O.S. van Roosmalen

Lift Case
Software Architecture Document

Version 2.0

Revision History

Date	Version	Description	Author
30-05-00	1.0	Initial version	Onno van Roosmalen
13-06-00	1.1	Minor textual changes. Class description templates added.	Onno van Roosmalen
15-07-00	2.0	Revision of state transition diagrams. Inclusion of specifications of interfaces.	Onno van Roosmalen

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

In this document the architectural model of the Lift-Case software is described and the main architectural design-decisions are explained. It supplies a reasonably complete quasi-formal description and specification of the software subsystems of a Lift Case implementation.

1.2 Scope

The architecture shall satisfy the requirements described in the Stakeholder Requests and the Supplemental Specification document, but only for as far as the information in these document pertains to the "Change Floor" Use Case. The Use Case Realization document will serve as a main starting point for determining the architecture.

1.3 References

[1] Lift Case – Stakeholder Requests

[2] Lift Case – Supplementary Specification

[3] Lift Case – UCS-Change Floor document

[4] Lift Case – Glossary

[5] Lift Case – Use Case Realization document

2. Architectural Representation

The result of the architectural design of the Lift-Case software is a package structure described in a set of UML diagrams. Of each package (except for may be the top-level one) the interface is described in detail. A package interface consists of the following.

1. Public classes of the package.
2. Public methods of the classes of a package (public attributes are avoided except when final).
3. Specification of behavior of the public methods in terms of contracts (pre-post conditions) referring to the behavioral description in the form of state charts.

The architectural description contains at least these aspects but usually provides more detail about package implementation than is strictly necessary. This is done to indicate the feasibility of the design. Class specifications are given using OCL. In addition, state charts are provided to describe the autonomous behavior of objects of the classes. Together, state charts and OCL specification give an accurate description of structure and semantics of the interface.

3. Architectural Goals and Constraints

There are a number of requirements and constraints on the software system that have an impact on the architecture or represent architectural design choices.

3.1 Constraints

There are the following constraints:

1. The software shall be implemented in Java.
2. The software shall run on standard platform (PC).
3. The software shall interact with the user through a regular PC user interface, mainly a mouse. The user interface shall be realized through the use of standard GUI libraries (e.g. java.awt).

These constraints have a number of consequences on the usability of the Lift Case as a workshop assignment. In particular typical real-time issues can be demonstrated with the case but actual real-time behavior cannot be guaranteed (the standard Java and Windows environment are too unpredictable for that). Typical issues on concurrency and synchronization can be illustrated very well, though with the Java language (see section 3.2).

3.2 Software Requirements

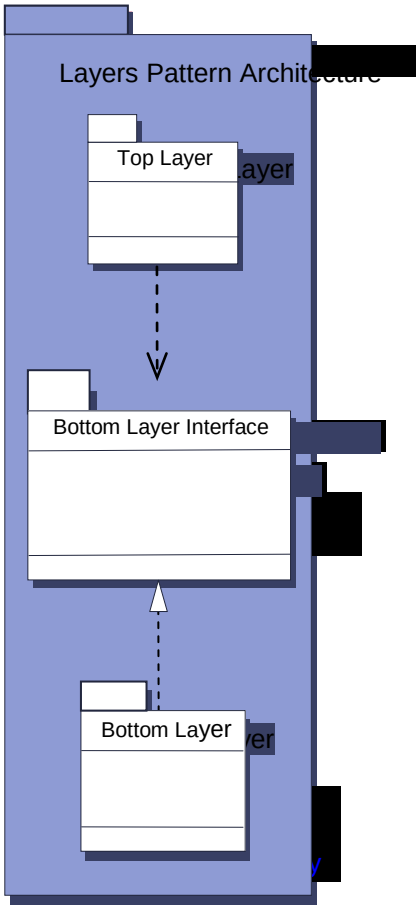
The most important non-functional requirements are the following.

1. The software shall be easily portable to situation with a different implementation of the sensors (buttons, detectors) and actuators (doors, motors) than the current representation on a regular PC display.
2. The core solution of the lift problem shall reusable within the limits set by the implementation constraints.
3. Changes required to cover other Use Cases shall be minimal.
4. The implementation shall illustrate typical issues of real-time programming, in particular: event handling, task (thread) design, task reduction heuristics, and task synchronization.
5. Because of the intended use of the case as workshop assignment, the internal quality (e.g. readability) of the software should be very high.

4. Logical View

4.1 Choice of architectural patterns

The architectural-design workflow takes the analysis model (see UseCaseRealization.doc and Solution**.ppt) as input. Since the analysis does not introduce packages, the analysis classes are to be distributed over design packages in an initial design step. At first it is appropriate to consider subsystems to deliver candidate packages. However, for the lifts system no useful independently reusable subsystems can be identified that consist of a non-trivial number of classes. Alternatively, the *Layers* architectural pattern can be considered. In the light of the requirements formulated in section 3.2, in particular portability and reuse. All classes that will encapsulate hardware software interaction will be placed in the bottom layer. Classes with control responsibility are placed in the top layer.



Top layer:	LiftSystem LiftCage
Bottom layer:	Chime Door Motor LiftButton Indicator

The major issue in the design of the lift system is the de-coupling of the application logic in the top layer from the classes involving hardware/software-interface and visa versa. This is to say that the core solution is to be made completely independent of the actual implementation of sensors and actuators. This is precisely what the *Layers* architectural pattern delivers. This pattern leads to the introduction of an additional package containing only interface descriptions.

