A New Approach to the Quantitative Measurement of Software Flexibility

Abbas Rasoolzadegan
Ferdowsi University of Mashhad, Mashhad, Iran, rasoolzadegan@um.ac.ir

***Abstract*—Software flexibility is the ease with which a software system can be modified for use in applications or environments other than those for which it was specifically designed. Software flexibility is not an absolute term. It is an important aspect of software quality. Quantifying software flexibility is increasingly becoming necessary. We have recently proposed a new approach (referred to as $SDA_{Flex\&Rel}$) to the development of reliable yet flexible software. In this paper, a new approach is proposed to quantitatively measure the flexibility of the software developed using $SDA_{Flex\&Rel}$, thereby making precise informal claims on the flexibility improvement. Moreover, the effectiveness of the proposed measurement approach is empirically investigated in the multi-lift case study that has already been conducted to demonstrate the feasibility of $SDA_{Flex\&Rel}$. The results confirm the flexibility improvement promised by $SDA_{Flex\&Rel}$.**

***Keywords*— design patterns, flexibility, quantitative measurement, software metrics.**

I. INTRODUCTION

Flexibility can be defined as the ability of a system to respond to potential internal or external changes affecting its value delivery, in a timely and cost-effective manner. Thus, flexibility for an engineering system is the ease with which the system can respond to uncertainty in a manner to sustain or increase its value delivery [11], [18]. Uncertainty is a key element in the definition of flexibility. Uncertainty can create both risks and opportunities in a system, and it is with the existence of uncertainty that flexibility becomes valuable.

Rapid technological developments pervade every aspect of daily life, having a direct effect on the software we use. Every element of the software's operational environment is in a state of constant flux: Frequent changes in the hardware, operating system, cooperating software, and client's expectations are motivated by performance improvements, bug-fixes, security breaches, and attempts to assemble synergistically ever more sophisticated software systems [7]. Classic and contemporary literature in software design recognizes the central role of flexibility in software design and implementation. Structured design, modular design, object-oriented design, software architecture, design patterns, and component-based software engineering, among others, seek to maximize flexibility. Textbooks about software design emphasize the flexibility of particular choices, thereby implying the superiority of the design policy they advocate. But despite the progress made since the earliest days of software engineering, from the 'software crisis' through 'software's chronic crisis', evolution (formerly 'maintenance') of industrial software systems has remained unpredictable and notoriously expensive, often exceeding the cost of the development phase. Flexibility has therefore become a central concern in software design and in many related aspects in software engineering research [8], [10], [16-17], [19].

An artifact is hardly flexible in absolute terms [1-2]. Instead, it may be flexible towards a specific class of changes and inflexible towards another one. Predicting the class of changes is the key to understanding software flexibility. Moreover, an artifact A is more flexible than another artifact B towards a particular evolution step if the number of changes required for A is less than those required for B. 'Evolution step' is regarded as the unit of evolution with relation to a particular class of changes in design or implementation [3-4].

We have recently proposed a Software Development Approach (SDA). This approach, referred to as $SDA_{Flex\&Rel}$ in this paper, promises to develop reliable yet flexible software [5]. In $SDA_{Flex\&Rel}$, formal (Object-Z) and semi-formal (UML) models are transformed into each other using a set of bidirectional formal rules. Formal modeling and refinement in Object-Z ensure the reliability of software. Visual models (UML diagrams) facilitate the interactions among stakeholders who are not familiar enough with the complex mathematical concepts of formal modeling methods. Applying design patterns to visual models improves the flexibility of software. The transformation of formal and visual models into each other through the iterative and evolutionary process, proposed in [5], helps develop the software applications that need to be highly reliable yet flexible. The workflow of $SDA_{Flex\&Rel}$ is illustrated in Fig. I.

In this paper, we quantitatively measure the flexibility improvement promised by $SDA_{Flex\&Rel}$ and empirically investigate such improvement in the multi-lift case study that has already been conducted to demonstrate the feasibility of $SDA_{Flex\&Rel}$. Reference [6] elaborately presents the results of applying $SDA_{Flex\&Rel}$ to the multi-lift system.

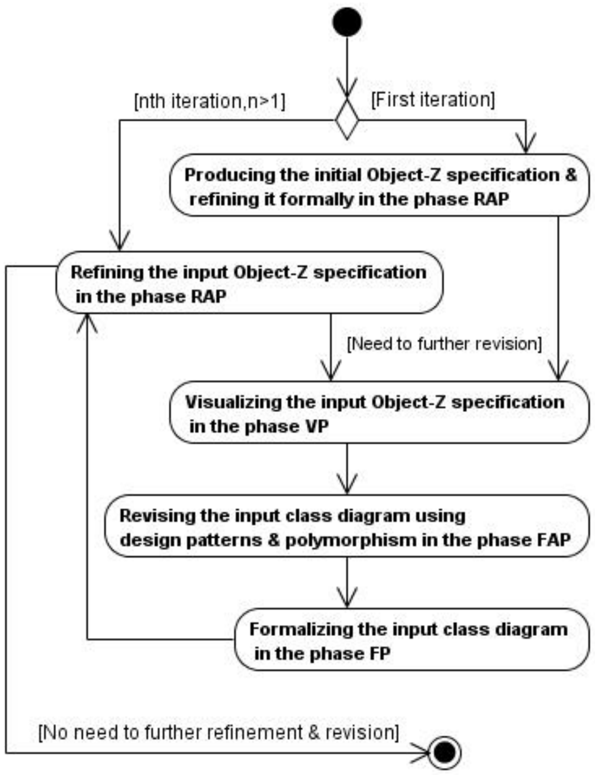
[nth iteration,n>1]
[First iteration]
Producing the initial Object-Z specification & refining it formally in the phase RAP
Refining the input Object-Z specification in the phase RAP
[Need to further revision]
Visualizing the input Object-Z specification in the phase VP
Revising the input class diagram using design patterns & polymorphism in the phase FAP
Formalizing the input class diagram in the phase FP
[No need to further refinement & revision]

Fig. 1. The workflow of $SDA_{Flex\&Rel}$ using UML activity diagram.

