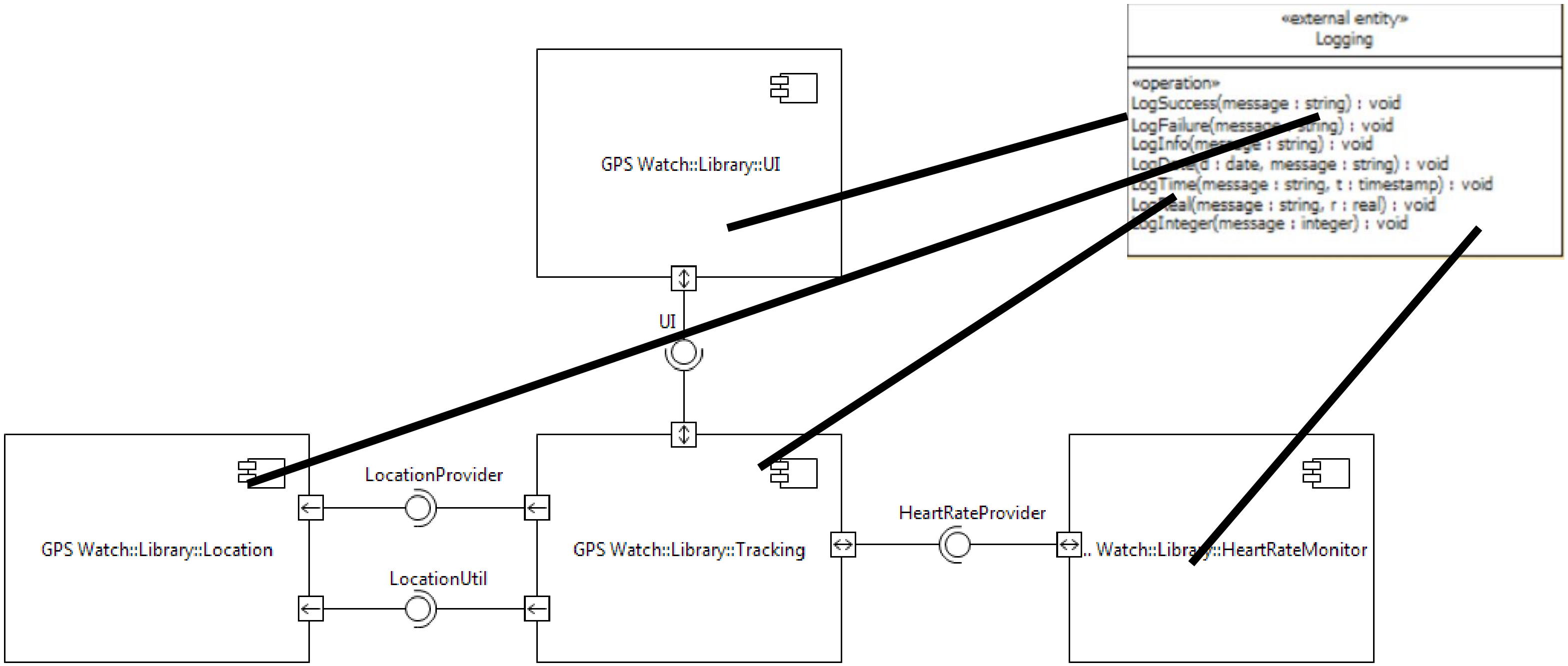
And for scaffolding

- OAL or Java for Verifier
- Hand-written code for target

Ports vs Bridges

Favor Components and ports except…

- When surfacing connections between elements is unnecessary or unhelpful



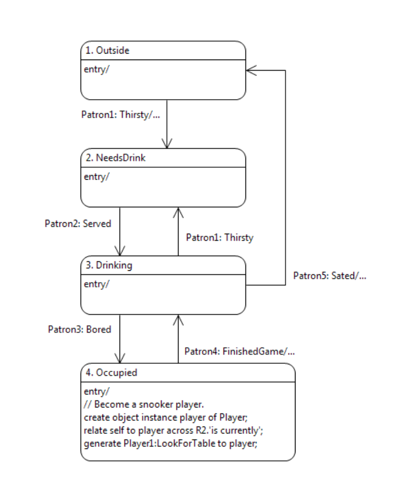
12. Model-Driven Testing

12

Model-Driven Testing

Model-driven testing is the notion that you can use models to build tests.

