

KnowledgeFrame Developer Guide

Version 5.2

Templates 4 Business Inc.

Web: <http://www.t4bi.com>

Address: Suite 1010 – 1177 W. Hastings St., Vancouver, B.C. Canada V6E 2K3

E-mail: contact@t4bi.com

Table of Contents

1.	Introduction.....	4
2.	KF Overview.....	6
2.1.	Architecture Vision.....	6
2.2.	KF Application Design.....	7
2.3.	Main Components.....	9
2.3.1.	Application Site.....	9
2.3.2.	Login and Security System.....	9
2.3.3.	Main Menu and Hierarchy of Modules.....	10
2.3.4.	Modules, Pages and Blocks.....	10
2.3.5.	Blocks and Actions.....	11
2.3.6.	Entities and Associations.....	13
2.3.7.	Business Rules.....	15
2.3.8.	Application Logic.....	16
2.3.9.	Custom Components and Behaviors.....	17
2.3.10.	Runtime Infrastructure.....	18
2.3.11.	Development Infrastructure.....	18
3.	Java Packages and Classes.....	20
3.1.	Site and Application Definition.....	20
3.2.	Framework.....	21
3.3.	Model.....	21
3.4.	Controller.....	22
3.5.	View.....	23
3.6.	Runtime.....	23
3.7.	Rule Engine Framework.....	23
3.7.1.	Declarative Approach.....	24
3.7.2.	Programmatic Approach.....	26
4.	Using Standard Application Components.....	28
4.1.	Entities, Attributes and Associations.....	28
4.1.1.	Overview.....	28
4.1.2.	Defining Structural Properties.....	29
4.1.3.	Defining Keys.....	32
4.1.4.	Defining Associations.....	32
4.1.5.	Defining Behavioral Properties.....	33
4.1.6.	Entity Property Methods Reference.....	33
4.1.7.	Attribute Property Methods.....	35
4.1.8.	Association Property Methods.....	38
4.1.9.	Event Activation Methods for Programmatic Rules.....	38
4.2.	Modules, Pages and Blocks.....	39
4.2.1.	Overview.....	39
4.2.2.	Defining Pages and Blocks in a Module.....	39
4.2.3.	Module Property Methods.....	41
4.2.4.	Page Property Methods.....	41
4.2.5.	Block Property Methods.....	42
4.3.	Using Symbolic Expressions and Bind Variables.....	43
5.	Development Environment Configuration.....	45
5.1.	Directory Naming Conventions.....	45
5.2.	KF Directory Structure.....	45

5.3.	JDBC.....	48
5.3.1.	Oracle JDBC.....	48
6.	Application Server Deployment	49
6.1.	Create a WAR File for Deployment	49
6.1.1.	Components of a WAR file	49
6.2.	Deploying the WAR file to the Server.....	51
6.3.	Generating KF Documentation.....	51

1. Introduction

KnowledgeFrame is a Java development framework, targeted at development of enterprise-level applications, using a lightweight and flexible technical architecture and low technology footprint.

Development frameworks are a de-facto standard in developing enterprise Java applications today – building all components from ground up is simply not viable.

A framework is a set of common prefabricated software building blocks that designers and developers can use, extend or customize for specific application system solutions. With frameworks developers do not have to start from scratch each time they write an application. Frameworks are built from collections of objects and so both the design and the code of the framework can be reused.

A framework captures the programming expertise necessary to solve a particular class of problems. Designers and developers reuse frameworks to obtain such problem-solving expertise without having to develop them independently.

The following benefits are expected when using a mature and comprehensive Application framework:

- Reduced project costs, time and risk.
- High quality product.
- Lower costs and time to respond to business or technology change drivers.

As Java frameworks go, KnowledgeFrame (“KF”) is fairly specialized. It does not try to provide the universality of very general frameworks, such as Sun Microsystems’ Java Server Faces or Apache Struts or Tapestry – frameworks that can be used to develop anything from data input applications to interactive multimedia to portal-style applications. KF focus is narrower, aiming at database-centric application with rich data model, extensive business rules and workflows, and tight centralized role-based security model. This is the type of applications that, several years ago, would be developed using tools such as Oracle Forms or Sybase PowerBuilder.

