

# COMP-667 Software Fault Tolerance

---

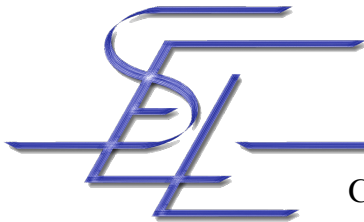
## Dependable Requirements Engineering

Jörg Kienzle

School of Computer Science, McGill University  
Montreal, Canada

Contributing authors:

Sadaf Mustafiz, Shane Sendall, Aaron Shui, Alexander Romanovsky, Christophe Dony,  
Hans Vangheluwe, Ximeng Sun

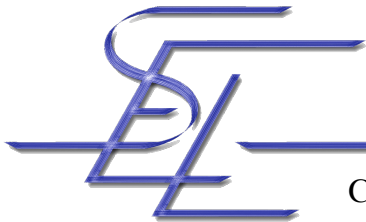


**McGill**

# Overview

---

- Use Cases
  - Use Case Template
  - Actors
  - Use Case Granularity
  - Use Case Diagrams
- Dependability-Focused Requirements Engineering Process
  - Motivation
  - Context-Affecting Exceptions
  - Safety and Reliability Handlers
  - Service-Affecting Exceptions
  - Dependability Assessment
- Conclusion & Future Work

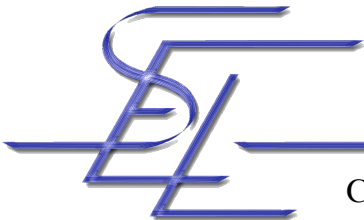


McGill

# Requirements Elicitation (1)

---

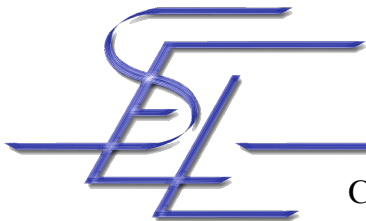
- Discover the requirements of the system to develop
  - User expectations
  - Functional requirements
  - Non-functional requirements / qualities
    - Distribution
    - Security
    - Safety
    - Reliability
    - Fault Tolerance
    - Availability



# What are Use Cases good for?

---

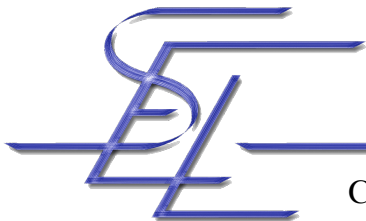
- Discover and *document* the functional requirements of the desired system
- In a way that *all important participants of a project can understand*
- In a way that is clearly related to the motivation for the system (e.g., business vision)
- In a complete, consistent, and verifiable manner



# Use Cases

---

- Use Cases capture interactions between the system and the environment to achieve *user goals*
- Use cases capture who (actor) does what (interaction) with the system, with what purpose (goal), without dealing with system internals.
- A complete set of use cases specifies all the different ways to use the system, and thus defines all behavior required of the system, bounding the scope of the system.
- Designed to be understood by non-technical parties



McGill

# Use Case Comments

---

- Being a black-box view of the system, use cases are a good approach for finding the What rather than the How
- A black-box matches users view of the system: things going in and things coming out
- A use case sums up a set of scenarios:
  - Each scenario goes from trigger to completion
- Use cases can help formulate system tests:
  - “Is this use case built into the system?”

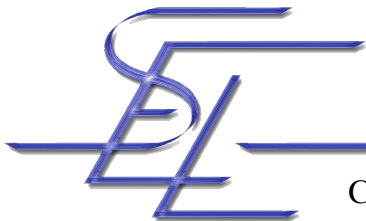


McGill

# Actors: What Are They?

---

- “The actors represent what interacts with the system.” [Jacobson ‘92]
- An actor represents a role that an external entity such as a user, a hardware device, or another system plays in interacting with the system
- A role is defined by a set of characteristic needs, interests, expectations, behaviors and responsibilities [Wirfs-Brock ‘94]



# Actor Comments

---

- An actor communicates by sending and receiving messages to/from the system under development.
- A use case is not limited to a single actor.
- Sources, i.e. how to discover actors:
  - People: Workshops, Meetings, etc.
  - Documentation: user manuals and training guides are often directed at roles representing potential actors
  - Domain Model

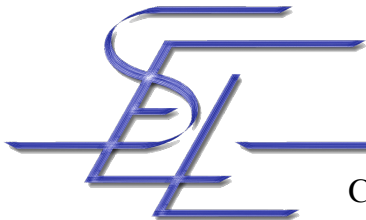




# How to Find Actors

---

- Look for external entities that interact with the system
  - Which persons interact with the system (directly or indirectly)? Don't forget maintenance staff!
  - Will the system need to interact with other systems or existing legacy systems?
  - Are there any other hardware or software devices that interact with the system?
  - Are there any reporting interfaces or system administrative interfaces?



# Actor Categories

---

- Jacobson (1992) categorized actors into two types:
  - Primary Actor
    - actor with goal on system (sometimes off-stage)
    - obtains value from the system
    - Sometimes, primary actors interact with the system through facilitator actors
  - Secondary Actor
    - actor with which the system has a goal
    - supports “creating value” for other actors
  - Facilitator Actor
    - actor or device that is used by a primary actor or secondary actor to communicate with the system



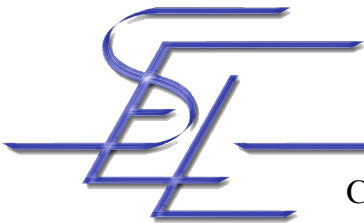
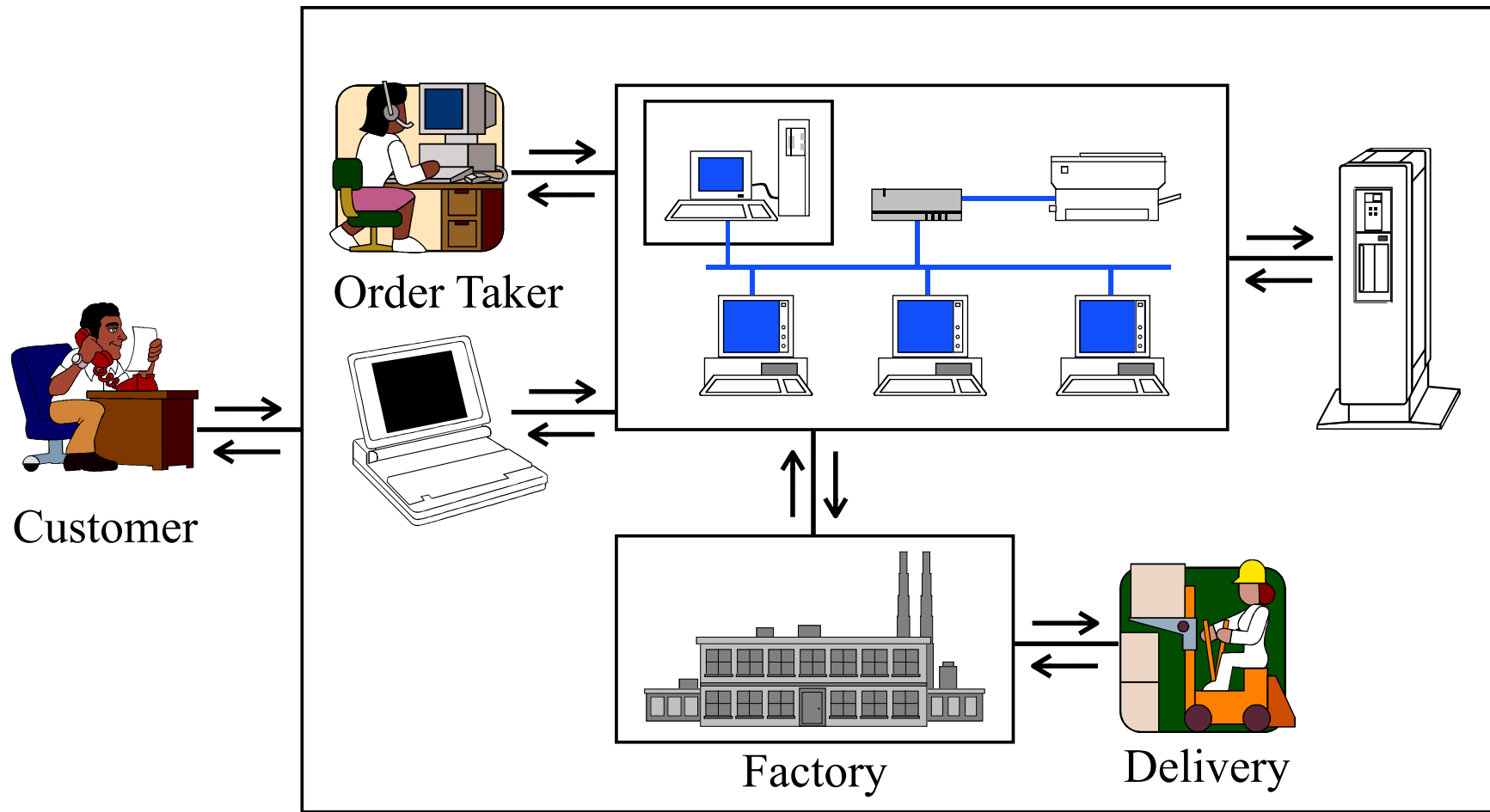
# System Boundary

---

- The system boundary defines the separation between the system and its environment
  - Clear definition is extremely important
- Movement of the system boundary often has a large effect on what should be built
- A common area of conflict between stakeholders arises when they assume different system boundaries, and hence refer to different systems!



# System Boundary Example



# Use Case Description

---

- Use cases are primarily textual descriptions
- Use case steps are written in an easy-to-understand structured narrative using the vocabulary of the application domain
- A use case description includes
  - How the use case starts and ends
  - The context of the use case
  - The actors that interact with the system
  - All the circumstances in which the primary actor's goal is reached and not reached
  - What information is exchanged



# Structured Use Case Template

---

- Use Case Name
- Scope
- Level
- Intention in Context
- Multiplicity
- Primary Actor
  - Secondary Actors
- Main Success Scenario
  - Sequence of Interaction Steps
- Extensions & Exceptions
  - Additional / Alternative Interaction Steps



# Interaction Steps

---

- An interaction step either
  1. Refers to a lower level use case
  2. Describes a base interaction step between the system and the environment
    - A base interaction step must always contain the word *System* and (at least) an actor and
      - Describes an *input interaction*, when an actor sends an input event to the system, or
      - Describes an *output interaction*, when the system sends an output event to an actor
  3. Describes an optional system processing step or communication happening in the environment that is included for clarity.
- Interaction steps are numbered to reflect their sequencing
  - The “.” notation, i.e. “3.1”, denotes sequential “sub”steps
  - Letters, i.e. “3a”, denotes alternatives to a step
  - The “||” symbol denotes parallelism



# Single-Cabin Elevator Example

**Use Case:** TakeElevator  
**Scope:** Elevator Control System  
**Primary Actor:** User  
**Intention:** The intention of the *User* is to take the elevator to go to a destination floor.  
**Level:** User Goal  
**Main Success Scenario:**  
1. *User* Call[s]Elevator  
2. *User* Ride[s]Elevator  
**Extensions:**  
1a. Cabin is already at *User's* floor..  
1b. *User* is already inside..

