Design Patterns to Implement Safety and Fault Tolerance

Hemangi Gawand
Homi Bhabha National Institute
Bhabha Atomic Research Center & Indira Gandhi Center of Atomic Research - India

R.S.Mundada
Homi Bhabha National Institute
Bhabha Atomic Research Center - India

P.Swaminathan
Homi Bhabha National Institute
Indira Gandhi Center of Atomic Research - India

ABSTRACT
This paper discusses an object orient approach based on design pattern and computational reflection concept to implement non- functional requirements of complex control system. Firstly we brief about software architecture design, followed by control-monitor safety pattern, Tri-Modular redundancy (TMR) pattern, reflective state pattern and fault tolerance redundancy patterns that are use for safety and fault management. Reflection state pattern is a refinement of the state design pattern based on reflection architectural pattern. With variation in reflective design pattern we can develop a well structured fault tolerant system. The main goal of this paper is to separate control and safety aspect from the application logic. It details its intent, motivation, participants, consequences and implementation of safety design pattern.

General Terms
Design pattern, Safety pattern, Fault tolerance.

Keywords
Reflective Design pattern, fault tolerance, safety tactics, tri-modular redundancy and digital distributed control system.

1. INTRODUCTION
A design pattern is a description of a set of successful solutions of a recurring problem within a context. A pattern is therefore made-of three pillars: a problem, a context and a solution. Design patterns are mostly described using a combination of text, Unified Modeling Language (UML) diagrams and sample code fragments. The intention is to make them easy to read and use [1].

Modern object oriented system generally include various non-functional requirements that can increase system complexity especially when dealing with distributed control system of complex plants. The development of such system requires the use of appropriate techniques in order to control this additional complexity and to make software, more structured, safer, easier to understand, maintain and reuse. Safety and fault tolerance has been key non- functional requirement that needs to be handled. This paper provides software methods to support safety and fault tolerance using design patterns.

In this paper we present software safety, redundancy and fault tolerance implementation in form of various design patterns. It details its intent, motivation, participants, consequences and implementation.

Sections are categorized as mention below:-

1. Section 2:- Safety Tactics for software architecture design. This section details various safety tactics that makes up software architecture and safety model which implements this tactics.
2. Section 3:- Software safety patterns that details control monitor pattern 1, 2, 3 and TMR pattern.
3. Section 4:- Reflection and Fault Tolerance Redundancy pattern section details its framework and specifying design pattern at different level of abstraction.
4. Section 5:- Conclusion

2. SAFETY TACTICS FOR SOFTWARE ARCHITECTURE DESIGN
Software architecture of a system comprises of software elements, relation among them and properties of both. Software architecture ideally satisfies below requirement:-

1. Fault tolerance
2. Fault avoidance
3. Modularity
4. Ease of modification and change
5. Technology transparency

Software architecture also considers the inevitability of failure as part of the design process. Failure can be:-

1. Random: - Failure due to physical causes or degradation mechanism.
2. Systematic: - Failure due to flaws in system. System subjected to the same conditions fail consistently.[2]

Security Tactics goes hand in hand with safety and are followed to resist, detect or recover from attacks. It also provides confidentiality, integrity as well as assurance. Figure 1 detail about safety tactics.

2.1 Overview of Safety Model
Safety Model helps us to provide a process for integrating safety tactics in designing of software architecture and hence lay foundation for ‘safe’ software architecture. Key features that need to be handled by safety model are as below:-

1. Failure classification.
2. Failure causes
3. Failure behavior
4. Failure recovery

Currently there are various methods for failure analysis. Few methods are listed below.

1. Manual method
2. Cause Effect Sheet
3. Failure mode effect analysis (FMEA)
4. Software Method

For designing and developing safety tactics in software following task needs to be carried out.

1. Analysis of existing system and safety design technique used in it
2. Organizing safety tactics based on methods to handle failure
   i) Failure avoidance
   ii) Failure detection
   iii) Failure containment
3. Documenting safety tactics base on type of failure. Failure can be
   i) Minor: - subsystem failure that does not causes whole main sub system to stop.
   ii) Major: - subsystem failure that causes whole main sub system to stop.

