PROJECT INFORMATION

Project Title: Systemic Seismic Vulnerability and Risk Analysis for Buildings, Lifeline Networks and Infrastructures Safety Gain
Acronym: SYNER-G
Project N°: 244061
Call N°: FP7-ENV-2009-1
Project start: 01 November 2009
Duration: 36 months

DELRIVERABLE INFORMATION

Deliverable Title: D7.2 Code for the SYNER-G methodology
Date of issue: November 2012
Work Package: WP7 – Build prototype software
Deliverable/Task Leader: VCE – Vienna Consulting Engineers

REVISION: FINAL

Project Coordinator: Prof. Kyriazis Pitilakis
Institution: Aristotle University of Thessaloniki
e-mail: kpitilak@civil.auth.gr
fax: + 30 2310 995619
telephone: + 30 2310 995693
Abstract

In the Syner-G Project the development of a prototype software for end user was one of the main goals. One objective was to implement the methodologies developed in the various other work packages into the software package. Due to complexity it was decided not to print out all of the written code. The whole code is open source: it can be freely downloaded from the Syner-G Homepage www.syner-g.eu.

Keywords: Software, Code implementation, IT
Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 244061.
Deliverable Contributors

VCE  David Schäfer  Authors
Thomas Gruber
Anna Bosi
Nicola Ondracek
Helmut Wenzel
# Table of Contents

1 Introduction ..................................................................................................................... 10

2 Java.................................................................................................................................. 11
   2.1 INTRODUCTION [1] ................................................................................................. 11
   2.2 JAVA PLATFORM OVERVIEW [3] ........................................................................... 11
      2.2.1 The Java language ........................................................................................ 11
      2.2.2 The Java compiler ........................................................................................ 12
      2.2.3 The Java Virtual Machine ............................................................................. 12
      2.2.4 The garbage collector ................................................................................... 12
      2.2.5 The Java Development Kit ........................................................................... 12
      2.2.6 The Java Runtime Environment . ................................................................. 12
   2.4 PHILOSOPHY OF JAVA: OBJECTS AND OBJECT-ORIENTED PROGRAMMING [2] ............................................................................................................................. 15

3 History of Java – The Java-Timeline.............................................................................. 18
   3.1.1 1991: Exploring Green Fields in Consumer Electronics ......................... 18
   3.1.2 1992: An Innovation Before Its Time......................................................... 18
   3.1.3 1993: Refocusing Efforts to Meet Current Demand ................................ 18
   3.1.4 1994: The First Glimpse of Future Potential ................................................. 18
   3.1.5 1995: The World Takes Notice ..................................................................... 18
   3.1.6 1996: A Language Without Boundaries ....................................................... 18
   3.1.7 1997: Possibilities Become Reality ............................................................... 19
   3.1.8 1998: Futuristic Innovations of a Global Community ..................................... 19
   3.1.9 1999: Focusing on Market Opportunities ...................................................... 19
   3.1.10 2000: Technology That Changes Our Daily Lives ....................................... 19
   3.1.11 2001: A Powerful Community Shapes the Future ........................................ 19
   3.1.12 2002: Taking Java Real-Time ....................................................................... 20
   3.1.13 2003: Remote Computing: From Desktops to the Moon ............................... 20
   3.1.14 2004: Looking to an Open Future ................................................................... 20
   3.1.15 2005: From Web Animation to a Worldwide Force ....................................... 20
   3.1.16 2006: Java Is Open Sourced ........................................................................ 21
   3.1.17 2007: Making Rich Internet Applications Easier to Build ............................. 21
3.1.18 2008: Taking Interactivity to the Next Level with Blu-ray............................... 21
3.1.19 2009: Looking Ahead to a New Chapter ............................................................. 21
3.1.20 2010: Progress for the Java Platform .................................................................. 21
3.1.21 2011: Moving Java Forward ............................................................................. 22

4 Eclipse .......................................................................................................................... 23

4.1 ABOUT THE ECLIPSE FOUNDATION [5]................................................................. 23
  4.1.1 What is Eclipse and the Eclipse Foundation? ...................................................... 23
  4.1.2 IT Infrastructure .............................................................................................. 23
  4.1.3 Intellectual Property (IP) Management ............................................................ 23
  4.1.4 Development Community Support .................................................................. 24
  4.1.5 Ecosystem Development ................................................................................. 24
  4.1.6 A Unique Model for Open Source Development ............................................. 25
  4.1.7 History of Eclipse Foundation ....................................................................... 25

4.2 ECLIPSE SOFTWARE [5] ........................................................................................ 25
  4.2.1 Origin ............................................................................................................. 25
  4.2.2 Introduction .................................................................................................... 26
  Architecture: .......................................................................................................... 26
  4.2.3 26

References ...................................................................................................................... 28
List of Figures

Fig. 2.1 Using the intermediate Java bytecode ................................................................. 14
Fig. 2.2 Relationships between objects ......................................................................... 17
Fig. 4.1 Architecture ...................................................................................................... 26
1 Introduction

In the Syner-G Project Syner-G the development of a prototype software for end user was one of the main very important goals. One objective was to implement the methodologies developed in the various other work packages into the software package. This document describes the usage of the platform, gives examples and explanations about the IT-background and shows how the platform has been developed over time. Due to complexity it was decided not to print out all of the written code. The whole code is open source:, which means it can be freely downloaded freely onfrom the Syner-G Homepage www.syner-g.eu.
2 Java

2.1 INTRODUCTION [1]

The Java™ programming language is a general-purpose, concurrent, classbased, objectoriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design has avoided including new and untested features.

