University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1997-24

Application Frameworks in the Java Environment

Antti Viljamaa

Helsinki, March 1997

Report C-1997-24

University of Helsinki
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

Application Frameworks in the Java Environment

Antti Viljamaa

Department of Computer Science, University of Helsinki
Report C-1997-24
March 1997
32 pages

Abstract

This report is a product of the FRED project (FRamework EDitor for Java) ongoing at the universities of Helsinki and Tampere. The main goal of the project is to enhance the construction of reusable object-oriented software by improving the tools and technology supporting *frameworks* and *design patterns*. On a general level the project also aims at increasing the overall knowledge of frameworks and design patterns in the industry. *Java* has been selected as the implementation language for the tools going to be developed, because of its object-oriented nature, platform neutrality, and the overall research interest towards it.

In this report the suitability of the Java language for framework development is evaluated. This includes the description of the language and the run-time environment, as well as Java's object-oriented features and standard packages. Also a brief look at some frameworks created using Java is given and finally the available Java software development environments are analyzed.

Keywords: Object-Oriented Frameworks, Java, Software Development

CR Classification: D.1.5, D.2.6, D.3

Contents

1. GENERAL REQUIREMENTS FOR A FRAMEWORK LANGUAGE 1

2. JAVA: A LANGUAGE AND A RUN-TIME ENVIRONMENT 3

3. OBJECT-ORIENTED FEATURES OF JAVA 7

4. AN OVERVIEW OF THE STANDARD JAVA PACKAGES 11

5. JAVA FRAMEWORKS 16

5.1 NETWORKING FRAMEWORKS 16
5.2 BUSINESS FRAMEWORKS 21
5.3 MISCELLANEOUS FRAMEWORKS 23

6. JAVA SOFTWARE DEVELOPMENT ENVIRONMENTS 25

7. CONCLUSION 28

REFERENCES 29

1. General Requirements for a Framework Language

Object-orientation is a basic requirement for a language used to build *object-oriented frameworks* [Deu89]. A framework is a set of objects — an application core — that captures the special expertise in some application domain to a reusable form. A whole working application (or a significant portion of it) can be specialized from this skeleton by adding new features to it. The development and use of object-oriented frameworks rely on inheritance, dynamic binding, object composition and delegation. Although different kinds of frameworks have been developed using traditional procedural languages, the object-oriented paradigm offers more powerful concepts for software and design reuse.

The use of a framework requires normally a rather deep understanding of the features and functionality of the framework. On the other hand, framework usage should be easier, quicker and cheaper than building a new system from scratch. Therefore the language and tools applied in framework development should provide some efficient means for documenting the frameworks. Diverse comments in the source code are not enough for explaining the structure and applicability of the framework. The language used to build frameworks should also support the prevention of false uses of a framework. A good documentation guides towards an appropriate application of the framework, but a framework language should also include the kinds of constructs for encapsulation and information hiding that literally force the framework user to utilize it in a right way.

Broad portability is a natural requirement for a framework language. Frameworks should ease the software development process by being suitable cores for widest possible spectrum of applications. Porting existing software to new platforms is a common and very time-consuming task, and therefore many frameworks are designed to facilitate just that process. However, if the implementation language is portable in itself, a framework developer can concentrate on more important problems found in the actual application domain.

Normally framework design concerns rather high-level decisions. Consequently, it is natural to use a high-level programming language for the actual implementation. This smoothens the transfer from design phase to implementation. On the whole, if the language used for building frameworks is simple and includes enough high-level features implying overall robustness, the framework developer can focus on the framework, not on some language specific structures.

In this report the suitability of the Java language for framework development is evaluated. This includes the description of the language and the run-time environment (chapter 2), as well as Java's object-oriented features (chapter 3) and standard packages (chapter 4). Also a brief look at some frameworks created using Java is given (chapter 5) and finally the available Java software development environments are analyzed (chapter 6). This report is closely related to two other reports produced within the FRED

project[1] (FRamework EDitor for Java) underway at the universities of Helsinki and Tampere. These reports include *A Survey of Frameworks* [Hau97] and *Tools Supporting the Use of Design Patterns in Frameworks* [Vil97]. The former describes general framework features and introduces some existing frameworks, and the latter emphasizes pattern-oriented tools developed for framework construction and usage.

[1] The main goal of the FRED project is to enhance the construction of reusable object-oriented software by improving the tools and technology supporting *frameworks* and *design patterns*.

2. Java: a Language and a Run-time Environment

Java is an object-oriented programming language developed by Sun Microsystems [ArG96, Sun96f]. The specifications for Java are freely available so that others can also create compatible compilers and development tools. In this report the references [ArG96, Sun96f, Fla96, and Lea96] were used as a general source of knowledge concerning Java.

Syntactically Java is based on C++ [Str86], but semantically these languages are very different from each other. Java is an interpreted language, so the output of the compilers is not executed directly by a computer. Instead, a Java compiler generates byte code from the original Java source. Byte code is an intermediate form of code that must be interpreted by a Java interpreter on each computer running Java applications. Figure 2.1 shows how software development proceeds in the Java environment.

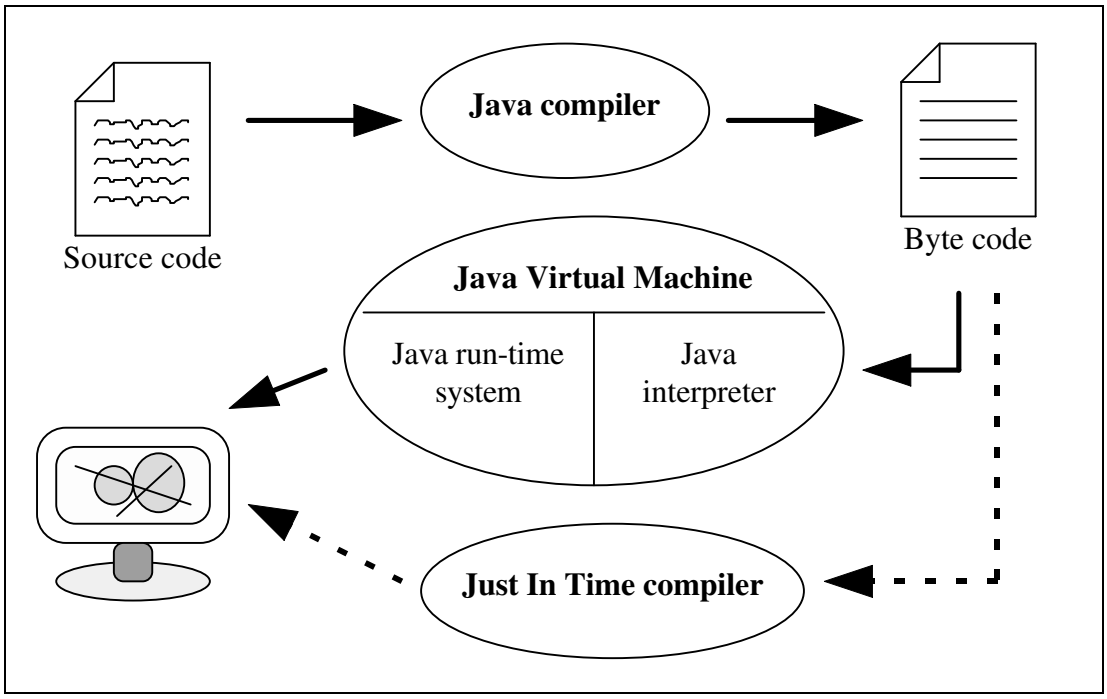

Figure 2.1: Software development in the Java Environment

The byte code provides an architecture neutral object file format. A (compiled) Java program can be run on any system that implements the *Java interpreter* and *run-time system*. These two together are called the *Java Virtual Machine* (JVM). A system must include a JVM, if it is to run Java software. A wide range of computer manufacturers and operating system developers has agreed to maintain up-to-date versions of the JVM.

In addition to interpreting byte code, another way to execute Java programs is to use a *Just In Time compiler* (JIT) written for a specific platform. A Just In Time compiler compiles Java byte code on-the-fly to

the specific native machine code. This method results in nearly as efficient execution as when using C++ or C. Different vendors have developed efficient JIT compilers for a number of platforms (see chapter 6).

The most important contribution of Java is the universal programming interface for all computers[2]. This means the highest possible portability for frameworks: a framework written in Java runs automatically on any system that includes an implementation of the Java Virtual Machine, not even the source code needs to be recompiled on different platforms. Although architecture neutrality is useful for ordinary stand-alone frameworks, it is specially important for network frameworks where different parts of the application could be running on different computer platforms.

There is no actual linking process when developing programs using Java. Linking in Java means just loading new classes into the environment, which is an incremental and pretty cheap process. This implies rapid prototyping and easy experimentation, which facilitates especially framework development. The design process of a framework is usually iterative: different solutions are tested and the framework evolves based on gathered experiences [KoM95, FoO95]. This is because the initial starting point for the design of application frameworks differs crucially from ordinary applications. When developing a framework there is no particular application with a fixed set of requirements that could be analyzed. In contrast, there is an unlimited amount of possible applications which could be developed from one particular framework. An application framework may be adjusted to situations which the designer of the framework has not even considered beforehand.

