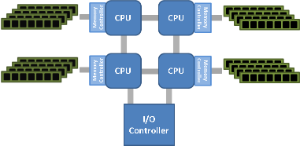
# CHAPTER 1

# INTRODUCTION

In Symmetric Multiprocessor (SMP) systems all physical memory is seen as a single pool. All CPUs share hardware resources and same address space – single instance of kernel. When physical memory and IO devices are equidistant in terms of latency from the set of independent physical CPUs (sockets), the system is called UMA (Uniform Memory Access). In UMA configuration, all physical CPUs access memory from the same memory controller and share the same bus. System configured with a single CPU socket (a socket may have multiple logical cores, each with two hyper threads) is UMA. Building a large SMP server is difficult due to physical limitation of shared bus and higher bus contention with increase in number of CPUs. NUMA (Non Uniform Memory Access) architecture allows designing a bigger system configuration but at a cost of varying memory latencies.

Typically, NUMA systems are made up of a number of nodes, each with its own CPUs, memory and IO devices. Nodes are connected via high speed interconnect, for example Intel QPI - Quick Path Interconnect. Each node uses the interconnect to access memory and IO devices on remote CPUs.  Each NUMA node acts as a UMA SMP system with fast access to local memory and IO devices but relatively slower access to remote nodes memory. By providing each node with its own local memory, it reduces contention issues associated with a shared memory bus, found in UMA servers and thus allow systems to achieve higher aggregate memory bandwidth.  In general, cost of accessing memory increases with the increase in distance from CPU. Thus, more data fails to be local to the node that will access it, the more memory intensive workload performance will suffer from the architecture.

Since each CPU core (physical CPU is made up of multiple logical cores) maintains its own set of caches, it may introduce [cache-coherency](http://frankdenneman.nl/2016/07/11/numa-deep-dive-part-3-cache-coherency/) issues due to multiple copies of same data. Cache coherency means any variable that is to be used by CPU must have a consistent value. In other words, load instruction (read of the memory location) must retrieve the last store (write to the memory). Without cache coherency, data modified in one CPU cache may not be visible to other CPU and that may result in data corruption.



*Fig 1:NUMA topology*

As microprocessor vendors aggressively pursue the production of bigger multi-core multi-chip systems (Intel’s Nehalem-based and Oracle’s Niagara-based systems are typical examples), the computing industry is witnessing a shift toward distributed and cache coherent Non-Uniform Memory Access (NUMA) architectures. These systems contain multiple nodes where each node has locally attached memory, a local cache and multiple processing cores. Such systems present a uniform programming model where all memory is globally visible and cache-coherent. The set of cache-coherent communications channels between nodes is referred to collectively as the interconnect. These inter-node links normally suffer from higher latency and lower bandwidth compared to the intra-node channels. To decrease latency and to conserve interconnect bandwidth, NUMA-aware policies encourage intra-node communication over inter-node communication.

Creating efficient software for NUMA systems is challenging because such systems present a naive uniform “flat” model of the relationship between processors and memory, hiding the actual underlying topology from the programmer. The programmer must study architecture manuals and use special system-dependent library functions to exploit the system topology. NUMA-oblivious multithreaded programs may suffer performance problems arising from long access latencies caused by inter-node coherence traffic and from interconnect bandwidth limits. Furthermore, inter-node interconnect bandwidth is a shared resource so coherence traffic generated by one thread can impede the performance of other unrelated threads because of queueing delays and channel contention. Concurrent data structures and synchronization constructs at the core of modern multithreaded applications must be carefully designed to adapt to the underlying NUMA architectures. Through this project we present a new class of NUMA-aware mutex locks that vastly improves performance over prior NUMA aware and classic NUMA-oblivious locks.

# 

# CHAPTER 2

# LITERATURE SURVEY

This chapter provides a literature survey of works surrounding Java and research into design patterns.A survey of proof systems, Java language semantics and design pattern representations are covered to show how these works are related to each other and how they have developed. design patterns can be represented in such a way as to allow an automated proof system to decide whether or not a class (or set of classes) realises a design pattern in Java. In order to do this, we must be able to:

• Formally describe design patterns

• Analyse a method’s implementation

• Devise a proof system capable of showing the formalised design patterns meet the Java implementation

**2.1 Design patterns**

Design patterns have been used by the object-oriented community for many years in the seminal catalogue .One of the key features of a design pattern is that it is an abstraction from the underlying implementation, which means that a pattern is by definition an intangible product. As a result, a specification of a design pattern needs to work at the abstract level rather than the concrete.There are several works that specify or reason about design patterns. Each has a differentway of representing patterns, depending on the nature of how the patterns are used. A briefdescription of each is given below.

**2.1.1 Specification**

The specification of patterns has become an interesting question since the adoption of pattern catalogues. As noted earlier, patterns are often specified by example using UML as a notation for presenting a concrete example.However, since the pattern itself is abstract, and the example concrete, the reader needs to infer the important parts of the pattern in order to use it.

As a result, there has been work towards a specification for design patterns that allows the pattern to be represented as an abstract concept, rather than exemplified. Some specifications are also aimed at the automated processing or refactoring of design patterns.

**2.1.2 Tool support**

Several pieces of work aimed at the specification of design patterns for use within tools have been done at Utrecht University. The first describes tool support for object-oriented patterns in which patterns are integrated into the development process. The fragment tool provides templates from which design patterns can be instantiated in code; in essence, these templates form the specification of the pattern itself. The goal is to allow patterns to be treated as “first-class citizens in an integrated object-oriented development environment” LOOP focusses on the Java programming language, and has an immediate benefit in that the Java programming environment itself is written in Java.

As such,the program can be easily parsed and processed using tools inherent in the Java operating environment. Additionally, the tool is built upon the Java Refactoring Browser which allows classes and methods to be refactored. A more detailed view of refactoring and design patterns is given below.

Patterns are represented as declarative constraints on a set of classes. These declarative constraints can be combined to form pattern specifications, which can then be applied to existing classes in the system.

This leads to two definitions:Pattern fragment An object-oriented artifact that specifies part of a pattern (a method call,required signature etc.).Pattern constraint A requirement that a pattern must do a particular action, such as calling a method or instantiating a class.The pattern specifications can also include meta-information about the pattern itself, such as documentation or intent of the pattern. These are used within the tool to present information to the user when selecting or instantiating a pattern.As well as using the tool for instantiating patterns, it can be used to refactor and repair patterns.

Thus, names of the classes implementing the patterns may be refactored, but the fragments defining the pattern will still be associated with the code; and hence will still be documented as implementing the pattern. If the pattern is broken during the refactoring, then the fragments can be used to repair the break; for example, if a singleton class is split, the fragment can be detached and then re-attached to the correct class; and from there, the pattern can be re-instantiated.

This was followed up with a framework for representing patterns which defined a way of extending the definition of patterns so that they can be broken down into groupings. In particular, it describes patterns not in terms of structural relations, but in terms of the participants of those patterns. It also highlights a number of very useful observations:

• Some design patterns may share similarities with others, both in terms of their intent and implementation. If this is the case, then the common features can be described as minipatterns, which may then provide building blocks as a way to integrate other patterns in the future.

• Structure is not enough to describe the pattern. The problem is that the structure is only part of the solution; how the classes are related in intent can be just as important as how they are implemented (see the discussion of Command earlier).

• Participants are always present in a pattern. Patterns are only useful when they are used,but it is not always easy to see how the patterns are connected. For example, they may be directly connected via an instance variable reference, or they may be indirectly connected by many list-like data structures.

• Designing an abstract pattern language for transformation into several languages makes sense if there are more than three languages. However, the abstract pattern language would have to encompass every feature from every language, and would have to have emulated features for those languages which did not support it. Additionally, a number of specific features in a language may require tailoring to a specific solution. The author concludes: “All in all, it is practically impossible to provide readable translations between two arbitrary object-oriented languages without loss of meaning” .

**2.1.3 Pattern specification**

Amnon Eden has worked towards a representation for design patterns. He started with the concept of the PatternsWizard in which patterns are processed by a tool and in which patterns are represented in the “Pattern Specification Language” or PSL. This described patterns in terms of Java metaprograms, which when run, would generate an instance of a design pattern. Interestingly, whilst the pattern specification language was implemented in Java,the objects being modified were Eiffel.The definitions of patterns in PSL are represented as tricks that the wizard can use to apply a pattern to an existing class (or set of classes).

