J. Software Engineering & Applications, 2010, 3: 303-311
doi:10.4236/jsea.2010.34036 Published Online April 2010 (http://www.SciRP.org/journal/jsea)

Scientific Research

Separation of Fault Tolerance and Non-Functional Concerns: Aspect Oriented Patterns and Evaluation

Kashif Hameed, Rob Williams, Jim Smith

University of the West of England, Bristol Institute of Technology, Bristol, UK.
Email: {Kashif3.Hameed, Rob.Williams, James.Smith}@uwe.ac.uk

Received January 1st, 2010; revised January 30th, 2010; accepted February 1st, 2010.

ABSTRACT

Dependable computer based systems employing fault tolerance and robust software development techniques demand additional error detection and recovery related tasks. This results in tangling of core functionality with these cross cutting non-functional concerns. In this regard current work identifies these dependability related non-functional and cross-cutting concerns and proposes design and implementation solutions in an aspect oriented framework that modularizes and separates them from core functionality. The degree of separation has been quantified using software metrics. A Lego NXT Robot based case study has been completed to evaluate the proposed design framework.

***Keywords*:** *Aspect Oriented Design and Programming, Separation of Concerns, Executable Assertions, Exception Handling, Fault Tolerance, Software Metrics*

1. Introduction

Adding fault tolerance (FT) measures and other non-functional requirements to safety critical and mission critical applications introduces additional complexity to the core application. By incorporating handler code, for error detection, checkpointing, exception handling, and redundancy/diversity management, the additional complexity may adversely affect the dependability of a safety critical or mission critical system.

One of the solutions to reduce this complexity is to separate and modularize the extra, cross-cutting concerns from the true functionality.

Although Rate of Change (ROC) based plausibility checks for error detection and recovery have been addressed by [1,2], unfortunately none of the previous studies propose the separation of these error handling concerns from true functionality to avoid complexity.

At the level of design and programming, several approaches have been utilized that aim at separating functional and non-functional aspects. Component level approach like IFTC [3], computational reflection and meta-object protocol based MOP [4] have shown that dependability issues can be implemented independently of functional requirements.

The evolving area of Aspect-Oriented Programming & Design (AOP&D) presents the same level of independence by supporting the modularized implementation of crosscutting concerns.

Aspect-oriented language extensions, like AspectJ [5] and AspectC++ [6] provide mechanisms like *Advice* (behavioural and structural changes) that may be applied by a pre-processor at specific locations in the program called *join point.* These are designated by *pointcut* expressions. In addition to that, static and dynamic modifications to a program are incorporated by *slices* which can affect the static structure of classes and functions.

The current work thus proposes some generalized aspect oriented design patterns representing fault tolerance error detection and recovery mechanisms like ROC plausibility checks, exception handling, checkpointing and watchdog. Moreover some additional design patterns for developing robust mission/safety critical software are also presented. Software metrics like coupling, cohesion and size have been applied quite successfully to access and evaluate the quality attributes of OO software systems [7, 8]. However separation of concerns (SOC) especially cross cutting ones in the light of new abstraction addressed by AO software development demands some additional metrics. The current work reviews these additional metrics like concern diffusion over the components (CDC), concern diffusion over the operations (CDO) and concern diffusion over the lines of code (CDLOC). The

Copyright © 2010 SciRes

SOC metric suite is later applied on the proposed AO patterns in an empirical case study. This helps evaluating the degree to which AOSD modularizes the FT concerns and its impact on other quality attributes.

The validation and dependability assessment of proposed AOFT patterns has already been done in an earlier work by the author [9].

2. Aspect Oriented Exception Handling Patterns

Exception handling has been deployed as a key mechanism in implementing software fault tolerance through forward and backward error recovery mechanisms. It provides a convenient means of structuring software that has to deal with erroneous conditions [10].

In [11], the authors addresses the weaknesses of exception handling mechanisms provided by mainstream programming languages like Java, Ada, C++, C#. In their experience exception handling code is inter-twined with the normal code. This hinders maintenance and reuse of both normal and exception handling code.

Moreover as argued by [12], exception handling is difficult to develop and has not been well understood. This is due to the fact that it introduces additional complexity and has been misused when applied to a novel application domain. This has further increased the ratio of system failures due to poorly designed fault tolerance strategies.

Thus fault tolerance measures using exception handling should make it possible to produce software where 1) error handling code and normal code are separated logically and physically; 2) the impact of complexity on the overall system is minimized; and 3) the fault tolerance strategy may be maintainable and evolvable with increasing demands of dependability.

In this respect, [4] has proposed an architectural pattern for exception handling. They address the issues like specification and signaling of exceptions, specification and invocation of handlers and searching of handlers. These architectural and design patterns have been influenced by computational reflection and meta-object protocol.

However, most meta-programming languages suffer performance penalties due to the increase in meta-level computation at run-time. This is because most of the decisions about semantics are made at run-time by the meta-objects, and the overhead to invoke the meta-objects reduces the system performance [13].