The Java programming language is strongly and statically typed. This specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at run-time. Compile time normally consists of translating programs into a machine-independent byte code representation. Run-time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

The Java programming language is a relatively high-level language, in which details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's free or C++'s delete). High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. The language does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way.

The Java programming language is normally compiled to the bytecoded instruction set and binary format defined in The Java™ Virtual Machine Specification, Java SE 7 Edition.

2.2 JAVA PLATFORM OVERVIEW [3]

Java technology is used to develop applications for a wide range of environments, from consumer devices to heterogeneous enterprise systems. In this section a high-level view of the Java platform and its components is given.

2.2.1 The Java language

Like any programming language, Java language has its own structure, syntax rules, and programming paradigm. The Java language's programming paradigm is based on the concept of object-oriented programming (OOP), which the language's features support.

The Java language is a C-language derivative, so its syntax rules look much like C's: for example, code blocks are modularized into methods and delimited by braces ({{ and }}), and variables are declared before they are used. Structurally, the Java language starts with
packages. A package is the Java language's namespace mechanism. Within packages are classes, and within classes are methods, variables, constants, and so on.

2.2.2 The Java compiler

When programming for the Java platform, source code in .java files must be written and then compiled. The compiler checks the code against the language's syntax rules, then writes out bytecodes in .class files. Bytecodes are standard instructions targeted to run on a Java virtual machine (JVM). In adding this level of abstraction, the Java compiler differs from other language compilers which write out instructions suitable for the CPU chipset the program will run on.

2.2.3 The Java Virtual Machine

At run time, the JVM reads and interprets .class files and executes the program's instructions on the native hardware platform for which the JVM was written. The JVM interprets the bytecodes just as a CPU would interpret assembly-language instructions. The difference is that the JVM is a piece of software written specifically for a particular platform. The JVM is the heart of the Java language's "write-once, run-anywhere" principle. Your code can run on any chipset for which a suitable JVM implementation is available. JVMs are available for major platforms like Linux and Windows, and subsets of the Java language have been implemented in JVMs for mobile phones and hobbyist chips. In the next section, a more thorough review of the Java Virtual Machine will be presented.

2.2.4 The garbage collector

Rather than forcing you to keep up with memory allocation (or use a third-party library to do this), the Java platform provides memory management out of the box. When your Java application creates an object instance at run time, the JVM automatically allocates memory space for that object from the heap, which is a pool of memory set aside for your program to use. The Java garbage collector runs in the background, keeping track of which objects the application no longer needs and reclaiming memory from them. This approach to memory handling is called implicit memory management because it doesn't require you to write any memory-handling code. Garbage collection is one of the essential features of Java platform performance.

2.2.5 The Java Development Kit

When you download a Java Development Kit (JDK), you get — in addition to the compiler and other tools — a complete class library of prebuilt utilities that help you accomplish just about any task common to application development. The best way to get an idea of the scope of the JDK packages and libraries is to check out the JDKAPI documentation.

2.2.6 The Java Runtime Environment

The Java Runtime Environment (JRE; also known as the Java runtime) includes the JVM, code libraries, and components that are necessary for running programs written in the Java language. It is available for multiple platforms. You can freely redistribute the JRE with your applications, according to the terms of the JRE license, to give the application's users a platform on which to run your software. The JRE is included in the JDK.
2.3 A MORE THOROUGH REVIEW OF THE JAVA VIRTUAL MACHINE [2]

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Pascal, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a compiler. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an interpreter, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: they can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, there is a program called "Virtual PC" that runs on Mac OS computers. Virtual PC is an interpreter that executes machine-language programs written for IBM-PC-clone computers. If you run Virtual PC on your Mac OS, you can run any PC program, including programs written for Windows. (Unfortunately, a PC program will run much more slowly than it would on an actual IBM clone. The problem is that Virtual PC executes several Mac OS machine-language instructions for each PC machine-language instruction in the program it is interpreting. Compiled programs are inherently faster than interpreted programs.)