**Use Case:** CallElevator  
**Primary Actor:** User  
**Intention:** User wants to call...  
**Level:** Subfunction  
**Main Success Scenario:**  
1. *User* pushes button, indicating to *System* in which direction she wants to go.  
2. *System* acknowledges *User's* request.  
3. *System* schedules ElevatorArrival for the floor the *User* is currently on.  
**Extensions:**  
2a. The same request already exists. System ignores the request..

- Main success vs. extensions
- Hierarchy





# Elevator Arrival Example

---

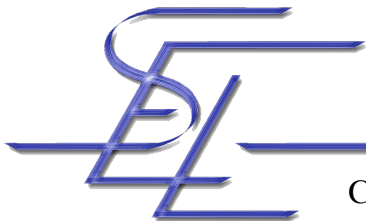
**Use Case:** ElevatorArrival

**Intention:** System wants to move the elevator to the User's destination floor.

**Level:** Subfunction

**Main Success Scenario:**

1. *System* asks *Motor* to start moving in the direction of the destination floor.
2. *Floor Sensor* informs *System* that elevator is approaching destination floor.
3. *System* requests *Motor* to stop.
4. *System* requests *Door* to open.

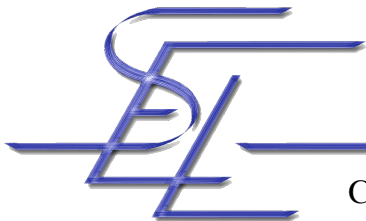


McGill

# Granularity of Use Cases

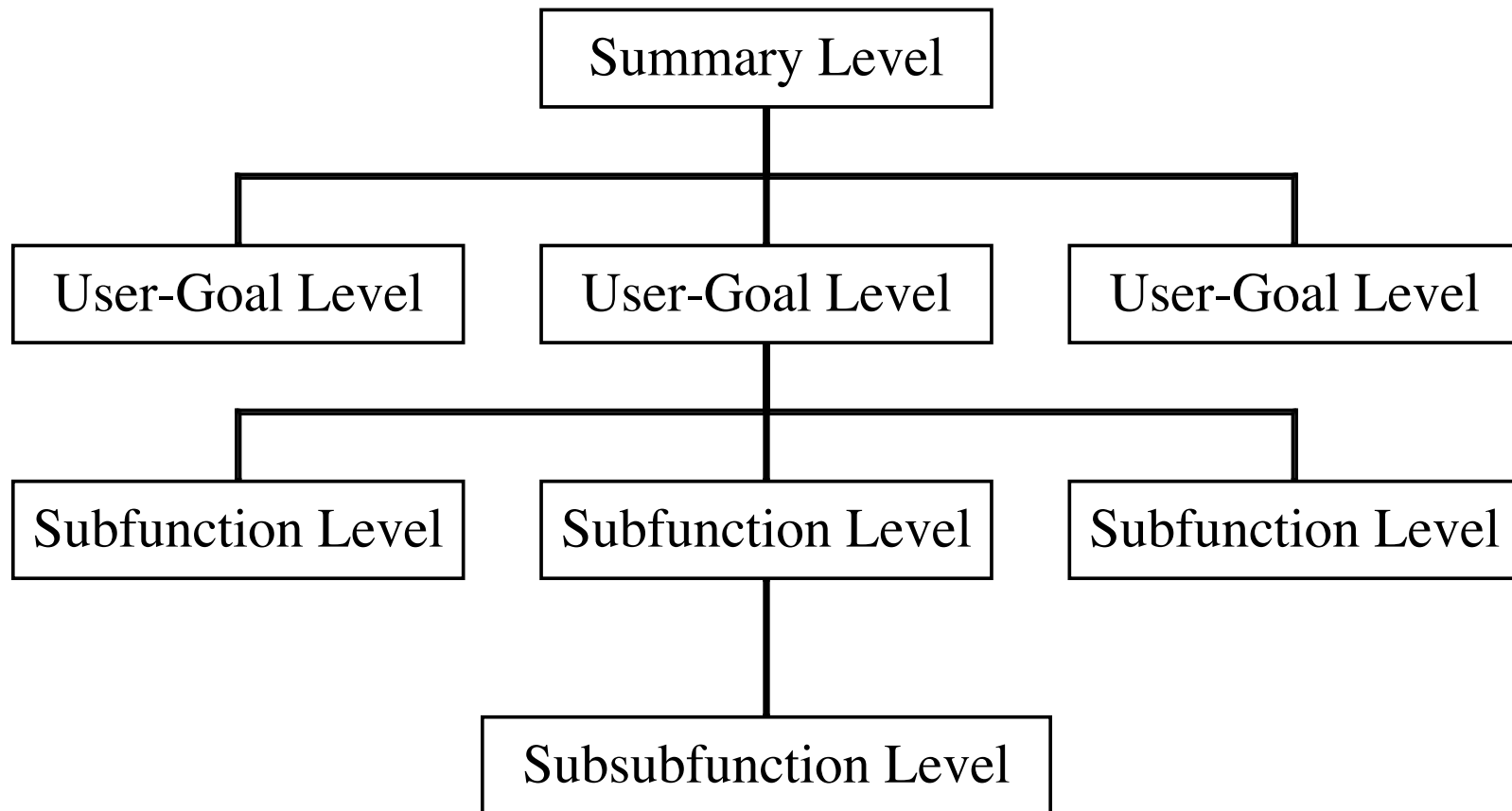
---

- **Summary Level**
  - are large grain use cases that encompass multiple lower-level, user goal use cases; they provide the context (lifecycle) for those lower-level use cases
  - they can act as a table of contents for user goal level use cases
- **User Goal Level**
  - are usually done by one person, in one place, at one time; the (primary) actor can normally go away happy as soon as this goal is completed
  - achieve a single, discrete, complete, meaningful, and well-defined task of interest to an actor
- **Subfunction Level**
  - provide “execution support” for user-goal level use cases; they are low-level and need to be justified, either for reasons of reuse or necessary detail
- The “Interaction Step” at one level of abstraction forms the “Why” for the next level down



# Use Case Hierarchy

---



# Summary Level Example

---

**Use Case:** Manage Funds By Bank Account

**Scope:** Bank Accounts and Transactions System

**Level:** Summary

**Intention in Context:** The intention of the Client is to manage his/her funds by way of a bank account. Clients do not interact with the System directly; instead all interactions go through either: a Teller, a Web Client, or an ATM, which one depends also on the service.

**Multiplicity:** Many Clients may be performing transactions and queries at any one time. Each Client performs its transactions sequentially.

**Primary Actor:** Client

**Main Success Scenario:**

1. Client opens an account.

2. Client identifies with the system.

*Step 3 can be repeated according to the intent of the Client*

3. Client performs task on account:

deposit money, withdraw money, transfer money, get balance.

4. Client closes his/her account.

**Extensions:**

3a. System fails to identify the client; use case continues at step 2.



**McGill**

# User-Goal Use Case Example

---

**Use Case:** Deposit Money

**Scope:** Bank Accounts and Transactions System

**Level:** User Goal

**Intention in Context:** The intention of the *Client* is to deposit money on an account. Clients do not interact with the System directly; instead, for this use case, a Client interacts via a *Teller*.

**Multiplicity:** Many Clients may be performing deposits at any one time.

**Primary Actor:** Client

**Main Success Scenario:**

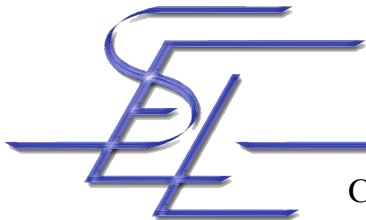
*Client requests Teller to deposit money on an account, providing sum of money.*

1. *Teller requests System to perform a deposit, providing deposit transaction details.*
2. *System validates the deposit, credits account with the requested amount, records details of the transaction.*
3. *System informs Teller that deposit was successful.*

**Extensions:**

2a. *System ascertains that it was given incorrect information:*

2a.1 *System informs Teller about error; use case continues at step 2.*



**McGill**

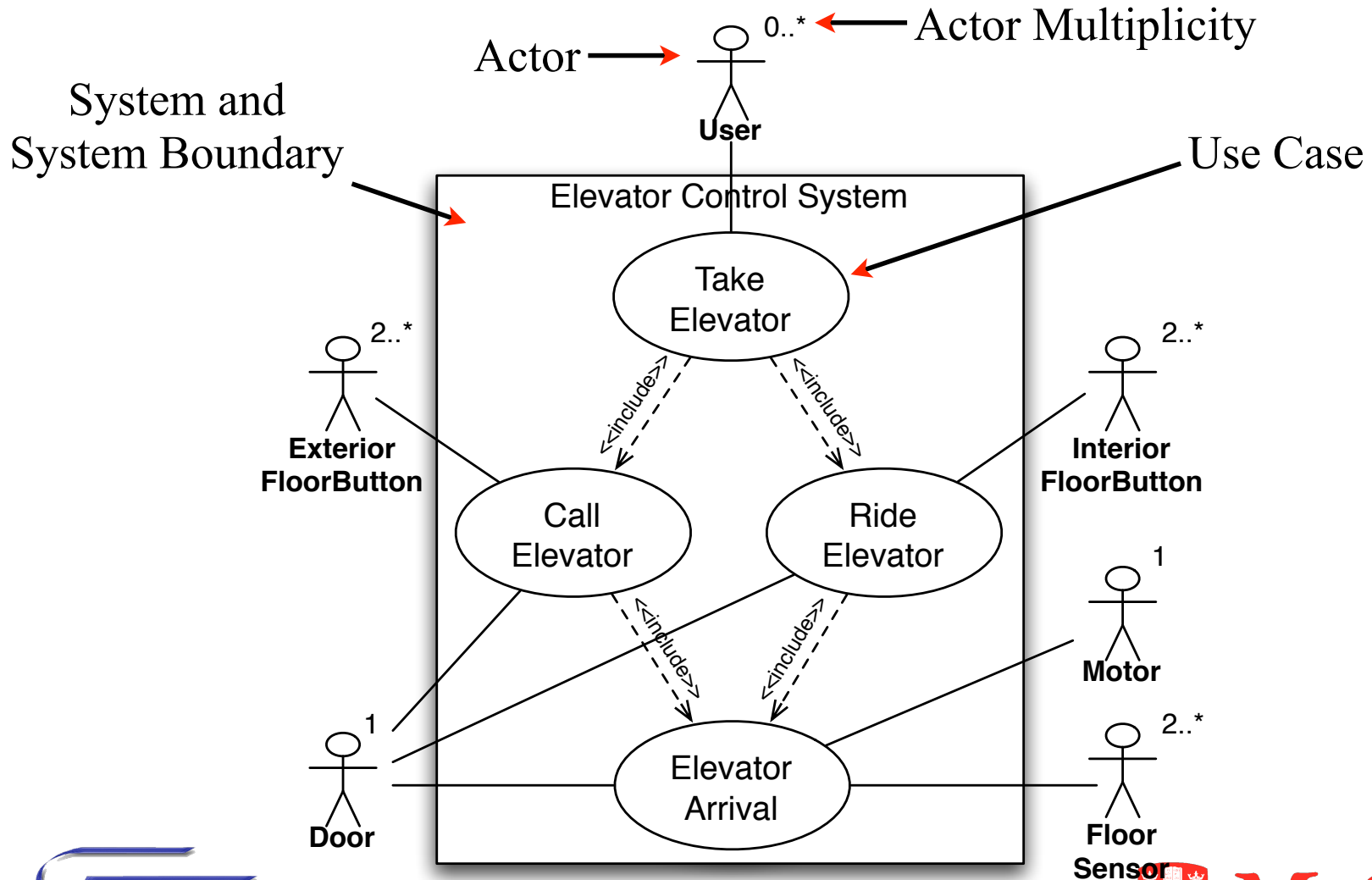
# Use Cases in UML

---

- UML provides a graphical representation for use cases called the *use case diagram*.
- It allows one to graphically depict:
  - actors,
  - use cases,
  - associations,
  - dependencies,
  - generalizations,
  - packages,
  - and the system boundary.