Interpretation makes Java very dynamic by its nature: classes are loaded (even across the network) as they are needed[3]. In fact, a Java environment consists of a pool of byte code files (local or remote class definitions), which can be used in different applications, without the need for linking them into several different binaries, as is the case when using traditional compiled languages. Java's dynamic features also include a run-time representation for classes — an object's class can be checked during execution. Java's interpretative nature has its shortcomings too: the execution speed of a high-level interpreter is never as efficient as when running native machine code. JIT compilers speed the execution, but they too result in slower speed than when executing an optimized native machine code.

One of the main goals in designing Java was to make it simple [Sun96f]. It can be learnt quickly since it has a relatively small number of language constructs and its syntax is already familiar to C++ programmers. Although Java's syntax is very similar to C++, Java differs from C++ in that many error-prone features of C++ have been left out [Mar97, JaS97]. In contrast to C++, Java is a robust language that eliminates certain types of programming errors completely, thereby assisting in writing correct frameworks.

---

[2] The idea of a universal programming interface is by no means new. In fact already in the late 50's UNCOL (Universal Computer Oriented Language) was proposed to be an intermediate language for all programming languages and compilers. At that time the idea didn't rouse much of an acceptance [Bau76].

[3] The problem is that high bandwidth is needed to send sophisticated applications over the network.

Java's memory model itself is very sound, because of its automatic memory management. Memory management is handled with *garbage collection*, so programmers do not have to worry about memory leaks and pointer problems — such as accidental overwriting of memory and corruption of data.

Java has no goto statement; instead it includes `break` and `continue` structures and sophisticated means for writing elegant error handlers using an object-oriented *exception mechanism*. A programmer can place related error handling procedures in one place with no need to spread them all over the source code. In a framework application a default error handling code can be located inside the framework, and a framework user may specify her own special handlers when extending the framework. In addition, the Java language includes no header files, no operator overloading, and no multiple inheritance. Java is a strongly typed language. It enables extensive compile-time checking for potential type-mismatches and a number of run-time checks by the interpreter (including boundary checks for string and array accesses). Java also insists explicit method declarations, which reduces the number of trivial errors (compared to programs written in C++).

Java has a built-in set of classes (called *packages*) which help framework development even further. The package `java.awt` includes classes for producing graphical user interfaces (GUIs), `java.io` consists of the basic IO classes, `java.net` introduces a set of classes for networking, and `java.lang` contains the classes that are most central to the Java language. Packages are discussed in more detail in chapter 4.

Java provides means for developing framework documentation. Documentation can be handled flexibly using Web pages written in HTML [BeC95]. Since Java provides strong support for producing applications with HTML browsing capabilities, it should be possible to build sophisticated framework documentation tools based on HTML and Java.

As the Internet has gained such a notable role in everyday use of computers, it is predictable that a significant portion of software developed in the near future will include at least some networking features. Java is a language designed primarily for programming on the Internet, and therefore it satisfies some central requirements for networking frameworks: it offers strong support for distribution and programming multiple concurrent threads of execution. Threads are especially useful in programming server frameworks.

Java's Internet features do not imply that it is just a language for writing glittering *applets*[4] on Web pages. Instead Java provides features that enable the development of whole new kinds of applications. As Java facilitates the down loading and execution of programs from arbitrary sources on the Internet, one can easily imagine the growth of *document-centric systems*, where people send each other documents packaged with the tools needed for handling them. In fact, the evolution of Java has also led to the concept of *network computers* — remote devices that will not need disk drives, since they get all their data and program logic

[4] Applets are small Java programs embedded in WWW pages written in HTML.

from the network. Java also enables the development of more complex distributed systems, where a number of people communicate with one another and possibly work together to complete a common task.

A notable degree of security is very important in networking applications. Java is considerably secure in this respect. Although some security faults have been reported (for example in Netscape browsers), they have usually been caused by buggy implementations of the Java Virtual Machine [Pri96]. The language itself provides good means for guaranteeing security. Java has been designed with the premise "nothing is to be trusted" in mind. Actually the security features of the language go hand in hand with its robustness. Java's memory allocation model defines no pointers and a Java compiler does not handle memory layout decisions, so that a programmer is not able to guess the actual memory layout of a program just by looking at its declaration. Memory references in compiled Java code are resolved to real memory addresses at run-time by the Java interpreter. The Java run-time system uses a byte-code verification process to ensure that the code loaded from an arbitrary host does not violate Java's restrictions. This includes, for example, a name space mechanism, which ensures that loaded classes are placed in a different scope than the standard Java classes. In this way the original system classes cannot be replaced by classes loaded from the network. All these features make the development of viruses and other hostile programs rather difficult. On the whole Java can be considered reasonably safe.

3. Object-Oriented Features of Java

A framework must be extendible without modifying the existing code and it has to provide a basic functional logic for the applications derived from it. This implies that the main control of the whole application has to lie inside the part that is provided by the framework, and therefore the framework must be able to call the user defined extension objects as needed (the *Hollywood principle* [Liu96]). These goals can be achieved using object-oriented mechanisms.

All Java programs consist of classes. A class defines the knowledge and behavior for objects that can be instantiated from the particular class. A class can inherit features (methods and data) from another class. Java's object-orientation goes from the ground up: most things are objects except simple numeric types, characters, and the boolean type. The package `java.lang` includes the `Object` class, which is the ultimate superclass of all Java classes. Being an object-oriented language Java provides basic constructs for implementing object-oriented frameworks. *Design patterns* [GHJ95] are important models for constructing flexible frameworks[5]. Java's object-oriented features are designed so that they encourage programming in a design pattern style: using interfaces and delegation rather than implementation inheritance.

Usually a framework consists of abstract classes, which the application developer inherits to make concrete implementations. This is called the *white-box approach* [HJE95, DuJ96]. In a *black-box solution* [HJE95, DuJ96] the application developer constructs the application by creating objects from complete classes and by assembling them. White-box frameworks rely on class inheritance. They can be written in Java using the class extension mechanism: a class can inherit behavior from another class via extending that class (using the `extend` keyword). In a framework certain application dependent methods (*hot spots* [Pre95]) may have a default implementation or they could be declared as abstract (meaning that they have no implementation at all). The framework user can extend those classes and override default implementations or define implementations for methods left abstract in the framework to make complete applications.

In a black-box approach object composition and delegation are used for building frameworks. This is the preferred method in object-oriented development, because it makes applications more flexible and tolerant to changes [GHJ95]. Normally, inheritance should be used only when specializing some general classes into specific ones, not just to copy implementation code. When using delegation, the definitions of abstract interfaces — through which objects are invoked — become the most important aspects of the design. Java offers a dedicated construct, `interface`, for that purpose. Interfaces are like pure abstract classes: they declare only method signatures, not their implementation. At the beginning of a class declaration there is a list of interfaces that the class implements. Every interface can be considered as a new

[5] In addition to design patterns such as those described in [GHJ95], also language specific *idioms* [Cop92] for Java have been introduced. For example, patterns on concurrent programming in Java are discussed in [Lea96].

type, and a class is type-compatible with all the interfaces it implements. Interfaces can extend several other interfaces, but a class can only extend one other class, so there is no real multiple inheritance in Java.

Java's interface concept is mainly designed for avoiding the need for multiple inheritance of classes[6]. Interfaces also make the utilized design patterns (building blocks of frameworks) more clearly distinguishable from the source code, because patterns very often rely on interfaces. Explicit interface declarations make programs more readable.

Objects are created based on classes. Creation is achieved with the special `new` method. During the creation the system calls a constructor method, which the user may define. *Method overloading* can be used for specifying more than one constructor method for a class. The specific constructor that is called during object creation is chosen based on the argument types of the methods. To avoid confusion, operators can not be overloaded in Java.

Classes may declare ordinary *instance variables* of which every class instance (object) has its own copy, and *class variables* which are common to every instance of the class. Constants (immutable variables) are declared with the `final` keyword. Java also has *class methods*. They are invoked through a class name, rather than through an instance name. A class method is unrelated to the instances created from the class, so it can be called before any objects have been instantiated.

Object destruction is automatically handled by a garbage collector sometime after the last reference to a particular object has been disposed of. The garbage collector runs as a low-priority thread and does most of its work while nothing else is happening, for instance when the system waits for user responses. Therefore the garbage collection should not bother the application's activities too much, although it can never be as efficient as properly coded explicit allocation and deallocation of memory. Automatic memory management is especially suitable for writing language simulators and other language tool frameworks that involve interpretation of source code. Writing a memory manager for such applications can be quite a complicated task.

Before an object is removed from the memory, a special `finalize` method is called for that object. In a `finalize` method all resources (files, network connections, etc.) allocated for the particular object can be released. However, it must be noted that there is no guarantee about when the garbage collector will run, or in which order the finalizers of different objects will be called. Finalizers are not automatically chained according to the class hierarchy (like C++ destructors). Instead, the finalizer of an object's parent must be called explicitly if necessary [Mar97].