Therefore we propose generalized aspect based patterns for monitoring, error detection, exception raising and exception handling using a static aspect weaver. These patterns would lead to integration towards a robust and dependable aspect based software fault tolerance. The following design notations have been used to express aspect-oriented design patterns shown in **Figure 1**.

2.1 Error Detection and Exception Throwing Aspect

Error detection and throwing exceptions has been an anchor in implementing any fault tolerance strategy. This aspect detects faults and throws range, input and output type of exceptions. The overall structure of this aspect is shown in **Figure 2**. The *GenThrowErrExcept* join points the *NormalClass* via three pointcut expressions for each type of fault tolerance case.

RangeErrPc: this join points the *contexMethod*() only. It initiates a before advice to check the range type errors before executing the *contextMethod*(). In case the assertions don't remain valid or acceptable behavior constraints are not met, *RaneErrExc* exception is raised.

InputErrPc: this join points the *contextMethod*() further scoped down with input arguments of the *contextMethod***()**. It initiates a before advice to check the valid input before the execution of the context method. Incase the input is not valid it raises *InputErrExc***.**

OutputErrPc: this join points the *contextMethod*()

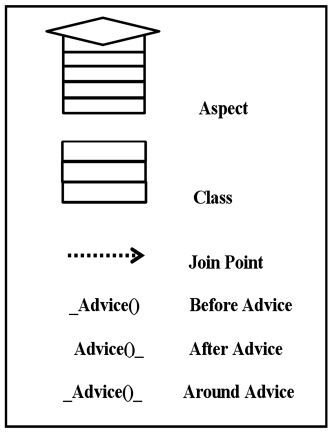


Figure 1. Aspect oriented design notations

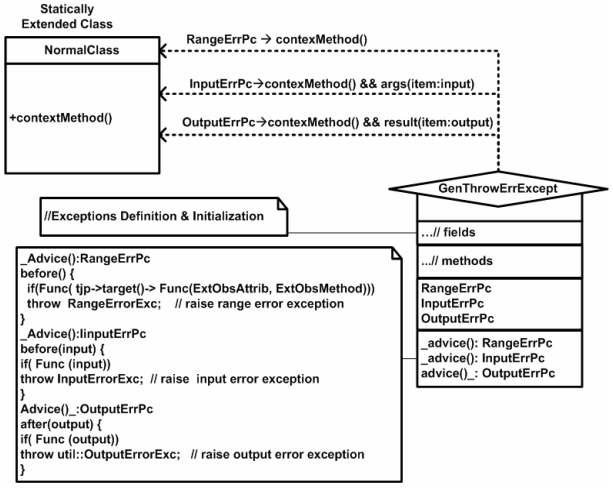


Figure 2. Error detection, exception throwing

Copyright © 2010 SciRes

further scoped down with results as output of the *contextMethod*(). It initiates an after advice to check the valid output after the execution of the context method. Incase the output is not valid it raises **OutputErrExc**.

2.2 Rate of Change Plausibility Check Aspect

This aspect as shown in **Figures 3** and **4** is responsible for checking the erroneous state of the system based on the rate of change in critical signal/data values. Once an erroneous state is detected, the respective exception is raised. Various exceptions are also defined and initialized in this aspect. The *pointcut GetSensorData* defines the location where error checking plausibility checks are weaved whenever a critical data/sensor reading function is called. The light weight ROC-based plausibility assertions are executed in the *advice* part of this aspect.

2.3 Catcher Handler Aspect

The *CatcherHandler* aspect as shown shown in **Figure 5(a)** is responsible for identifying and invoking the appropriate handler. This pattern addresses two run-time handling strategies.

The first strategy is designated by an *exit_main* pointcut expression. It checks the run-time *main*() function for various fatal error exceptions and finally aborts or exits the main program upon error detection. This aspect may be used to implement safe shut-down or restart mechanisms in safety critical systems to ensure safety, if a fatal error occurs or safety is breached.