Safety pattern describe in section 3 is one of the method based on failure avoidance technique while fault tolerance follows detection and containment

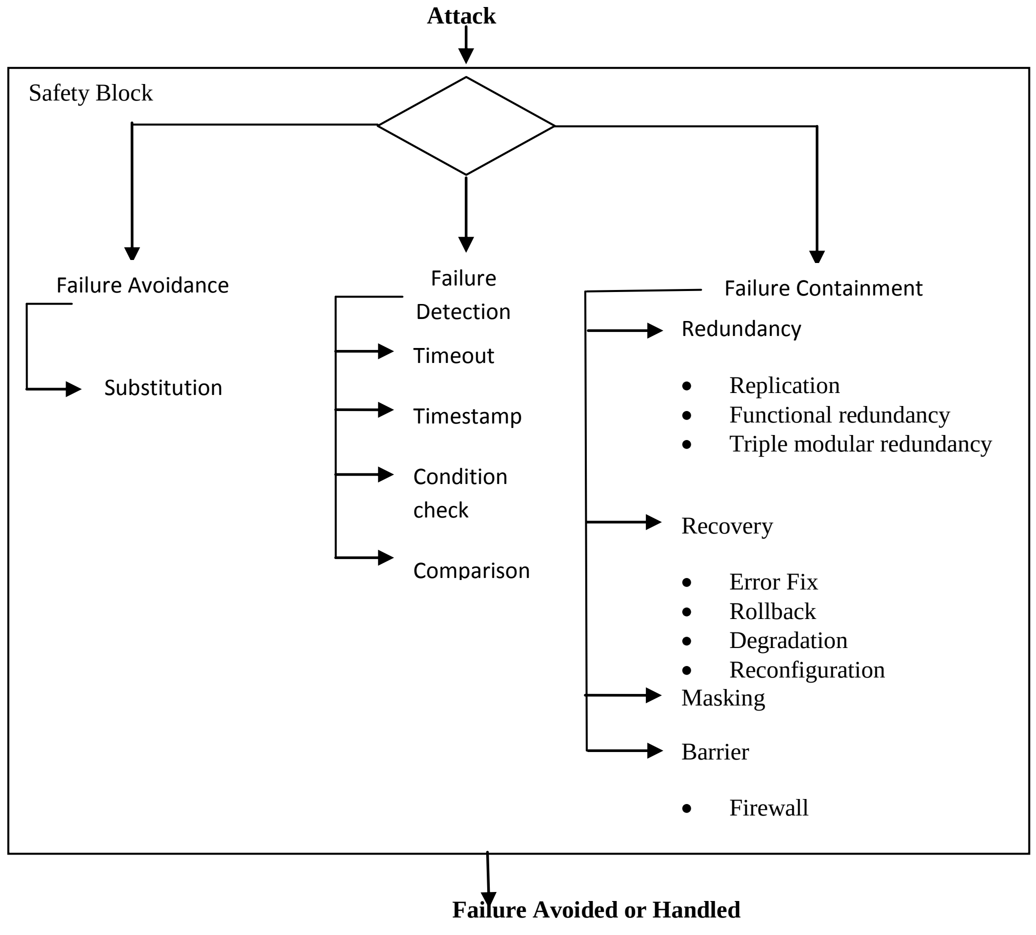


Figure 1:- Safety Tactics hierarchy

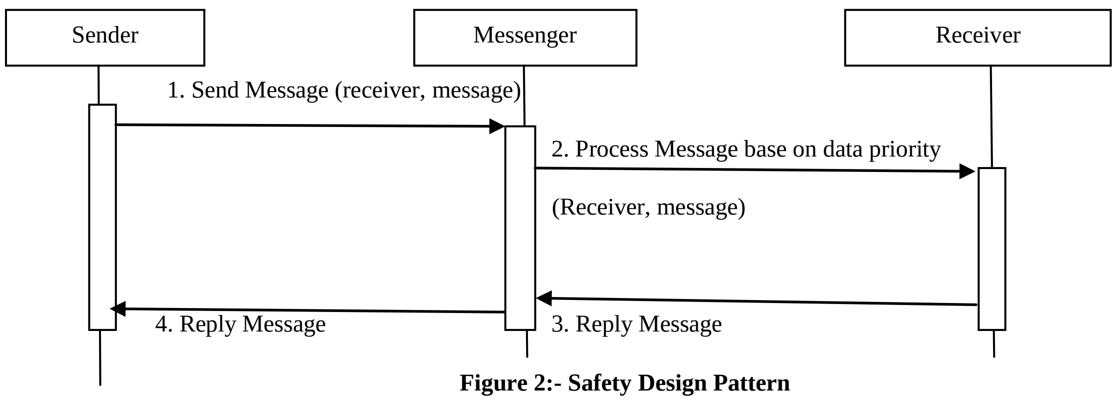

Sender
Messenger
Receiver
1. Send Message (receiver, message)
2. Process Message base on data priority
(Receiver, message)
3. Reply Message
4. Reply Message