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java Virtual Machine, or JVM. The machine language for the Java Virtual Machine is called Java bytecode. There is no reason why Java bytecode couldn't be used as the machine language of a real computer, rather than a virtual computer. But in fact the use of a virtual machine makes possible one of the main selling points of Java: the fact that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the JVM in the same way that Virtual PC simulates a PC computer. (The term JVM is also used for the Java bytecode interpreter program that does the simulation, so we say that a computer needs a JVM in order to run Java programs. Technically, it would be more correct to say that the interpreter implements the JVM than to say that it is a JVM.)

Of course, a different Java bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program. And the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.
Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are many reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a fairly small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It would be much harder to write a Java compiler for the same computer.

Furthermore, many Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don’t want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program.

When Java was still a new language, it was criticized for being slow. Since Java bytecode was executed by an interpreter, it seemed that Java bytecode programs could never run as quickly as programs compiled into native machine language (that is, the actual machine language of the computer on which the program is running). However, this problem has been largely overcome by the use of just-in-time compilers for executing Java bytecode. A just-in-time compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a just-in-time compiler is a Java bytecode program, and its task is to execute that program. But as it is executing the program, it also translates parts of it into machine language. The translated parts of the program can then be executed much more quickly than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a just-in-time compiler can significantly speed up the overall execution time.

It should be noted that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages could be compiled into Java bytecode. However, it is the combination of Java and Java bytecode that is platform-independent, secure, and network-compatible while allowing you to program in a modern high-level object-oriented language.

(In the past few years, it has become fairly common to create new programming languages, or versions of old languages, that compile into Java bytecode. The compiled bytecode programs can then be executed by a standard JVM. New languages that have been developed specifically for programming the JVM include Groovy, Clojure, and Processing.)
Jython and JRuby are versions of older languages, Python and Ruby, that target the JVM. These languages make it possible to enjoy many of the advantages of the JVM while avoiding some of the technicalities of the Java language. In fact, the use of other languages with the JVM has become important enough that several new features have been added to the JVM in Java Version 7 specifically to add better support for some of those languages.)

2.4 PHILOSOPHY OF JAVA: OBJECTS AND OBJECT-ORIENTED PROGRAMMING [2]

Programs must be designed. No one can just sit down at the computer and compose a program of any complexity. The discipline called software engineering is concerned with the construction of correct, working, well-written programs. The software engineer tries to use accepted and proven methods for analyzing the problem to be solved and for designing a program to solve that problem.

During the 1970s and into the 80s, the primary software engineering methodology was structured programming. The structured programming approach to program design was based on the following advice: To solve a large problem, break the problem into several pieces and work on each piece separately; to solve each piece, treat it as a new problem which can itself be broken down into smaller problems; eventually, you will work your way down to problems that can be solved directly, without further decomposition. This approach is called top-down programming.

There is nothing wrong with top-down programming. It is a valuable and often-used approach to problem-solving. However, it is incomplete. For one thing, it deals almost entirely with producing the instructions necessary to solve a problem. But as time went on, people realized that the design of the data structures for a program was at least as important as the design of subroutines and control structures. Top-down programming doesn't give adequate consideration to the data that the program manipulates.

Another problem with strict top-down programming is that it makes difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top-down programming tends to produce a design that is unique to that problem. It is unlikely that you will be able to take a large chunk of programming from another program and fit it into your project, at least not without extensive modification. Producing high-quality programs is difficult and expensive, so programmers and the people who employ them are always eager to reuse past work.

So, in practice, top-down design is often combined with bottom-up design. In bottom-up design, the approach is to start "at the bottom," with problems that you already know how to solve (and for which you might already have a reusable software component at hand). From there, you can work upwards towards a solution to the overall problem.

The reusable components should be as "modular" as possible. A module is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. The idea is that a module can be "plugged into" a system. The details of what goes on inside the module are not important to the system as a whole, as long as the module fulfills its assigned role correctly. This is called information hiding, and it is one of the most important principles of software engineering.
One common format for software modules is to contain some data, along with some subroutines for manipulating those data. For example, a mailing-list module might contain a list of names and addresses along with a subroutine for adding a new name, a subroutine for printing mailing labels, and so forth. In such modules, the data themselves are often hidden inside the module; a program that uses the module can then manipulate the data only indirectly, by calling the subroutines provided by the module. This protects the data, since it can only be manipulated in known, well-defined ways. And it makes it easier for programs to use the module, since they don't have to worry about the details of how the data are represented. Information about the representation of the data is hidden.

Modules that could support this kind of information-hiding became common in programming languages in the early 1980s. Since then, a more advanced form of the same idea has more or less taken over software engineering. This latest approach is called object-oriented programming, often abbreviated as OOP.

The central concept of object-oriented programming is the object, which is a kind of module containing data and subroutines. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal state (the data it contains) and that can respond to messages (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses. If you send it a message telling it to add a name, it will respond by modifying its state to reflect the change. If you send it a message telling it to print itself, it will respond by printing out its list of names and addresses.

The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other. There is not much "top-down" in the large-scale design of such a program, and people used to more traditional programs can have a hard time getting used to OOP. However, people who use OOP would claim that object-oriented programs tend to be better models of the way the world itself works, and that they are therefore easier to write, easier to understand, and more likely to be correct.