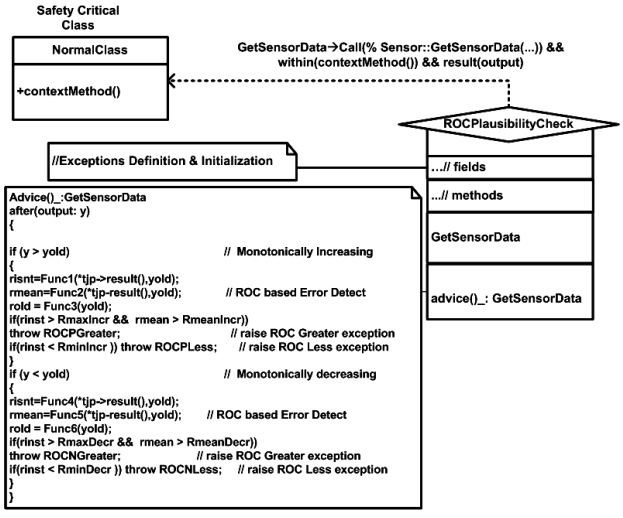


Figure 3. Rate of change aspect pattern structure

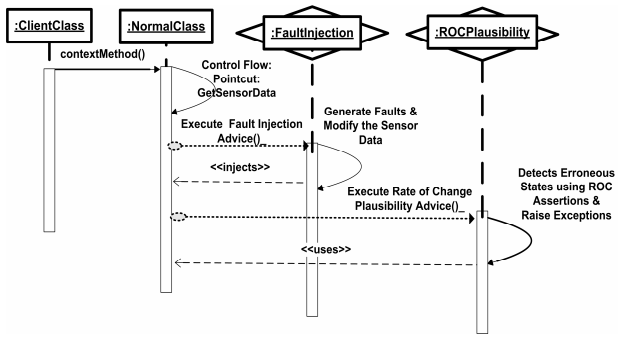


Figure 4. Rate of change aspect pattern dynamics

It can be seen in the **Figure 5(a)** below that *exit_main* pointcut expression join points the main() run-time function. Whereas *caller_return* pointcut expression join points every call to the *contextMethod*(). Moreover *exit_main* and *caller_return* pointcut expressions are associated with an around advice to implement error handling. The tjp→proceed() allows the execution run-time main() and called functions in the try block.

The **advice** block of the catcher handler identifies the exception raised as a result of in-appropriate changes in the rate of signal or data. Once the exception is identified, the recovery mechanism is initiated that assign new values to signal or data variables based on previous trends or history of the variable.

2.4 Dynamics of Exception Handling Aspect

This scenario shown in **Figure 5(b)** represents a typical error handling case. It simulates two error handling strategies. In the first case, control is returned from the caller to stop the propagation of errors along with a system warning. In the second case the program exits due to a fatal error. This may be used to implement shutdown or restart scenarios. Moreover the extension of a class member function with a *try* block is also explained. A client object invokes the *contextMethod*() on an instance of *NormalClass*. The control is transferred to *CatcherHandler* aspect that extends the *contextMethod*() by wrapping it in a *try* block and executes the normal code. In case an exception is raised by previous aspect, the exception is caught by the *CatcherHandler* aspect. This is shown by the catch message. The condition shows the type of exception *e* to be handled by the handler aspect. *CatcherHandler* aspect handles the exception e. the caller_return strategy warns or signals the client about the exception and returns from the caller. The client may invoke the *contextMethod*2() as appropriate. In exit_main strategy, the control is retuned to client that exits the current instances as shown by the life line end status.

3. Watch Dog Aspect

A watchdog is a common concept used in real time systems for detecting and handling errors in real time sys

Copyright © 2010 SciRes

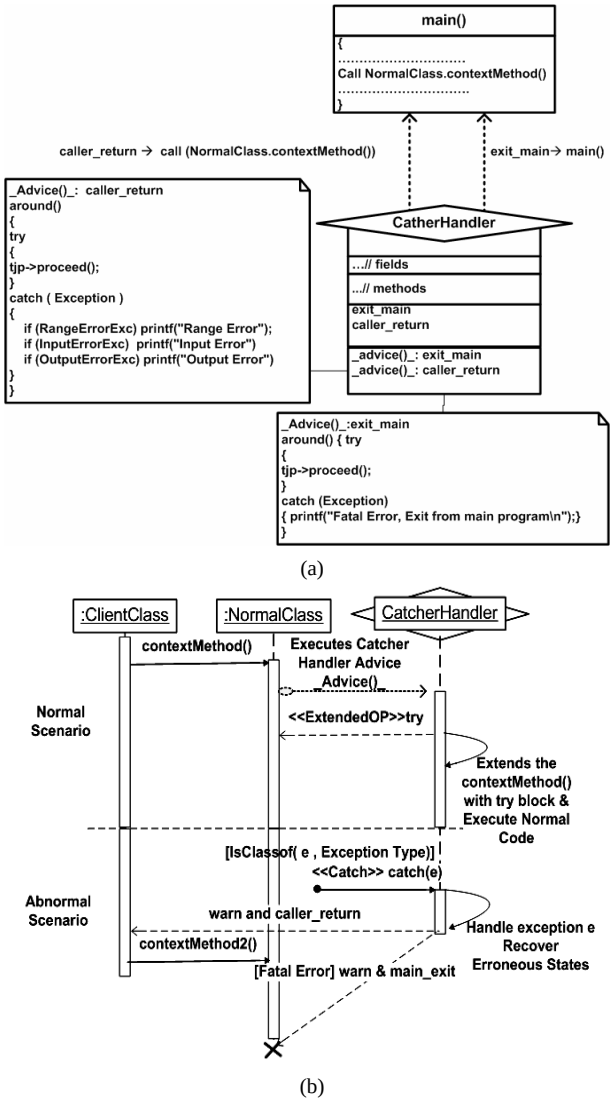


Figure 5. Catcher handler aspect. (a) Structure; (b) Dynamics

tems. It is a component that detects error by receiving a delayed or null service response. Based on such timing faults, it initiates a corrective action, such as reset, shutdown, alarm to notify attending personnel, or signaling more elaborate error-recovery mechanisms. Sometimes software watchdogs are more active by performing periodic built-in-tests (BIT). Synchronous tasks are more prone to such timing based faults resulting in mission failures.

