The
BISS AWT
framework

Version 0.86
January 1997

Contents

List of Figures

Chapter 1

Introduction

1.1 What is BISS AWT

The BISS java framework is a framework of about 150 Java classes for building graphical applications with state-of-the-art user interfaces. Instead of following the "OS-native" look-and-feel (like java.awt), it implements its own "Java-native" look-and-feel. The framework can be extended without the need to write native libraries and uses a more powerful mechanism for event processing than the standard, low-level approach of the JDK 1.0.x.

It also forms a suitable base to implement a stand-alone Java AWT because it requires much less support by native libraries. This is due to the fact that BISS AWT just uses the generic components of java.awt (like Graphics, Frame, Canvas and Panel) and rebuilds all high level widgets (like Lists, TextAreas etc.) on top of this. To prove this, BISS AWT comes with a native library build for the publicly available *kaffe* virtual machine.

The structure of BISS AWT even makes it possible to use it as a replacement for the non-generic, high-level java.awt.peer classes (ListPeer etc.). It therefor also is a good platform for "clean-room" java.awt compliant AWT implementations.

1.2 History

By October 1995, we began to look for a programming platform that should enable us to port our large C++ framework for OS/2 (called BSA) to Unix and Windows. The BSA, which currently consists of about 500+ classes, covers all aspects of OS/2 application programming, but definitely has its strengths in creating large-scale applications with sophisticated user-interfaces (beyond usual presentation manager capabilities). It is mainly used by some European insurance companies and also contains powerful tools to ease source programming (called CThrough) and to graphically specify static and dynamic behavior of apps without the need to program sources at all (called the ObjectDesigner).

After some validation tests, we decided to go with Java by end of 1995. We did this mainly because of the inherent technical strengths of Java as an application programming system. But by then, less was known about how the available Java implementations behaved outside of small, Web centric "applets".

Soon after, we realized that the standard java.awt implementation from the Sun JDK could not be used for our purposes because of several reasons:

- it had bugs in all implementations, which all behaved differently with respect to these bugs
- substantial state-of-the-art GUI features were missing (e.g. pop-up menus, notebooks)
- and most important: it was not extendable (by writing Java classes) with respect to existing widgets (TextArea etc.)

Most of this was caused by the JDK implementation of java.awt, which provides a platform native look-and-feel and therefor heavily utilizes native libraries. These libraries in turn require a large support infrastructure of peer classes and are the single most important source of problems for bug fixing as well as porting Java to other platforms.

We found Randy Chapmans Linux port working in most cases, but it was crucial for us to proof Javas capabilities on at least Linux, OS/2, NT and Windows 95.

Therefor, by February of 1996 we started to write replacements of the existing java.awt widgets. These replacements were not meant to be a complete substitute of the java.awt, they were designed so that they could live in coexistence. But from the beginning it was clear that all of this work had to be done solely in Java (because of portability and extendability).

Besides deriving all classes from the generic parts of the standard java.awt (Canvas, Panel etc.), we also decided to use a more powerful mechanism to connect widgets to non-visible parts of the application (AKA models). Rather than using a non Java-native approach like methodRefs, we have chosen the standard Observer/Observable pattern. It is more powerful than the chained handleEvent() mechanism and (in a slightly more generalized form) is not constrained to a single class-subtree.

A useful side effect of this design is the fact that widget implementations use the same type domain like the rest of the Java app (e.g. Text widgets work on common Java String objects rather than on C-string buffers, Lists work with Vectors containing arbitrary objects instead of using separate, platform specific representations of string lists).

The first real test case for this framework was a package called biss.jde which implements a source programming toolset that will be expanded to the size of the CThrough system.

By middle of 1996, with the release of JDK 1.0.2, Sun started to tighten their license policy (at least they seemed to do so). As a logical consequence, many people began to think about free Java implementations. At this time, there already was a usable, yet not matured, free virtual machine available called *kaffe* (built by Tim Wilkinson). But there was no java.awt. Clause Lecommandeur soon after started his *sawt* project. Its goals are to create an X-based native library / peer layer without Motif (the toolkit used by the JDK). Otherwise it follows the same design patterns, and that means there is a lot of work to do because it basically has to implement its own toolkit to build widgets like TextAreas, Lists etc. It is targeted just on the X environment.

At first, it was just a silly idea. But then we began to realize that it was feasible to use the existing biss.awt class hierarchy to implement the java.awt.peer interfaces for the non-generic widgets like Lists, TextAreas, TextFields, Buttons etc. This opened up some great opportunities:

- really portable java.awt widgets with application specific look & feel
- just a very small native layer is required to implement the native support

By August 1996, we started to talk about these opportunities with some of the main Linux-Java contributors, namely Randy Chapman, Karl Asha and Miguel de Icaza. We got very encouraging responses, which convinced me to try to make BISS AWT the GUI base for the upcoming Free Java (Jolt).

Even with the attitude to make BISS AWT a platform for free Java programming, there was the question of how compatible to be with respect to the standard JDK, i.e. if we should build something that was able to run unmodified Java code written for the plain JDK. Given the fact that BISS AWT can perfectly run on top of the standard JDK (is downward compatible), I was somewhat reluctant to spend too much effort on this. Now that we have *two,* more or less compatible/coexisting, standard JDK event handling mechanisms (JDK 1.1 and 1.0.2), my second thoughts seem to be justified. However, after some discussions, we decided to modify the biss.awt so that it could get a platform for java.awt implementations. To prove its suitability, we also started to implement a native layer based upon the kaffe VM and Xlib.

1.3 Goals

Given the history above, it is easy to deduce the list of goals BISS AWT should meet. They are (with decreasing importance):

1. provide an easy-to-use, comprehensive, Java-extendable GUI library which can be used with all standard Java implementations

2. provide a base for an efficient and small, stand-alone Java GUI implementation (consisting of Java classes and native libraries)

3. to enhance the usability of such a stand-alone Java GUI implementation, provide as much standard java.awt compatibility as possible (but stay orthogonal, don't reinvent the wheel again within the same library)

The bottom line is: BISS AWT is not meant to be a global replacement of the standard java.awt, nor is it primarily a public java.awt implementation. Its main purpose is to give Java developers, who are challenged to build complex GUI applications, a library / toolset combination with a high degree of freedom and independence (with respect to Java platforms, operating systems and even shrink wrapped systems). And we are dedicated to extend this up to a complete, stand-alone AWT implementation. To say it with other words: we want autarky, and we want it in the most portable way.

This goal is motivated by a vision. Given the excellent runtime modularization and robustness of Java, it should be feasible to use it as some kind of a high-level shell on top of various operating systems. Classes would act as modules, main() methods would be the external commands of these modules. It would be some kind of a programming system positioned between Tcl and Oberon. It would be both portable and coexisting with other applications. It would be a dream becoming true.

Chapter 2

Technical Design

2.1 Principles and Design Rules

This section describes the system- and language- independent rules we tried to apply to the overall design of BISS AWT. This is a somewhat philosophical topic, but there are so many systems without any evidence of a pre-implementation design that it seems justified (at least to remind ourselves) to spend a few thoughts on this issue. If you are familiar with systems design and just want to know about how certain BISS AWT tools or packages can be used, you might skip this section.

The first rule to mention always should be the *meta rule*: these things don't exist *per se*, their only legitimation is to fulfill some purpose. If it comes to a conflict between different rules, decide on how to proceed not by the best-match on all incorporated rules. Decide on what these rules are used for in this context. Use them as a guideline, not as a corset.

2.1.1 Small Number of Components

This really looks too commonplace to mention. Of course, there should not be a plethora of classes, packages or concepts (since all have to be understood and maintained). But then there is evolution. More and more features are added to the framework and sooner or later it ends up violating exactly this simplest of all rules.

The consequence first seems uncomfortable: during this evolution, there has to be a constant struggle to make things smaller again. Features should just be added if they fit in or if overall complexity could be reduced otherwise. Fighting the evil entropy shall never be ceased.

2.1.2 Leveled Tree Structure

The term "leveled tree structure" summarizes three different aspects:

- all components of the system should be organized into hiearachical levels
- there should be a unidirectional "uses" relation between components
- the more system-oriented ("low-level") a layer is, the less components it should contain which, in turn, should be more general

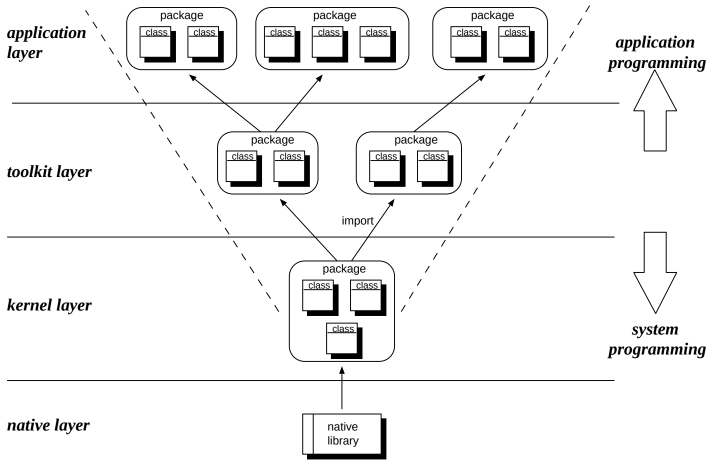


Figure 2.1: leveled tree structure

What is behind this rule is simply the observation that the most easy and obvious way to organize a framework is by using a hierarchical structure (tree) for the elements (e.g. classes) which is overlayed by a sequential structure (level set) for element groups (e.g. packages).

2.1.3 Generalized Packages

There are many possible organizational units on top of simple classes (e.g. applications, modules, libraries etc.). Most of them don't map very well to implementation languages. Instead of introducing such non-formal constructs (which would have to be supported / enforced by a non-standard tool layer), BISS AWT utilizes ordinary Java packages for the following purposes:

- to group code for related applications (e.g.the biss.jde package)
- to manage non-code resources (e.g. package properties / classes, string dictionaries, image sets etc.)
- to form layers (set of packages, each package strictly belonging to a single layer)

2.1.4 Usage of Patterns

If the "leveled tree" is the right measure to ease understanding of the big picture, consistent usage of a set of commonly accepted "patterns" is the right choice to let you understand the details of a framework. Things just look familiar if you identify a pattern you already found in other parts of the system.

The term "pattern" is used with the meaning described by the well known *pattern book*. In short, it is a set of classes and instances with some common scheme of collaboration to fulfill a specific purpose.

2.1.5 Inheritance vs. Parameterization

If it comes to increasing reusability, most people today first think of class inheritance (due to the non-technical hype OO experienced during the last years).

Class inheritance first and foremost is a mechanism to factor out common things (states and behavior) of a set of partly equivalent types. It is the right measure to avoid redundancy in a set of known classes, not to increase general reusability of a class. If it is misused like that, the number of classes in the whole system will explode. Think of implementing application specific Button behavior in Java programs by deriving a Button class (with re-implemented handleEvent() methods) for every specific Button usage (Edit, Save, Open, Cancel, ...).

With respect to reusability, it is much better to strive for generic classes which utilize parameterization to adapt to a variety of different contexts. Parameters don't have to be restricted to static behavior (colors, size etc.),they may also include collaborators (like Observers registered within an Observable) or actions (e.g. objects representing commands). With its single rooted type system, its `instanceof` operator, and especially its `interface` construct, Java is very suitable for this kind of programming. We consider this as a very important mechanism for implementing reusable GUI frameworks. Its exploitation is just limited by how easy it is to understand such a parameterized class model (classes should turn into amorphous monsters, they still should be designed for a single, specific purpose).

2.2 Mechanisms and Patterns

While the last section was rather abstract, this one lists the specific Java mechanisms and pattern implementations we used within the BISS AWT.

First of all, we tried to exploit already existing Java classes / patterns for all general mechanisms that can be found in the biss framework. We strongly feel that Javas standard libraries should not be proliferated with non-orthogonal concepts and classes (which would make it harder to understand). It there is a corresponding mechanism within the standard java libraries, this one should be used even if it is not the most efficient one.

2.2.1 Lib Classes

So called "lib classes" are collections of static functions that can be applied to certain standard Java types (e.g. biss.StringLib for java.lang.String). The main purpose of lib classes is to implement general functions which are missing in the standard Java classes without the need to subclass them. Subclassing of basic java.lang.* types should be avoided as far as possible in order to ensure coexistence of different class libraries.

A typical example is a Date conversion function which is capable of dealing with European date formats (dd.mm.yy). This is implemented in a class biss.DateLib that contains just such "missing" general functions working on common java.util.Date objects. Lib classes are usually concentrated in the *biss* package

2.2.2 PackageProperties and Package Classes

Given all the variety of potential platforms where Java apps can be run, there is a generalized need to parameterize these applications. In order to ensure that these apps are "runnable", it can never be assumed that such settings exist. But if they do, they should provide a flexible, yet general way to tailor the app to specific needs (ranging from display characteristics to special file system specifications).