You should think of objects as "knowing" how to respond to certain messages. Different objects might respond to the same message in different ways. For example, a "print" message would produce very different results, depending on the object it is sent to. This property of objects -- that different objects can respond to the same message in different ways -- is called polymorphism.

It is common for objects to bear a kind of "family resemblance" to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same class. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent "drawable objects." They would, for example, all presumably be able to respond to a "draw yourself" message. Another level of grouping, based on the data needed to represent each type of object, is less obvious, but would be very useful in a program: We can group polygons and curves together as "multipoint
objects," while lines, rectangles, and ovals are "two-point objects." (A line is determined by its endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it.) We could diagram these relationships as follows:

![Diagram of object relationships](image)

**Fig. 2.2 Relationships between objects**

DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be subclasses of DrawableObject. The class Line would be a subclass of TwoPointObject and (indirectly) of DrawableObject. A subclass of a class is said to inherit the properties of that class. The subclass can add to its inheritance and it can even "override" part of that inheritance (by defining a different response to some method). Nevertheless, lines, rectangles, and so on are drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful mean for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.
3  History of Java – The Java-Timeline

3.1.1  1991: Exploring Green Fields in Consumer Electronics

At the end of 1990, Sun Microsystems leads the world in the workstation computer market, which is continuing to grow at a healthy clip. Recognizing the potential for applying Sun's innovation and expertise in other areas, the company gathers some of its top engineering minds to examine the up-and-coming consumer electronics market—and inadvertently begins what would become a revolution in computer programming.

3.1.2  1992: An Innovation Before Its Time

Sun's Green Team creates an interactive, handheld home-entertainment device that provides the first glimpse of the potential for its new, processor-independent programming language. The cable companies aren't interested in the technology.

3.1.3  1993: Refocusing Efforts to Meet Current Demand

Sun's Green Project becomes FirstPerson, a wholly owned subsidiary of Sun Microsystems that focuses on building technology for highly interactive devices. The group is later rolled back into Sun and the engineers change their focus from set-top boxes to online services, CD-ROMs, and desktop platforms.

3.1.4  1994: The First Glimpse of Future Potential

Sun engineers Patrick Naughton and Jonathan Payne used the Oak programming language to write WebRunner (later renamed HotJava), the first browser that supports moving objects and dynamic executable content.

3.1.5  1995: The World Takes Notice

With approximately 16 million internet users worldwide, the internet era has begun and Java emerges into the limelight. The object-oriented programming tool merges information and programming to make surfing the Web a more dynamic experience.

3.1.6  1996: A Language Without Boundaries

At the first-ever JavaOne developer conference, more than 6,000 attendees gather to learn more about Java technology and hear a slate of speakers that include Tim Berners-Lee, director, World Wide Web Consortium; James Gosling, vice president and Sun fellow; Dr. Alan Baratz, president of JavaSoft; Dr. Eric Schmidt, Sun chief technology officer and corporate executive officer; and Scott McNealy, Sun chairman of the board, president, and CEO. With a broad range of Java-related product announcements from Sun and other companies and an exhibit hall filled with more than 160 businesses displaying Java products and services, it appears that a whole new industry is growing around a language launched just a year earlier.
3.1.7 1997: Possibilities Become Reality

With approximately 400,000 developers working in Java, it is now the second programming language in the world. More than 10,000 developers flock to the second annual JavaOne developer conference, where Sun announces improved security and compatibility for Java and a range of licensees who plan to take Java beyond the desktop in futuristic devices such as smartcards.

3.1.8 1998: Futuristic Innovations of a Global Community

The 15,000 attendees at this year's JavaOne developer conference sport a Java Ring, a mobile security device and data carrier disguised as a personal fashion accessory. Special readers at the conference's coffee stations automatically deliver each attendee a cup of coffee customized to their preferences, which are stored in the ring's embedded microprocessor. The ring is capable of supporting multiple Java applets that can be loaded dynamically to accomplish a range of tasks—like logging onto a PC, getting cash from an ATM, starting a car, or exchanging contact data with a business acquaintance—and provides a personal demonstration of the potential for Java technology.

3.1.9 1999: Focusing on Market Opportunities

Sun announces a redefined architecture for the Java platform that makes it simpler for software developers, service providers, and device manufacturers to target specific markets. With the introduction of Java 2 Platform, Standard Edition (J2SE) for desktop and workstation devices; Java 2 Platform, Enterprise Edition (J2EE) for heavy-duty server systems; and Java 2 Platform, Micro Edition (J2ME) for consumer devices, it's now easier to capitalize on the Java platform for a growing range of opportunities.