[6] Multiple inheritance not only leads into trouble when implementing a language, but is also usually a sign of *implementation inheritance* banned by the pattern community [GHJ95].

In Java, a class can be declared as `final`. This prevents the user from making unwanted extensions of that class. The final mechanism is a useful feature in framework construction. A framework is typically a collection of reusable code that an application programmer utilizes to create a working application. A user of a framework wants to get a full advantage of the framework without having to thoroughly explore the contents of the framework. There is always the risk that even with the help of exclusive documentation, the user of a framework applies it in a catastrophically wrong way. Final classes are one useful tool for preventing the user from extending the wrong classes and in that way exploiting the framework in a false manner.

An example of a Java program is presented in figure 3.1. It is an applet that prints two messages to a scrollable text region. The printing is handled via creating two different threads that both use the same text area for their output.

```
import java.applet.*;
import java.awt.*;

class TxtPrint {
  protected TextArea txt_;

  public TxtPrint(TextArea txt) {
    txt_ = txt;
  }

  public void print(String msg) {
    txt_.appendText(
      ”Echoing text: ” + msg);
  }
}

final class RunTxtPrint
  extends TxtPrint
  implements Runnable {

  protected String msg_;

  public RunTxtPrint(String msg,
    TextArea txt) {
      super(txt);
      msg_ = msg;
  }

 public void run() {
   print(msg_);
 }
}
```

```
public class MyApplet extends Applet {
  protected TextArea txt_;
  protected RunTxtPrint morning_;
  protected RunTxtPrint night_;

  public MyApplet() {
    txt_ = new TextArea(5, 50);
    morning_ = new RunTxtPrint(
      ”Good morning!\n”, txt_);
    night_ = new RunTxtPrint(
      ”Good night!\n”, txt_);
  }

  public void init() {
    add(txt_);
  }

  public void start() {
    new Thread(morning_).start();
    new Thread(night_).start();
  }
}
```

Figure 3.1: An example Java program

The class `TxtPrint` holds a reference to the text region being used (an object of the class `TextArea`). The class also includes the method `print` for creating the output. The class `RunTxtPrint` includes the code for the threads. It inherits (extends) the class `TxtPrint` and implements the standard

Java interface `Runnable`, so that it can be executed in a thread. The class is declared as `final`, so no extensions can be derived from it. The method `run` is called when a thread is started.

The class `MyApplet` extends the class `Applet` and overrides the methods `init` and `start`. The method `init` adds the text region (created in the constructor `MyApplet`) to the applet display. The method `start` creates two threads and passes different instances of the class `RunTxtPrint` as the constructor parameters. The threads are started after their creation. The first thread displays the message "`Echoing text: Good morning!`" and the other one the message "`Echoing text: Good night!`" to the text area.

Many of Java's properties — including its overall dynamic nature, portability, lightweight linking process, robust memory management, simple language structures, useful standard packages, possibility of coupling with HTML documents using the applet mechanism, and clear object-oriented features — suggest that Java is a suitable language for demanding software development, for instance for framework construction. However, Java has its disadvantages too. Frameworks are usually rather large and complex systems, and there are no actual experiences or proof of Java's scalability to real-world-scale applications[7]. Further more, some researchers have proposed that new language structures, for example *operator overloading* and *generic classes* (templates)[8], should be attached to Java before it can be considered as satisfactory [Mar97]. Also some programmers have seen Java's lack of multiple inheritance as a drawback. Performance is one thing to consider as well when writing frameworks, and it has to be said that applications written in Java usually require more computing capacity than those written in other languages. Because Java is an interpreted language, its execution speed is about 20 times slower than with fully compiled languages [Fla96].

---

[7] The large scale Java frameworks presented in chapter 5 are yet pretty much under construction.

[8] It can be argued that a template mechanism (such as in C++) offers a rather type-safe method for creating general containers. In Java, containers are implemented by handling instances of the ultimate superclass `Object`, which means that containers can hold any kind of objects.

4. An Overview of the Standard Java Packages

Java's package mechanism is a good way to collect closely related classes together in a framework. A framework could be a package included in an application built from the framework. The package concept is also an important mechanism for encapsulation and data hiding, and thereby another way to restrict and guide the use of a framework. Java supports an exclusive set of visibility modifiers for classes and their features. Using these modifiers the framework designer can control how class properties can be invoked (differently) from (1) a non-subclass in the same package, (2) a subclass in the same package, (3) a non-subclass in a different package and (4) a subclass in a different package.

Because classes in Java can be loaded dynamically into the environment, the naming conventions have been carefully designed to avoid conflicts. There are no global variables in Java. Every variable and method is declared inside a class, and each class belongs to a package. Thus, every feature (variable or method) has a unique identifier, which consists of a package name, a class name and the feature name. For example the instance variable `elementCount` in the class `Vector` belonging to the package `java.util` can be referred by `java.util.Vector.elementCount`[9]. The designers of the Java language have proposed an Internet-wide naming scheme for packages, where the first qualifier in the name path of a package would be the domain name of the organization connected to the Internet. The relationships between Java source code files, byte code files, packages and directories are presented in figure 4.1.

The package names conform to the directory structure formed by the Java byte code class files. In the source code format a Java program consists of one or more Java source files. A source file (with the extension `.java`) can include several class definitions. One of these classes can be declared as public (visible outside the package), and this class must have the same name as the source file (without extension). Every class definition is compiled into its own byte code file (with the extension `.class`)[10]. The byte code files must be located in the directories named after the packages containing the particular classes. For example, all the classes in a package `MyClasses.Containers.OrdContainers` have to be located in the directory `MyClasses/Containers/OrdContainers`[11]. The source files can exist in an arbitrary directory. Classes in different source files can belong to the same package. Statement `package` in the beginning of the source file defines the package for the classes in that file.

There are eight standard packages included in a standard Java environment (JDK 1.0): `java.applet`, `java.awt`, `java.awt.image`, `java.awt.peer`, `java.io`, `java.lang`, `java.net` and

---

[9] The `import` statement is used to avoid the need for typing full path names to classes. For example, after the statement `import java.util.*`, the class `java.util.Vector` can be referred simply as `Vector`.

[10] One of these classes defines the method `main`, which is the starting point for interpretation. This class is passed as an argument to the Java interpreter. The execution of a Java program ends, when the `main` method has been interpreted to its finish or, if there are threads still alive at this point, when the last thread has terminated.

[11] The Java interpreter looks for user-defined classes relative to the environment variable `CLASSPATH`.

`java.util`. Java's packages provide features (such as networking mentioned earlier) that would be regarded as frameworks in some other languages. When using Java a framework developer does not have to build these useful features from scratch.