Figure 2:- Safety Design Pattern

3. SAFETY DESIGN PATTERN (SDP)

3.1 Intent
The Safety Design Pattern allows the interchange of information between components and applications. Figure 2 details message flow as described in section 3.3

3.2 Motivation
In normal design we classify the classes based on its utilization or actor for e.g. sender, receiver or interface etc and with class instance all the parameters and related functions are created. When it comes to safety critical application where all data cannot be handled in one class with equal priorities, data needs to be split or categorized. In this pattern data is widely classified majorly into 2 types: - Control and monitor. As a consequence, software engineering processes are significantly improved as well as data handling become easier.

3.3 Participants
1. **Message Sender**: Component that sends the message.
2. **Message Recipient (Receiver)**: Component that receives the input message and may produce a reply (output message) after processing it. The component may be instructed to perform computations based on the input message.
3. **Messenger**: Intermediary that transfers the message from the sender to the recipient. The sender and the recipient don’t need to be concerned about how the message is transferred (communication protocol, message format, encryption/security mechanism, etc.) and the transformations performed on the message along the way. Messenger is optional element. Based on data priority, request is forwarded.
4. **Message**: any piece of information (i.e. data) that needs to be interchanged between sender and recipient. Two messages are usually involved:- input message and output message (or reply message). The reply message is not optional.

3.4 Consequences
Encapsulation: - The messaging design pattern maximizes encapsulation. Each component is a self-contained/independent unit. The only mechanism of communication with other components and applications is via messaging.

Decoupling: - SDP minimizes coupling. Again each component is a self-contained unit that can perform independently from the rest of the system.

Reusability: - SDP improves reusability. Applications are also able to reuse components from other applications at the component level i.e. a single component can be extracted from another application.

QA/Testing process: - SDP facilitates testing and debugging efforts. Components are tested as independent units by sending messages to the component and verifying the expected reply messages (black-box testing).

Design process: - SDP improves and simplifies the design process.

Development process: - Since each component that relies on messaging is self-contained, a large team of people can cooperate in the development effort without stepping on each other's code/work. In the ideal situation, responsibility for one component/package can be given to an individual. The rest of the team only needs to know the input/output messages that someone else’s component are designed to handle. No need to change someone else’s code. The need for creating, maintaining and merging several versions of the code is also minimized or eliminated.

Speed of development and cost: - SDP is able to substantially improve the speed of development and reduce cost.

SDP behaves like a state machine. It can be extended to provide fault-tolerant capabilities in a very natural and intuitive fashion by replicating components and coordinating their interaction.

4. A FRAMEWORK FOR THE FORMAL SPECIFICATION OF DESIGN PATTERNS
In this section we introduce our frame work that allows the specification of patterns at different levels of abstraction.

4.1 Pattern Specification
The structural aspect of patterns is represented by subclasses participating in the pattern and associations between them. Classes are represented as set of instances, each of which is represented by an identity taken from an infinite sent of object identities. As such we use the term object and object identity interchangeably. The generalized UML pattern framework is an object-oriented framework that provides support for the base classes that the standard pattern implementation model extends. The specialized patterns framework provides additional functionality such as role-marking and traceability features for pattern participants.

4.1.1 Generalization Relationships
A generalization relationship, which is also called an inheritance or is-a relationship, implies that a specialized, child, class is based on a general, parent, class.
Figure 3 illustrates, a generalization relationship connector appears as a solid line with an unfilled arrowhead pointing from the specialized, child C/C++ class to the general, parent class. Sample C++ code shows its implementation.

4.1.2 Association relationships
An association is a structural relationship that indicates that objects of one classifier, such as a class and interface, are connected and can navigate to objects of another classifier.
Figure 4 gives example of association relationship with sample code.

//UML representation	**//sample C++ code**

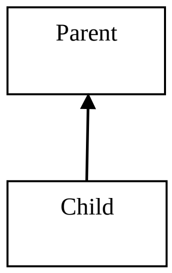
	**Class** Parent {
	protected:
	bool flag;
	int count;
	public:
	Parent();
	virtual ~Parent();
	};
	
	Class Child : public Parent {
	public:
	Child();
	Virtual ~Child();
	};

Figure 3:- Generalization Relation.