The "lift" package contains the algorithmic lift control aspects (responsible classes *LiftSystem* and *LiftCage*) the "devices" package contains the interfaces of the boundary classes (such as *Door*, *Motor*) that provide an abstract view on the hardware, and a concrete devices package (here *GUIdevices*) that implements these interfaces. To obtain the desired dependencies between these packages (e.g. following from the *Layers* architectural pattern) some design patterns were applied. A detailed description of the package interfaces is described in later sections. The model resulting from this workflow can be found in Solution**.ppt. The package and class diagrams from this model are depicted in figure 1 and 2.

In addition to the packaging, the architecture model differs from the analysis model in the following ways.

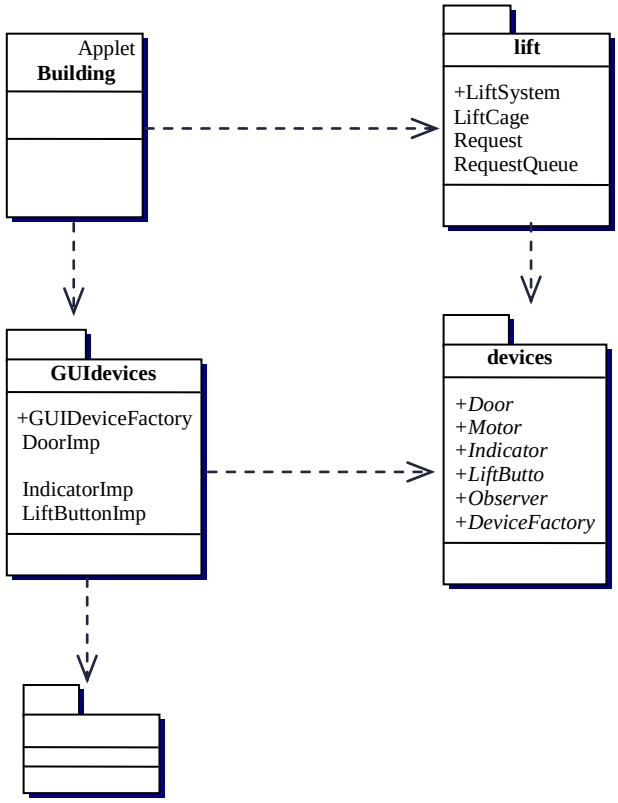
1. It is decided that doors on the user interface will each be sensitive to mouse events rather than the use of door buttons. Door buttons will not be placed on the user interface to avoid cluttering of features.
2. Chimes are not included in the design and omitted from the user interface because no distinction can be made between chimes at different locations on a PC implementation.
3. Only floor doors remain in the design. No distinction is made between cage doors and floor doors. Floor doors will be elements in the user interface.
4. Class Building is maintained as the root class in a separate client package. The class takes care of construction of the application objects. It can be seen as the client of the lift system (e.g., a building management system)

Deploying the lift system on a different platform only requires, in principle, substitution of the *GUIdevices* package by an appropriate other concrete devices package. Changes in the lift-allocation strategy are

entirely localized in the *lift* package. Inclusion of other Use Cases will, very likely, only have effect on the required interfaces of the *devices* package but not on the package structure itself.

4.2 Architecturally Significant Design Packages

The resulting packaging is shown in figure 1. Some classes have been added to the analysis classes to obtain the de-coupling required for the Layers pattern and to obtain a pure interface package. Three design patterns are required here: *Observer*, *Adapter* and *Abstract Factory*.



Application of the *Observer* pattern introduces one new class, namely *Observer*. It is applied as follows.