When the trick is applied to a class, features of that pattern are instantiated. As such, the trick encodes the pattern in an imperative way, whichis primarily useful when adding an instance of a pattern to a class.Although not declarative (and thus not easy for use in detecting patterns) PSL did provide some key features that were used later on, and echoes similar observations made by others. For example, it introduced the idea of micro-patterns, by observing that there are some common features in design patterns such as, and that by representing these as individual units it is possible to build up a more fine-grained level of pattern library.

**2.2 Detection**

Kyle Brown investigated automated design pattern detection in Java with a tool called KT. Its aim was to detect where patterns are used in existing Java code, with the intention of being able to detect potential patterns and allow the user to make the patterns more explicit.

Since patterns are often composed of inheritance, aggregation/association and messaging information, these can be used as hints to find where a pattern may be present.Since patterns will often have some key features that are identifiable, a scan of existing code looking for these key features highlights potential uses of a particular pattern. Patterns that have multiple key features may be easier to detect; or at least, the tool can give a higher confidence that a pattern is present.

**2.3 Refactoring**

Refactoring is the ability to change a computer system in such a way as to leave the behavior unchanged. Refactoring was introduced to formally explain how behaviour-preserving transformations8 can be made on existing code. It was made into a tool in the Java Refactoring Browser and now refactoring is a key component in Integrated Development Environments (IDEs). More recently, the catalogue has been presented .Since refactoring changes the design of code, but does not change its functionality, it is often used to improve the design of code in order to make it more flexible, more elegant or more maintainable. It is therefore possible to use a sequence of refactorings to introduce a design pattern into a system. Refactoring to instantiate patterns has been investigated in work such as and introducing design patterns.Refactoring to introduce design patterns is a way of changing code in order to make a pattern present. For example, in order to implement the Command pattern, it is necessary to have an abstract class (representing a generic command) and then subclasses that provide the required behaviour. It is possible to use low-level refactorings to introduce a Command pattern manually; for example, one could be achieved as follows:

1. Create a new target class to act as the placeholder for the command.

2. Select the source class containing the method that needs to be encapsulated as a command.

3. Create an instance of the command in the source class.

4. Move the method from the source class to the target class (and replace it with an instance call).

5. Declare a concrete subclass of the command class.

6. Push down the definition of the method into the subclass.

7. Make the parent class abstract.

**CHAPTER 3**

**SOFTWARE REQUIREMENT SPECFICATION**

**3.1 Purpose:**

This project aims to overcome the limitations of traditional mutex locks and spin locks and improve the performance of these synchronization constructs by optimal utilization of the NUMA architecture.

**3.2 Used Software:**

The following table shows the software tools that are used within the project development during the design and implementation.

|  |  |
| --- | --- |
| **SOFTWARE** | **USE** |
| g++ v4.9 and gcc v6.3 | Used for compiling the code. |
| Red Hat 6 v4.2.0 | It provides the environment for implementing the locks. |
| NUMA libraries(numactl, libnuma) | .  Provide the necessary functions for handling numa nodes. |

**Table 1 includes a list of softwares used .The table is roughly categorized by the purpose of the software.**

**3.2 Usage of softwares in the project**

Practically all described software were utilized for creating and addressing locality of the proposed lock objects, thereby their increasing the performance.

**3.2.1 Usage of C:**

The C language provides various constructs that support kernel level programming. We make use of these constructs to develop our own version of mutex locks.

One form of coding allowed with the gcc compiler is the ability to do inline assembly code. As its name implies, inline assembly does not require a call to a separately compiled assembler program. By using certain constructs, we can tell the compiler that code blocks are to be assembled rather than compiled. Although this makes for an architecture-specific file, the readability and efficiency of a C function can be greatly increased.

Here is the inline assembler construct:

1 asm (assembler instruction(s)

2 : output operands (optional)

3 : input operands (optional)

4 : clobbered registers (optional)

5 );

For example, in its most basic form,

asm ("movl %eax, %ebx");

Let's further explore the passing of parameters.

**Output Operands:**

On line 2, following the colon, the output operands are a list of C expressions in parentheses preceded by a "constraint." For output operands, the constraint usually has the = modifier, which indicates that this is write-only. The & modifier shows that this is an "early clobber" operand, which means that this operand is clobbered before the instruction is finished using it. Each operand is separated by a comma.

**Input Operands:**

The input operands on line 3 follow the same syntax as the output operands except for the

write-only modifier.

**Clobbered Registers (or Clobber List):**

In our assembly statements, we can modify various registers and memory. For gcc to know that these items have been modified, we list them here.

**Parameter Numbering:**

Each parameter is given a positional number starting with 0. For example, if we have one output parameter and two input parameters, %0 references the output parameter and %1 and %2 reference the input parameters.

In practice (especially in the Linux kernel), the keyword asm might cause errors at compile time because of other constructs of the same name. You often see this expression written as \_\_asm\_\_, which has the same meaning.

**3.3 Hardware Requirements**

The following table shows the hardware requirements required for the development of project.

|  |  |  |
| --- | --- | --- |
| NUMBER | DESCRIPTION | ALTERNATIVES |
| 1 | PC with @GB hard-disk, Intel Xeon family processor with a minimum of two NUMA nodes and 256MB RAM | Any configuration more than this can also be used. |

**Table 3:Hardware Requirements**

**3.4 Functional Requirements**

The following table shows the functional requirements required for the development of the project. These requirements are basically plays a major role in the development of the project.

|  |  |  |  |
| --- | --- | --- | --- |
| NUMBER | Requirements | Essential or Desirable | Description of requirements |
| 1 | numactl | Essential | Helps us to control the accessing and addressing of NUMA nodes. |
| 2 | libnuma | Essential | Provides the numa.h library that can be used to access NUMA related functions. |

**Table 4 Functional Requirements**

**3.5 Non Functional Requirements**

Non Functional Requirements are called qualities of a system. In system engineering and requirements engineering, a non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors.

**a.Testability:**

Software testability is a degree to which a software artifact support testing in a given test context. Testability is not only an intrinsic property of a software artifact and cannot be measured directly. Instead testability is an extrinsic property which results from interdependency of the software to be tested and test goals, test methods used, and test resources .

**b. Maintainability:**

With the proposed system the application can be maintained in order to isolate defects, correct defects, meet new requirements, make future maintenance easier, or cope with a changed environment.

**c. Scalability:**

The proposed system has the ability to handle growing amounts of shared memory access in a graceful manner and also it has ability to be enlarged to accommodate that growth.

**d. Performance:**

It improves the performance of mutex and spin locks when compared to their traditional implementation in terms of speed of access.

**CHAPTER 4**

**DESIGN AND ANALYSIS**

**4.1 The history of patterns**

The term pattern was coined by an architect, Christopher Alexander . He noted that when a building was designed, several key features were reproduced similar to previous designs.The advantage of reusing an architectural design is obvious; once a design has been shown to solve a particular problem (such as the simple arch, introduced by the Romansfootnote Yes, but apart from arches, what have the Romans ever done for us? then it can be reused in other buildings for a fraction of the effort taken to design the first instance.

Christopher Alexander noted that several elements were repeatedly reused, and termed these‘patterns’. He catalogued a set of patterns that could be used to recreate building designs easily,and for each pattern, gave it a name (so that it could be referenced easily), described the key features of it (so that it could be reproduced in different situations), and described its advantages and disadvantages (to allow architects to choose between different patterns). Importantly, these patterns could be used in conjunction with each other to provide a complete solution, rather than as separate components.

His work on architectural designs was adopted by object-oriented pioneers to reformulate design patterns to be used in software systems. Catalogues of design patterns appeared such as “Pattern Oriented Software Architecture” and also “Patterns: Elements of reusable software” , colloquially known as “Gang of Four” or “GoF” after the four authors who wrote it.

**4.1.1 Software design patterns**

Design patterns in software applications follow the same goal as Alexander’s design patterns.Each software pattern is named (e.g. Visitor, Singleton, Bridge) in order to provide a common vocabulary between software engineers; key features are demonstrated (with code examples);and a set of advantages and disadvantages are given when considering using the design pattern.

Additionally, patterns often have other embellishments; aliases (some patterns are known by different names: for example, Listener and Observer are the same pattern), relationships with other patterns (to allow choices between different solutions) and other comments indicating the cost or performance benefits between the two. shows an example of a pattern.