In this regard we present a watchdog aspect (**Figure 6**) to make such tasks fault tolerant by weaving an advice code. Thus every synchronous mission critical task is monitored against a deadline that is derived from the worst case execution time of the overall task. As the dead line is expired, the mission is aborted. The watchdog aspect is presented below. It can be seen that every call to a **contextMethod()** of a **NormalClass** is weaved with a timing check to see whether time delay between current and previous call exceeds the dead line or not. The watchdog aspect communicates with an external clock interface to receive time stamps. Thus the watch dog aspect separates timing concerns from the true functionality. It also localizes the definition and signaling of exceptions.

4. Save Data and Checkpointing Aspect

Some tasks require context related critical data to be stored for post analysis and executing recovery mechanisms. Every call to these tasks is weaved with a data saving advice. **SaveData** aspect (**Figure 7**) provides checkpointed stable recovery data to be used in ROC based error detection and recovery mechanisms.

Whenever a critical function is called, **SaveData** aspect stores the contextual state information with the help of **MemoryInterface.**

5. System Configuration & Initialization Aspect

Most real time systems rely on sensors in-order to attain

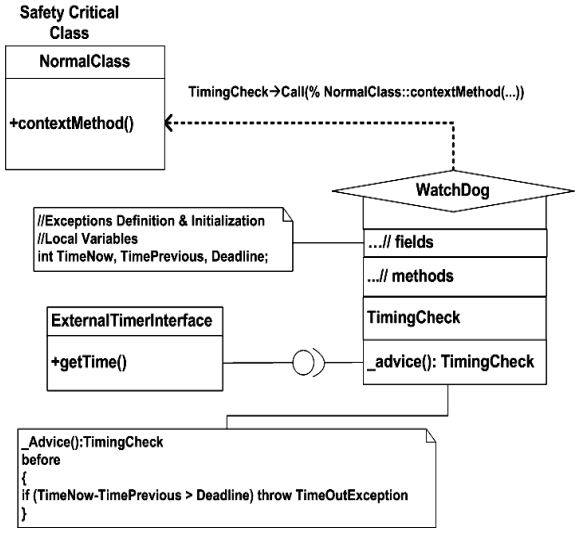


Figure 6. Watchdog aspect

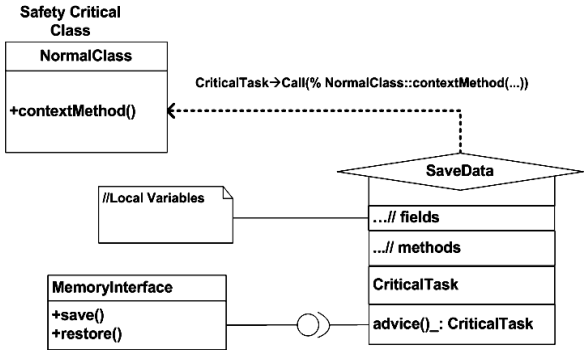


Figure 7. Save data & checkpointing aspect

Copyright © 2010 SciRes

physical information from external environment. These sensors need to be configured and initialized depending upon the modes of operation. For example a Lego NXT robot (Tribot) used in our case study uses light, ultrasonic and rotation sensors to carry out tasks. These sensors are mapped on respective ports of the NXT brick. Moreover they must be initialized before starting actual tasks. It has also been observed that rotation sensors are reinitialized as the direction of rotation changes (when Tribot start traversing backward). All such requirements either cut-across true functional concern or emerges as additional non-functional requirement. Such requirements have been implemented in an aspect thus separating them from true functional concern. This aspect is weaved as a startup advice in the control flow of main program.

6. Mission Pre-Conditions Aspect

Mission critical real time systems require some pre-conditions or constraints to be met before starting the core task. For example Tribot check the voltage level of batteries and ambient light before staring its mission so that it could fulfill its tasks reliably. If the above constraints are not met, mission is aborted. Such constraints are cross cutting to core functional requirements and thus implemented as a separate aspect as shown as shown in **Figure 8**.

As soon as the software finishes system initialization, the said aspect acquires environmental data from the **SensorInteface** and checks against the pre-conditions or constraints. If the constraints are not met, an exception is thrown and mission is aborted.

7. Case Study

In order to evaluate proposed AO design patterns, a case study has been carried out using a LEGO NXT Robot (Tribot). This uses an Atmel 32-bit ARM processor running at 48 MHz. Our development environment utilizes AspectC++ 1.0pre3 as aspect weaver [6].

The Tribot has been built consisting of two front wheels driven by servo motors, a small rear wheel and an arm holding a hockey stick with the help of some standard Lego parts. Ultrasonic and light sensors are also available for navigation and guidance purposes.