In order to make this as portable and robust as possible, these settings should be specified in simple text files (analogous to Unix resources) rather than using mysterious (and notoriously broken) binary "repositories".

The java.util.Property class was designed to fulfill these requirements and therefor forms the base on which package classes are implemented.

Basically, package classes provide some kind of package local properties (as opposed to the system properties of standard Java). Such properties may be used frequently from various different locations of an app. It therefor is appropriate to transform properties to their final representation (e.g. a Point instead of a String containing two integer tokens). This can also be found in the standard Java system (e.g. static File.separatorChar which is initialized from the system property "file.separator"). But rather than (sometimes redundantly) spreading these static fields over various classes, we have chosen a convention of having one specialized class in each package which is in need of such properties. This so called *package class* is named after the package it belongs to (e.g. Jde for package biss.jde). It consists just of public static fields which are initialized from standard Java property files by means of String conversions (with ensured initialization) which are supplied by class biss.PackageProperties. It may also include some static functions which are of general interest for the particular package (something like a package specific java.lang.System class).

The most obvious example of a package class is biss.awt.Awt:

```
class Awt {
  static PackageProperties Values =
                             new PackageProperties("biss.awt");
 ..
  public static Font SysFont;
 ..
  static {
    ..
    SysFont = Values.getFont( "SysFont", "Helvetica BOLD 12");
    ..
  }
}
```

Property files are simple text files which consist of key-value pairs and are read in via the standard java.util.Properties.load() method. They are currently looked up in the following order:

- if a system property "java.resources" was defined (via "java -D<key>=<value>.."), its value is used as the directory where the Property files are searched
- if a directory ~/.java exists, this one is searched next
- finally, a dir ./.java is assumed to locate the Property file

Keys in a Property file should be fully qualified (include the package prefix) in order to enable accumulation of several package properties in a single file (which is not yet supported). File biss.awt is an example of such a Property file:

```
# file ~/.java/biss.awt:  standard biss.awt properties / parameters
..
biss.awt.SysFont=Dialog BOLD 14
..
```

Do not use backslashes (' \') within path specifications (java.util.Property might translate it, e.g '..\n..'). Input has to be conforming to the standard java.util.Properties (e.g. no spaces before or after the '='),

2.2.3 ObserverSockets

The ObserverSocket is a generalization of the standard Observer/Observable mechanism found in java.util. Within the biss framework, it is used for type-decoupling of various components (server components always should use as less type information as possible about their potential clients in order to increase reusability).

Despite of being a very useful pattern for GUI frameworks, the Observer / Observable mechanism has two deficiencies:

- type-independent application
- context information

The plain Observer mechanism requires the observed objects class to be derived from Observable and therefor is not applicable for arbitrary type subtrees (e.g. the widget classes).

In addition, it is often useful to let the Observer know about what kind of change within the Observable caused the notification without introducing specialized "state" objects. While this could be done by passing the information via the Observable.notifyObservers(Object arg) parameter, this often is inconvenient because the parameter is required to pass additional information, the event might happen too frequently (performance penalty by creating parameter objects), or the usage of notifyObservers() might be too widespread (imposing a consistency problem).

```
class SomeServer extends Observable {
  int State = UNINITIALIZED;
  public int getState();
  ..
  void initialize (..)  {
    ..
    State = INITIALIZED;
    notifyObservers(..)
  }
  ..
}
```

```
class SomeClient implements Observer {
  SomeServer Server = ..;
  ..
  public void update ( Observable obs, Object arg ) {
    ..
    if ( obs == server ) {
      switch ( server.State ) {
      ..
      }
    }
    ..
  }
}
```

Both can be achieved by adding observable fields to the observed class instead of deriving the whole thing from a particular base. Class **biss.ObserverSocket** (which in turn is an **Observable**) is used for this purpose and just adds some additional capabilities to the plain vanilla Observable (extra parameter, category etc.).

```
class SomeServer extends <whatever> {
  public ObserverSocket OsInitialized;
  ..
  void initialize(..)  {
    ..
    OsInitialized.notifyObservers( arg)
    ..
  }
}
class SomeClient implements Observer {
  SomeServer Server = ..;
  ..
  public void update ( Observable obs, Object arg ) {
    ..
   if ( obs == Server.OsInitialized ) {
      ..
    }
    ..
  }
}
```

ObserverSocket instances may have a few other attributes compared with ordinary Observables. They may have a symbolic name (or socket-class) which can be used to symbolically refer to a specific socket (or a set of sockets). There also is an additional argument placeholder in case the standard Observable argument is not sufficient.

With this generalization, the Observer mechanism is used consistently throughout the whole biss framework (not just for "model" components). Most classes can be thought of as finite-state-machines where each transition (change of state) that might be interesting for

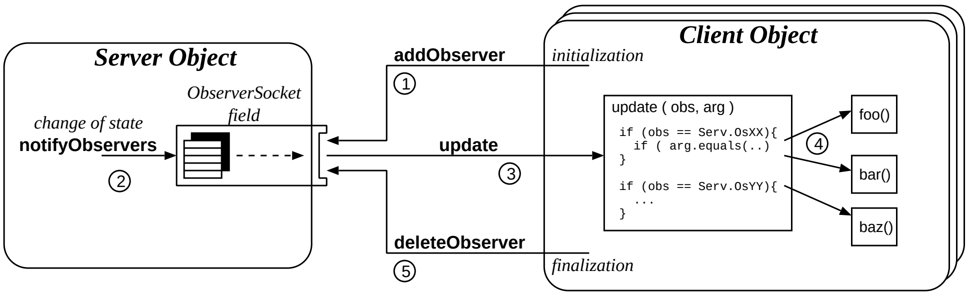


Figure 2.2: ObserverSocket

some client maps to such an ObserverSocket field. Just classes where all transitions at all times should be reported to all clients are derived directly from Observable (to avoid extra effort for keeping the Observer registration consistent).

It shouldn't be concealed that we don't consider the ObserverSocket mechanism being a grand new invention (neither is the Observer which has been around for several years implemented in many different languages). When we started with biss.awt, we first used some more sophisticated things like methodRefs (implemented by dynamically created reference classes). But we decided againt such complicated mechanisms because almost all of our goals could also be achieved with the much simpler Observable/Observer (and its beefed up ObserverSocket variant). We were not forced to introduce new concepts or change existing classes (like JDK 1.1) in order to get a more elaborate event handling. This one has equal power and runs on every Java platform.

2.2.4 MLP triad as an MVC replacement

MLP, which is short for *Model, Logical view, Physical view*, is an adaption of the classical *Model, View, Controller* pattern introduced by Smalltalk 80.

It is an approach to organize applications consisting of visible parts (views) and non-visible parts (models) in a way that strives for the best decoupling of these components. Ideally, a view should not know about the model which provides its logical contents, a model should neither know about who is displaying the provided information, nor should it know about how it is displayed. Models should also be capable of dealing with several views at the same time. Within the MVC, this was best described by the term "pluggable views". The weak spot in MVC always has been the controller part, mainly because:

- most available GUI platforms do their input acquisition directly via widgets
- there only is a formal protocol for the Model - View communication (a specialized Observer scheme), there is nothing like that for the Controller

MLP uses the Observer / ObserverSocket approach for all three components.

2.2.4.1 Physical View

A *physical view* is the most simple part of it and consists just of the definition of the view (its components and layout). Usually, it holds just instance fields for its components and a default constructor to define the layout. In order to ease user interface prototyping, (pane specific) pop-up menus and simple layout related behavior (e.g. zooming of components) can be implemented in a physical view too. But otherwise the physical view does not care or even know about displayed contents or user input reactions. Physical views should be generated by some kind of a graphical interface editor (to support the tedious layout work) and are executable units (have a main func to ease prototyping). Physical views have to be derived from some kind of a Frame or Dialog.

Look at biss.jde.LibViewer as an example:

```
class LibViewer extends TopWindow ..  {
  ..
  ObjectPane ItemPane = ..; // fields holding the widget components
  CheckButton FileBtn = ..;
  ..
  public LibViewer () {
    setLayout( ..);
    add( ItemPane, ..);
    ..
    pack();
    show();
  }
  ..
  public static void main ( String[] args ) {
    new LibViewer();
  }
}
```

2.2.4.2 Model

Models should be designed as FSMs with a suitable number of ObserverSockets. They should not care about the nature of potential clients (either physical or logical views). Usually, there is more than just a single Model class involved and there is some kind of an "access-" or "master-" model which manages various different low level models. Often these access-models are designed as singletons (to ease multiple synchronized views on the same data). Class **biss.jde.SourceLibrary** (managing lots of **biss.jde.CompileUnit** instances) may be used as an example. There is no need to derive model classes from a specific base.

```
class SourceLibrary { // access model for more primitive model
objects
  ..
  static SourceLibrary defaultLib;
  Vector CU; // container for "CompileUnits"
  ObserverSocket OsScanReady;
  ..
  void scanSourceDirs (..)  {
    CU = ..;
    ..
    OsScanReady.notifyObservers(..);
  }
  ..
  public static SourceLibrary getDefault () {
    if ( defaultLib == null )
      defaultLib = ..;
    return defaultLib;
  }
}
```

Models are the right components to gain reusability of a system. Much more than the logical or physical views, they can be structured in a way which is not specific to a single application (they should serve a whole set of equal-branched applications).

2.2.4.3 Logical View

The *logical view* brings in the dynamic behavior. It connects the physical view to the model by means of ObserverSocket entries. Its most important part therefor is the update() method. A logical view usually is the main entry point of an application (its main() method starts the application) and therefor usually (directly or indirectly) creates required physical view or model objects. There is a 1:1 relation between a physical and logical view which also has to manage the "state" (e.g. selections) of a view. Logical views have to implement the Observer interface, they don't have to be derived from a particular base. Class **biss.jde.LibBrowser** may serve as an example:

```
class LibBrowser implements Observer {
  LibViewer View = new LibViewer();
  SourceLibrary Lib = SourceLibrary.getDefaultLib();
  ..
```

```
    public LibBrowser () {
      ..
      Lib.OsScanReady.addObserver( this);
      View.FileBtn.OsAction.addObserver( this);
      ..
      if ( !Lib.isInitialized() )
        Lib.scanSourceDirs();
      ..
    }
    ..
    public void update ( Observable obs, Object arg ) {
      ..
      else if ( obs == Lib.OsScanReady ) { ..
        updateContents();
      }
      ..
      else if ( obs == View.FileBtn.OsAction ) { ..
        setFileMode(); ..
      }
      ..
    }
    ..
    void updateContents() {
      ..
      View.ItemPane.setContents( Lib.getDisplayItems());
    }
    ..
  }
```

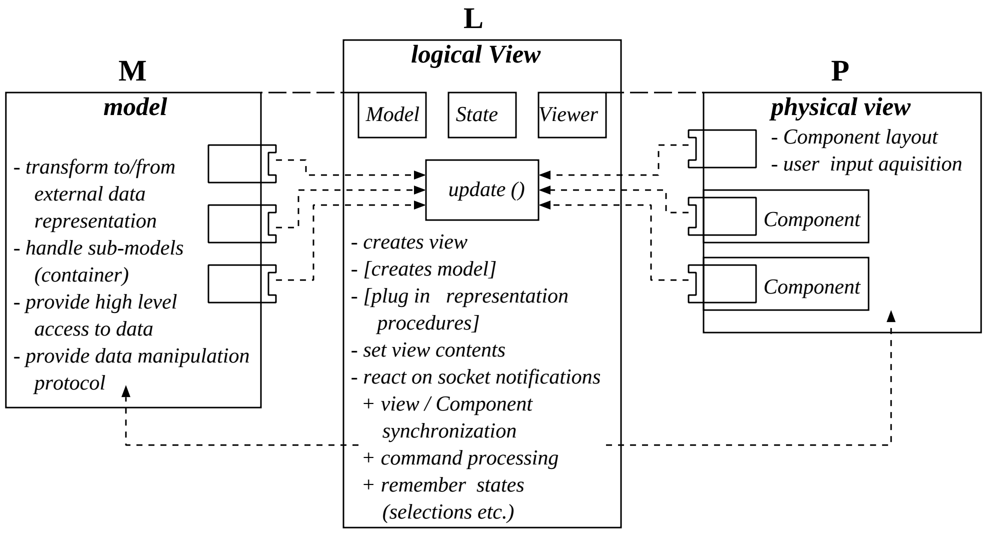
In addition to react on Observable notifications, another important role of the logical View is to remember the view state (e.g. selections, active related views/dialogs, expanded objects etc.).