The purpose of this document is to introduce Java and Web developers to KF, and provide practical help and guidance to developing KF applications. We will start with an overview, terminology and explanation of the main parts. As a part of that, we’ll go through the process of developing a simple application end-to-end.

Following that, we will describe the main building blocks of the application as available today “out of the box”, and later we’ll describe options and methods for extending and customizing the framework. We will conclude with a discussion about possible future development directions.

KnowledgeFrame is neither a ready-to-use business application, nor a completely open platform supporting “any” Java project. It assumes that an enterprise data model is in place, and that the development follows patterns typical for data-centric enterprise application. While these default patterns can be changed and overridden when needed, the standard – and most productive – use of KnowledgeFrame is to re-use the accumulated project experience, not to try to reinvent the wheel on each project from scratch.



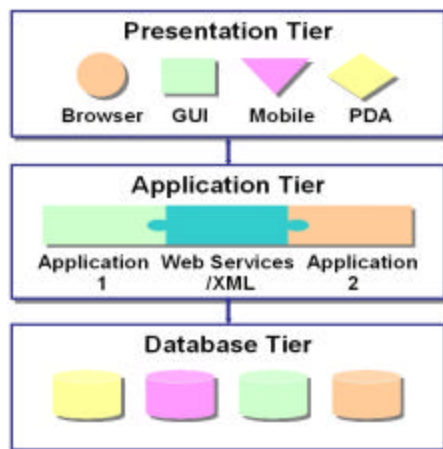
KnowledgeFrame is optimized for an iterative development approach. Experience from many projects shows that abstract notations and conceptual models are excellent tool for communication within a skilled technical team; however, they do not work very well for communication with business users. This includes traditional entity-relationship diagrams, process flows, business function hierarchies, and most UML diagrams (including the highly celebrated use-case diagrams and scenarios). Business users respond much better to live screens and reports. On many projects, the business requirements are finalized only when users can test live screens and validate their requirements and expectations interactively.



2. KF Overview

2.1. Architecture Vision

The following schematic emphasizes the separation of the logical application tiers; presentation, application logic (including rules) and data. The abstraction of components in these tiers is accepted across the IT industry as a best practice to achieve the benefits previously outlined. Within each tier components are appropriately abstracted and provide consistent interfaces to also avoid excessive interdependence.



This type of application architecture is sometimes referred to as “Model – View – Controller”, or MVC. It provides clean separation between the data access layer (“Model”), pure application and business logic (“Controller”) and user presentation (“View”).

A finished enterprise application is a complex system of highly inter-related components, developed on multiple levels over a period of time – its components cannot be explained or built in a strictly sequential manner. To introduce KF to the reader, we will present major separate conceptual layers and building blocks in a simplified sequential list form, simulating the process of accessing the finished application by the business user, “from the outside, drilling down into the details”. Presented this way, a user will come into contact with the following layers or components:

- Application Site
- Login and Security System
- Main Menu and a Hierarchy of Modules
- Modules, Pages and Blocks
- Blocks and Actions

-
- Entities and Associations
 - Business Rules
 - Application Logic
 - Custom Components and Behaviors
 - Runtime Infrastructure
 - Development Infrastructure (*not really visible to the end user*)

Individual layers and components are discussed in the section Main Components.

2.2. KF Application Design

KnowledgeFrame is optimized for projects which follow a declarative development paradigm, and use an iterative approach during design and build (emphasizing feedback from business users). Frequently, the development process starts from 2 starting points – a Data Model (expressed as a relational data model e.g. via ERD diagrams, or as a UML persistent model) and a Business or Functional Model (expresses as a Business Function Hierarchy, or via Use Cases and Scenarios). Actual KF components are then defined based on the interaction between the Data view and Business Function view of the application. In the initial phase, it is highly advisable to pre-fabricate the overall application structure from pre-built KF components (*“looking at the forest and not at the trees”*). All basic components can have many aspects of their appearance and behavior customized via settings (*“meta-data”*), without any programming.