In object-oriented systems, such as Java using a design pattern will result in the

creation (or modification) of one or more classes. However, software design patterns can stillbe used in other types of languages; for example, list-based processing in Prolog is a kind ofpattern, and recursive function programming in ML is another type of pattern.Catalogues define conditions for a successful pattern. They must:

1. have a name

2. have been used in several different situations

3. provide a common solution to a common problem

4. have consequences (benefits and drawbacks) of using the solution

The first requirement is obviously necessary; developers need to have a common way of identifying (and talking about) a particular pattern. As patterns are used more often, developers become much more familiar with the terms used discussing them, and thus communication regarding the pattern can be achieved much more quickly.

The second requirement is necessary because design patterns are meant to be reused in several different situations. It is not until a pattern has been used in several different situations

that common parts of the pattern can be identified, and highly specific parts of the pattern can be factored out.

The third requirement is necessary because if the problem is not a common one, then the pattern will not be used frequently. If it is not frequently used, it will become forgotten and thus not achieve its potential. Patterns are like genes; the fitter the pattern is (the more it is used; i.e. the more generic problem it solves) the more likely it will be to be remembered and reused. The last requirement is necessary because in order to properly choose a pattern to solve a particular problem, the developer must know what the advantages and disadvantages are of using that pattern.

**4.2 Artefact**

Each design pattern places a number of constraints on each class. These may be constraints on the inheritance relationship, or on navigability with other classes, or on the implementation or expected behaviour of certain methods .The term artefact will be used to indicate a single constraint on a class. A design pattern can therefore be defined as a set of artefacts that must be present in a class or a set of classes. Thisis similar to the idea of a fragment in but is more encompassing; a fragment merely relates the collaborators together in a design pattern.

Similarly, it defines relations that associate participants in a design pattern, but also includes some aspects of behaviour such as message forwarding. The term artefact is also used in as a way of detecting part of a design pattern’s implementation.Each artefact consists of one or more JAVAstatements. These can then be grouped together to form pattern variants, which in turn can be grouped to provide a pattern definition. An artifact is a smaller requirement than a mini-pattern; the former is a constraint on the way one featuremust be implemented, but a mini-pattern is a common idiom that may be repeated over several design patterns or other code implementations.

**4.3 Formally defining patterns**

In order to verify whether a pattern has been implemented correctly, it is necessary to be able to define the design pattern in such a way as to allow a proof engine to compare the definition against a suspected realisation of that pattern. Unfortunately, there are no immediately available formal definitions of design patterns, as catalogues tend to describe patterns verbally .In some cases, the reader is expected to distill the key points of the design pattern by interpretation of provided example code, as well as understand the basic intent of the design pattern from the description.Additionally, the choice of target language can dramatically affect the way in which a pattern is realised.

Although most object-oriented languages share a large core of similar features(inheritance, overriding) there are a number of features that may be different (static or dynamictyping, static or dynamic method dispatch, single or multiple inheritance). As such, patterns tend to be adapted for the particular language that they are written in, to take advantage of the language’s core features, with the result that some patterns are less useful in certain languages. For example, Python has no need for an Iterator pattern since the ability to iterate over lists is built into the core language, and this is used exclusively to provide iteration over sets of data.However, we need to define patterns in order to be able to verify their implementation. There are three approaches which may be used to define patterns; each of which are discussed below.

**4.4 Run-time semantic definition**

The semantics of programming languages is a widely researched subject. One way of specifying a design pattern would be to create a constraint as to how a program would operate if a particular pattern was present. We can imagine a set of design pattern definitions as a set of Hoare triples , such that given an initial state of the program, a resulting state is guaranteed by the correct existence of a pattern. In order to do this, we need to determine for each pattern what the semantic behaviour needs to be. We could then use tools such as iContract or ESC/Java to determine whether or not the pattern correctly obeyed its specification.

Work such as Bali and LOOP have investigated the semantics of the Java programming language using operational semantics in order to reason about Java programs. Although much of the work in investigating the semantics has been an investigation of the Java language, they provide a foundation to build upon for reasoning about Java programs.However, reasoning about programs is a hard problem; even for simple program execution flows, proofs can become very complex.

The problem is magnified for imperative programming languages; unlike functional or declarative languages (where state is essentially immutable), the combinations of possible state values combined with the possibility of state changing during the program’s execution leads to complex proofs. For example, Bali required some 2400 lines of Isabelle/HOL theories in order to prove that Javalight was typesafe. Other approaches, such as have also been used to prove operational semantics of the Java language.Should a powerful proof system be available, then it may be feasible to verify whether a design pattern is implemented correctly or not .

**4.4.1 Declarative constraint definition**

The final way of specifying patterns is to represent them as a set of constraints in the way the pattern may be represented. These may be constraints on the structure (for example, expressing the existence of a particular inheritance hierarchy), or constraints on method implementations(this method should be side effect free).These constraints can then be checked against the implementation to determine whether or not a pattern is present. In some respect, this is a combination of both approaches; it mixes in some semantic behaviour of the class (this method should invoke another method) as well as

structural relationships (this class should implement this interface).

Furthermore, these can be used to provide specific information when a proof has failed, because it is expecting a pattern to have a certain appearance/behaviour. Marco Mijers proposed breaking down the patterns into fragments as a means of defining them, and was used to build a framework for representing patterns. Fortunately, it is not necessary to develop a full Java semantics in order to verify individual design patterns. Most design patterns only need basic structural and semantic requirements to be met in order to realise a pattern correctly, such as the ability to instantiate a new class, or to ensure that a method is forwarded from one class to another. As a result, a much weaker semantics (than a full language semantics) is required; nevertheless, one which is both useful and tractable.

As an example, consider the Singleton pattern once more; we can express it informally as requiring:

1. Provide a private constructor

2. Provide a single instance via a static final variable

Although this will admit only a certain type of Singleton, it is a specification which is relatively easy to specify, and discharge against an implementation.

**4.4.2 Elements of patterns**

In order to represent a pattern as a set of constraints, we need to identify what the key aspects of each design pattern are. This will also affect the requirements of the pattern language and the proof engine .Although the pattern catalogues such as present a list of design patterns, the descriptions of design patterns are very abstract. Given that the book’s intent is to provide a template or overview of how a pattern can be implemented, this is not unreasonable. Also,since the examples are expected to be exemplary rather than prescriptive, they may be shown in different languages (such as Java, C++ or Java). Furthermore, although a template may be shown for a design pattern, it is quite likely that realisations of the pattern in real-life code may be (subtly) different. As a trivial example, the Command pattern shows an abstract class named Command. However, it is not the name of the class that is important(or even that it be an abstract class; it could be an abstract interface). So even with a simple pattern, there may be many variations that exist in real software.

## 4.5Input Design

In an information system, input is the raw data that is processed to produce output. During the input design, the developers must consider the input devices such as PC, MICR, OMR, etc.

Therefore, the quality of system input determines the quality of system output. Well designed input forms and screens have following properties −

* It should serve specific purpose effectively such as storing, recording, and retrieving the information.
* It ensures proper completion with accuracy.
* It should be easy to fill and straightforward.
* It should focus on user’s attention, consistency, and simplicity.
* All these objectives are obtained using the knowledge of basic design principles regarding −
  + What are the inputs needed for the system?
  + How end users respond to different elements of forms and screens.

### Objectives for Input Design

The objectives of input design are −

* To design data entry and input procedures
* To reduce input volume
* To design source documents for data capture or devise other data capture methods
* To design input data records, data entry screens, user interface screens, etc.
* To use validation checks and develop effective input controls.

### Data Input Methods

It is important to design appropriate data input methods to prevent errors while entering data. These methods depend on whether the data is entered by customers in forms manually and later entered by data entry operators, or data is directly entered by users on the PCs.

**A system should prevent user from making mistakes by −**

* Clear form design by leaving enough space for writing legibly.
* Clear instructions to fill form.
* Clear form design.
* Reducing key strokes.
* Immediate error feedback.

**Some of the popular data input methods are −**

* Batch input method (Offline data input method)
* Online data input method
* Computer readable forms
* Interactive data input

### Input Integrity Controls

Input integrity controls include a number of methods to eliminate common input errors by end-users. They also include checks on the value of individual fields; both for format and the completeness of all inputs.

Audit trails for data entry and other system operations are created using transaction logs which gives a record of all changes introduced in the database to provide security and means of recovery in case of any failure.

## 4.6 Output Design

The design of output is the most important task of any system. During output design, developers identify the type of outputs needed, and consider the necessary output controls and prototype report layouts.

### Objectives of Output Design

The objectives of input design are −

* To develop output design that serves the intended purpose and eliminates the production of unwanted output.
* To develop the output design that meets the end users requirements.
* To deliver the appropriate quantity of output.
* To form the output in appropriate format and direct it to the right person.
* To make the output available on time for making good decisions.