3.1.10 2000: Technology That Changes Our Daily Lives

In the five years since it was announced, Java technology has progressed from a tool to animate Websites to the end-to-end Java 2 platform that spans from smartcards and small consumer devices to enterprise datacenter servers. The breadth of the products leveraging Java technology is vast and includes ATM machines, two-way pagers, mobile phones, personal organizers, games and game machines, cameras, industrial controllers, point-of-sale terminals, desktops, servers, and more. And at the JavaOne developer conference, Apple co-founder and Chief Executive Steve Jobs joins Sun Chairman and CEO Scott McNealy on stage to announce that Apple would bundle Java 2 Standard Edition with every version of its new Mac OS X operating system.

3.1.11 2001: A Powerful Community Shapes the Future

Fewer than half of the 136 Java Specification Requests under review by the Java Community Process (JCP) were submitted by Sun: a sign of how strong and engaged the Java community has become. As Sun's customers, partners, and the industry at large call for Web services support in the Java 2 Platform, Enterprise Edition (J2EE), Sun joins with fellow JCP members to define a new version of J2EE that integrates native support for important Web services technologies. And at the JavaOne developer conference, Sony
Computer Entertainment announces that it will integrate the Java platform into the PlayStation 2 entertainment system, extending the system's ability to download new applications and services dynamically and securely from the internet.

3.1.12 2002: Taking Java Real-Time

During a keynote at the JavaOne developer conference, a sumo battle between two Java technology—powered robots highlights the power of real-time Java. With one robot controlled by embedded devices integrated with wireless Java phones in an end-to-end Web services architecture and the other controlled by James Gosling, vice president and Sun fellow, the battles end in a draw. But the demo illustrates that Java can be applied end to end from the database, through an application server, Web services, wireless connectivity, and down to Java 2 Platform, Micro Edition (J2ME) real-time Java and interfacing to the real world while accommodating diverse clients.

3.1.13 2003: Remote Computing: From Desktops to the Moon

Sun Chairman and CEO Scott McNealy demonstrates the advantages of a Java Card during his keynote at Comdex trade show, showing the IT crowd how a Sun Ray system using Java Card technology enables a user to access a computing session securely from a remote server. Already, thousands of Department of Defense officials use Java Cards to authenticate, and Belgium is building a national ID system around this technology. The SunRay systems can save companies a lot of money and significantly reduce complexity—just as they have done at Sun, where Sun Ray terminals are available in employee cafeterias and spare offices so that employees (including McNealy) can access their desktop instantly.

3.1.14 2004: Looking to an Open Future

At the JavaOne developer conference, the big debate is whether Java should be open sourced. Currently, Sun requires that projects officially based on Java be certified as compatible with the Java specification; amendments to Java must go through Java Community Process (JCP) procedures. Open source advocates seek a freer path for Java. During a panel discussion at JavaOne, representatives from IBM and the Apache Software Foundation endorse an open source model for Java, while Java creator and Sun Fellow James Gosling, along with Sun Vice President and Fellow Rob Gingell and Red Monk Analyst James Governor oppose the move. Gosling warns that allowing multiple, open source implementations of Java technologies could yield the incompatibilities that happened with Unix and is happening again with Linux distributions.

3.1.15 2005: From Web Animation to a Worldwide Force

Java celebrates its tenth anniversary with huge celebrations at JavaOne developer conference and at Sun headquarters. Sun estimates Java now drives more than $100 billion of business annually. It counts more than 4.5 million Java developers, 2.5 billion Java-enabled devices, and 1 billion Java technology-enabled smart cards. Analyst firm Ovum estimates that 708 million Java-enabled handsets were circulating by June 2005.
3.1.16 2006: Java Is Open Sourced

Sun Microsystems releases Java to open source development in a move to drive further adoption of the technology. It is available under the GNU General Public License, the same contract that governs use and development of the Linux operating system. Sun is offering for free all three versions of Java under the GPL: Java Platform, Standard Edition (Java SE), used in building desktop applications; Java Platform, Enterprise Edition (Java EE), used in building server-side applications; and Java Platform, Micro Edition (Java ME), used for handheld devices.

3.1.17 2007: Making Rich Internet Applications Easier to Build

At the JavaOne developer conference, Sun announces JavaFX, a new family of Java technology-based products designed to make it easier to build rich Web sites and Java applications across a broad range of devices. “This is a major step towards helping consumers access the best content on the internet from every device and a significant opportunity for Sun and its partners to deliver a whole new line of products,” says Rich Green, Sun Software executive vice president.

3.1.18 2008: Taking Interactivity to the Next Level with Blu-ray

Rock and roll legend Neil Young takes the stage at the JavaOne developer conference to announce that his long-awaited musical archives will be released on Blu-ray Disc, powered by Java technology. The 10-disc first volume of Young’s archives includes songs, videos, film clips, rare audio interviews, archival photos, press coverage, lyric manuscripts, biographies, tour dates, and more. With Java technology—powered navigation, users can browse visual memorabilia while listening to songs or interviews and download more content directly to the Blu-ray Discs as it becomes available.

