Specifying and Designing Exception Handling with FMEA

Tsuneo Nakanishi, Kenji Hisazumi and Akira Fukuda
Faculty of Information Science and Electrical Engineering
Kyushu University
744 Motooka, Nishi, Fukuoka 819-0395, Japan
Email: {tun, nel, fukuda}@f.ait.kyushu-u.ac.jp

***Abstract*—This paper proposes a methodology to specify and design exception classes and exception handling codes used in the `try-catch-finally` exception handling control structure, which is available in C++, Java and similar programming languages. Poorly described specifications of exceptional operations cause ad-hoc, individual dependent use of the `try-catch-finally` exception control structures and fail in poorly designed exception classes and duplicated codes in the exception handling codes. Therefore, the methodology employs HAZOP (hazard and operability analysis) and FMEA (failure modes and effects analysis) to specify the exceptional operations in a consistent manner. HAZOP is used to find failure modes of the specified normal operations and then FMEA is applied to the failure modes to specify their countermeasures (namely, exception handling). Commonality and variability analysis of the specified countermeasures is performed. The result of this analysis is used to design exception classes and exception handling codes, which leads disciplined use of the exception handling control structure and elimination of duplicated codes in exception handling.**

***Keywords*-*FMEA; HAZOP; exception handling; commonality and variability analysis*

I. Introduction

The system required higher safety and reliability must provide valid behaviors, even if it cannot provide the specified normal services due to failures. The system should avoid occurrence of anticipated failures as much as possible. Furthermore, if a failure occurs unfortunately, the system should detect its occurrence and localize, compensate, or mitigate negative effects brought by the failure to minimize the damage.

Such exceptional services are realized in a valid and correct manner through a sound development process, especially in large and complicated systems. Exceptional services must be analyzed, specified, designed, implemented and then tested, as normal services are realized so. Apart from distinguishing normal and exceptional services explicitly in development, system behaviors on failures are usually specified to a greater or lesser extent during requirements and specifications phase. However, it is impossible to identify failures sufficiently that will occur at the component level, since the system is not decomposed at all at this phase. As the system is refined and decomposed in later phases, more design decisions are made and more failure modes become visible. Countermeasures to the failure modes identified in later phases must be studied and specified. Their specifications must be integrated in the system specification as exceptional services. We should keep it in mind that two thirds of system failures are due to design faults hidden in exception handling that occupies over two thirds of the system [1].

Exceptional services tend to be specified insufficiently in immature development sites. They are often decided individually by designers or programmers. That brings duplicated and/or irregular design of exceptional operations as well as a considerable amount of development rework. Furthermore, that will cause large scale modification in exceptional operations in case we enhance the existing system with additional functions.

The similar problem occurs also in implementation phase. The `try-catch-finally` statement is an exception handling control structure available in C++, Java and other similar programming languages. This exception handling control structure contributes to increase readability and reusability of exception handling codes as long as its usage is well disciplined. However, if exceptional operations are not specified and designed in a systematic manner, the exception handling control structure tends to be used in an ad-hoc, individually dependent manner. Sometimes, exceptions are ignored without taking responsible actions, although they should be processed or transferred to the caller. Absence of comprehensive view on exception handling fails in distribution of code clones doing the almost same but a little bit different things in the exception handling control structure. Moreover, poorly designed exception classes make exception handling chaotic.

This paper presents a methodology to specify exceptional operations at the class member function level and defines exception classes for the `try-catch-finally` exception handling control structure. The proposed methodology employs HAZOP (Hazard and Operability Analysis) [2] and FMEA (Failure Modes and Effects Analysis) [3]. HAZOP is a risk analysis method to identify risks to the system and its stakeholders brought by the system under consideration when a concerned property deviates from its intended extent. FMEA is a failure analysis method to study countermeasures to failures of the system under consideration. The proposed method identifies failures of the normal operation with HA-

ZOP and then, studies countermeasures to the failures with FMEA. FMEA establishes comprehensive view on exception handling and helps consistent and disciplined design of exception handling. Moreover, the proposed methodology performs commonality and variability analysis of the countermeasures, which the authors believe the novel idea in the field of exception handling. Commonality and variability analysis is a technique that has been commonly performed in software product line engineering [4] to separate common and variable structures and behaviors among products. Introduction of this idea into exception handling contributes to eliminate duplicated codes in exception handling codes for different exceptions and design well-structured exception classes.