### Let us now go through various types of outputs –

### External Outputs

Manufacturers create and design external outputs for printers. External outputs enable the system to leave the trigger actions on the part of their recipients or confirm actions to their recipients.

Some of the external outputs are designed as turnaround outputs, which are implemented as a form and re-enter the system as an input.

### Internal outputs

Internal outputs are present inside the system, and used by end-users and managers. They support the management in decision making and reporting.

There are three types of reports produced by management information −

* **Detailed Reports** − They contain present information which has almost no filtering or restriction generated to assist management planning and control.
* **Summary Reports** − They contain trends and potential problems which are categorized and summarized that are generated for managers who do not want details.
* **Exception Reports** − They contain exceptions, filtered data to some condition or standard before presenting it to the manager, as information.

### Output Integrity Controls

Output integrity controls include routing codes to identify the receiving system, and verification messages to confirm successful receipt of messages that are handled by network protocol.

Printed or screen-format reports should include a date/time for report printing and the data. Multipage reports contain report title or description, and pagination. Pre-printed forms usually include a version number and effective date.

**4.7 UML Diagrams**

The Unified Modeling Language (UML) is a graphical language for OOAD that gives a standard way to write a software system’s blueprint. It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the structures and the relationships in a complex system

The Conceptual Model of UML encompasses three major elements:

* Basic building blocks
* Rules
* Common mechanisms

**Basic Building Blocks**

The three building blocks of UML are:

* Things
* Relationships
* Diagrams

**Things**

There are four kinds of things in UML, namely:

* **Structural Things** : These are the nouns of the UML models representing the static elements that may be either physical or conceptual. The structural things are class, interface, collaboration, use case, active class, components, and nodes.
* **Behavioral Things** : These are the verbs of the UML models representing the dynamic behavior over time and space. The two types of behavioral things are interaction and state machine.
* **Grouping Things** : They comprise the organizational parts of the UML models. There is only one kind of grouping thing, i.e., package.
* **Annotational Things** : These are the explanations in the UML models representing the comments applied to describe elements.

**Relationships**

Relationships are the connection between things. The four types of relationships that can be represented in UML are:

* **Dependency** : This is a semantic relationship between two things such that a change in one thing brings a change in the other. The former is the independent thing, while the latter is the dependent thing.
* **Association** : This is a structural relationship that represents a group of links having common structure and common behavior.
* **Generalization** : This represents a generalization/specialization relationship in which subclasses inherit structure and behavior from super-classes.
* **Realization** : This is a semantic relationship between two or more classifiers such that one classifier lays down a contract that the other classifiers ensure to abide by.

**Diagrams**

A diagram is a graphical representation of a system. It comprises of a group of elements generally in the form of a graph. UML includes nine diagrams in all, namely:

* Class Diagram
* Object Diagram
* Use Case Diagram
* Sequence Diagram
* Collaboration Diagram
* State Chart Diagram
* Activity Diagram
* Component Diagram
* Deployment Diagram

**Rules**

UML has a number of rules so that the models are semantically self-consistent and related to other models in the system harmoniously. UML has semantic rules for the following:

* Names
* Scope
* Visibility
* Integrity
* Execution

**Common Mechanisms**

UML has four common mechanisms:

* Specifications
* Adornments
* Common Divisions
* Extensibility Mechanisms

**Specifications**

In UML, behind each graphical notation, there is a textual statement denoting the syntax and semantics. These are the specifications. The specifications provide a semantic backplane that contains all the parts of a system and the relationship among the different paths.

**Adornments**

Each element in UML has a unique graphical notation. Besides, there are notations to represent the important aspects of an element like name, scope, visibility, etc.

**Common Divisions**

Object-oriented systems can be divided in many ways. The two common ways of division are:

**Division of classes and objects** : A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.

**Division of Interface and Implementation** : An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

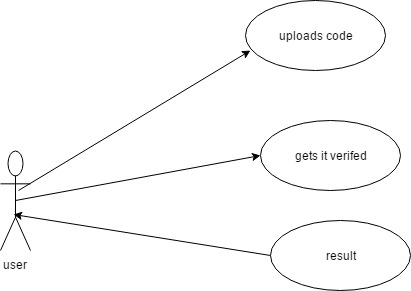
**Extensibility Mechanisms**

UML is an open-ended language. It is possible to extend the capabilities of UML in a controlled manner to suit the requirements of a system.

The extensibility mechanisms are:

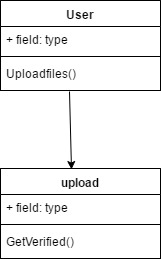
* **Stereotypes** : It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones.
* **Tagged Values** : It extends the properties of UML building blocks.
* **Constraints** : It extends the semantics of UML building blocks.

**4.7.1 UseCase Diagram**

****

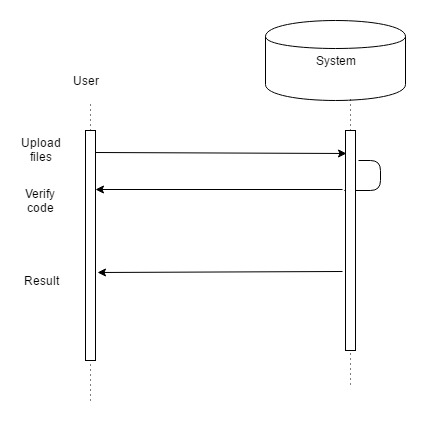
**Fig.3:UseCase Diagram.**

**4.7.2 Class Diagram:**

****

**Fig.4:Class Diagram**

**4.7.3 Sequence Diagram:**



**CHAPTER 5**

**CODING AND IMPLEMENTATION**

**5.1 Implementation**

In this phase, we implemented all the three modules of our project(listed below) adhering to the design mentioned earlier. We used C language to implement mutex locks and C++ to implement mcs-spinlocks.

**5.1.1 Modules**

* Implementing our own variant of Mutex Locks.
* Implementing MCS spinlocks.
* Comparing the performance of our lock implementations with the traditional locks.

**5.1.2 Modules Description:**

**5.1.2.1 Implementing our own variant of mutex locks:**

Mutex is a lock that we set before using a shared resource and release after using it, when the lock is set, no other thread can access the locked region of code.

**5.1.2.2 Implementing MCS locks:**

**5.1.2.3 Comparing the performance of our lock implementations with the traditional locks**

* 1. **Code**
     1. **Code for implementation of mutex locks**

#ifndef MYMUTEX\_H

#define MYMUTEX\_H

#define \_GNU\_SOURCE

#include<linux/futex.h>

#include<sys/time.h>

#include<sys/syscall.h>

#include<unistd.h>

#include<pthread.h>

#define atomic\_xadd(P, V) \_\_sync\_fetch\_and\_add((P), (V))

#define cmpxchg(P, O, N) \_\_sync\_val\_compare\_and\_swap((P), (O), (N))

#define atomic\_inc(P) \_\_sync\_add\_and\_fetch((P), 1)

#define atomic\_dec(P) \_\_sync\_add\_and\_fetch((P), -1)

#define atomic\_add(P, V) \_\_sync\_add\_and\_fetch((P), (V))

#define atomic\_set\_bit(P, V) \_\_sync\_or\_and\_fetch((P), 1<<(V))

#define atomic\_clear\_bit(P, V) \_\_sync\_and\_and\_fetch((P), ~(1<<(V)))

#define cpu\_relax() \_\_asm\_\_ \_\_volatile\_\_ ( "pause" : : : )

/\* Compile read-write barrier \*/

#define barrier() asm volatile("": : :"memory")

/\* Pause instruction to prevent excess processor bus usage \*/

/\*#define cpu\_relax() asm volatile("pause\n": : :"memory")\*/

/\* Atomic exchange (of various sizes) \*/

static inline void \*xchg\_64(void \*ptr, void \*x)

{

\_\_asm\_\_ \_\_volatile\_\_("xchgq %0,%1"

:"=r" ((unsigned long long) x)

:"m" (\*(volatile long long \*)ptr), "0" ((unsigned long long) x)

:"memory");

return x;

}

static inline unsigned xchg\_32(void \*ptr, unsigned x)

{

\_\_asm\_\_ \_\_volatile\_\_("xchgl %0,%1"

:"=r" ((unsigned) x)

:"m" (\*(volatile unsigned \*)ptr), "0" (x)

:"memory");

return x;

}

static inline unsigned short xchg\_16(void \*ptr, unsigned short x)