//UML representation	**//sample C++ code**

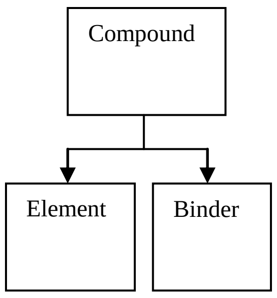
	**Class Compound** {
	protected:
	Element a[1..10];// array of elements
	Binder b; // b as object of Binder class
	public:
	compound();
	virtual ~compound();
	};

Figure 4:- Association Relation.

4.1.3 *Dependency relationships*

In class diagrams, a dependency relationship indicates that a change to one class, the supplier, might cause a change in the other class.Figure 5 gives example of dependency relation in UML digram and its C++ implementation.

//UML representation	**//sample C++ code**

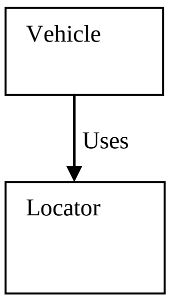
	Class Vehicle{
	Public:
	Vehicle();
	virtual ~Vehicle();
	void getlocation(Locator& loc); // function uses locator class
	
	};

Figure 5:- Dependency Relation

4.2 Types of Safety Design Pattern (SDP)

Safety Design pattern classifies the Input data as Control data and Monitor data. Control data is process by control subsystem or class while monitor data by monitor subsystem respectively. These two systems can behave independently as well as with feedback loop. Based on the connection between them, they are widely classified in to 3 types as detailed below:-

1. Control –Monitor Pattern 1
2. Control –Monitor Pattern 2
3. Control –Monitor Pattern 3

4.2.1 *Control –Monitor Pattern 1*

As shown in figure 6 the input data is send to control as well as monitor sub systems. Control process data output is feed to monitor sub system. Respective output from control and monitor subsystems is given to display or other process system. If any previous data is required for calculation, feedback is provided to monitoring system from output/ other process system.

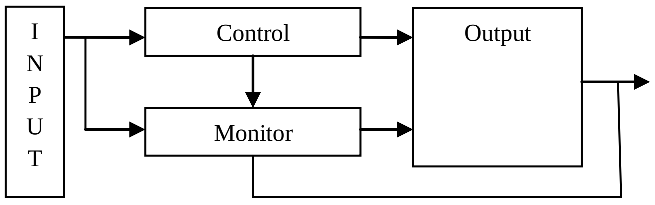


Figure 6:- Control Monitor Safety Pattern 1

4.2.2 *Control –Monitor Pattern 2*

In this Pattern Input class is separate for control and monitor subsystem. Figure 7 gives block diagram control monitor pattern 2. It is similar to control-monitor pattern 1 for its operation.

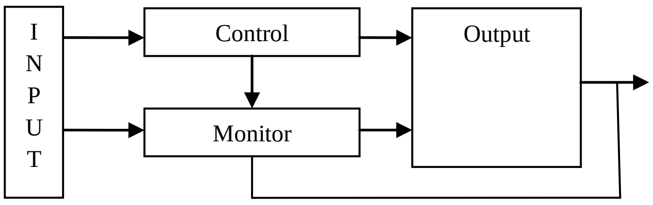


Figure 7:- Control Monitor Safety Pattern 2

4.2.3 *Control –Monitor Pattern 3*

In this Pattern there are 3 differences compare with pattern 1

1. Input class is separate for control and monitor subsystem
2. Control and monitor subsystem have association for cross checking the output before it is feed to other system.
3. No feedback to Monitor subsystem

Figure 8 details about pattern.

