O.S. van Roosmalen

Lift Case
Supplementary Specification

Version 2.0

Revision History

Date	Version	Description	Author
27-05-00	1.0	Adapted from inception-iteration document	Onno van Roosmalen
17-07-00	2.0	Lift allocation considerations added	Onno van Roosmalen

Table of Contents

Supplementary Specification

1. Introduction

The Supplementary Specification captures the system requirements that are not readily captured in the use cases of the use-case model. Such requirements include:

- Legal and regulatory requirements, including application standards.
- Quality attributes of the system to be built, including usability, reliability, performance, and supportability requirements.
- Other requirements such as operating systems and environments, compatibility requirements, and design constraints.

1.1 Purpose

The purpose of this document is to define requirements of Hospital Lift System. The Supplementary Specification lists the requirements that are not readily captured in the use cases of the use-case model. The Supplementary Specifications, the Stakeholder Requests and the use-case model together capture a complete set of requirements on the system.

1.2 Scope

This document applies to the Hospital Lift System Case for a Software Engineering course. The supplementary requirements partially emerge from the hospital lift domain and partially from the intention to use the case as an illustration for a Software Engineering course. This specification defines some functional as well as non-functional requirement, such as reliability, usability, performance, and supportability. It contains the requirements that were not stated in the Stakeholder Requests document (reference 2) and use-case description.

1.3 References

1. Lift Case – Vision
2. Lift Case – Stakeholder Requests

2. Functionality

2.1 Lift allocation

Lift allocation is a functional aspect that influences, to a large extend, the satisfaction of non-functional requirements. This section will address the issue without making a distinction between functionality and non-functionality.

2.1.1 Request handling

Lift and floor requests are submitted to the lift system in an unpredictable fashion. Their rate is exclusively determined by the arrival of users at the lift and their individual intentions with respect to the use of the system.

There are two types of requests, namely, (1) lift requests for a lift cage in the directions either up or down and (2) floor requests submitted by a user in a cage to request halting the cage at a particular floor. Lift requests are submitted to the system by pressing a lift request button at a particular floor, up or down, and floor requests by pressing a floor request button inside a particular lift cage.

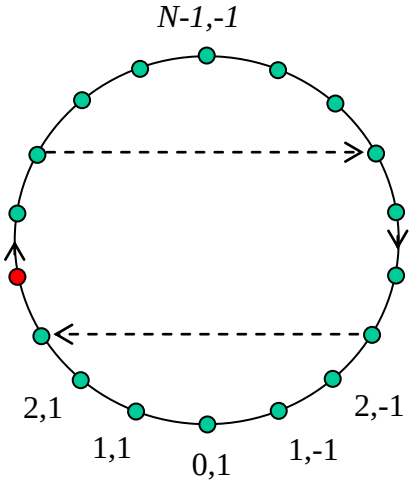
By their nature, floor requests can only be handled by the cage in which they were submitted. In contrast, lift requests can be handled by any of the available cages. However, one and preferably only one cage must handle a request. This cage must be allocated to the request by the lift system in an optimal way, that is, a way that best satisfies all performance and throughput requirements. Optimality is determined by a number criteria, namely (1) the waiting time for a request submitted by an individual user, (2) the average time

needed to satisfy a service request, (3) the total number of requests handled per time unit. There are subtle differences and conflicts between these criteria that are not being addressed here. An ad hoc developed allocation procedure is described in this section. It is an important requirement that the design of the lift system software shall anticipate on changes with regard to the described allocation procedure. First, a systematic approach is described to order requests submitted to a lift cage, then an approach for allocating lift requests to a cage.

2.1.2 Ordering requests

There are three types of requests that a cage must handle: a lift request for obtaining a lift going up at a particular floor *f*, a lift request for going down at a floor *f* and a floor request for a floor *f*. Each of these cases is denoted as a request *r* by *r=(f,1), r=(f,-1)* and *r=(f,0)* respectively. The parameter *N* is used as the number of floors.