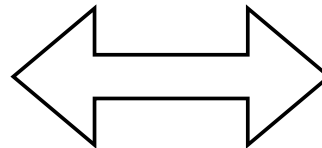
# Example Use Case Diagram



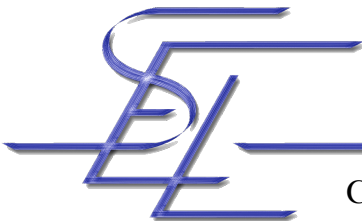
# Use Case Model

---

- A Use Case Model consists of:
  - (at least one) use case diagram and
  - use case descriptions for each “ellipse”



**Use Case:** Open Account  
**Scope:** Bank System  
**Level:** User Goal  
**Intention in Context:** The intention of the Client is to...

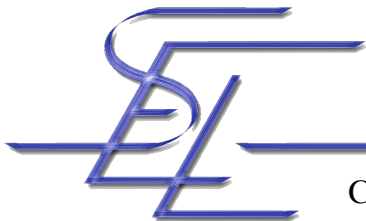
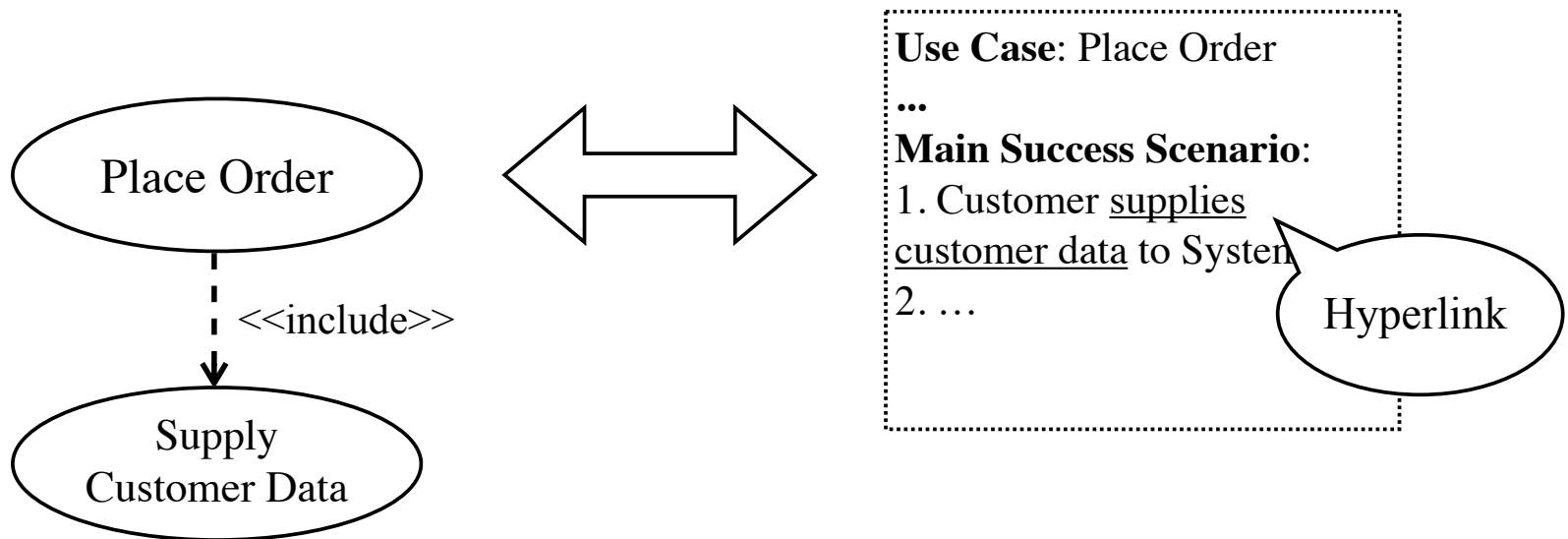


**McGill**



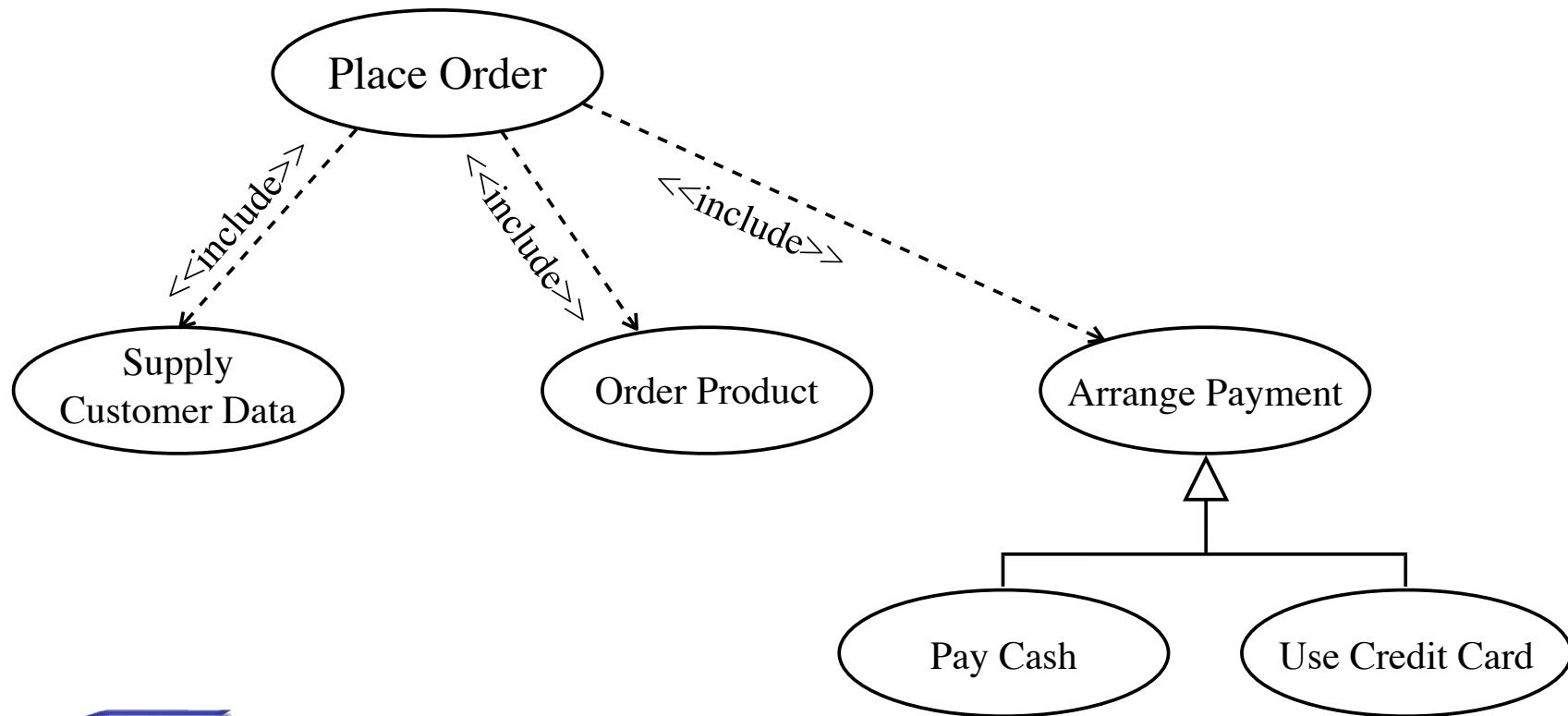
# <<include>> Relationship

- An <<include>> relationship means that the base use case *explicitly* incorporates the behavior of another use case *at a location specified in the base*.



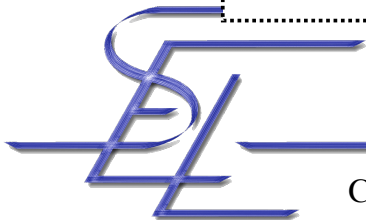
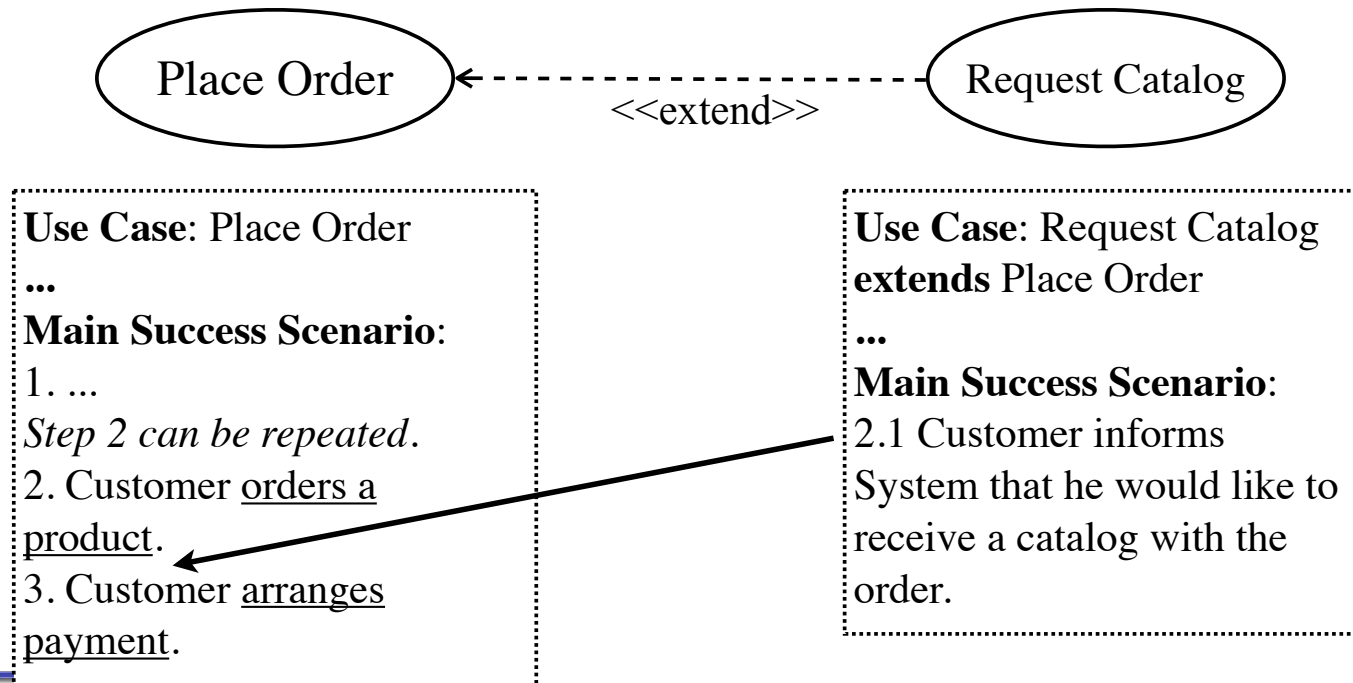
# Generalization/ Specialization