The paper is organized as follows: Section 2 describes HAZOP and FMEA and its application to software. Section 3 gives a brief description on the `try-catch-finally` exception handling control structure of C++, Java and similar programming languages. Section 4 presents the proposed methodology with an example. Section 5 describes related works. Section 6 concludes the paper.

II. HAZOP and FMEA

HAZOP [2] is a risk analysis method to identify possible risks to the system and its stakeholders, which was originally used in chemical process engineering. In HAZOP, risks are identified by drawing up hazardous scenarios caused when a concerned property of the system such as temperature, pressure, velocity *etc.* deviates from its intended extent. Guide words are applied to the properties to facilitate imagination of hazardous scenarios; *more*, *less*, *none*, *reverse*, and *other than* are examples of the guide words. Countermeasures to the hazardous scenarios are studied. The result of HAZOP is summarized in the tabular format.

FMEA [3] is a failure analysis method used to assess and improve reliability and safety of the system. FMEA is so generic that it has been used (maybe, more than HAZOP) in various industries such as aviation, space, automotive, nuclear *etc.* to develop and operate safety critical systems for several decades.

In FMEA, various stakeholders of the system come together; identify failure modes for each component of the system; analyze what negative effects will be brought to the component, the subsystem, and/or the system for each failure mode; and study countermeasures to compensate or mitigate the effects. Moreover, for each failure mode of the component, the stakeholders evaluate criticality of the negative effects and, based on the evaluation, prioritize the countermeasures to be realized. The criticality is basically evaluated in terms of probability of failure mode occurrence and severity of the negative effects. FMEA with this probability and severity evaluation is sometimes referred to as FMECA (Failure Modes, Effects, and Criticality Analysis). (See [5], [6], for some methods of criticality evaluation in FMECA.) The result of FMEA is also summarized in the tabular format.

Since FMEA is inherently a bottom-up analysis method starting from failure modes of the component, it may seem impossible to apply FMEA to the system extent until the system is completely decomposed into the components. However, it is absolutely unreasonable for large and complicated systems to perform FMEA and modify the system to increase reliability and safety after the system is completely decomposed. That will force us to abandon a large part of detailed design artifacts, or require a huge amount of development rework to satisfy non-functional requirements such as performance or for other reasons. Therefore, FMEA has evolved from a simple, component oriented method toward a process oriented method that performs analysis at various granularity of system decomposition along with stepwise refinement of the system. That is, FMEA is applied to each subsystem after decomposition of the system first; design of the system is improved at the subsystem level based on its result; the subsystems are decomposed into components; FMEA is applied to each component of the subsystem to improve the design of the subsystem in the similar manner.

FMEA is sometimes time-consuming. However, negative effects to the system brought by failure modes are not fully diverse; rather, we can observe a considerable amount of duplication. It is possible to deal with multiple failure modes having the identical negative effect as a group. This group is referred to as *Fault Equivalent Class* [7]. This concept contributes to reduce duplicated works in analysis and the scale of FMEA. It is reported that 12,401 failure modes are shrinked into 1,759 failure equivalent classes in the case study of the cabin management system of Boeing 777. The failure modes obtained in FMEAs of different abstraction levels are also grouped in the same failure equivalent class if their negative effects to the system are identical. That enables partial reuse of FMEA results performed in earlier phases in later phases.

HAZOP and FMEA are used for similar objectives and their results are summarized in similar tabular formats. The proposed method uses the “deviation” concept and the idea of guide words of HAZOP to identify failures of the normal operation. Countermeasures to the failures are studied with FMEA, not with HAZOP.

III. Exception Handling Control Structure

The proposed methodology assumes use of the `try-catch-finally` exception handling control structure used in C++, Java and other similar programming languages. This section is dedicated to remind the readers of exception handling control structure.

These programming languages have the exception handling control structure of the form shown below:

```
try {
  ...
}
catch (ExClass formalExInstance) {
  ...
}
finally {
  ...
}
```

Normal operations are implemented in the `try` block, exceptional operations are implemented in the `catch` block, and clean-up operations are implemented in the `finally` block. Each `try` block can follow one or more `catch` blocks with different exception classes. The `finally` block is optional. `try`, `catch` and `finally` blocks can include another `try-catch-finally` block.