3.1.19 2009: Looking Ahead to a New Chapter

With the announced acquisition of Sun by Oracle, the opening session of the JavaOne developer conference features Sun Chairman Scott McNealy and Oracle CEO Larry Ellison onstage together. “Oracle’s middleware is based 100% on Java,” Ellison says. “Java was a very attractive platform for us because it is open and allowed us to extend the platform,” Ellison adds that Sun and James Gosling, Sun vice president and the founder of Java, have done a fantastic job inventing, expanding, and opening up Java and giving it to the world. Looking ahead, he says, “we see increased investments in Java coming from the Sun—Oracle combination and an expansion of the overall community, and we're very excited about that.”

3.1.20 2010: Progress for the Java Platform

The Java Specification Requests (JSRs) for the next two releases of the Java Platform — JSR-336 and JSR-337 — are formally approved by an overwhelming majority in the Java Community Process (JCP) Executive Committee vote. Next, the Java standard will progress through the JCP while the open source reference implementation will be delivered through the OpenJDK project. The plan, which includes community feedback and is endorsed by the
JCP Executive Committee, calls for standardization of these technologies in Java SE 7 within 2011.

3.1.21 2011: Moving Java Forward

The Java Community Process holds a special election to fill three vacant seats on the Java SE/Java EE Executive Committee, and two of the positions go to Java User Groups: the London Java Community, and SouJava, the Brazilian Java User Group, which was nominated by Oracle. This is the first time that two Java User Groups are on this leadership committee.
4 Eclipse

4.1 ABOUT THE ECLIPSE FOUNDATION [5]

4.1.1 What is Eclipse and the Eclipse Foundation?

Eclipse is an open source community, whose projects are focused on building an open
development platform comprised of extensible frameworks, tools and runtimes for building,
deploying and managing software across the lifecycle. The Eclipse Foundation is a not-for-
profit, member supported corporation that hosts the Eclipse projects and helps cultivate both
an open source community and an ecosystem of complementary products and services.

The Eclipse Project was originally created by IBM in November 2001 and supported by a
consortium of software vendors. The Eclipse Foundation was created in January 2004 as an
independent not-for-profit corporation to act as the steward of the Eclipse community. The
independent not-for-profit corporation was created to allow a vendor neutral and open,
transparent community to be established around Eclipse. Today, the Eclipse community
consists of individuals and organizations from a cross section of the software industry.

The Eclipse Foundation is funded by annual dues from our members and governed by a
Board of Directors. Strategic Developers and Strategic Consumers hold seats on this Board,
as do representatives elected by Add-in Providers and Open Source committers. The
Foundation employs a full-time professional staff to provide services to the community but
does not employ the open source developers, called committers, which actually work on the
Eclipse projects. Eclipse committers are typically employed by organizations or are
independent developers that volunteer their time to work on an open source project.

In general, the Eclipse Foundation provides four services to the Eclipse community: 1) IT
Infrastructure 2) IP Management, 3) Development Process, and 4) Ecosystem Development.
Full-time staff are associated with each of these areas and work with the greater Eclipse
community to assist in meeting the needs of the stakeholders.

4.1.2 IT Infrastructure

The Eclipse Foundation manages the IT infrastructure for the Eclipse open source
community, including CVS/SVN code repositories, Bugzilla databases, development oriented
mailing lists and newsgroups, download site and web site. The infrastructure is designed to
provide reliable and scalable service for the committers developing the Eclipse technology
and the consumers who use the technology.

4.1.3 Intellectual Property (IP) Management

An important aspect of Eclipse is the focus on enabling the use of open source technology in
commercial software products and services. We consciously promote and encourage
software vendors to use Eclipse technology for building their commercial software products
and services. This is made possible by the fact that all Eclipse projects are licensed under
the Eclipse Public License (EPL), a commercial friendly OSI approved licensed.
The Eclipse Foundation also undertakes a number of steps to attempt to ensure the pedigree of the intellectual property contained within Eclipse projects. The first step in the due diligence process is trying to ensure that all contributions are made by the rightful copyright holder and under the Eclipse Public License (EPL). All committers are required to sign a committer agreement that stipulates all of their contributions are their original work and are being contributed under the EPL. If a committer is sponsored to work on an Eclipse project by a Member organization, then that organization is asked to sign a Member Committer Access to ensure the intellectual property rights of the organization are contributed under the EPL.

The second step is that the source code related to all contributions which are developed outside of the Eclipse development process are processed through the Eclipse Foundation IP approval process. This process includes analyzing selected code contributions to try to ascertain the provenance of the code, and license compatibility with the EPL. Contributions that contain code licensed under licenses not compatible with the EPL are intended to be screened out through this approval process and thus not added to an Eclipse project. The end result is a level of confidence that Eclipse open source projects release technology that can be safely distributed in commercial products.