In case of physical view components with configurable display behavior (e.g. biss.awt.Lists), the logocal view usually provides (or dispatches) the draw methods.

This part of the MLP should neither deal with data formats, data manipulation or view layout. It is meant to be the "glue object"between its collaborators. Within the triad, the logical view usually shows the lowest reusability (is specific to a certain application).

Logical view programming can be augmented by specialized tools too. For example, the update() method could be created by a specialized editor which recursively traverses all visible fields in search for potential Observable / ObserverSocket objects.

We don't think of the MLP as the only way to design interactive applications (because there is no implication of special bases, a physical and logical view might be identical in simple cases). Neither do we enforce this pattern. But we definitely had it in mind when we designed the biss.awt package.



2.2.5 Type Decoupling Choices

Given the ObserverSocket mechanism described above, there is the following list of choices (with decreasing degrees of coupling) to invoke actions within a set of peer objects (not to be mixed with Java peers):

2.2.5.1 Explicit Call

This is a static (compile-time) binding between components. It is the most efficient one, but obviously causes the tightest coupling. Because it requires the incorporated types to know about each other, it may introduce a mutual reference problem (which sometimes just can be avoided by introducing additional interfaces).

2.2.5.2 Explicit ObserverSocket - Field Reference

This scheme already uses the ObserverSocket mechanism described above, i.e. only the client has to know about the servers type. By explicit, it is meant that the client directly references a particular ObserverSocket field within the server. This has the significant implication that a whole chain of fields (up to the specified socket) has to be accessible (visible) for the server type (e.g. `PersonViewer.NameEntry.OsValidate`).

This has the obvious drawback of forcing a lot of public fields (which otherwise should be better hidden for information hiding purposes, e.g. the components of a logical view).

But it also has its benefits. Branching within an update() method can be done by means of identity checks

```
...
if ( obs == PersonViewer.NameEntry.OsValidate )
  ...
```

which could be significantly more efficient in case of frequent events. Of course, this should be used with special care to avoid NullPointerExceptions within the context of temporary objects (e.g. if the above `PersonViewer` might be just a transient dialog which has to be checked for existence before referring to one of its fields).

Using direct ObserverSocket field references also opens up the opportunity to build specialized editors for linkage specification. Because this tool would be based just upon field type analysis (which is done by other tools like the CUBrowser anyway), it would be much easier to implement than a scheme like the event delegation model of the JDK 1.1. It also requires much less classes and interfaces.

2.2.5.3 Symbolic ObserverSocket Reference

This is a slightly more decoupled version of an ObserverSocket based client - server object communication. Rather than using explicit field names, the client just uses the symbolic attribute of a socket to identify the server within its update() method:

```
if ( obs instanceof ObserverSocket ) {
  if ( ((ObserverSocket)obs).getCategory().equals( ``SaveAction'')
)
    ...
}
```

Because of the increased costs and the need for a well known set of socket names, this scheme is used rather seldom.

2.2.5.4 Generic Command Parameters

The ObserverSocket mechanism can be exploited even further by means of the simple interface

```
package biss;
public interface Command {
  public boolean execute();
}
```

To make the client react on something which is not hardwired within the clients class, the server object could pass an update() parameter object which implements the Command interface. To catch all these cases in a generic way, the clients update() method then would have a prefix like:

```
...
if ( arg instanceof Command )
  ((Command)arg).execute();
...
```

This is some kind of a generic client utilization. Because the client still is the Observer, the command would be executed just within the clients context.

2.2.5.5 Stand-alone Command Objects

Within package biss, there is a class **AutarkCommand** which implements both the biss.Command and the java.util.Observer interfaces. Objects instantiated from this class can be used to trigger an action on a server event (notification) regardless of whoever registered this object within the ObserverSocket.

```
...
SomeServer.OsSomeEvent.addObserver( new MyCommand(..)  );
...
```

This scheme can be used regardless of the state of the object which executed the above code. It actually might not even exist any more at the time the command is executed. This is definitely the mechanism with the best type decoupling, but it also comes with the highest costs (with respect to both speed and space).

Chapter 3

Packages

This chapter describes the contents of the BISS AWT packages and shows how to use its main tools. It is not intended to list every class in detail, only the main class groups / categories, the key concepts and a brief description of the tools found within a package are presented (which should be sufficient in combination with the source comments).

Currently, the biss framework consists of four packages: biss, biss.awt, biss.jde and biss.awt.kernel. While the first three packages closely follow the leveled tree approach (biss as a basic layer, biss.awt as a toolkit and biss.jde as an application package), biss.awt.kernel is a bit different. As a matter of fact, it was added just to show how biss.awt can be exploited to get a stand-alone Java GUI implementation.

3.1 Package biss

This package contains about 20 classes required by the other biss packages. There is not much structure within this package, it merely is a collection of useful classes and interfaces which all have in common that they can be used outside of biss.awt applications as well. The prevalent group of classes are the *Lib Classes* like **StringLib**, **FileLib**, **DateLib** and **VectorLib** (other lib classes may follow). The package currently lacks elaborated container classes (but this might indicate that you can already live with the standard Java classes). This is certainly not true for the **Queue** class, which is very useful for queue applications (class Vector is not suitable for that purpose because front insertions are very inefficient).

3.1.1 Package Properties

Currently, there are just two entries of interest (which can be set in the property file ~/.java/biss):

biss.Language this property can be used to implement some kind of national language support (e.g. to access String dictionaries for multi-language applications, look at the biss.calendar example)

biss.EnvPath this is a global variable to store pathnames for arbitrary ressource lookup (image files, data files etc.)

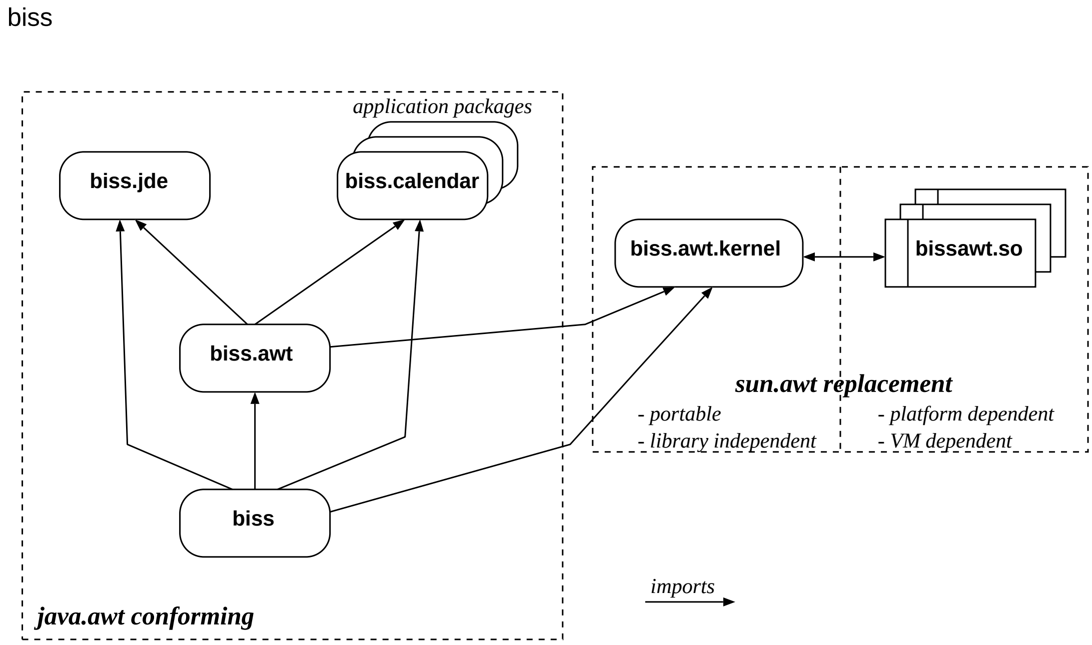


Figure 3.1: BISS AWT package structure

3.2 Package biss.awt

Here comes the main body of classes. This package is the real objective of BISS AWT and provides a comprehensive set of highly reusable GUI components which look and feel the same on all platforms (at least they should do so). It makes it possible to completely skip the native java.awt components (like java.awt.TextArea etc.) by just relying on the generic java.awt parts (like java.awt.Component, Container, Window, Frame, Canvas, Panel, Graphics, Color etc.).

Certain other packages have done that before (to implement single, specific widgets). But by consistently adopting the ObserverSocket mechanism described above, BISS AWT shows some characteristics which sets it apart:

- high level interface (good reusability) without the need to derive classes to bring in dynamic behavior
- Java - extendable (without the need for native support)
- suitable for tool support (e.g.. to specify a layout graphically)

In addition, package biss.awt implements a lot of java.awt.peer interfaces. This makes it a good base for a stand-alone java.awt development.

Six different groups of classes can be identified within this package:

3.2.0.1 Utility Classes

This category includes rather simple classes like **Key** and **Border**, but also very important concepts like the **GUIHandler**. This class (in combination width several interfaces like

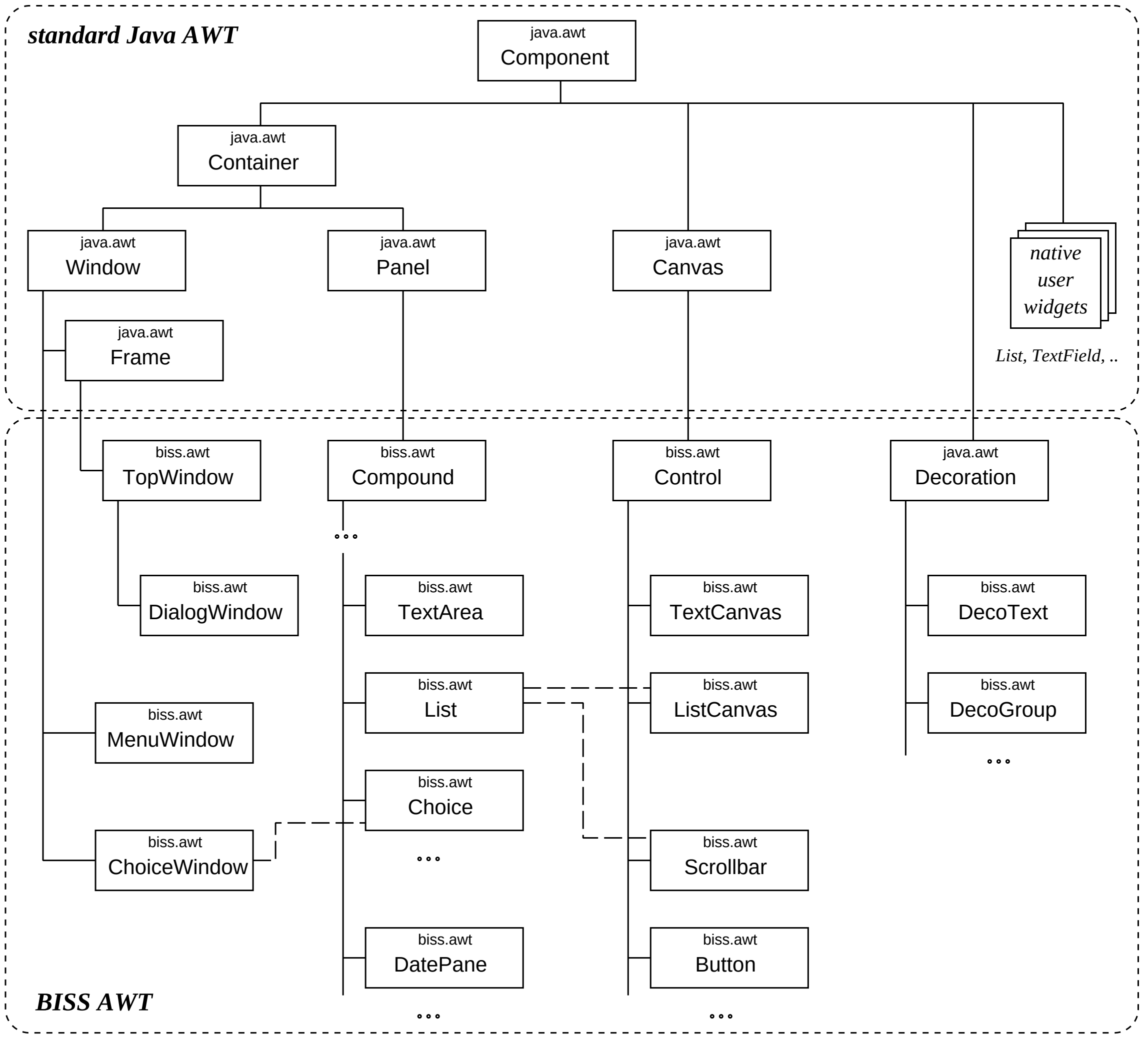


Figure 3.2: package biss.awt

FocusInterface, **KbdInterface** etc.) implements a delegation mechanism which is used for all biss.awt components to centralize event processing code (which is much needed because of the lack of an accessible common base, see below).