If it is impossible to continue execution of the `try` block or its callee functions, `throw` statement shown below should be executed:

```
throw actualExInstance;
```

Execution of the `throw` statement terminates execution of the `try` block. A `catch` block with the formal instance (`formalExInstance`) of an exception class (`ExClass`), which is compatible to the exception class of the thrown actual instance (`actualExInstance`), is responsible for the exception issued by the `throw` statement. The processor tries to find such a `catch` block out of the `catch` blocks of the current `try` block. If it is not found, the processor tries to find a `catch` block to be executed out of the `catch` blocks of another `try` block containing the current `try` block directly. The processor continues this backward traversal of nested `try` blocks recursively until it finds the `catch` block to be executed. Furthermore, if the `catch` block to be executed is not found in the function, the processor try to find it out of the `catch` blocks of the `try` block containing the current invocation point in the caller. The processor performs this backward traversal of function calls recursively, whenever there is no more `try` block to be checked in the function. After the proper `catch` block is found and executed, the processor transfers its execution to the point immediately after the exception handling structure having the executed `catch` block and the stack frames for the terminated `try` blocks and functions are released. The `finally` block is executed whenever control leaves the exception handling structure to which it belongs, regardless of whether a `catch` block is executed or not.

The actual instance of the exception class specified in the `throw` statement (`actualExInstance`) can be referenced in the corresponding `catch` block as its formal instance (`formalExInstance`). Therefore, the exception class is used not only to distinguish exceptions but also to pass data and control information required for exception handling by embedding them as its attributes.

IV. Specifying and Designing Exception Handling with FMEA

In this section, we propose the methodology to specify and design exception handling with FMEA. The methodology assumes that the system has already been decomposed into classes and the classes have already been designed for normal operations.

The methodology is described below in a stepwise manner with an example of the login form of the graphical user terminal. A user inputs his/her name and password in the text fields and then presses the login button on the form shown in Figure 1 to use the terminal. A member function `Login()` of class `LoginForm` is called on the login button press. The member function inquires of the user DB if the input name and password are correct by calling a member function `Auth(username, passwd)` of class `UserDB`. If they are correct, the member function closes the form and notifies its caller that the login is accepted. Otherwise, the member function waits user's further input of his/her name and password without closing the form or notifies its caller that the login is failed with closing the form.

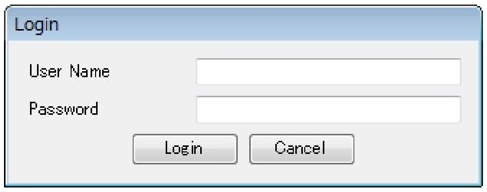

Figure 1. Login Form

A. Describing Normal Specifications

First, we describe the normal specification of each member function which are identified in class design. The normal specification of a member function is a sequential description of its internal operations from invocation to termination where the function successfully provides its service specified in the class specification. The steps must be in even granularity. The step should be in active verbal form, namely "do ...", and should not include multiple operations. The step can include conditional or iterative operations. A pseudo-code describing only normal operations of the member function can be dealt with a normal specification.

Normal specifications of `UserDB.Auth(username, passwd)` and `LoginForm.Login()` are shown in the *Normal Operation* column of Tables I and II, respectively.

B. Deriving Failure Modes

Second, we apply the methodology deriving failure modes, which was proposed by the authors [8], to normal specification steps in active verbal form "do ..." and look for failure modes of each step by imaging cases where the step

Table I
FMEA RESULTS ON `UserDB.Auth(username, passwd)`

Normal Op.	Failure Modes/Causes	Detection	Effects	Countermeasures	Not. to Callers
1) Checking if the user DB is connectable.	The user DB cannot be accessed.	The handle for user DB access is NULL.	I: User authentication is impossible.	**Run-Time:** Throw an exception.	The user DB is not accessible.
2) Reading the password of the user specified by the user name from the user DB.	Reading of the user DB is failed. / The user DB is locked.	The return value and the error code from the DB library	I	**Run-Time:** Throw an exception.	The user DB is locked.
	Reading of the user DB is failed. / Other reasons	The return value and the error code from the DB library	I	**Run-Time:** Throw an exception.	The user DB is unreadable for a certain reason.
	The specified user is not found.	The number of the matched records is zero.	I	**Run-Time:** Throw an exception.	The specified user is not found. + The specified user name
	Two or more specified user is found.	The number of the matched records is equal to or greater than two.	II: The user DB is logically corrupted. Further accesses may destroy the user DB completely and prohibit even partial recovery.	**Run-Time:** Throw an exception.	The specified user is found too much. + The specified user name
3) Checking if the password is correct.	The password is not correct.	The password is not equal to one in the user DB.	I	**Run-Time:** Terminate the function without authenticating the user.	Return value: false
4) End user authentication.			(The user is authenticated successfully.)	**Run-Time:** Terminate the function with authenticating the user.	Return value: true