As the overall application structure is established and stabilized, the development proceeds via

- increasing customization of components (either declaratively, or by overriding default component behavior in application-specific subclasses)
- defining Business Rules and attaching them to components
- writing custom application logic and attaching it to components

Rules are used to implement data quality and presentation requirements. A Rule can reconfigure almost any aspect of an Application or Business Component. Meta-data is cached in the application as Java Classes with full set and get method support, allowing the majority of System Requirements to be implemented using a relatively small number of existing KnowledgeFrame Classes. Any requirements which cannot be met using the base KnowledgeFrame Class can generally be achieved through a Rule.

Most application code is written Java and developed in a Java IDE (T4Bi developers use Eclipse 3.0 or JDeveloper 10g, but other popular Java tools should be useable as well) and inherits from its appropriate Knowledge-Frame Class, extending or overriding behavior as appropriate. The custom code is linked into the Application declaratively through the KF Design Utilities.

The Java classes are built in a way that promotes:

- extensive compile-time checking for consistency and dependencies
- extensive start-up time data consistency checks (most major components are test-built and their association “traversed” when the application is started for the first time, reducing the need to re-test all modules after structural changes)
- coexistence between application custom code, and classes that can be re-generated from design tools, database schema and data dictionary, or various meta-data repositories

The main high-level development steps for a KF application are as follows:

- Develop a set of data and functionality requirements for the application. Unlike traditional “waterfall” tools, KF supports and encourages an iterative development approach.
- Develop a physical database model for persistent data used by the KnowledgeFrame application (e.g. in Oracle Designer or a UML tool with support for relational persistence).
- Define a set of Business Objects, mapping the data model to the Java object world. Implement those Business Objects as KF Entities, Attributes and Associations.
- Break down the application functionality into a set of Modules. The initial set of Modules can follow the outline of the Business Functions Hierarchy in Oracle Designer, or the groups of Use Cases and Business Scenarios in UML analysis. Modules are organized into the application Main Menu, which is (by default) a hierarchical structure. The main menu is accessible from a central navigation point of the whole application (called the Home Module).
- Each Module contains a set of screens, process steps, interactions etc. that will be manageable for the development team to implement, yet intuitive for the business users to use. Developers will find it difficult to develop, test and maintain a module containing more than 20-30 pages; at the same time, most business users will not be willing to accept an application implemented as hundreds of single-page modules.
- For each module, define a set (or sequence) of pages, and design mapping of pages to Entities. In KF terminology (similar to Oracle Forms)
- Define the authentication, authorization and security strategy, and implement it in the Login Module. Ensure that the Login Module links each user with appropriate application Roles. Once the Login Module is executed successfully, it will pass control to the Home Module, and will apply the application Roles to the Module and Entity structure. This may restrict access to some Modules or data.
- Test and refine the application, using mostly standard KF building blocks. Aim to stabilize the data model and data mappings.

- Develop a set of Business Rules and/or custom algorithms, executing either on the data level (Entities, Attributes, and Associations) or on the functional level (Modules, Pages, and Blocks).
- Develop programmatic interfaces, integration points, and customizations for all application components, including the user interface, navigation, presentation etc.

2.3. Main Components

This section describes the framework structure, and shows how the main elements are presented in the development environment, and in the running application.

2.3.1. Application Site

A KF Application Site is a runtime image of one or more KF applications, deployed on a database (typically Oracle, single-instance or RAD) and a J2EE application server (or a cluster of load-balanced application servers connected to the same Oracle database). An Application Site is identified by a single J2EE deployment file (a *WAR* or *EAR* file), and a single Web URL for user login. Simple deployments typically support one business application per application site; however, it is possible to deploy a family of related applications into a single Application Site, and make them available e.g. as sub-trees in the Main Menu.

An Application Site typically presents a common look & feel, by sharing presentation style files (Cascading Style Sheets, icons and graphics, static text), and must have shared security (single login and Party / Role registry). In addition to KnowledgeFrame modules, the Application Site may contain static contents and integrated 3rd party code.

A KF application user understands the Application Site as a URL that must be used to navigate to the application(s) login page.