Another heavily utilized class is the **Layouter** (which is actually the base the Designer tool was build upon). This class implements a LayoutManager that provides a very fine grained control of specific component resize behavior. It s main idea is to describe first a bounding box relative to the parents extensions and then to apply fixed offsets to this bounding box to get the actual component coordinates (either specified in pixels or SystemFont units). Since the (relative) bounding box is allowed to degenerate (being either a line or even a point), this is a very flexible mechanism. In addition, the Layouter may be used to specify different layouts (for the same container) which can be referenced by a symbolic name. This is a very useful feature to implement widgets like notebooks or to zoom certain widgets within its container.

3.2.0.2 TopLevel windows

There are just two classes within biss.awt which can be used to instantiate decorated toplevel windows (managed by the windowmanager of the specific platform): **TopWindow** and **DialogWindow**.

The first one basically is a java.awt.Frame with some added ObserverSocket (like OsOpen, OsClose, OsCommand etc.) and a mechanism to handle overlay windows (which should be shown in front of the corresponding TopWindow instance).

The **DialogWindow** is not related to the java.awt.Dialog. It serves a similar purpose (to enforce some kind of sequential user action), but it does so without the need for any modal event processing. Within TopWindow, there already is support for Z-order management and per-TopWindow component disabling (provided by the addOverlay() method). Class DialogWindow just adds the dialog specific Cancel behavior. Otherwise (transient) DialogWindows are treated like ordinary windows, i.e. communication with the owner is set up by means of ObserverSockets instead of return values. The reason behind this is threefold:

- modal windows can be dangerous for the whole system (Windows, OS/2)
- modal window processing is highly platform specific
- modal windows do not map very well to event - based GUI program structure (e.g. if the owner should continue to display its contents, modal dialogs do not stop owner code from being executed - it just happens invisible)

This scheme is just a bit inconvenient for very simple, value-returning dialogs like StringPrompters or ConfirmDialogs. Otherwise it is considered to be a rather worthwhile structural "constraint" (number of windows should be kept small to avoid cascades of dialogs, sequential processing of windows should be avoided to make use of the GUI strengths, etc.).

3.2.0.3 Transient windows

There are two classes which produce non-windowmanager handled windows: **MenuWindow** and **ChoiceWindow** (both don't have a CompileUnit / file of their own).

For several reasons, these windows are among the most demanding features of a java.awt implementation (again, don't think just of X), which means they often behave somewhat different (not to say erroneous) on different platforms:

- they have to be positioned explicitly (depending on the pointer or Choice component position)
- they should be exclusive transients (there is just a single instance of these windows allowed at the same time)
- there should not be decorations (like titlebars etc.)
- there should not be size constraints (alignments, minimum sizes, etc.)

3.2.0.4 Controls

This group is derived from **java.awt.Canvas** and includes the basic interactive widgets. It can be subdivided into public classes like **TextField**, **Button** or **Checkbox** (which can be instantiated explicitly), and hidden classes like **TextAreaCanvas** or **ListCanvas** (which are just used within Compound objects). None of the Control classes has children, all of them provide a rich set of ObserverSockets.

3.2.0.5 Decorations

Decoration derived classes are no real windows (don't have peers). They are a useful abstraction for rather static decorations of complex views (mostly TopWindows). Their only purpose is to do some specific drawing (images, boxes, text labels etc.) within their "parent". This does not require a real window construct and could be done perfectly well by using the existing java.awt.Component class as a base (it basically just requires a paint() reimplementation). However, Component only has has an (empty) package private constructor, which makes it impossible to derive directly from another package. We therefor decided to add class Decoration to package java.awt as some kind of a Trojan horse. This, in general, is certainly a bad practice (adding classes to foreign packages), but in this particular case the gains in both speed (no addNotify(), no peer object creation) and space (no native object) seem to justify it (especially if you think of complex windows with a lot of text labels).

3.2.0.6 Compounds

This is perhaps the most interesting group of classes within biss.awt. In difference to the more complex native java.awt widgets (like **List** or **TextArea**), there are no "hidden" components (like Scrollbars etc.) within their biss.awt counterparts. All widgets have to be composed by explicitly using Controls and Decorations and grouping them within some sort of Container. The abstract base for all these containers if the **java.awt.Panel** derived **biss.awt.Compound**. Most Compounds consist of a working canvas which can be scrolled (e.g. TextArea, List). These specialized Compounds are augmented by the (Compound derived) abstract base ScrollablePane (and its Control derived helper ScrollableCanvas).

Class **biss.awt.List** should be mentioned separately because of its drawing features. In difference to the usual List implementation (which just capable of displaying ordinary Strings), there are several choices of how to display its items (one by one) which are checked in the following order:

1. if the OsDrawObject socket is not empty, its Observer is notified to draw the item

2. if the the object to draw implements the **SelfDrawingObject** interface, it is drawn by means of its drawSelfIn() method

3. if nothing of the above holds true, the object is converted to a String (if required) and drawn as simple text (standard java.awt.List behavior)

There is even more support for sophisticated drawing (not just restricted to Lists), which is implemented by means of class **PaneDrawer** and interface **DrawablePane**. Sometimes, drawing of items is either context sensitive (depends on items before and after the currently drawn item), or it is too time consumptive to be done during normal scrolling operations (should be done asynchronously). A PaneDrawer instance is a thread which is created for a tuple consisting of a **DrawablePane** (e.g. a List object) and a **Queue** object (to store draw requests). It simply waits for a new draw request to arrive (via the Queue) and then calls the interface method drawPane(). This, for instance, is implemented in **ScrollablePane** by triggering the OsDrawPane socket. It is up to the client to register for this socket and to take appropriate action on being notified by it. For instance, biss.jde.LibBrowser uses this mechanism to call a method drawFileInfo() which certainly would be too slow to be done every time an item has to be drawn (because it queries information about the file system). Another nontrivial example is the syntax highlighting within the biss.jde.CUBrowser text pane (which is an ordinary **TextArea** utilizing a **PaneDrawer** to color syntactic constructs asynchronously).

There is another behavior of List which should be mentioned. Because there is no native widget behind **biss.awt.List** (all is done within Java), it can make use of Java objects not only for its items but also for the container these items are stored within. This means, you can simple hand over an existing **Vector** of arbitrary objects (even with heterogenous items) to a **biss.awt.List** widget and it will use exactly this **Vector**. This, in turn, means there is no redundancy between model data and display data (which always requires consistency checking efforts). A repaint of a List item always displays its up-to-date contents. This really shows the potential of widgets which are written completely in Java (and are able to stay within the Java type system without falling back to some primitive native data representation like `char*`).

It is no surprise that within the Compounds group there can be found many new and more sophisticated widgets like:

LeveledList a hierarchically organized List which allows to expand / collapse subtrees interactively

EditList a List which implements *in situ* edit capabilities for List entries (by means of an TextField popping up on top of the selected List entry

Panner a two dimensional scrolling widget

Notebook which is a widget to switch between different alternative layouts ("pages") of a container (this is a rather uncommon, but favorable implementation of a notebook, please refer to the class documentation for its legitimation)

DatePane a specialized widget which implements a front end for selecting dates

Since biss.awt does not use platform dependent native widgets to implement things like Lists, it has no "native look & feel". It was targeted towards some kind of a "Java application look & feel", which should look and behave the same as much as possible on all platforms. However, there are just a few things which are not instantly obvious. For instance,we use small red triangles instead of borders (which disturb any 3d effect) to mark the focus component. Pop-up menus remain open (due to their java.awt.Window nature) when switching to non-java applications (but are closed again by activating any other Java window). The *Alt* modifier is used for TopWindow global hotkeys, *Ctrl* for pane local hotkeys. Most of this behavior (like colors, extensions, fonts, key modifiers, border styles and text styles) can be configured either by means of the package class biss.awt.Awt or by means of static fields within certain classes (like DefaultBorderStyle within biss.awt.Label).

There should be a hint to a specific structural deficiency of biss.awt, the "no common root" - problem. Since biss.awt works on top of the unmodified, standard java.awt generic classes, it has to use **Frame**, **Panel** and **Canvas** as bases for its own widget classes (**Component** can not be modified). This means, there is no common base available for biss.awt.* to implement common behavior and state. This, in turn, means that just delegation mechanisms (like the GUIHandler) can be used to factor out these common things. Sometimes this is not possible or suitable (creates more overhead than what would be saved), which causes certain things to be implemented somewhat redundantly in the biss.awt root classes **TopWindow**, **Compound** and **Control** (e.g. the polymorphic canGetFocus() method). Special care should be taken to keep this redundancy at an absolute minimum.

A feature which was added afterwards to biss.awt is the peer role of many of its widgets (like **biss.awt.Button** implementing **java.awt.ButtonPeer**, **biss.awt.List** implementing **java.awt.ListPeer** etc.). Since most of the functionality was already there, it was mainly a matter of renaming methods to accomplish this (with less than 100 lines of code within the whole biss.awt package). The efforts included the following list of topics:

- add a PeerTarget field to Control and Compound
- modify the handleEvent() methods within these classes to forward event handling to this PeerTarget (if set)
- add specific peer-role constructors to the relevant classes (List, Button, TextArea etc.)

While the actual source modifications were relatively easy, understanding of its runtime object structure is not. Objects of package biss.awt acting as peers have "real" (native) peers by themselves. They are just the missing link between the specialized java.awt classes and the generic peer objects (like biss.kernel.Peer).

From the added indirection (e.g. biss.awt.Button between java.awt.Button and biss.awt.kernel.Peer), it is obvious that using biss.awt.* classes in a peer role is slightly less efficient (both speed and space) than using them in a non-peer role (which is definitely the recommended way). This feature only exists to implement some backward compatibility.

3.2.1 Package Properties

Many aspects of biss.awt display characteristics can be set (using its property file usually located in ~/.java/biss.awt). This includes fonts used for various purposes (menus, buttons,

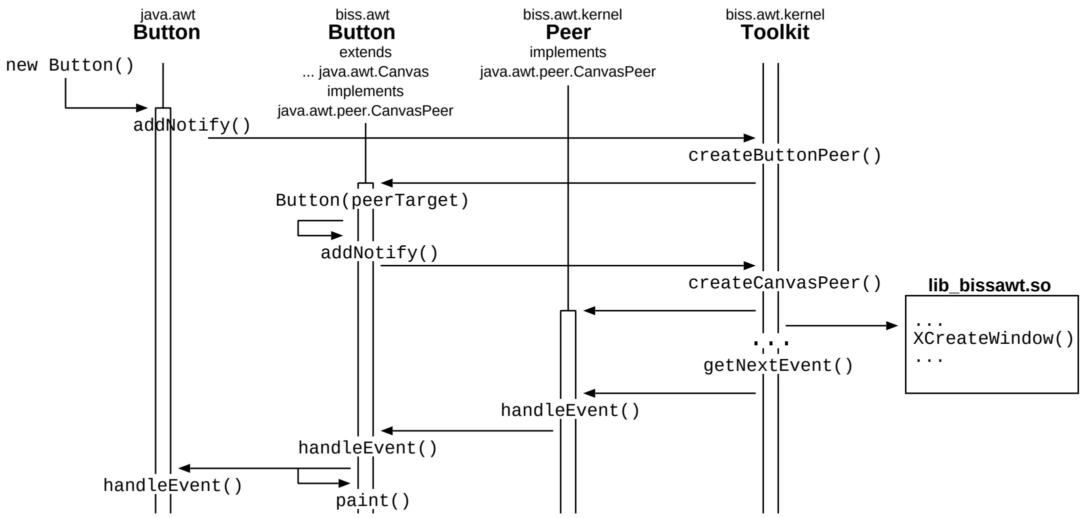


Figure 3.3: Peer Role of biss.awt.* Objects

labels etc.), colors (again, for many different widgets), and some dimensions (Scrollbar width etc.).

Because of its usage as a system unit, **biss.awt.SysFont** is the most important setting. The Awt class tries to get some suitable defaults (depending on the automatically detected ScreenSize and the used toolkit), but there is a good chance that this setting has to be changed manually. This is also true for other font settings like **biss.awt.ButtonFont**, **biss.awt.MenuFont** etc.).