An interesting task has been chosen to validate our design. In this example Tribot hits a red ball with its hockey stick avoiding the blue ball placed on the same ball stand. It makes use of the ultrasonic and light sensors to complete this task. This task is mapped on a goal-tree diagram as shown in **Figure 9**.

Any deviation in full-filling the OR goals and corresponding AND sub-goals is considered as a mission failure.

8. Aspects Evaluation via Software Metrics

Although software metrics like coupling, cohesion and size has been used to access the software quality for quite some time, yet the separation of concerns especially cross cutting ones with the aid of aspect oriented software development demands some additional metrics suite for its assessment. In this regard [14-16] have proposed additional metric suite for separation of concerns. This metric suite has been utilized in [17] to access the quality of some large scale software systems.

These additional metrics measure the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. For all the employed metrics, a lower value implies a better result. Some of these metrics used in our study are explained below.

8.1 Separation of Concerns Metrics

Separation of concerns (SoC) refers to the ability to identify, encapsulate and manipulate those parts of software that are relevant to a particular concern. The metrics for

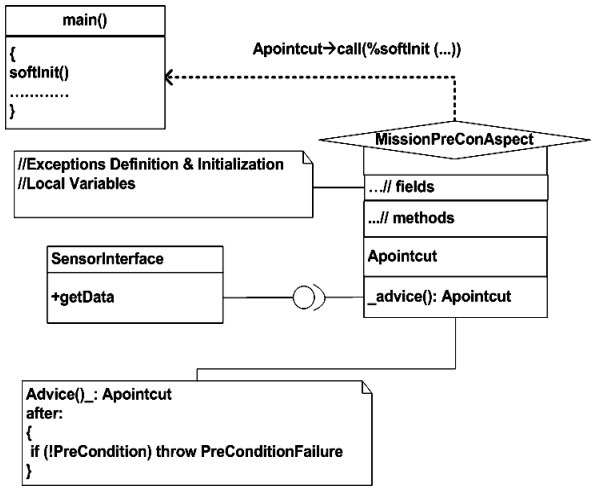


Figure 8. Mission pre-condition aspect

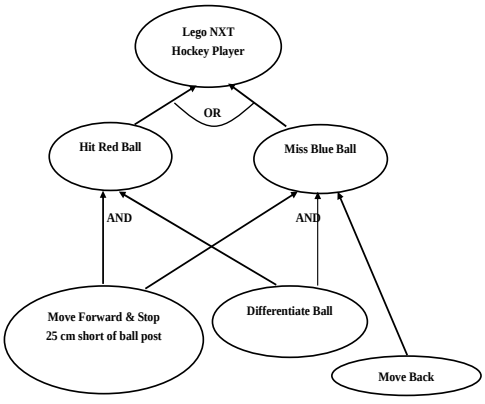


Figure 9. Lego NXT robot case study: Goal tree diagram

Copyright © 2010 SciRes

SoC measurement are:

Concern Diffusion over Components (CDC)

This metric measures the degree to which a single concern in the system maps to the components in the software design. The more direct a concern maps to the components, the easier it is to understand. It is also easier to modify and reuse the existing components.

Definition: CDC is measured by counting the number of primary components whose main purpose is to contribute to the implementation of a concern. Furthermore, it counts the number of components that access the primary components by using them in attribute declarations, formal parameters, return types, throws declarations and local variables, or call their methods.

Concern Diffusion over Operations (CDO)

One way of measuring the code tangling is by counting the number of operations affected by concern code. If a concern is scattered around more operations, it becomes harder to understand, maintain and reuse.

Definition: CDO is measured by counting the number of primary operations whose main purpose is to contribute to the implementation of a concern. In addition, it counts the number of methods and advices that access any primary component by calling their methods or using them in formal parameters, return types, throws declarations and local variables. Constructors also are counted as operations.

Concern Diffusion over LOC (CDLOC)

The intuition behind this metric is to find concern switching with in the lines of code. For each concern, the program text is analyzed line by line in order to count transition points. The higher the CDLOC, the more intermingled is the concern code within the implementation of the components; the lower the CDLOC, the more localized is the concern code.

Definition: CDLOC counts the number of transition points for each concern through the lines of code. The use of this metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas. The shadowed areas are lines of code that implement a given concern. Transition points are the points in the code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. An extensive set of guidelines to assist the shadowing process is reported in [15].

8.2 Coupling Metrics

Coupling is an indication of the strength of interconnections between the components in a system. Highly coupled systems have strong interconnections, with program units dependent on each other [14]. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Excessive coupling between components is detrimental to modular design and prevents reuse. The more independent a component is, the easier it is to reuse it in another application [14]. The metrics in this category are Coupling between Components (CBC) and Depth of Inheritance Tree (DIT).

Coupling between Components (CBC)

This counts the coupling between classes, classes and aspects and between other aspects. It counts the classes used in attribute declarations *i.e.* C2 and C3 depicted in figure below. It also counts the number of components declared in formal parameters, return types, throws declarations and local variables. Moreover classes and aspects from which attribute and method selections are made are also included.