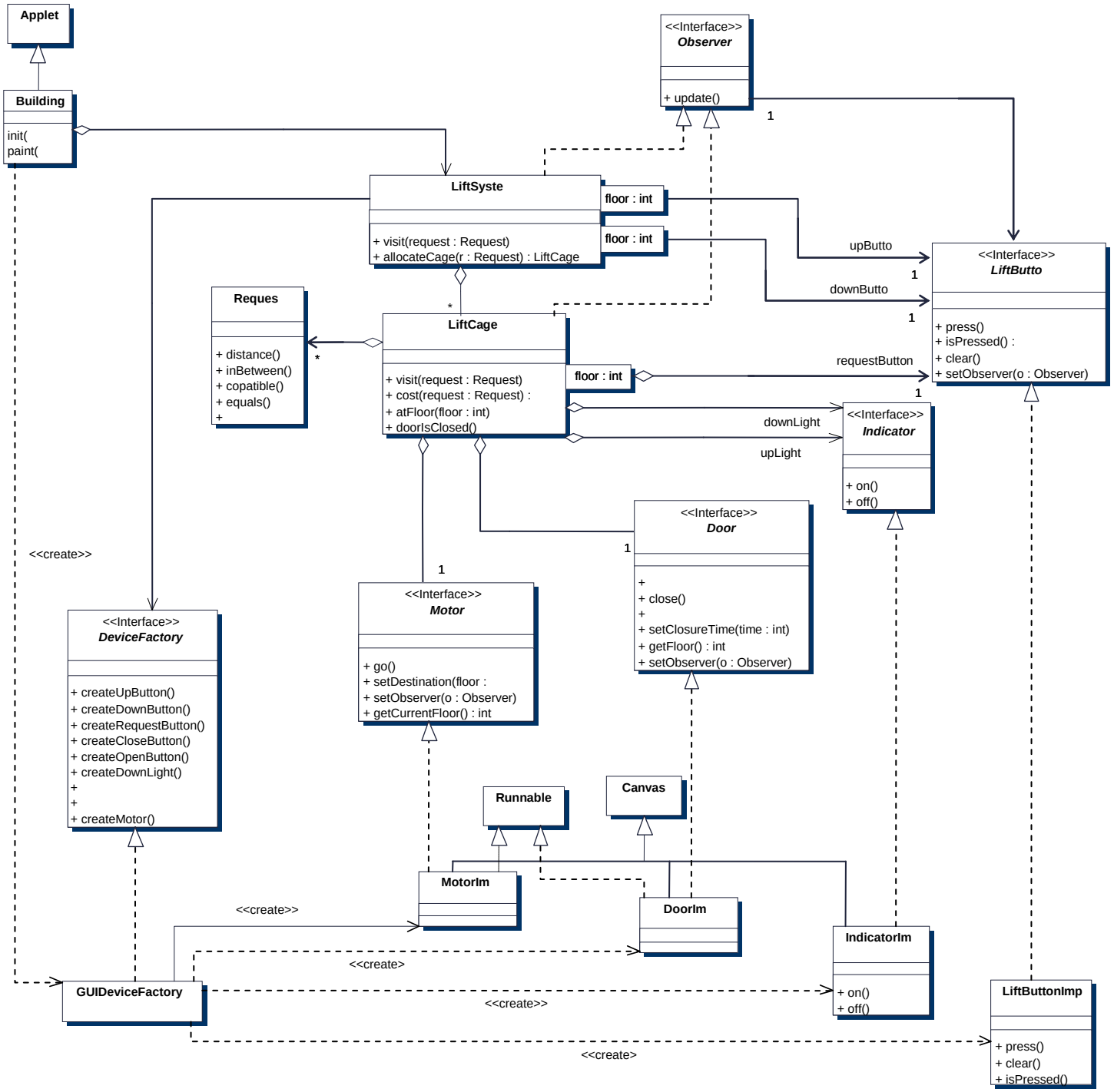
Observer Pattern Participants	**Classes in this design**
Subject	Abstraction is not introduced
Concrete Subject	*DoorImp* (from *GUIdevices*) *MotorImp* (from *GUIdevices*) *LiftButtonImp* from (*GUIdevices*)
Observer	*Observer* (from *devices*)
Concrete Observer	Anonymous in *LiftSystem* and *LiftCage*

Application of the Adapter patterns introduces only anonymous classes. It is applied as follows

Adapter Pattern Participants	**Classes in this design**
Client	*DoorImp* (from *GUIdevices*) *MotorImp* (from *GUIdevices*) *LiftButtonImp* from (*GUIdevices*)
Target	*Observer* (from *devices*)
Adapter	Anonymous in *LiftSystem* and *LiftCage*

Adaptee	*LiftSystem* (from *lift*) *LiftCage* (from *lift*)

These two patterns help to make *LiftSystem* and *LiftCage* completely reactive to external events. And take care that bottom layer classes have no knowledge whatsoever of top layer concepts. This is a prerequisite for introducing an interface package.



To enable top layer classes to construct the appropriate number of device objects the abstract factory interface is introduced. This pattern is applied as follows.

Abstract Factory Pattern Participants	**Classes in this design**
Client	
AbtractFactory	*DeviceFactory* (from *devices*)
ConcreteFactory	*GUIDeviceFactoryr* (from *GUIdevices*)
AbstractProduct	*Door* (from *devices*) *Motor* (from *devices*) *LiftButton* (from *devices*) *Indicator* (from *devices*)
ConcreteProduct	*DoorImp* (from *GUIdevices*) *MotorImp* (from *GUIdevices*) *LiftButtonImp* (from *GUIdevices*) *IndicatorImp* (from *GUIdevices*)

The resulting classes are divided into packages as follows.

1. main: *Building.*
2. lift: *LiftSystem, LiftCage.*
3. devices: *Door, Motor, LiftButton, Indicator DeviceFactory*
4. javadevices: *DoorImp, MotorImp, LiftButtonImp, IndicatorImp, DirectionLight, GUIDeviceFactory .*

4.3 Detailed Design Considerations

State charts are used to describe the behavior of objects of a many of the classes. Some of the classes (notably *LiftCage*) can therefore be conveniently implemented using the *State* pattern. It is not uncommon to introduce the states as anonymous inner classes.

The adapter pattern is used in combination with the Observer pattern to handle callbacks from the concrete devices package. Adapters (like states) are most conveniently introduced as anonymous inner classes. These anonymous classes are used in a way that the client has absolutely ho knowledge of the communication items expected by the adaptee.

4.4 Interface specification “lift” package

4.4.1 Class LiftSystem

4.4.1.1 Roles and responsibilities

The object of this class controls the state of the complete lift system. It may do so, by delegating substantial part of this responsibility to its aggregates.

4.4.1.2 Model

```
nFloors : Integer
nCages : Integer
```

4.4.1.3 Operations

```
context LiftSystem::visit(f : Integer, d : Integer)
pre:  0<=f and f< nFloor and (d=-1 or d=1)
The lift system will handle the request, i.e. a lift will arrive
at floor f in direction d (-1 is down, 1 is up) within the
constraints described in the requirements.
```

Note: a more accurate specification requires a model of the lift system that is considered here in the realm of the implementation specification. The analysis model can be considered a specification in this sense as well.