- Explicitly incorporating a “super use case” results in incorporating any child use case



# <<extend>> Relationship

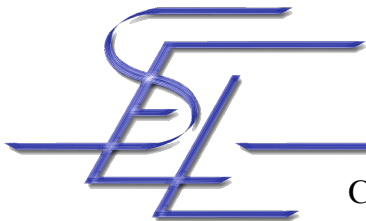
- An <<extend>> relationship means that the extending use case *adds new interaction steps to the base use case at locations specified in the extending use case.*



# Motivation for Dependability-Focused RE

---

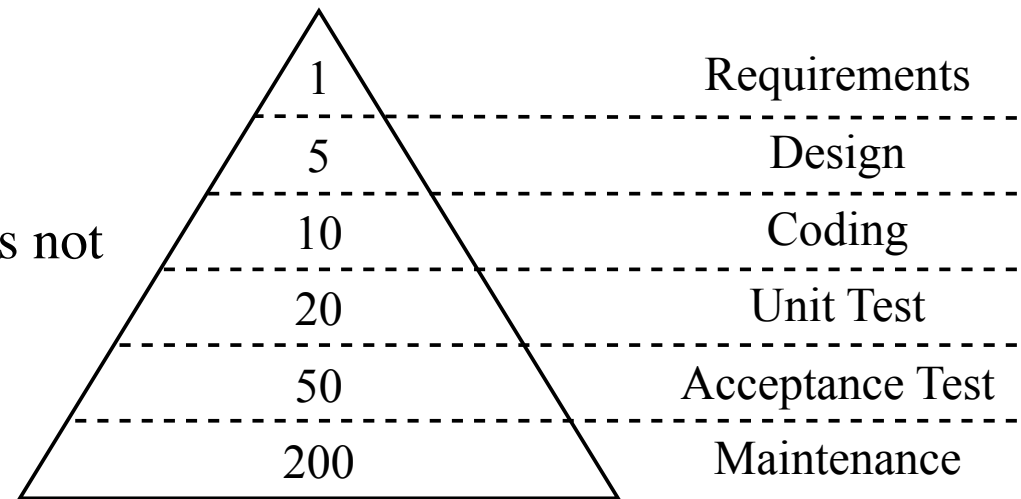
- The major cause of common faults are flawed specifications [Bishop 95]
  - Incompleteness
  - Ambiguity
- Non-identified exceptional situations can lead to
  - Lack of functionality
  - Unreliable system behavior
  - Unexpected system behavior
    - Operation faults
- Idea: extend use case-based requirements elicitation to discover dependability requirements and specify how to deal with exceptional situations



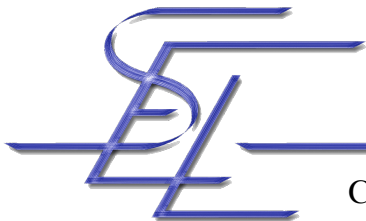
McGill

# More Motivation

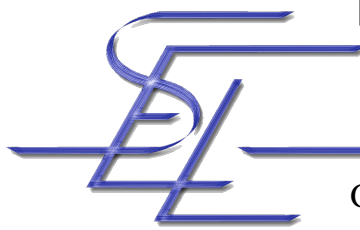
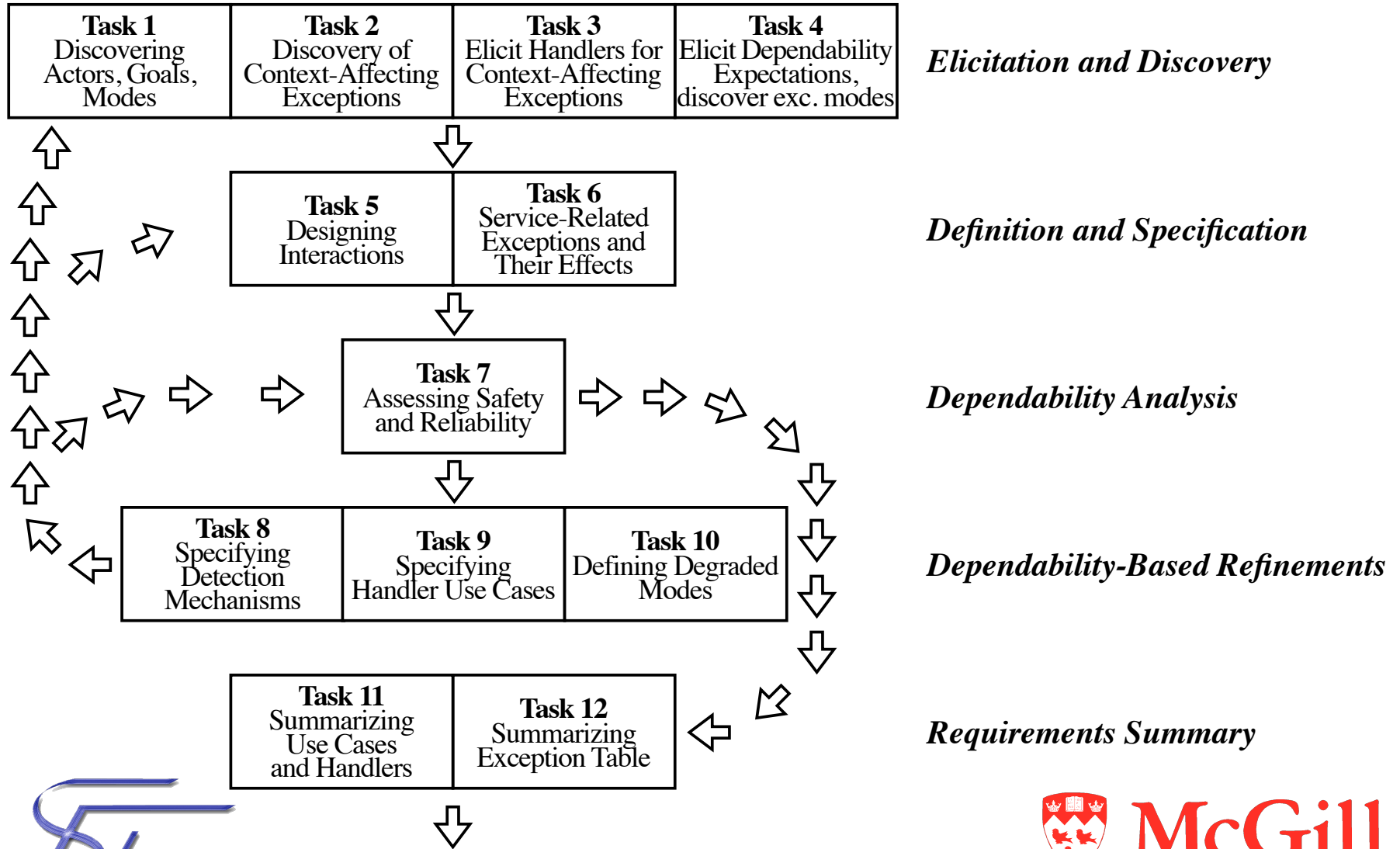
- Faults / omissions made at the requirements stage are expensive to fix later
- Stated requirements might be implemented, but the system is not one that the customer wants
- Need to determine and establish the precise expectations of the customer!



Relative Cost to Repair a Defect  
at Different Lifecycle Phases [Davis 93]



# Process Overview



# Task 1: Discovering Actors, Goals and Modes

---

1.1 Brainstorm services/goals and outcomes

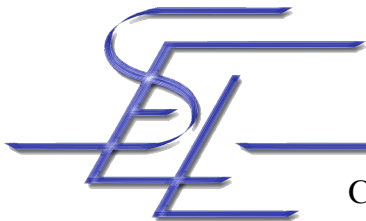
1.2 Brainstorm actors

1.3 Classify services/goals and actors

1.4 Decompose services into subgoals

1.5 Brainstorm operation modes

- An *operation mode* is defined by the set of services that the system offers when operating in that mode
  - Example: cell-phone with child-safe mode



McGill

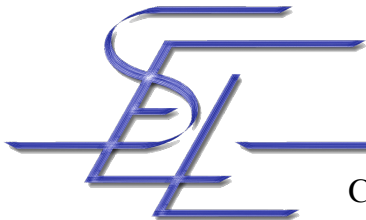
# Task 2: Discovering Context-Affecting Exceptions

---

2.1 Brainstorm context-affecting exceptions

2.2 Define new exceptional detection actors

- Context-Affecting Exceptions
  - Exceptional situation arising in the environment that affect the context in which the system operates
    - Temporary situation or permanent situation
  - Cannot be detected by the system
    - Exceptional actors signal the situation to the system
  - System safety threatened
  - User goals change
- Example
  - Fire outbreak in an elevator, signalled by a smoke detector



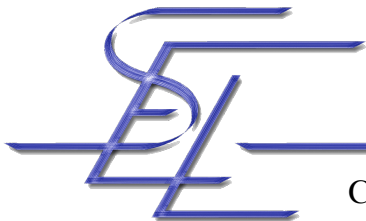
McGill



# Discovering Context-Affecting Exceptions

---

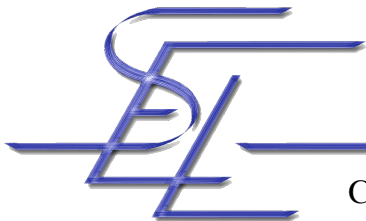
- Discovered in a top-down manner
- System Level
  - What situation prevents the system from being operational?
    - Operational needs: power source, accessibility, connectivity
  - What situation prevents the system from providing safe service? In these situations, should the system provide some other service?
    - Emergencies, safety concerns, malicious behavior
- User-goal Level / Subfunction-level Goal
  - What situations / conditions / changes in the environment prevent the system from satisfying a primary actor's goal (or subgoal)? In such situations, can the system partially fulfill the service?
  - What situations take priority over the primary actor's goal?
  - What situations / conditions / changes in the environment could make the primary actor change his goal? In such situations, how can the primary actor inform the system of the goal change?



