ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

A COMPARATIVE STUDY ON HIERARCHICAL STATE MACHINE PATTERN AND STATE PATTERN

M.Sc. THESIS

Özdemir KAVAK

Department of Computer Engineering

Computer Engineering Programme

JANUARY 2015

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

A COMPARATIVE STUDY ON HIERARCHICAL STATE MACHINE PATTERN AND STATE PATTERN

M.Sc. THESIS

Özdemir KAVAK
(504111525)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Asst. Prof. Dr. Ahmet Cüneyd TANTUĞ

JANUARY 2015

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

HİYERARŞİK DURUM MAKİNESİ TASARIM KALIBI VE DURUM TASARIM KALIBI ÜZERİNE KARŞILAŞTIRMALI BİR ÇALIŞMA

YÜKSEK LİSANS TEZİ

Özdemir KAVAK
(504111525)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Yrd. Doç. Dr. Ahmet Cüneyd TANTUĞ

OCAK 2015

Özdemir KAVAK, a M.Sc. student of ITU Graduate School of Science Engineering and Technology student ID 504111525, successfully defended the thesis entitled "**A COMPARATIVE STUDY ON HIERARCHICAL STATE MACHINE PATTERN AND STATE PATTERN**", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Asst. Prof. Dr. Ahmet Cüneyd TANTUĞ** Istanbul Technical University	.....
Jury Members :	**Asst. Prof. Dr. Tolga OVATMAN** Istanbul Technical University	.....
	Asst. Prof. Dr. Mehmet AKTAŞ Yıldız Technical University	.....

Date of Submission : 10 December 2014
Date of Defense : 20 January 2015

To my family,

FOREWORD

I would like to express my deepest gratitude to Assist. Prof. Dr. Ahmet Cüneyd Tantuğ for his understanding, guidance, advice and supervision throughout the development of this thesis study. I also would like to thank to my executives at TÜBİTAK for the support and encouragement on this academic study.

December 2014

Özdemir Kavak
Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

OO	: Object oriented
GoF	: Gang of Four
HSM	: Hierarchical State Machine
FSM	: Finite State Machine
SP	: State Pattern
WMC	: Weighted Method Call
DIT	: Depth of Inheritance Tree
NOC	: Number of Children
NOM	: Number of Methods
CBO	: Coupling Between Objects Classes
RFC	: Response For a Class
LCOM	: Lack of Cohesion in Methods
NMA	: Number of Methods Added
NMO	: Number of Methods Overridden
NIM	: Number of Methods Inherited
SIX	: Specialization Index per Class
NHE	: Number of Handled Event
DSIT	: Depth of State Inheritance Tree
NOCS	: Number of Child State
NEA	: Number of Events Added
NEO	: Number of Events Overridden
CEH	: Complexity of Event Handler
CEnH	: Complexity of Enter Handler
CExH	: Complexity of Exit Handler
ISO	: International Organization for Standardization

LIST OF TABLES

Page

LIST OF FIGURES

Page

A COMPARATIVE STUDY ON HIERARCHICAL STATE MACHINE PATTERN AND STATE PATTERN

SUMMARY

State machines are an indispensable element of our lives. People interact with these state machines in order to purchase products from vending machines, to enter a metro station through a turnstile, and so on. Large numbers of problems can be modeled by the help of FSMs. Communication protocol design, electronic design automation, language processing and other engineering applications are among these problems.

FSMs are also important for implementing application behavior. FSM is a very compact way to represent a set of complex rules and conditions. FSMs define simple rules to manage complex behavior of the software.

There are many techniques for FSM implementation. If the literature is reviewed, three implementation techniques draw attention; switch statement method, table method, and OO SP. In addition, another technique of implementing a FSM is HSM pattern. HSM pattern is different from other methods because it implements HSMs.

All of these techniques have their own advantages and drawbacks. In this thesis study, we focused on SP and HSM pattern. The effects of applying these design concepts on software quality are investigated. SP is OO which is introduced by GoF, however HSM pattern is not. At first glance, OO solution may seem to be more effective than HSM pattern, however without a mathematically grounded comparison, it is open to doubt.

In order to investigate the effect of these two design patterns on software quality in a sane way, a module of interlocking software simulator, which is responsible for route allocation for railway traffic management system, is picked up. Same module is designed and implemented three times with the same functionality. Two versions of the module is implemented using HSM pattern with and without inheritance. These implementations are named as Type-1(without inheritance), Type-2(with inheritance). Third version of the software is designed and implemented with SP and named as Type-3. In SP implementation, inheritance is used as long as it makes sense.

In this study, we aim to compare SP and HSM pattern from the viewpoint of developers. Additionally, we also study the effect of inheritance in HSM pattern.

Software has many characteristics defining software quality. ISO/IEC-25010:2011 standard is reviewed in order to find quality characteristic, which is most important concern of software developers. We found that maintainability, which is a software quality characteristic in the standard, is most related to software design.

The next step of the study is to find a way for measuring maintainability of three implementations. Previous studies are reviewed and some metrics are found. These

metrics are WMC, DIT, NOCS, CBO, LCOM, RFC, which are introduced by Chidamber and Kemerer, and NMA, NMO, NIM, SIX, which are introduced by Lorenz and Kidd.

These metrics are well known for measuring the design of the software; however, they are only applicable to OO software designs. SP implementation can be measured with these CK and LK metrics; however, HSM pattern implementations of interlocking simulator cannot be measured because HSM pattern is not OO.

Despite the fact that SP and HSM pattern belong to different software paradigms, they share a common property; both of them implement state machines. Starting from this point of view, new metric suit proposed for these patterns. New metrics are originated from CK and LK metrics. Proposed metrics are NHE, DSIT, NOCS, NEA, NEO CEH, CEnH, and CExH.

NHE, CEH, CEnH, and CExH are complexity related metrics. DSIT and NOCS measure inheritance. NEA and NEO correspond to NMA and NMO in LK metrics respectively.

Three versions of the interlocking software are measured with these state-oriented metrics. Comparisons are made as pairs; Type-1/Type-2 and Type-2/Type-3.

First comparison showed that using inheritance effectively in HSM pattern increases software quality in terms of maintenance. Second comparison showed that SP version of interlocking simulator become more complex than HSM pattern version, thus increasing the effort required to maintain the software.

Comparison results of the different designs applied to the interlocking software simulator, which is summarized above, discussed, and published in the related conference and publications.

HİYERARŞİK DURUM MAKİNESİ TASARIM KALIBI VE DURUM TASARIM KALIBI ÜZERİNE KARŞILAŞTIRMALI BİR ÇALIŞMA

ÖZET

Günlük hayatta durum makineleri önemli bir yere sahiptir. İnsanlar içecek satın almak için satış makinelerini, bilet veya jeton ile metroya binmek için turnikeleri kullanırlar. Etkileşimde bulunduğumuz tüm bu makineler durum makineleri ile modellenmiştir. Bunun dışında çok sayıda problem durum makineleri kullanılarak modellenebilir. İletişim protokolü tasarımı, elektronik devre tasarımı, dil işleme ve diğer mühendislik uygulamaları bu problemlerden birkaçıdır.

Sonlu durum makineleri(SDM), yazılım davranışını gerçeklemekte de önemli bir yere sahiptir. SDM'ler karmaşık kurallar ve koşullar kümesini gerçeklemenin kolay bir yoludur. SDM, karmaşık yazılım davranışının programlanmasında basit kurallar dizisi sağlar. SDM'leri yazılımda gerçeklemenin bir çok yolu vardır. Bu konuda araştırma yapıldığında özellikle üç tekniğin çok kullanıldığı görülmektedir. Bu teknikler switch cümlesi yöntemi, tablo yöntemi ve durum tasarım kalıbı yöntemidir. Bu tekniklere ek olarak SDM gerçeklemenin bir diğer yolu da hiyerarşik durum makinesi yazılım tasarım kalıbıdır. Hiyerarşik durum makinesi tasarım kalıbı metodu diğer yöntemlerden gerçeklediği durum makinesinin hiyerarşik durumda olmasından dolayı farklıdır.

Durum makinesi gerçeklemek için kullanılan bu yöntemlerin her birinin üstün yönlerinin olduğu gibi sakıncalı yönleri de mevcuttur. Bu tezde durum tasarım kalıbı ve hiyerarşik durum makinesi tasarım kalıbı ele alınmış, bu yöntemleri uygulamanın yazılım kalitesi üzerindeki etkileri araştırılmıştır.

GoF tarafından ortaya konulan durum tasarım kalıbının temeli nesne yönelimli olmasıdır. Fakat hiyerarşik durum makinesi yazılım tasarım kalıbı nesne yönelimli değildir. İlk bakışta nesne yönelimli olan yöntemin hiyerarşik olan yöntemden daha etkili olacağı düşünülebilir fakat sağlam temeller üzerine dayanan bir karşılaştırma yapmadan kesin bir kanıya varmak tartışmaya açıktır.

Bu tasarım kalıplarının yazılım kalitesi üzerindeki etkisini araştırmak için demiryolu sinyalizasyon sistemlerinde güzergah tanziminden sorumlu olan anklaşman yazılımı benzetiminin bir modülü seçilmiştir. Bu sistemin seçilmesinin nedeni sistemin durum makinesi ile gerçeklemeye uygun olmasıdır. Bunun yanında anklaşman yazılımında durum geçişlerinde ve durumlara giriş ve çıkışta yapılması gereken işlemlerin değişkenlerin set/reset edilmesi, zamanlayıcıların başlatılıp-durdurulması gibi karmaşık olmayan işlemleri içermesi sebebiyle farklı tasarımları uygulamanın etkilerini gözlemlemek daha basit olmuştur. Uygulama olarak benzetim yazılımı aynı işlevsellikte üç kez tasarlanmış ve gerçeklenmiştir. İki sürüm hiyerarşik durum makinesi tasarım kalıbıyla, kalıtım kullanmadan ve kalıtım kullanarak tasarlanmış ve gerçeklenmiştir. Bu sürümler sırasıyla Type-1 ve Type-2 olarak isimlendirilmiştir.

Üçüncü sürümde durum yazılım tasarım kalıbı kullanılmış ve Type-3 olarak isimlendirilmiştir.

Hiyerarşik durum makinesi tasarım kalıbının kalıtımlı ve kalıtımsız uygulanmasıyla, kalıtımın yazılım kalitesi üzerindeki etkisi araştırılmıştır. Durum tasarım kalıbının kullanıldığı sürümde anlam ifade ettiği sürece kalıtım kullanılmıştır. Type-2 ve Type-3 tasarımlarında durumlar için kullanılan kalıtım ağacı birbirinin aynısıdır. Böylece hiyerarşik durum makinesi tasarım kalıbı ve durum yazılım tasarım kalıbını daha sağlıklı bir şekilde karşılaştırmak mümün olmuştur.

Bu tezde gerçeklenen farklı tasarımların karşılaştırması yazılımların kalitesi üzerinden yapılmıştır. Fakat yazılım kalitesi bir çok kişi tarafından farklı değerlendirilebilir; son kullanıcı için yazılım kalitesi kullanım kolaylığı olarak tanımlanırken, testçi için kalite, yazılımın test edilebilirliğidir. Bu çalışmada gerçeklenen tasarımların karşılaştırılması yazılım geliştiricinin bakış açısına göre yapılmış, yazılım geliştiriciyi ilgilendirmeyen özellikler göz ardı edilmiştir. Yazılım geliştiriciyi ilgilendiren yazılım kalite karakteristiklerini bulmak için ISO/IEC-25010:2011 standardı incelenmiştir. Bu belgede bulunan yazılım bakım yapılabilirliğinin, yazılım tasarımıyla en ilgili yazılım kalite karakteristiği olduğu belirlenmiştir.

Çalışmadaki bir sonraki adım, gerçeklenen üç sürümün yazılım bakım yapılabilirliğinin nasıl ölçüleceğini bulmak olmuştur. Bu amaçla önceki çalışmalar taranmış ve bazı metrikler bulunmuştur. Bu metrikler Chidamber ve Kemerer tarafından önerilen WMC, DIT, NOCS, CBO, LCOM, RFC ve Lorenz ve Kidd tarafından önerilen NMA, NMO, NIM, SIX metrikleridir.

Yazılım dünyası tarafından tasarım kalitesini ölçmek için uzun zamandan beri kullanılan bu metrikler sadece nesne yönelimli yazılımlar için geçerlidir. Durum tasarım kalıbı ile gerçeklenen sürüm bu metrikler ile ölçülebilir fakat hiyerarşik durum makinesi tasarım kalıbı kullanılarak gerçeklenen iki sürüm, nesne yönelimli olmadıkları için bu metrikler ile ölçülemez. Bu sebepten dolayı gerçeklenen tüm sürümlere uygulanabilecek bir metrik kümesine ihtiyaç duyulmuştur.