4.5 Implementation specification “lift” package

4.5.1 Class LiftSystem

4.5.1.1 Roles and responsibilities

The object of this class controls the state of the complete lift system. It may do so, by delegating substantial part of this responsibility to its aggregates (e.g. the lift cages). Ultimately, a lift cage must service the requests from the lift request buttons. The selection of the appropriate cage is the responsibility of the lift system object. It can make the allocation on the basis of a cost that can be assigned to the handling of a request by each of the cages. The cage with the minimal cost can best handle the request.

4.5.1.2 Model

```
nFloors : Integer
nCages : Integer
cage: LiftCage[nCages]
upButton[0..nFloors-1] : LiftButton
downButton[1..nFloor] : LiftButton
```

4.5.1.3 Operations

```
context LiftSystem::allocateCage(r  : Request) : Integer
pre:  0<=r.floor and r.floor< self.nFloor and
      (r.direction=-1 or r.direction=1)
post: result=self.cage->
      select(c : LiftCage | self.cage->forAll(c.cost()<=cost()))
```

```
context LiftSystem::visit(f : Integer, d : Integer)
pre:  0<=f and f< nFloor and (d=-1 or d=1)
post: let r : Request in
      self^visit(r) and
      r.oclIsNew and r.floor=f and r.direction=d

context LiftSystem::visit(r : Request)
pre:  0<=r.floor and r.floor< nFloor
      and (r.direction=-1 or r.directiond=1)
      post: let c : LiftCage = allocateCage(r) in
            c^visit(r)
```

4.5.1.4 Internal events and conditions

On button presses, buttons notify their observer. On an `update()` this observer calls the `visit(r : Request)` operation of the lift system with the appropriate `Request` object.

4.5.2 Class LiftCage

4.5.2.1 Roles and responsibilities

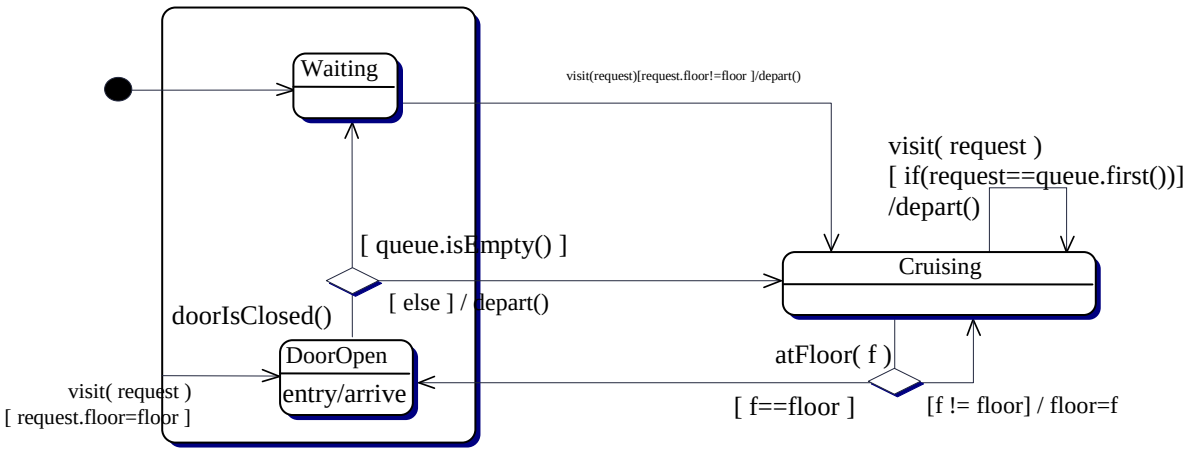
Each objects of this class manages and controls the state of a lift cage for as far as this task cannot be delegated to its aggregates. In particular it receives and schedules the floor requests for the cage and takes care these requests are serviced under the functional constraints that apply. It also coordinates the actions of its aggregates such as the motor and the doors.

4.5.2.2 Model

```
nFloors : Integer
nCages : Integer
floor : Integer
requestButton[0..nFloors-1] : LiftButton
door[0..nFloors-1] : Door
motor : Motor
queue : OrderedSet(Request)
```

4.5.2.3 States

`State::Waiting`	The lift cage is idle at a floor with the doors closed
`State::DoorOpen`	The lift cage is at a floor with the doors not closed
`State::Cruising`	The lift cage is moving between floors



4.5.2.4 Operations

```
context LiftCage::atFloor(f : Integer)
post: self.state@pre=State::Cruising implies
      if f==floor
      then self.state=State::DoorOpen and self^arrive()
      else floor=f
      endif

context LiftCage::cost(r : Request) : Integer
post: result=' a suitable cost calculation result '

context LiftCage::doorIsClosed()
post: self.state@pre=State::DoorOpen implies
      if queue.isEmpty()
      then self.state=State::Waiting
      else self.state=State::Cruising and self^depart()
      endif

context LiftCage::visit(r : Request)
post: queue=queue@pre->prepend(r)->sortedBy(distance(r))
      if self.floor=r.floor then
            (self.state@pre=State::Waiting or
             self.state@pre=State::DoorOpen)
             implies self.state=DoorOpen and self^arrive())
      else  (self.state@pre=State::Waiting
             implies self.state=Cruising and self^depart())
            and
            (self.state@pre=State::Cruising and
             self.queue.first()=r
             implies self.motor.destination=r.floor )
      endif

context LiftCage::arrive()
post:self.queue=self.queue@pre->excluding(self.queue@pre.first())
     and self.door[floor]^open

context LiftCage::depart()
post: self.motor.destination=self.queue.first().floor and
      self.motor^go()
```

4.5.2.5 Internal events

On button presses, buttons notify their observer. On an `update()` this observer calls the `visit(r : Request)` operation of the lift cage with the appropriate `Request` object. On motor and door events the appropriate calls to `atFloor()` and `doorIsClosed()` are made.