# Results of Task 2

---

- For each discovered context-affecting exception
  - Define a name
  - Elaborate a short description describing the situation
  - Identify new system services, i.e. exceptional goals
    - These services are triggered by the occurrence of the exception
  - Exceptional actors
    - Exceptional primary actors detect the occurrence of the exception and signal it to the system
    - Exceptional secondary actors are actors needed by the system to handle the exception



# Task 3: Eliciting Handlers for CA Exceptions

---

3.1 Discover and classify exceptional services

3.2 Decompose exceptional services into subgoals

3.3 Discover new exceptional secondary actors

- For each context-affecting exception, a handler use case outline is defined that describes the exceptional service that is provided by the system, (i.e. how the system is supposed to react in that situation)
  - Handlers are classified as safety or reliability handlers
  - Linked to the context in which they are
- Example
  - Fire outbreak in an elevator, signalled by a smoke detector
    - Safety handler directs elevator cabin down to the ground floor



McGill

# Task 4: Eliciting Dependability Expectations

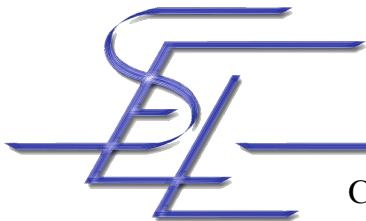
---

4.1 Eliciting dependability expectations for each service

4.2 Document provided reliability and safety of mandatory secondary actors

4.3 Discover exceptional modes of operation

- For each goal / service that the system provides, *expected* safety and reliability is specified
  - Reliability specified with “chance of success”, e.g. 99.97%
  - Safety specified with “chance of safety violation”, e.g. 0.0002%
    - Depending on the application, different safety levels can be defined, e.g. DO-178B
    - This is where discussions on “acceptable risk” should take place among stakeholders



McGill

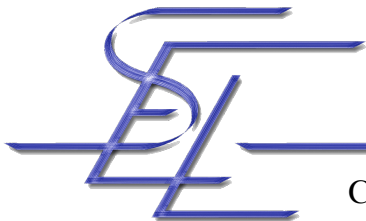
# Exceptional Modes

---

- Dependable systems should not offer services they can not provide in a reliable and safe way
- ➔ When an exceptional situation is encountered, reliability and safety of future service provision should be evaluated
- ➔ If system cannot guarantee dependable service provision, a mode switch is necessary

Operation Mode = Set of Offered Services  
(with defined minimal reliability and safety)

(Emergency Modes, Degraded Modes, etc..)



McGill

# Task 5: Designing Interactions

---

5.1 Design goal interaction steps

5.2 Specify goal outcomes

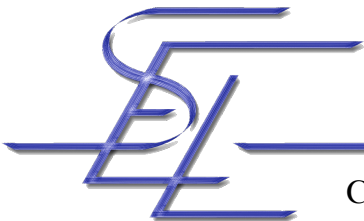
5.3 Define new (exceptional) secondary actors

5.4 Design handler interaction steps

5.5 Specify handler outcomes

5.6 Add mode switches to handler steps, if needed

- Possible goal and handler outcomes
  - <<success>>, <<failure>>, <<abandoned>>, <<degraded>>



McGill

# Elevator Arrival Example

---

**Use Case:** ElevatorArrival

**Intention:** System wants to move the elevator to the User's destination floor.

**Level:** Subfunction

**Main Success Scenario:**

1. *System* asks *Motor* to start moving in the direction of the destination floor.
2. *FloorSensor* informs *System* that elevator is approaching destination floor.
3. *System* requests *Motor* to stop.
4. *System* requests *Door* to open.

Use case ends in <<success>> *FloorReached*.

- Write detailed interaction scenarios for each use case and handler
- Each step is either an *input interaction* or an *output interaction*



McGill

# User Emergency Example

**Handler Use Case:** UserEmergency

**Handler Class:** Safety

**Contexts & Exceptions:** TakeElevator{EmergencyStop}

**Intention:** User wants to stop the movement of the cabin.

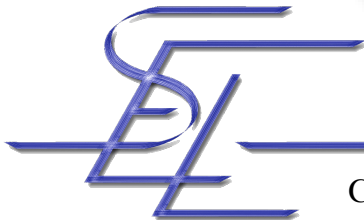
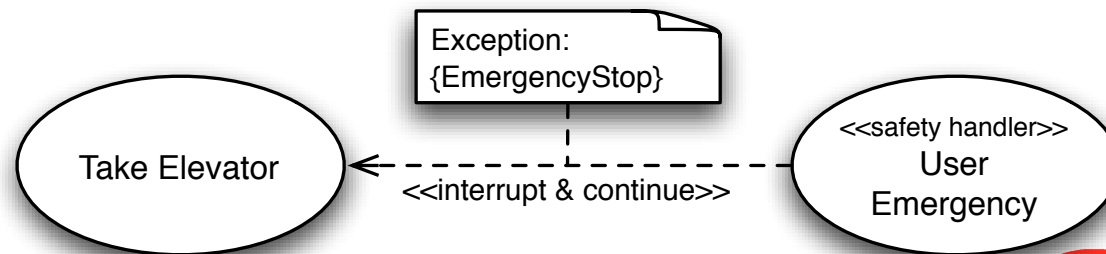
**Level:** User Goal

**Frequency & Multiplicity:** Since there is only one elevator cabin, only one User can activate the emergency at a given time.

**Primary Actor:** User (interacts by means of Emergency Button)

**Main Success Scenario:**

1. *System* initiates Emergency Brake. New Exceptional Facilitator Actor  
*System* clears all pending requests.
3. User informs *System* that emergency is over by toggling the *Emergency Button*.
4. *System* deactivates *Emergency Brakes* and awaits the next request.



McGill



# Fault Assumptions

---

- System (to be built) fault-free
- Faults in the environment
  - Actors fail to provide input to the system
  - Actors fail to provide requested service to system
  - Communication failure
  - Protocol violations
- These situations interrupt the flow of normal interaction that leads to the fulfillment of the user goal



# Task 6: Defining Service-Related Exceptions

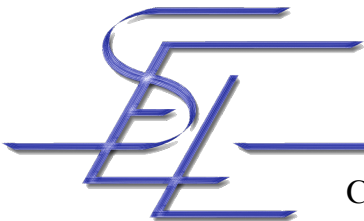
---

6.1 Document expected reliability and safety for actors

6.2 Annotate subgoal and handler steps with reliability and safety

6.3 Define service-related exceptions

- Consider the importance of each interaction step
  - Reliability:  
How essential is the interaction step for the successful completion of the user goal / subgoal?
    - Annotate essential steps with a <<reliability>> tag and specify the success probability, if known
  - Safety:  
Does the failure of this interaction step threaten system safety?
    - Annotate critical steps with a <<safety>> tag and an appropriate safety level
- Consider feasibility of each interaction step
  - Is it possible for the system to be in a state in which the execution of the step is impossible?
  - Are there service-related exceptional situations in which an entire sub-goal cannot be executed?



# Different Source of Problems

---

- Input Problems
  - If omission of input from an actor can cause the goal to fail different options of handling the situation have to be considered.
    - Prompt again after timeout
    - Use default input
    - Temporary system shutdown for safety reasons
- Output Problems
  - Whenever an output triggers a critical action of an actor, then the system must make sure that it can detect eventual communication problems or failure of an actor to execute the requested action.
    - Example: Motor fails to stop.
    - Additional hardware or timeouts might be necessary to ensure reliability.
    - Example: Movement Sensor (exceptional detection actor)



# Results of Task 6

---

- For each discovered service-related exception
  - Define a name
  - Elaborate a short description describing the situation
  - Add exceptions to the exceptions section of the use cases and handlers



# Elevator Arrival Example

---

**Use Case:** ElevatorArrival

**Intention:** System wants to move the elevator to the User's destination floor.

**Main Success Scenario:**

1. System asks Motor to start moving in the direction of the destination floor.

Reliability: 99%

2. Floorsensor informs System that elevator is approaching destination floor.

Reliability: 98% Safety-index: 2 (minor effects)

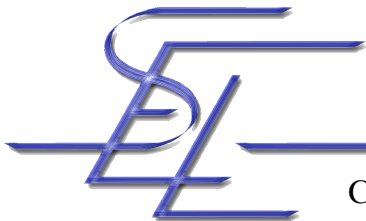
~~3. System requests Motor to stop.~~

Reliability: 99% Safety-index: 4 (catastrophic effects)

~~4. System requests Door to open. Reliability: 97%~~

**Exceptions:**

Exception{MissedFloor}, Exception{MotorFailure},  
Exception{DoorStuckClosed}



Reliability numbers do not reflect reality!



McGill

# Task 7: Dependability Assessment

---

7.1 Map use cases and handlers to DA-Charts

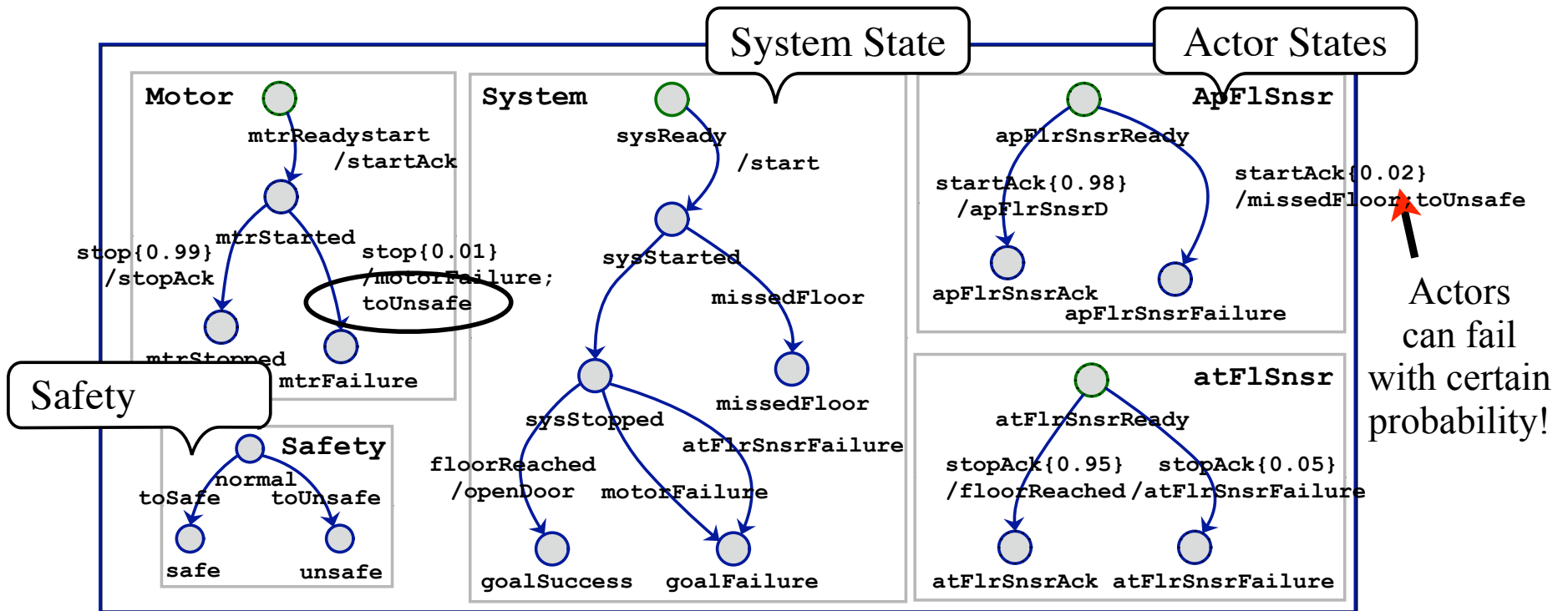
7.2 Perform reliability and safety analysis

7.3 Compare dependability analysis results with expected dependability values

- DA-Chart comprise:
  - A *System* component
    - Input interactions are mapped to events
    - Output interactions are mapped to transition actions
  - One *orthogonal component for each actor*
    - Input interactions are mapped to probabilistic transition actions
    - Output interactions are mapped to probabilistic events
  - A *safety status* component
    - Failed safety-critical interactions trigger *Unsafe* events



# Dependability Assessment Charts



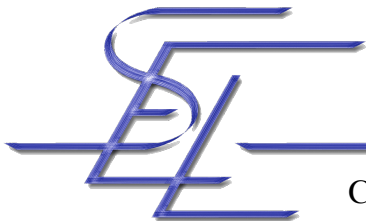
Sequencing according to use case,  
goalSuccess/goalFailure states  
Fault-free - no probabilities



# Tool Support

---

- Tool support for DA-Charts based on AToM<sup>3</sup>
  - DA-Chart support built by extending the state chart meta-model with probabilities
- Analysis done by mapping DA-Charts to Markov chains
  - Safety = Probability to end up in the *Safe* state
  - Reliability = Probability to end up in the *GoalSuccess* state
- Elevator Arrival
  - Safety: 97.02%   Reliability: 92.169
- Careful: These numbers represent “best achievable” safety / reliability, not actual!