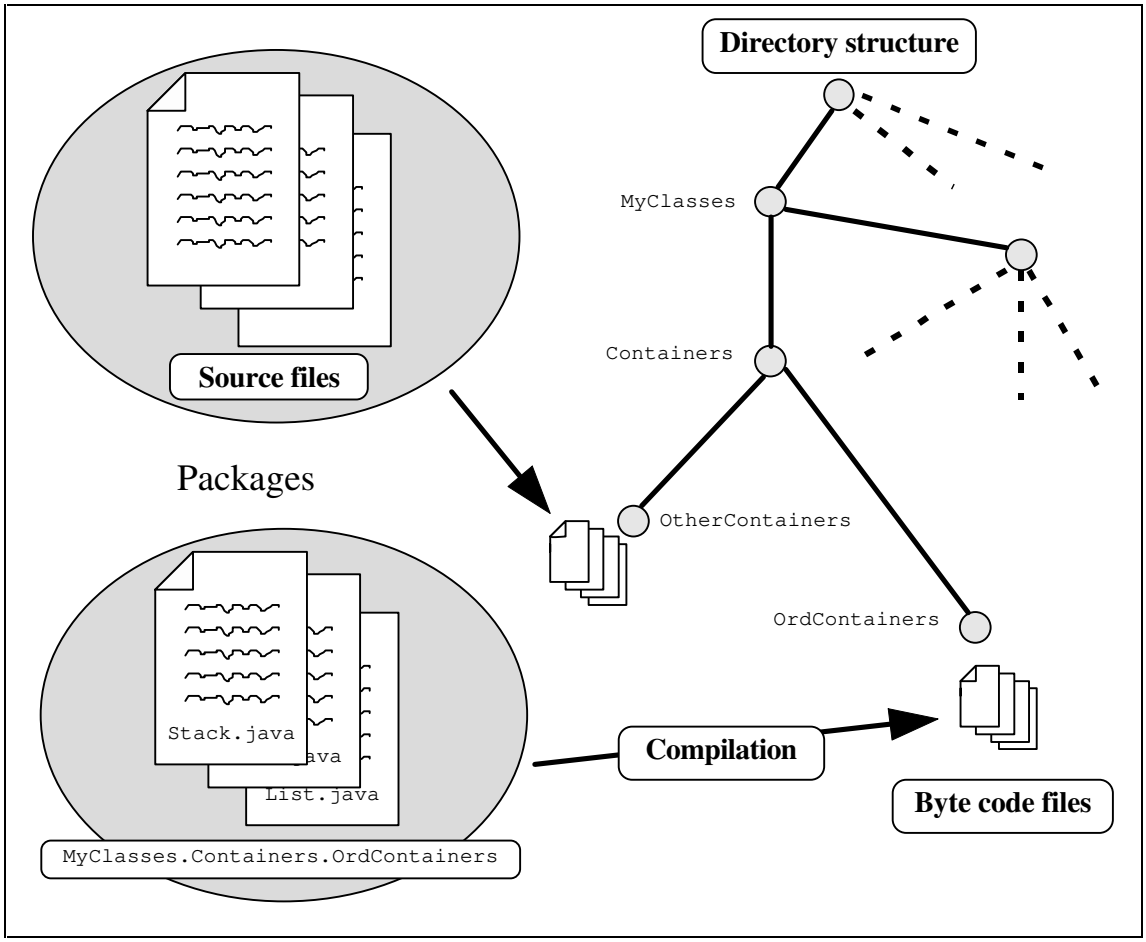

Figure 4.1: Source code files, byte code files, packages, and directories

The `java.applet` package contains only one class: `Applet`. An applet can be implemented simply by inheriting this class and overriding some critical methods (for instance `init`, `start`, and `stop`). These methods are called automatically by the Web browser being used. The `java.applet` package itself is a good example of an object-oriented white-box framework: the control of execution lies inside the framework, and the user just inherits proper class(es) and overrides critical hot spots to create a working application.

The `java.awt` package defines the Abstract Windowing Toolkit (AWT). AWT consists of graphics tools, a set of graphical user interface components and a manager for their automatic layout. AWT enables the development of platform independent GUIs. The design of AWT utilizes several design patterns

[GaJ96]. The `java.awt.image` package includes useful classes for implementing image processing facilities.

The `java.awt.peer` package consists entirely of interface declarations. All the interfaces correspond to some GUI components. The interfaces define the methods that the GUI components of a specific platform must support. Porting the Java GUI components to a new platform is a matter of implementing each of the methods in the interfaces on top of the native GUI components on the particular platform. The `Toolkit` object in the `java.awt` package collects together the GUI component implementations of a specific platform. `Toolkit` contains methods for creating instances of these components. The convention used here is similar to the *Abstract factory* pattern [GHJ95].

The `java.io` package contains classes for input and output functions. The main structures in this package are `InputStream` and `OutputStream`. These are abstract classes defining methods for reading and writing data. Their subclasses (for example `FileInputStream` and `FileOutputStream`) define stream based connections to different kinds of sources.

The `java.lang` package provides the classes that are most central to the Java language. It includes, for instance, the classes `Object`, `String` (an immutable character container), `StringBuffer` (a mutable buffer of characters), `Runnable`, `Thread`[12], and `Throwable` (root class for exception classes). Also Java's reflexive nature becomes clearly visible in this package. The package includes classes such as `Compiler` (which provides an interface to the Java compiler system) and `Class` (which is a representation of a Java class, enabling the dynamic loading and instantiation of classes).

The `java.net` package defines classes for networking applications. Class `URL` allows a Java application to open access to remote files on the Internet. The class represents the Internet Uniform Resource Locator (URL address), and it has a simple interface: the object referred to by the URL can be down-loaded, or streams may be opened to read from or write to the object. Pure stream-based network connections can be achieved with the `Socket` class. It allows a connection to a specific port on a specific host and reading and writing data using the stream classes mentioned earlier. The class `ServerSocket` provides a simple way to implement a server that offers connections to clients.

The `java.util` package provides useful utilities — for instance container classes (`Hashtable`, `Vector`, `Stack`) and an interface for iterating through containers (`Enumeration`). In addition, the package contains a parser class `StringTokenizer` which converts strings into tokens. The classes

[12] An object that needs to be executed in a separate thread just implements the interface `Runnable`. A new thread is created by instantiating an object of the class `Thread`. The task that implements the `Runnable` interface is passed as a constructor parameter. Threads can be controlled with the `start`, `isAlive`, `stop`, `suspend`, `sleep`, `join` and `interrupt` methods [Lea96]. Java supports Hoare's *monitor/ condition variable paradigm* [Hoa74] for the synchronization of threads.

`Observer` and `Observable` provide yet another small framework within the Java language. These classes make up the infrastructure for implementing the *Observer* pattern [GHJ95] in Java. The class `Observable` defines the methods for objects whose states need to be observed. The class allows adding and deleting `Observer` objects to and from a list of "observer" objects that need to be informed when the state of the "observed" object changes. The `Observer` class defines the `update` method, which `Observable` uses when notifying `Observers`.

Java is a language evolving at a rapid speed. The new version of Java JDK (1.1) [Sun97b] is already available. Some of the new packages are especially interesting from a framework developer's point of view. These features include, for example, *JavaBeans* [Sun96a] (a component model that provides mechanisms that allow objects from different vendors to interact with each other) and *Remote Method Invocation (RMI)* [Sun97a] (enabling the invocation of a remote Java object's methods).

Java's special language features and standard packages propose certain kinds of frameworks. Especially World Wide Web frameworks and distribution on the whole are popular application domains for Java. It has to be noted, however, that the language is still very young, and it is natural that the first Java frameworks to appear utilize the special features of Java. When the language stabilizes, there will probably be frameworks for all kinds of application domains. Though, Java's interpretative nature and automatic garbage collection may restrict its use in some real-time frameworks which require ultimate speed and accurate timing.

The application area of applets is naturally more restricted than that of the ordinary applications. That is because the Java specification intentionally limits what applets are entitled to do, so that distributing viruses or other hostile programs as Java applets would not be possible. A Java applet is basically a class (extended from the standard class `Applet`) that is loaded and run by an already running application such as a Web browser. Applets (loaded across the network) must be treated as untrusted. To assure that applets don't perform any unwanted actions in the computer that they are loaded to, they must be forced to run in a very limited environment. Different Web browsers may restrict the behavior of applets in slightly different ways, or they may even let the user specify some of the restrictions. Generally, the following are not allowed for applets [Fla96][13]:

- reading, writing or deleting local files
- reading, renaming or creating local directories
- creating network connections to other hosts besides the one that the applet was loaded from
- listening or accepting network connections on any port of the local system

[13] When an applet is loaded from the local disk, the Web browser may relax some of the restrictions.

- creating top-level windows without warning labels (this is intended to prevent Trojan Horses)
- invoking local applications
- quitting the Java interpreter
- creating or manipulating threads that are not part of the same `ThreadGroup` as the applet
- specifying classes that control Java's network handling
- accessing or loading classes in any package other than the standard Java API and
- defining classes that are part of the packages on the local system.

The security restrictions are implemented by the `java.lang.SecurityManager` class. Different Web browsers may define their own policies by subclassing this class.

5. Java Frameworks

There are already many application frameworks written in Java. In this chapter some of these are introduced. The frameworks are loosely classified into networking, business, and miscellaneous frameworks on the basis of their application domain. Many of the frameworks are still at design phase or otherwise under construction.

5.1 Networking Frameworks

Java Web Server (Jeeves)

Java Web Server (Jeeves) [Sun96c] is JavaSoft's framework for creating network servers. Jeeves consists of a *base server framework* and a *HTTP server* built using the base framework. The base server framework introduces general classes for implementing servers that may fork multiple connection threads for serving multiple clients.

The basic operation of the framework proceeds in the following manner. The server starts by creating the *worker threads* that handle the connections. The main control of the framework is handled by a *server acceptor thread*, which is continuously waiting for connection requests. When a request arrives, the acceptor picks up one of the server handler threads from the worker pool to manage the connection to the client. The number of available server handlers (the size of the worker pool) is dynamically configurable. The server acceptor thread ensures that new connections are not accepted unless there are free handler threads in the pool. If the pool is empty and the limit on the number of threads is not exceeded, a new handler thread is forked. On the other hand, if a thread has been waiting a very long time for a client and there are more than the minimum amount of handler threads available, the particular thread will expire.

An application developer can extend the basic framework for individual purposes. The Java Web Server package comes with one extension: the HTTP server. The elementary architecture of the HTTP server is shown in figure 5.1. The HTTP server uses the base framework for implementing the service for incoming HTTP requests. Requests are dispatched by the *acceptor handler* to *HTTP server handler threads*. The handler thread does the authorization and the name translation, based on the server administration settings. After that the handler delegates the request to a *servlet*[14] available for the purpose.

The servlets of the HTTP server are created, when the server starts up. The HTTP server provides some basic servlets (for instance *FileServlet* and *CgiServlet*) for standard HTTP functionality. The framework user can create her own additional servlets. When implementing new servlets, one must notice

[14] Servlets are applet-like Java objects (represented as byte code) that can be loaded dynamically from an arbitrary source on the net. Similarly to applets, servlets are identified by an URL address, but they do not have any graphical representation [Sun96g].

that several handler threads may use the same servlet concurrently. Though, servlets must employ Java's synchronization features for their critical methods. The servlets may be either local (trusted) servlets or (untrusted) servlets loaded from the network. For security reasons, an untrusted servlet is run in a thread located in a separate thread group. For every untrusted servlet there is one local *StubServlet* used to communicate with the untrusted *NetServlet*.