4.5.3 Class Request

4.5.3.1 Roles and responsibilities

Request objects capture information pertaining to a lift or floor request given to the lift system. Typical information is the floor of the request, the direction and the source (button) of the request.

4.5.3.2 Model

```
nFloors : Integer
floor : Integer
direction : Integer
button: LiftButton
```

4.5.3.3 Invariants

```
context Request inv:
0<=floor and floor<nFloors and -1<=direction and direction<=1
```

4.5.3.4 States

None. Request objects are immutable.

4.5.3.5 Operations

```
context Request::distance(r : Request) : Integer
post: result= self.distance(r.floor, r.direction)

context Request::distance(f : Integer, d : Integer) : Integer
post: if self.direction=0 then
            result = (f-floor).abs()
      else if d=0 then
            result = self.distance(f,1).min(self.distance(f,-1))
      else
            result = (f*d-self.floor*self.direction).mod
                                                (2*(nFloors-1))
      endif endif


context Request::inBetween(r1 : Request, r2 : Request) : Boolean
pre:    r1.direction <> 0
post:   result = (r1.distance(self)<r1.distance(r2))


context Request::equals(r : Request) : Boolean
post:   self.floor=r.floor and self.direction=r.direction


context Request::compatible(r : Request)
post:   self.floor=r.floor and self.direction.r.direction<>-1


context Request::setHandled()
post:   self.button^clear()
```

4.6 Interface specification “devices” package

4.6.1 Interface DeviceFactory

4.6.1.1 Responsibilities

Provides an interface for the creation of actual devices in a particular deployment of the lift system.

4.6.1.2 Model

```
nFloors : Integer
nCages : Integer
```

4.6.1.3 Operations

```
context DeviceFactory::numberOfFloors(): Integer
post: result=self.nFloors

context DeviceFactory::numberOfCages(): Integer
post: result=self.nCages

context DeviceFactory::createUpButton(f : Integer):LiftButton
pre:  0<=f and f<self.nFloors-1
post: result.oclIsNew() and result.state=LiftButton::Disabled

context DeviceFactory::createDownButton(floor:Integer):LiftButton
pre:  1<=f and f<self.nFloors
post: result.oclIsNew() and result.state=LiftButton::Disabled

context DeviceFactory::createRequestButton(f:Integer,c:Integer):
                                                    LiftButton
pre:  0<=f and f<self.nFloors and 0<=c and f<self.nCages
post: result.oclIsNew() and result.state=LiftButton::Disabled

context DeviceFactory::createCloseButton(c:Integer):LiftButton
pre:  0<=c and c<self.nCages
post: result.oclIsNew() and result.state=LiftButton::Disabled

context DeviceFactory::createOpenButton(c:Integer):LiftButton
pre:  0<=c and c<self.nCages
post: result.oclIsNew()and result.state=LiftButton::Disabled

context DeviceFactory::createUpLight(c : Integer): Indicator
pre:  0<=c and c<self.nCages
post: result.oclIsNew() and result.state=Indicator::Off

context DeviceFactory::createDownLight(c : Integer): Indicator
pre:  0<=c and c<self.nCages
post: result.oclIsNew()and result.state= Indicator::On

context DeviceFactory::createDoor(f:Integer,c:Integer): Door
pre:  0<=f and f<self.nFloors and 0<=c and c<self.nCages
post: result.oclIsNew()and result.state= Door::Closed

context DeviceFactory:: createMotor(c:Integer): Motor
pre:  0<=c and c<self.nCages
post: result.oclIsNew() and result.state= Motor::Idle
```

4.6.2 Interface Door

4.6.2.1 Roles and responsibilities

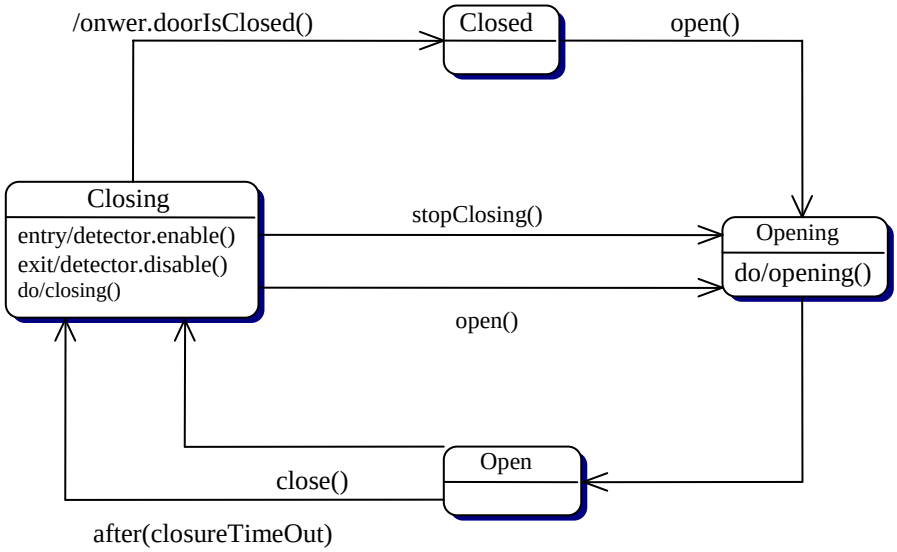
Each door handles all actions and activities that pertain to a door such as opening, closing, detecting obstructions in a closing door and handling door button events. The door notifies its observer of the vent of the door becoming closed.