# Refining Dependability

---

- What can be done if the calculated dependability is lower than the expected dependability?
- Determine “weak” steps
- Either increase reliability of step
  - Buy better hardware
  - Make communication links more reliable
  - Replicate hardware
  - ➔ No effects on requirements / use case structure
- Or redesign interactions to decrease importance of step
  - Continue with task 8 and task 9



McGill

# Task 8: Specifying Detection Mechanisms

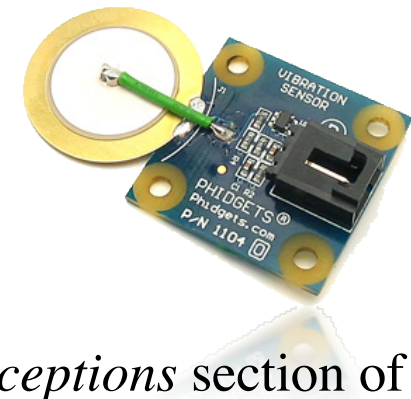
---

8.1 Add detection actors

8.2 Add detection interaction steps for standard use cases and revisit goal outcomes

8.3 Add detection interaction steps for handlers and revisit handler outcomes

- Before recovery measures can be taken, the exceptional situation has to be detected
- Detection might require:
  - Additional secondary actors
  - Additional hardware, so called *detector actors*
    - Sensors
  - Timeouts
- The occurrence of an exception is documented in the *exceptions* section of the use case template



McGill

# Elevator Arrival Example

---

**Use Case:** ElevatorArrival

**Intention:** System wants to move the elevator to the User's destination floor.

**Level:** Subfunction

**Main Success Scenario:**

1. System asks Motor to start moving towards the destination floor.

② FloorSensor notifies System that elevator is approaching destination floor.  
Reliability: 98% Safety-index: 2

3. System requests Motor to stop. Reliability: 99% Safety-index: 4

④ AtFloorSensor informs System that elevator is stopped at destination floor.  
Reliability: 95%

5. System requests Door to open. Reliability: 97%

⑥ DoorSensor notifies System that door is open. Reliability: 95%

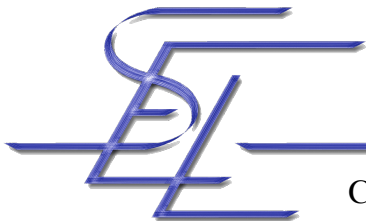
**Exception:**

②a. Exception{MissedFloor}

④a. Exception{MotorFailure}

⑥a. Exception{DoorStuckClosed}

} Very often, timeouts have to be used to detect the exception

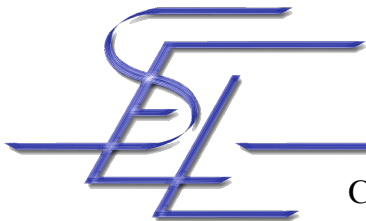


McGill

# Task 9: Specifying Handler Use Cases

---

- Depending on the application domain (and the opinion of the stakeholders), a handler use case performs additional interactions to
  - Continue to provide the original service (reliability handler)
  - Offer a degraded service instead (reliability handler)
  - Take actions that prevent a catastrophe (safety handler)
  - Bring the system to a safe halt (safety handler)
- Behaviour should be intuitive to the people that interact with the system



# Task 10: Defining Degraded Modes

---

- Evaluate the effects of each service-related exception on future service provision
- If promised reliability and safety levels cannot be maintained, a degraded operation mode should be defined
- After completing task 10, the process returns to task 5 (i.e. 5.4 Design Handler Interaction Steps), and then dependability is re-assessed



# Example Refinement: Emergency Brake

**Handler Use Case:** EmergencyBrake

**Handler Class:** Safety

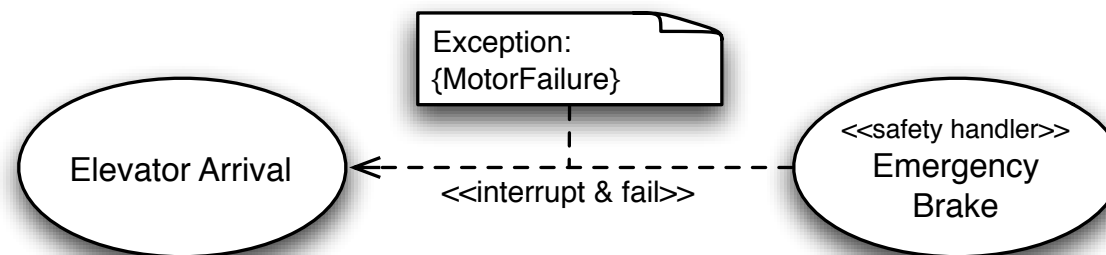
**Context & Exception:** ElevatorArrival{MotorFailure}

**Intention:** System wants to stop operation of elevator and secure the cabin.

**Level:** Subfunction

**Main Success Scenario:**

1. System stops Motor.
2. System activates EmergencyBrakes.  
Reliability: 99.99% Safety-index: 4
3. System turns on the EmergencyDisplay.

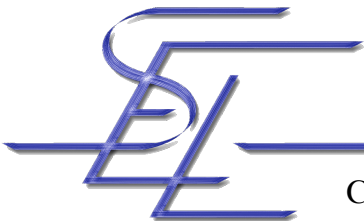
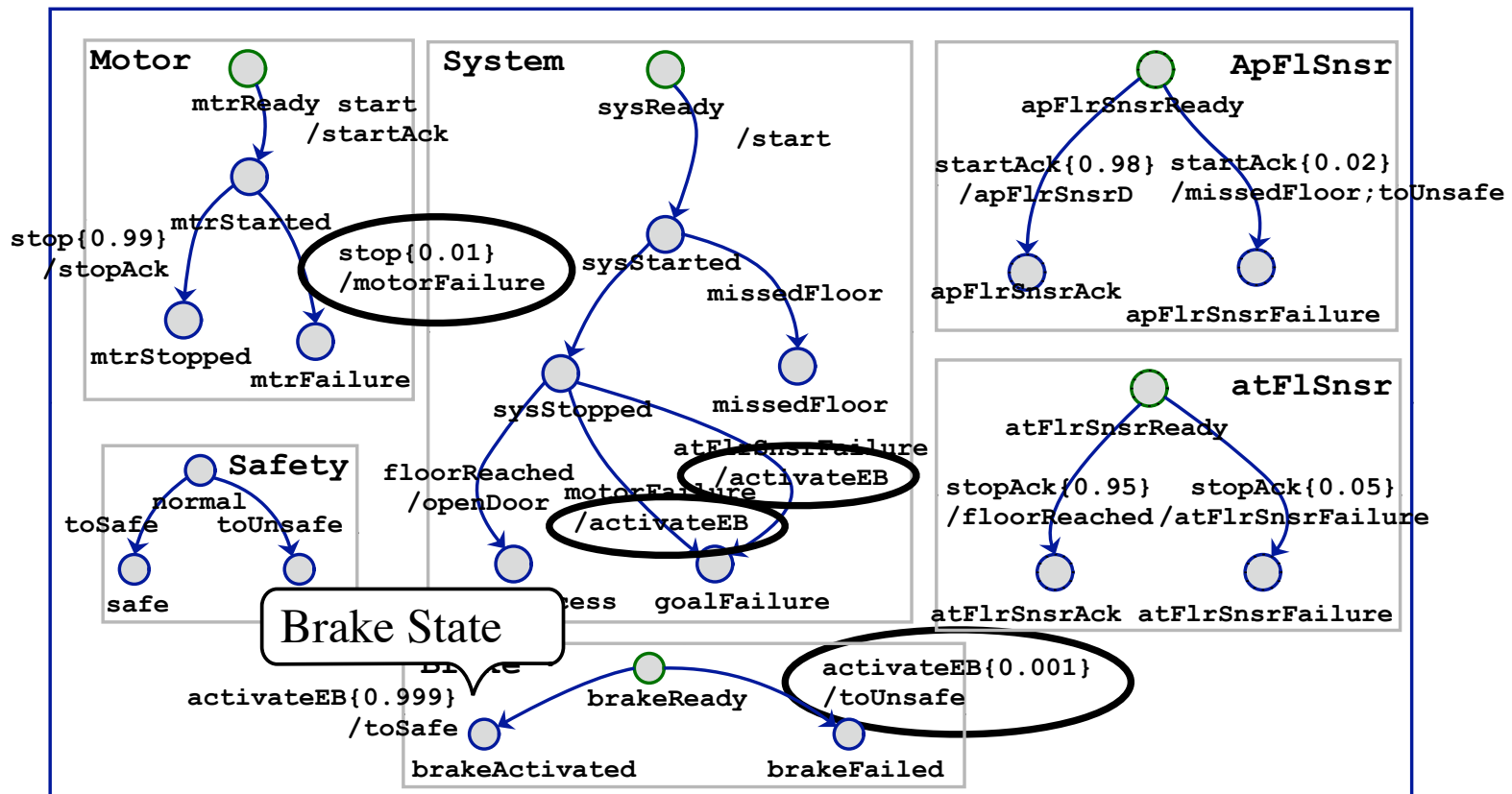


Reliability numbers do not reflect reality!