2.3.2. Login and Security System

As a framework for enterprise applications, KF puts a lot of emphasis on party identification and role-based security. Every user interacts with the application via one or more individual sessions, which must always be linked to a particular Party ID. *(Should parts of applications be open for public access, the recommended technique is to create a generic Party ID in the party Registry, e.g. "public", and customize an alternative login page to make the login process invisible to the visitor).* It is expected that the Party Registry will assign one or more Roles to each Party ID, as managed by the application administrator; simple applications may just use one standard Role for everybody.

The Party Registry is a separate mechanism, driven by customer's corporate standards. KF comes with a simple default implementation, which supports the definition of parties, encrypted passwords and associated rules in a set of simple SQL tables. Organizations which already have common login and security standards, based on e.g. native Oracle database login, LDAP or Windows Active

Directory, may develop simple KF login adapters via a custom login page. Once the Party ID and list of application roles is established during Login, KF will maintain that information (independently of a login mechanism) in a special object, Session Context (SCX), which is the main runtime internal representation of a user's session.

A KF application user will see the Login and Security System as a standard Login Page, prompting the user for username and password (and optionally other information, such as preferences, user domain etc.). Side-effects of this component may be visible elsewhere in the application, e.g. in the form of a User Preference or Password Change Modules. Also, certain application behaviors are driven by the Login and Security System – for example, the user session will time out after a predefined period of time (typically 30 or 60 minutes), to protect the database resources from being blocked by users who “walk away” from their session without completing a Logout.

2.3.3. Main Menu and Hierarchy of Modules

A typical KF application consists of many separate functional modules, which can be called independently or organized into a workflow. A Module is a collection of Web pages (and other, more detailed, structures) that implements one atomic business function, or can act as a discrete step in a business process. Breaking down application functionality into Modules is a design and business preference issue – we have applications implemented as one really complex module, and we have applications implemented as more than a hundred discrete modules.

When the user successfully completes a Login and has the Session Context (SCX) established, the next step is a special module called the Home Module (or “Main Menu”). Its only purpose is to handle navigation to all the applications Modules, and providing an equivalent of “Home Page”, where the application Modules will return when they are done with their work. The default implementation of Home Module is a collapsible / expandable hierarchical menu system, which presents individual Modules organized into one tree, with an arbitrary level of sub-trees and branches. *An Application Site may choose to customize the Home Module to present a different visual paradigm, e.g. in a portal-like arrangement.*

KF provides the necessary tools to have the Home Module behavior controlled by common security based on Party ID and Roles. The default implementation (“Main Menu”) uses that feature, by allowing all Modules to specify their security requirements (via Role names, and permissions for individual Roles). A Module or sub-tree of the Main Menu is visible only if the current user has at least one of the Roles required by the Module or sub-tree, and the Role can still put additional restrictions on the usage of the Module (e.g. allow just read-only access to data).

2.3.4. Modules, Pages and Blocks

A design of a Module typically starts by cross-referencing a Business Function against a set of data objects (“Entities”, see below) needed to accomplish the business functionality. Given that the Web standard mechanism for presenting information is a set of stateless HTML pages connected via navigation elements (URLs or form submit buttons), a Module is always implemented as a set of dynamic page templates with corresponding navigation actions. Each Page has

static contents (at least header, footer and graphical outline). The dynamic data-driven content of pages is provided by placing Blocks into specified page locations. All inter-page navigations are triggered via Actions, represented as HTML control elements (URLs or buttons) in Blocks, or in the Page outline. *(Both Blocks and Actions are discussed in more detail in the next section).*

KF provides a generic mechanism to provide practically any page layout (as long as it is supported by HTML), but interestingly enough, all KF applications so far have been developed using only two Page templates:

- **Base Page Layout**, in which all Blocks are simply arranged vertically below each other. Each Block has the total width of the browser window available, and the vertical position of the next block (if any) is determined based on the height of the previous Block. This page arrangement is popular because of its simplicity, flexibility and efficient use of space (especially if the Block contents is more complex and “busy”).
- **Navigation Page Layout**, consisting of a dynamic collapsible navigation tree on the left, and a detailed single-block presentation on the right. This page arrangement is very powerful for presenting complex multi-layer or hierarchical data structures in an intuitive way, resembling e.g. Windows Explorer or many design tools. The navigation tree can be expanded and collapsed, and the Navigation Page implementation provides a set of productivity utilities and shortcut (built-in simple overview reporting, duplicate functionality, automatic lists of items in sub-categories etc.).