4.6.2.2 Model

```
Observer : Observer
ClosureTimeout : Integer
state : <<Enumeration>>State
```

4.6.2.3 States

`State::Closed`:	The door is closed
`State::Opening`:	The door is opening with constant speed. If the opening process is not interrupted the opening time should be about 3 seconds.
`State::Open`:	The door is open. Without interference the door should go to the Closing state within closureTimeout milliseconds
`State::Closing`:	The door is closing with constant speed. If the closing is not interrupted the closing time is about 3 seconds



4.6.2.4 Operation specifications

```
context Door::close()
post: self.state@pre=State::Open implies
      self.state=State::Closing

context Door::open()
post: (self.state@pre=State::Closed or
       self.state@pre=State::Closing) implies
       self.state=State::Opening
```

```
context Door::setClosureTime(t : Integer)
post: self.closureTimeout=t

context Door::setObserver(Observer o)
post: self.observer=o

context Door::stopClosing()
post: self.state@pre=State::Closing implies
      self.state=State::Opening
```

4.6.2.5 Internal events and conditions

When the closing or opening activities terminate, anonymous events will cause state transitions. When an obstruction in the door is present while the door is closing an internal call to the `stopClosing()` is generated. When a close button or open button is pushed this generates an internal call to the `close()` and `open()` operations.

4.6.2.6 Other constraints

The response to an obstruction with a call to the open operation, should be reliable and timely.

4.6.3 Interface Indicator

4.6.3.1 Roles and responsibilities

Each indicator displays a light indicating a direction. The indicated direction should correspond to the lift cage's preference of travel.

4.6.3.2 Model

```
state : <<enumeration>> State
```

4.6.3.3 States

`State::On` : Indicator is lit.
`State::Off` : Indicator is extinguished.

4.6.3.4 Operations

```
context Indicator::on()
post: self.state=State::On

context Indicator::off()
post: self.state=State::Off
```

4.6.4 Interface LiftButton

4.6.4.1 Roles and responsibilities

Each button handles presses from the user and notifies its observer of such events. The press operation can is invoked on a user event on the physical user interface

4.6.4.2 Model

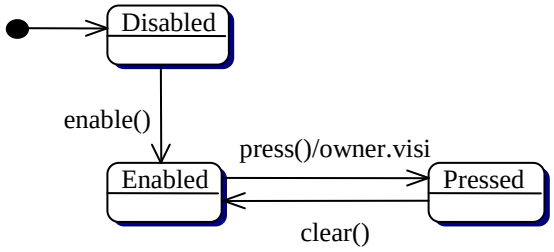
```
observer : Observer
buttonId : Integer
state : <<enumeration>>State
```

4.6.4.3 States

`State::Disabled:` The button is not lit and will not get lit when pressed

`State::Enabled:` The button is in its default state, i.e. it is not lit and will respond to a press.

`State::Pressed:` The button is lit to indicate that a request has been placed



4.6.4.4 Operations

```
context Button::clear()
post: self.state@pre=State::Pressed implies
      self.state=State::Enabled

context Button::setObserver(o : Observer)
post: self.observer=o and
      (self.state@pre=State::Disabled implies
       self.state=State::enabled)

context Button::press()
post: state@pre=State::Enabled implies state=State::Pressed
      and observer^update()

context Button::isPressed() : Boolean
post: result=(self.state=State::Pressed)
```

4.6.5 Interface Motor

4.6.5.1 Roles and responsibilities

Each motor handles all actions and activities that pertain to moving a lift cage. Cruising, detecting floors, and stopping at destinations. The motor indicates important events to its observer. The important events to indicate are the moment the motor passes a floor sensor, or stops at a floor.

4.6.5.2 Model attributes

```
observer : Observer
floor : Integer
destination : Integer
direction : Integer
state : <<enumeration>>State
```