4.1.4 Development Community Support

The Eclipse community has a well earned reputation for providing quality software in a reliable and predictable fashion. This is due to the commitment of the committers and organizations contributing to the open source projects. The Eclipse Foundation also provides services and support for the projects to help them achieve these goals.

The Foundation staff help implement the Eclipse Development Process. This process assists new project startup and ensures that all Eclipse projects are run in an open, transparent and meritocratic manner. As part of this process, the Foundation organizes member community reviews for projects to ensure consistent interaction between the projects and the broader membership.

The Eclipse community organizes an annual release train that provides a coordinated release of those Eclipse projects which wish to participate. The release train makes it easier for downstream consumers to adopt new releases of the projects because 1) all the projects are available on the same schedule, so you don't have to wait for independent project schedules, and 2) a level of integration testing occurs before the final release to help identify cross project issues.

4.1.5 Ecosystem Development

A unique aspect of the Eclipse community and the role of the Eclipse Foundation is the active marketing and promotion of Eclipse projects and wider Eclipse ecosystem. A healthy vibrant ecosystem that extends beyond the Eclipse open source community to include things like commercial products based on Eclipse, other open source projects using Eclipse, training and services providers, magazines and online portals, books, etc, are all key to the success of the Eclipse community.

To assist in the development of the Eclipse ecosystem, the Eclipse Foundation organizes a number of activities, including co-operative marketing events with Member companies, community conferences, online resource catalogs (EPIC and Eclipse Live), bi-annual Members meetings and other programs to promote the entire Eclipse community.
**4.1.6 A Unique Model for Open Source Development**

The Eclipse Foundation has been established to serve the Eclipse open source projects and the Eclipse community. As an independent not-for-profit corporation, the Foundation and the Eclipse governance model ensures no single entity is able to control the strategy, policies or operations of the Eclipse community.

The Foundation is focused on creating an environment for successful open source projects and to promote the adoption of Eclipse technology in commercial and open source solutions. Through services like IP Due Diligence, annual release trains, development community support and ecosystem development, the Eclipse model of open source development is a unique and proven model for open source development.

**4.1.7 History of Eclipse Foundation**

Industry leaders Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgain formed the initial eclipse.org Board of Stewards in November 2001. By the end of 2003, this initial consortium had grown to over 80 members.

On February the 2nd, 2004 the Eclipse Board of Stewards announced Eclipse’s reorganization into a not-for-profit corporation. Originally a consortium that formed when IBM released the Eclipse Platform into Open Source, Eclipse became an independent body that will drive the platform’s evolution to benefit the providers of software development offerings and end-users. All technology and source code provided to and developed by this fast-growing community is made available royalty-free via the Eclipse Public License.

The founding Strategic Developers and Strategic Consumers were Ericsson, HP, IBM, Intel, MontaVista Software, QNX, SAP and Serena Software.

**4.2 ECLIPSE SOFTWARE [5]**

**4.2.1 Origin**

In November 2001, IBM released $40 million worth of software tools into the public domain. Starting with this collection of tools, several organizations created a consortium of IDE providers. They called this consortium the Eclipse Foundation, Inc. Eclipse was to be “a universal tool platform — an open extensible IDE for anything and nothing in particular.” This talk about “anything and nothing in particular” reflects Eclipse’s ingenious plug-in architecture.

The initial codebase originated from VisualAge. In its default form it is meant for Java developers, consisting of the Java Development Tools (JDT). Users can extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules. Language packs provide translations into over a dozen natural languages.
Released under the terms of the Eclipse Public License, Eclipse is free and open source software. Eclipse began as an IBM Canada project. It was developed by OTI (Object Technology International) as a Java based replacement for the Smalltalk based VisualAge family of IDE products, which itself had been developed by OTI. In January 2004, the Eclipse Foundation was created. The Eclipse Foundation turned itself from an industry consortium to an independent not-for-profit organization. Among other things, this meant having an Executive Director — Mike Milinkovich, formerly of Oracle Corporation. Apparently, Milinkovich is the Eclipse Foundation’s only paid employee. Everybody else donates his or her time to create Eclipse — the world’s most popular Java development environment. According to IBM Chief Technology Officer Lee Nackman, the name “Eclipse” was chosen to target Microsoft’s Visual Studio product.

4.2.2 Introduction

Eclipse is an open source, extensible, multi-language software development environment, comprising an IDE (Integrated Development Environment), and a plug-in system to extend it. At its heart, Eclipse isn’t only a Java development environment. Eclipse is a vessel — a holder for a bunch of add-ons that form, a Java, C++, or even a COBOL development environment. Each add-on is called a plug-in, and the Eclipse that you normally use is composed of more than 80 useful plug-ins. While the Eclipse Foundation was shifting into high gear, several other things were happening in the world of integrated development environments. IBM was building WebSphere Studio Application Developer (WSAD) — a big Java development environment based on Eclipse. And Sun Microsystems was promoting NetBeans. Like Eclipse, NetBeans is a set of building blocks for creating Java development environments. But unlike Eclipse, NetBeans is pure Java. So a few years ago, war broke out between Eclipse people and NetBeans people. And the war continues to this day.