{

\_\_asm\_\_ \_\_volatile\_\_("xchgw %0,%1"

:"=r" ((unsigned short) x)

:"m" (\*(volatile unsigned short \*)ptr), "0" (x)

:"memory");

return x;

}

/\* Test and set a bit \*/

static inline char atomic\_bitsetandtest(void \*ptr, int x)

{

char out;

\_\_asm\_\_ \_\_volatile\_\_("lock; bts %2,%1\n"

"sbb %0,%0\n"

:"=r" (out), "=m" (\*(volatile long long \*)ptr)

:"Ir" (x)

:"memory");

return out;

}

typedef int mutex;

#define MUTEX\_INITIALIZER {0}

int sys\_futex(void \*addr1, int op, int val1, struct timespec \*timeout, void \*addr2, int val3)

{

return syscall(SYS\_futex, addr1, op, val1, timeout, addr2, val3);

}

int mutex\_init(mutex \*m, const pthread\_mutexattr\_t \*a)

{

(void) a;

\*m = 0;

return 0;

}

int mutex\_destroy(mutex \*m)

{

/\* Do nothing \*/

(void) m;

return 0;

}

int mutex\_lock(mutex \*m)

{

int i, c;

/\* Spin and try to take lock \*/

for (i = 0; i < 100; i++)

{

c = cmpxchg(m, 0, 1);

if (!c) return 0;

cpu\_relax();

}

/\* The lock is now contended \*/

if (c == 1) c = xchg\_32(m, 2);

while (c)

{

/\* Wait in the kernel \*/

sys\_futex(m, FUTEX\_WAIT\_PRIVATE, 2, NULL, NULL, 0);

c = xchg\_32(m, 2);

}

return 0;

}

int mutex\_unlock(mutex \*m)

{

int i;

/\* Unlock, and if not contended then exit. \*/

if (\*m == 2)

{

\*m = 0;

}

else if (xchg\_32(m, 0) == 1) return 0;

/\* Spin and hope someone takes the lock \*/

for (i = 0; i < 200; i++)

{

if (\*m)

{

/\* Need to set to state 2 because there may be waiters \*/

if (cmpxchg(m, 1, 2)) return 0;

}

cpu\_relax();

}

/\* We need to wake someone up \*/

sys\_futex(m, FUTEX\_WAKE\_PRIVATE, 1, NULL, NULL, 0);

return 0;

}

#endif

**Code for verification of mutex lock:**

#define \_GNU\_SOURCE

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<sched.h>

#include "mymutex.h"

#define LIMIT 100000000

long long sum = 0;

mutex \*mut;

void \* increment( void \* arg ){

for ( int i = 0 ; i <= LIMIT ; i ++ ){

mutex\_lock(mut);

sum += 1;

mutex\_unlock(mut);

}

pthread\_exit ( 0 );

}

void \* decrement( void \* arg ){

for ( int i = 0 ; i <= LIMIT ; i ++ ){

mutex\_lock(mut);

sum -= 1;

mutex\_unlock(mut);

}

pthread\_exit ( 0 );

}

int main( void ){

pthread\_t t1,t2;

cpu\_set\_t cpu1, cpu2;

pthread\_attr\_t \*attr1, \*attr2;

attr1 = numa\_alloc\_onnode ( sizeof ( pthread\_attr\_t) , 0 );

attr2 = numa\_alloc\_onnode (sizeof (pthread\_attr\_t),1 );

pthread\_attr\_init ( attr1 );

CPU\_ZERO(&cpu1);

CPU\_SET( 1 , &cpu1 );

pthread\_attr\_setaffinity\_np( attr1 , sizeof ( cpu\_set\_t ) , &cpu1 );

pthread\_attr\_init ( attr2 );

mut = malloc ( sizeof ( pthread\_mutex\_t) );

mutex\_init ( mut, NULL );

pthread\_attr\_init ( attr2 );

CPU\_ZERO(&cpu2);

CPU\_SET( 9 , &cpu2 );

pthread\_attr\_setaffinity\_np( attr2 , sizeof ( cpu\_set\_t ) , &cpu2 );

mut = numa\_alloc\_onnode ( sizeof ( pthread\_mutex\_t) , 0 );

mutex\_init ( mut, NULL );

pthread\_create ( &t1 , attr1, increment , NULL );

pthread\_create( &t2, attr2 , decrement , NULL );

pthread\_join ( t1 , NULL );

pthread\_join( t2 , NULL );

printf("\n sum = %lld \n" , sum );

exit(0);

}

**5.2.2 Code for implementation of MCS locks**

#include <atomic>

class mcs\_lock {

public:

class scoped\_lock;

private:

std::atomic<scoped\_lock \*> \_tail;

public:

mcs\_lock(const mcs\_lock&) = delete;

mcs\_lock& operator=(const mcs\_lock&) = delete;

mcs\_lock() : \_tail(nullptr) {}

class scoped\_lock {

mcs\_lock &\_lock;

scoped\_lock \* volatile \_next;

volatile bool \_owned;

public:

scoped\_lock(const scoped\_lock&) = delete;

scoped\_lock& operator=(const scoped\_lock&) = delete;

scoped\_lock(mcs\_lock &lock) : \_lock(lock), \_next(nullptr), \_owned(false) {

scoped\_lock \*tail = \_lock.\_tail.exchange(this);

if (tail != nullptr) {

tail->\_next = this;

while (!\_owned) {

// spin me right round

}

}

}

~scoped\_lock() {

scoped\_lock \*tail = this;

if (!\_lock.\_tail.compare\_exchange\_strong(tail, nullptr)) {

while (\_next == nullptr) {

// baby right round

}

\_next->\_owned = true;

}

}

};

};

**5.2.3 Code for comparison for mcs spin locks**

#include "mcs\_lock.hpp"

#include "mcs\_lock.hpp"

#include <atomic>

class spin\_lock {

public:

class scoped\_lock;

private:

std::atomic<scoped\_lock \*> \_lock;

public:

spin\_lock(const spin\_lock&) = delete;

spin\_lock& operator=(const spin\_lock&) = delete;

spin\_lock() : \_lock(nullptr) {}

class scoped\_lock {

spin\_lock &\_lock;

public:

scoped\_lock(const scoped\_lock&) = delete;

scoped\_lock& operator=(const scoped\_lock&) = delete;

scoped\_lock(spin\_lock &lock) : \_lock(lock) {

scoped\_lock \*expect = nullptr;

while (!\_lock.\_lock.compare\_exchange\_weak(expect, this)) {

expect = nullptr;

}

}

~scoped\_lock() {

\_lock.\_lock = nullptr;

}

};

};

template<class mutex, class lock\_object>

void spinner(mutex \*m, int \*val, std::size\_t n) {

for (std::size\_t i = 0; i < n; ++i) {

lock\_object lk(\*m);

(\*val)++;

}

}

#include <chrono>

template<typename Clock>

class timer {

std::chrono::time\_point<Clock> \_t0;

public:

timer() : \_t0(Clock::now()) {}

timer(const timer&) = delete;

timer& operator=(const timer&) = delete;

void reset() { \_t0 = Clock::now(); }

operator std::chrono::milliseconds() const {

return std::chrono::duration\_cast<std::chrono::milliseconds>(Clock::now() - \_t0);

}

};

#include <algorithm>

#include <functional>

#include <iostream>

#include <mutex>

#include <string>

#include <thread>

#include <vector>

template<class mutex, class lock\_object>

void test(const std::string &name, std::size\_t nthreads) {

std::vector<std::thread> threads;

mutex m;

int i = 0;

timer<std::chrono::high\_resolution\_clock> t;

std::generate\_n(std::back\_inserter(threads), nthreads,

[&m, &i, nthreads]() { return std::move(std::thread(&spinner<mutex, lock\_object>, &m, &i, (1<<28) / nthreads)); });

std::for\_each(threads.begin(), threads.end(), std::mem\_fn(&std::thread::join));

std::cout << name << " " << nthreads << ":\t" << static\_cast<std::chrono::milliseconds>(t).count() << "ms" << std::endl;

}

int main(void) {

for (std::size\_t i = 1; i <= std::thread::hardware\_concurrency(); ++i) {

test<spin\_lock, spin\_lock::scoped\_lock>("spinlock", i);

}

for (std::size\_t i = 1; i <= std::thread::hardware\_concurrency(); ++i) {

test<std::mutex, std::unique\_lock<std::mutex> >("std::mutex", i);

}

for (std::size\_t i = 1; i <= std::thread::hardware\_concurrency(); ++i) {

test<mcs\_lock, mcs\_lock::scoped\_lock>("mcs\_lock", i);

}

return 0;

}

**CHAPTER 6**

**TESTING**

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub assemblies, assemblies and/or a finished product It is the process of exercising software with the intent of ensuring that the Software system meets its requirements and user expectations and does not fail in an unacceptable manner. There are various types of test. Each test type addresses a specific testing requirement.