Test case



Types of Testing

There are two types of tests. Those that

are coupled only to the interface

owned by the Q&A (and anyone focused on the what not the how)

include knowledge of the models

owned by the modelers

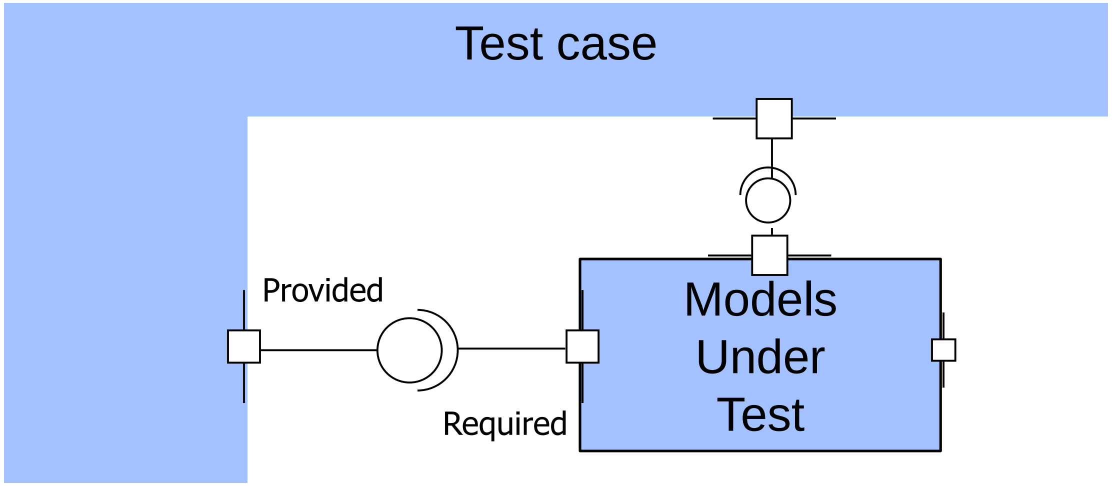
Black-Box Testing

Black-box testing tests the system from the outside.

Black-box testing knows only:

- what the actor wants from the system
- the interface

It treats the system as a “black box”.

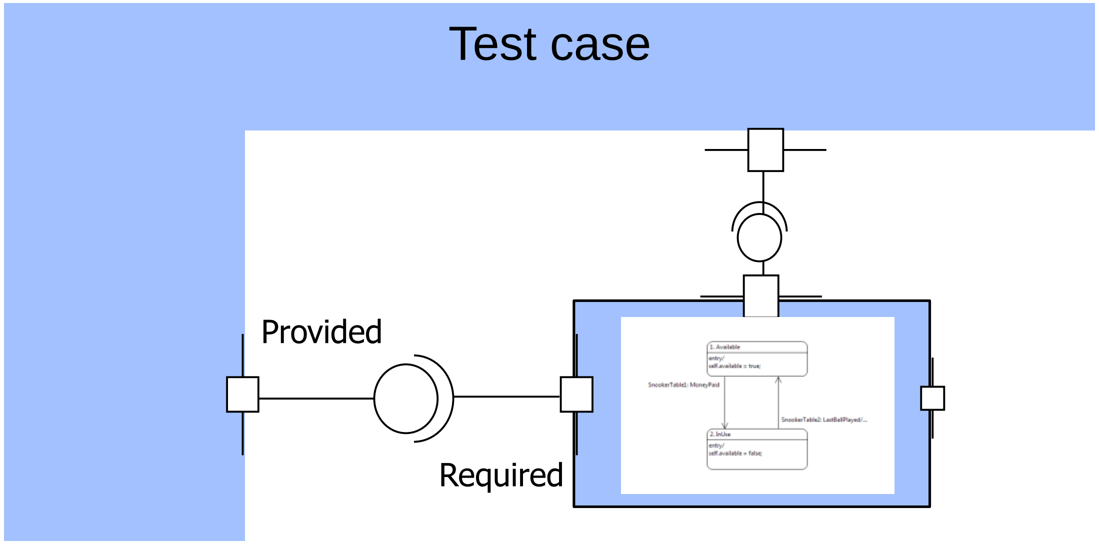


White-Box Testing

White-box testing tests the system from the inside.