cannot be accomplished successfully. The methodology assumes abnormal deviation of properties of objects, relationships among objects and behaviors as failures. Four HAZOP guide words *no*, *less*, *more* and *other than* are applied to the properties to derive failure modes by their deviation. To derive failure modes on behaviors, the methodology examines properties common to all the behaviors and specific to each behavior. The common properties are beginning time, ending time, duration, frequency, rate, interval, order *etc.*

The second normal operation step of `UserDB.Auth(username, passwd)`, "Reading the password of the user specified by the user name from the user DB." has a behavior *read* and objects *password*, *user*, *user name* and *user DB*. The HAZOP guide words are applied to their properties. The properties of the behavior *read* are the common ones mentioned above as well as *success or fail* and *the number of the records* which are specific to *read*. The properties of the objects *user* is *the number of the users* and the properties of the object *password* are *the number of the passwords* and *length*. Table III shows candidates of failure modes found by HAZOP application to these properties.

They are all candidates of failure modes. The candidates which are actually hazardous are adopted as the failure modes and listed in the *Failure Modes/Causes* column of the FMEA table after adjustment of their terms.

Countermeasures can be different depending on the cause of the failure. Therefore, if the abstraction level of a derived failure mode is too high to find a single countermeasure for the failure mode, we apply FTA [9] to the failure mode to identify its causes and study countermeasures depending on the causes. For example, in Table I, causes of the failure mode "Reading of the user DB is failed." are analyzed. A couple of causes, "The user DB is locked." and "Other reasons", are found and the countermeasures for them are studied separately. (The countermeasures for the both causes are same, namely, "Throw an exception." However, note that exception objects representing different meanings are thrown.)

If the function calls other functions, the contents of the *Notification to Callers* column must be duplicated in the *Failure Modes/Causes* column. For example, the failure

Table II
FMEA RESULTS ON `LoginForm.Login()`

Normal Op.	Failure Modes/Causes	Detection	Effects	Countermeasures	Not. to Callers
1) Getting the user name from the *User Name* field.	The user name is not specified.		I	**Run-Time:** i) Displaying a message telling the user name is not specified; ii) Clearing the *User Name* and *Password* fields; iii) Focusing the *User Name* field; iv) Continuing the login form.	Nothing.
	Invalid characters are used in the user name.		III: The user DB will reject login at Step 3.	(Unnecessary)	Nothing.
	The user name is too long.		III	**Dev. Time:** Set the maximum length of the user name field to the maximum length of the user name.	Nothing.
2) Getting the password from the *Password* field.	The password is not given.		IV: No problem because some users may not set their passwords.	(Unnecessary)	Nothing.
	Invalid characters are used in the password.		III	(Unnecessary)	Nothing.
	The password is too long.		III	**Dev. Time:** Set the maximum length of the password field to the maximum length of the password.	Nothing.
3) Asking the user DB if login is possible.	The user DB is not connected.	Exception from UserDB.Auth()	I	**Run-Time:** i) Displaying a message that the user DB is unavailable; ii) Terminating the login form.	Nothing.
	The user DB is locked.	Exception from UserDB.Auth()	I	**Run-Time:** i) Displaying a message telling that the user DB is locked; ii) Focusing the login button; iii) Continuing the login form. (Because the user DB may be unlocked later.)	Nothing.
	The user DB is not available for other reasons.	Exception from UserDB.Auth()	I	**Run-Time:** i) Displaying a message telling that the user DB is unavailable; ii) Terminating the login form.	Return value: false
	No user is found.	Exception from UserDB.Auth()	I	**Run-Time:** i) Displaying a message telling that the user is unknown; ii) Clearing the *User Name* and *Password* fields; iii) Focusing the *User Name* field; iv) Continuing the login form.	Nothing.
	Two or more users are found.	Exception from UserDB.Auth()	II	**Run-Time:** i) Displaying a message telling that the user DB is logically corrupted; ii) Terminating the login form.	Return value: false
	Password is not correct.	Return value from UserDB.Auth()	I	**Run-Time:** i) Displaying a message telling that password is incorrect; ii) Clearing the *Password* field; iii) Focusing the *Password* field; iv) Continuing the login form.	Nothing
4) Terminating the form.			(The user accomplishes login successfully.)		Return value: true