The iterative and evolutionary process illustrated in Fig. 1 continues until a final product with a desired quality is achieved. Fig. 2 illustrates the details of an iteration of $SDA_{Flex\&Rel}$ which consists of the following phases:

- Reliability Assurance Phase (RAP) which supports formal specification and refinement in Object-Z.
- Visualization Phase (VP) which transforms Object-Z models into UML ones.
- Flexibility Assurance Phase (FAP) which revises UML models from the viewpoints of design patterns and polymorphism.
- Formalization Phase (FP) which transforms UML models into Object-Z ones.

In the phase FAP of the proposed approach, the flexibility of the software being developed improves using Software Engineering (SE) principles such as design patterns. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

The flexibility of the software developed using $SDA_{Flex\&Rel}$ is directly proportional to the flexibility of those design patterns used in the phase FAP during the different iterations of the development process. Therefore, to quantify the software flexibility, we can quantitatively measure the flexibility of the design patterns used during the development process [7].

To quantify flexibility and make precise informal claims on the flexibility of design patterns, a notion called 'evolution complexity' can be used. The complexity of an evolution step measures how inflexible is the design/implementation being evolved towards a particular class of changes. The fewer the changes are required, the more flexible it is. As illustrated in Fig. 3, software

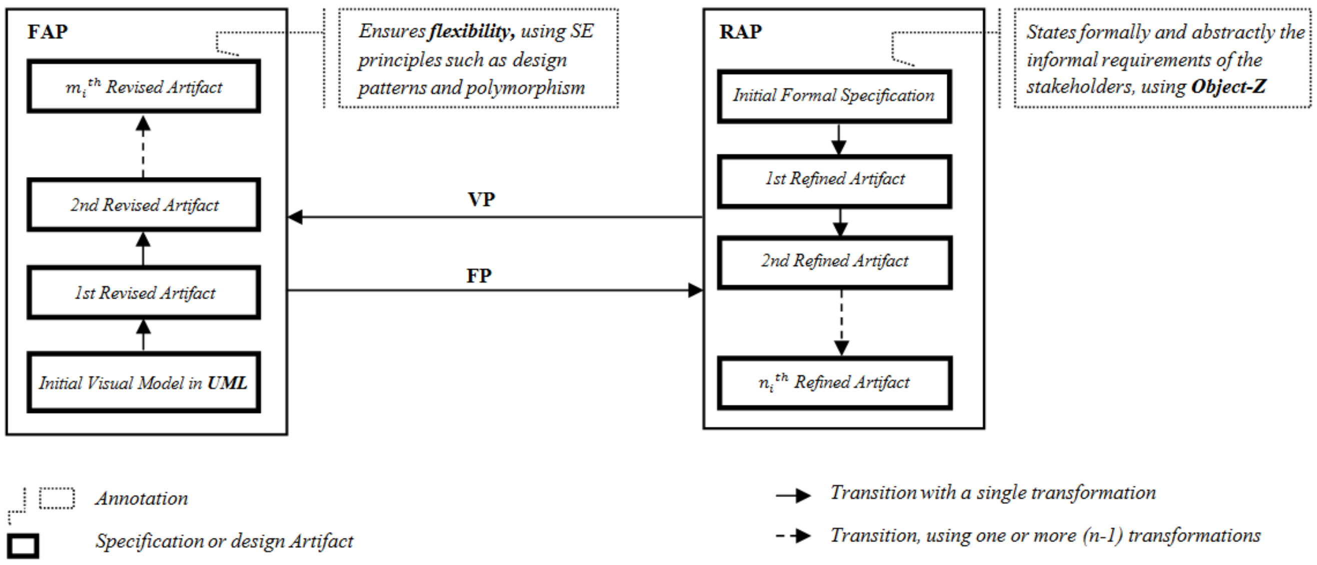
FAP
Ensures flexibility, using SE principles such as design patterns and polymorphism
m_i^{th} Revised Artifact
2nd Revised Artifact
1st Revised Artifact
Initial Visual Model in UML
VP
FP
RAP
States formally and abstractly the informal requirements of the stakeholders, using Object-Z
Initial Formal Specification
1st Refined Artifact
2nd Refined Artifact
n_i^{th} Refined Artifact
Annotation
Specification or design Artifact
Transition with a single transformation
Transition, using one or more (n-1) transformations

Fig. 2. A schematic view of an iteration *i* of $SDA_{Flex\&Rel}$.

evolution can be described as the process during which changes occur in an old problem, which entail changes in the corresponding design/implementation. To distinguish between changes in problems and changes in the corresponding designs/implementations, we refer to the former as shifts and to the latter as adjustments, jointly represented as an evolution step [8].

Let us represent the set of problems as $\mathbb{P}$ and the set of designs/implementations as $\mathbb{DI}$. An evolution step can be represented as a mapping of the combination of the old problem $p_{old} \in \mathbb{P}$, the shifted problem $p_{\text{shifted}} \in \mathbb{P}$, and the old design/implementation $di_{old} \in \mathbb{DI}$ into the adjusted design/implementation $di_{adjusted} \in \mathbb{DI}$. This mapping can thus be represented as a mathematical function $\mathcal{E}$, called the evolution function. This function maps each tuple $< p_{old}, p_{\text{shifted}}, di_{old} >$ to $di_{adjusted}$, such that:

$$\mathcal{E}: \mathbb{P} \times \mathbb{P} \times \mathbb{DI} \to \mathbb{DI}, \tag{1}$$

$$\mathcal{E}(p_{old}, p_{\text{shifted}}, di_{old}) = di_{adjusted}$$

Where the old design/implementation di_{old} realizes p_{old}, and $di_{adjusted}$ realizes p_{shifted}. Therefore, an evolution step can be formulated as:

$$\mathcal{E} = \ll p_{old}, p_{\text{shifted}}, di_{old} >, \mathcal{E}(p_{old}, p_{\text{shifted}}, di_{old}) > \tag{2}$$