A lift cage travels either up or down and passes floors in a prescribed order. The floors that can be visited, together with the possible direction in which they can be visited, can be placed on a circle as given in the figure. Each position *p* on the circle is a combination of the floor number, *f*, and the direction of traveling, *d*=±1 (*up*=1 *down*=-1). Thus *p=(f,d).* Note that, when referring to positions (rather than requests), the value *d=0* is not permitted. A lift cage moves clockwise on this circle. When a lift cage changes direction from up to down other than at the top floor, the position of the cage will cut across from left to right on the circle. When it changes direction from down to up it will cut across from right to left as exemplified by the dashed arrows.



A distance between positions on the circle is defined. This distance is the number of segments passed going clockwise from the first to the second position remaining on the circle.

$$distance(p_1, p_2) = (p_2.floor * p_2.direction - p_1.floor * p_1.direction) \bmod 2.(N-1) .$$

Note that this distance is not commutative: $distance(p_1, p_2) \neq distance(p_2, p_1)$. Instead if $p_1 \neq p_2$ we have the property:

$$(distance(p_1, p_2) + distance(p_2, p_1)) = 2.(N-1) ,$$

stemming from a constant distance (namely full circle) to travel to another position and back.

Requests, *r*, and positions, *p*, are not interchangeable with regard to this distance measure. Requests may have no preferred direction, i.e. *r.direction=0*, and can in that case be associated with two positions on the circle. Also the current position, *c*, of the lift cage may have no preferred direction (e.g. when the cage is

Waiting for a request). We call a request compatible with a position, if the request can be handled by visiting the position. It can be expressed as follows:

$$compatible(p, r) = (p.floor=r.floor) \wedge (p.direction*r.direction\neq -1)$$

Compatibility between two requests instead of a position and a request can also be defined like this. Compatible requests are requests that may be simultaneously satisfied by visiting the same floor. A request for a lift in upward- and a request in downward direction on the same floor cannot be simultaneously satisfied, so they are not compatible.

The distance between request r_1 and r_2 , when both have *direction=0,* is defined as the number of floor-transitions required for arriving at the floor of r_2 from r_1 associating positions p_1 and p_2 with r_1 and r_2 in a way that minimize this number.

$$distance(r_1,r_2)|_{r1.direction=0,\ r2.diecrtion=0} = min\ (p_1\ ,\ p_2 : compatible(p_1,r_1) \wedge compatible(p_2, r_2): distance(p_1\ ,\ p_2)\ ) = |\ r_1.floor - r_2.floor\ |$$

The distance a request *r* with *r.direction=0* to a position is:

$$distance(r\ ,\ p\ )\ |_{r.direction=0} = min\ (r_1\ : compatible(p\ ,\ r_1\ ) : distance(r\ ,\ r_1\ )\ ) = |\ r\ .floor - p\ .floor\ |$$

This means that direction neutrality remains during travel at least when calculating distances. The distance from a position to a request is:

$$distance(\ p\ ,\ r\ ) = min\ (p_1\ : compatible(p_1\ ,\ r\ ) : distance(p\ ,\ p_1\ )\ )$$

This distance measure is introduced to be able to order requests directed to a particular cage. All requests are placed in a request queue, *Q* and are given by this placement an absolute order. Naturally, where a request should be placed depends on the current position of the cage of moment of submission of the request. For example, if a request can be satisfied along the way traveling from the current lift cage position to the first item in the queue, the request will be placed in the front of the queue. Hence, two notions are required, the current lift cage position and the function *between(r, (r_1, r_2))* which establishes whether a request *r* can be satisfied along the way from r_1 to r_2.

The current lift cage position is denoted by *c=(f, d)* with *d=0, ±1*. An assumption is that *d=0* only occurs when the lift cage has no requests and is in the *Waiting* state. Because it is the only situation in which it is assumed to show no preference for direction. Even if *Q* is empty and the door is still open the lift cage has a preference to continue in the direction in which it was going. Changes from *d=+1* to *–1* and v.v. only occur at two moments: (1) when a lift request is satisfied in the direction opposite of the direction of travel (this request is than in the front of the queue), and (2) at the moment that a request in the front of the queue is selected as the new destination and the shortest travel path involves a change in direction.