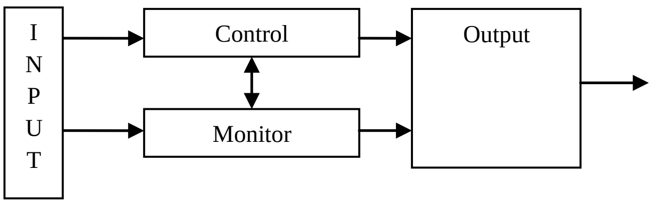
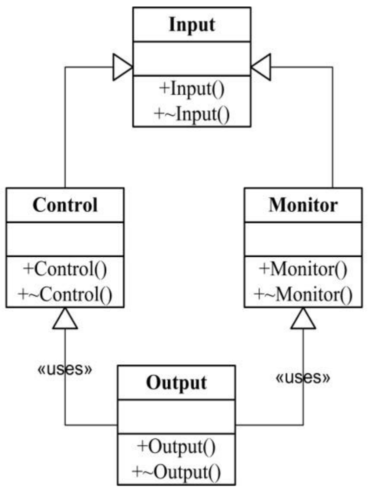


Figure 8:- Control Monitor Safety Pattern 3

Generic Class diagram to represent Control Monitor Safety Pattern is as shown in figure 9.

//UML representation



```
//Sample C++ code
Class Output{
Public:
Output();
virtual ~Output();
//additional  function using Control and Monitor class varies as per //type 1,2 or 3
};

Class Control : public Input {
public:
Control();
Virtual ~Control();
// additional Control data specific functions varies as per //type 1,2 or 3
};

Class Monitor : public Input {
public:
Monitor();
Virtual ~Monitor();
// additional Monitor data specific functions varies as per //type 1,2 or 3
};

Class Input {
public:
Input();
Virtual ~Input();
};
```

Figure 9:- Control Monitor Pattern

4.3 Triple Modular Redundancy Pattern

Triple Modular Redundancy Pattern (TMR) is a pattern used to enhance reliability and safety in situations where there is no fail-safe state. The TMR pattern offers an odd number of channels i.e. three operating in parallel, each in effect checking the results of all the others. The computational results or resulting actuation signals are compared, and if there is a disagreement, then a two-out-of-three majority wins policy is invoked [4]. Figure 10 shows TMR pattern.

Figure 11 details about TMR pattern implementation in UML and sample code in C++.

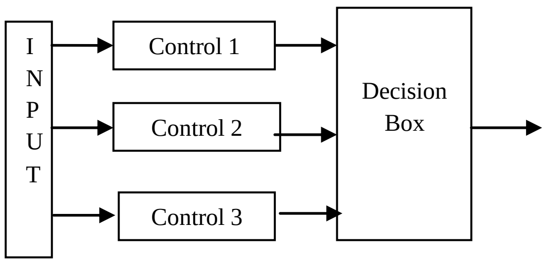
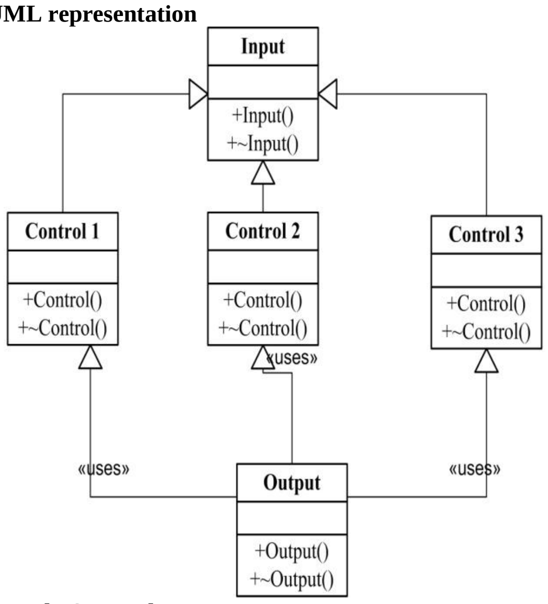


Figure 10:- TMR Pattern

//UML representation



```
//Sample C++ code
Class Output{
Public:
Output();
virtual ~Output();
//additional  function using Control and Monitor class
};
Class Control1 : public Input {
public:
Control1();
Virtual ~Control1();
};
Class Control2 : public Input {
public:
Control2();
Virtual ~Control2();
};
Class Control3 : public Input {
public:
Control3();
Virtual ~Control3();
};
Class Input {
public:
Input();
Virtual ~Input();};
```