6.2 Testing Objectives:

There are several rules that can server as testing objectives.

They are

* Testing is a process of executing a program with the intent of finding an error.
* A good case is one that has a high probability of finding an undiscovered error.
* A successful test is one that uncovers an undiscovered error.

If testing is conducted successfully according to the objectives stated above, it will uncover errors in the software. Also testing demonstrates that software functions appear to the working according to specifications that performance requirements appear to have been set.

**6.3 Types of Testing:**

**(i)Unit Testing:**

Unit testing focuses verification effort on the smallest unit of software design the module. This is also known as module testing. The unit testing is always white box oriented and the step can be contacted in parallel for modules. In this testing each module is found to be working satisfactory as regards to the expected output from the module.

**(ii)Integration Testing:**

Integration testing is a symmetric technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objectives are to take unit tested modules and build a program structure that has been dictated by design.

A set of errors encountered. Correction is difficult because the isolation of causes is complicated by the vast expense of the entire program. Using integrated test plans prepare in the design phase of the system development as guide, the integration testing was carried out. All the errors found in the system were corrected of the next testing steps.

**(iii)System Testing:**

System testing is series of different tests whose primary purpose is to fully exercise the computer based system. Although each test has a different purpose, all the work should verify that all system elements have been properly integrated and perform allocated functions.

**(a)White Box Testing:**

White Box Testing is a testing in which in which the software tester has knowledge of the inner workings, structure and language of the software, or at least its purpose. It is purpose. It is used to test areas that cannot be reached from a black box level.

**(b)Black Box Testing:**

Black Box Testing is testing the software without any knowledge of the inner workings, structure or language of the module being tested. Black box tests, as most other kinds of tests, must be written from a definitive source document, such as specification or requirements document, such as specification or requirements document. It is a testing in which the software under test is treated, as a black box .you cannot “see” into it. The test provides inputs and responds to outputs without considering how the software works.

(iv)Validation Testing:

All the culmination of integration testing, software is completely assembled as package, interfacing error have been uncovered and corrected and a final series of software tests – the validation testing begins. Validation testing can be defined in many ways, but a simple definition is that validation succeeds when the software functions in a manner that can be reasonably expected by the user/customer. Software validation conformity is followed with the following requirements.

* The function or performance characteristics conform to specification and are accepted.
* A deviation from specification uncovered and a deficiency list is created.

(v)Output Testing:

The output generate or displayed by the system under consideration are tested by asking the users about the format required by them. Here, the format is considered into two ways. One is on the screen and the other is printed format.

The output format on the screen is found to be correct as the format was designed in the system design phase according to the user needs. The output testing does not result any correction in the system.

(vi)Performance Testing:

This testing is designed to test the runtime performance of software within the context of an integrated system. This testing occurs throughout all steps in the testing process.

(vii)User Acceptance Testing:

User acceptance of the system is the key factor for the success of any system. The system under consideration was tested for the user acceptance by constantly keeping in touch with the prospective system users at the time of developing and making change were ever required.

### Testing NUMA

### For memory access, two factors determines application performance: Latency and bandwidth. Latency is the time required for the application to fetch data from the processor’s cache hierarchy (L1-L3) and from the physical memory located on local or remote NUMA nodes. Besides latency, microcontroller bandwidth also plays an important role in how fast data can be fed to cpu. Thus measuring memory latencies and throughput are important metrics to establish a baseline for the system under test.

For our project convenience, we have taken an Ec2 instance(m4.10xl) from Amazon Web Services(AWS). The following table describes the specifications of this NUMA supported instance.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instance | L1 | L2 | L3 | Node 0 | Node 1 |
| M4.10xl | 32k | 256k | 30M | Cpu:20  Mem:80G | Cpu:20  Mem:80G |

#### Memory Latency Test

lmbench suite is used to measure memory latency. Latency is reported in nanoseconds (ns) unit for the range of memory sizes. For memory sizes that fits into L1-L3 caches are counted as cache latency and larger sizes represent memory latency.

To install, type:

$ sudo apt-get update ; sudo apt-get -y install lmbench

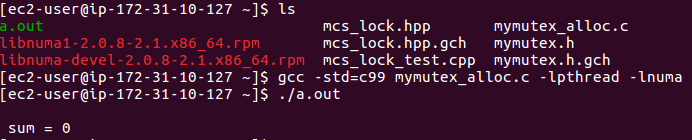
The following Table shows the latency occurred among different levels of NUMA nodes.

|  |  |  |  |
| --- | --- | --- | --- |
| Instance type | Local Latency | Remote Latency | Interleave across all Nodes Latency |
| M4.10xl  2 NUMA nodes | 10ns | 12ns | 11ns |

Test Results of our own variant of mutex locks implementation on NUMA nodes.

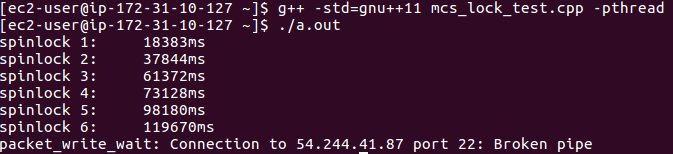
|  |  |  |
| --- | --- | --- |
| Tests done on AWS Ec2 Instance (M4.10xl) | | |
| Lock & Shared object allocation. | Thread mapping Node 0 | Thread mapping Node 1 |
| 0 | 28.138 | 34.359 |
| 1 | 32.268 | 25.919 |

**Test Screenshot1:**

****

Test Results of implementation of spinlocks(mcs locks) on NUMA nodes.

**Test Screenshot 2:**

****

**CONCLUSIONS**

This project aims to prove the hypothesis that it is possible to represent patterns as a

set of constraints on the implementation of one or more Java classes, such that it is

possible to verify whether they realise a pattern correctly.In order to investigate this hypothesis, it was necessary to investigate design patterns to try and ascertain what makes a design pattern have the Quality without a Name, or to identify the key features of design patterns in order to verify their correct implementation.

Given that the pattern is specified as a set of constraints on its implementation, it’s necessary to realise that patterns can be implemented in different ways. Thus, even for common patterns such as Singleton, there may be several ways of solving the problem. The ability to group several variations of a pattern together is an important part of any work that deals with patterns, since there is no one standard way of defining a specific pattern.The same approach could be used with any other proof system that needs to access a potentially small subset of a large amount of data in order to only evaluate what is needed; however, this requires a proof system that is capable of such interaction hooks.

**FUTURE ENHANCEMENTS**

This work has presented a way of representing design patterns in terms of their implementation in Java classes (referred to as the class model in ). The results show that it is possible to represent some (but not all) design patterns using this method. Part of the result of representing patterns in terms of their implementation shows that many patterns are actually implemented using common mini-patterns. For example, the Factory Method described in could be described as a mini-pattern, from which other (larger) patterns such as Abstract Factory are implemented. Specific investigation into patterns and their implementation in Java has led to some obvious candidates for mini-patterns.

It would be interesting to find out whether or not the approaches of using the class model are applicable to other languages. Since this work has focussed entirely on Java, it may be the case that there are aspects that have been made possible because of using the Java language. There may also be additional ways of representing patterns that have not been considered because of limitations or difficulties in defining the patterns in the Java language.

**REFERENCES**

[1] A. Blewitt. A formal catalogue of design patterns. Technical Report Blue book note 1373, Division of Informatics, Edinburgh University, 7 2000.

[2] A. Blewitt. Hedgehog: Automatic Verification of Design Patterns in Java. PhD thesis, School of Informatics, University of Edinburgh, 2005. Unpublished PhD Thesis. <http://www.bandlem.com/Alex/Papers/PhDThesis.pdf>.

[3] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of Java design patterns. In ASE 2001: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, pages 324–327. IEEE Computer Society Press, November 2001. <http://www.ed.ac.uk/_stark/autvjd.html>.

[4] F. Buschmann. Pattern-oriented Software Architecture: A System of Patterns. Wiley, April 1996.

[5] Y. Cheon. A Runtime Assertion Checker for the Java Modeling Language. PhD thesis, Department of Computer Science, Iowa State University, April 2003.