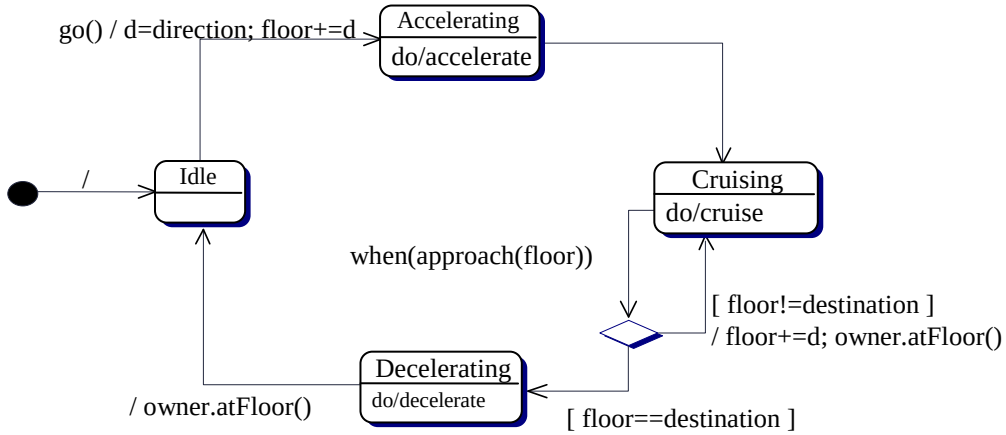
4.6.5.3 States

`State::Idle`: The motor and the attached lift-cage are not moving. Under normal conditions the lift-cage must be at a floor. The floor number is observable at entry.

`State::Accelerating`: The motor is accelerating the lift-cage at departure from a floor.

`State::Cruising`: The motor is moving the lift-cage at non-zero constant speed.

`State::Decelerating`: The motor is decelerating the lift-cage before arrival at a floor. Acceleration and deceleration should be constant and not exceed a certain acceptable value.



4.6.5.4 Operations

```
context  Motor::getCurrentFloor()
post: result=self.floor

context  Motor::setObserver(o : Observer)
post: self.observer=o

context Motor::setDestination(f : Integer)
post: self.destination=f


void Motor::go()
post: (self.state@pre=State::Idle and
       self.floor<>self.destination)
       implies (self.state=State::Accelerating and
            if self.floor<self.destination then
                self.d=+1
            else self.d=-1 endif)
```

4.6.5.5 Internal events and conditions

`when(Motor::approach(floor : Integer) : Boolean)`

The motor reaches a particular distance from a floor. At this distance deceleration must be started to arrive at the floor. Not starting deceleration at this point implies that the floor must be passed without stopping and the actual floor number must be changed by one.

Note: the arrival at a floor is indicated to the observer by the successive generation of the same `update()` event. One event at the moment the previous floor was passed, the other at arrival.

4.6.6 Interface Observer

4.6.6.1 Responsibilities

Notification interface. Devices can use this interface to notify an observer of relevant state changes

4.6.6.2 Model

Empty

4.6.6.3 Operations

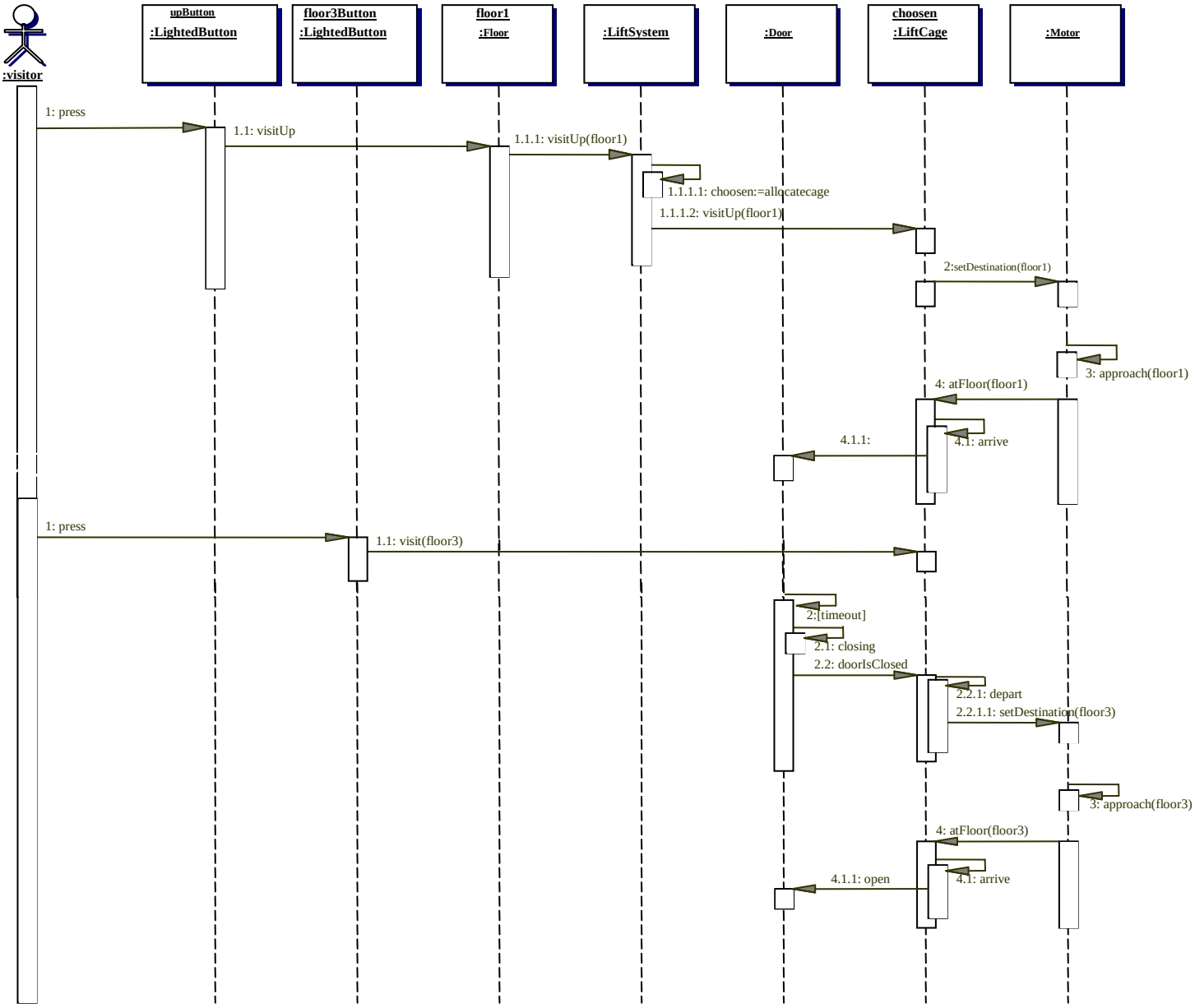
```
context Observer::update()
post: true
```