Figure 11:- TMR class diagram

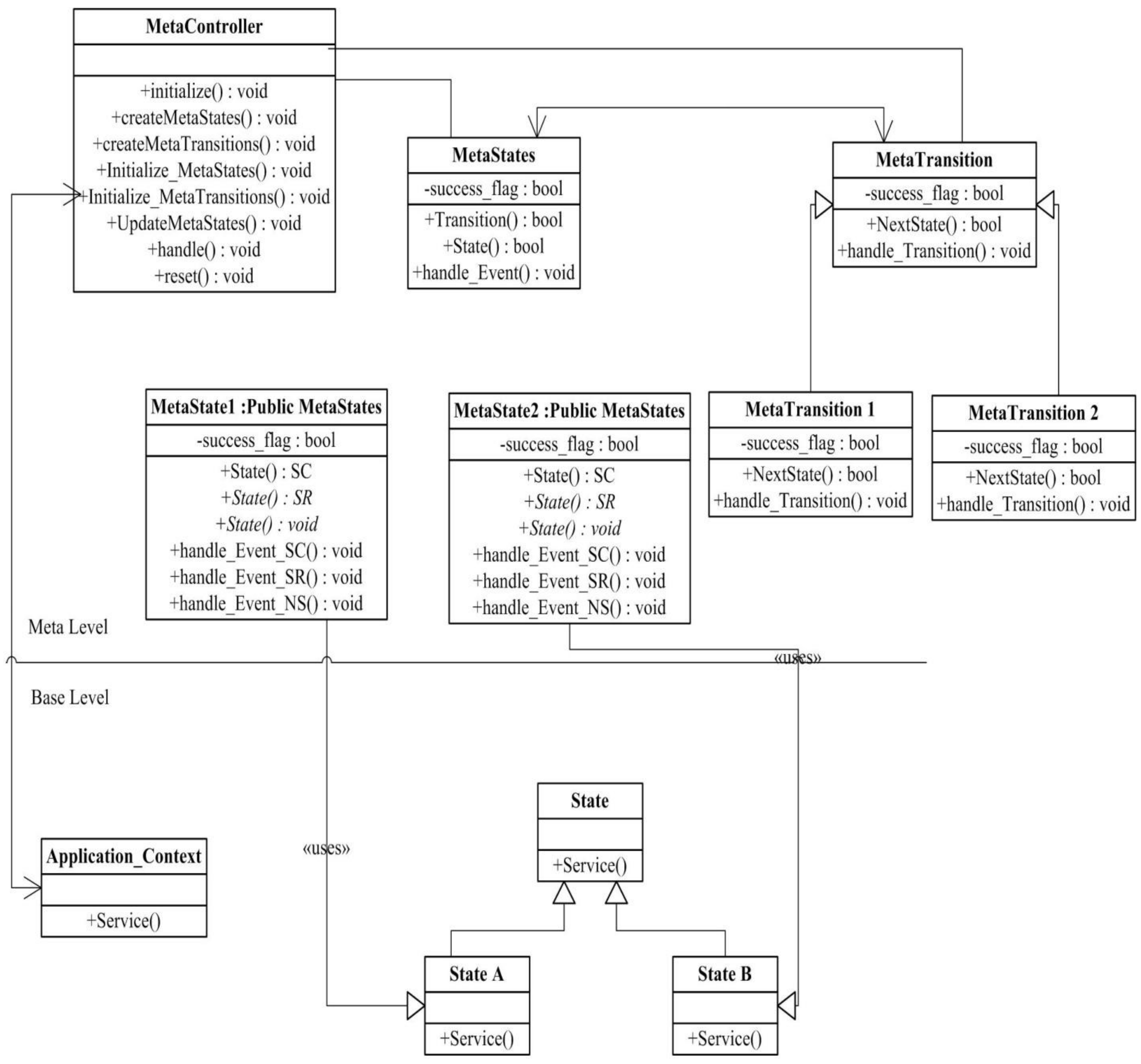


Figure 12:- Reflective State pattern

4.4 The Reflective State Pattern

Reflective State Pattern is a refinement of the state design pattern based on the reflection architectural pattern. The State design pattern presents a solution to implement state dependant behavior of a context object by means of state objects. It allows the context object to change its behavior dynamically using the delegation mechanism. The Reflection architectural pattern defines a software architecture that separates an application into two parts: the base level that implements the functional requirements, i.e., the application's logic and the meta-level which implements the control aspects [2]. Figure 12 gives class diagram for the reflective state pattern.

There are 3 main questions that should be considered in the state machine implementation:-

1. Where should the definition and initialization of the possible state objects be located?
2. How and where should the input events and guard-conditions be verified?
3. How and where should the execution of state transitions be implemented?