In addition to a collection of Pages, a Module provides basic navigation and security infrastructure, allowing it to be invoked from either the Main Menu or from other Modules, and to synchronize its security requirements (e.g. list of Roles allowed to see and use the Module).

When invoked from the Main Menu, a Module always first presents a default Page first. All subsequent navigation within the Module is done via the Page or Block Actions.

When one Module calls another, it also has the ability to navigate directly to a specific Page (Block). Most meaningful Pages and Blocks within the Module (other than the default Page) typically require certain “application context” to execute properly – e.g. a detailed page with a list of employees may require the department number to which all the employees belong. That’s why navigation from another Module also has the ability to pass “application context” information from the caller.

When a Module exits back to the Main Menu or to the calling Module, it will clean up all resources used by the Module (cached objects, database cursors etc.). If the Module is returning control to another calling Module, it also has the ability to pass resulting values back to the caller.

2.3.5. Blocks and Actions

A **Block** is defined as a class that presents one data access object (Entity, see next section) in the application’s user interface, and provides application logic and

business functionality for that presentation. The concept and implementation of Blocks is critical to the whole KF development philosophy, and it is fair to say that an overwhelming majority of all development effort on a typical KF project is in the area of Blocks and their Entities.

Dynamic data in KnowledgeFrame is presented via Blocks. Most frequently used Blocks are data-centric Blocks, built to present one or more data records from the database; other built-in Blocks are on Login Page, in the Main Menu, to present data in non-HTML formats (e.g. Microsoft Excel or Adobe Acrobat). In addition to built-in Blocks, many applications developed custom Blocks for specific purposes, such as performing interactive data analysis, or performing geo-spatial analysis via navigable maps.

Each Block is uniquely linked to exactly one Page, and if the Block is data-related, it is linked to exactly one Entity. An Entity, however, can be used by more than one Block (in the same or different Modules). It is also possible to define Blocks which are not Entity-based – either static, or having their data access completely custom-written.

The built-in Blocks most frequently used by developers as building blocks (*no pun intended*) in KF applications include:

- List Block, presenting a result of a database query as a read-only list of records, arranged in a grid format (one record per row). The built-in features of the List Block include filtering by any criteria, column-wise sorting, scrolling of results of long queries N records at a time, control over presentation features, and the ability to automatically export data to Excel. This Block is typically used for data searches, browsing, navigation, and for simple reporting purposes.
- Edit Block, presenting a single record at a time in a form-like presentation style, as multiple rows (and one or more columns). The Edit Block is the main tool for data entry and updates.
- List Edit Block allows data entry and editing for multiple records in a grid-like format, same as List Block. It makes multi-record data entry faster and more productive, as long as the width of data in the entity allows entering all the data on a single row in a standard Web browser.

An **Action** is a visible control element (URL or button) which triggers a specific piece of functionality, built into KF or custom-written for the application. Actions exist to save and commit data of the current Block to the database, to navigate to another data record, Page or even Module, to execute data search or value lookup, to run reports or export currently selected data to Excel, to execute specific algorithms, or even to launch other applications.

A majority of Actions in a typical application is defined on the Block level. Some Actions can be defined on a single-record level, or even on a data element level (e.g. the built-in Lookup Action, implementing a foreign-key-based lookup). It is also possible to define Actions on the Page level (by default, built-in actions “Return to Main Menu” and “Logout” are defined on the Page level).

A Module does not have its own user interface (the user “sees the Module” through its Pages), but the events of entering or exiting a Module can be used to attach Action-like functionality.

2.3.6. Entities and Associations

The concept of Entity is central to the design of all modern multi-tier system, as a principal unit of mapping of persistent data into the runtime application objects. For data-centric applications (which is what most enterprise applications are), this is the critical point of the whole system design. Different frameworks and vendors use slightly different terminology for the same concept - Business Objects, Data Objects, Data Access Objects (DAO), Entity Beans, Java Data Objects, Java Business Objects, Persisted Classes, etc. – with wide differences in technical implementation. The common ideas and pattern in this concept include:

- Mapping of one specific object or pattern in the persistent database
- Encapsulating access to the object and individual instances (“records”) and fields via a programming interface
- Providing additional information and control about the data, via meta-data or via programmatic tools
- Linking data-centric business functionality and data-centric business rules to data access, frequently with the goal to guarantee data consistency and data quality (“no information is allowed to be stored in the database unless it meets all of the following rules ...”)