Hence, except when queuing a request in an empty queue (trivial case), one may assume that current position *c* has a direction $c.d \neq 0$. This is important because the *between* function can only be appropriately defined when the first item is a position (Floor requests in the direction of travel are then favored over lift requests in opposite direction as with most lift systems.) In that case we have:

$$between(r, (p_1, r_2)\ ) = distance(p_1, r) < distance(p_1, r_2)\ ,$$

A new request *r* is placed in the queue between two requests r_1 and r_2 , when:

$$\neg\ between(r, (c, r_1\ )) \wedge between(r, (c, r_2\ ))$$

2.1.3 Allocating lift requests to cages

The allocation of lift requests to cages can best be based on the relative cost of handling the request by each of the available cages. A cost function abstracts from details of the state of the cage such that the behavior of a cage may be changed without it having a substantial effect on the allocation procedure. Furthermore it allows having different cost calculations for different cages.

The cost-computation, that is proposed here, is not predictive. It is based on a snapshot of the system. Thus, it only takes into account the currently pending requests submitted to a cage and the current location and direction of a cage. This choice may have major impact on the software structure and one shall anticipate different choices.

Thus: ***three*** approximations are made in the proposed approach. (1) A single parameter of the cages (namely the cost) is compared. (2) The cost is computed at the moment a request is submitted and allocated, however, between submission and satisfaction of a request this cost will change and other allocations may become more optimal. (3) The computation does not anticipate on future requests nor does it make use of historic information.

The cost-function may involve various factors that must be weighed to obtain a desired behavior.

- The bare travel time (i.e. without accounting for stops) it takes to handle the submitted lift request. It can be based on the distance measure given in the previous paragraph.
- The number of stops that have to be made before the request is handled. (Note: the stop time is not predictable, users can delay departure of a lift cage).
- The time by which other requests are delayed because of added stop due to the new request.

The cost algorithm shall distinguish these various cost components such that their relative contribution can be changed easily.

2.2 Lift warning system

For the purpose of illustrating extensibility the lift can be considered an integral part of the Hospital Safety and Security System.

- Manual warning mode
  Each lift cage contains a button (guarded by a glass pane, that must be smashed before the button can be used) that signals the control system of the hospital.
  The computer system's behavior will depend upon the context. If the lift cage is at floor level, the floor and cage doors will be opened and the lift shaft will be excluded from scheduling.
- Automatic warning mode
  The lift system will automatically warn the control system of the hospital in case it detects any malfunctioning, inconsistency or other abnormal behavior in the system.
  In case normal operation is jeopardized, the lift system will degrade to a level that acceptable from a safety point of view.
  In case a lift shaft is excluded from scheduling the system will check if a stop at the nearest floor is possible.

3. Usability

3.1 Workshop case

The case shall be useable as a workshop case or illustration case during a Software Engineering course . This requires that that several extension can be defined that may have non-trivial impact on the design and implementation. There should also be requirements typical for real-time systems with regard to performance, timeliness, reliability and portability.

3.2 Rush hour considerations

During the visiting hours, it is expected to transport 1400 people (an estimation based on the annual growth) in 30 minutes.

The number of 1400 visitors is based on the assumptions that in the future there will be 1000 beds, and 70% of the patients is visited by 2 people (visitor index equals 1.4).

3.3 Acceleration

Acceptable acceleration for regular use is restricted to 4 m/s^2.
Acceleration during patient transport is restricted to 1m/s^2.

3.4 Empty lift call

The response time of an 'empty lift'[1] is 30 seconds or less.

4. Reliability

TBW.

5. Performance

The performance should such that the extreme circumstances, like during rush hour, can be handled.

6. Supportability

6.1 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. (This is actually related to the portability requirement).
3. The changes required to cover the 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, task synchronization, and schedulability.

7. Design Constraints

There are the following design constraints:

1. UML shall be employed as a design language.
2. Ideas on design by contract shall be followed as closely as possible.
3. The RUP process shall be followed as closely as possible.
4. The software shall be implemented in Java.
5. The software shall run on standard platform (PC).
6. 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 realized. These and other issues will be discussed in an evaluation document.

[1] An 'empty lift' is a lift cage that has been called for special purposes and has no outstanding requests. In most cases it is used for patient transport.

8. Online User Documentation and Help System Requirements

Not necessary for this project.

9. Purchased Components

The development environment is considered freely available.