Another setting which often requires a manual change (e.g. if using the Xinside server with default settings) is **biss.awt.GlobalKeyMod**. This setting is used to specify the Java key modifier which should be used for TopWindow-global hotkeys (e.g. '12' or '4' for Alt-key modifier on X servers). If your BISS AWT seems not to recognize global hotkeys (e.g. Alt-Z to zoom biss.jde.CUBrowser views), this is where you can fix it.

biss.awt.DlgXUnit and **biss.awt.DlgYUnit** are used to specify "portable" layout extensions (the Layouter can use this unit for its offsets). Usually, the SysFont based default value is appropriate.

biss.awt.ScreenSize describes the actual screen dimensions, which are used to scale printer output (e,g, images). If you want to get the most accurate output, you probably have to set this property (its default is just based on the ScreenSize).

biss.awt.GifPaths are used to specify a path list which is searched when locating GIF files via the **getImage()** method of class Awt.

Another setting which shouldn't even exist is **biss.awt.AwtToolkit** (which is set automatically from the default toolkits complete class name). Its only purpose is to efficiently enable platform specific workarounds for existing Java implementations). Hopefully, this will be gone some day.

3.3 Package biss.jde

This package serves two different purposes. First, it provides a toolset to browse and edit ordinary Java sources in general and the BISS AWT sources in particular. Second, it represents

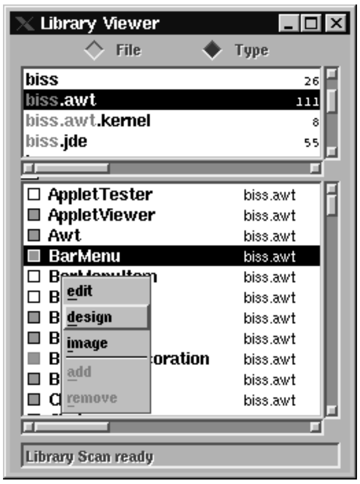


Figure 3.4: LibBrowser

a non-trivial example of a biss.awt based application.

The biss.jde is some kind of an IDE which works on regular Java source files, called **CompileUnits**, which each may contain several class or interface definitions. The key concept of biss.jde is to provide a general tool (the **CUBrowser**) for programming arbitrary CompileUnits, and a set of specialized tools for programming particular, biss.awt specific classes (most notably the **Designer**). All of these tools can be thought of as specialized editors (even if they process graphical input like the Designer). There outputs are just plain Java sources.

On order to make use of biss.jde, you have to set its package properties (via its property file ~/.java/biss.jde) according to your configuration. See the description below or class biss.jde.Jde for details.

This package currently contains about 50 classes. There are 5 physical/logical view pairs, the rest deals with scanning, parsing and management of Java source files.

3.3.1 LibBrowser

The LibBrowser is the starting point of the whole biss.jde application, which can be invoked by simply typing:

```
> java biss.jde.LibBrowser
```

It is used to display all CompileUnits which can be found in a set of (recursively processed) directories specified in the biss.jde property file (setting "biss.jde.SrcRoots").

Its main components are two Lists: the **ContainerPane** and the **ItemPane**. Currently, there are two modes (which can be set via two CheckButtons at the top of the view) to control the contents of these panes: FileMode and TypeMode.

In **FileMode**, the ContainerPane shows all directories being parsed, the ItemPane shows all CompileUnits (= *.java sources) within the selected directories (you can select more than one item in an ObjectPane by simply pressing the Ctrl key during a mouse click). If no directory is selected, all CompileUnits are being displayed. This is the initial mode of the LibBrowser.

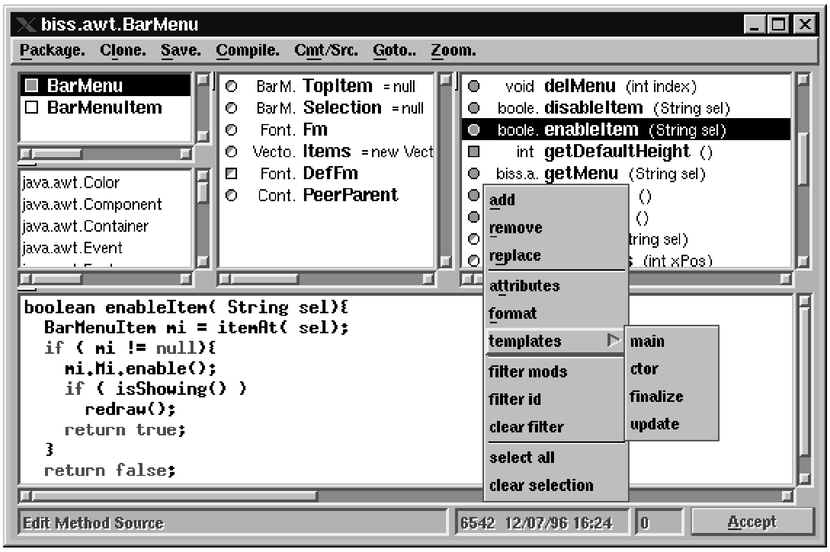


Figure 3.5: CUBrowser

In **TypeMode**, the ContainerPane shows all packages that have been found within the parsed CompileUnits. The ItemPane shows all types (classes or interfaces) which belong to the selected package (again, all types if there is no ContainerPane selection).

By double clicking (or hitting the enter key), a CUBrowser can be opened on the selected CompileUnit / type. Adding or deleting CompileUnits can be done via the pop-up menu (on mouse button 2) of the ItemPane. Additional LibBrowsers can be spawned with the ContainerPane pop-up.

3.3.2 CUBrowser

This view is used to browse, modify and compile the contents of a single CompileUnit (*.java file). It is some kind of a class browser tailored to the specific needs of Java.

It has five major panes:

TypePane this pane, located at the top left displays all types (classes / interfaces) found within the CompileUnit.

ImportPane which is located below the TypePane, shows all imports of the CompileUnit.

DataPane right of the TypePane, displays all fields (both instance and static) of the selected type.

MethodPane located at the upper right, displays all methods of the selected type.

TextPane this large pane which occupies the bottom half of the CUBrowser is used to show and edit the textual representation of the selections from the upper panes (either type, imports, datas or methods). If there is more than one selection (e.g. several methods), all corresponding items are displayed.

Below the TextPane there is a row of StaticText widgets: the **ModeInfo** (displaying the current mode), the **FileInfo** (displaying size and timestamp of the CompileUnit) and the ChangeInfo. The **ChangeInfo** shows a number (of not yet saved but accepted changes to the CompileUnit) and an optional '+' if there currently are unaccepted changes (modifications) within the TextPane.

At the bottom right of the View, there is the **AcceptButton** which is used to validate modifications (according to the currently displayed mode in the ModeInfo).

The CUBrowser knows two different levels of modifications: *unaccepted* (not yet validated modifications of the TextPane) and *accepted* (validated changes which are not yet saved to disk). Each time the CompileUnit is saved, The ChangeInfo count is reset to zero.

In case of an operation which would loose a not yet accepted or saved modification, the AcceptButton and the ModeInfo is displayed in alert mode. The operation is not yet performed, giving the user exactly one chance to accept the change. If the user repeats the action that caused the alert, this action is performed and the unaccepted/unsaved changes are lost.

When accepting modifications, the TextPane contents are parsed according to the current mode (basically Type, Import, Data or Method). This gives the parser additional info so that it can be more tolerant than the original Java syntax (e.g. a simple list of names without "import" keywords is sufficient to specify a list of imports).

In all modes, more than a single construct (type, import data or method) can be entered. Datas and Methods can me mixed. To display / edit all items of one of the upper Lists, simply press the F12 button (or choose "select all" from the pop-up) within it. Hitting the ESC key resets all List selections.

When entering new methods or datas, modifiers (public, static etc.) are copied from the current selection. However, modifiers can also be specified explicitly. The current modifiers of an item (type, data or method) are displayed as symbols (box for static, circle for instance, no fill for private, half fill for protected, full fill for public, diagonal fill for package default).

All Panes have pop-up menus (brought up by clicking the right mouse button) to add, delete, set modifiers, set filters or set/reset selections.

Underscore chars in Buttons and Menubars can be used as global hotkeys (e.g. alt-a for the AcceptButton, alt-c for compile, shift-Ctrl instead of alt for Windows and OS/2).

Underscore chars in pop-up menus can be used as pane local hotkeys (e.g. Ctrl-a for paste in TextAreas).

There are SplitBars (which can be dragged around by mouse) between all panes to change the geometry of the view. In addition, there are three different zoom levels which can be toggled by activating the Zoom menubar option (by mouse or alt-z).

The Src/Cmt menubar option can be used to switch between source of comment display/edit.

3.3.3 MethodBrowser

This tool, which can be launched by double-clicking on a MethodPane entry in the CUBrowser, is used to further analyse and modify the contents of a single method. It is still in its very infancy.

Usually, methods should be small (about 1 page of text). This means there should be no need at all for this tool. However, the biss.awt structure has one method which tends to be rather lengthy: the update() method of Observers. Especially the logical views (like

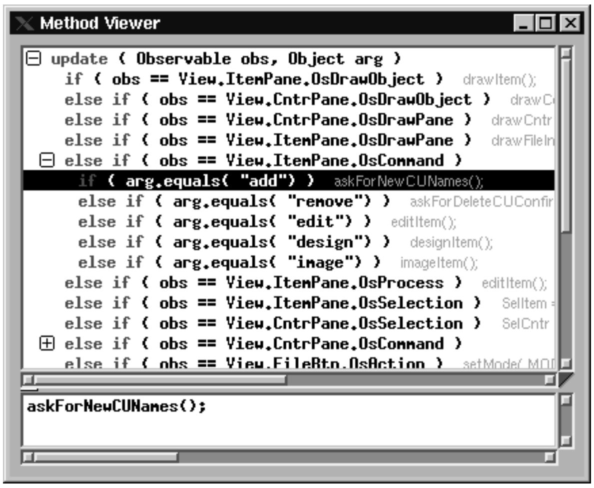


Figure 3.6: MethodBrowser

LibBrowser or CUBrowser) may contain many different entries in this method, which are all layed out in a scheme like:

```
public void update ( Observable obs, Object arg ) {
  if ( obs == View.ItemPane.OsDrawObject )
    drawItem();
  else if ( obs == View.CntrPane.OsDrawObject )
    drawContainer();
  ...
  else if ( obs == View.ItemPane.OsCommand ) {
    if ( arg.equals( "add") )
      askForNewCUNames();
    ...
  }
  ...
```

The upper **StructurePane** shows the contents of a method as a list of hierarchically grouped statements (this is an example of the biss.awt.LeveledList widget).

The lower **ActionPane** displays the sources for the selected entry and can be used to modify it.

In it final version, you will be able to copy / drag method entries from CUBrowsers to this tool in order to specify notifications and appropriate actions. Again, this is not available yet and functionality as well as layout of this tool may change in the future.

3.3.4 Designer

The Designer is a code generating tool to create and modify the layout of *physical view* classes. It is launched via LibViewer pop-up menu item 'design' and works on the same CompileUnit as all other BISS JDE embedded tools (e.g. CUBrowser), so CompileUnit modifications by the Designer can never go out of sync with other, simultaneously opened tools.

The Designer tool consists of a pair or two tightly coupled TopLevel windows:

- the Designer itself (with two notebook pages)
- the LayoutCanvas

It is important to notice that the graphical editing within the LayoutCanvas does not try to be as WYSIWYG as possible. While there can be extensions to do this (by means of plug-in drawing classes), we decided against it in the basic Designer version because of the following reasons:

- many widgets / decorations (like text labels etc.) may have transparent areas which would not be visible if drawn WYSIWYG
- some widgets cannot be recognized without displaying data (e.g. a List and a TextArea)
- the Designer would by tightened to a closed set of component classes (in its current form, you can use every Component class, the Designer treats them all in a generic way)
- fine-tuning layouts with respect to WYSIWYG representations is useless if you think of cross-platform portability with different screen resolutions / fonts / colors

We strongly feel that the current graphical representation is more suitable for professional use (to specify many complex views in a portable way). Besides, the preview mechanism (which does not require a compile) is fast enough (usually < 3 sec) to compensate the drawbacks of this design. To verify this, open up a Designer on the biss.jde.CUViewer class (by first selecting it in the LibBrowser and then selecting the “design” popup-menu item), look at its design representation and then push the “Preview” button within the Designer.

To create a new view first insert an empty CompileUnit to the class libray via LibViewer pop-up menu item 'add'. Then you can launch the Designer on that CU. It automatically creates the code components needed for further design operations on that CU (class, layout field, default ct, open() and init() methods) :

```
public class MyView
extends TopWindow {
...
Layouter Lo = new Layouter( this);
...
public MyView () {
  super( ``Title'');
  open();
}
...
public void open () {
  Lo.add( Button1, ...);
  ...
  init();
  if ( Bounds != null)
    openIn( Bounds);
  else
    openIn( x, y, width, height);
}
...
public void init () {
}
...
```

To modify manually created views with the Designer, the code must include at least the components described above, otherwise the CU is not designable. In this case modify the code manually to match the Designer criteria or build a new CU with the Designer and copy your code in there.