Durum tasarım kalıbı ve hiyerarşik durum makinesi tasarım kalıbı farklı paradigmalara ait olsa da ortak bir özelliği paylaşmaktadırlar. Bu özellik ikisinin de durum makinesi gerçeklemesidir. Bu yaklaşımdan yola çıkarak yeni bir metrik kümesi önerilmiştir. Bu metriklerin tasarlanmasında CK ve LK metriklerinden esinlenilmiştir. Önerilen metrikler NHE, DSIT, NOCS, NEA, NEO, CEH, CEnH ve CexH'den oluşturmaktadır. NHE, DSIT, NOCS, NEA, NEO metrikleri nesne yönelimli yazılım metriklerinde sırasıyla WMC, DIT, NOC, NMA ve NMO metriklerine karşılık gelmektedir.

NHE, CEH, CenH ve CexH metrikleri karmaşıklıla ilgili iken DSIT ve NOCS metrikleri kalıtımı ölçmektedir. NEA ve NEO, LK metriklerinde sırasıyla NMA ve NMO metriklerine karşılık düşmektedir.

Gerçeklenen üç anklaşman benzetim yazılımı nesne yönelimli yazılım metriklerinden esinlenerek oluşturulan durum yönelimli metrikler ile ölçülmüştür. Karşılaştırmalar, Type-1 ve Type-2 kendi arasında, Type-2 ve Type-3 kendi arasında olmak üzere iki şekilde yapılmıştır.

Type-1 ve Type-2 arasındaki karşılaştırma sonuçları, hiyerarşik durum makinesi tasarım kalıbında kalıtımı etkili bir şekilde kullanmanın durumların ortalama karmaşıklıklığını azalttığını göstermiştir. Böyle bir sonucun çıkmasının sebebi

kalıtımı etkili bir şekilde kullanarak sorumlulukların durumlar arasında daha dengeli bir şekilde dağıtılmış olmasıdır. İkinci karşılaştırma ise durum makinesi tasarım kalıbı ile gerçeklenen Type-3'ün Type-2'den daha karmaşık olduğunu göstermiştir. Bu sonucun ana sebebi iki tasarım kalıbının giriş ve çıkış işlemlerinin gerçeklenmesinde izledikleri yöntemlerin farklı olmasıdır. Sonuç olarak karmaşıklığın yüksek olmasından dolayı durum makinesi tasarım kalıbı ile gerçeklenen yazılımın bakım yapılabilirlik maliyeti hiyerarşik durum makinesi tasarım kalıbı ile gerçeklenen yazılıma göre daha yüksek olacaktır.

Anklaşman yazılımına uygulanan yukarıda bahsi geçen bu farklı tasarımların karşılaştırma sonuçları ilgili konferansta sunulmuş ve yayınlanmıştır.

1. INTRODUCTION

In modern life, the behavior of finite state machines can be found in many devices such as vending machines, elevators, traffic lights, combination locks, automated teller machines, or turnstiles. These machines perform sequence of predetermined actions when initiated by a triggering event or condition.

Finite state machine (FSM) or finite state automaton, or simply state machine is a model of behaviors, which consist of finite number of states, transitions between these states and transactions.

Finite state machines are important in many different fields. These fields include electrical engineering, linguistics, computer science, mathematics, logic and so on. In computer science, FSMs are used in design of digital system hardware, software engineering, speech recognition, network protocols, and implementation of application behavior.

A software application, which has event-based behavior, can be implemented easily with techniques based on state machines.

In the 1980's, another type of state machine is introduced: statecharts [1]. Statecharts introduce hierarchically nested states. It provides a way of capturing common behavior for reuse purposes.

State machines can be implemented by using several techniques. Although SP implementation is one of the most popular methods, nowadays usage of hierarchical state machine (HSM) pattern is increasing. In this thesis, 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 OO metrics is not applicable. On the other hand, they share an important common property; both patterns implement state machines. For this reason, a new metric suit for state-oriented programming is proposed. As a result of this study, it is shown that using inheritance effectively in HSM pattern increases software quality. It is also shown

that HSM pattern is more maintainable than SP pattern because HSM pattern handles enter/exit actions more effectively than SP.

1.1 The Purpose and Scope of the Study

The purpose of the study is to investigate the effect of using different techniques for implementing state machines on software quality.

Within the scope of this study, same software is designed and implemented by applying different software designs. Changes on metric values have been evaluated from the viewpoint of software developer. For this purpose, ISO/IEC:25010:2011 software quality model is investigated in order to find design related characteristics of software. In this thesis, quality term is used to refer maintainability of the software.

Furthermore, in this thesis, in order to compare these software implementations with same metrics, new state-oriented metric suit that can be used for all implementations, is proposed.

1.2 Outline

This thesis is organized as follows:

In section 2, **literature review** is performed. Previous studies, which are related to topic of this study, are examined.

In section 3, information about **software quality** is given. ISO-25010:2011 standard is explained. Quality characteristics are reviewed from the perspective of software developer. Some of the OO software metrics are given and explained.

In section 4, **proposed metrics for state oriented software design**, state-oriented metrics are proposed and explained with simple examples.

In section 5, information about **state machine implementation techniques** is given.

In section 6, **methodology**, 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 7, **results**, each of interlocking simulator implementations are measured with proposed state-oriented metrics. With the help of state-oriented metrics the effect of inheritance usage in HSM pattern analyzed and given. Furthermore, In the light of obtained metric scores, HSM pattern, and SP implementations are compared.

In last section, **conclusion** of the study is presented, and some future works are suggested.

2. LITERATURE REVIEW

2.1 Previous Studies

Previous studies can be grouped under two main categories;

- Empirical studies which measure effects of the software patterns
- Studies which investigate how to measure maintainability

In [2], factors of testability, which is a sub-characteristic of maintainability, are investigated. Software metrics related to testability are evaluated by means of two case studies of large Java projects. It is shown that size related metrics such as NOM are correlated with testability of software because a large class is required to be tested with a large corresponding test class. Results obtained from case study show that inheritance metrics, DIT and NOC, are not correlated with testability. However, if all inherited methods of a class are needed to be tested, DIT metric can be used to measure testing effort needed.

In [3], in order to find maintainability related metrics, an empirical study is done based on the maintenance history of a OO system. It is found that the size and import coupling metrics are strong predictors for measuring maintainability for the analyzed projects.

In their paper [4], Basili et al. collected data on eight medium-sized projects, which are designed and implemented to meet same requirements. They stated that CK metrics, except LCOM, are good indicator for class fault-proneness. Results showed that high WMC, DIT, RFC, CBO increases the probability of fault detection. Unexpectedly, empirical results showed that the larger the NOC, the lower the probability of fault detection. This result is explained by the fact that the most of the investigated classes have at most one child and reused classes have many children. Fault proneness is a very important asset when the maintainability is a subject matter.

In [5] and [6], some of the CK metrics, which are DIT, NOC, RFC, LCOM, WMC, and some other size metrics are suggested as a maintainability predictor metrics.

Software having lower metric scores of these metrics is considered more maintainable [7].

In [8], the effects of the application of some design pattern on software metrics are researched. Mediator, Bridge, and Visitor patterns are compared with their non-pattern alternatives. It is stated that applying bridge pattern decreases average NOC and DIT metric score. It is also shown that application of Mediator pattern decreases CBO. However, these results are not obtained from a project having large number of classes, or a framework.

In [9], an experimental work is done to investigate the software maintenance that employ some design patterns (abstract factory, composite, decorator, façade, observer, and visitor) and compare them with their simpler alternatives. In this study, it is stated that using design pattern has positive effects in most of maintenance activities. Although, maintenance time increases in a few cases, it is suggested that design patterns should be chosen incase unexpected new requirements may appear.

In [10], Bieman et al. conducted an industrial case study to investigate the correlation between code changes and design patterns. In case studies, it is found that classes playing roles in design patterns are more change-prone than other classes. However, it is also stated that pattern participant classes provide the most significant functionality to the system and they are expected to be changed more frequently than other classes. As we consider non-pattern version of the designs, most of the changes are modification whereas changes are addition of new classes in design pattern.

In their study [11], Ampatzoglou et al. propose a methodology for comparing design patterns to equivalent adhoc designs with respect to several quality attributes. Some design patterns are evaluated with respect to various quality characteristics. In three cases, design patterns provide a more maintainable design. However, there are also cases implying that design pattern is not the best solution.

In [12], relation between OO design patterns, OO metrics, and software error-proneness is studied. For this purpose, various open source software projects and experimental source codes have been analyzed in order to find defected parts of software designs by the help of OO metrics. In the light of obtained metric results, defected parts of the software were redesigned and implemented with appropriate design patterns. In this study, all of the projects analyzed are design pattern free and

object-oriented. Power-Grab project is redesigned using façade, state, and singleton design patterns. As a result, it is stated that state pattern significantly reduced WMC and CBO values. However, designs used in original version of Power-Grab project are not mentioned.

In [13], the connection between design patterns and OO quality metrics are investigated similar to [12]. However, in this thesis, effect of design pattern usage on software maintainability is sought. First, a project called Routing Software is analyzed and suitable design patterns are applied. Routing Software is redesigned and implemented in four phases. Output of each phase is used as an input of next phase, so design patterns are applied cumulatively. In some phases, more than one design pattern is applied, and then results are evaluated. Applying design pattern at the same time or applying one after another may effect results. In this thesis, state pattern is not investigated.

In [14], the effect of some OO and real time software design patterns on another aspect of software, which is performance, is investigated. For this purpose, non-OO projects have been redesigned and implemented by using OO language with and without OO and real time software design patterns. Responsibilities of software programs developed are same as the original version. It is shown that by applying state pattern carefully, some improvement in overall performance of real time software can be obtained.

3. SOFTWARE QUALITY

Assuring the quality of software is the most challenging activity throughout the software life cycle. In [15], software quality is defined as:

"degree to which the software product satisfies stated and implied needs when used under specified condition".

Through the years, some software quality models are published to define software characteristics that effect software quality.

Quality models help software team to

- define software requirements
- confirm the scope of requirement description
- define software design goals
- define software testing goals
- define quality control condition
- define acceptance condition for a finalized software product [15]

Quality of the software might change according to people because stated and implied expectations from software may be different. For an end user, quality can be described as ease of use whereas quality can be code readability for a developer.

3.1 ISO/IEC 25010:2011

ISO/IEC 25010:2011 standard is one of the established quality model. This international standard defines "a software product quality" and "a quality in use model".

The characteristics defined by quality in use model and product quality model are applicable to all computer systems and software products. The characteristics defined

in this standard provide technical language for defining, measuring, and evaluating system and software product quality.

A quality in use model has five characteristics. These characteristics are valid when a product is used in a particular context of use. A product quality model has eight characteristics, which are about dynamic attributes of the computer system and static attributes of the software.

3.1.1 Quality in use model

Quality in use is a measure of the quality of the software, hardware, and environment. Characteristics of the users, goals, and social environment also effect quality in use. Quality in use model has three characteristics;

- Usability in use
    - Effectiveness in use
    - Efficiency in use
    - Satisfaction in use
    - Usability in use compliance
- Flexibility in use
    - Context conformity in use
    - Context extendibility in use
    - Accessibility in use
    - Flexibility in use compliance
- Safety
    - Operator health and safety
    - Public health and safety
    - Environmental harm in use
    - Commercial damage in use
    - Safety compliance

Quality in use model is related to complete human-computer system. Characteristics of this model mostly related to human factor. In this thesis, software is evaluated from the viewpoint of developer or tester. Thus, quality in use model is not an appropriate model.

3.1.2 Software product quality model

Software product quality model has eight characteristics, which are divided in to sub-characteristics i.e. Figure 3.1.

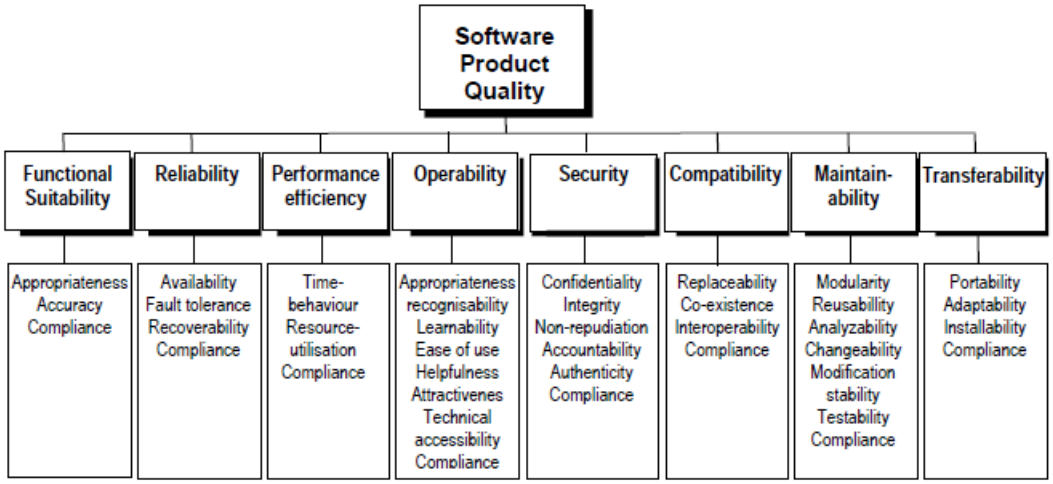

Figure 3.1 : ISO/IEC 25010 software product quality model.