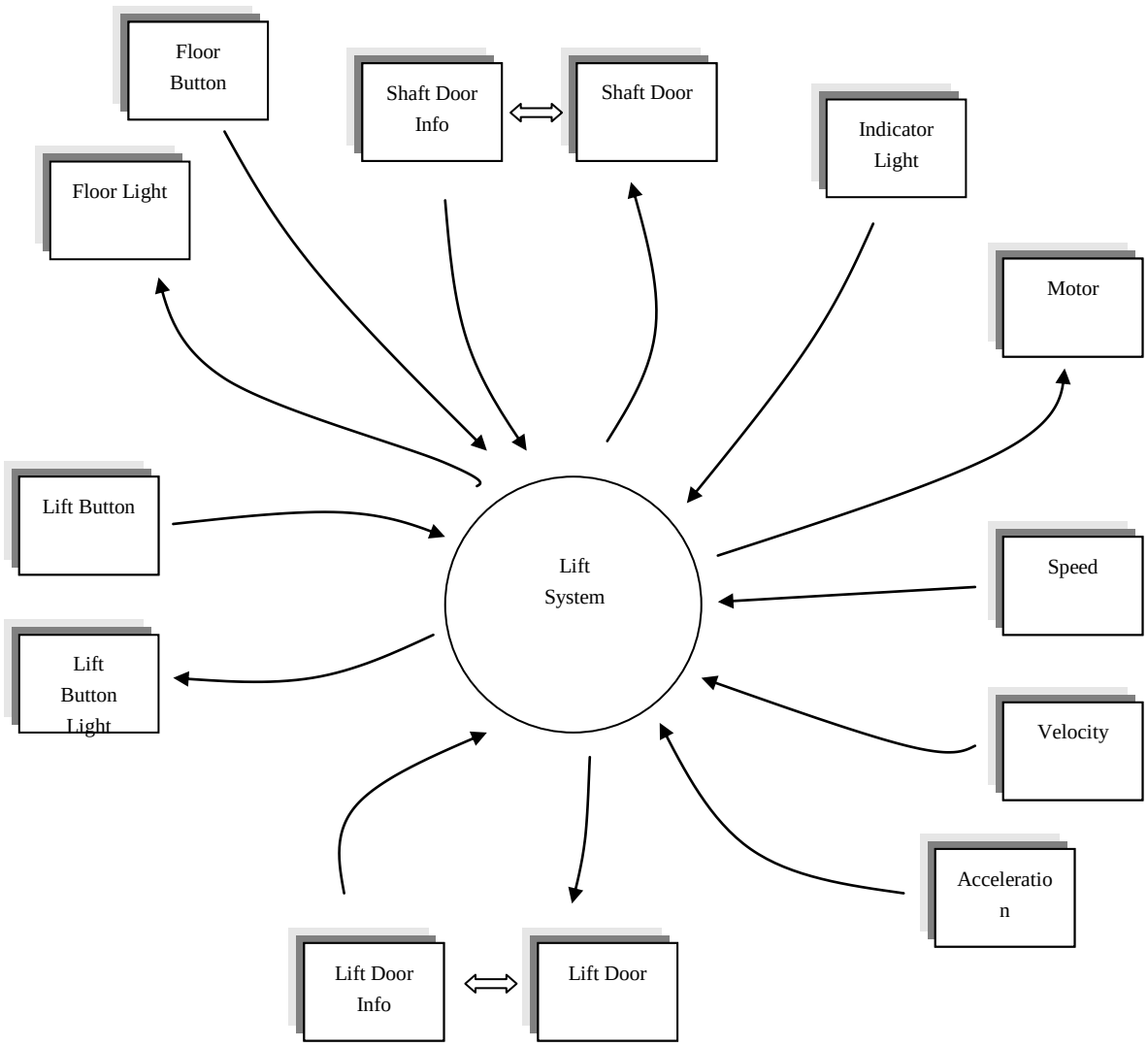
10. Interfaces

10.1 User Interfaces

Apart from the regular button interface and door sensors, there are no other user interfaces in this project.

10.2 Hardware Interface

Not applicable for this project. The design shall anticipate on generally appropriate hardware interfaces for lift systems. The figure indicates typical sensor and actuator types for a lift system.