New coupling dimension are also defined in [14] in order to support aspect oriented software development (AOSD). For *e.g*. access to aspect methods and attributes defined by introduction (couplings C4, C5, C7, C8, C10), and the relationships between aspects and classes or other aspects defined in the pointcut (couplings C6, C9) as depicted in **Figure 10**. Thus overall this metric encompasses nine coupling dimensions (from C2 to C10). If a component is coupled to another component in an arbitrary number of forms, CBC counts only once.

Depth of Inheritance Tree (DIT)

DIT is defined as the maximum length from a node to the root of the tree. It counts how far down the inheritance hierarchy a class or aspect is declared. This metric encompasses the coupling dimensions C1 and C11 illustrated in **Figure 10**.

8.3 Lego NXT Software Measures Analysis

Software metrics are attained for the Lego NXT Robot case study as shown in **Figure 11**. In this case study, a C++ based true functionality has been made fault tolerant by weaving various concerns in 30 places using 7 aspects and 10 independent point cut expressions. These 7 aspects represent different concerns that otherwise may be added to actual true concern making the code more tangled, non maintainable and non reusable.

Separation of Concern Measures

Separation of concerns has been evaluated using CDC, CDO and CDLOC figures attained in the above case study.

The Concern Diffusion over the components (CDC) metrics measures the mapping of a single concern on various components. It can be inferred from **Figure 12** below that there is 64% reduction in mapping of true concern on the components present in the system due the introduction of aspects. Moreover the individual aspects implementing cross cutting concerns don't present large CDC figures that means, the aspects are loosely coupled with the system and thus can be more powerful candidates for reusability.

Same behavior has been observed in CDO measures as shown in **Figure 13**. As argued in [15,16], the code tang-

Copyright © 2010 SciRes

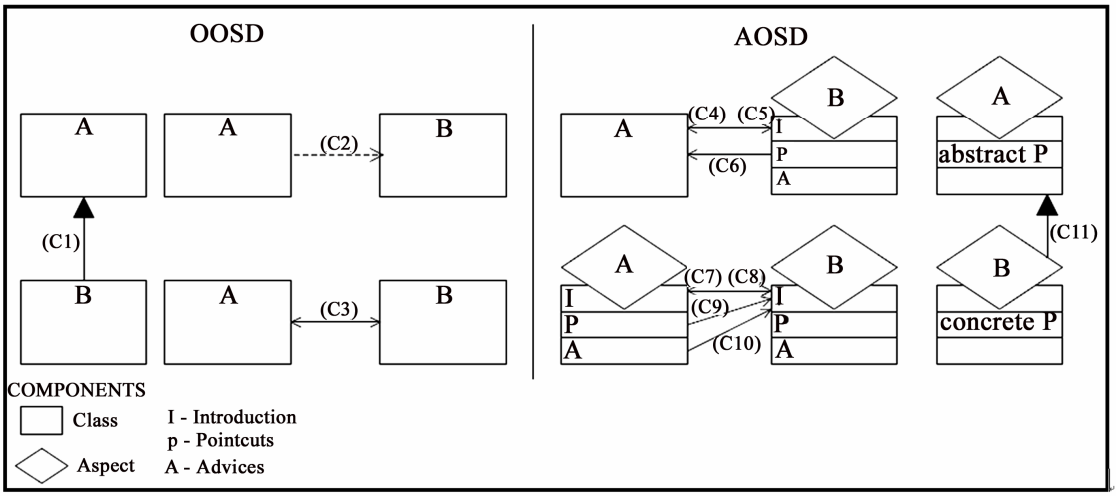


Figure 10. Coupling dimensions on AOSD [14]

Concerns	Type of Concerns (Cross Cutting Places)	Components		Operations				Exception Handling		Pointcut	Separation of Concerns			Couping
		Class/ Interface	Aspects	functions	Advice			Exception Definition & Throwing	try/catch blocks		CDC	CDO	CDLOC	
					Before	After	Around							
Tribot Case Study (TCS)	Functional Concern	3	0	17				0	0		7	17	2	3
Mission Pre-Condition (MPC)	Non-Functional (1 place)	1	1	2		1		2	0	1	3	3	0	3
System Configuration and Initialization (SCI)	Cross-Cutting (2 places)	1	1	1	2			0	0	2	2	3	0	2
Save Data (SD)	Cross-Cutting (5 places)	2	1	4	1			0	0	1	2	3	0	3
WatchDog (WD)	Cross-Cutting (5 places)		1	1	1			1	0	1	1	3	0	1
ROC Plausibiltiy Check Error Detection (ROC)	Cross-Cutting (2 places)		1			1		4	0	1	1	1	2	1
Exit-Main Catcher Handler (EMH)	Cross-Cutting (1 place)		1	2			1	0	1	1	1	3	1	1
Return Caller Catcher Handler (RCH)	Cross-Cutting (14 places)		1				2	0	2	3	2	7	1	3
True Functionality (TF)	0	3	0	17	0	0	0	0	0	0	7	17	2	3
Cross-Cutting Concerns (CC)	30	4	7	10	4	2	3	7	3	10	12	23	4	14

Figure 11. Lego NXT software metrics

ling may be visualized by observing the diffusion of a concern in different operations. Again the true concern seems more tangled in different operations as compared to cross cutting concerns implemented as aspects.