Software product quality model is suitable for determining quality characteristics that concern software developer's quality expectations. Thus, in this thesis, software product quality model is chosen as a quality model.

3.1.2.1 Functional suitability

Functionality is the software's ability to meet stated requirements. In this thesis, SP, and HSM pattern versions of interlocking simulator has same functionality.

3.1.2.2 Reliability

Reliability characteristic is related to performance of software product. Performance is not a subject matter in this thesis.

3.1.2.3 Performance efficiency

Performance efficiency is associated with time behavior and resource utilization of software. Performance related issues are not within the scope of this thesis.

3.1.2.4 Operability

When operability is a subject matter, software is expected to be understood, learned, and used easily by the user. The software should provide help incase users need assistance. It also has to be attractive to the user. Moreover, users with specified disabilities can use software product easily. This type of quality characteristic is not within the scope of this thesis.

3.1.2.5 Security

A system must protect its elements from accidental or malicious access, usage, alteration, destruction, or disclosure. Security related sub-characteristic of quality model is not within the scope of this thesis.

3.1.2.6 Compatibility

Compatibility is the capability of more than one software modules sharing the same hardware or software environment to transfer data or carry out their tasks. Compatibility is not within the scope of this thesis.

3.1.2.7 Transferability

Transferability is the ability to transfer software product from one environment to another. All implementations run on the same environment, thus this type of quality characteristic is not within the scope of this thesis.

3.1.2.8 Maintainability

Maintainability is the degree to which the software product can be modified. Bug fix and improvements are accepted as modification. Changes in environment and requirements can be reasons for software modification. Modularity, reusability, analyzability, changeability, modification stability, testability, and compliance are sub-characteristics of maintainability.

Modularity emphasizes that alteration of one component of the software must have minimal effect on other components. A component is desired to be independent of other components. Reusability is the capability of using a software asset in another software system.

Analyzability sub-characteristic states that software product can be examined for defects or causes of failures in the software.

Changeability is the degree to which software product enables modification including changes in designing, coding, and documentation. Software product expected to be stable after modification. Modification of software should not cause unexpected behavior according to modification stability sub-characteristic.

Testability is the validation of the modification applied to the software. Maintainability compliance is the conformity of software to the standards or rules about maintainability.

After we reviewed software quality characteristics, we found that maintainability is one of the most design related software characteristics. In this thesis, software designs are evaluated from the viewpoint of maintainability.

Before measuring the software in terms of the characteristics explained above, these characteristics need to be supported by a measurable basis. Software quality metrics are used for this purpose.

From earlier studies [2, 4, 5, 6, 12], OO software metrics are determined to compare different designs implemented in this thesis. These metrics are WMC, DIT, NMA, NMO, and NOCS.

3.2 Metrics

In order to describe entities in real world, numbers or symbols are assigned to them as attributes [16].

In [17], Fenton gives definition of measurement:

"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules."

A metric is a property to measure any attribute of any system. Software metrics describe various activities related to measurement in software engineering. These activities vary from obtaining numbers characterizing properties of software code to models, which help engineers make prediction about the resource needs and software quality [18].

Software metrics help engineers understand design and structure of software without reading source code.

Many software metrics, which measures size, complexity, and coupling attributes of software, are introduced through the years. Object-oriented metrics are the most applied among these metrics.

3.2.1 Object Oriented Metrics

3.2.1.1 Chidamber and Kemerer metrics

Need for process improvement in software development forced managers to use new and improved approaches like object-orientation. Chidamber and Kemerer defined six new OO metrics in [19]. These metrics have strong theoretical and mathematical background, where as previously proposed metrics are criticized for being not theoretically grounded. The metrics are evaluated against previously introduced measurement principles.

Weighted methods per class (WMC)

Definition: Let $\boldsymbol{C_1}$ be a class. Assume $\boldsymbol{M_1, \dots, M_n}$ are methods of $\boldsymbol{C_1}$ with complexity $\boldsymbol{c_1, \dots, c_n}$ respectively. Then;

$$WMC = \sum_{i=1}^{n} ci \qquad \textbf{(4.1)}$$

Complexity is not defined elaborately by CK. Decision of how to calculate the complexity of a method is left to analyzer. Usually McCabe's cyclomatic complexity, which is the maximum number of linearly independent execution paths, is chosen to calculate WMC [20].

If a class has n methods with complexity score of one, then WMC is equal to the number of methods, which is n.

Viewpoints:

- The complexity of methods and the number of methods of a class give idea about how much time and effort needed to develop and maintain the class.
- A class having large number of methods has more impact on its children classes, because children inherit all methods in the class.

- Classes having large numbers of methods are possibly more application specific then class with few methods.

Depth of inheritance tree (DIT)

Definition: The depth of inheritance tree is the depth of inheritance of the class. If multiple inheritance is involved, DIT is the maximum distance from the node to the root of the tree.

Viewpoints:

- A class that is deeper in the hierarchy is likely to inherit large number of methods. Inheriting greater numbers of methods results in more complex class and its behavior becomes unpredictable.
- The deeper the class in the hierarchy tree, the greater the design complexity, because more methods are inherited.
- Classes located in the deep level of inheritance tree are likely to reuse inherited methods.

Lorenz and Kidd suggested a threshold of six levels for individual classes. In [21], it is stated that classes deep in the inheritance tree are more error prone.

Number of children (NOC)

Definition: NOC is the number of immediate subclasses of a class in the inheritance tree.

Viewpoints:

- Having large number of children for a class indicates high degree of reuse for that class.
- A class with great number of children may indicate improper abstraction of that class.
- The number of children may indicate the impact a class has on the software design. A Class with large number of children may need more testing of the methods in that class.

Coupling between object classes (CBO)

Definition: Coupling between object classes represents the number of classes coupled to that class. If a class uses methods or attributes of other class, that class is said to be coupled to other class. Being inherited from another class is also accepted as coupling.

Viewpoints:

- Strongly coupled object classes are undesirable because they are obstacle to modular design and they prevent reuse. Reusing a class in another software is easier if the class is independent.
- Inter-object class couples needs to be minimized in order to get modular design. High number of coupled classes makes software sensitive to changes. Maintaining such software is difficult.
- The metric is useful for estimating the complexity of testing different parts of software. Designs with high inter-object class coupling require tough testing to detect bugs.

Response for a class (RFC)

Definition: Let $\{\boldsymbol{M}\}$ be set of all methods in the class and $\{\boldsymbol{R_i}\}$ be set of methods called by method $\boldsymbol{i}$,

$$RS = \{M\} \bigcup_{all\ i} R_i \tag{3.2}$$

$$RFC = |RS| \tag{3.3}$$

Viewpoints:

- If large number of methods is executed in response to a message, tester needs deep understanding in order to test and debug the class because it becomes more sophisticated.
- Execution of large number of methods in response to message calls results in class that is more complex.
- Considering maximum number for possible responses helps allocating enough time for testing.

Lack of cohesion of methods (LCOM)

Definition: Let $\boldsymbol{C_1}$ be a class with $\boldsymbol{n}$ methods ($\boldsymbol{M_1}, \boldsymbol{M_2}, \ldots, \boldsymbol{M_n}$). Let $\{\boldsymbol{I_j}\}$ be a set of instance variables used by $\boldsymbol{M_i}$.

There are n such sets $\{\boldsymbol{I_1}\}, \ldots, \{\boldsymbol{I_n}\}$. Let $\boldsymbol{P} = \{(\boldsymbol{I_i}, \boldsymbol{I_j}) | \boldsymbol{I_i} \cap \boldsymbol{I_j} = \emptyset\}$ and $\boldsymbol{Q} = \{(\boldsymbol{I_i}, \boldsymbol{I_j}) | \boldsymbol{I_i} \cap \boldsymbol{I_j} \neq \emptyset\}$. If all $\boldsymbol{n}$ sets $\{\boldsymbol{I_1}\}, \ldots, \{\boldsymbol{I_n}\}$ are $\emptyset$ then let $\boldsymbol{P} = \emptyset$. Then, LCOM of a class is given in equations **(3.4)** and **(3.5)**.

$$LCOM = |P| - |Q|, if\ |P| > |Q| \quad \textbf{(3.4)}$$

$$= 0\ otherwise \quad \textbf{(3.5)}$$

LCOM measures the correlation between the methods and the local instance variables of a class. A high score of LCOM points out lack of cohesion.

Viewpoints:

- Cohesiveness of methods is preferable, because it encourages encapsulation.
- Lack of cohesion suggests that classes should be divided in to smaller classes.
- Finding dissimilarity of methods assists in discovery of flaws in the design of classes.

Low cohesion may increase number of errors during the development process because it increases complexity.

3.2.1.2 Lorenz and Kidd metrics

Lorenz and Kidd have introduced some OO metrics [7].

Number of methods added (NMA)

Definition: Number of methods defined in a sub-class. Overridden and inherited methods are not included in NMA.

Viewpoints:

- The less added new methods to inherited class, the more inheritance is justified.

It is good to add less new methods to classes, which are located deep in the inheritance tree. Extending a class by adding new methods results in more error-prone class [21].

Number of methods overridden (NMO)

Definition: NMO is the number of methods overridden by a subclass.

Viewpoints:

- Classes having large number of overridden methods point out design problem.

The more overriding methods the class have, the greater the confusion.

Number of methods inherited (NMI)

Definition: NMI is the total number of methods inherited by a subclass.

Specialization index per class (SIX)

Definition: SIX is calculated by using DIT, NMO, NMA, and NMI. It is given in equation **(3.6)**.

$$SIX = \frac{NMO \times DIT}{NMO + NMA + NMI} \quad \textbf{(3.6)}$$

It measures the extent to which sub-classes replace their superclass's behavior.

Viewpoints:

- Base classes SIX metric score is zero.

Redefining a method as early as possible decreases time spend to development and maintain the class, because inheritance in deep levels decreases understandability [30].

4. PROPOSED METRICS FOR STATE ORIENTED SOFTWARE DESIGN

CK defined six metrics for OO software [19]. In addition, NMA and NMO metrics are introduced in [7]. 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 introduced and added to metric suit because both SP and HSM pattern implement states. These new metrics can give ideas about software quality of both SP and HSM pattern.

4.1 Number of Handled Event (NHE)

If complexity of all methods is optimal (namely one), then WMC for a class is the number of implemented methods for that class. In OO programming, classes provide functionalities via methods. In state oriented programming, functionalities are provided by states which handling events. Number of events handled by a state can help engineer measure quality. Class with high WMC score indicates complexity [13]. 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 [12]. Similarly, a state handling too many events needs to be divided into smaller ones.

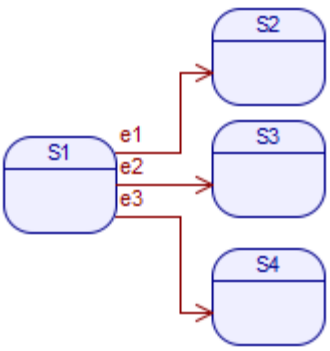

Figure 4.1 : NHE.

In Figure 4.1, S1 handles three events; e1, e2, e3.

NHE(S1): 3

4.2 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.

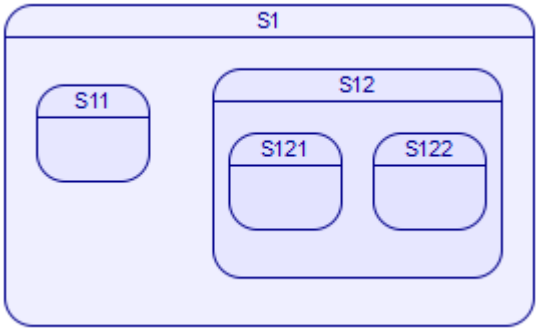

Figure 4.2 : DSIT.

DSIT scores of states in Figure 4.2 are given below;

DSIT(S1): 0

DSIT(S11):1

DSIT(S12):1

DSIT(S121):2

DSIT(S122):2

4.3 Number of Child State (NOCS)

In OO programming, NOC metric simply measures the number of immediate descendants of the class. NOC can be adapted to state oriented 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 scores of states in Figure 4.2 are given below;

NOCS(S1): 2

NOCS(S11):0

NOCS(S12):2

NOCS(S121):0

NOCS(S122):0

4.4 Number of Events Added (NEA)

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

In OO, adding too many methods shows that inheritance is misused. Similarly, improper usage of inheritance can be measured by NEA metric in state machines.