As already shown in the code section, the Designer uses a powerfull, all-purpose LayoutManager, the BISS AWT Layouter. For details, refer to the Layouter CU documentation.

The view consists of 2 Notebook pages:

3.3.4.1 Layout Page

This very populated page contains widgets to set the layout specification of a target component (selected within the LayoutCanvas)

Unit this group affects design session settings like CompileUnit name (=name of physical view class = file name, current layout selection, adding layouts and grid settings

Grid SpinButton to specify a layout grid in pixels

DlgUnit if set, all offsets of the view will be specified / stored in DlgUnits instead of pixels

Layout SpinButton to switch between the alternate layouts of the view

New adds a new alternate layout to the view

Save button which generates sources for the logical view class according to the current specifications

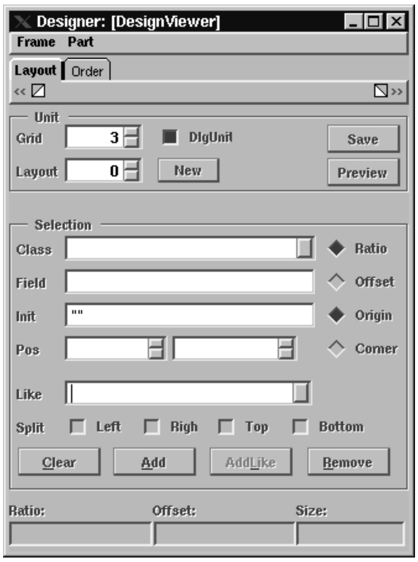


Figure 3.7: Designer Layout Page

Preview feeds a specialized ClassLoader with a temporary created WYSIWYG (bytecode) class of the current layout

Selection this group is used to add design objects to the current layout, or to modify the selected object

Class this Choice provides a set of currently supported Component classes

Field specifies the field name

Init the ct parameters for the field

Pos the 'Pos' SpinButtons (x, y) show (and fine-tune) the location of the point selected via the position Checkboxes(Origin / Corner and Ratio / Offset)

Like this Choice is meant to add a shadow of an object specified within the default layout (layout 0) to your current layout (typically BarMenus or Notebook controls)

Split the 'Split' Checkboxes automatically adds the appropriate Splitbars to the selected Component

PosInfo the position Labels at the page bottom inform about the selections position specification (xOrigin, yOrigin)(xCorner, yCorner), the left one shows the ratio part (in permille), the middle one the offset part (pels or DlgUnits), the right one shows the extends of the selection

Note that the dialog actions immediately modifies the selection, so first hit the 'Clear' button to deselect it before specifying 'add entry' settings.

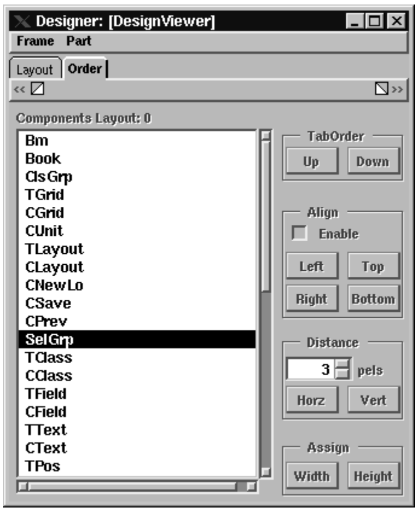


Figure 3.8: Designer Order Page

3.3.4.2 Order Page

The 'Order' page of the Designer view is used to specify the Tab-order of the current layouts Components. It also provides some group and alignment positioning functionality.

Align this group aligns all multi-selected Components (multiselection toggles by Ctrl-MouseButton1 from the ComponentPane or within the DesignFrame) relative to a positioning cross displayed in the DesignFrame. This cross toggles by clicking the 'Enable' Checkbox

Distance this group moves all multi-selections to equidistant positions

Assign reshapes all multi-selections to the size (width or height) of the master selection

3.3.4.3 Layout Canvas

The DesignFrame is a view opened automatically by the Designer to display the current layout Component placements. To select an object click MouseButton1 over the object (multi-selection Ctrl-MouseButton1), to deselect click over empty Frame space. The selection settings are shown in the layout page of the DesignViewer. The red handle points symbolize the positions of a selection (multi-selection green points), the blue ones (origin and corner) the ratio position part. To move a handle point click MouseButton2 over it and drag to the destination position, to move the master selection (red) or all multi-selections (green) drag MouseButton2 somewhere within it. Any position modification is immediately updated within the DesignViewer.

When you finished designing, don't forget to recompile the modified CU to put the changes to work.

3.3.5 ImageEditor

This is a specialized, graphical editor for **biss.awt.ClassImageSource**s (our portable way to handle icons and bitmaps).

Figure 3.9: Layout Canvas

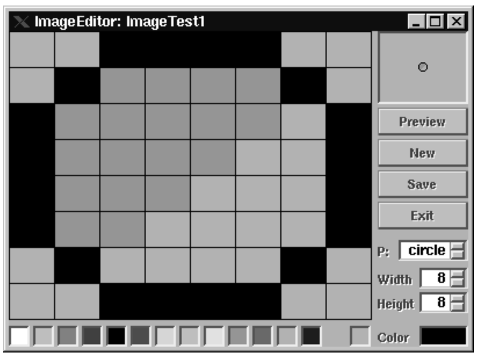


Figure 3.10: ImageEditor

It looks and behaves like a usual, simple icon or bitmap editor. The output created is a Java source file containing a biss.awt.ClassImageSource derived class that acts as an ImageProducer (based upon a MemoryImageSource) and does not require any local file system or network access in order to reproduce the image.

3.3.6 Package Properties

Various aspects of the biss.jde package can be set via its property file (which usually is located in ˜/.java/biss.jde). At least the following entries should be specified:

biss.jde.SrcRoots this is the single most important biss.jde setting which controls where the system looks for CompileUnits to parse (e.g. "biss.jde.SrcRoots=˜/java /usr/local/java"). Slashes '/' have to be used on all platforms as separators (even on OS/2 and Windows) because the standard Property.load() interprets backslashes as character escape chars (like '\n' for newline). Path separators can be either ' ', ';' (on all platforms), or ':' (on Unix platforms).

biss.jde.Excludes this setting specifies (in the same format) directories which should be excluded from the search.

biss.jde.ClassRoot tells the compiler where to store *.class files (maps to the "-d <dir>" option of javac)

For other settings (desktop location of views, syntax colors etc.), please refer to the Package class Jde.

3.4 Package biss.awt.kernel

This package is a stand alone implementation of a Java windowing system which does not need any of Suns (or any other vendors) native awt libraries or hidden classes (sun.awt.motif.* or equivalents). It consists of a Java layer (mainly with classes which are pre-determined by the java.awt) and a native layer consisting of plain *.c files (to implement the platform specific GUI access).

The main idea behind the design of this package is to do all specialized window management, event handling and drawing within biss.awt and restrict biss.awt.kernel to generic functions in order to keep it as small and portable as possible.

In addition, since the most work of biss.awt.kernel is done in the native layer, the Java layer should be kept portable. This means, implementing support for a new virtual machine or different platform should only involve writting a new native layer.

What seems a bit strange is the import of toolkit-level biss.awt classes into this system-level package (wrong import direction). There are two justifications for this design-anomaly:

- given the minimal expenditures to implement the peer role of the biss.awt widgets, it doesn't make sense to duplicate their functionality just to conform to the layer structure
- other classes imported by biss.awt.kernel (e.g. the GifProducer) can be used perfectly well in a non-kernel context too, there is no need to hide them just because of this single usage

3.4.1 The Java layer

This currently consists of 6 classes. They are mostly pre-determined by the java.awt design which makes use of a specific native lib by means of several abstract classes (java.awt.Toolkit, java.awt.Graphics etc.). As a consequence, the following classes can be found within biss.awt.kernel.

3.4.1.1 Toolkit

Toolkit reimplements java.awt.Toolkit and is the starting point for every AWT. It has the following responsibilities:

- create a NativeLib object to load the native library (libkaffe_bissawt.so)
- create biss.awt peer (e.g. biss.awt.Button) objects for high level java.awt targets (e.g. java.awt.Button)
- create biss.awt.kernel.Peer objects for the generic java.awt components (Canvas etc.)
- initialize the native counterparts of these peer objects from within the right thread
- asynchronously watch for Java events to arrive and dispatch them to their corresponding peer objects

Class Toolkit does all of this in a straight forward way, which requires further explanation just for two design decisions: registering peers and event acquisition.

Peer objects are kept within the **PeerTable** array field, which is used as a linear hash table from within the native lib (display.c getNextEvent). The reason why peers are registered in a Java data structure is to make sure that peer objects (and the whole graph of objects behind them) can be handled by normal garbage collection. If they would not show up in the Java layer, there would be a good chance that garbage collection would prematurely destroy them or that it would not recycle them at all (depending on the garbage collection subsystem used by the VM). The reason why a linear hash table is used in favor of java.util.Hashtable is that it is only accessed from the native layer. It would be significantly less efficient if the native layer would have to make Java callbacks in order to lookup entries. With open addressing the access seems to be effective enough to enable population rates of up to 80% (and therefor is no substantial waste of space). By using PeerTable like it is, peers are accessible from within the native layer (main access) and the Java layer (for future use, e.g. to dump all peers etc.).

Event acquisition at it topmost level requires a asynchronously executing loop. This is performed in the **run()** method (which is executed within its own thread). If this would be done within the native layer, it again would require a C-to-Java callback for every event delivered (something we would like to prevent). Another important aspect of event acquisition is if it should be done by means of blocking native calls. For compatibility reasons (lowest demand on a specific virtual machine implementation), we currently used non-blocking calls, which means we have to take care of lots of sleeps(). In order to prevent the "bumpy" event processing which is usually caused by this, we use a hint flag (NativeLib.AttentionHint) to indicate high-frequency series of events (e.g. key repeats). However, it should be possible to switch to a more efficient blocking scheme if the VM has sufficient support for kernel threading. If kernel threading is used, it has to be checked if the platform specific windowing subsystem requires windows to be created within the message processing thread (Presentation

Manager does, for instance). With the CreateQueue field, there already is some support to handle this.

It should be noticed that Java painting (which is itself generic by means of a polymorphic paint() method) does not use explicit Java event dispatching (i.e. it is not done from within any visible handleEvent() method). We therefor do the paint call from within the Toolkits run() method too. This means that Toolkit also has to include a Graphics object to do the painting (for efficiency reasons).

There is a potential pitfall within the Toolkit constructor. To enable recycling of array elements within the PeerTable (not yet used and recycled elements have to be distinguishable for open address hashing), a dummy Peer object is created. This requires class Peer to be statically initialized before the Toolkit constructor returns. If the default toolkit is either directly or indirectly referenced from within Peer (or any other class referenced by the Toolkit constructor), this will end up in two competing default toolkit objects (which are set in java.awt.Toolkit on return of our constructor).

3.4.1.2 NativeLib

This is the only class that actually calls the native library (provides the native method definitions). We decided to route every native call through this class in order of being able to make use of the Java inherent synchronization mechanism (all lib methods are synchronized). With sufficient kernel threading support by the VM / the specific platform, it should be possible to rely solely on synchronized methods and to skip this issue within the native layer (which currently uses the macros LOCK and UNLOCK to do this).

3.4.1.3 Peer

Peer is the implementation of all required (generic) peer interfaces (from java.awt.peer). It is a rather uncommon design to concentrate the interfaces in one class, but since it just has to cover CanvasPeer, PanelPeer, WindowPeer and FramePeer (and this mostly involves forwarding to the NativeLib object), it seems to be justified. The responsibilities of this class include:

- maintain parent / child relationship (**Parent** field)
- link to native window objects (store handles in the **Id** field)
- link to target object (either java.awt or biss.awt, by means of the **Target** field)
- generic Java event processing (method **handleEvent()**)
- handle toplevel window attributes (menubars of Frames)

Within the handleEvent() method, there has to be a special treatment of WINDOW_MOVED events for Frame and Window instances (both don't have a Parent) in order to take care of propper child resizes. This event usually is caused by an interactive rearrangement of the window (via some windowmanager), i.e. there is no automatic reshape() call within its context (which has to be done explicitly in addition to a general invalidation).

The parent - child tree of the native window objects is completely mirrored by a tree of Peer objects.

3.4.1.4 Image

Images might supposed to be easy objects, but they are not. There are many different facets of Images

- they have to be transformed from an external representation by means of an ImageProducer
- they have to settle on an internal representation once they are initialized by this producer
- there has to be a rendering protocol which is based upon this internal representation
- it has to be decided if the internal representation is an invariant (i.e. is not changed when the image is resized etc., which may have serious efficiency impacts)
- they have to include an abstraction for off-screen drawing space (which is a totally different thing compared to the above)

The transforming external representations is done by ordinary ImageProducers like biss.awt.GifProducer. In difference to the standard java.awt implementation, this biss.awt class is available for general use.