We can measure flexibility in terms of the cost of the evolution process. ‘Evolution cost metric’ ($C^{\mu}_{Modules}$) measures the cost of executing an evolution step $\mathcal{E} = \ll p_{old}, p_{\text{shifted}}, di_{old} >, di_{adjusted} >$ in terms of the software complexity $\mu(m)$ of each module m affected by the adjustments [8-9]:

$$C^{\mu}_{Modules}(\mathcal{E}) \triangleq \sum_{m \in \Delta Modules(di_{old}, di_{adjusted})} \mu(m) \tag{3}$$

where μ can be any software complexity metric such as LoC (lines of code) or CC (Cyclomatic Complexity), and $\Delta Modules(di_{old}, di_{adjusted})$ designates the symmetric set-difference between the set of modules in di_{old} and the set of modules in $di_{adjusted}$, namely:

$$(Modules(di_{old}) \backslash Modules(di_{adjusted})) \cup \tag{4}$$
$$(Modules(di_{adjusted}) \backslash Modules(di_{old}))$$

The evolution complexity of a design/implementation di towards an evolution step ($\mathcal{E}$) is formulated as $O(C^{\mu}_{Modules}(\mathcal{E}))$. If the evolution complexity of di towards $\mathcal{E}$ is fixed and independent of its size ($O(C^{\mu}_{Modules}(\mathcal{E})) = 1$), di is flexible towards $\mathcal{E}$, but if the evolution complexity of di directly or indirectly grows as a function of the size of di, it is inflexible towards $\mathcal{E}$. It is worth mentioning that evolution complexity does not measure the actual cost of the evolution processes requires, but how it grows. We can quantify the flexibility of each design pattern towards specified evolution steps by calculating the corresponding evolution complexity.

The rest of this paper is organized as follows: In section two, the evolution complexity of each of the design patterns that have been used during the development of the multi-lift system is calculated. Section three discusses the conclusions.

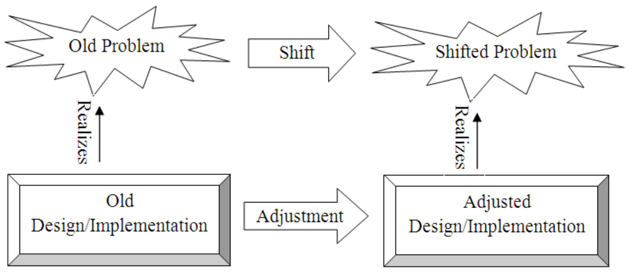

Fig. 3. An evolution step [8].

II. QUANTIFYING THE FLEXIBILITY OF THE MULTI-LIFT SYSTEM DEVELOPED USING SDA$_{\text{FLEX\&REL}}$

A non-trivial case study, called the multi-lift system, has been taken as a test bed to evaluate the feasibility of SDA$_{\text{Flex\&Rel}}$. This system includes parallel, distributed, embedded, and real-time software. A detailed report of this empirical study has been presented in [6]. Such a system needs high reliability and flexibility. As an instance, the dispatching strategy should be continuously updated for each lift according to some criteria such as manager policies and traffic modes, which dynamically change. These variable factors increase the necessity of designing a flexible controller having the potential to change the control strategy dynamically.

In the process of developing the multi-lift system using SDA$_{\text{Flex\&Rel}}$, the Observer, Strategy, and Mediator design patterns have been used during the phase FAP to improve the system flexibility. In this section, we investigate the usability of the evolution cost metric for corroborating informal claims on the flexibility of these design patterns.

A. *Observer Pattern*

The applications of the Observer pattern are [10]:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets vary and reuse them independently.
- When a change to one object requires changing others, and you do not know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

If at least one of the above-mentioned conditions holds in a part of software design, this part has the potential to be revised by the Observer pattern. This pattern defines a one-to-many dependency between one object named subject and its dependent objects, referred to as observers. All observers are notified and updated automatically once the state of the corresponding subject changes. Fig. 4 illustrates a part of the initial class diagram of the multi-lift system. As illustrated in the left column of Fig. 4, there are three dependencies between the objects of this part:

1. Whenever the traffic information (trafficinfo), managed by TrafficManager, changes, the value of traffic features (objects of TrafficFeature) should be updated using the method MeasureFeature.
2. Whenever the value of a traffic feature is updated, the suitability percentage of traffic modes (objects of TrafficMode) should be updated by the method CalculateSuitabilityPercentage.
3. Once the suitability percentage of a traffic mode is updated, the method CalculateCurrentTrafficMode of the class ControlStrategyGenerator determines the current traffic mode.

According to the applications of the Observer pattern, this part has the potential to be revised using this pattern. The right column of Fig. 4 illustrates the revised version. In the Observer pattern, subjects implicitly know their observers. Any number of objects can observe a subject. Observers can be attached to subjects or be detached from them through the interface of subjects. Each subject sends a notification to its observers through calling their Update method whenever a change occurs to make the state of its observers consistent with its own. Moreover, an observer may ask the subject for information to reconcile its state with the state of the subject. This pattern claims that:

- It minimizes the coupling between a subject and its observers. A subject has the list of its observers. These observers conform to the interface of an abstract class named Observer. The subject knows only Observer, not all concrete classes of Observer.
- It provides broadcast communication. A subject automatically broadcasts notifications to all its observers. The subject does not know how many dependent objects exist. It is only responsible for broadcasting notifications. Therefore, observers can be added or removed at any time in a flexible way.