[6] J. Coplien. Pattern Languages of Program Design 1. Pattern Languages of Program Design. Addison Wesley, June 1995.

[7] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, Systems Research Center, Digital Equipment Corporation, December 1998.

[8] A. Eden. LePUS - a declarative pattern specification language. Technical report, Tel Aviv University, 1998. http: //www.cs.concordia.ca/\_faculty/eden/lepus/.

[9] A. Eden. Precise Specification of Design Patterns and Tool Support in Their Application. PhD thesis, Department of Computer Science, Tel Aviv University, 2000. <http://www.eden-study.org>.

[10] A. Eden, J. Gil, and A. Yehudai. Precise specification and automatic application of design patterns. In 12th Annual Conference of Automated Software Engineering, 1997.

[11] D. Ehms. Patternbox eclipse tool. <http://www.patternbox.com>.

[12] Esc/Java 2. web. http: //www.sos.cs.ru.nl/research/escjava/index.html.

[13] C. Flanagan, K. R. M. Leino, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), June 2002. http: //www.hpl.hp.com/personal/Mark Lillibridge/ESCOverview/revised-p25-leino.pdf.

[14] M. Fowler. Refactoring: Improving the design of existing code. Object Technology Series. Addison-Wesley, 2000.

[15] M. Fowler and K. Scott. UML Distilled. Addison-Wesley Professional, September 2003.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of reusable object-oriented software. Professional Computing Series. Addison Wesley, 1995. ISBN 0-201-63361-2.

[17] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison Wesley, 1996. http: //java.sun.com/docs/books/jls/html/index.html.

[18] D. Gruijs. A Framework of Concepts for Representing Object-Oriented Design and Design Patterns. PhD thesis, Department of Computer Science, Utrecht University, August 1998.

[19] B. P. F. Jacobs and E. Poll. A logic for the Java Modeling Language JML. Technical Report

CSI-R0018, Computing Science Institute, University of Nijmegen, December 2000. <http://www.cs.kun.nl/> csi/reports/info/CSI-R0018.html.

[20] R. Kramer. iContract - The Java Design by Contract Tool. In Proceedings of the Technology of Object Oriented Languages and Systems. IEEE Computer Society, 1998.

[21] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user’s manual. Technical Report 2000-002, Compaq Systems Research Center, October 2000.http://gatekeeper.dec.com/pub/DEC/SRC/

technical-notes/abstracts/src-tn-2000-002. html.

[22] M. Meijers. Tool support for object-oriented design patterns. Master’s thesis, Utrecht University, 1996. INF-SCR-96-28.

[23] B. Meyer. Eiffel: The Language. Prentice-Hall, 1999.

[24] B. Meyer. Design by Contract. Prentice Hall, 2002.

[25] M. ´O Cinn´eide. Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, University of Dublin, Trinity College, October 2000.

[26] W. F. Opdyke. Refactoring Object-Oriented frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992. <ftp://st.cs.uiuc.edu/> pub/papers/refactoring/opdyke-thesis.ps.Z.

[27] S. Stelting and O. Maassen. Applied Java patterns. Java series. Prentice Hall, December 2001.

[28] D. Syme. Declarative Theorem Proving for Operational Semantics. PhD thesis, University of Cambridge, Computer Laboratory, 1998.

[29] L. Tokuda. Evolving Object-Oriented Designs with Refactorings. PhD thesis, The University of Texas at Austin, December 1999.

[30] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In Automated Software Engineering, pages 89–120, 2001.

[31] J. Vlissides. Pattern Hatching: Design Patterns Applied. Software Pattern Series. Addison Wesley,July 1998.

**APPENDIX**

**Appendix A:Abbreviations**

1.HyperText Markup Language (HTML).

2.Model View Controller(MVC).

3.Pattern Specification Language(PSL).

4.Integrated Development Environment(IDE).

5.Cascadding Style Sheets(CSS).

6.HyperText Transfer Protocol(HTTP).

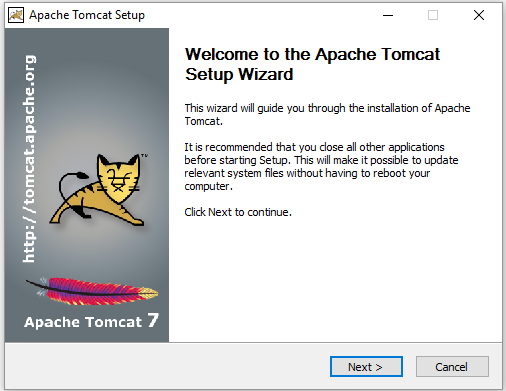
7.Application Program Interface(API).

8.User Interface (UI).

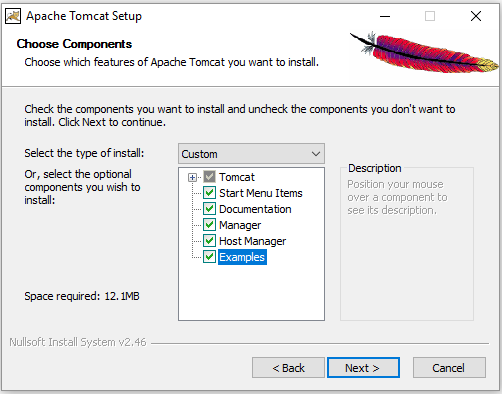
**APPENDIX B: Software installation process:**

**Apache Tomcat:**

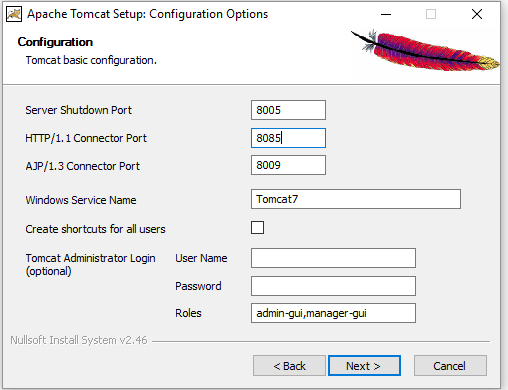
1.Open Apache Tomcat .exe file **(Fig.19)**



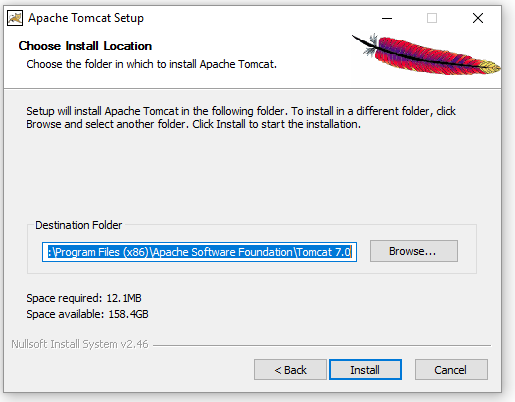
2.Apache Tomcat Setup. Select full option **(Fig.20)**

****

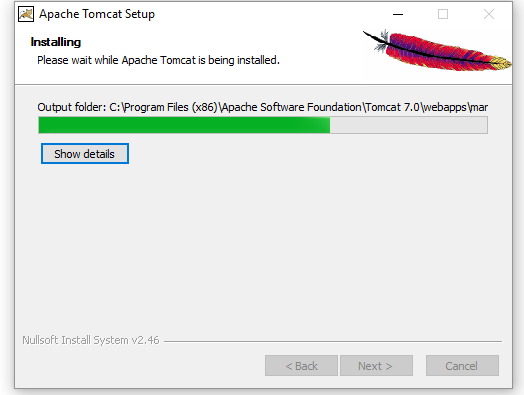
3.Give the Configuration Options **(Fig.21)**



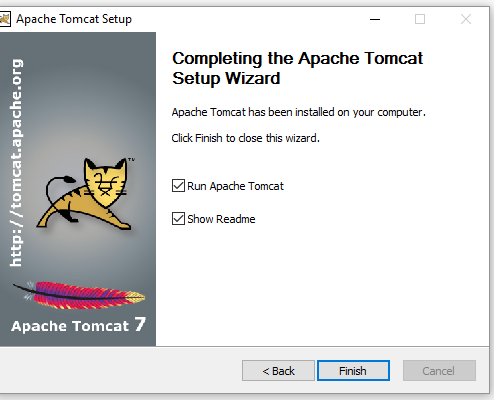
4.Choose The installation location. **(Fig.22)**