The internal representation is kept in the Java layer and is generically stored in the fields **Model** and **Pixels**. We think that it should be up to the external ImageProducer to decide on the most suitable format (to optimize storage consumption). However, this currently is constrained by the native lib draw function, assuming either an IndexColorModel-based byte array or an int array with standard Java RGB values. Once the internal representation of an image has been initialized, the image may act as an ImageProducer of itself. Thus, class biss.awt.image implements both the ImageConsumer and the ImageProducer interface.

The different states of an image object are distinguished by means of the **State** field, which can have the values:

UNINITIALIZED the image data is available (via the Producer) only in its external representation

QUEUED there already was a request to transform it to its internal representation. This request will be processed asynchronously by means of the **ProdThread** waiting on the **ProdQueue** (both are static fields of class Image)

INITIALIZING the production is already on the way (the image was already removed from the ProdQueue), but data is not yet available

PARTLYINIT still transforming from external to internal representation. Some data is already available (this is not yet used by the display functions)

INITIALIZED internal representation has been fully initialized. Subsequent pixel data requests can be satisfied by the image itself (either via ImageProducer protocol or direct access)

The native drawing mechanism directly works on the Image **Pixel** and **Model** fields instead of using an intermediate production. This has the shortcoming that possible internal representations are limited to the native layers capabilities. A Java production could first transform any internal representation to a single format understood by the native lib drawing format . But this would have the disadvantage that image data temporarily would have to be kept redundant. Given the immense memory requirements of some images, we decided against it (to avoid accumulating speed and space problems for every redraw).

The internal representation is kept as an invariant, i.e. it is not modified during a scaled redraw. As a consequence, scaling is done within the native layer (to speed it up).

There is no permanent upload of pixel data to the X-server. The **Id** field is just used for handles of off-screen drawing images (which have to be kept in the servers address space anyway).

3.4.1.5 Graphics

There is not much to say about the Graphics reimplementation. By far the most methods just forward to NativeLib. Caching of Graphics objects is done in the native layer, i.e. the createGraphics() methods of Peer and Image return new Java Graphics objects (which are linked to cached native lib GCs during initialization). Accessing Image functionality within drawImage is a bit awkward because class Image already imports Graphics (it has to create Graphics objects in its getGraphics() method). To avoid mutual import, this has to be done by utilizing the (not very well documented) checkImage() and prepareImage() methods of the Toolkit object. Unfortunately, this one has to be accessed indirectly via biss.awt.Awt.DefToolkit because biss.awt.kernel.Toolkit, in turn, creates Images. Sometimes, it is just laboriously to prevent mutual imports (in order to get a clean build sequence).

3.4.1.6 FontMetrics

This could be kept even shorter. FontMetrics objects don't have a substantial functionality, they just provide storage for font information which is computed in the native layer. It seems that class FontMetrics should better be non-abstract (metrics information can be stored in the corresponding Font object and its generic pData field).

Actually, there is another class **Dbg** which is just used for optional debugging purposes. To remove all debugging/logging from Java code, comment out the body of the log() method of this class and recompile biss.awt.kernel with optimization turned on ("-O") to inline functions.

3.4.2 The Native Layer

This layer is both platform and virtual machine dependent. The current BISS AWT comes with an implementation based on plain Xlib (no Xt, Motif or something else used here) which can be run with the *kaffe* VM of Tim Wilkinson (>= version 0.7.0). It was developed on an ELF-based Linux platform. Its description therefor tends to be specific to this environment. However, there are 7 major tasks every native support layer has to accomplish:

- window management (create, map, resize and destroy)
- event handling (aquire events, map them to peer windows, augment translation of native events to Java events)
- graphics support (provide graphics access objects and primitive drawing functions)

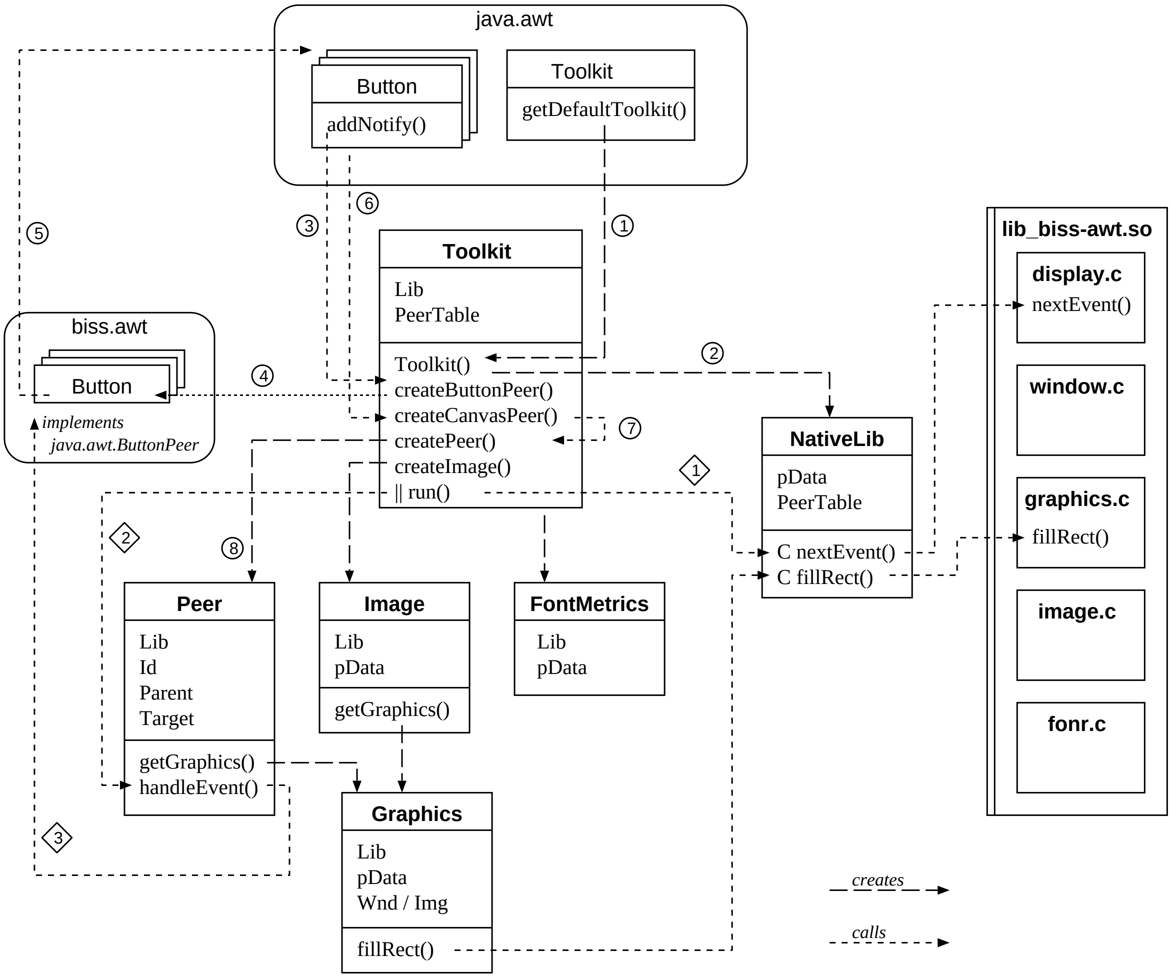


Figure 3.11: package biss.awt.kernel

- color handling (map Java colors to pixel values and vice versa)
- font handling (create and query characteristics of platform specific fonts)
- image handling (create, display and manipulate native images)
- cache time or memory intensive native constructs (GCs, fonts etc.)

This again maps pretty good to the file structure of the native layer

3.4.2.1 awt.h

This header file includes macros for accessing Java fields from within the native layer. Using these macros is the preferred way to access these fields to ease modification of data structures and class definitions.

All specific structures used within the native layer can be found in this file.

It also holds the kaffe VM specific synchronization macros (which are copied verbatim and have to be kept consistent with the original kaffe sources)

3.4.2.2 display.c

This file contains the

- peer hashtable access functions (used within getNextEvent())
- native lib initialization
- event acquisition function **getNextEvent()** (which is its most important part)

To avoid superfluous XEventsQueued calls, the number of currently available, not yet processed X events is stored within the **EventsAvailable** field of class NativeLib. In general, this function has to avoid any Xlib queries which are not strictly required because it is called quite frequently (by the event polling loop inside of the Toolkit run() method).

The getNextEvent() function also contains provisions to avoid some common Xlib programming errors (setting the focus to not yet mapped windows, getting desktop window positions for toplevel window ConfigureNotifies etc.).

3.4.2.3 window.c

Like what is implied by its name, this file contains code to manipulate native window objects. Function **createWindow()** is the only which actually creates them (by means of XCreateWindow). All conditional processing (e.g. setting window manager hints for Frame instances) is done by branching on Java field values. All windows except of toplevel windows are mapped automatically (i.e. become visible as soon as the toplevel window is explicitly mapped by show()).

Most other functions of window.c have very close Xlib equivalents.

An interesting design decision for this file would be if, and how, event selection is performed (one of the main objectives of JDK 1.1). Currently, we just do event compaction within display.c.

3.4.2.4 graphics.c

Within this file, you will find

- routines to cache a number of GCs
- a rather complex function to initialize a GC
- a lot of primitive drawing functions

The caching is done in a very simple way by storing unused (returned) GCs in an array. Each time a new GC is requested, this array is searched first.

The function **initGraphics()** is the most interesting one of this file. It is called whenever a new Graphics object is created or whenever the context of an existing Graphics object is changed. This function sets all relevant GC attributes (foreground, background, clipping attributes etc.) according to the current context object of this Graphics object (either a Peer or an Image).

3.4.2.5 color.c

This file holds the code to

- analyse and store information about the visual in use
- provides Java RGB to pixel conversions
- provide pixels to Java RGB conversions

At a first glance, this seems quite easy to do. But is has its obstacles, except of image.c, color conversion requires the most memory and incorporate the most awt.h structure definitions.

Since Java inherently uses an RGB color model, transforming to and from PseudoColor pixels is not very easy to do. Even if we reject using private colormaps, there are two possible strategies for RGB to pixel conversions. First, one could heavily depend on XAllocColor to either find a direct match or to allocate a new, globally shared color map entry. But this is slow and instantly eats up all available colorcells (other apps might depend on). We use a more sophisticated scheme. During native lib initialization, in case of a PseudoColor visual) all colormap entries are queried and stored in **Pseudo.Clrs**. From this array, a sorted array (**Pseudo.RgbSum**) containing the RGB sums of each cell is computed, the corresponding index to the colormap entry is stored in another array (**Pseudo.Idx**). Each time an RGB to pixel conversion is requested, function **pseudoPixelOfRgb()** first computes the RGB sum of the requested color. It then looks up (by bisection) a range of entries in Pseudo.RgbSum which have a tolerable RGB sum difference. Finally, it iterates through this range computing the sums of the absolute component differences of each cell. The entry with the minimal difference makes the game (is the "closest" color), but only if its value does not exceed a certain maximum (in which case a new cell is requested by XAllocColor). On the average, this mechanism requires just about 30 iterations (compared with >200 for a linear search in the colormap). It is significantly faster in case of frequent color conversions (e.g. during computation of scaled images).

Pixel to RGB conversion is trivial to perform with the Pseudo.Clrs array (simple index access).

True and DirectColor visuals are much easier to handle. To increase performance, specific Java RGB to pixel RGB transform parameters are computed during lib initialization (mask and shift value for each color component). A direct equivalence (8-8-8 pixel RGBs), which does not have to be a DirectColor visual, is indicated by the flag **Rgb.Direct**.

It is important to note that none of the above measures would be fast enough to replace local caching of pixel values in the **pData** field of Java **Color** objects. This technique, which can be essential for applications making intensive use of colors, is transparently implemented in function **getPixelValue()**.

Standard Java colors (static color objects) are initialized like this during lib initialization. This is important in case of a PseudoColor visual because there might not be an available colorcell any more ate the time the color is requested (leading to variable colors in different runs of the same application).

3.4.2.6 font.c

There are no such hidden obstacles within font.c. The only thing worth to mention about its font support is a little extension we implemented in **loadFont()**. If the font name is not one of the standard Java names ("Helvetica", "Courier", etc.), it is used verbatim for the first two components of an X font specification. This allows the (non-portable) use of all available fonts.