Advantages of white-box tests include:

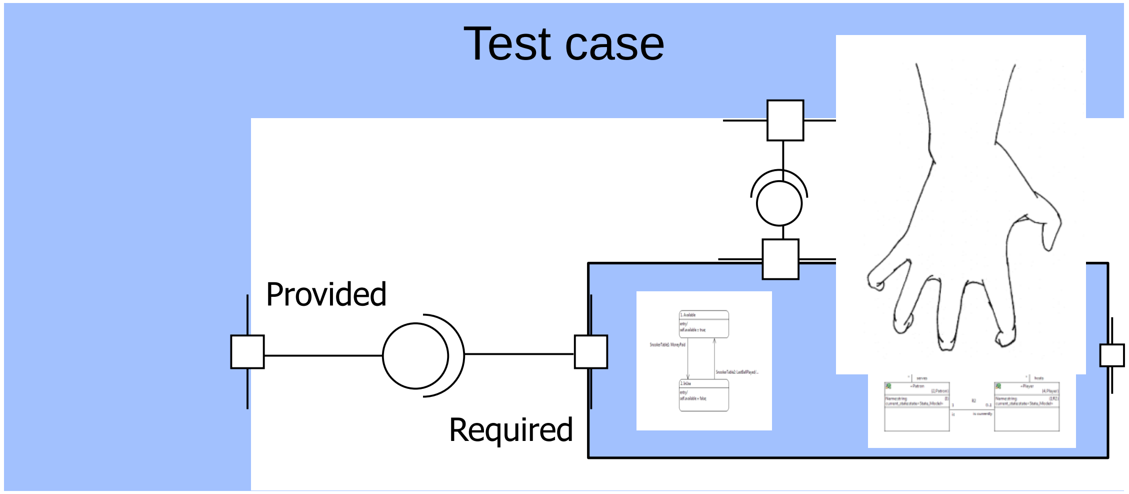
- Increased visibility and access, enabling
- finer-grained, more detailed testing
- Often simpler to build



White-Box Tests

White-box testing is all about the models. They can:

- Create and delete instances
- Access attributes and association links
- Anything you can do to a model



Use Cases

A use case says how a role uses a system to meet some goal.

Therefore, the use case becomes the basis for building tests.

Testing requires:

- Preconditions
- Stimulus
- (Expected) Postconditions
- (Actual) Postconditions
- A determination

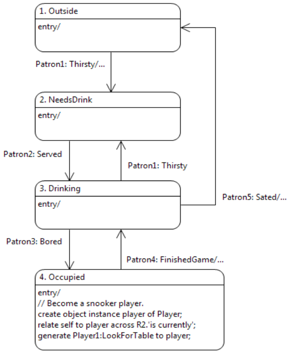
> You have these from the use cases.
>
> (Except for the last two!)

Testbench

A testbench supplies:

- The test execution framework
- Models of things outside the system
- Models of pieces of the system that are not yet available
- The test suite