5. Use-Case View

The sequence diagram implementing the “Change Floor” Use-Case is given in the figure below. There are slight differences with the analysis version given in the “Use Case Realization” document. In particular the assumed threading in the motor and door objects has been made more explicit. Note, however, that these implementation issues can be solved differently in the device implementation package as long as the communication behavior of that package remains as specified.

The scenario depicted is as follows:

1) The person presses a lift-up button at floor i
2) The lift cage “cage“ arrives and the door opens
3) The person enters the lifts cage
4) The person presses the floor button for floor j
5) The door closes and the lift departs
6) The lift cage “cage“ arrives and the door opens.
7) The person exists the lift cage



Process View

In this architecture, the lift package classes are all completely reactive to call made from other packages. In particular:

At initialization: called from main package.
At device events: called from `(GUI)devices` package.

This makes careful consideration of race conditions necessary. It is a shame that Java does not offer primitives to lock and unlock objects completely at will of the programmer. Locking can take place only in synchronized blocks, and blocks can only be nested such that locks are always given up in reverse order as in which they are acquired.

The number of threads is kept to a minimum by creating thread objects as soon as they are needed, rather than statically allocating threads to active objects. Because there are so many potentially active objects this drastically reduces the number of existing threads in a snapshot. This reduction has to be balanced against the thread creation overhead. It is entirely possible to use thread pools to avoid thread creation. This thread pool may have a minimal size. For the `GUIdevices` variant of the `devices` package implementation one requires then just 1 thread for the buttons and one thread per cage that can be shared by the door and the motor (note door and motor are never simultaneously active).

```
package GUIdevices

context GUIDeviceFactory inv:
Thread.allInstances->size()<=nCages +1

endpackage
```

(Note: this seems not a legal OCL expression since the class `Thread` does not reside in the indicated package. It is puzzling how else one should specify this kind of constraint.) This very limited number of threads is stemming from the lift cage invariant:

```
context LiftCage inv:
self.door->select(state<>Closed)->size() +
self.motor->select(state<>Idle))->size()<=1
```

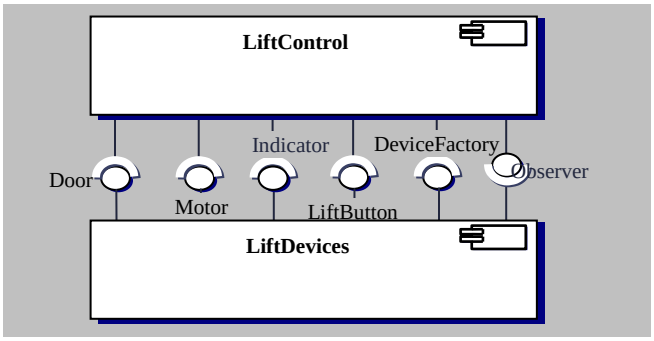
This implies that in each cage only one door or the motor object is active and will need a thread object in existence.

6. Deployment View

The software is to run on a single computer.

7. Component View

The component structure is based on the package structure and organization of the logical model. Two `jar`-files are produced `liftDevices.jar` and `liftControl.jar`. The first contains the `GUIDevices` and the second `GUIdevices` packages.



8. Size and Performance

No estimates are made on performance on the basis of the architecture. The present version can be used to analyze performance issues.

After initialization, all classes, except the request class, have a constant number of instances in each snapshot. The number of request can vary largely depending on the state of the system but are also subject to an upper bound.

```
context LiftSystem inv:
LiftSystem.allInstances->size()=1

context LiftCage inv:
LiftCage.allInstances->size()=nCages

context Request inv:
Request.allInstances->size()<=nCages*nFloor+2*(nFloors-1)

context DeviceFactory inv:
Device.allInstances->size()=1

context Door inv:
Door.allInstances->size()=nFloors*nCages

context Indicator inv:
Indicator.allInstances->size()=2*nFloors*nCages

context LiftButton inv:
LiftButton.allInstances->size()=2*(nFloors-1)+nCages*(nFloors+2)

context Motor inv:
Motor.allInstances->size()=nCages


context Observer inv:
Observer.allInstances->size()=nCages*(2*nFloors+1)+2*(nFloors-1)
```

9. Quality

This architecture has explicitly addressed the following non-functional aspects (see section 3.2)

1) Portability through a layered architecture
2) Reuse by isolating the aspects of lift scheduling and control in a single package
3) Internal quality by specific attention on issues of race conditions and thread synchronization. Also aspects of design by contract have deserved much attention. Most interface aspects are fully specified in OCL.

Extensibility with respect to other Use Case still has to be demonstrated.