Concern diffusion over the lines of code (CDOLC) is a measure of how much tangled and inter-winded is the code implemented for a component. The larger the value, more tangled is the code with other concerns.

CDLOC for the core functionality (TCS) counts to 2 that seems a reasonable reduction as compared to non aspect oriented implementation. The seven non-functional/cross cutting concerns may add to present a larger CDLOC (**Figure 14**).

It can also be observed from the CDLOC dispersion that there are some indicators of bit code tangling with the true functionality especially for the aspects responsible for error detection and exception handling. Upon code reviewing it was observed that some critical contextual information is required that resulted in concern switching. Apart from that, overall concern switching for each component is reasonably small to be considered better candidates for reusability and maintainability. There were four concerns implemented with null concern switching in this study.

Coupling Measures

As observed in the study by [17], the coupling between components for various concerns has not increased a lot in our case as well. Coupling between components seem to be uniformly distributed with an average value of 2 as shown in **Figure 15**. Apart from core functionality (TCS), the increased coupling has been observed in the concerns implementing error detection and recovery

Copyright © 2010 SciRes

mechanisms. This is due to the fact that aspects implementing these concerns are coupled with the core concern for acquiring contextual information used in error detection and recovery mechanisms.

Exception Throwing & Handling Measures

It can be seen from **Figures 16(a)** and **16(b)** that exceptions definition and throwing have been localized in the components responsible for error detection like ROC plausibility Checks and Watch Dog. Moreover, exception handling has also been localized in their respective aspects without diffusing any other component.

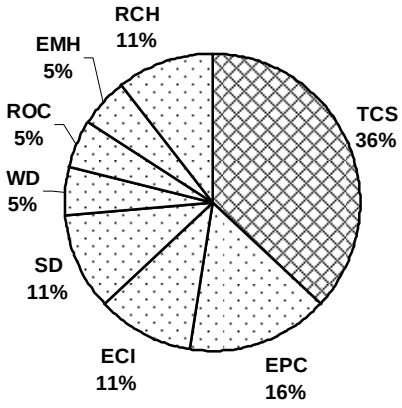


Figure 12. CDC dispersion

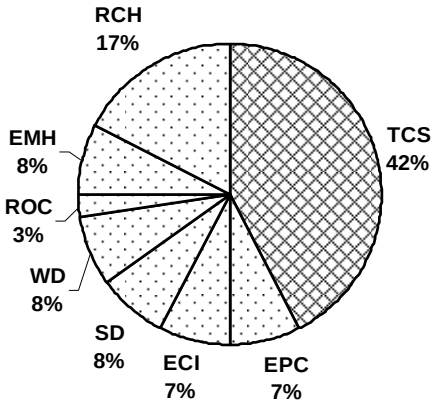


Figure 13. CDO dispersion

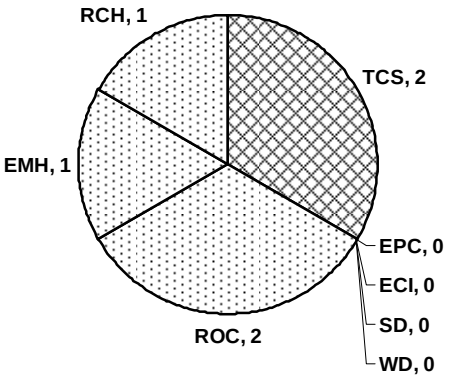


Figure 14. CDLOC dispersion

9. Conclusions & Future Work

The current work proposes AO design patterns for developing fault tolerant and robust software applications.

The aspect oriented design patterns under this framework bring additional benefits like the localization of error handling code in terms of definitions, initializations and implementation. Thus error handling code is not duplicated as the same error detection and handling aspect is responsible for all the calling contexts of a safety critical function. Reusability has also been improved because different error handling strategies can be plugged in separately. In this way, aspect and functional code may both be ported more easily to new systems.

Although a detailed analysis of concerns separation through aspects by refactoring large scale software ap-