Table III
HAZOP GUIDE WORDS APPLICATION TO "READ" (UPPER), "USER" (MID) AND "PASSWORD" (LOWER)

Properties	Type	No Guide Word	Applied Guide Words			
		ϕ	*no*	*less*	*more*	*other than*
beginning time	time instant	Start reading at right timing	Do not start reading	Start reading too early	Start reading too late	/
ending time	time instant	End reading at right timing	Do not end reading	End reading too early	End reading too late	/
...	...	...	...	...	...	...
success/fail	boolean	Read successfully	Fail in reading	×	×	×
# of the records	number	Read a right number of the records	Read no record	Read too few records	Read too many records	/
# of the users	number	A right number of the users	No user	Too few users	Too many users	/
# of the passwords	number	A right number of the passwords	No password	Too few passwords	Too many passwords	/
The length of the password	number	A right length of the password	Null password	Too short password	Too long password	/
...	...	...	...	...	...	...

modes at the third normal operation step of the operation `LoginForm.Login()` in Table II are from the *Notification to Callers* column of the `UserDB.Auth(username, passwd)` in Table I.

C. Assessing Each Failure Mode

Third, we assess each failure mode; study how to detect the failure mode, how the failure mode brings negative effects, how to devise countermeasures against the failure mode, and what kind of information should be given to the caller; and complete the FMEA table. The FMEA table for the proposed methodology has, in addition to *Normal Operation* and *Failure Modes/Causes* columns, *Detection*, *Effects*, *Countermeasures* and *Notification to Callers* columns described below.

Detection: If the failure mode can be detected in the member function, we describe how to do it briefly.

If the failure mode can be detected in the callee of the member function, we describe how the callee notifies the member function that the failure mode is detected. See the *Notification to Callers* column described below for the details.

Effects: We describe how the failure mode brings negative effects with grouping them into fault equivalent classes described in Section 2 and assigning a sequential number referred to as *Fault Identifier Number* (*FIN*). Negative effects which are previously described in earlier and current phases are referenced by FINs, instead of describing the same thing twice or more. In the *Effects* column of Tables I and II, negative effects appeared first time are described with new FINs in the Roman numeral, however, ones appeared previously are just referenced by their FINs.

Countermeasures: We describe countermeasures to the failure mode, namely, how to avoid the failure mode and/or how to localize, compensate or mitigate the effect brought by the failure mode.

Various types of countermeasures are possible. Development time countermeasures are ones taken before release of the product, while run-time countermeasures are ones taken after release of the product. Some of run-time countermeasures are executed as exception handling.

Note that the countermeasures must be subject to the specification on the exceptional behaviors specified in earlier phases if they are available.

Notification to Callers: We describe manners and contents of notification to the caller when this member function encounters the failure mode. This notification is not mandatory and should not be performed in vain to keep information hiding and loose coupling. The notification is needed in the following cases:

- This member function cannot manage the failure mode within its specified responsibility.
- The caller of this member function can provide worthful actions to localize, compensate, or mitigate the effect of the failure mode.
- Expected reactions to the failure mode can be different depending on the caller of this member function.

There are some manners to notify the caller of failure mode occurrence such as the return value or reference parameters of the function, exceptions thrown by the function, global variables, invocation of prescribed or preregistered functions *etc.* The contents of the notification are data used in exception handling and/or control information which changes behaviors of exception handling in the caller. Note that description in this column will appear in the *Detection* column of the FMEA tables for the callers of this function.

Others: We can describe criticality, namely probability of failure mode occurrence and severity of the negative effects, for each failure mode to prioritize countermeasures to be taken.

D. Performing Commonality and Variability Analysis of Failure Mode Countermeasures

Fourth, we perform commonality and variability analysis of run-time failure mode countermeasures and construct a variability model of them. As the variability model, the proposed methodology uses an extension of the feature model [10], which is often employed in product line engineering [4] to describe commonality and variability among products in terms of their features.