10.3 Software Interfaces

Only interfaces with the java platform and libraries are relevant in this project.

10.4 Communications Interfaces

The lift system should allow extensions like communication with e.g. with a Hospital Safety System.

11. Licensing Requirements

Not applicable for this project.

12. Legal, Copyright and Other Notices

Not applicable for this project.

13. Applicable Standards

One could possibly consult:

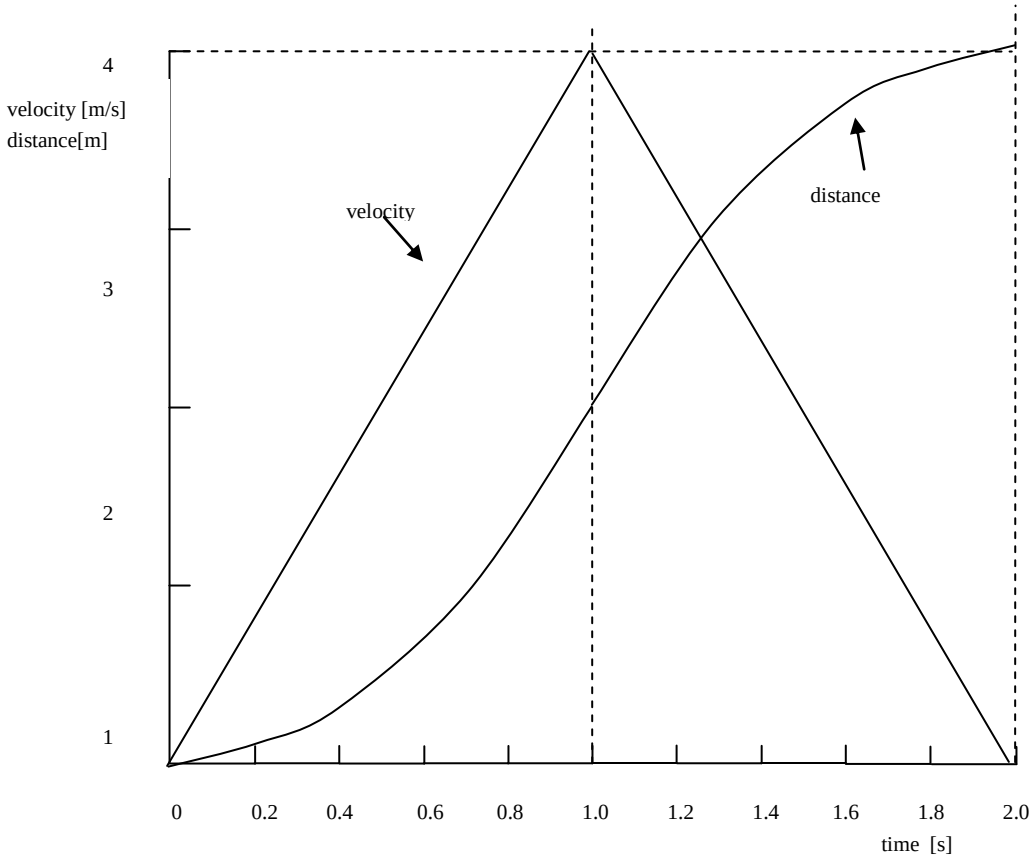
ISO-9000, hospital sub-standards

GHO, good hospital practices

EHSR, European Hospital Standards and Regulations

14. Appendix 1 – Some calculations

From the usability figures (section 3.3) it is known that acceleration 4 m/s^2. The time to travel from floor m to floor n takes 1+|m-n| seconds (see figure above). From data gathered during experiments and from experiences in other hospital, it is estimated that in a ten-floor hospital, the average number of stops during visiting hours is approximately five. The time needed to complete such a trip (all times in seconds):



travel time (up, 1+10sec)	11
5 stops (open-wait-close, 20 sec)	100
travel time (down)	11
5 stops	100
---	
total trip time	122

Having 8 lift cages that can carry 15 people each it is possible to transport 120 people in 122 seconds.

So the number of people per second: 120/122	≈ 1
To transport 1400 people takes	= 1400 s
	≈ 23 minutes