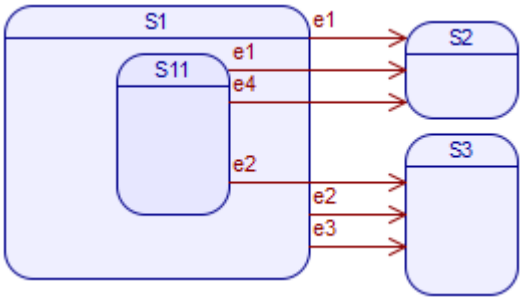

Figure 4.3 : NEA.

In Figure 4.3, S11 is inherited from S1. It handles three events; e1, e2, and e4, however only e4 is new event added to the inheritance. Thus, NEA score of S11 is one.

4.5 Number of Events Overridden (NEO)

NMO is the number of implemented methods in a sub-class. Similarly, a state can handle event that is already handled by its super state. For a state, number of events that are overridden in sub-state is represented by NEO metric.

Inherited attributes should be used without modification. Otherwise, inheritance loses its significance. In OO, high score of NMO metric indicates a design problem [13]. Similarly, increase in NEO score implies poor design.

In Figure 4.3, 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 two.

4.6 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. Enter, exit and init conditions of state implementation are also counted. In SP, CEH is calculated by summing complexity of all event handlers for a state class.

4.7 Complexity of Enter Handler (CEnH)

For a state, CEnH is the complexity of executing enter actions.

4.8 Complexity of Exit Handler (CExH)

For a state, CExH is the complexity of executing exit actions.

4.9 Validation of Proposed Metrics

Weyuker has developed some essential properties that a metric has to satisfy [22]. These properties are listed below;

- Monotonicity
- Granularity
- Interaction Increases Complexity
- Noncoarseness
- Nonuniqueness
- Design Details are Important
- Permutation
- Renaming property

- Nonequivalence of Interaction

Chidamber and Kemerer validated their OO metrics according to this list except the permutation property, which is stated to be inappropriate for OO design metrics by Cherniavsky and Smith in [23].

In this thesis, proposed metrics are state oriented. However, they can be validated like OO metrics.

Let A and B be two distinct states, then $\forall A, \forall B\ \mu(A) \leq \mu(A;B)\ \&\ \mu(B) \leq \mu(A;B)$.

This property states that metric for combined state is equal or greater than either of the component states; A and B.

According to granularity property of Weyuker, there must be a finite number of class sharing same metric value. This property is met by our state metrics because there are a finite number of states with same metric score.

$\exists A, \exists B\ \mu(A) + \mu(B) < \mu(A;B)$, where A and B are the two different states. This property states when two states are merged, metric value of combined state may be greater than the sum of two individual state metric value. Interaction between states may increase complexity.

$\exists A, \exists B\ \mu(A) \neq \mu(B)$, where A and B are the two different states. This property states that for a given state A, there is at least one state, B, such that their metric value is not equal.

$\exists A, \exists B\ \mu(A) = \mu(B)$, where A and B are the two different states. This property states that there exist some states that their metric values are same. This property is valid for our proposed metrics.

$\exists A, \exists B\ A \equiv B\ and\ \mu(A) \neq \mu(B)$, where A and B are the two different states. Their metric scores may be different. Design and implementation details of A and B influence the metric score.

In SP, states are represented as classes whereas they are represented as methods in HSM pattern. Changing states names, which means changing class name in SP and changing method name in HSM pattern does not alter metric value. Thus, Weyuker's renaming property is satisfied.

Let A and B be states with same metric score. Then,

$$\exists A, \exists B, \exists C \; \mu(A) = \mu(B) \; and \; \mu(A; C) \neq \mu(B; C)$$

$$\exists A, \exists B, \exists C \; \mu(A) = \mu(B) \; and \; \mu(C; A) \neq \mu(C; B).$$

This property states that if another state C is merged with these states separately; new combined states metric score may be different because interaction of C with A and B may be different.

5. STATE MACHINE IMPLEMENTATION TECHNIQUES

5.1 Doubly Nested Switch Method

Doubly nested switch method is the most popular technique among other approaches. Algorithm consists of two levels of switch statements as illustrated in Figure 5.1. First level is controlled by a scalar state variable and second level is controlled by an event signal variable [24]. Coder can nest the switches first by event and then by state.

```
void dispatch(unsigned const sig) {
switch(myState) {
     case STATE_1:
          switch(sig) {
               case SIGNAL_1:
                    tran(STATE_X)
                    ...
                    break;
               case SIGNAL_2:
                    tran(STATE_Y)
                    ...
                    break;
          }
          break;
     case STATE_2:
          switch(sig) {
               case SIGNAL_1:
                    ...
                    break;
                    ...
          }
          break;
     ...
}
```

Figure 5.1 : Doubly nested switch method.

Signals and states are typically represented as enumerations as shown in Figure 5.2.

```
enum Signal {
SIGNAL 1, SIGNAL 2, SIGNAL 3, ...
};
enum State {
STATE_X, STATE_Y, STATE_Z, ...
};
```

Figure 5.2 : Signals and states.

Main advantage of doubly nested switch method is being simple to implement. It requires enumerating states and triggers to represent states and events. It also has a small memory usage because it requires one scalar variable to store current state.

However doubly nested switch method does not promote code reuse. Event dispatching time is not constant. It depends on the number of cases and location of the event and state in the switch block. Nested switch implementation does not support hierarchical structure. This makes manual added entry/exit actions more error prone. One state's entry action is distributed and repeated in many places, thus it results in maintenance problems.

5.2 State Table Method

State table contains arrays of transitions for each state [24]. The content of the cells are transitions, which are represented as pairs (action, next state) as shown in Table 5.1.

Table 5.1 : State table.

States→ \ Events→	EVENT_1	EVENT_2	EVENT_3	EVENT_4
STATE_1				
STATE_2				
STATE_3	action1() STATE_1			
STATE_4				

In state table approach, dispatching process consists of three steps:

- Identification of transition to be taken as a state table lookup
- Execution of action
- Changing current state

State table consists of two parts: a generic and reusable processor part and an application specific part. Initializing transition table, defining action functions, enumerating states and signals depend on the application.

State table implementation maps directly to the regular state table representation of a FSM. It requires enumeration of events and states. This method provides relatively good performance for event dispatching. It takes constant time, $O(1)$. Event processor code can be re-used without any change.

Beside these advantages, it also has disadvantages. First, it requires a large state table, which is sparse and wasteful. In order to represent functions, it requires a large number of fine grain functions. State table initialization is also complicated. Finally, it is not hierarchical. To deal with state nesting, entry/exit actions, and transition guards, hardcoding is required in the transition action functions. This hardcoding makes table state approach error-prone and hard to maintain.

5.3 State Pattern [25]

State pattern is a behavioral object-oriented pattern which is introduced in [25] by GoF. Problem definition of state pattern consists of an object whose behavior is dependent on its state. Solution to this problem follows as

- For each state, create class implementing a common interface
- Delegate operations, which are state-dependent, from the context object to its current state object
- Make sure the context object always points to a state object which reflects its current state

In SP, states are represented as sub-classes of an abstract state class. Abstract state class defines a common interface for handling events. UML class diagram is shown in Figure 5.3.

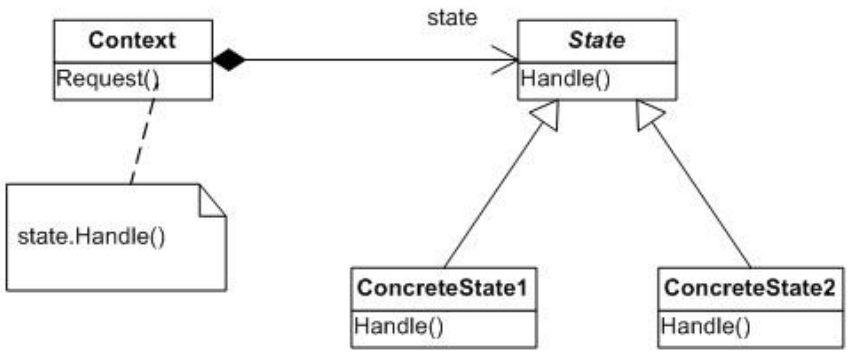

Figure 5.3 : UML class diagram of SP.

SP depends on polymorphic attributes of OO programming [26]. Each event corresponds to a virtual method. Context class handles the processing and delegates all events for processing to the current state object. State transitions are accomplished by reassigning pointer to the current state object. In order to add new events, new methods to the abstract state class and concrete class which handles them must be added. Adding new states is done by sub-classing the abstract state class.

State sub-classes override only event-handler methods corresponding to events that are handled in these states. Concrete class declares all concrete states as static members. Context class also grants friendship to all concrete sub-classes so sub-classes can access context class's attributes. Accessing context attributes from sub-state methods is not direct and violets encapsulation.

For every signal event, context class duplicates the interface of the abstract state class by declaring a method. However, implementation of these methods is fixed as prescribed by the SP. Context class invokes appropriate methods of the state class polymorphically. After being invoked by the context class, concrete state sub-classes implement the specific actions inside their event methods.

Some of the advantages of SP are summarized below. It localizes the state specific behavior in separate sub-state classes. State transitions are very efficient because it is handled by reassigning current state pointer to the new state. By using late binding mechanism, state pattern provides good performance for event dispatching. If action execution is omitted, event dispatching takes constant time. This performance is mostly better than indexing into state table and invoking a method via function pointer. However, this performance is only valid when the selection of the appropriate event handler is not taken into consideration. In practice, in order to perform such selections switch statement is used.

Signature of each event handler can be customized, and event parameters can be made explicit. SP implementation does not require enumerating states and events as nested switch and state table implementations do. It is memory efficient.

SP localizes state-specific behavior into state classes. It also divides behavior for different states into separate state classes. By implementing new sub-classes, new states and transitions can be added without modifying other state classes. SP distributes behavior for different states across several state sub-classes, which

increases the number of classes. Having many states is not compact but it is far better than a class, which has large conditional statements.

In state programming, large conditional statements are not desirable because they result in less explicit code. Modification and extension of such code is very difficult and error prone. On the other hand, SP encapsulates each state transition and action in a sub-class.

HSM pattern guarantees execution of entry and exit actions upon entering/exiting states [27]. 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 this pattern because HSM pattern uses HSM engine to manage execution order of entry-exit events, where SP depends on polymorphism. SP design is not hierarchical. Thus, manual handling of entry and exit events is error-prone and complex. This disadvantage is explained below with simple example: light state machine.

Let event *evOn* occurs while current state is *Off*. Execution order follows as

- *Off::exit()*
- *Off::evOn()*
- *On::on()*
- *Bright::brighten()*

sequentially as shown in Figure 5.4. Let event *evDim* occurs when current state is *Dark*. Execution order follows as

- *Dark::exit()*
- *Dark::evDim()*
- *Bright::brighten()*

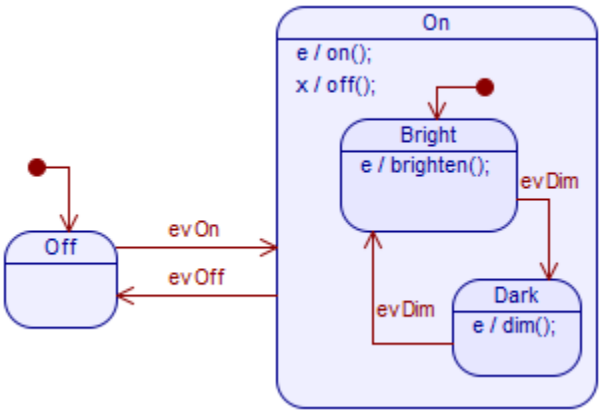

Figure 5.4 : Light statechart.

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 Figure 5.4, 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 (Figure 5.5). This situation can be handled in some way however, this increases complexity.

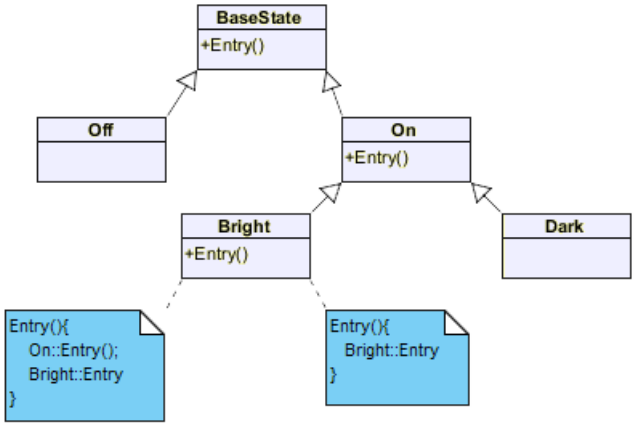

Figure 5.5 : UML class diagram of light state machine.

5.4 Hierarchical State Machine Pattern

In order to understand HSM pattern clearly, hierarchical state machine must be understood first.