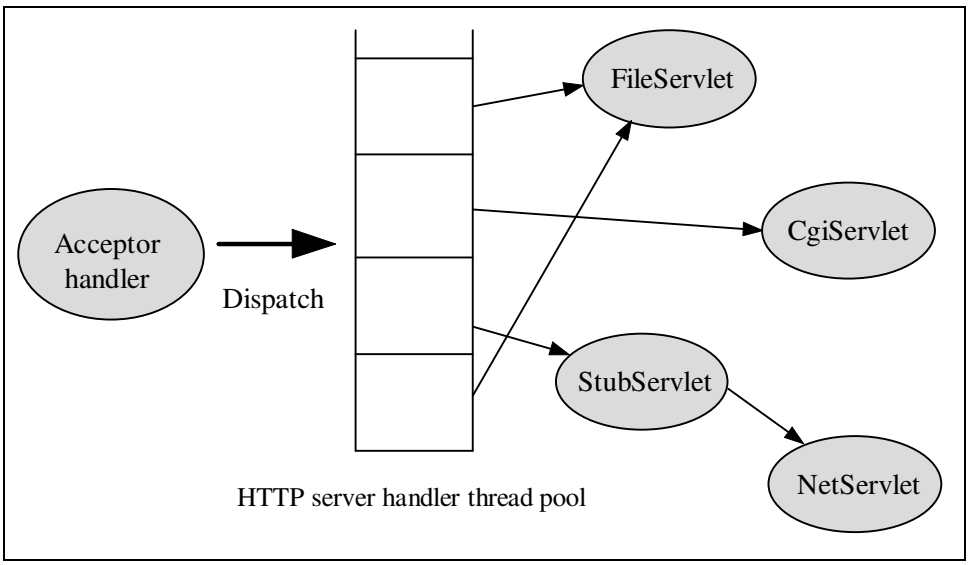

Figure 5.1: The basic architecture of the HTTP Web Server [Sun96c]

Java Telephony API (JTAPI)

JavaSoft's Java Telephony API [Sun96b] provides a reusable set of objects that forms a framework for telephone call control applications. The framework can be easily extended to create custom applications. The skeleton of the framework is simple, but powerful enough to enable the extension to real and rather complex systems. JTAPI is designed to work as a direct service provider interface to applications or as a Java interface to existing telephony APIs (such as SunXTL, TSAPI and TAPI).

JTAPI can be applied in two different architectural configurations. First, it can be used in a setting where a centralized server holds direct access to the available telephony resources. In this context, an application (or an applet) based on JTAPI runs on a remote workstation and uses telephony resources via accessing the centralized telephony server. Communication between the server and the JTAPI application is handled with a remote access mechanism — for example Java Remote Method Invocation (RMI) [Sun97a]. The second possible configuration for JTAPI applications is that the application is running directly on a computer that has its own telephony resources.

JTAPI consists of the Core package (the basic framework) and several extension packages. The Core defines elementary telephony functionality: placing, answering, and dropping calls. The basic call model defined by the Core package is presented in figure 5.2.

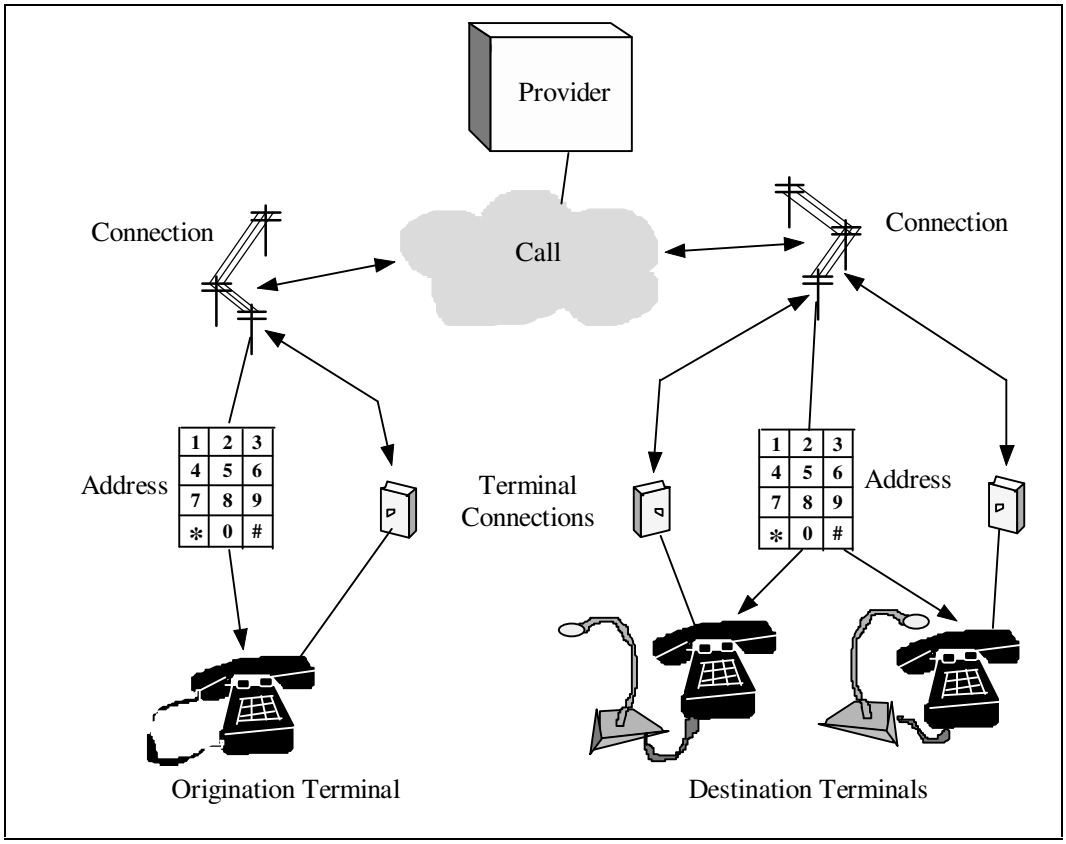

Figure 5.2: The basic objects of the Java Telephony API [Sun96b]

The *Provider* object is an abstraction of telephony service provider software. It hides the service-specific aspects and enables Java applications to interact with the telephony system in a device-independent manner. The *Call* object represents a telephone call. Two or more *Connection* objects may be attached to it during a telephone call. Connection objects model the communication link between Call objects and *Address* objects. Address objects represent telephone numbers and *Terminal* objects physical telephone devices connected to them. Terminal objects may share a common telephone number (Address object), but each Terminal object has a distinct *TerminalConnection* object representing the relationship to the corresponding Call object. A TerminalConnection object signals a Terminal when there is an incoming call and monitors the activity of the Terminal during the call.

The extension packages follow the design of the Core. All applications implemented with JTAPI utilize the Core package and possibly one or more extensions. For example, an applet with simple telephony

capabilities needs only the Core package, and more complex applications may utilize the extension packages and even broaden their functionality for custom purposes.

The standard extension packages of JTAPI include, for instance, the *Call Control Package* (introducing additional call control functionality — for example call transfer and conference calls), the *Call Center Management Package* (including Automated Call Distribution and call routing especially for large distributed call centers), the *Terminal Set Management Package* (managing the controlling of physical features of a telephone set — for instance speaker volume), and the *Mobile Phones Package* (providing support for cellular telephones).

The Caltech Infosphere Infrastructure

The Caltech Infosphere is a project under way in the Computer Science Department of California Institute of Technology. One of the project's goals is to provide a framework (Infrastructure [Cal96]) that supports peer-to-peer communication among processes spread across a network. The framework is designed to support the development of verifiably correct distributed applications. It is not intended to be used in applications with very strict performance requirements and possibly disastrous consequences upon a failure. Instead the framework is designed for the fast growing class of collaborative Web applications, for example distributed calendar programs. The implementation uses Java's socket classes and thread primitives.

The framework introduces two abstract compositional units which these collaborative applications consist of: *dapplet* and *session*. Dapplet is a multithreaded communicating object that processes messages arriving at its input ports. Messages possibly cause state changes in a dapplet and new messages to be sent to the output ports of the dapplet. Sessions are groups of composed dapplets. They are temporary by nature and form networks of dapplets to carry out a certain task, for example arranging a meeting time for a group of people in a calendar application.

Each dapplet has an Internet address (IP address and port id) associated with it. A session forms an application, which is actually a network of dapplets. A session consists of many different types of dapplets. Associated with each session is an *initiator dapplet*, which is responsible for linking dapplets together. The initiator dapplet sends a request to the component dapplets to link themselves up to form a session. A dapplet may accept or reject the request (based upon some access control list).