According to (3), we use C^1_{Classes} to corroborate the above-mentioned claims and to make them precise. In other words, we assume that the cost of adding, removing, or changing each modular unit (m) is equal to 1 ($\mu(m) = 1$). Moreover, 'class' is assumed as the modular unit. Thus, the evolution cost metric is estimated by calculating the

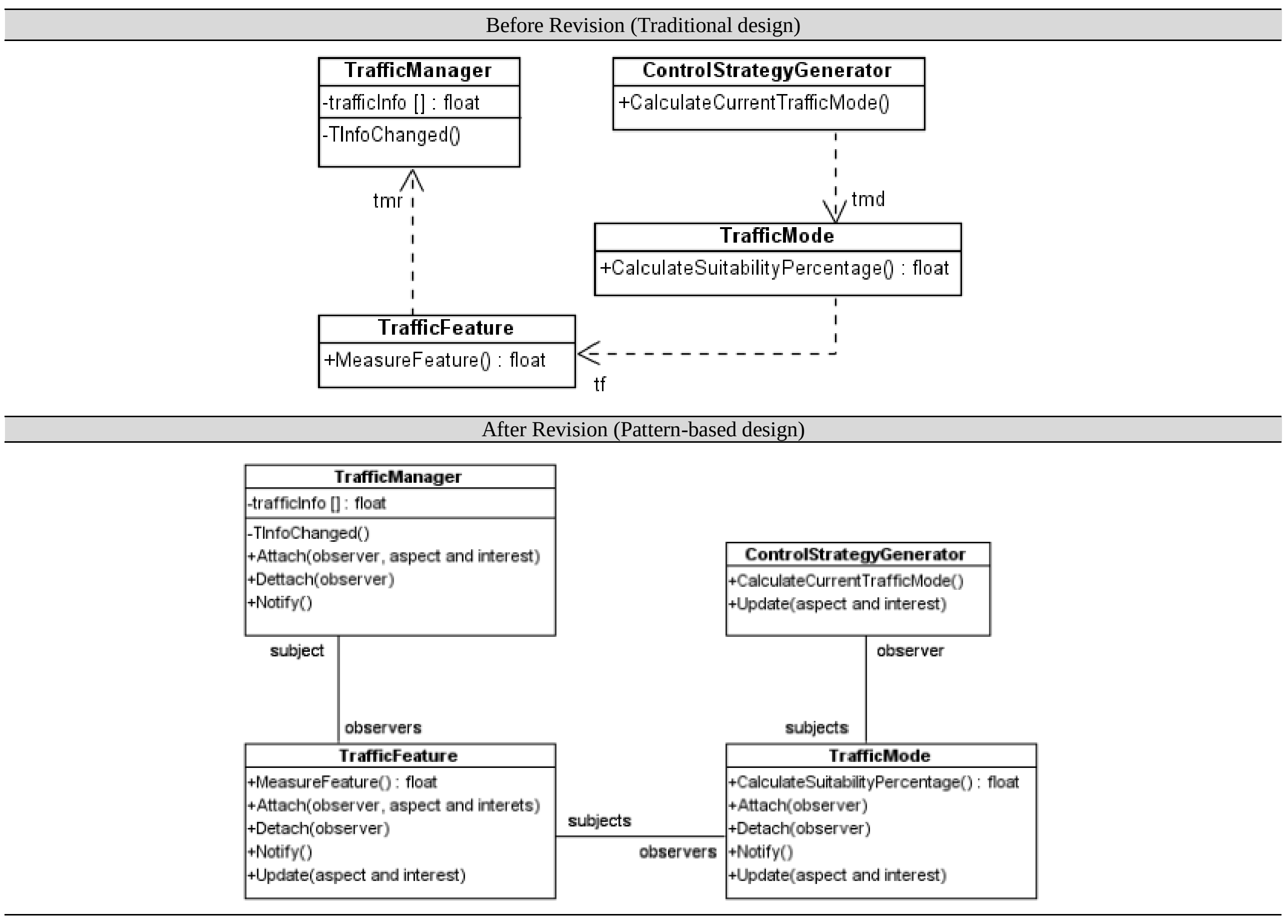
Before Revision (Traditional design)
TrafficManager
-trafficInfo [] : float
-TInfoChanged()
ControlStrategyGenerator
+CalculateCurrentTrafficMode()
tmr
tmd
TrafficMode
+CalculateSuitabilityPercentage() : float
TrafficFeature
+MeasureFeature() : float
tf
After Revision (Pattern-based design)
TrafficManager
-trafficInfo [] : float
-TInfoChanged()
+Attach(observer, aspect and interest)
+Dettach(observer)
+Notify()
ControlStrategyGenerator
+CalculateCurrentTrafficMode()
+Update(aspect and interest)
subject
observers
observer
subjects
TrafficFeature
+MeasureFeature() : float
+Attach(observer, aspect and interets)
+Detach(observer)
+Notify()
+Update(aspect and interest)
subjects
observers
TrafficMode
+CalculateSuitabilityPercentage() : float
+Attach(observer)
+Detach(observer)
+Notify()
+Update(aspect and interest)

Fig. 4. First revision of the initial class diagram of the multi-lift system using the Observer pattern.

number of the classes that are added, removed, or adjusted as a result of the evolution. The results of this analysis are summarized in Table 1.