HSM implements features of UML state charts [27]:

- Nested states and handling of group transitions and reactions

- Guaranteed execution of entry and exit actions
- Guaranteed execution of initialization for each state
- Simple implementation of conditional event responses(guards)
- Inheriting and specializing state models

Transition Topology Types on Nested States

There are seven types of transition in HSM. They are shown in Figure 5.6.

- a- self-transition [*source* = *target*]: It does not probe any of super-states. It can be checked directly because source and target states are same.
- b- [*source* = *target->super*]: It probes the super-state of the target state. It involves entry to source, however it does not involve exit from target.
- c- [*source->super* = *target->super*]: It is the most common transition topology. Additional probing is required for the super-state of the source state. First source state is exited and later target state is entered.
- d- [*source->super* = *target*]: This transition does not require additional probing. It involves exit from source target. However it does not involve enter to the target state.
- e- [*source* = any of *target->super*…]: Probing the super-state of the target is required until a match is found or until the top state is hit. The array *entry[]* stores the *target* state hierarchy and it is used to retrace the entry in the reverse order. This type of transition does not require exiting the source state.
- f- [source->super = any of target->super…]: Traversal of the *target* state hierarchy which is stored in the array *entry[]* is required in order to find LCA. After finding LCA, the subsequent entry proceeds from lca-1.

g- [any of *source->super*… = any of *target*…]: For every super-state of the source, traversal of the *target* state hierarchy which is stored in the array *entry[]* is required.

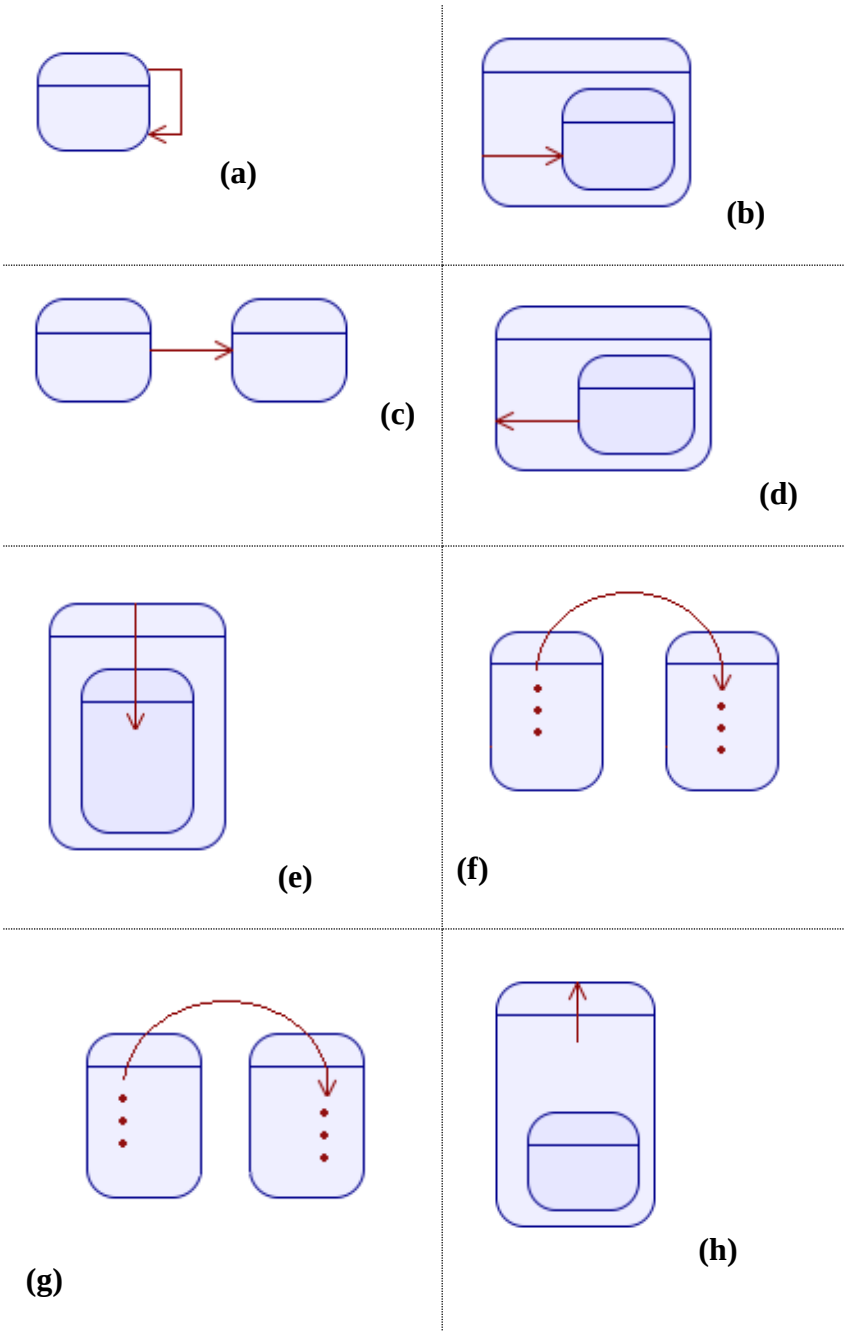

Figure 5.6 : HSM transition topologies.

To demonstrate how HSM works, some transitions are explained in detail on calculator HSM. Calculator HSM is shown in Figure 5.7.

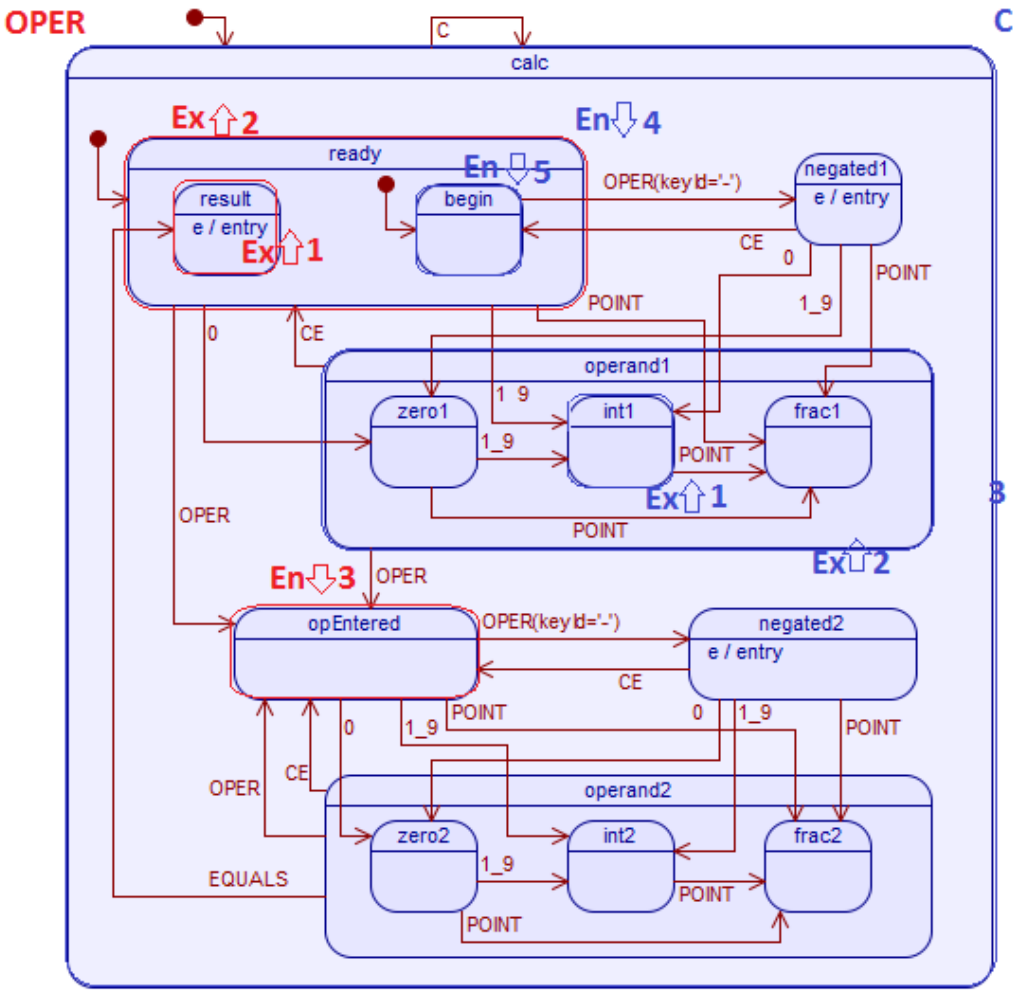

Figure 5.7 : Statechart of a calculator.

Calculator HSM has 15 state, 11 of them has no sub-state. It has 8 events which are *C*, *CE*, numbers (*1*,..,*9*), *0*, *POINT*, *OPER*, *EQUALS*, *IDCANCEL*.

Two scenarios, which are highlighted in Figure 5.7, are explained below.

- While in *result* state, *OPER* event occurs: First, result state tries to handle *OPER* event, however it cannot handle it, so it forwards it to its super state *ready*. LCA of the source and target states are *calc* state. Before entering the target state, exit routine is executed two times according to LCA. This results in exiting *result* and *ready*. After exit operations, source state *opEntered* is entered. This transition is shown in Figure 5.7 via red drawings.

- While in *int1* state, *C* event occurs: When *C* event occurs HSM goes to the starting state, which is *begin* state. *int1* state cannot handle *C* event itself, so it forwards to its super-state, however super-state *operand1* cannot handle *C*

event too. Forwarding this event goes on until the event is handled by a state. In this HSM, only *calc* state can handle *C* event. Exit is executed from the source state until the *calc* state, so *int1*, *operand1* are exited respectively. Handling of event *C* by *calc* state is self-transition. In this situation, target state is *calc*. However, HSM's current state cannot point to a state, which has child states. First HSM goes into *ready* state and *begin* state respectively. This ensures execution of entry routines of *ready* and *begin* states respectively. This transition is shown in Figure 5.7 via blue drawings.

HSM pattern implementation uses some attributes of previously introduced approaches.

HSM pattern is not an OO design pattern like SP. In HSM pattern, states are represented as instances of *State* class. In SP, *State* class is intended for sub-classing; however, in HSM pattern purpose of *State* class is for inclusion as is. In Samek's approach [27], state machine is constructed by composition, not inheritance. State specific behaviors are handled in the event handler method of *State* class. UML class diagram of HSM pattern is shown in Figure 5.8.

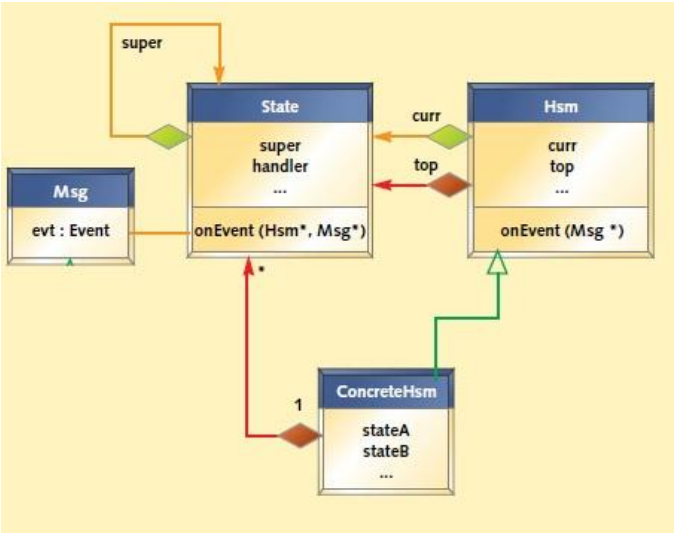

Figure 5.8 : UML class diagram of HSM pattern [27].

State machine contains at least one state, which is top state. Concrete HSMs are created by inheriting from *HSM* class, adding arbitrary numbers of new states, and defining event handlers. Unlike the SP, event handler methods of *HSM* or its sub-classes have direct access to concrete class's attributes. In event handlers, one level of switch statement is required for event handling. Each event handler communicates

with the HSM engine with a return value. If event handler can process that event, it returns a null pointer. Otherwise, it returns the message (event) itself for further processing by higher-level states.

All event handler methods implement entry/exit actions, and other default transitions. Pre-defined events include *ENTRY_EVT*, *EXIT_EVT* and *START_EVT*. These events are created and dispatched to the event handlers by the state machine engine upon each transition.

Hierarchical structure of state machine is decided upon construction. In the constructor of concrete *HSM* class, topology of state machine is constructed by initializing the participating states. Initializing includes setting the super state pointers, which define the hierarchical structure of state machine and event handlers for each state.

Exit actions precedes any actions associated with the transition and these actions precedes any entry actions associated with newly entered state. In order to find which exit actions to execute, LCA of the source and sink state is found. Exit actions are executed until the least LCA state is reached. Enter actions are executed in order from LCA to the target state. Calculating LCA of two states is expensive, however it is not necessary to find it repeatedly because same source and sink always have same LCA result. It can be calculated once and stored for future use.