KF has selected a data access and encapsulation technique modeled on top of the standard Java SQL interface (JDBC), with a heavy emphasis on runtime metadata and business rules. This selection was motivated by several factors:

- KF applications are targeted at non-trivial, non-literal data models, beyond default mapping to physical SQL tables or views. A KF entity can be based on an inner or outer join, or use more advanced SQL constructs in its definition (sub-selects, “WHERE (NOT) EXISTS”, CONNECT BY hierarchies, optimization clauses, functional indexes etc.). To be able to do that, the Entity definition needs full access to all database native SQL constructs.
- In enterprise data models, a certain level of abstraction is typically used – many entities are not stored as literal tables, but are embedded in supertype tables, or constructed on the fly from joins, name-value-pair tables or even external data sources. To be able to interpret specific data, raw data access needs to be coupled with sufficiently rich meta-data, which provide business-specific context to more generic tables.
- The Java community is slowly moving towards a consensus on the best technique for object-relational mapping. The implementation of the originally proposed standard, Enterprise Java Beans, is infamous for excessive complexity and unacceptable runtime performance overhead. Sun’s attempt at a developing a competing standard, JDO (Java Data

Objects), or comparable efforts by the open-source community (e.g. Hibernate) are only slowly gaining industry acceptance. New proposed versions of the standards, EJB 3 and JDO 2, are supposed to fix the current situation, but as of the time of writing this document (September 2004), none of them is ready for immediate acceptance. In that situation, JDBC represents the only universal standard available today (and it will remain a technology layer underlying EJB and JDO, if and when they will be universally accepted).

The JDBC orientation is quite apparent in the design of the KF data access layer, and in fact, a good familiarity with JDBC concepts tends to help developers understand the mechanics of KF Entities. The terminology KF uses in this area should be familiar to analysts and designers with either an Entity-Relationship Modeling background, or using a UML tool with relational-persistence-modeling capabilities:

- An **Entity** is an abstraction of a persistent data object, e.g. a SQL table or view. An Entity must always have a primary key, and be based on a set of records in the database. In the JDBC world, the closest equivalent to an Entity is a Prepared Query (or “database cursor”) – in fact, KF defines a Query object derived from an Entity (a Query is a view of the Entity, with optional additional WHERE and ORDER BY clauses; more than one Query can be derived from the same Entity). An Entity can be updateable, in which case the underlying SQL statement must directly map to physical data columns, or the Entity implementation must provide its own implementation of the methods insert(), update() and delete().
- An **Attribute** is an abstraction of one database column in the Entity. Built-in attributes are typically built on SQL and Java primitive data types (integer, float, timestamp, character string), but custom-defined Attributes can be defined as well (e.g. abstractions for various types of surrogate keys, spell-checked strings, multimedia and external document types stored as CLOB or BLOB etc.). A type of an attribute is related to the concept of Domain, which is a class specifying a type of an Attribute. In the JDBC world, an Attribute is represented by 1 bind variable
- An **Association** is an abstraction of a relation between 2 Entities, such as a Foreign Key. An Association is a meta-data-only object. It is usually projected to the JDBC world via additional WHERE clauses to SQL statements being executed.

At runtime, these definition data structures lead to runtime-only objects being created, including Query (similar to JDBC Prepared Query), which – when executed – creates a Data Row Set, which is a collection of Data Rows.

Of all KF classes, the richest set of meta-data values and methods is associated exactly with Entities, Attributes and Associations. Given the data-centric nature of KF applications, a lot of functionality and most of data-integrity and data quality is accomplished via meta-data and business rules associated with Entities, Attributes and Associations.