TABLE I
THE COMPLEXITY OF EVOLVING THE OBSERVER PATTERN VS. TRADITIONAL DESIGN TOWARDS SHIFTS IN OBSERVERS AND SUBJECTS

Evolution step / Design policy	Change/Add/Remove observer	Change/Add/Remove subject
Observer pattern	$O(1)$	$O(1)$
Traditional design ('anti-pattern')	$O(\theta \times \lvert\text{Subjects}\rvert)$ $0<\theta\leq 1$	$O(\theta \times \lvert\text{Observers}\rvert)$ $0<\theta\leq 1$

The results show that the complexity of evolving the Observer pattern or each Observer-based design towards shifts in observers ($\mathcal{E}_1$) and subjects ($\mathcal{E}_2$) is fixed ($O(1)$) because a subjects knows only the abstract class of its observers, not all its concrete observers. Therefore, we can conclude that the Observer pattern as well as each design based on this pattern (such as the design illustrated in the right column of Fig. 4) is flexible towards both $\mathcal{E}_1$ and $\mathcal{E}_2$. As shown in Table 1, the evolution complexity of a traditional design (such as the design illustrated in the left column of Fig. 4) towards $\mathcal{E}_1$ and $\mathcal{E}_2$ is proportional to the number of subjects ($\theta \times |\text{Subjects}|$) and observers ($\theta \times |\text{Observers}|$), respectively. As a result, it is inflexible towards both $\mathcal{E}_1$ and $\mathcal{E}_2$.

B. *Strategy Pattern*

We can use the Strategy pattern when [10]:

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- An algorithm uses data that clients should not know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

If at least one of the above-mentioned conditions holds in a part of the initial design of software, this part has the potential to be revised by the Strategy pattern. This pattern configures a class named *context* with one of several behaviors. Fig. 5 illustrates another part of the initial class diagram of the multi-lift system. As illustrated in this figure, the central controller (the class *CentralController*) contains an external request allocator (the class *ExternalRequestAllocator*). The role of such an allocator is to select the most suitable lift to respond to the current external request according to some parameters such as current values of the evaluation criteria (objects of the class *EvaluationCreteria*).

There are different strategies to respond to external requests according to various parameters such as managers' policies (the association class *ManagerPolicy*) and the current traffic mode. These strategies need to change at run time according to values of the above-mentioned parameters. In order to meet the required flexibility for changing these strategies at run time, this part of the class diagram has been revised based on the Strategy pattern. The Strategy design pattern claims that:

- It provides a family of algorithms and behaviors as hierarchies of strategy classes for contexts to extend reusability.
- It provides an alternative for subclassing. It encapsulates various algorithms in distinct strategy classes. This makes the algorithms have the ability to change or extend independently of the contexts easily.
- It eliminates conditional statements that are used for the selection of the desired behavior by encapsulating behavior in discrete strategy classes.

To measure the flexibility of the strategy design pattern,

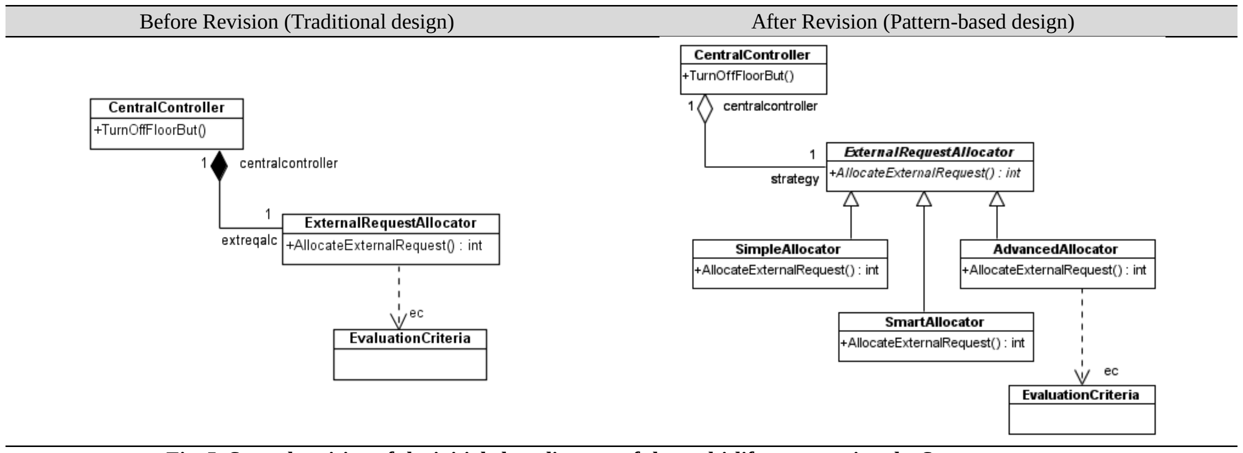

Fig. 5. Second revision of the initial class diagram of the multi-lift system using the Strategy pattern.

we assume that the cost of adding, removing, or changing a modular unit m is proportional to the number of those statements of m that are added, removed, or adjusted as a result of the evolution ($\mu(m) = LoS$ (Lines of Statements)). Moreover, 'class' is assumed as the modular unit. Thus, the evolution cost metric is estimated by calculating the number of the statements that are added, removed, or adjusted as a result of the evolution. We use $C_{Classes}^{LoS}$ to corroborate the above-mentioned claims and to make them precise. The results of this analysis are summarized in Table II.

The results show that the complexity of evolving the Strategy pattern as well as each Strategy-based design towards shifts in strategies ($\mathcal{E}$) is fixed ($O(1)$) because the strategies can be changed or extended independently of the contexts. Therefore, we can conclude that the Strategy pattern or each design based on this pattern (such as the design illustrated in the right column of Fig. 5) is flexible towards $\mathcal{E}$.

As shown in Table II, the evolution complexity of a traditional design (such as the design illustrated in the left column of Fig. 5) towards $\mathcal{E}$ is proportional to the number of strategies ($\theta \times |Strategies|$) because of the corresponding conditional statements, so it is inflexible towards $\mathcal{E}$.