4.2.3 Architecture:
Eclipse 3.0 (released on June 21, 2004) selected the OSGi Service Platform specifications as the runtime architecture. The Eclipse SDK includes the Eclipse Java Development Tools, offering an IDE with a built-in incremental Java compiler and a full model of the Java source files. This allows for advanced refactoring techniques and code analysis. The Visual Editor project allows interfaces to be created interactively, thus allowing Eclipse to be used as a RAD tool. Eclipse’s widgets are implemented by a widget toolkit for Java called SWT, unlike most Java applications, which use the Java standard Abstract Window Toolkit (AWT) or Swing. Eclipse’s user interface also uses an intermediate GUI layer called JFace, which simplifies the construction of applications based on SWT.

When you download Eclipse, you get the Java IDE (this is the Java Development Toolkit, the JDT) and the Plug-in Development Environment (the PDE) with it. If you only want to develop Java, it’s easy to think of Eclipse as a Java IDE because that’s the main tool you’ll be using. Eclipse itself, however, is a universal tool platform. The JDT is really an addition to Eclipse—it’s a plug-in, in fact. Eclipse itself is a platform which provides support for tools beyond just Java. These tools are implemented as plug-ins, so the platform itself only needs to be a relatively small software package.

The platform provides the support to the plug-ins. If you want to develop Java, you use the JDT plug-in that comes with Eclipse; if you want to develop in C/C++, you can use the CDT plug-in. Installing a plug-in is easy, as we’re going to see in our later posts. All you have to do is download a plugin, drop into the Eclipse plugins directory and restart Eclipse. Eclipse does some checking on each plug-in when it starts, but the plug-ins are not loaded until they’re needed in order to save processing time and memory space.

Eclipse architecture is mainly comprised of Eclipse SDK. Eclipse SDK contains the following components:

- **Workbench IDE** – Comprises of JDT, PDE, Workbench UI, Workspace feature, Search/Compare functionalities etc. It’s the basic graphical interface you work with when you use Eclipse. It’s got toolbars and menus for you to use, and its job is to present those items and the internal windows.
- **Rich Client Platform (optional)** – comprises of Workbench Text Editor, JFace Text, Outline / Properties functionalities etc.
- **Rich Client Platform (base)** – comprises of Workbench UI (Editors / Views / perspective wizards), JFace (viewer classes to bring model view controller programming to SWT, file buffers, text handling, text editors), SWT (portable widget toolkit), OSGI (standard bundling framework) etc.

The stated goals of Eclipse are “to develop a robust, full-featured, commercial-quality industry platform for the development of highly integrated tools.” The Eclipse foundation has focused on three major projects:

- **The Eclipse Project** – is responsible for developing the Eclipse IDE workbench (the “platform” for hosting Eclipse tools), the Java Development Tools (JDT), and the Plug-In Development Environment (PDE) used to extend the platform.
- **The Eclipse Tools Project** - is focused on creating best-of-breed tools for the Eclipse platform. Current subprojects include a Cobol IDE, a C/C++ IDE, and an EMF modeling tool.
- **The Eclipse Technology Project** - focuses on technology research, incubation, and education using the Eclipse platform.
References

Deliverable 2.1 “D2.1 – General methodology for systemic vulnerability assessment,
Syner-G, Project No. 244061, FP7-ENV-2009-1
MAEviz software tool, http://mae.cee.illinois.edu/software_and_tools/maeviz.html
E.Gamma, L. Nackmann, J. Wiegand, „Eclipse Plug.ins“, Third Edition, Addison Wesley,
The Java™ Language. Specification: Java SE 7 Edition. Authors: James Gosling, Bill Joy,
Guy Steele, Gilad Bracha, Alex Buckley Specification: JSR-000091 Java™ Language
Copyright © 2011 Oracle America, Inc. and/or its affiliates. All rights reserved. 500 Oracle
Parkway M/S 5op7, California 94065, U.S.A. http://docs.oracle.com/javase/specs/

Introduction to Programming Using Java, Sixth Edition Version 6.0, June 2011 Author: David
J. Eck (eck@hws.edu) ©1996–2011, David J. Eck. This work is licensed under a Creative Commons Attribution-Noncommercial-ShareAlike
3.0 License. (This license allows you to redistribute this book in unmodified form for non-
commercial purposes. It allows you to make and distribute modified versions for non-
commercial purposes, as long as you include an attribution to the original author, clearly
describe the modifications that you have made, and distribute the modified work under
the same license as the original. Permission might be given by the author for other uses.
See the license for full details.)
http://math.hws.edu/javanotes/

Introduction to Java programming, Part 1: Java language basics. Object-oriented
programming on the Java platform. J Steven Perry (steve.perry@makotoconsulting.com)


http://www.eclipse.org/org/