2.3.7. Business Rules

Business Rules are classes attached to KF application objects (Modules, Pages, Blocks, Entities, Attributes and Associations) which:

- Are executed as a response to specific events (e.g. "BEFORE DATABASE INSERT")
- Are autonomous and physically independent of other rules (are "atomic")
- Are designed to meet one specific goal (e.g. "ensure that each inserted record meets certain criteria" or "create an audit record")
- Can succeed or fail (with varying degrees of severity), and by failing, they can influence other rules or built-in functions (e.g. other rules and functions will not be executed if the rule failed with a severity higher than "warning")

The full set of events currently built into KF includes:

- Create a New (empty) Data Record
- Database Query
- Database Insert
- Database Update
- Database Delete
- Browser (HTML) Screen Paint
- Process HTML Results Returned from the Browser
- Value Lookup
- Enter a Module
- Exit a Module
- Exit a Row

Other events can be added to KF as needed.

In release 5.1, KF supports both a declarative approach and a programmatic approach to Business Rules.

A **declarative approach** is a more traditional approach, which encourages encapsulation of rules in separate Java classes, and attaching those classes to KF objects (Modules, Pages, Blocks, Entities, Attributes, Associations) to execute before or after specific events (e.g. "after database insert", "before screen paint" etc.).

A **programmatic approach** has been developed recently, to address complex business scenarios, where we need to resolve a nested or non-linear set of dependencies (“if A or B but not C, or D and (exists at least one E where ...)”).

KF contains a basic library of pre-built business rules, directly useable for the declarative approach. These rules typically deal with rule pre-conditions based on attribute values (comparing 2 attributes, or an attribute as a constant, using basic operations, like “equal”, “greater/less than”, “like”, etc.), plus a resulting action (setting a data or meta-data value, generating a warning or an error etc.). The application can develop more specific rules by extending the pre-built Rule classes, or can use the programmatic approach to implement essentially any Java expression or algorithm.

2.3.8. Application Logic

In KF application, the overall structure of the application is typically defined by the KF methodology. Specific application code can be attached to individual components (Modules, Pages, Blocks, Entities, Attributes, Associations, and Rules). Even if a complex functionality exists as a separate package or library, it tends to be incorporated to KF via adapter classes or methods, and attached to a convenient point from which it is always invoked.

For example, a spell checking application (developed by a 3rd party, Keyoti Systems) is attached to a specific sub-class of Attribute, called “Spell Checked Attribute”. When that Attribute is shown in the User Interface, a spell-checker control is displayed as well, allowing the user to start the algorithm.

A great majority of application logic in existing application is implemented as subclass-methods or Business Rules attached to Entities, Attributes or Blocks. There is some application logic in several Modules and Associations, but these situations are relatively rare. This does make sense – in a highly interactive application based on dynamic data, most activity occurs when a user interacts with a data object (Entity) via an application object (Block).

In an enterprise application methodology world, there is an ongoing discussion about the merits of a model-driven development, versus custom (coded) development. In a real world, we have seen a need for both – a “high-level picture” of the application needs to be synchronized and coordinated from a data and application modeling tool, but there are enough local specifics and custom “twists” in every non-trivial application to make 100% generation impractical.

KF resolves this problem using Java inheritance. *(As of the time of writing this document, this technique has been used for Entities, Attributes and Associations, where this phenomenon became a problem in real life. In theory, we may want to do something similar for Module, Pages, Blocks, Rules and the Main Menu – however, none of our existing projects have seriously needed to continue re-generating in parallel to manual updates, and the technique has not been applied to those objects yet).* The KF Entity implementation recommends one Entity to be generated as one class. Using a naming convention, the class which is generated (from a design tool, from a 3rd party meta-data repository etc.) is called **XXXEntity** . In addition to that, a subclass (which may initially be almost empty) is, by the same naming convention, called **XXXEntity**, and that class is actually used by the

application. If we need to re-generate later on, we re-generate only the *EntityDef* classes, and all re-generated features that are not overridden by custom code are immediately useable. All custom and application code goes to the Entity classes. In case the re-generated code does any change that creates a conflict with the custom code (e.g. removing a column referenced in the custom code), the Java compiler will alert the developer right away.

2.3.9. Custom Components and Behaviors

The dominant methods for implementing custom components and behaviors are taken straight out of the Java textbook – **abstract classes** and **inheritance**.