Things might be different on other platforms. The OS/2 Presentation Manager, for instance, requires a font to be created in the context of a PS (=presentation space, some sort of an GC). Accessing this font for use with this PS via its handle is fast. Using this handle without the PS it was created for is not allowed. Creating fonts (obtaining a new handle for another PS) is slow. Since the current native layer implementation just caches GCs (independently of fonts), and makes unconstraint usage of the cached GCs, this is a problem which still remains unsolved.

3.4.2.7 image.c

Most of the important facts for implementing this module already have been mentioned in the description of class Image. This class (once initialized) holds the complete pixel information as Java types (int[] or byte[]). Each time an Image object has to be displayed by drawImage() (in graphics.c), **setXImage()** converts this pixel data into a corresponding XImage. This XImage object is created during lib initialization, only its data member needs to be allocated by setXImage(). This is further minimized by using a global buffer (expanded in case its initial size does not already fit the request) for the pixel data.

Drawing scaled images is handled the same way. The current scaling algorithm has its strengths in conserving all pixel information (by extensive interpolation), but is not very efficient.

This implementation of the native layer does not use C-to-Java callbacks. Mainly because this is one of the less portable things in Java, but also because it seems to be slow (at least in some Java implementations). Communication from C to Java is performed by setting Java object field values. Most of them have primitive types (int, long etc.), a fact that might be important for some garbage collectors (which depend on knowing about storing references within Java objects). Most fields are accessed directly, but there are some nasty exceptions (like `..unhand(unhand(peer)->Target)->width..` in display.c) which are hard to circumvent.

Heap operations are avoided, most of the required memory is allocated during lib initialization.

3.5 Other classes

At the moment, we have three groups of classes outside of these packages:

3.5.1 java.awt.Decoration

Even if the package implies something else, this class is ours, not from Sun. It is some kind of a Trojan horse to enable non-peer Components (we call them Decorations) which otherwise would not be possible because of the package default Component constructor (not accessible from outside of java.awt). However, decorations are such a useful feature (for text labels, gifs, boxes etc.) that we decided to go with this little hack. What we like about them is the fact that they are really flyweight objects whose geometry handling can be done consistently by a LayoutManager. And it really was a breeze to implement them (except of the above obstacle).

3.5.2 PSGr

This Postscript Graphics implementation belongs to the great jlpr package of E.J.Friedman-Hill from Sandia National Lab (ejfried@ca.sandia.gov, http://herzberg.ca.sandia.gov). We put it in here just because of convenience. Within the biss framework, we use the unmodified PSGr as a base class for a more specialized Graphics implementation which is utilized for printing (screen snapshots, pane contents and whole documents). The biss.awt framework is ideal for this purpose because we do all painting inside of Java and therefor just have to (temporarily) switch Graphics instances (from screen to Postscript) in order to get the printing almost for free. However, printing is beyond the scope of this document.

3.5.3 JAS

The well known JAS (Java bytecode assembler) package of K.B. Sriram (availabe from http://www.blackdown.org/~kbs/jas.html) is used for on-the-fly bytecode creation (e.g. by the biss.jde.DesignPreviewer). Again, we are just standing on the shoulders of giants here. The unmodified JAS classes are just included to ease installation of the BISS AWT.

Chapter 4

A Case Study - the calendar application

The calendar application is a simple, plain ASCII file based, date planner and reminder. It is just meant to be an example of how to build real-world applications with the mechanisms, views and tools described in the previous chapter. Anyway, if you like it, just feel free to use it.

All data is stored and read from an ASCII file that can be specified by the biss.calendar.DefDateFile property setting in the calendars configuration file ~/.java/biss.calendar (default is 'dates.dat').

4.1 How it looks like

The calendar application is started by executing the command

> \> `java biss.calendar.DateBrowser`

or, in case you also want to start a background thread watching your dates to expire, by the command:

> \> `java biss.calendar.DateWatchDog`

The following is a brief description of its main views and how they can be used.

4.1.1 DateBrowser

The DateViewer is the Dialog view to add, modify and remove dates.

- select the current date by mouse-clicking a day from the DatePane on the right. The DatePane contents can be shifted by the next-month, prev-month or today buttons on top of it. To leaf through the selection, use the buttons beside its labelbox.
- choose the ItemPane view-mode with the buttons at the top:

Day a detailed timetable view on the selected day, recommended for any edit operations. This viewmode uses some more property settings:

  - biss.calendar.DayStartHour (begin of time table, default is 8 am),

  - biss.calendar.DayEndHour (end of time table, default is 8 pm) and
  - biss.calendar.DayTimeSlot (timetable slotsize, default is 20 minutes).

  The box drawn right from the time label shows some additional information about a timeslot occupation: filled white means not occupied, filled gray a single occupation, a number represents multiple occupation count and filled red warns about several occupations overlapping in time.

  Week an overview of the selected days week dates

  Month an overview of the selected days month dates

- add, modify or remove a date by selecting the appropriate timeslot from the ItemPane. If the slot already has dates, they are expanded to the DateEntryPane right of it. Selecting an entry from this pane expands it to the 'Selected date' group at the bottom to edit its details:

  Selection date, starttime and endtime

  Text textual description

  Category Choice with a set of predefined categories (can be set by the biss.calendar.Categories property setting)

  Details optional details-file, the File button opens a FileEditor on it

  Alarm optional date and time, used by the DateWatchdog

The Modify, Insert and Remove buttons refer to the selected date, the Store button saves all dates to disk. Most actions are also available from the BarMenu:

View select day / week / month viewmode

Date modify / insert / remove selection

Print prints a WYSIWYG copy of the ItemPane to a postscript file (name mangled from selection date and viewmode e.g. tt16.01.1997.ps) that can be used by any postscript previewer / printer (nice quality, have a look at it !)

Store save all dates to disk

Options

  Goto... opens a prompter to enter the date to jump to

  Clone launches another DateViewer on the same date

  CleanUp removes all entries.

Help not yet implemented (will show something like this)

Note that in this beta version, all date and time entries must be specified in European format (24 hours <hh:mm> time, e.g. 16:33 and <dd.mm.yyyy> date, e.g. 16.01.1997)

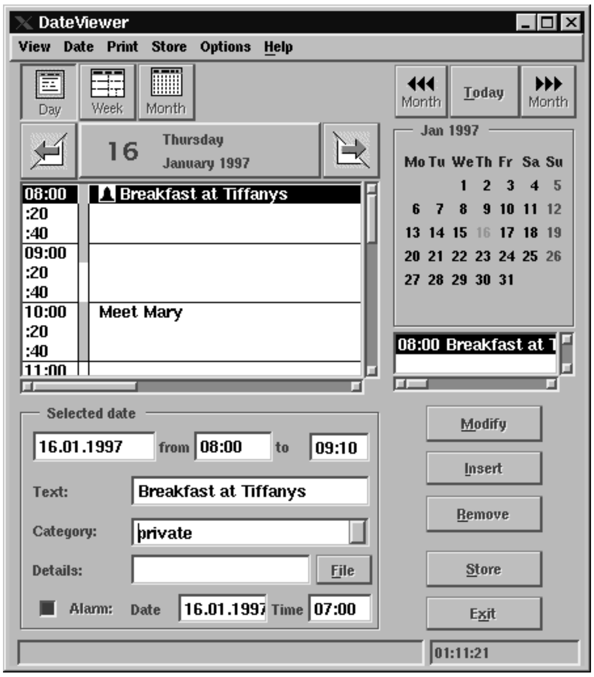


Figure 4.1: DateBrowser

Figure 4.2: DateController

4.1.2 DateWatchdog

Once you are familiar with the usage of the DateBrowser, you should use the DateWatchdog application instead, which is a combination of the DateBrowser and a background thread watching its alarm dates. The check interval can be specified by the biss.calendar.WatchDelay property setting (default is 5 minutes). The DateController view is used to launch a DateBrowser (via Edit button) or to exit the application (close views and watcher thread).

When the Watchdog detects pending alarm dates a notification dialog like this comes up:

To view, edit or remove an alarm date, select the entry from the ItemPane and push the Edit button to open a DateViewer on it. Most probably you expect the alarm entry to be processed now, so uncheck the DateViewers alarm Checkbox for that entry, store the modification to disk and close the dialogs. If you do not, the next check will bring up the

Figure 4.3: Watchdog

alarm again.

Appendix A

BISS AWT International License Agreement

1. **LICENSE GRANT**

   (a) BISS GmbH grants you ("Licensee") a non-transferable, non-exclusive, royalty-free, limited license to use a copy of the BISS AWT for Java Source and Binary Code ("Licensed Software"), including the right to distribute verbatim copies of the Licensed Software, provided that you publish on each copy an appropriate copyright notice and disclaimer of warranty and limitation of liability; also a copy of this copyright notice must be included and distributed with each copy of the Licensed Software. You may also modify and distribute portions of Licensed Software provided that you distribute the whole of the work to third parties under the terms of this license at no cost.

   (b) If you modify the source code or any other portions of the Licensed Software, you must place an appropriate notice in every changed file and clearly mark all changes to the source code at the respective places. If such modifications are sent back to BISS GmbH, and BISS GmbH, in its sole discretion, makes avaliable upgrades to the Licensed Software, these modifications may be implemented in these upgrades.

   (c) BISS GmbH is under no obligation to provide maintenance or support for the Licensed Software or to notify you of updates or upgrades. You may execute a commercial license agreement with BISS GmbH to receive full commercial grade support for Licensed Software.

   (d) If you distribute Licensed Software, it must be distributed as a whole with all source files, binaries and other files that came with the original package. If parts of Licensed Software are incorporated in applications where the distribution of source code is not required, the source code part of Licensed Software may be omitted from such distribution and only the required binaries may be included. In this case you must include a text in the sign-on message or information window of the software clearly stating that parts of the application are built using Licensed Software ("Built using BISS AWT for Java"). In any case this license agreement must be included with such distribution.

   (e) Work based on Licensed Software (e.g. applications using BISS AWT classes like browsers, editors, word processors, etc.) does not fall under this license agreement and may be distributed under the terms of your choice.

2. **COPYRIGHT**

   (a) The Licensed Software is copyrighted by BISS GmbH. You shall reproduce and apply any copyright or other proprietary rights notices included on or embedded in Licensed Software to any copies of Licensed Software or derived binaries, in whole or in part, in any form.

   (b) If Licensed Software is made available electronically (e.g. via the Internet) in its original or modified form, a notice must be included as of where to find the original form of Licensed Software (e.g. a link to BISS' Home Page).

3. **FEES**

   (a) Use of Licensed Software in accordance with the terms of this license shall not require payment of a license fee.

4. **TERMS AND TERMINATION**

   (a) Since you are not required to sign this license it shall commence on the date you start using Licensed Software whereby you accept this license.

   (b) The license terminates immediately, if you fail to comply with any of the terms or conditions of this license. Upon termination, use of Licensed Software by you, including use of all modifications made by you shall be immediately discontinued, all copies of Licensed Software shall be destroyed by you, and the license granted to you hereunder shall expire.

5. **DISCLAIMER OF WARRANTY AND LIMITATION OF LIABILITY**

   (a) SINCE LICENSED SOFTWARE IS LICENSED FREE OF CHARGE IT IS PROVIDED "AS IS" AND THERE IS NO WARRANTY OF ANY KIND. YOU ACCEPT ALL RISKS AND LIABILITIES ASSOCIATED WITH ITS USE OF LICENSED SOFTWARE.

   (b) BISS GMBH DOES NOT MAKE AND HEREBY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTIES RELATING TO THE LICENSED SOFTWARE, INCLUDING BUT NOT LIMITED TO, WARRANTIES OR NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR ANY WARRANTIES THAT MIGHT ARISE FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF LICENSED SOFTWARE IS WITH YOU. SHOULD LICENSED SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

   (c) IN NO EVENT WILL BISS GMBH BE LIABLE FOR ANY LOST REVENUES, DATA, OR PROFITS, OR DIRECT, INDIRECT, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, EVEN IF BISS GMBH HAS BEEN ADVISED OF THE POSSIBILITY OR PROBABILITY OF SUCH DAMAGES.

(d) LICENSED SOFTWARE IS NOT FAULT-TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN ON-LINE CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES, OR WEAPON SYSTEMS, IN WHICH THE FAILURE OF LICENSED SOFTWARE OR DERIVED BINARIES COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE ("HIGH RISK ACTIVITIES"). BISS GMBH SPECIFICALLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK ACTIVITIES. YOU MUST NOT USE, DISTRIBUTE OR OTHERWISE MAKE AVAILABLE LICENSED SOFTWARE OR DERIVED BINARIES FOR HIGH RISK ACTIVITIES.

6. **GOVERNING LAW**

   (a) Any action related to this license will be governed by German law, excluding choice of law rules.

BISS GmbH Chaukenweg 12 D-26388 Wilhelmshaven Germany
BISS AWT International License Agreement March 1st, 1997