The original feature model is a static model to represent variability among products in the product line (or product line variability), that is, the feature model represents how each product can select the features to be equipped. The variability model used in the proposed methodology represents not only product line variability but also variability in dynamic behaviors of the system (or run-time variability), that is, the variability model additionally represents how the system changes its behavior in terms of the features.

The variability model represents product line variability in an almost same manner as the original feature model: mandatory features are represented by nodes without any decoration, optional features are represented by ones with decoration of the white circle, and the nodes corresponding to a set of alternative features are grouped by the solid arc across the edges from the nodes to their parent node. It is also same as the original feature model that the solid thin edge represents the feature of the parent node partially consists of the feature of the child node and the solid thick edge represents the feature of the parent node is implemented by the feature of the child node. However, generalization/specialization relationship between the features is represented by the white arrow from the child node corresponding to the specialized feature to the parent node corresponding to the specialized feature, although the dotted thin edge is used in the original feature model.

Furthermore, the variability model represents run-time variability with symbolic extension to the original feature model. The solid edge means that the feature of the child node is always executed if the feature of the parent is executed. The dotted edge means that the feature of the child node may or may not be executed if the feature of the parent is executed. The dotted arc across the edges means that the features of the children of the bundled edges are alternatively executed if the feature of the parent node of the bundled edges is executed.

Figure 2 shows a variability model for the countermeasures described in Table II. The model does not include any variability on software construction, since the example system is not in product line development, but variability on software execution. For example, the variability model shows that the error message is always displayed but the message to be displayed is alternatively selected out of "Unknown user name", "User DB is locked", *etc.* The variability model tells us that the input fields may not be cleared, but if cleared, the user name field is always cleared and the password field may or may not be cleared.

There are some reasons why the authors use this variability model. While the class of the class diagram can represent only structural aspect of exception handling, the feature of the variability model can represent relating structural and behavioral aspect of exception handling in a consolidated manner. Moreover, variability modeling is a powerful technique to facilitate separation of concerns. For example, in Figure 2, features "Clearing fields" and "Focusing UI Object" are modeled separately. Since this separation enables independent handling of these behaviors, we can avoid scattering of similar or duplicated codes on these behaviors in the exception handling. The variability model can be used also in product line development, not only in single product development, since it inherits the properties of the original feature model.

E. Describing Exceptional Specifications

Fifth, we describe the exceptional specification of the member function. At the same time, we add and/or modify the normal specification of the member function if necessary.

We classify features in the variability model into *normal features* executed as normal operations in the `try` block and *exceptional features* executed as exceptional operations in the `catch` block. The features which terminate a series of normal operation steps for its execution should be exceptional features. The other features can be executed as either. After the classification, we describe the exceptional specification relating to the exceptional features in the same format as the normal specification. Moreover, we add and/or modify the existing normal specification to add the normal specification relating to the normal features identified in run-time failure mode countermeasures.

For the example of Figure 2, we can regard all the features in the variability model as exceptional features, since all the behaviors relating to the features require termination of the normal operation.

F. Designing Exception Classes

Finally, we design exception classes with referencing variability models. The variability model, which has already been constructed for each member function of the class, contains features that represent run-time behaviors as countermeasures for failure modes, namely exceptional operations, and data used in the exceptional operations. Moreover, note that the variability model represents control information specifying which feature should be executed conditionally in exceptional operations.

Before designing exception classes, to reduce the number of exception classes, variability models of member functions