6. METHODOLOGY

6.1 Goal

In this work, the primary effort is finding answers for the following two questions:

- What are the advantages of HSM pattern over SP from the quality perspective?
- Does properly used inheritance increase software quality in HSM pattern?

6.2 Case Study: Interlocking Simulator

In this thesis, one module of interlocking simulator is implemented. For comparison purpose, same functionality is coded by applying different software designs in C++;

- Type-1 HSM pattern (single layer)
- Type-2 HSM pattern (multi-layer)
- Type-3 SP (multi-layer)

Interlocking is a system of signal equipment that prevents conflicting actions. An interlocking's responsibility is prevent signals from giving proceed sign unless the route to be used is proven safe. Safety depends on many field side elements such as level crossings, switches, tracks etc., and thus increases the number of states in state machine model.

When allocating a route, interlocking software's state moves from one state to another depending the field side equipment's condition. When changing state, some operations are executed such as changing direction of switch, closing-opening crossover arm and so on. Most of the transactions are setting and resetting boolean variables, starting, pausing and stopping timers etc. There is no conditional statement in transactions. For handling events, good design is required otherwise developer can be lost in complexity.

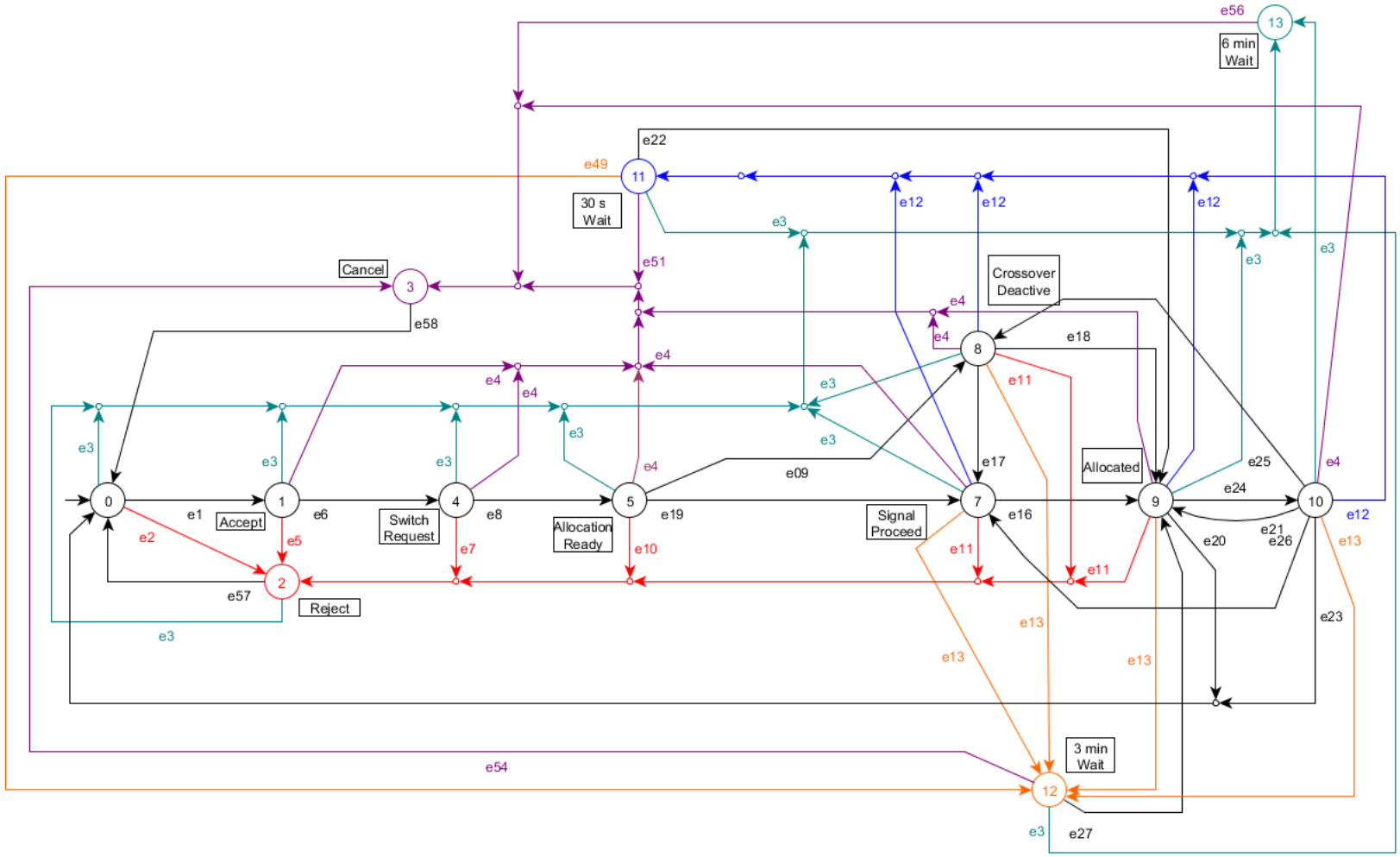

Figure 6.1 : State diagram of interlocking simulator.

Interlocking simulator module's FSM diagram is given in Figure 6.1. The FSM has 13 states and over 50 transitions. This FSM also contains enter and exit actions to be executed upon entering and exiting states.

6.3 Implementation

Interlocking module is redesigned and implemented according to the methods listed above.

6.3.1 Type-1 HSM pattern (single layer)

Interlocking software module is implemented by using Samek's HSM engine [27]. In this design, all states are inherited from top state. None of the states is inherited from other states. This version of interlocking simulator is intended for investigating and measuring effects of absence of inheritance on software quality in HSM pattern.

Type-1 has 13 states. All of these states are leaf because none of states is inherited except top state.

6.3.2 Type-2 HSM pattern (multi-layer)

Same software is re-designed and implemented. However, this time similar states are grouped under the same super states in order to take advantage of inheritance property of HSM.

Type-2 is implemented to compare it with the Type-1 for understanding effects of inheritance usage in HSM.

This design has 18 states, 39 transitions. 13 of the states are leaf. Each state is implemented as a separate method. States handle enter, exit and other events in their handler method.

In Type-2, number of states is increased however; number of handled events per state is decreased.

State-2 is implemented via *Stt_2_Hndlr* function as shown in Figure 6.2.

```
Msg const * Tanzim_HSM::Stt_2_Hndlr( Msg const *msg )
{
      switch (msg->evt){
            case ENTRY_EVT:
                  Field::instance()->bitValues[
                  m_route->getAddress("TalepReddedildiAs")] =
true;
                  Field::instance()->bitValues[
                  m_route->getAddress("TalepKabulAs")] = false;
                  m_route->timer_t4.start();
                  return 0;
            case EXIT_EVT:
                  Field::instance()->bitValues[
                  m_route->getAddress("TalepReddedildiAs")] =
false;
                  m_route->timer_t4.stop();
                  return 0;
            case START_EVT:
                  return 0;
            case evt_e57:
                  STATE_TRAN(&Stt_0);
                  return 0;
      }
      return msg;
}
```

Figure 6.2 : Implementation of state-2.

Upon entering and exiting state-2, enter and exit actions are executed respectively. HSM engine calls state functions with event parameters representing the event. As seen in the Figure 6.2, state-2 handles enter, exit and *evt_e57*. Handling these different type of events is implemented with single layer switch-case statement. State transition occurs only in *evt_e57* for state-2. If state-2 handles the event, its handler function returns null. If state-2 cannot handle the event, it returns the event for further processing by higher level of states.

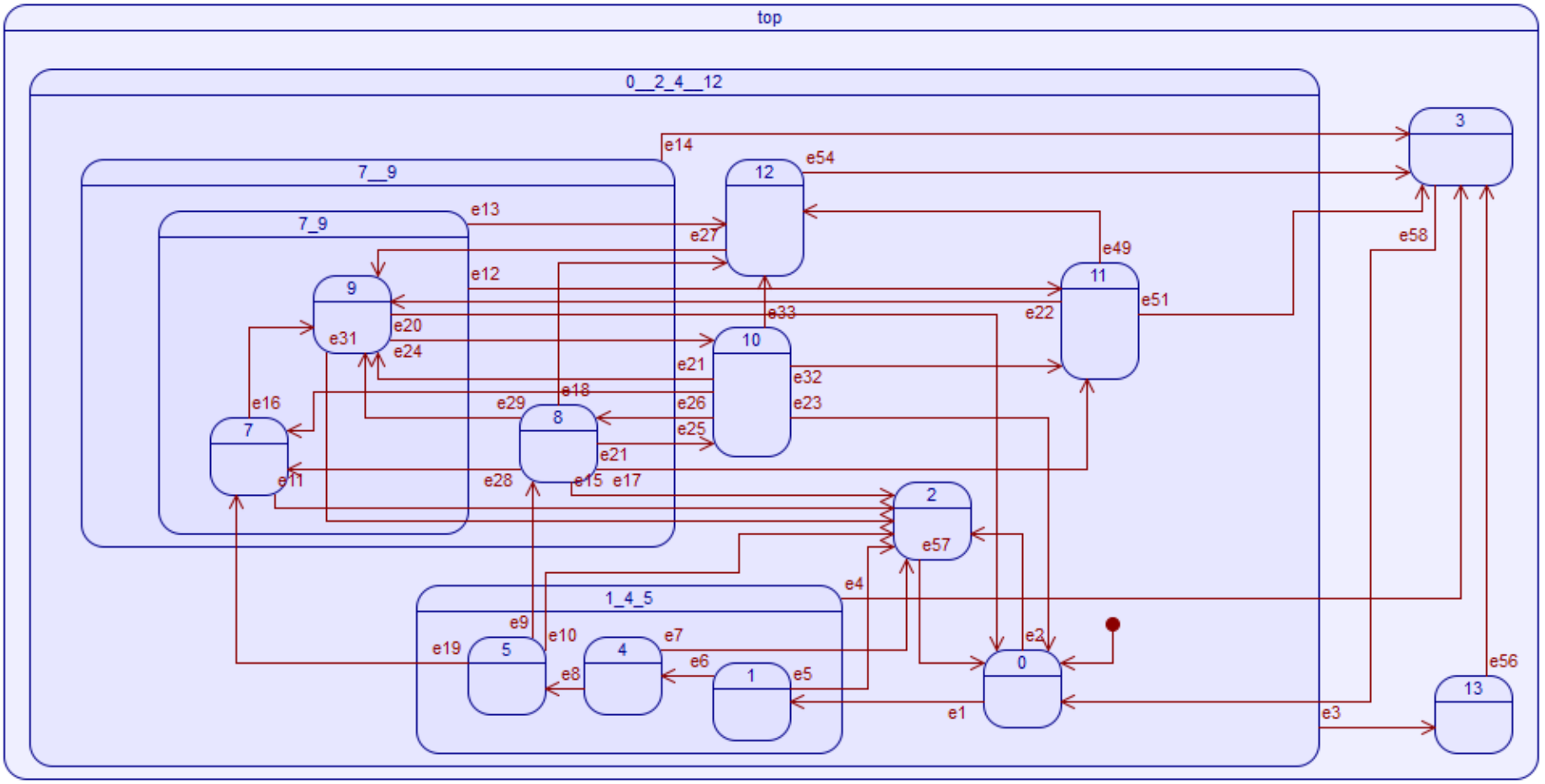

Figure 6.3 : UML statechart diagram for HSM pattern(Type-2) implementation.

6.3.3 Type-3 SP (multi-layer)

SP implies implementing each state in a separate class. It does not say anything about inheriting a state class from other classes. However, in our OO design, inheritance is heavily used as HSM pattern (multi-layer) implementation. Type-2 and Type-3 designs share same state inheritance tree. Type-3 SP design UML class diagram is shown in Figure 6.4.

In all implementations, including Type-3, state and event names are given as numbers, because it is hard to give meaningful names for all states and events.

Route_BaseState is abstract and root. *Tz_Stt_0__2_4__12*, *Tz_Stt_1_4_5*, *Tz_Stt_7__9*, and *Tz_Stt_7_9* classes are also abstract. State machine's current state cannot point to these abstract states. State machine current state can be leaf states; *Tz_Stt_0*, *Tz_Stt_1*, *Tz_Stt_2*, *Tz_Stt_3*, *Tz_Stt_4*, *Tz_Stt_5*, *Tz_Stt_7*, *Tz_Stt_8*, *Tz_Stt_9*, *Tz_Stt_10*, *Tz_Stt_11*, *Tz_Stt_12*, *Tz_Stt_13*.

Abstract states are used for grouping other states that handles same events with same operations. For example, *Tz_Stt_7* and *Tz_Stt_9* are inherited from *Tz_Stt_7_9*, because they handle *evt_e12* and *evt_e13* in a same manner.

Enter and exit methods are added to class in case they are necessary.

6.4 Gathering Metrics

In this study, all metrics used for measuring these implementations are collected manually for each implementation, because there is no tool that can collect the metrics proposed in this thesis.