The implementation of the control aspect of state machine should be separated from the functional aspect. Classes should be loosely coupled to facilitate their reutilization and extension. Reflection architectural pattern separate the state pattern in to two levels, the meta-level and the base level. In the meta-level, the elements of the state diagram are represented by the Meta-State and the Meta-transition class hierarchies. The State machine's controller is represented by Meta-Controller class.

Class responsibilities are as below:

Meta-State: - This class is responsible for creating and initializing the state objects at the base level. Meta-State meta-object broadcasts the handling of the incoming event to its Meta-Transition meta-objects so that they can verify if a transition should be triggered.

Meta-Transition: - This subclass has information about transition function, has to perform actions associated with the transition. It has the reference to the next Meta-State that can be reached by the transition.

Meta-Controller: - This class is responsible for handling the intercepted service requests targeted to the context object at base level invoke through application. This class is responsible for creating and initialization of all Meta objects.

4.5 The Fault Tolerance Redundancy Pattern

Figure 13 gives the class diagram of fault tolerance redundancy pattern as described.

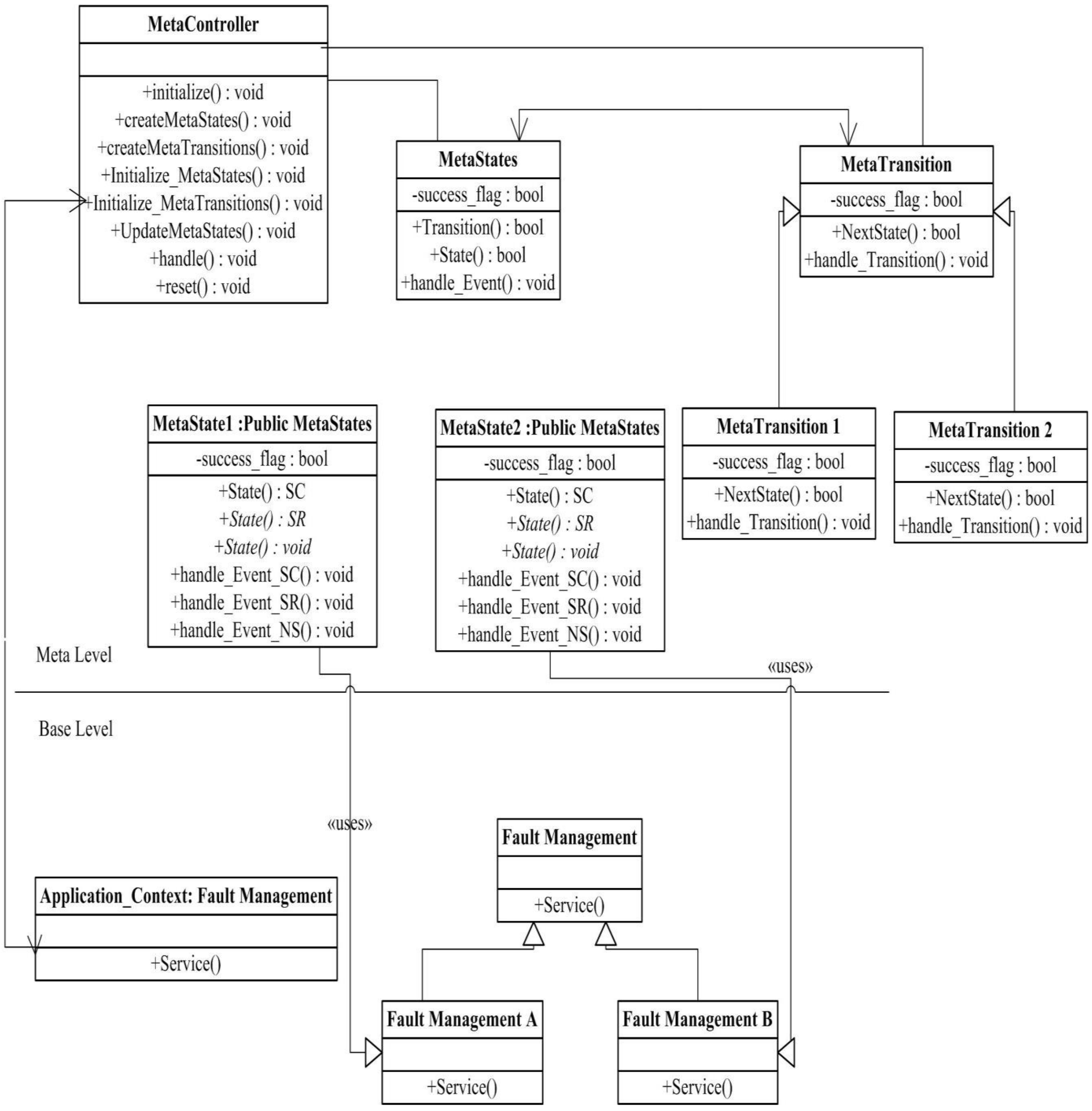


Figure 13:- Fault Tolerance Redundancy pattern

The reflection pattern can be modified to the fault tolerance domain. This pattern has same structure and semantics like reflection pattern with below difference.