5.Installing **(Fig.23)**

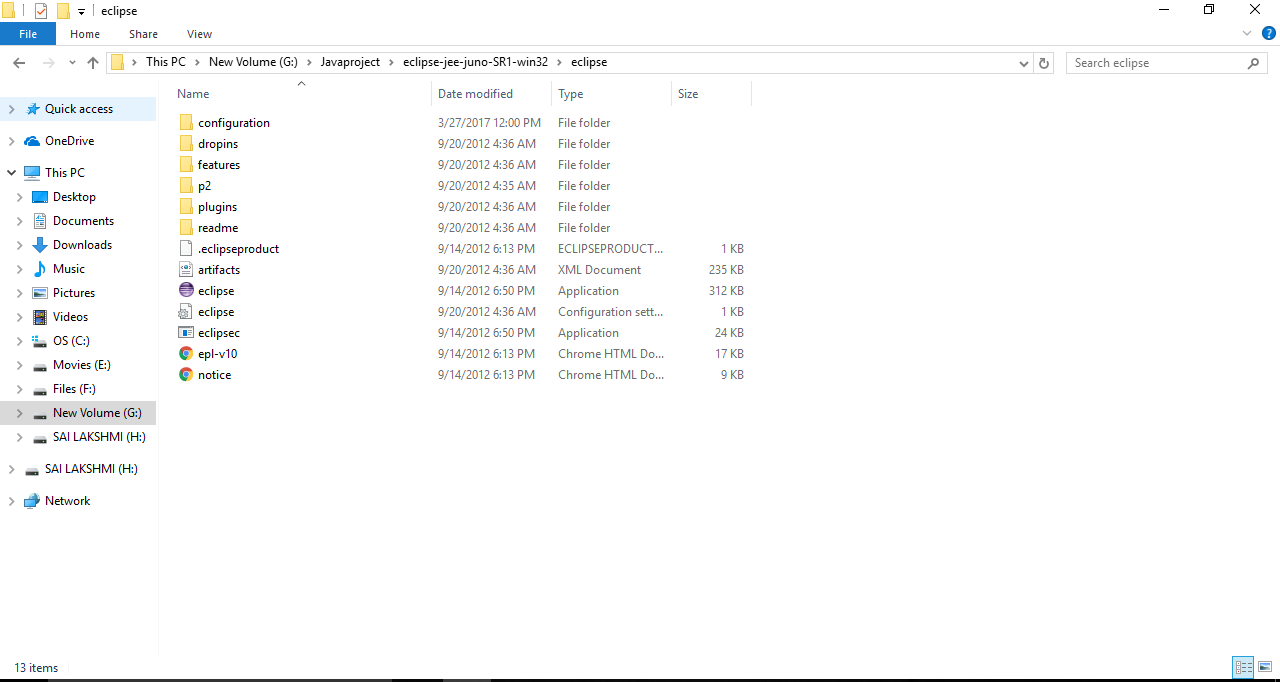


6.Completion Window **(Fig.24)**

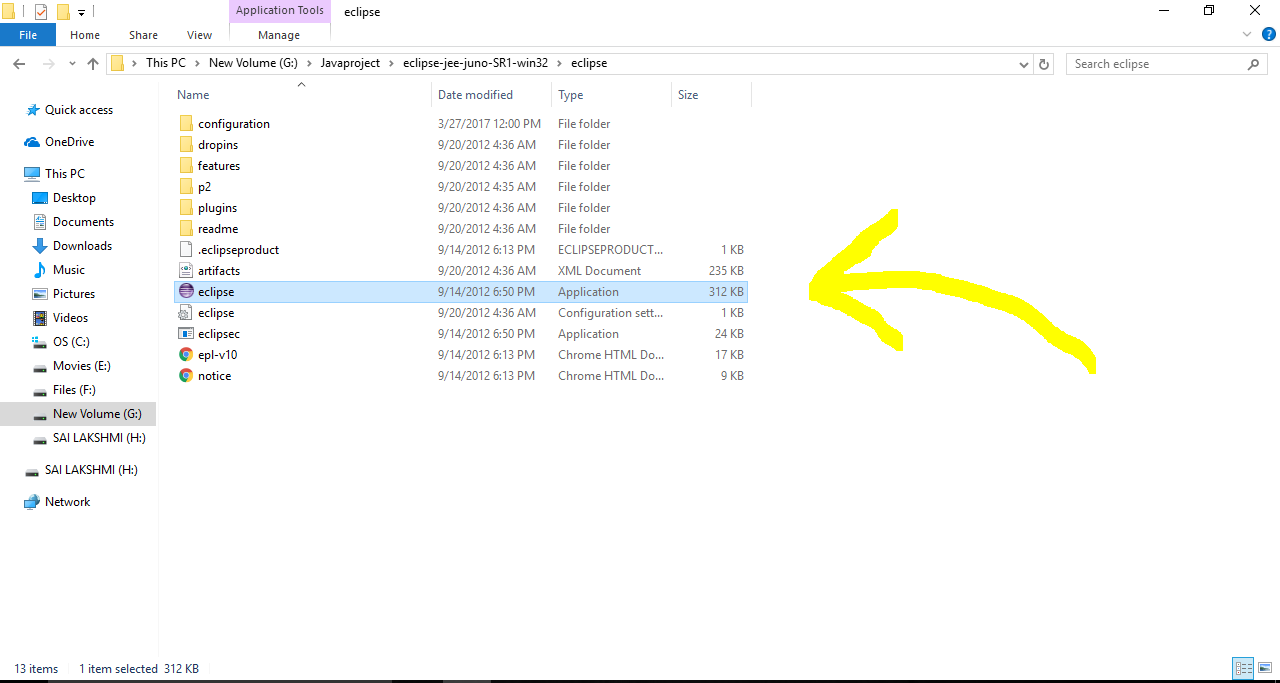


**Eclipse IDE**

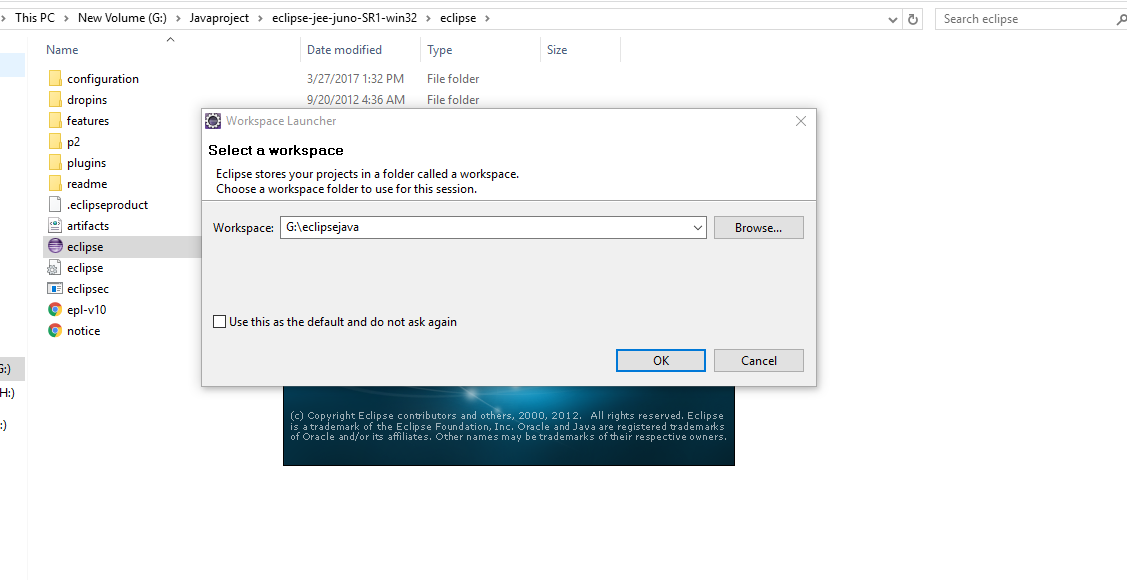
1.Unzip the File **(Fig.25)**



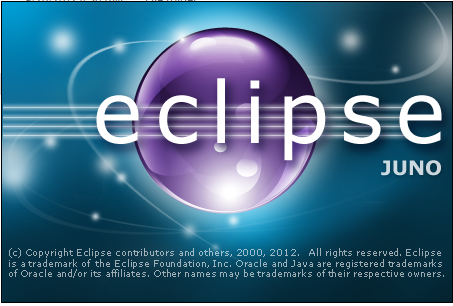
2.Run Eclipse.exe **(Fig.26)**



3.Select workspace **(Fig.27)**



4.Eclipse Window **(Fig.28)**



5.Eclipse Window 2 **(Fig.29)**