When calculating NHE in HSM pattern implementations, Type-1 and Type-2, number of cases in switch-case statement in method implementing state is counted. *START_EVT*, *EXIT_EVT* and *ENTER_EVT* are not counted. In SP implementation, Type-3, event handler methods of state class including inherited event handler methods are counted. In Type-3, *Route_BaseState* is omitted because it is abstract, thus does not handle any event.

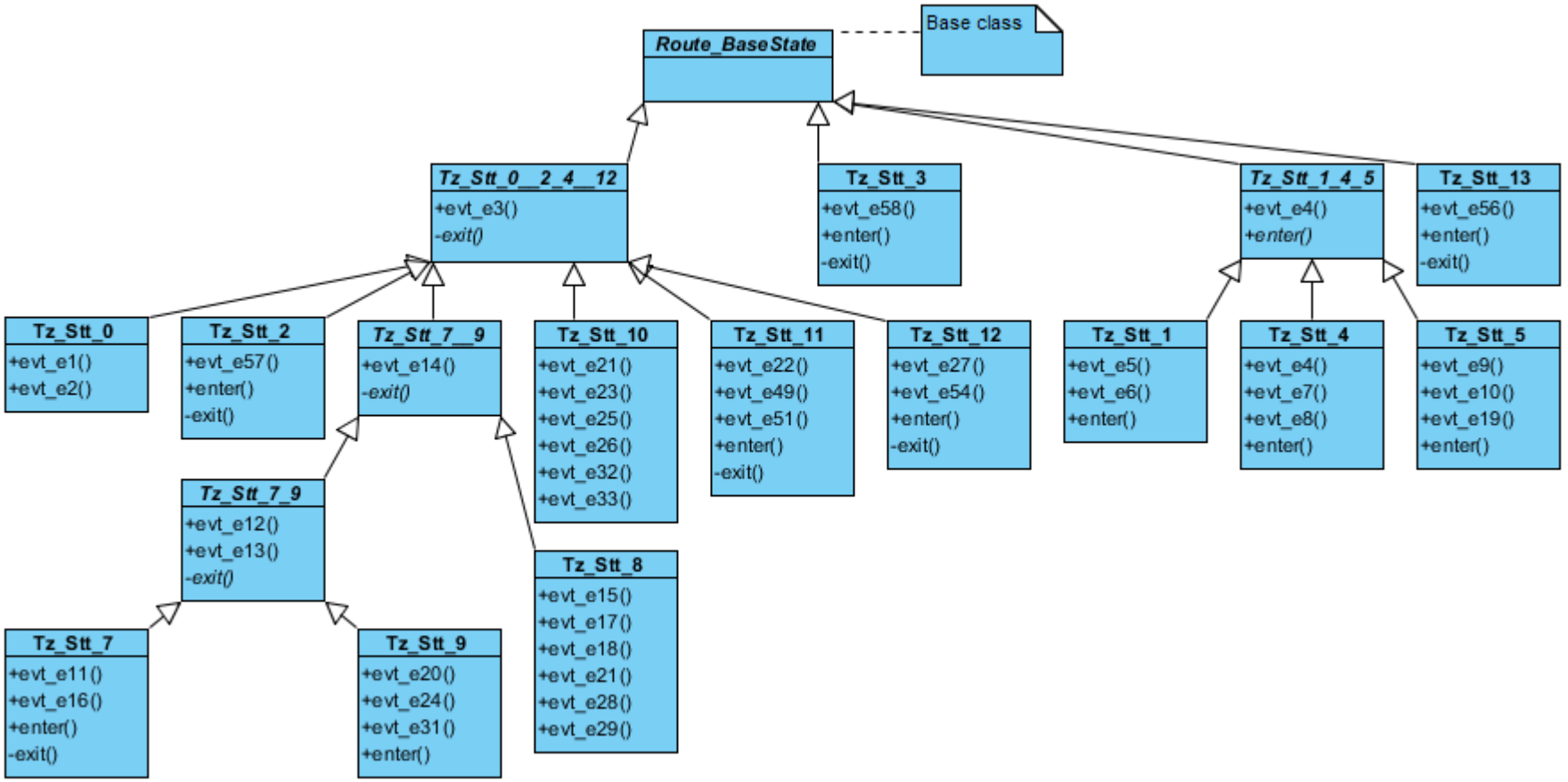

Figure 6.4 : UML class diagram of SP implementation.

DSIT and NOCS are calculated from the state inheritance topology of designs. In HSM pattern designs, top state's DSIT score is zero. In SP pattern, base class, which is *Route_BaseState*, is also included when DSIT is calculated.

In HSM pattern implementations, NOCS is calculated by summing the number of child state for each state. In SP implementation, NOCS is calculated like NOC of a class.

NEA is calculated by only summing the number of events handled in state's handler method. Events that are also handled in super state of the state are not counted. In SP, event handler methods only added to inheritance are summed. Overriding event handler methods of state class is not counted.

In HSM implementations, Type-1 and Type-2, NEO is calculated by summing events that are also handled by super-state. In SP, It is calculated by summing number of overridden event handler methods of a class.

For a state in HSM pattern implementations, CEH is calculated as the McCabe complexity of the method representing the state. *START_EVT*, *EXIT_EVT* and *ENTER_EVT* in switch-case statement are also included to calculation. In SP, CEH is sum of complexity of all event handler methods of a state.

In Type-1 and Type-2, CEnH and CExH is McCabe complexity of enter and exit operations respectively. In SP, CEnH and CExH is McCabe complexity of enter and exit methods of a class respectively.

7. 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.

7.1 Effect of Using Inheritance in HSM Pattern

First, simulator software is implemented without inheritance. Then same simulator is designed and implemented with inheritance. Metric results obtained from Type-1 and Type-2 is explained below.

7.1.1 Results

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 as illustrated in Figure 7.1. 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%.

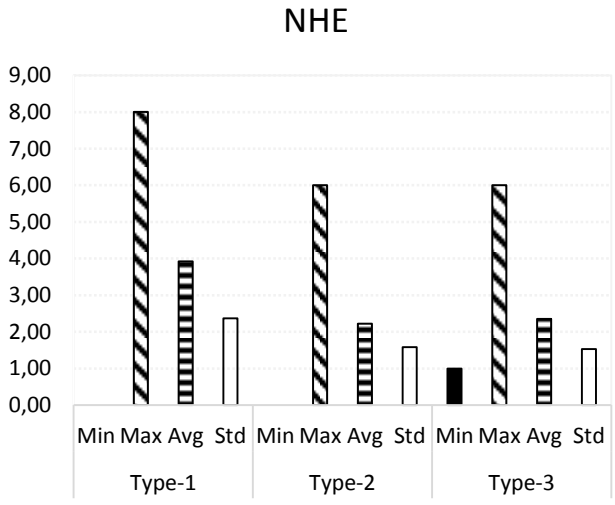

Figure 7.1 : NHE measurement.

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 Figure 7.2. 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.

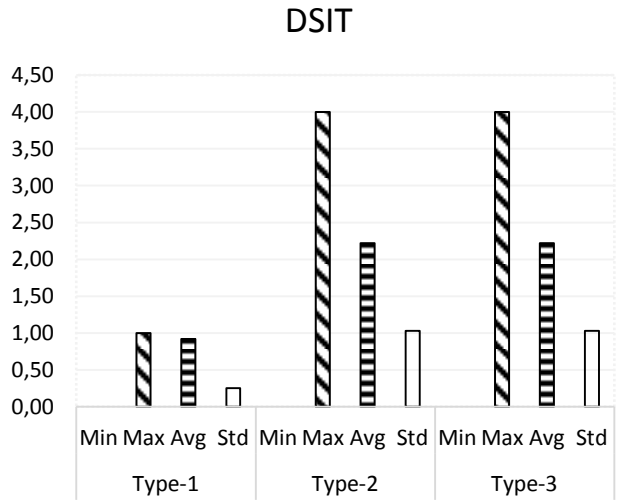

Figure 7.2 : DSIT measurement.

NOCS: Figure 7.3 shows the NOCS scores of three implementations. In Type-1 and Type-2, top states have 13 and 7 child states respectively. In Type-1, maximum score of NOCS is too high because all states are inherited from top state. As stated in [19], high values of NOC may point to misusage of abstraction. Similarly, in state programming, high score of NOCS indicates that inheritance is not used properly in Type-1.

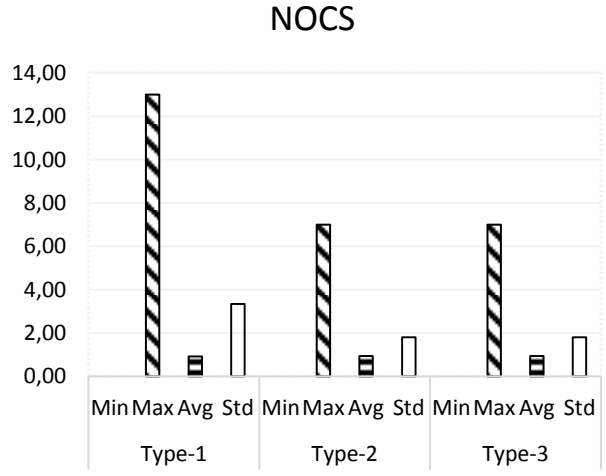

Figure 7.3 : NOCS measurement.

NEA: NEA scores are given in Figure 7.4. 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 [21], new methods added by subclass results in error-prone class. Same approach is also valid for states and NEA metric.

In Type-1 and Type-2, NHE and NEA metric scores are same although they measure different attributes of states. This situation is resulted from states, which does not override its base (super) state's event handler.

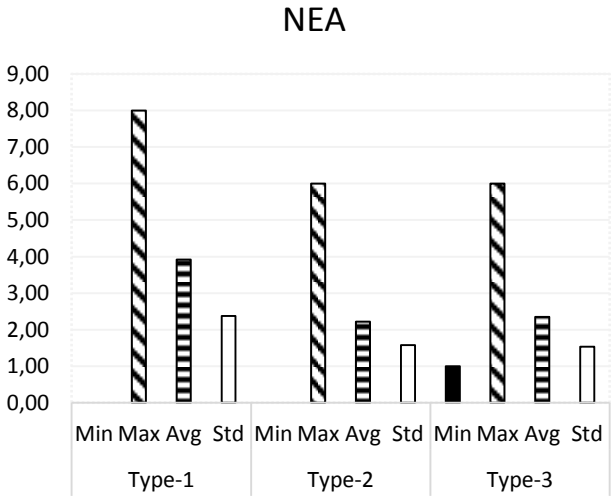

Figure 7.4 : NEA measurement.

NEO: NEO score of both implementations is zero, because none of states overrides its parent state's handlers. NEO metric is not decisive for comparing Type-1 and Type-2 implementations.

CEH: Figure 7.5 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.

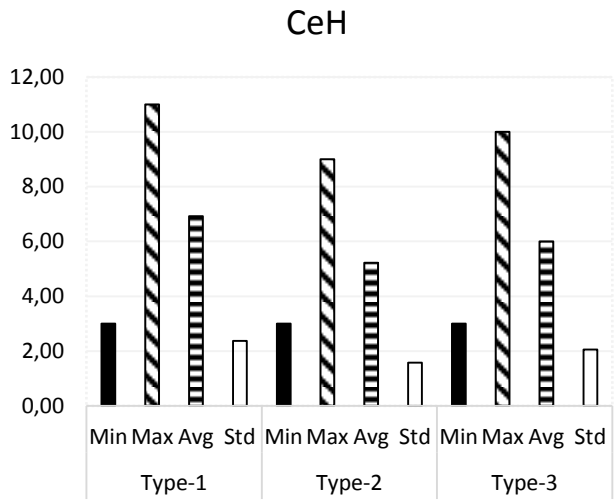

Figure 7.5 : CeH measurement.

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

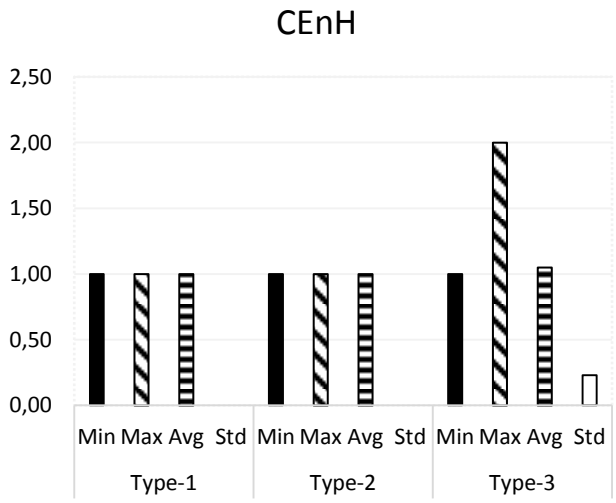

Figure 7.6 : CEnH measurement.