The base level class represent the fault tolerant component (FTC) along with main context class invoke by application and the redundant components. FTC defines fault tolerance services while redundant class implements the same. Meta-Transition class implements recovery tests from fault. This can be implemented for n-versions of the state objects to enhance the system reliability and availability.

5. CONCLUSION

Design patterns are means of improving design quality, flexibility and productivity that can be fully exploited by the UML pattern.

In this paper we defined an object orient approach based on design pattern and computational reflection concept to implement non- functional requirements. This has facilitated the understandability of software architecture design, control-monitor safety pattern, Tri-Modular redundancy (TMR) pattern, reflective state pattern and fault tolerance redundancy patterns that can be use for safety and fault management as described in this paper.

The main goal of this paper is to separate control and safety aspect from the application logic is achieved by defining different pattern. These patterns can be readily used in similar applications with minor changes.

6. REFERENCES

[1] Toufik Taibi, Angel Herranz, Juan Jose Moreno – Navarro, “Stepwise Refinement Validation of Design Patterns formalized in TLA+ using TLC Model checker” Journal of Object Technology, Volume 8, No-2, March – April 2009

[2] Weihang Wu, Tim Kelly,” Safety Tactics for Software Architecture Design” COMPSAC '04 Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01

[3] Tashjian, B.M, “The failure modes and effects analysis as a design tool for nuclear safety systems,” Power Apparatus and Systems, IEEE Transactions on Volume:94,Issue: 1, Part: 1 Publication Year: 1975 , Page(s): 97 – 103

[4] Bocking, S, “Object-Oriented Network Protocols,” INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE Volume: 3, Digital Object Identifier:10.1109/INFCOM.1997.631153 Publication Year: 1997, Page(s): 1245 - 1252 vol.3

[5]http://my.safaribooksonline.com/book/softwareengineering-and-development/patterns/0201699567/safety- and-reliability-patterns/ch09lev1sec4

[6] Bidokhti, N,”FMEA Is Not Enough;” Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual Digital Object Identifier: 10.1109/RAMS.2009.4914698 Publication Year: 2009, Page(s): 333 – 337

[7] Jim Becker,” A Failure Mode And Effects Analysis (FMEA) Process For Distributed Computing Systems A Guidance Paper,” Integrating Error Models with Fault Injection, 1994. Third Workshop on Publication Year: 1994, Page(s): 39 – 40

[8] Trivedi Kishor, “Probability and Statistics with Reliability, Queuing, and Computer Science Applications,”

[9] Magnus Penker and Hans-Erik Eriksson; “Business Modeling With UML: Business Patterns at Work”.

[10] Hunt, J.E.; Price, C.J.; Lee, M.H” Automating the FMEA process,” Intelligent Systems Engineering. Volume: 2, Issue: 2 Publication Year: 1993, Page(s): 119 – 132

[11] Duell, M,” Looking beyond software to understand software design patterns, Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International Digital-Object-dentifier:10.1109/CMPSAC.1999.812724 Publication Year: 1999 , Page(s): 312 - 313

[12] Masuda, G.; Sakamoto, N.; Ushijima, K," Redesigning of an Existing Software using Design Patterns", Principles of Software Evolution, 2000. Proceedings. International Symposium on Digital Object Identifier: 10.1109/ISPSE.2000.913234 Publication Year: 2000 , Page(s): 165 - 169

[13] Fuping Zeng; Aizhen Chen; Xin Tao;"Study on Software Reliability Design Criteria Based on Defect Patterns", Reliability, Maintainability and Safety, 2009. ICRMS 2009. 8th International Conference on Digital Object Identifier: 10.1109/ICRMS.2009.5270095

[14] Luciane Lamour Ferreira and Cecília Mary Fischer Rubira ,“Reflective Design Patterns to implement Fault Tolerance