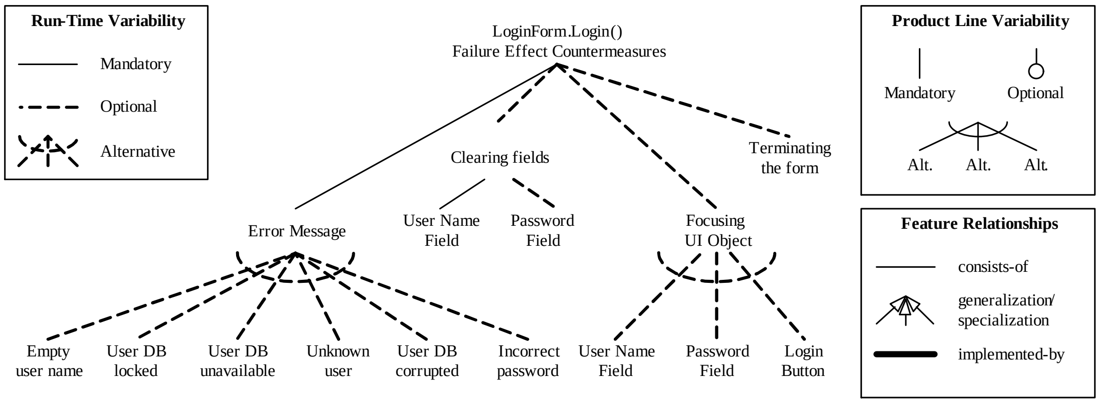

Figure 2. A Variability Model for Run-Time Failure Mode Countermeasures

should be merged and integrated if they are closely related and share a lot of features. The merging and integration are performed based on the names of the features. The variability model may be required to be refactored in this activity. For example, we have to rename the features, if they have the same name although they have different meanings. Moreover, we have to restructure the variability model for integration, if the same features from different variability models are organized in different tree structures.

We define an exception class for each integrated variability model. The features which represent control information changing run-time behaviors of the exceptional operations are reduced into attributes for flagging. The features which represent data used in the exceptional operations are reduced into attributes. Reference and conversion of these attributes are defined as member functions of the exception class.

The exception class reduced from the variability model shown in Figure 2 is as follows:

```
class ExLoginForm : public Exception
{
public:
  enum LoginFormError {
    ERR_EMPTY_USER_NAME,
    ERR_USERDB_LOCKED,
    ...
  } login_form_error;
  enum ClearingFields {
    UserNameField = 0x01,
    PasswordField = 0x02,
  } clearing_fields;
  enum FocusingUIObject {
    UserNameField,
    PasswordField,
    LoginButton
  } focusing_UI_object;
  bool terminating;
  string getErrorMessage();
  ...
};
```

In this reduction, a feature representing data or behaviors taken alternatively such as “Error Message” and “Focusing UI Object” is reduced to the enumeration data member. A feature representing data or behaviors taken multiply such as “Clearing fields” is reduced to the bit field data member. A feature representing data or behaviors taken optionally such as “Terminating the form” is reduced to the Boolean data member.

Considering overheads, use of `try-catch-finally` exception handling control structure should be carefully limited. Countermeasures can be implemented by general control structures and variables instead of exception handling control structures. If we can write the same thing without losing readability and producing duplicated codes in exception handling, use of general control structures is preferable. The proposed methodology will be helpful also in such a case to realize exception handling codes in a consistent manner and without including duplication.

V. Related Work

This work is refined from authors’ previous work [11] presented as a non-peer-reviewed, ongoing paper in Japanese.

Although FMEA was originally applied to mechanical and hardware systems, efforts applying FMEA to software has been continued so far [8], [12], [13], [14], [15], [16], [17].

HAZOP is an effective methodology to find possible failure modes in a comprehensive manner. The failure mode derivation methodology used in this work applies HAZOP guide words to properties of behaviors and their targetting objects [8]. The methodology is an extension of Kouno’s work [18]. HAZOP is applied to software for UML in Hansen’s work [19] and for state transition diagram in Kim’s work [20], for example. Both works are for descriptions in higher abstraction level than this work.

In this work, FTA is also used to seek for the causes of the failure and study countermeasures depending on the causes, in case the abstraction level of the failure mode is higher. The approach looking for the causes from the failure mode is taken by Lutz *et al.* [12] and Goddard [13], for example.

On the other hand, Maxion *et al.* presents an approach based on the viewpoints given by fish-bone diagram that classifies exceptional conditions into computation, hardware, input and output, library, data, return value, external environment, and null pointer and memory problems, which can be abbreviated as "CHILDREN". [21]

VI. Conclusion and Future Work

In this paper, the authors proposed a methodology to specify and design exception handling for `try-catch-finally` exception handling control structure of C++, Java and other similar programming languages.

The proposed methodology applies a HAZOP based failure mode derivation method proposed by the authors to each normal operation step of the member function. FMEA is performed to study countermeasures to the identified failure modes. The table produced in FMEA facilitates consistent and disciplined design of exception handling. Commonality and variability analysis is applied to the countermeasures studied in FMEA. Commonality and variability analysis contributes to eliminate duplicated codes in exception handling codes. A variability model constructed by the analysis is used to design exception classes.