TABLE II
THE COMPLEXITY OF EVOLVING THE STRATEGY PATTERN VS. 'SWITCH' OR 'MULTIPLE CONDITIONAL' STATEMENTS TOWARDS SHIFTS IN STRATEGIES

Evolution step / Design policy	Change/Add/Remove observer
Strategy pattern	O(1)
'Switch' or 'multiple conditional' statements	$(\theta \times \|Strategies\|)$ $0<\theta\leq1$

C. Mediator Pattern

We may use the Mediator pattern when [10]:

- A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- The behavior distributed between several classes should be customizable without a lot of subclassing.

If at least one of the above-mentioned conditions holds in a part of the initial design of software, this part has the potential to be revised by the Mediator pattern. The Mediator pattern defines an object named *mediator*. This object encapsulates how a set of objects, referred to as *colleagues*, interact.

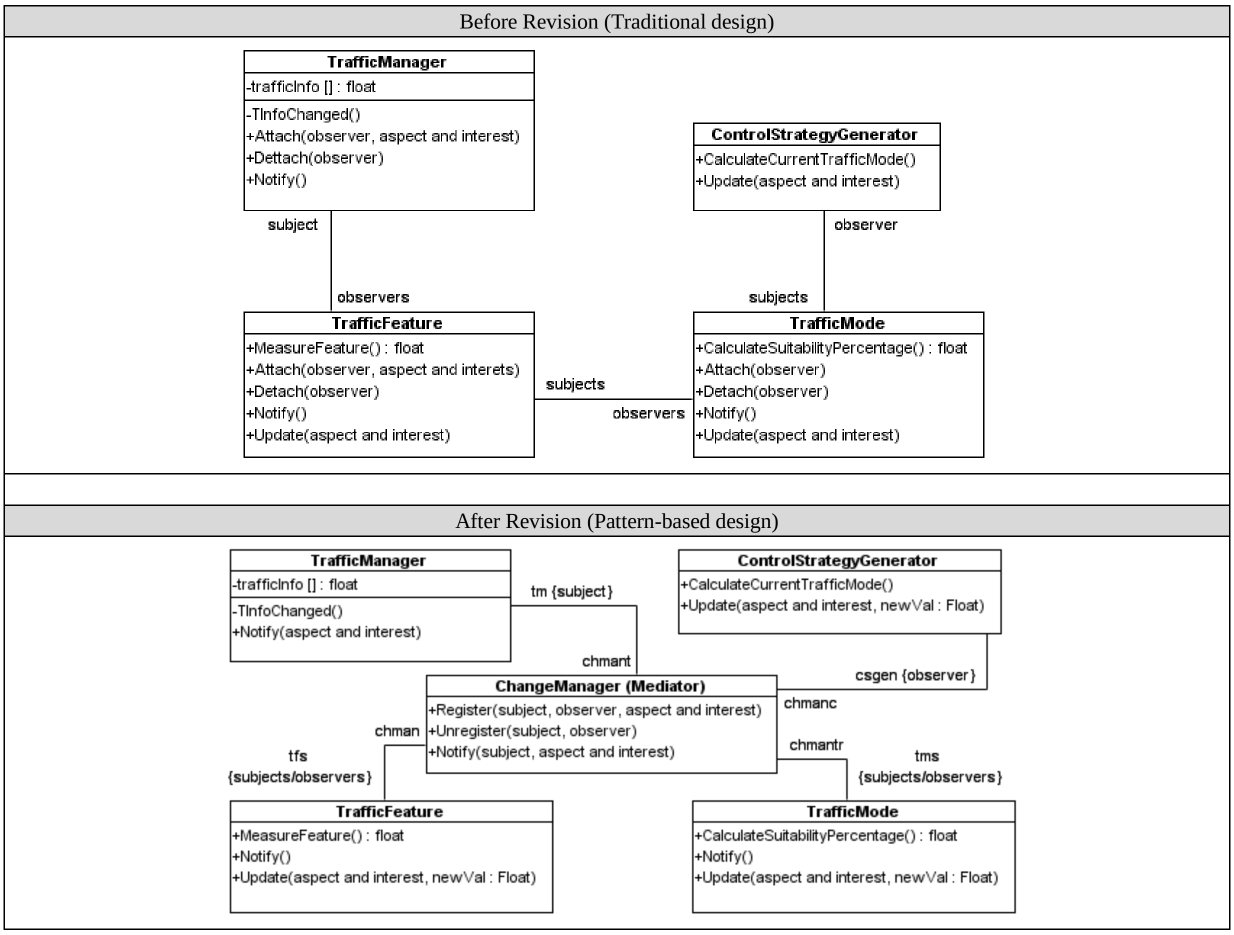
Before Revision (Traditional design)
TrafficManager
-trafficInfo [] : float
-TInfoChanged()
+Attach(observer, aspect and interest)
+Dettach(observer)
+Notify()
subject
observers
ControlStrategyGenerator
+CalculateCurrentTrafficMode()
+Update(aspect and interest)
observer
subjects
TrafficFeature
+MeasureFeature() : float
+Attach(observer, aspect and interets)
+Detach(observer)
+Notify()
+Update(aspect and interest)
subjects
observers
TrafficMode
+CalculateSuitabilityPercentage() : float
+Attach(observer)
+Detach(observer)
+Notify()
+Update(aspect and interest)
After Revision (Pattern-based design)
TrafficManager
-trafficInfo [] : float
-TInfoChanged()
+Notify(aspect and interest)
tm {subject}
chmant
ControlStrategyGenerator
+CalculateCurrentTrafficMode()
+Update(aspect and interest, newVal : Float)
csgen {observer}
chmanc
ChangeManager (Mediator)
+Register(subject, observer, aspect and interest)
+Unregister(subject, observer)
+Notify(subject, aspect and interest)
chman
tfs
{subjects/observers}
chmantr
tms
{subjects/observers}
TrafficFeature
+MeasureFeature() : float
+Notify()
+Update(aspect and interest, newVal : Float)
TrafficMode
+CalculateSuitabilityPercentage() : float
+Notify()
+Update(aspect and interest, newVal : Float)

Fig. 6. Third revision of the initial class diagram of the multi-lift system using the Mediator pattern.