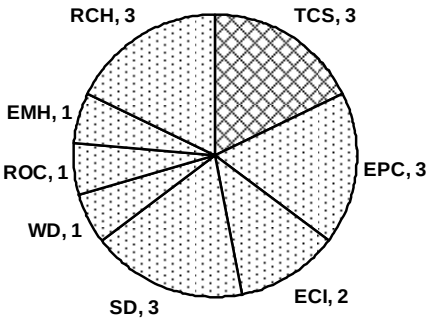
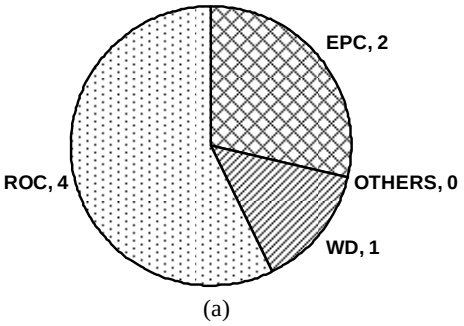
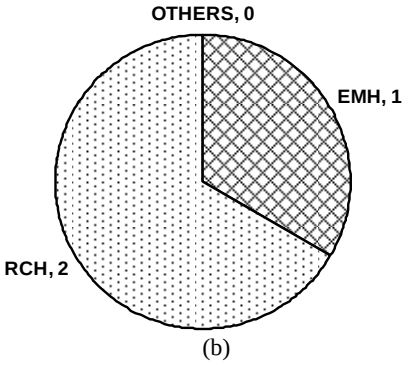


Figure 15. Coupling between components



(a)



(b)

Figure 16. (a) Exceptions define & throw; (b) Try/catch blocks

Copyright © 2010 SciRes

plication has been provided in [17]. Our case study also compliments some of the results. It has been observed that localization of exception management (definition, initialization and throwing) and exception handling improves modularity. It has been observed that fault tolerance concerns when implemented as aspects have resulted in considerable reduction in diffusion of concerns over the core functionality. The concern diffusion in terms of LOC does indicate clear separation and localization of error management related issues. However some code tangling has been observed with error detection based aspects. This is due to the sharing of context information needed for detecting erroneous states. Apart from that CDLOC measures too small in the true functionality. Coupling has been increased in the components responsible for error detection. Thus overall there has been an improvement in separation of concerns at the cost of slightly increased coupling.

This further probes the need for incorporating an error masking strategy like Recovery Blocks and N-Version Programming. An aspect oriented design version of these strategies is also under consideration.

REFERENCES

[1] M. Hiller, *et al*., "Executable Assertions for Detecting Data Errors in Embedded Control Systems," *Proceedings of the International Conference on Dependable Systems & Networks*, New York, June 2000, pp. 24-33.

[2] M. Hiller, "Error Recovery Using Forced Validity Assisted by Executable Assertions for Error Detection: An Experimental Evaluation," *Proceedings of the* 25*th EUROMICRO Conference*, Milan, Vol. 2, September 1999, pp. 105-112.

[3] P. A. C. Guerra, *et al*., "Structuring Exception Handling for Dependable Component-Based Software Systems," *Proceedings of the* 30*th EUROMICRO Conference* (*EUROMICRO*'04), Rennes, 2004, pp. 575-582.

[4] A. F. Garcia, D. M. Beder and C. M. F. Rubira, "An Exception Handling Software Architecture for Developing Fault-Tolerant Software," *Proceedings of the* 5*th IEEE HASE*, Albuquerque, November 2000, pp. 311-332.

[5] AspectJ Project Homepage. http://eclipse.org/aspectj/

[6] AspectC++ Project Homepage. http://www.aspectc.org

[7] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994, pp. 476-493.

[8] V. Basili, L. Briand and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, October 1996, pp. 751-761.

[9] K. Hameed, R. Williams and J. Smith, "Aspect Oriented Software Fault Tolerance," *Proceedings of* 4*th International Conference on Computer Science & Education* (*WCE*09), London, Vol. 1, 1-3 July 2009.

[10] L. L. Pullum, "Software Fault Tolerance Techniques and Implementation," Artech House Inc., Boston, London, 2001.

[11] F. C. Filho, *et al.*, "Error Handling as an Aspect," *Workshop BPAOSD*'07, Vancouver, 12-13 March 2007.

[12] A. Romanovsky, "A Looming Fault Tolerance Software Crisis," *ACM SIGSOFT Software Engineering Notes*, Vol. 32, No. 2, March 2007, p. 1.

[13] K. Murata, R. N. Horspool, E. G. Manning, Y. Yokote and M. Tokoro, "Unification of Compile-Time and Run-Time Metaobject Protocol," *ECOOP Workshop in Advances in Meta Object Protocols and Reflection* (*Meta*'95), August 1995.

[14] C. Sant'Anna, *et al*., "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework," *Proceedings of the* 17*th Brazilian Symposium on Software Engineering*, Salvador, October 2003, pp. 19-34.

[15] A. Garcia, *et al.*, "Agents and Objects: An Empirical Study on Software Engineering," Technical Report 06-03, Computer Science Department, PUC-Rio, February 2003. ftp://ftp.inf.puc-rio.br/pub/docs/techreports/(file03_06_garcia.pdf)

[16] A. Garcia, *et al.*, "Agents and Objects: An Empirical Study on the Design and Implementation of Multi-Agent Systems," *Proceedings of the SELMAS*'03 *Workshop at ICSE*'03, Portland, May 2003, pp. 11-22.

[17] F. C. Filho, *et al*., "Exceptions and Aspects: The Devil in Details," *Proceedings of the* 14*th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, 5 November 2006.

Copyright © 2010 SciRes