The proposed methodology will be used not only for newly development but also for refactoring of exception handling codes in the existing system. The future work includes large scale application of the methodology to real applications and empirical validation of the methodology.

References

[1] Flaviu Cristian, "Exception Handling and Tolerance of Software Faults," *Software Fault TOlerance*, M. R. Lyu, ed., Chapter 4, John Wiley & Sons, 1995.

[2] IEC Standard, *Hazard and Operability Studies (HAZOP Studies): Application Guide*, IEC 61882 ed1.0, 2001.

[3] IEC Standard, *Analysis Techniques for System Reliability: Procedure for Failure Mode and Effects Analysis (FMEA)*, IEC 60812 ed2.0, 2006.

[4] Paul Clements and Linda Northrop, *Software Product Lines: Practice and Patterns*, Addison-Wesley, 2001.

[5] John B. Bowles, "The New SAE FMECA Standard," *Proc. Annual Reliability and Maintainability Symp. (RAMS) 1998*, pp. 48–53, Jan. 1998.

[6] John B. Bowles, "Fundamentals of Failure Modes and Effects Analysis," Tutorial Notes, *Annual Reliability and Maintainability Symp. (RAMS) 2003*, Jan. 2003.

[7] C. Steven Spangler, "Equivalence Relations within the Failure Mode and Effect Analysis," *Proc. Annual Reliability and Maintainability Symp. (RAMS) 1999*, pp. 352–357, Jan. 1999.

[8] Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda, "A Software FMEA Method and Its Use in Software Product Line," *IEICE Technical Report*, Vol. 111, No. 481, pp. 19–24, Mar. 2012. (in Japanese)

[9] IEC Standard, *Fault Tree Analysis (FTA)*, IEC 61025 ed2.0, 2006.

[10] Kyo-Chul Kang, Jaejoon Lee, and Patrick Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, Vol. 9, No. 4, pp. 58–65, July/August 2002.

[11] Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda, "Specifying and Designing Exception Handling with using FMEA," *IPSJ SIG Technical Report*, Vol. 2012-SE-175, No. 14, pp. 1–8, Mar. 2012. (in Japanese)

[12] Robyn R. Lutz and Robert M. Woodhouse, "Experience Report: Contributions of SFMEA to Requirements Analysis," *Proc. 2nd Int. Conf. on Requirements Engineering (ICRE '96)*, pp. 44–51, Apr. 1996.

[13] Peter L. Goddard, "Software FMEA Techniques," *Proc. Annual Reliability and Maintainability Symp. (RAMS) 2000*, pp. 118–123, Jan. 2000.

[14] John B. Bowles and Chi Wan, "Software Failure Modes and Effects Analysis for a Small Embedded Control System," *Proc. Annual Reliability and Maintainability Symp. (RAMS) 2001*, pp. 1–6, Jan. 2001.

[15] Dong Nguyen, "Failure Modes and Effects Analysis for Software Reliability," *Proc. Annual Reliability and Maintainability Symp. (RAMS) 2001*, pp. 219–222, Jan. 2001.

[16] Nathaniel Ozarin, "Failure Modes and Effects Analysis during Design of Computer Software," *Proc. Annual Reliability and Maintainability Symp. (RAMS) 2004*, pp. 201–206, Jan. 2004.

[17] Ajit Ashok Shenvi, "Software FMEA: A Learning Experience," *Proc. India Software Engineering Conf. (ISEC) 2011*, pp. 111–114, Feb. 2011.

[18] Tetsuya Kouno, "An Application of HAZOP to Risk Analysis of Software Requirement Specification," *Proc. Japan Symp. on Software Testing 2012*, pp. 37–42, Jan. 2012. (in Japanese)

[19] Klaus M. Hansen, Lisa Wells, and Thomas Maier, "HAZOP Analysis of UML-Based Software Architecture Descriptions of Safety-Critical Systems," *Proc. 2nd Nordic Workshop on the Unified Modeling Language (NWUML) 2004*, pp. 59-78, Aug. 2004.

[20] Zoohaye Kim, Yutaka Matsubara, and Hiroaki Takada, "A Safety Analysis Method Based on State Transition Diagram," IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, Vol. J95-A, No. 2, pp. 198-209, Feb. 2012. (in Japanese)

[21] Roy A. Maxion and Robert T. Olszewski, "Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study," *IEEE Trans. on Software Engineering*, Vol. 26, No. 9, Sep. 2000.