The framework provides the following basic components. Each dapplet has a set of *message queues* (inboxes and outboxes) for sending and receiving messages (subclasses of Message). Each inbox has a global address (the address of the dapplet) and a local reference within the dapplet. Each outbox may have several inboxes associated with it. Each association forms a *message channel*. A message put to an outbox is delivered to every inbox bound to it. An application can associate new inboxes to an outbox or remove

existing ones, send messages to an outbox, and ask for all the addresses of the inboxes bound to it. For an inbox an application can invoke a blocking receive operation or just check for its contents.

The project has identified the following requirements for the framework: it has to cope with wide variations in session duration, be able to save its state across multiple temporary sessions (for example in calendar applications), provide an appropriate library of operating system level services (for instance checkpointing, termination detection, and multiway synchronization), and cope with network problems (variety of delays, faults, and undelivered messages). The framework also tries to capture the different patterns occurring in a collaboration behavior of concurrent objects. The goal is that a framework user can write code like she would write for a sequential application and needs not to worry about synchronization or other difficult aspects of distributed computing.

The SAIC Agent Framework

The SAIC Agent Framework [SAI96] is a framework for the development of distributed co-operative agents for different kinds of Internet searches. An example could be a "shopper agent", whose target is to find the lowest cost for a given product. Designing this sort of agent is not possible with the usual keyword- or phrase-based searches that for example *Lycos* and *AltaVista* accomplish. Agents built with this framework are able to navigate and operate in a diverse set of operating systems and windowing environments. Agents interact with each other via the *Facilitator agent*. It is the central authority on a LAN that knows how to communicate with all the agents in the area. A Facilitator contains data on each agent such as the agent's name, address, and the type of information it is interested in receiving. Each host within the agent network includes an *object server* that has the ability to convert objects (agents) to a hardware-independent format and pass them to other servers. Agents are persistent, meaning that they can be written to disk for later use. *Little Brother* is one of the example applications derived from the framework. It asks remote computers for their status and performs some maintenance tasks.

The Collaborative Application Framework

The Collaborative Application Framework [AKN96] is a rather pure black-box framework for creating *conference applications*, in which several people can virtually run the same application simultaneously. The elementary idea of the framework is simple: each participant runs her own copy of the application and the activity of each user is multicasted to all the other participants.

The framework has two major components: *Collaboration Client* and *Collaboration Server*. Collaboration Client intercepts all input events in the user application[15] and sends them to the Collaboration

[15] This is accomplished via replacing the standard Java GUI (AWT) with a version where all the event handlers for graphical widgets have been overridden.

Server. Collaboration Server is aware of all participants in the session and echoes all the events to all the Collaboration Clients. When a Collaboration Client receives an event from the server, it sends the event to its application program. As a consequence, it seems like the user of the application had caused the event herself. The communication between Collaboration Clients and the Collaboration Server is handled using Java's `Socket` class.

A person running a conference application sees two different graphical interfaces: the actual interface of the application and the *Conference Control Interface* provided by the framework. Via the Conference Control Interface the user can ask for a token that allows to echo her events to other users. Only one user at a time can hold the token.

Two rather similar frameworks to the Collaborative Application Framework are *Habanero* [NCS97] and the *Session Framework* [Str96]. Habanero modifies single-user applications into multi-user shared applications. Applications become collaborative by sharing the same state, and by sharing *Actions*, which are delivered to each collaborative client to change its state. The Session Framework allows multiple independent objects to communicate using a generalized *publish/subscribe protocol*. The framework can be used, for instance, to connect multiple applets running on the same Web page.

5.2 Business Frameworks

San Francisco

San Francisco [DHA96] is IBM's code name for a project that aims at creating "shareable frameworks" for software development in Java. The basic idea is to provide a common foundation for developing business applications. These include financial systems, manufacturing planning, and logistics. The first frameworks to be developed are *General Ledger*, *Accounts Receivable*, *Order Processing*, and *Warehouse Management*.

San Francisco consists of four layers (see figure 5.3). The lowest layer is called the *Base*. It interacts with the operating system through the Java Virtual Machine. The key function of the Base layer is to create interfaces to the operating system, other applications, and the network. These interfaces can be utilized directly by application developers, but they also form a foundation for the upper layers. The Base layer includes functions such as finding objects, keeping track of their names, controlling access to them, resolving conflicts, and security administration. The Base layer also defines the *Object Model Classes*. These are low-level objects from which higher level objects can be created.

The next layer above the Base is called the *Common Business Objects*. It consists of a large number of objects that perform functions commonly needed within a business application. These include payment terms, calendar, discounts, currency transformations, and handling of measurement units. Common

Business Objects are the building blocks from which application developers can select and create business applications. Common Business Objects can be extended to perform custom functions.

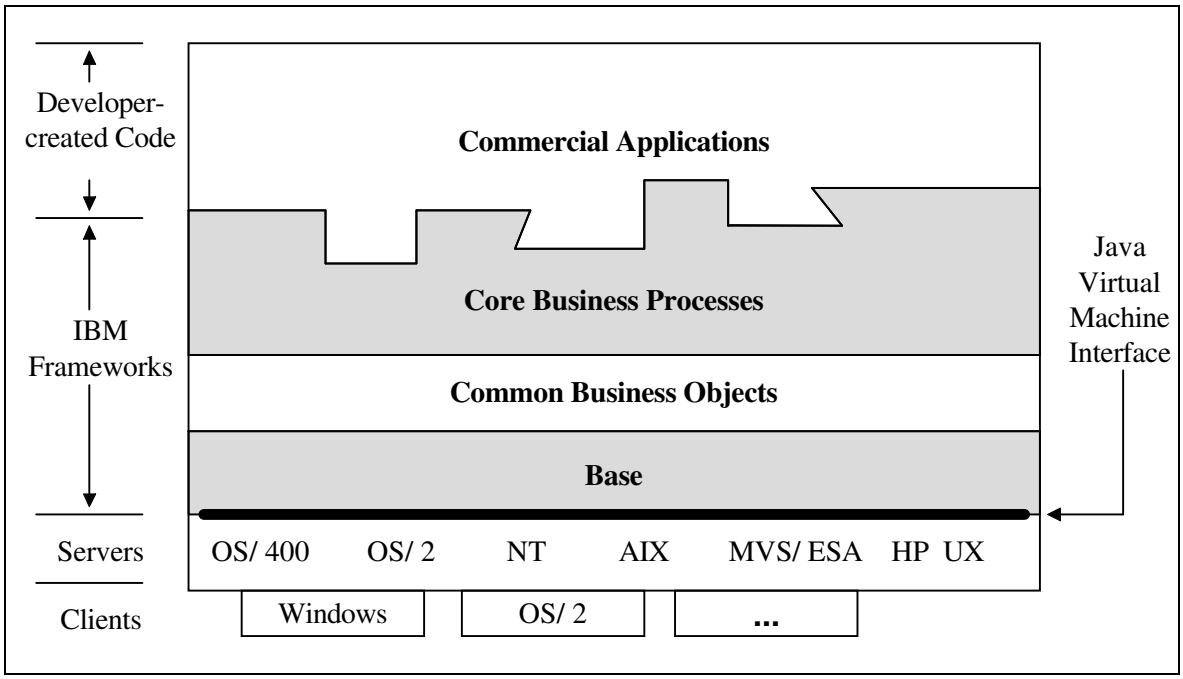

Figure 5.3: The layers of San Francisco

The third layer of San Francisco — *Core Business Processes* — is the one that actually consists of application frameworks. Each framework is built for one specific type of application. The final layer is intended to be created by IBM's customers, who buy and extend the San Francisco frameworks suitable for their needs. The applications in the highest layer can also be frameworks capturing the knowledge of one particular software developer on some application domain.

Java Electronic Commerce Framework

The Java Electronic Commerce Framework [Sun96h] is a framework under construction. It is designed for electronic commerce (payment and financial) applications. Its purpose is to make purchasing as a "seamless part of browsing the Internet", allowing applets to charge for execution and content delivery and create an open platform for purchasing, banking, and finance. The initial version of the framework includes a common graphical user interface, a secure encrypted database, a capability mechanism providing fine-grained access control, means for cryptography, and an infrastructure for purchasing.

5.3 Miscellaneous Frameworks

TurboSprite for Java

TurboSprite for Java [SCG96] is a small framework for creating animation with *sprite* objects. Sprite is a term used in game programming meaning a graphical object moving around the screen. TurboSprite handles the managing, moving, and rendering of the sprites. The framework has three major components: *SpriteSurface*, *SpriteEngine*, and *Sprite*. SpriteSurface represents the surface into which the Sprites are drawn, SpriteEngine is responsible for moving and rendering sprites, and Sprite is an abstract base class which must be extended for specific sprites. An extension must provide an implementation for the actual drawing method. TurboSprite is a good example of Java's adequate speed for animation purposes.

Demeter/Java

Demeter/Java [LiO96] is a framework for writing programs in terms of *traversal strategies*[16], thereby helping the development of easily maintained programs. A traversal strategy describes how a navigation through a group of related objects is done to accomplish a certain task. Traversal strategies are implemented using a small descriptive language rather than hard-coding traversals into applications. This minimizes needs for updates when the structure of the manipulated objects changes. Traversal strategies are described in Demeter/Java with a small special language integrated in Java.