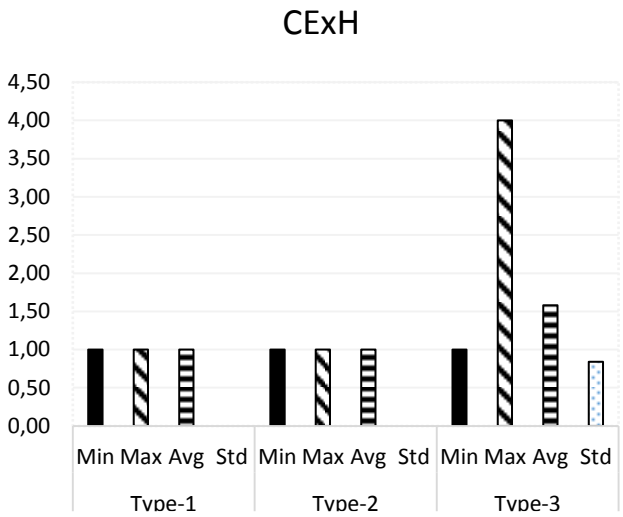

Figure 7.7 : CExH measurement.

7.1.2 Evaluation of results

Statistical testing for significant difference between Type-1 and Type-2 is given in Table 7.1. If *t* is greater than 1.96, then difference is significant otherwise not.

Table 7.1 : Significance test for Type-1 and Type-2.

Metrics	***t*(test statistics, confidence level: 95%)**
NHE	3.127
DSIT	7.606
NOCS	0.027
NEA	3.127
NEO	NAN
CEH	3.127
CEnH	NAN
CExH	NAN

Table 7.1 shows that NHE, DSIT, NEA, and CEH are significant for comparison.

By comparing Type-1 and Type-2 from the perspective of inheritance, we found that DSIT metric is inversely proportional to the complexity metrics. As inheritance is applied in Type-2, average values of NHE, NEA, and CEH are decreased. By sub-stating, responsibilities are distributed among states and this results in decrease in average complexity. Complex software is hard to maintain and error-prone. Such software modules also need to be tested more elaborately. Cost of handling enter and exit events in both Type-1 and Type-2 is same because they implement these events in a same manner.

7.2 HSM Pattern and SP

In Type-3, same simulator is designed and implemented by using SP. Comparison of Type-2 and Type 3 is given below.

7.2.1 Results

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 Figure 7.1). NHE metric scores of both implementation are similar because they handle same events in same states.

DSIT and NOCS: Both implementations have same DSIT and NOCS score because their state inheritance tree is same. These inheritance-related metrics do not help us compare Type-2 and Type-3 implementations.

NEA: NEA score is same as NHE in Type-2 and Type-3 because none of states re-defines its base (super) state's behavior.

NEO: In all implementations, states do not override its base state's properties. Thus NEO score is zero for all implementations. NEO results do not help us compare these implementations in this study.

CeH: Type-2 and Type-3 have maximum CeH scores 9 and 10 respectively as shown in Figure 7.5. Average CEH scores of Type-2 and Type-3 are 5.22 and 6 respectively. Type-2 is less complex than the Type-3 but it is strong.

CEnH: All states in Type-2 have one CEnH score. However, in Type-3, one state has CEnH score of 2 as shown in Figure 7.6. Type-2 handles enter actions slightly effectively than Type-3.

CExH: All states in Type-2 have CExH score of one. Maximum CExH score in Type-3 is 4, average CExH score is 1.58 as shown in Figure 7.7. Complexity of handling exit actions in Type-3 is 50% higher than Type-2. This indicates that SP cannot handle exit actions as effective as Type-2 does.

7.2.2 Evaluation of results

Statistical testing for significant difference between Type-2 and Type-3 is given in Table 7.2.

Table 7.2 : Significance test for Type-2 and Type-3.

Metrics	***t*(test statistics, confidence level: 95%)**
NHE	0.345
DSIT	0
NOCS	0
NEA	0.35
NEO	NAN
CEH	2.784
CEnH	1.286
CExH	4.085

Table 7.2 shows that CEH, CExH are significant for comparison. CEnH seems not to be significant because state machine we implemented handle few numbers of enter events.

In Type-2 and Type-3, inheritance related metric scores (DSIT, NOCS) are same because they share same state topology. NHE and CEH metric results are also similar because they have same set of states handling events. Main difference between Type-2 and Type-3 is the way they handle enter and exit events. We found that CEnH and CExH scores of HSM pattern is less than SP. HSM pattern handles these events more effectively with the help of HSM engine. In some cases, execution order of enter/exit events depends on the source state or state to be reached. This makes SP pattern implementation more complex than HSM pattern version.

8. CONCLUSION

In this thesis, effects of applying different designs for state machines are empirically studied from the perspective of software maintainability. For experimental work, interlocking software simulator is designed and implemented three times with same functionality. Two versions are implemented with HSM pattern, third version is implemented with SP. In this study, comparison results are based on metrics.

CK and LK provide software metrics, such as WMC, DIT, NOC, NMA, and NMO. These metrics are well known and trustworthy; however, they are only applicable to OO software. SP can be measured with CK and LK metrics; however, these metrics are not suitable for HSM pattern, because of being not OO. For this reason, new metric suit is required.

We proposed state oriented metrics; NHE, DSIT, NOCS, NEA, NEO, CEH, CEnH and CExH. These metrics are originated from CK and LK metrics; however, CK and LK metrics measure class attributes where as our proposed metrics measure state attributes.

NHE is the number of events handled by a state. DSIT and NOCS metrics are similar to their corresponding OO metrics DIT and NOC respectively. DSIT metric provides position of the state in the state inheritance tree. NOCS is the number of child states that state has. NEA is the number of handled events added to the inheritance. Overridden events in a state are not counted. NEO represents the number of overridden events. CEH measures the complexity of state. CEnH and CExH measures enter and exit events respectively.

We measured three implementations with our state-oriented metrics. Comparisons are made as pairs; first, two versions of HSM pattern are compared. Main difference of these implementations is inheritance usage. Our empirical study showed that inheritance usage decreases average complexity of states. In the next comparison, we handle HSM pattern and SP implementations. These implementations are designed according to same inheritance tree. Metric-based comparisons show that HSM

pattern is good at handling enter and exit events. SP gets more complex when handling these events, thus decreases the maintainability of code.

In the light of results provided by this study, if a state machine to be implemented has complex enter and exit actions, it is wise to choose HSM pattern.

In this study, because of the practical reasons, comparisons are made on different versions of the same state machine model. This study does not claim any generalization of the observations on the experiments. Before generalizing the results, different systems that can be model with state machines need to be studied and verified. Nonetheless, this empirical study provides valuable information about SP and HSM pattern.

In this study, we aim to compare different designs applied to same state machine model from the viewpoint of maintenance. However, software has many quality characteristics. As a future work, effects of applying HSM pattern and SP on different aspects of software quality such as performance efficiency, compatibility, reliability can be studied. Moreover, SP variants such as Reflective State Pattern [28] or Persistent State Pattern [29] can be studied empirically.

As a summary, we made two main contributions in this project. One of them is introducing new state-oriented metric suit. By the help of this metrics, any software project, which belongs to different programming paradigm, can be measured. There are some empirical studies about SP [12], that analyzing the effect of using SP on software quality. However, there is no other work on the effect of HSM pattern on software quality in the literature.

REFERENCES

[1] **Harel, D.** (1987). Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8(3), 231-274.

[2] **Deursen, A., & Bruntink, M.** (2004, September). Predicting class testability using object-oriented metrics. Source Code Analysis and Manipulation, 2004 IEEE International Workshop on (pp. 136-145). IEEE.

[3] **Dagpinar, M., & Jahnke, J. H.** (2003). Predicting maintainability with object-oriented metrics –an empirical comparision. In Proceedings of the 10th Working Conference on Reverse Engineering(WCRE). IEEE Computer Society, 2003, 155-164. doi:10.1109/WCRE.2003.1287246.

[4] **Basili, V. R., Briand, L. C., & Melo, W. L.** (1996). A validation of object-oriented design metrics as quality indicators. Software Engineering, IEEE Transactions on, 22(10), 751-761.

[5] **Koten, C. V., & Gray, A. R.** (2006). An application of Bayesian network for predicting object-oriented software maintainability. Information and Software Technology, 48(1), 59-67.

[6] **Zhou, Y., & Leung, H.** (2007). Predicting object-oriented software maintainability using multivariate adaptive regression splines. Journal of Systems and Software, 80(8), 1349-1361.

[7] **Lorenz, M., & Kidd J.** (1994). Object-Oriented Software Metrics (pp. 66-71), Prentice Hall

[8] **Huston, B.** (2001). The effects of design pattern application on metric scores. Journal of Systems and Software, 58(3), 261-269.

[9] **Prechelt, L., Unger, B., Tichy, W. F., Brossler, P., & Votta, L. G.** (2001). A controlled experiment in maintenance: comparing design patterns to simpler solutions. Software Engineering, IEEE Transactions on, 27(12), 1134-1144.

[10**] Bieman, J. M., Straw, G., Wang, H., Munger, P.W., Alexander, R.T.** (2003, September). "Design patterns and change proneness: an examination of five evolving systems". Proceedings of 9th Software Metrics Symposium. pp. 40–49, IEEE. doi: 10.1109/METRIC.2003.1232454

[11] **Ampatzoglou, A., Frantzeskou, G., & Stamelos, I.** (2012). A methodology to assess the impact of design patterns on software quality. Information and Software Technology, 54(4), 331-346.

[12] **Aydınöz, B.** (2006). "The effect of design patterns on object-oriented metrics and software error-proneness" MSc. Thesis, Middle East Technical University

[13] **Türk, T.** (2009). "The effect of software design patterns on object-oriented software quality and maintainability" MSc. Thesis, Middle East Technical University

[14] **Ayata, M.** (2010). "Effect of some software design patterns on real time software performance" MSc. Thesis, Middle East Technical University

[15] **ISO/IEC-25010** (2011). Systems and software engineering – Systems and software Quality Requirements and Evaluation(SQuaRE) – Systems and software quality models, *International Organization for Standardization/International Electrotechnical Commission*, New York.

[16] **Fenton, N. E.** (1991). Software Metrics: A Rigorous Approach. Chapman & Hall.

[17] **Fenton, N. E., & Pfleeger, S. L.** (1996). Software Metrics- A Rigorous and Practical Approach. (pp. 5). International Thomson Computer Press.

[18] **Fenton, N. E., & Neil, M.** (2000). Software metrics: roadmap. Proceedings of the Conference on The Future of Software Engineering, 357-370. doi:10.1145/336512.336588.

[19] **Chidamber, S. R., & Kemerer, C. F.** (1994). A metric suit for object oriented design. Software Engineering, IEEE Transactions on, 20(6), 476-493.

[20] **McCabe, T. J.** (1976). Complexity measure. Software Engineering, IEEE Transactions on, 2(4), 308-320.

[21**] Briand, L. C., Daly, J., Porter, V. & Wüst, J.** (1998). "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems". 5th International Symposium on Software Metrics. pp. 246.

[22] **Weyuker, T. J.** (1988). Evaluating software complexity measures. Software Engineering, IEEE Transactions on, 14(9), 1357-1365.

[23] **Cherniavsky, J. C., & Smith, C. H.** (1991). On Weyuker's axioms for software complexity measures. Software Engineering, IEEE Transactions on, 17(6), 636-638.

[24] **Douglass, B. P.** (1997). Real-Time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley Longman.

[25] **Erich, G., Helm R., Johnson R., & Vlissides J.** (1995). Design Patterns: Elements of Reusable Object-Oriented Software. (pp. 305-314). Addison-Wesley.

[26] **Douglass, B. P.** (1999). Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns (pp. 647-651). Addison-Wesley.

[27] **Samek, M., & Montgomery, P.** (2010). State-oriented programming. Embedded Systems Programming, 13(8), 22-42.

[28] **Ferreira, L. L., & Rubira, C. M.** (1998). The reflective state pattern. Conference on Pattern Languages of Programs (PLOP'98), 1998.

[29] **Saude, A. V., Victorio, R. A. S. S., & Coutinho, G. C. A.** (2010). Persistent State Pattern. In Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP'10), 2010, doi: 10.1145/2493288.2493293.

[30] **URL-1**

<http://support.objecteering.com/objecteering6.1/help/us/metrics/metrics_in_detail/specialization_index.htm >, date retrieved: 15.10.2014

CURRICULUM VITAE

Name Surname: Özdemir KAVAK

Place and Date of Birth: Dinar/1987

E-Mail: ozdemirkavak@gmail.com

B.Sc.: Istanbul Technical University Computer Engineering 2010

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- Tantuğ, A. C., Kavak, Ö., 2014. A Comparative Study on State Programming: Hierarchical State Machine (HSM) Pattern and State Pattern, 6th *International Conference on Software Technology and Engineering(ICSTE 2014), Lecture Notes on Software Engineering*, 3(3), 229-233.