Test Bench

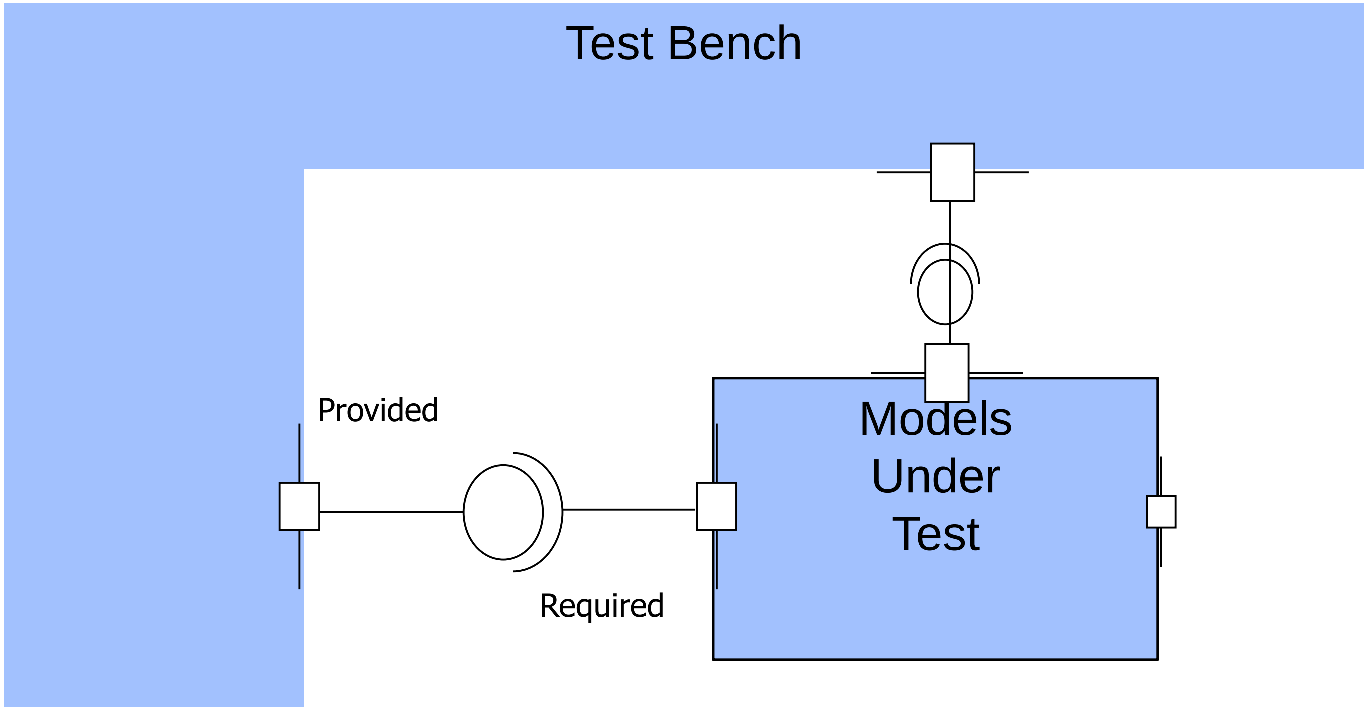


We must model what is around the system.

Testing Structure

Soooooo, create a:

- Test bench component with appropriate interfaces, then
- Connect it to the components under test



Use a Model

Use action language functions to establish preconditions, inject the initial stimulus and verify postconditions.

When necessary, use state models to

- inject additional stimuli,
- receive responses from models under test,
- Detect completion

Setup

create object instance FredsPlace **of** Pub;
FredsPlace.Name = "FredsPlace";

create object instance table **of** SnookerTable;
table.Number = 37;
table.available = **true**;

relate FredsPlace **to** table **across** R4.contains;

create object instance Tiny **of** Patron;
Tiny.Name = "Tiny";
.....

Workshop

Build a modeled test case, covering UC01 for the GPS Watch.