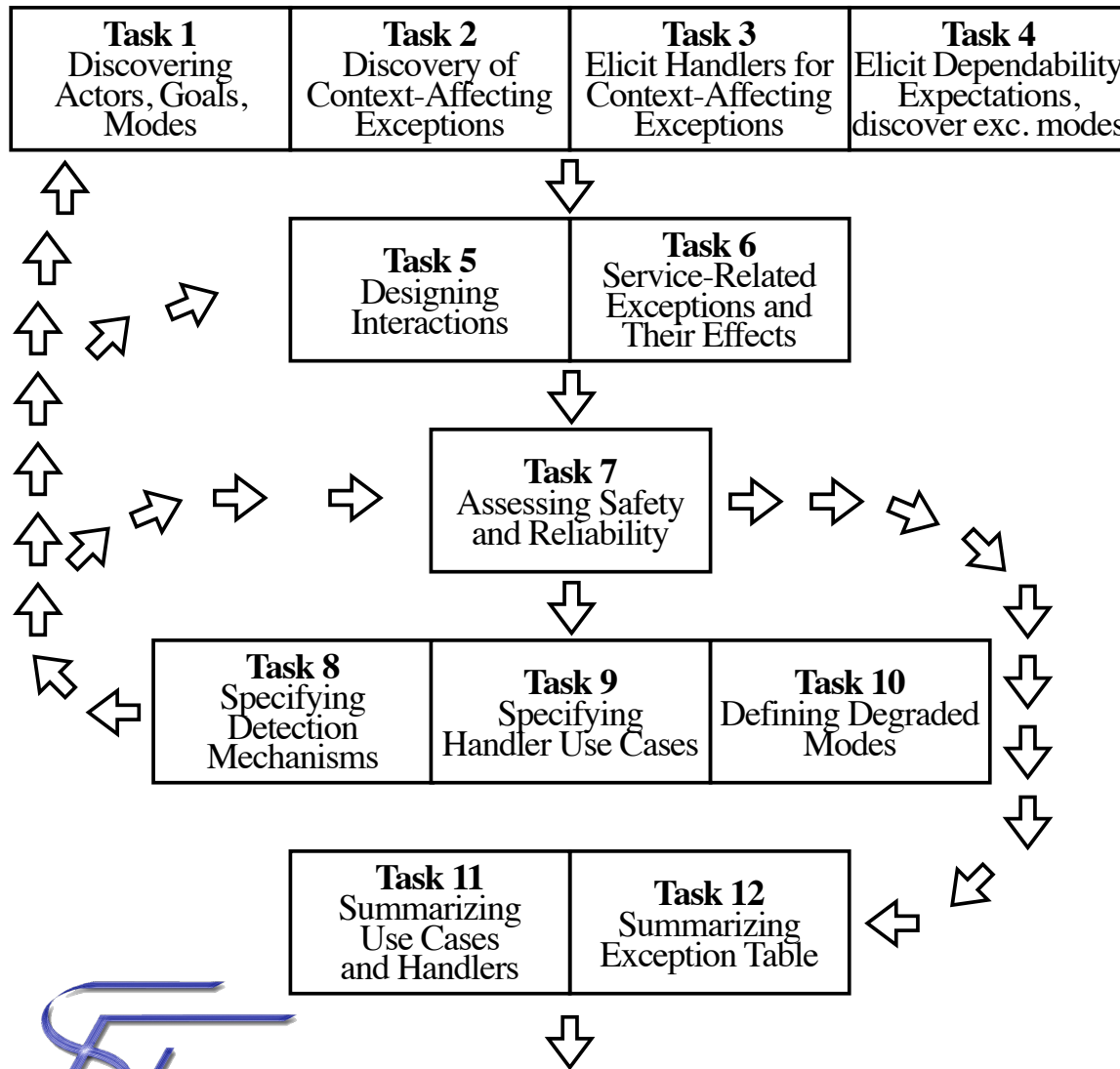


McGill

# Task 7: Dependability Assessment



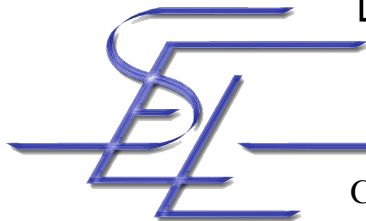
# DREP Overview (again)



When should a developer stop refining?

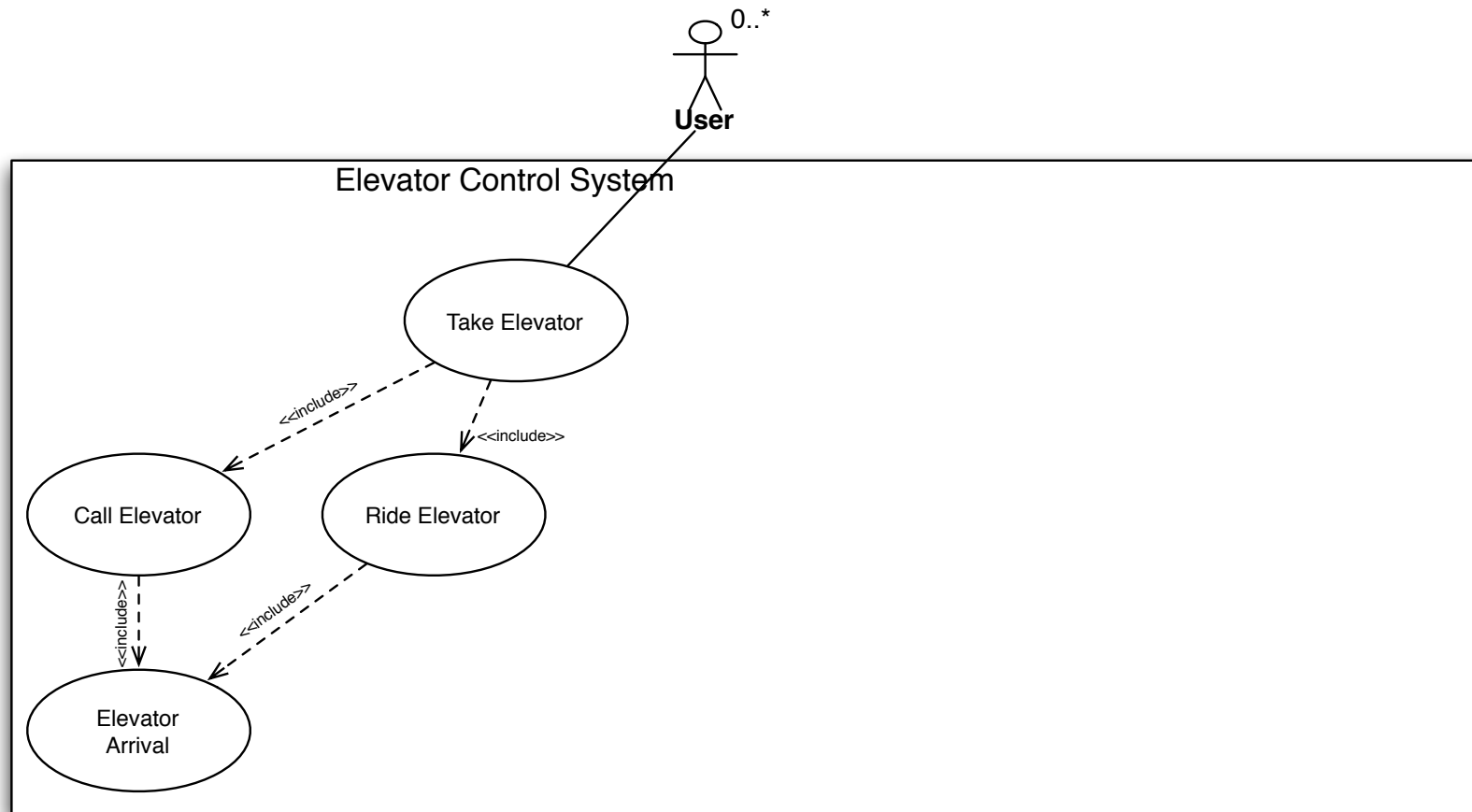
When the assessed dependability is acceptable!

