A. Cüneyd Tantuğ and Özdemir Kavak

Abstract—State machines can be implemented by using several methods. Although state pattern (SP) implementation is one of the most popular methods, nowadays usage of hierarchical state machine (HSM) pattern is increasing. In this paper our motivation is making a metric based comparison between HSM and SP pattern. In addition, effect of increase in usage of inheritance in HSM pattern is analyzed. SP and HSM pattern belongs to different software methodologies, thus usage of object-oriented (OO) metrics is not applicable. However, both patterns implement state machines. For this reason, a new metric suit for state machines is proposed. As a result of this study, it is shown that using inheritance effectively in HSM pattern increases software quality. In addition, it is shown that HSM pattern handles enter/exit actions more effectively than SP.

Index Terms—Hierarchical state machine, software metrics, state machine, state pattern.

I. INTRODUCTION

State machines can be implemented by several methods such as doubly nested switch statements, action-state table method, and state pattern (SP). Since SP is an object-oriented (OO) pattern, it comes into prominence when software quality is a subject matter [1]. Hierarchical State Machine (HSM) pattern, introduced in 2000s, is a more recent alternative technique of implementing state machines [2].

In this paper, the effect of inheritance usage in HSM pattern is investigated. Handling enter/exit transactions are explained for both HSM and SP around a simple example. In section III, interlocking simulator software is implemented three times with the same functionality; HSM pattern is used with and without inheritance as well as SP pattern based version of the same software. In section II, in order to compare these three implementations, new metric suit is introduced.

A. Using Inheritance in HSM

In SP, each state is represented as a separate class. By sub-classing, each state inherits properties of the base class. In HSM pattern, each state is implemented as a separate method. Inheritance is handled by the HSM engine, which is the most critical part of HSM pattern.

States handling same events with same actions can be grouped under same super-state. As a result, inheritance among states can be used. In Fig. 1, both S1 and S2 handle an event e with the same transition a. Both states go to the same state; S3, with event e. S1 and S2 can be grouped under S12. By doing so, the number of events and transitions is reduced with the help of inheritance.

B. Implementing Enter/Exit Event with SP

HSM pattern guarantees execution of entry and exit actions upon entering/exiting states [2]. SP is not designed to do so. However enter and exit actions can be executed in SP too but same functionality cannot be obtained with these patterns because HSM pattern uses HSM engine to manage execution order of entry-exit events, where SP depends on polymorphism.

Let event evOn occurs while current state is Off. Execution order follows as Off::exit(), Off::evOn(), On::on() and Bright::brighten() sequentially as shown in Fig. 2. Let event evDim occurs when current state is Dark. Execution order follows as Dark::exit(), Dark::evDim() and Bright::brighten().

SP does not define in which order enter and exit transactions are executed. However, when state changes, previous state’s exit action can be executed. Then new state’s enter action can be executed. However, as shown in HSM model in Fig. 2, actions to be executed depend on the target state to be reached. For this reason, Bright’s entry action,
which is Entry(), must be implemented in two different ways which is impossible (Fig. 3). This situation can be handled in some way, however this results increase in complexity.

**II. METRIC SUIT**

Chidamber and Kemerer defined six metrics for OO software [3]. In addition, NMA and NMO metrics are introduced in [4]. All these OO metrics are good indicators for evaluating software quality. Although the software quality of SP implementation can be measured with these metrics, HSM pattern’s quality cannot be measured because of being not OO. In order to compare SP and HSM pattern, new metric suit is required. For this purpose some metrics are introduced which are originated from OO metrics. Additionally, several state related metrics are also introduced and added to metric suit because both state and HSM pattern implement states. These new metrics can give ideas about both state and HSM pattern software quality.

**A. Number of Handled Event Number (NHE)**

In OO programming, Weighted Method Call (WMC) for a class is the number of implemented methods for that class. In OO programming classes provide functionalities via methods whereas states provide it by handling events. Number of events handled by a state can help measuring quality. Class with high WMC score indicates complexity [5]. Thus, same approach is also valid for NMA metric. If WMC of a class is too high, it is better to divide this class into smaller ones [6]. Similarly a state handling too many events needs to be divided into smaller ones.

![Fig. 4. NHE metric.](image)

In Fig. 4, S1 handles three events; e1, e2, e3. NHE(S1): 3.

**B. Depth of State Inheritance Tree (DSIT)**

DSIT metric provides position of a state in the state machine as DIT metric provides the position of a class in the inheritance tree. A state located in the deeper level can use event handler provided by a predecessor of that state.