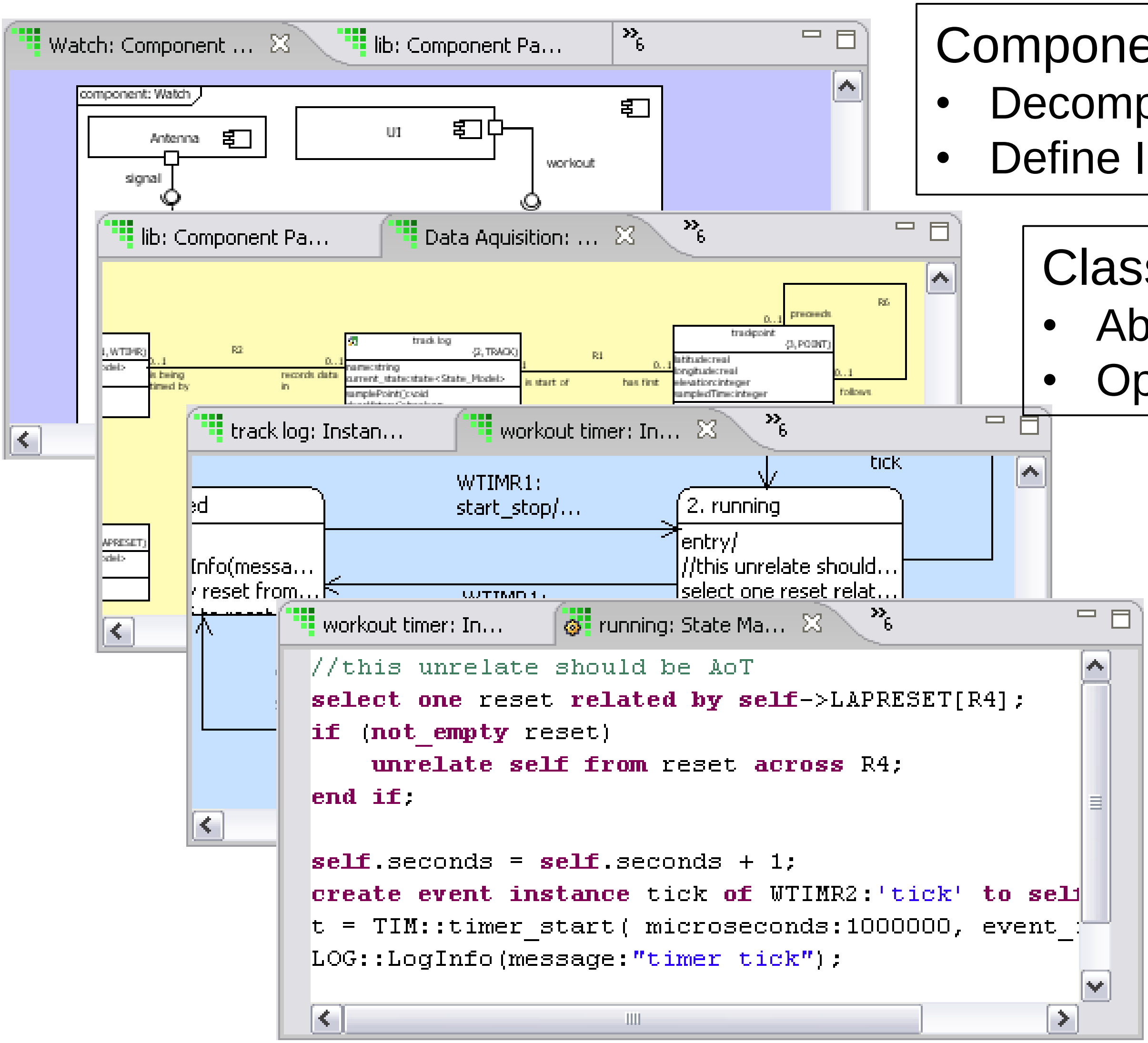
13. What’s Next?

13

Executable Model Hierarchy

High level

Low level



Component Diagram
- Decompose the application
- Define Interfaces

Class Diagram
- Abstractions
- Operations

State Diagram
- Lifecycle
- Event handling

Activities
- Processing

What's Next?

- Motivational Discussion
- Tool Introduction
- Requirements Clarification
- Basic xtUML Modeling
- Tool Training
- Completion of Case Study Model
- Team Modeling Exercise
- Advanced xtUML Modeling

You are here

Workshops

How to be most effective in workshops?

- Bridgepoint?
  - No. Focus is on modeling.
- Post-it notes?
  - Yes. Easy to move around, delete, rewrite
  - Use them for classes, states, components
- Flipcharts?
  - Yes. Good for collaboration.
  - Use for model canvas; post-it notes + drawn lines
- Phone cameras?
  - Useful for capturing prior versions