Finally: Build summary use case diagram and exception table





# Task 11: Use Case & Handler Summary

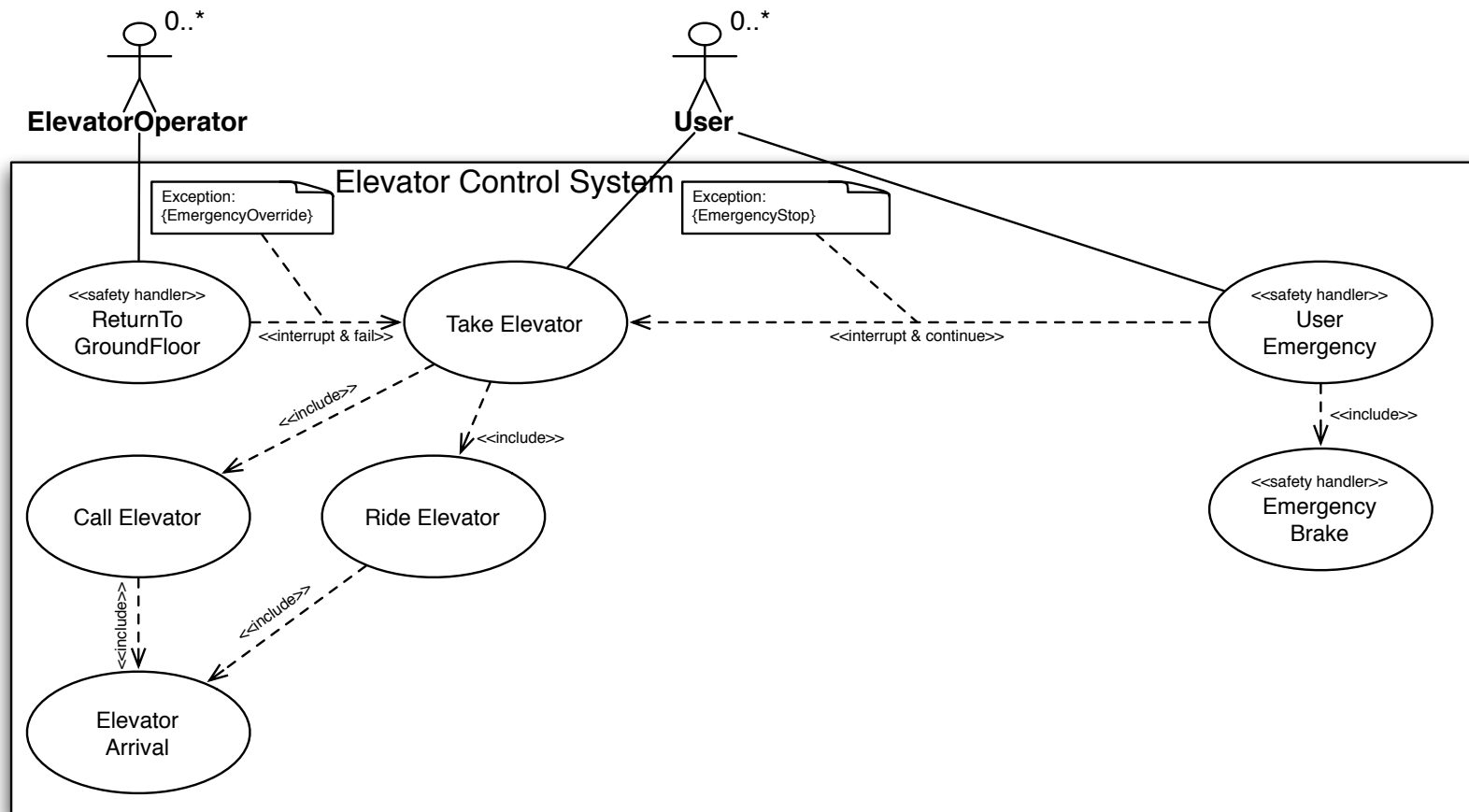


Main Scenario & Alternatives



McGill

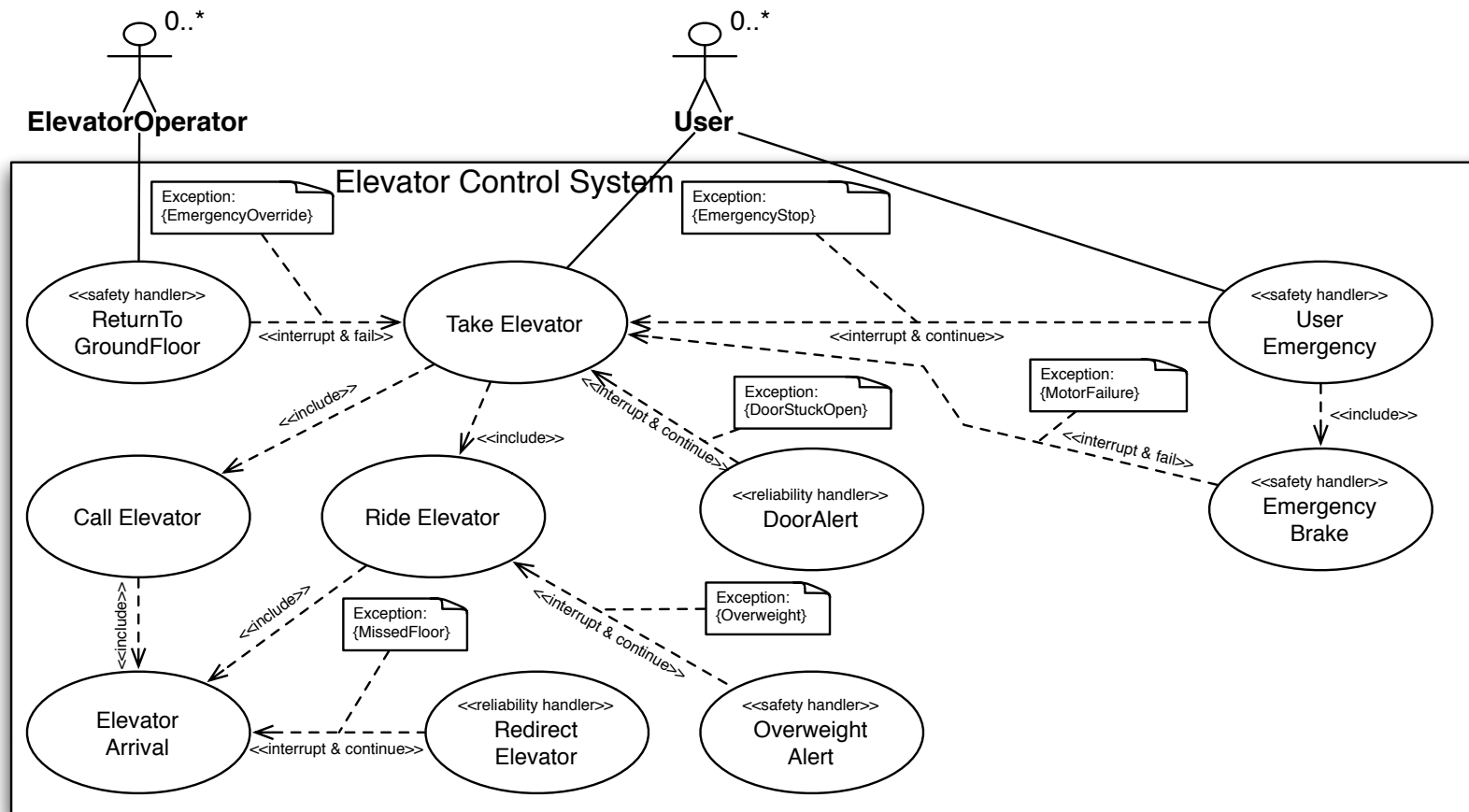
# Use Case & Handler Summary (2)



Environment-related Exceptions  
Environment Handlers



# Use Case & Handler Summary (3)



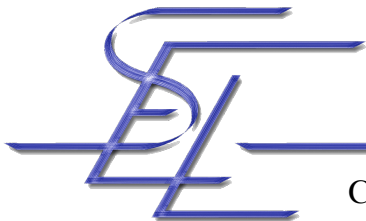
Service-Related Exceptions  
Detection Mechanisms & Handlers





# Task 12: Exception Summary

Exception	Description	Context	Handler	Detection
EmergencyStop	An emergency situation in the elevator cabin makes the User want to stop the elevator	TakeElevator	UserEmergency	Triggered by User actor pressing the emergency button
MotorFailure	Due to a motor or communication failure, the motor does not respond to requests	TakeElevator - or - ReturnToGround Floor	EmergencyBrake	Sensor detects cabin is approaching a floor beyond destination floor - or - timeout expires, and no sensor information has been sent
...				



# Conclusion

---

- Focussing on dependability during requirements engineering is essential
  - Discover the users expectations during exceptional situations
  - Predict achievable dependability before investing in any further development activities
- DREP
  - Dependability-aware Requirements Engineering Process
  - Tasks focus the developer on different aspects of dependability
  - Step-by-step instructions
  - Iterative - guided refinement until dependability is achievable
- Dependability-aware Modeling Notations
  - Separate exceptional from normal behavior
  - Separation enables separate quality control / development / priority
- Tool support



McGill

# Current Work

---

- Complete the RE Process
  - Consider requirements on how to handle faulty design / implementation
  - Establish detailed specification models
- Extend tool support
  - Map exceptional use cases to extended activity diagrams
  - Allow modifications in any formalism
  - Propagate changes back to the textual use case description
- Integrate *timeliness* into the approach
  - Needed to reason about mean time to failure (MTTF), mean time to repair (MTTR), and availability
- Map dependability requirements to architecture / design phase

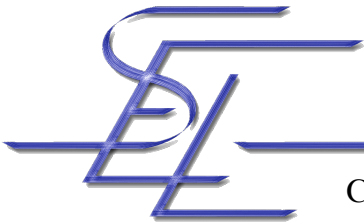


McGill

# SEL Projects Related to FT

---

- Fault Tolerance and Software Development
  - Software Architecture
    - Example: Client-Server, Layered, Event-Based, Pipe & Filter, etc...
    - Error confinement regions for different software architectures, exception propagation, fault tolerance models for architectures
  - Detailed Design
    - Designing with fault tolerance models - “Fault tolerance Design Patterns”
    - Structured Exception Handling
  - Implementation
    - AspectJ Implementation of AspectOptima needs to be aligned with our Reusable Aspect Models
    - Extension of AspectOptima to support exception handling and other extended transaction models
    - Extension of AspectOptima to support look-ahead
    - CaesarJ Implementation of AspectOptima

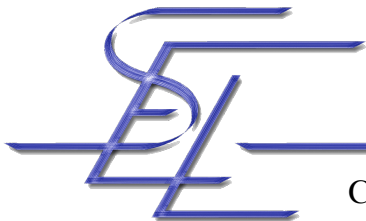




# Our References

---

- [1] Aaron Shui, Sadaf Mustafiz, Jörg Kienzle, Christophe Dony: Exceptional Use Cases. International Conference on Model-Driven Engineering Languages and Systems - MoDELS 2005, Lecture Notes in Computer Science 3713, Springer Verlag, 2005, p. 568-583.
- [2] Aaron Shui, Sadaf Mustafiz, Jörg Kienzle: Exception-Aware Requirements Elicitation with Use Cases. Advances in Exception Handling Techniques, LNCS 4119, Springer Verlag, 2006, p.221 - 242.
- [3] Sadaf Mustafiz, Ximeng Sun, Jörg Kienzle, Hans Vangheluwe: Model-Driven Assessment of Use Cases for Dependable Systems. Proceedings of MoDELS 2006, LNCS 4199, Springer Verlag 2006, p.558 - 573.
- [4] Aaron Shui: Exceptional Use Cases. Master Thesis, McGill University, 2005.
- [5] S. Mustafiz, X. Sun, J. Kienzle, and H. Vangheluwe, “Model-Driven Requirements Assessment of System Dependability,” Software and Systems Modeling, pp. 487 – 502, October 2008.
- [6] S. Mustafiz and J. Kienzle, “A Requirements Engineering Process for Dependable Reactive Systems,” in Methods, Models and Tools for Fault Tolerance (A. Romanovsky, C. Jones, J. L. Knudsen, and A. Tripathi, eds.), no. 5454 in Lecture Notes in Computer Science, pp. 220 – 250, Springer Verlag, 2009.
- [7] S. Mustafiz, J. Kienzle, and A. Berlizev, “Addressing Degraded Service Outcomes and Exceptional Modes of Operation in Behavioural Models,” in International Workshop on Software Engineering for Resilient Systems (SERENE '08), (New York, NY, USA), pp. 19 – 28, ACM, November 2008.



# Other References (1)

---

- Fred D. Davis: User acceptance of information technology: System characteristics, user perceptions and behavioral impacts. *International Journal of ManMachine Studies*, 38(3):475–487, March 1993.
- Bishop, P.: “Software Fault Tolerance by Design Diversity” In M. R. Lyu (ed.), *Software Fault Tolerance*, John Wiley & Sons, pp. 211-229, 1995.
- de Lara, J., Vangheluwe, H.: AToM3 : A tool for multi-formalism and meta-modelling. In: *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering(FASE)*. *Lecture Notes in Computer Science 2306*, Springer 2002, p. 174 – 188.
- A. Cockburn; Structuring Use Cases with Goals. *Journal of Object-Oriented Programming (JOOP Magazine)*, Sept-Oct and Nov-Dec, 1997.
- E. Ecklund, L. Delcambre and M. Freiling; Change cases: use cases that identify future requirements. *OOPSLA '96 - Proceedings of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, 1996. pp. 342 - 358.
- M. Fowler; Use and Abuse Cases. *Distributed Computing Magazine*, 1999. Available at <http://www.martinfowler.com/articles.html>
- M. Glinz; Problems and Deficiencies of UML as a Requirements Specification Language. *Proceedings of the Tenth International Workshop on Software Specification and Design*, San Diego, 2000, pp. 11-22.



# Other References (2)

---

- T. Korson; The Misuse of Use Cases. Object Magazine, May 1998.
- R. Malan and D. Bredemeyer; Functional Requirements and Use Cases. June 1999. Available at <http://www.bredemeyer.com/papers.htm>
- J. Mylopoulos, L. Chung and B. Nixon; Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. IEEE Transactions on Software Engineering, Vol. 23, No. 3/4, 1998, pp. 127-155.
- A. Pols; Use Case Rules of Thumb: Guidelines and lessons learned. Fusion Newsletter, Feb. 1997.
- S. Sendall and A. Strohmeier; From Use Cases to System Operation Specifications. UML 2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, S. Kent, A. Evans and B. Selic (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, 2000, pp. 1-15.
- R. Wirfs-Brock; The Art of Designing Meaningful Conversations. Smalltalk Report, February, 1994.