In Demeter/Java the idea of describing traversal strategies is combined with the *Visitor* pattern [GHJ95]. Using the Visitor pattern, the actual tampering of objects is separated from the manipulated objects into Visitors. Different Visitors manipulate objects in different ways. The benefit of the Visitor pattern is that when a new kind of manipulation needs to be introduced, the manipulated objects themselves need not to be changed.

Security Policy Representation Framework

Security Policy Representation Framework (SPRF) [Fra96] is a collection of classes for representing security policy statements as data structures. Besides just representing the security policy, the framework also provides methods for evaluating different policy decisions. At the lowest level the framework may be used to represent mathematical concepts like sets and mappings. These structures form a foundation for more complex concepts — access control lists, labels, and rules — used in security policy definitions. At the highest level the framework provides means for representing complete policies, including types of discretionary and non-discretionary access control.

---

[16] This type of coding is called *adaptive programming* (AP) [Lie96].

In a traditional operating system the security policy functions are hard-coded into the system using a special language. However, different organizations might have very different requirements for a security policy. The power of this framework is that it may be easily extended to create configurable security policy representation functions and an interpretable representation of a security policy. The framework can be used, for example, as a component of trusted operating systems.

6. Java Software Development Environments

Framework development requires proper tools. One of the main goals of the FRED project is to develop a set of tools for designing, implementing, and using frameworks written in Java. At the moment a framework tool of any kind is rare. Some of the existing tools are introduced in [Vil97]. Although there are no specialized framework tools available for Java, there is already quite a number of ordinary Java software development environments implemented.

JavaSoft distributes a minimal environment for developing Java applications and applets. This package — *The Java Developer's Kit (JDK) 1.02* [Sun96d][17] — includes a Java compiler, Java Appletviewer (for executing applets), a simple command line debugger and a document generator. The required Java class libraries (and the source code) are distributed with the package. JDK includes a variety of sample Java applets and a set of documentation for the Java class libraries. The provided documentation does not include any kind of search facilities. Application development is quite primitive with JDK. There is no project management mechanisms for grouping source files, no syntax highlighting Java source code editor, and no visual user interface building tools. JDK runs under Solaris, MS-Windows (NT/ 95), Linux, OS/2 and Macintosh.

Microsoft's *Visual J++* [Mic96] is a component environment for the Microsoft Developer Studio. It includes similar tools as Visual C++: for instance a source file project manager, source code and resource editors, and a debugger. The project manager can be used to navigate and browse the source code and to add variables and methods to classes. Navigation proceeds over the class hierarchy or the file structure. The resource editor enables visual construction of graphical user interfaces. *Java ResourceWizard* converts the Windows resources into portable Java code. The debugger contains the conventional functionality: a call-stack window, breakpoints, stepping through code, and multithreaded debugging. Visual J++ enables a programmer to use ActiveX (and OLE), but these features shall be avoided to retain platform neutrality. J++ contains automated support (*AppWizard*) for constructing simple application cores. AppWizard allows customization choices for multithreading, animation, mouse-event handlers, and applet parameters. Unlike JDK, Visual J++ includes a Just In Time compiler for speeding up the execution of Java programs. Visual J++ runs only under Windows 95 and Windows NT.

Symantec's *Café* [Sym96] is a standalone set of Java software tools based on the company's C++ development environment. Café is available for Windows 95, Windows NT, and Macintosh. Like J++, Café includes tools (*ProjectExpress*, *AppExpress*) for generating simple application and applet skeletons, and it allows visual construction of a GUI. Café utilizes a Windows resource editor to lay out the user interface widgets in Java applications. The user can drag and drop GUI widgets onto a form and attach event handlers to them. Café generates the required Java code from the created Windows resource files. The

[17] Improved versions of the same tools are introduced in JDK 1.1 [Sun97b].

environment provides syntax highlighting for Java source files during editing, and it allows the navigation through the source code by following the file structure or the class hierarchy. Navigation is easy: the user can view any Java declaration within user application or the standard Java packages via clicking the corresponding name in the source code.

The byte code compiler in Café is up to 13 times faster than the standard Sun Java Compiler [Sym96], and the environment also includes an optimized Java runtime system and an efficient Just In Time compiler. The execution speed of Java programs with Symantec's runtime support is faster than with J++ [Sym96]. Café integrates also the JDK tools into the development environment, offering a graphical support for the options and parameters of the Java compiler, an interpreter, and a debugger. Café's own debugger enables static and conditional breakpoints, call-stack tracing, watch expression usage, and execution of source code by stepping through every line or from procedure to procedure. It also allows the debugging of multithreaded Java programs.

SunSoft's *Java WorkShop* [Sun96e] consists of Java development tools for the creation of Java applets and applications. Java WorkShop is written in Java, and it runs on Solaris, Windows 95, and Windows NT 3.51. Java WorkShop includes the following tools. *Visual Java* can be used for constructing graphical interfaces. It enables, for example, the previewing of an interface's behavior without having to generate or compile Java code. *Project Manager* organizes source files, and *Source Editor* enables the writing and viewing of Java source code. The editor also allows an integration to a number of file version control systems including PVCS, SCCS and RCS. *Source Browser* can be used for viewing source code structure as a hypertext class hierarchy. The debugger of the environment is capable of examining multithreaded Java code. Future versions of Java WorkShop will introduce support for JavaBeans [Sun96a] components, an editor with syntax highlighting, remote debugging facilities, and the integration of Sun's JIT technology and fast Java byte code compilers. The current set of tools lacks the efficiency provided, for instance, by the Café environment. Java WorkShop's intuitive Web browser interface and its portability to both MS-Windows and Solaris (UNIX) are the greatest benefits of the environment.

Silicon Graphics has published a Java development environment — *COSMO 2.0* [SiG96a] — for the IRIX[18] operating system. COSMO includes a source editor, a project manager, and a debugger. It also introduces a visual user interface building tool (*Visual Builder*) that enables the dragging and dropping of GUI widgets as well as automatic generation of Java source code. Using the Visual Builder, developers can test the application interface without compiling. Once the design is finished, it can be transformed into Java. The COSMO environment also includes a performance optimizing translator that speeds run-time performance by an order of magnitude over interpreted code [SiG96a].

[18] IRIX is an extension of the UNIX operating system [SiG96b].

At the moment Borland has no special Java development environment, but a set of Java tools has been integrated to the *Borland C++ Development Suite 5.0* [Bor96a]. These tools are quite immature and not very reliable [Lam96]. Development Suite's Java tools include a project file manager, a syntax highlighting editor, a Just In Time compiler (*AppAccelerator*), and a debugger. The debugger is written in Java and it enables breakpoints, watch expressions, and a call-stack tracing. The debugger is slow and prone to crashing. The Borland C++ Development Suite offers no support for visual development of a GUI: the coordinates of various user-interface widgets must be calculated by hand and then hard-coded into the application.

JDK is a basic environment for Java software development. It offers a cheap possibility to get to know the language, but it does not include the necessary tools for developing large-scale software. Visual J++ and Symantec Café are rather similar environments. They both introduce a rich set of tools helping the software construction. The tools in the Café environment are somewhat more efficient than the ones in Visual J++. In Visual J++, the MS-Windows emphasis (for example ActiveX support) can be considered as against the general portability aims of Java. Java WorkShop adapts Java's general networking ideology in its WWW-based user interface. Java WorkShop is written in Java, which makes it portable to both Windows and UNIX. The disadvantage of Java WorkShop is its inefficiency. Borland's Java tool can be considered as a half-finished C++ extension.

New software development environments for Java are published continuously. Java tools to be introduced in the near future are, for example, Borland's *JBuilder* [Bor96b], IBM's *Visual Age for Java* [IBM96] and Sybase Inc. 's *Jato* [Syb96].

7. Conclusion

In this report we have evaluated Java's applicability for writing object-oriented frameworks, including a short introduction to the language and to the run-time environment. A deeper look was taken at the language's object-oriented features and the contents of the standard packages. Java was characterized as a robust high-level language, from which certain error-prone features commonly found in programming languages have been left out. Especially Java's sophisticated memory management and exception handling mechanisms were found to be useful for framework construction. We also noted that the design of Java's object-oriented features suits perfectly for pattern-style programming. Java's interpretative nature, on its behalf, makes the language very dynamic, and thereby software development in Java becomes similar as framework construction in general: because there is no time-consuming link process in Java, iterative software development methods can be used comfortably.

Although Java was noted not to restrict any of its possible application domains, Java's standard packages introduce a strong support for certain software development areas. Especially Java's seamless integration to the Internet offers good capabilities for writing frameworks utilizing the net and the distributed computing paradigm.

After evaluating the Java language itself, some of the existing frameworks written in Java were introduced. These frameworks seem to emphasize the use of Java's networking features, but there are also frameworks available for other purposes. The examined frameworks range from small frameworks for special purposes (for example TurboSprite) to rather complex ones like Java Web Server, Java Telephony API, and San Francisco which is going to introduce a whole family of frameworks related to business computing.