The diagram illustrated in the left column of Fig. 6 has already been revised using the Observer pattern (in Fig. 4). The flexibility of this part is improved further, using the Mediator pattern. An object named *ChangeManager* is introduced when the coupling between subjects and observers is complex. This object, as an instance of the Mediator pattern, is to keep these complex relationships. The main responsibilities of this object are 1) it defines an interface to connect a subject to its observers and manages this relationship. This omits the need for subjects to know their observers explicitly and vice versa, 2) it defines a straightforward update strategy and 3) it notifies and updates all related observers at the request of corresponding subject. The right column of Fig. 6 illustrates the newly revised version of this part after applying the Mediator pattern.

The Mediator design pattern claims that:

- It makes changing behavior easy through subclassing the mediator object without changing its colleagues.
- A mediator object decreases the coupling between its colleagues. Therefore, they can be varied and reused independently.
- Many-to-many interactions among the colleagues of a mediator object are replaced with one-to-many interactions between the mediator object and its colleagues. Understanding, maintenance, and extension of one-to-many relationships are easier, compared to many-to-many ones.

We use $\mathrm{C}^1_{\mathrm{Classes}}$ to corroborate these claims and to make them precise. The results of this analysis are summarized in Table III.

The results show that the complexity of evolving the Mediator pattern or each Mediator-based design towards shifts in behavior ($\mathcal{E}_1$), colleagues ($\mathcal{E}_2$), and relationships ($\mathcal{E}_3$) is fixed ($\mathrm{O}(1)$) because mediators and colleagues can be changed independently. Therefore, we can conclude that the Mediator design pattern or each design based on it is flexible towards $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_3$. As shown in Table III, the evolution complexity of a traditional design towards these three evolution steps ($\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$) is directly proportional to the number of colleagues ($\theta \times |\mathrm{Colleagues}|$), so it is inflexible towards $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_3$ because of the coupling between mediators and colleagues.

The results of the aforementioned analyses show that the revision of the initial class diagram of the multi-lift system using the Observer, Strategy, and Mediator patterns during the phase FAP of $\mathrm{SDA}_{\mathrm{Flex\&Rel}}$ improves the flexibility of the system. The flexibility is quantified using the evolution cost metric through calculating the complexity of evolution steps. In other words, the flexibility improvement claimed by these three design patterns is corroborated by the evolution cost metric. There is a direct relationship among the value of the evolution cost, the evolution complexity, and the flexibility of a design towards a particular evolution step [12-15], [18]. It is worth mentioning that there is no limitation on the application domain of the proposed method in measuring the flexibility of design patterns. As previously mentioned, the reason of selecting the three patterns Strategy, Mediator, and Observer is the design requirements of the multi-lift system used as the case study.

TABLE III
THE COMPLEXITY OF EVOLVING THE MEDIATOR PATTERN VS. TRADITIONAL DESIGN TOWARDS SHIFTS IN BEHAVIOR, COLLEAGUES, AND RELATIONSHIPS

Design policy Evolution step	Mediator pattern	Traditional design ('anti-pattern')
Change behavior	$\mathrm{O}(1)$	$O(\theta \times \lvert\mathrm{Colleagues}\rvert)$ $0 < \theta \leq 1$
Change colleague	$\mathrm{O}(1)$	$O(\theta \times \lvert\mathrm{Colleagues}\rvert)$ $0 < \theta \leq 1$
Extend relationships between colleagues	$\mathrm{O}(1)$	$O(\theta \times \lvert\mathrm{Colleagues}\rvert)$ $0 < \theta \leq 1$

III. CONCLUSION

In this paper, we quantify the flexibility improvement promised by the software development approach $\mathrm{SDA}_{\mathrm{Flex\&Rel}}$, which has recently been proposed to develop reliable yet flexible software. This approach improves software flexibility through preparing the ground for the visual revision of the structure and the behavior of the software being developed using design patterns. In such a case, software flexibility is directly proportional to the flexibility of those design patterns used during its development process. Therefore, to quantify the flexibility of software, the flexibility of each of the design patterns used during the development process of the software is quantitatively measured by calculating the complexity of evolution steps through the evolution cost metric. As an empirical study, the flexibility of the multi-lift system that has already been developed using $\mathrm{SDA}_{\mathrm{Flex\&Rel}}$ is quantified. The results confirm the promised flexibility improvement

ACKNOWLEDGMENT

This work was supported in part by: Ferdowsi University of Mashhad's Research Council, under grant #28190 (dated 14-11-92) on Static Fine Job.

REFERENCES

[1] J. Niu, "A Measurement Method of Software Flexibility Based on BP Network," in *Proc. Int. Workshop on Intelligent Systems and Applications (ISA)*, 2009, pp. 1-4.

[2] S. PENG, "User-Oriented Measurement of Software Flexibility," in

Proc. World Congress on Computer Science and Information Engineering, 2009, PP. 629-633.

[3] R. Martinho, “A Two-Step Approach for Modeling Flexibility in Software Processes,” in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, Italy, 2008, pp. 427-430.

[4] H. Oliver, O. Philipp, and B. Udo. (2010, Jan.). Improving Software Flexibility for Business Process Changes. *Business & Information Systems Eng.* [Online]. *2(1)*, pp. 3-13. Available: http://link.springer.com/article/10.1007/s12599-009-0086-8

[5] A. Rasoolzadegan and A. Abdollahzadeh. (2014, Jul.). Reliable yet Flexible Software through Formal Model Transformation (Rule Definition). *Journal of Knowledge and Information Systems (KAIS)*, [Online]. *40 (1)*, PP. 79-126. Available: http://link.springer.com/article/10.1007/s10115-013-0621-2

[6] A. Rasoolzadegan and A. Abdollahzadeh, “Specifying a Parallel, Distributed, Real-Time, and Embedded System: Multi-Lift System Case Study,” Information Technology and Computer Eng. Faculty, Amirkabir Univ. Technology, Tehran, Iran, Tech. Rep., 2011.