![Fig. 5. DSIT and NOCS metric.](image)

DSIT scores of states in Fig. 5 are given below;

- DSIT(S1): 0
- DSIT(S11): 1, DSIT(S12): 1
- DSIT(S121): 2, DSIT(S122): 2

**C. Number of Child State (NOCS)**

In OO programming, number of children (NOC) metric simply measures the number of immediate descendants of the class. NOC can be adapted to state metric with NOCS. NOCS measures the number of immediate descendants of a state. Class having high NOC and WMC values indicates a design problem. Similarly if a state has too many child state(NOCS) and handles too many events(NHE), it has a design problem too.

Metric values of states in Fig. 5 is given below;

- NOCS(S1): 2
- NOCS(S11): 0, NOCS(S12): 2
- NOCS(S121): 0, NOCS(S122): 0

**D. Number of Events Added (NEA)**

In OOP, functionalities are added via methods, NMA metric measures this attribute. On the other hand, states function by handling events. Corresponding NMA metric in the metric suit is NEA.

In Fig. 6, S11 is inherited from S1. It handles three (e1, e2, e4) events, NEA(S11): 3.

**E. Number of Events Overridden (NEO)**

Number of Methods Overridden (NMO) is the number of reimplemented methods in the sub-class. Similarly a state can handle event that is already handled by its super state. Number of the handled events are represented by NEO metric for a state.

In OO, adding to many methods shows that inheritance is misused. Improper usage of inheritance can be measured by NEO metric in state machines.

In Fig. 6, S11 is inherited from S1. It handles three (e1, e2, e4) events, NEO(S11): 3.
otherwise inheritance loses its significance. In OO, high score of NMO metric indicates a design problem [5]. Similarly increase in NEO score implies poor design. In Fig. 6, S1 handles three(e1, e2, e3) events. S11 handles e1 and e2 although they are handled by its super state S1. Thus, NEO value of S11 is 2.

F. Complexity of Event Handler (CEH)

CEH represents complexity while handling event. In HSM pattern implementation, this metric is obtained by calculating complexity of method implementing the state. However enter, exit and init conditions of state implementation are omitted. In SP, CEH is calculated by summing complexity of all event handlers for a state class.

G. Complexity of Enter Handler (CEnH)

Complexity in executing enter actions of the state.

H. Complexity of Exit Handler (CExH)

Complexity in executing exit actions of the state.

Corresponding metrics are not found for Coupling Between Objects (CBO), Lack of Cohesion (LCOM) and Response for A Class (RFC).

III. APPLICATION

In this study, interlocking simulator is implemented. For comparison purpose, same functionality is coded with different software designs in C++;
- Type-1 HSM pattern(single layer)
- Type-2 HSM pattern(multi-layer)
- Type-3 SP(multi-layer)

Interlocking software consists of many states and transitions. Most of the transactions are setting and resetting boolean variables. For handling events, good design is required otherwise developer can be lost in complexity.

A. Implementation

Type-1(HSM pattern): Interlocking software is implemented with Samek’s HSM engine [2]. In Type-1, all states except the top state have 1 DSIT score.

Type-2(HSM pattern): Same software is re-designed and implemented. However, this time similar states are grouped under the same super states.

Type-3(SP): In this OO design, inheritance is heavily used as HSM pattern (multi-layer) implementation.

B. Gathering Metrics

In this study, all metrics explained in Section II are collected manually for each implementation.

C. Goal

With these three implementations, answers to below questions are sought;
- Does properly used inheritance increase software quality in HSM pattern?
- What are the advantages of HSM pattern over SP? Why?

IV. RESULTS

First, effect of inheritance in HSM pattern is investigated. Latter, HSM pattern and SP are compared in terms of software quality by the help of metrics provided above.

A. Effect of Using Inheritance in HSM pattern

First, simulator software is implemented without using inheritance. Then same simulator is designed and implemented with inheritance.

NHE: If complexity of a state increases, NHE of the state increases too, thus decreases software quality like WMC does in OO programming. In Type-1 maximum score of NHE is 8, whereas Type-2 has NHE score of 6(Fig. 7). The average score of NHE in Type-1 is 3.92 whereas average score of NHE is 2.22 in Type-2. By using inheritance in HSM pattern effectively, interlocking simulator’s NHE metric score decreases 43.3%.

Fig. 7. NHE scores of all implementations.