The last chapter in this report introduced the currently available software development environments for Java. In general, the available tools are not as mature as those for example for C++. Symantec's Café stood out because of its efficiency, and on the other end Borland's Java tools integrated with their C++ environment did not gain much credits. Sun's Java WorkShop can be considered the most suitable environment for developing framework tools in the FRED project, because of Java WorkShop's portability to both Windows 95/ NT and Solaris UNIX.

Java's advanced support for WWW programming offers tempting possibilities to write tools for framework documentation and for other tasks with interfaces similar to WWW browsers. Sun's Java WorkShop is an excellent example of a Java tool with such an interface. Java's networking capabilities also allow the implementation of distributed framework tools, where a number of people interact with each other and possibly work concurrently to achieve a shared goal.

References

AKN96 Abdel-Wahab H., Kvande B., Nanjangud S., Kim O., Favreau J., Using Java for Multimedia Collaborative Applications. In: *Proc. of the 3rd International Workshop on Protocols for Multimedia Systems (PROMS'96)*, Madrid, October 1996. Internet: `http:// www.dit.upm.es/ %7eproms96/ paperszip.html`, 1995.

ArG96 Arnold K., Gosling J., *The Java Programming Language*. Addison-Wesley, 1996.

Bau76 Bauer F., Historical Remarks on Compiler Construction. In: *Compiler Construction — An Advanced Course* (Bauer F., Eichel J., eds.). Springer-Verlag, 1976, 603-614.

BeC95 Berners-Lee T., Connolly D., *Hypertext Markup Language - 2.0*. Internet: `http:// www.psinet.net.au/ ~trevor/ html/ html-spec_toc.html`, 1995.

Bor96a Borland International Inc., *C++ Development Suite 5.0*. Internet: `http:// www.borland.com/ borlandcpp/`, 1996.

Bor96b Borland International Inc., *JBuilder*. Internet: `http:// www.borland.com/ jbuilder/`, 1996.

Cal96 California Institute of Technology (Caltech), *Caltech Infosphere Infrastructure*. Internet: `http:// www.infospheres.caltech.edu/`, 1996.

Cop92 Coplien J., *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

Deu89 Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: *Software Reusability Vol. II* (Biggerstaff T., and Perlis A., eds.), ACM Press 1989, 57-71.

DHA96 D. H. Andrews Group, *IBM's San Francisco Project — Java Frameworks for Application Development*. Internet: `http:// www.dhagroup.com/ r_sanfra.html`, 1996.

DuJ96 Durham A., Johnson R., A Framework for Run-time Systems and Its Visual Programming Language. In: *Proc. OOPSLA '96*, ACM SIGPLAN Notices 31, 10, 1996, 406-420.

Fla96 Flanagan D., *Java in a Nutshell — A Desktop Quick Reference for Java Programmers*. O'Reilly & Associates, Inc., 1996.

FoO95 Foote B., Opdyke W., Lifecycle and Refactoring Patterns That Support Evolution and Reuse. In: *Pattern Languages of Program Design* (Coplien J., Schmidt D., eds.), Addison-Wesley, 1995, 239-258.

Fra96 Fraser T., *A Java Framework for Security Policy Representation*. University of Illinois at Urbana-Champaign, Internet: `http:// choices.cs.uiuc.edu/ Security/ SPRF/ sprf.html`, 1996.

GaJ96 Gamma E., Johnson R., *Java AWT*. Pattern Stories Web, Internet: `http:// st-www.cs.uiuc.edu/ cgi-bin/ wikic/ wikic?JavaAWT`, 1996.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Hau97 Hautamäki J., A Survey of Frameworks. Report, University of Tampere, Department of Computer Science, 1997.

HJE95 Hüni H., Johnson R., Engel R., A Framework for Network Protocol Software. In: *Proc. OOPSLA '95, ACM SIGPLAN Notices* 30, 10, 1995, 358-369.

Hoa74 Hoare C., Monitors: An Operating System Structuring Concept. *Communications of the ACM* 17, 10, 1974, 549-557.

IBM96 IBM Corporation, *Visual Age for Java*. Internet: `http:// www.software.ibm.com/ ad/ vajava/`, 1996.

JaS97 Jain P., Schmidt D., Experiences Converting a C++ Communication Software Framework to Java. *C++ Report* 9, 1, 1997, 50-66.

KoM95 Koskimies K., Mössenböck H., Designing a Framework by Stepwise Generalization. In: *Proc. of 5th European Software Engineering Conference (ESEC '95)*, Sitges, Spain. Lecture Notes in Computer Science 989, Springer-Verlag, 1995, 479-498.

Lam96 Lam J., Java and Java Tools: Waking Up the Web. *PC Magazine* 15, 11, 1996, 221-230.

Lea96 Lea D., *Concurrent Programming in Java — Design Principles and Patterns*. Addison-Wesley, 1996.

Lie96 Lieberherr K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

LiO96 Lieberherr K., Orleans D., *Preventive Program Maintenance in Demeter/Java*. Northeastern University, College of Computer Science, Boston, USA. Internet: `http:// www.ccs.neu.edu/ research/ demeter/ sources/ DemeterJava/ demo/ icse/ doug.html`, 1996.

Liu96 Liu C., *Smalltalk, Objects, and Design*. Manning Publications, 1996.

Mar97 Martin R., Java and C++: A Critical Comparison. *C++ Report* 9, 1, 1997, 43-49.

Mic96 Microsoft, *Visual J++*. Internet: `http:// www.microsoft.com/ visualj/`, 1996.

NCS97 The National Center for Supercomputing Applications (NCSA), *Habanero*. University of Illinois at Urbana-Champaign, Internet: `http:// www.ncsa.uiuc.edu/ SDG/ Software/ Habanero/ HabaneroHome.html`, 1997.

Pre95 Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

Pri96 Princeton Secure Internet Programming Team, *Java Security: Frequently Asked Questions*. Princeton University, Department of Computer Science, Internet: `http:// www.cs.princeton.edu/ sip/ java-faq.html`, 1996.

SAI96 Science Applications International Corporation (SAIC), *The SAIC Agent Framework*. Internet: `http:// riz.saic.com/ AIT/ agentref.html`, 1996.

SCG96 Silicon Commander Games, *TurboSprite for Java*. Internet: `http:// www.silicmdr.com/ TSHome.htm`, 1996.

SiG96a Silicon Graphics, *COSMO 2.0*. Internet: `http:// www.sgi.com/ Products/ cosmo/ code/ index.html`, 1996.

SiG96b Silicon Graphics, *IRIX 6.2 — Silicon Graphics Multiprocessor, Multi-threaded, UNIX Operating System*, Internet: `http:// www.sgi.com/ Products/ software/ IRIX6.2/ IRIX62DS.html`, 1996.

Str86 Stroustrup B., *The C++ Programming Language*. Addison-Wesley, 1986.

Str96 Strategic Technology Resources, *The Session Framework*. Internet: `http:// www.str.com/ java/ session/`, 1996.

Sun96a Sun Microsystems Inc., *Java Beans: A Component Architecture for Java*. Internet: `http:// splash.javasoft.com/ beans/`, 1996.

Sun96b Sun Microsystems Inc., *Java Telephony API*. Internet: `http:// www.javasoft.com/ products/ javatel/ index.html`, 1996.

Sun96c Sun Microsystems Inc., *Overview of the Java HTTP Server Architecture*. Internet: `http:// www.javasoft.com:80/ products/ java-server/`, 1996.

Sun96d Sun Microsystems Inc., *The Java Development Kit (JDK) 1.02*. Internet: `http:// java.sun.com/ product/ JDK/ 1.02/`, 1996.

Sun96e SunSoft, *The Java Workshop*. Internet: `http:// www.sun.com/ workshop/ java/`, 1996.

Sun96f SunSoft, *The Java Home Page*. Internet: `http:// java.sun.com/`, 1996.

Sun96g SunSoft, *Java Servlet Application Programming Interface White Paper*. Internet: `http:// jeeves.javasoft.com/ products/ java-server/ alpha2/ doc/ api.html`, 1996.

Sun96h Sun Microsystems Inc., *Java Electronic Commerce Framework Architectural Overview*. Internet: `http:// www.javasoft.com/ products/ commerce/ index.html`, 1996.

Sun97a Sun Microsystems Inc., *Java Remote Method Invocation*. Internet: `http:// splash.javasoft.com/ pages/ rmi.html`, 1997.

Sun97b SunSoft, *Java Development Kit, Version 1.1*. Internet: `http:// www.javasoft.com/ products/ jdk/ 1.1/ index.html`, 1997.

Syb96 Sybase Inc., *Jato*. Internet: `http:// www.powersoft.com/ products/ internet/ javatool/`, 1996.

Sym96 Symantec, *Café*. Internet: `http:// www.symantec.com/ cafe/ index_product.html`, 1996.

Vil97 Viljamaa J., Tools Supporting the Use of Design Patterns in Frameworks. Report C-1997-25, University of Helsinki, Department of Computer Science, 1997.