[7] H. B. Christensen, “Flexible, Reliable Software: Using Patterns and Agile Development,” Chapman and Hall/CRC, 1st ed., 2010.

[8] H. Eden and T. Mens. (2006, Jun.). Measuring Software Flexibility. *IEE Software*. [Online]. *153(3)*, pp. 113-126. Available:

http://ieeexplore.ieee.org/document/1645518/

[9] G. H. Z. LI. (2008, Apr.). Research on Flexibility Metrics in Software Architecture Level. *Computer Science.* [Online]. *35 (4)*. pp. 259-264. Available:

http://en.cnki.com.cn/Article_en/CJFDTOTAL-JSJA200804078.htm

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Pattern: Elements of Reusable Object-Oriented Software,” Addison-Wesley Publishing Company, Fifth ed., 1995.

[11] A. M. Ross, D. H. Rhodes, and D. E. Hastings. (2008, Apr.). Defining Changeability: Reconciling Flexibility, Adaptability, Scalability, Modifiability, and Robustness for Maintaining System Lifecycle Value. *Systems Engineering.* [Online]. *11(3)*, pp. 246–262. Available:

http://onlinelibrary.wiley.com/doi/10.1002/sys.20098/abstract

[12] R. S. A. DeLoach and V. A. Kolesnikov, “Using Design Metrics for Predicting System Flexibility,” in *Proc. Fundamental Approaches to Software Engineering*, 2006, pp. 184-198.

[13] T. Sasaki, N. Yoshioka, Y. Tahara, and A. Ohsuga, “Evaluation of Flexibility to Changes Focusing on the Variable Structures in Legacy Software,” in *Proc. Knowledge-Based Software Engineering: 11th Joint Conference*, 2014, pp. 252–269.

[14] Y. Gao and Y. Yang, “Flexibility of Software Development Method,” in *Proc. Advances in Intelligent and Soft Computing*, 2011, pp. 383-387.

[15] Y. Wang, M. Jia, J. Guo, and B. Zhang, “Evaluating Model of Software Flexibility of Domestic Foundational Software,” in *Proc. International Conference on Electrical and Control Engineering*, 2011, pp. 5906-5909.

[16] S. Jarzabek and H. D. Trung, “Flexible generators for software reuse and evolution,” in *Proc. 33rd international conference on Software engineering*, 2011, pp. 920-923.

[17] S. Niu, A. Xu, and Z. Song, “A flexible software framework with dynamic expansible signals,” in *Proc. 2014 IEEE UTOTESTCON*, 2014, pp. 355-359.

[18] B. Johnson, W. W. Woolfolk, R. Miller, and C. Johnson, “Flexible Software Design: Systems Development for Changing Requirements,” CRC Press, 1th ed., 2005.

[19] I. Kim, D. Bae, and J. Hong. (2007, Nov.). A component composition model providing dynamic, flexible, and hierarchical composition of components for supporting software evolution. *Journal of Systems and Software*. [Online]. *80(11)*, pp. 1797-1816. Available:
http://www.sciencedirect.com/science/article/pii/S0164121207000659

مجله علمی-پژوهشی رایانش نرم و فن‌آوری اطلاعات

جلد ۵، شماره ۱، بهار سال ۱۳۹۵ شاپا: ۲۳۸۳-۱۰۰۶

یک راه‌کار جدید برای اندازه‌گیری کمّی انعطاف‌پذیری نرم‌افزار

عباس رسول‌زادگان

استادیار، دانشکده مهندسی، دانشگاه فردوسی مشهد، مشهد، ایران، rasoolzadegan@um.ac.ir

چکیده- انعطاف‌پذیری نرم افزار عبارت است از میزان سهولت بهبود و توسعه یک نرم‌افزار به‌قسمی که در کاربردها و محیط‌هایی غیر از آنچه که در ابتدا برای آن طراحی شده است، نیز قابل استفاده گردد. انعطاف‌پذیری نرم‌افزار یک مفهوم مطلق نیست و از جمله جنبه‌های مهم کیفیت نرم‌افزار به شمار می‌آید. ضرورت اندازه‌گیری کمّی انعطاف‌پذیری نرم‌افزار به صورت روزافزون در حال افزایش است. اخیراً یک راه‌کار جدید برای توسعه نرم‌افزار قابل اعتماد و در عین حال انعطاف‌پذیر (به‌نام $\text{SDA}_{\text{Flex\&Rel}}$) توسط نویسنده ارائه شده است. در این مقاله یک راه‌کار جدید برای اندازه‌گیری کمّی انعطاف‌پذیری نرم‌افزاهای توسعه‌یافته به کمک $\text{SDA}_{\text{Flex\&Rel}}$ ارائه می‌گردد. برای این منظور سعی می‌گردد ادعاهای غیردقیق ارائه شده برای بهبود انعطاف‌پذیری به کمک الگوهای طراحی کمّی گردد. همچنین، کارایی راهکار اندازه‌گیری پیشنهادی به صورت تجربی و عملی در قالب یک مطالعه موردی با عنوان آسانسور چند کابینه که پیش از این برای امکان‌سنجی $\text{SDA}_{\text{Flex\&Rel}}$ نیز مورد استفاده قرار گرفته است، بررسی می‌گردد. نتایج بهبود انعطاف‌پذیری وعده داده شده توسط $\text{SDA}_{\text{Flex\&Rel}}$ را تأیید می‌کند.

واژه‌های کلیدی: انعطاف‌پذیری، اندازه‌گیری کمّی، الگوهای طراحی، معیارهای نرم‌افزار.