DSIT: In Type-1, none of the states are inherited from another state, thus state with maximum DSIT has DSIT value of 1. However, in Type-2, states have maximum 4 DSIT score as shown in Fig. 8. The average DSIT score of all states increases from 0.92 to 2.22 in Type-2. This result indicates that Type-2 benefits from inheritance more than Type-1 does.

Fig. 8. DSIT scores of all implementations.

Corresponding metrics are not found for Coupling Between Objects (CBO), Lack of Cohesion (LCOM) and Response for A Class (RFC).
NOCS: Fig. 9 shows the NOCS scores of three implementations. In Type-1 and Type-2, top states have 13 and 7 child states respectively. As stated in [3], high values of NOC may point to misusage of abstraction. Similarly, in state programming, high score of NOCS indicates that inheritance is not used in Type-1.

NEA: NEA scores are given in Fig. 10. In Type-1, inheritance is not used, thus score of NEA is same as NHE. Maximum NEA score of states is 8, average score of all states is 3.92 in Type-1. In Type-2, maximum score decreases to 6. Average score also decreases to 2.22. By applying inheritance, average NEA score decreases 43.3%. In OO programming, as stated in [7], new methods added by subclass results in error-prone class. Same approach is also valid for states and NEA metric.

NEO: NEO score of both implementations is 0, because none of states overrides its parent state's handlers.

CEH: Fig. 11 shows the CEH scores of three implementations. Average complexity of states in Type-1 is 6.92. The most complex state has CEH score of 11. In Type-2 state which is the most complex has CEH score of 9. In Type-2 average CEH score is 5.22. These results indicate that inheritance decreases complexity of states 24.5% in interlocking simulator.

CEnH-CExH: Complexity of enter and exit actions are 1 in both implementations for each state. This score is optimum.

B. HSM Pattern vs. SP

In Type-3, same simulator is designed and implemented by using SP.

NHE: Maximum NHE score of a state in both Type-2 and Type-3 is same, 6. However average score is slightly greater in SP (2.35-2.22, 5.5% decrease as shown in Fig. 7).

DSIT and NOCS: Both implementations have same DSIT and NOCS score because they have same states inheriting from same states.

NEA: NEA score is same as NHE in Type-2 and Type-3 because they handle same events in each state.

NEO: In all implementations, states do not override its base state’s properties. Thus NEO score is 0 for all implementations. NEO results do not help comparing these implementations in this study.

CeH: Type-2 and Type-3 have maximum CeH scores 9 and 10 respectively. Average CEH scores of Type-2 and Type-3 are 5.22 and 6 respectively. Type-2 is less complex than the Type-3. Reason for such a result is difficulties in handling enter and exit actions in SP.

CExH: All states in Type-2 have CExH score of 1. Maximum CExH score in Type-3 is 4, average CExH score is 1.58 as shown in Fig. 12. Complexity of handling exit actions in Type-3 is 50% higher than Type-2. This indicates that SP cannot handle enter and exit actions as effective as Type-2 does.

CEnH: All states in Type-2 have 1 CEnH score. However, in Type-3, one state has CEnH score of 2 as shown in Fig. 13. Type-2 handles enter actions slightly effectively than Type-3.

V. CONCLUSION

As a result, this study proves that using inheritance effectively in HSM pattern increases software quality. In addition, HSM pattern handles enter and exit actions more effectively than SP does. Also new metric suit, which can be applied to different software methodologies, is provided.

In this paper, provided metric suit is applied to three
versions of the same state machine. As a future work, same metric suit can be applied to different state oriented software and results can be verified.

REFERENCES


A. Cüneyd Tantuğ was graduated from Control and Computer Engineering Department of ITU, 2000. He received his M.Sc. degree from ITU, Istanbul, Turkey in the year 2002. He obtained his Ph.D. in computer engineering from ITU, Istanbul, Turkey in 2007. Currently he is an assistant professor in Computer and Informatics Faculty of ITU. He is also serving as the vice dean of the Computer and Informatics Faculty and as the director of ITU Center for Information Technology. He has attended several international conferences and has published articles in various international journals. He is a member of ITU NLP Research Group. His research areas include natural language processing and software engineering. Assist. prof. Tantuğ is IEEE, and ACL member.

Özdemir Kavak was graduated from Computer Engineering Department of Istanbul Technical University, Istanbul, Turkey in the year 2010. He has a total of about 4 years of working experience since 2010. Currently, he works as a researcher at the Scientific and Technological Research Council of Turkey, Gebze, Kocaeli, Turkey. His area of interest are software metrics, software quality and design patterns.