KF uses **abstract classes** whenever 2 software layers need to interact. For example, when a Main Menu invokes a Module, it does so via an abstract definition class, `com.t4bi.kf.controller.module.Module`. Any class inherited from than class will work in this context.

Although we typically build Modules from KF Pages and Blocks, the **Module** class does not specify any of these – so, it is possible to write a Module subclass which uses a different internal structure (e.g. calling a 3rd party application), and the interface is expected to work.

In the same way, all **Pages** in the Module are inherited from `com.t4bi.kf.controller.page.Page`. Although we use pre-built pages today, it is easily possible to define its own subclass of Page (which may or may not used Blocks in its presentation), and use it for any customized presentation purposes.

While overriding Modules and Pages remains an intriguing theoretical possibility, we actually have a fair bit of functionality for custom-written **Blocks** today. A good example is the Microsoft MapPoint interface that is used in one of our systems. Most data-based Blocks present database data in a form layout, or a grid (matrix) layout. The MapPoint Block simulates the form layout Block with 2 numeric values (Geographical Longitude and Latitude values of a point on the map – in this case, a real estate listing), but instead of showing it as 2 numeric fields, it calls the Microsoft MapPoint service to show a map segment with the immediate neighborhood of that location as a JPEG file, and allows the user to adjust the location using a mouse. The final location is then stored in the database as numeric value, as if the user would type in the numeric values in an Edit Block.

Overriding **Entities** is quite common, and KF itself provides a good example – a “Name-Value-Pair-Entity”. While a regular Entity stored values in specific table columns (“COLUMN1”, “COLUMN2” etc.), a Name-Value-Pair-Entity stores values in a more generic data structure, a simple table that has (at least) 3 columns:

- Parent ID (representing the PK of the “current record” where the current record represents the record that this value is associated with)
- Name (e.g. “COLUMN1”, “COLUMN2”) stored as a character string
- Value

While the mechanics of “SELECT”, “INSERT”, “UPDATE” and “DELETE” are completely different from a regular Entity (which means that methods like insert(), update() and delete() are overridden in the sub-class), the usage of this Entity in a Block is almost identical to a regular Entity, and the rest of the application is thus shielded from the Entity implementation details.

There are specific Attributes that can be customized as well – see the spell-checker example in the previous section.

2.3.10.Runtime Infrastructure

KnowledgeFrame protects developers from low-level dependencies on the runtime infrastructure. There is a specific set of KF packages, below **com.t4bi.kf.runtime** (and, to a lesser degree, **com.t4bi.kf.view**, which hides *some* of the HTML painting and result-processing dependencies), where all runtime dependencies are supposed to be handled. When additional dependencies are discovered on a project, the preferred method is to coordinate with the core KF5 development team, and add the new features to the core KF5 framework, to benefit all projects.

For example, the requirement to handle the additional requirement of deployment to a cluster of middle-tier servers with load-balancing capabilities (via adding support for “arrowhead cookies”) has been handled this way.

It is technically possible to subclass the KF5 runtime infrastructure classes as well – however, even in this case, a certain level of participation of KF5 developers (at least in an advisory role) is recommended.

In general, this area has been remarkably stable over the last several years (thanks mainly to the stability of the Sun servlet interface standard, and Oracle “thin” JDBC drivers), and changes in this area are an exception rather than a rule.

The Deployment section provides additional details about version requirements and deployment procedure for KF proper and for static contents accompanying KF.

2.3.11.Development Infrastructure

As of the time of writing this document (March 2005), the mainstream target Java version is 1.4.x, and the supported target Oracle version range from 8.1.7 to 10g. Any development tool capable of supporting J2EE development and test-deployment for these versions (with a complete runtime infrastructure – servlets, JSPs, XML parsers, Oracle thin JDBC drivers) can be used to develop and test KF5-based applications. The tool should be capable of live debugging and performance profiling on the selected server platform.

On existing projects, the most popular development environments include Oracle JDeveloper 10g and Eclipse 3.0. Many development, packaging and deployment tasks are accomplished via the ANT utility (version 1.3 or higher is recommended).

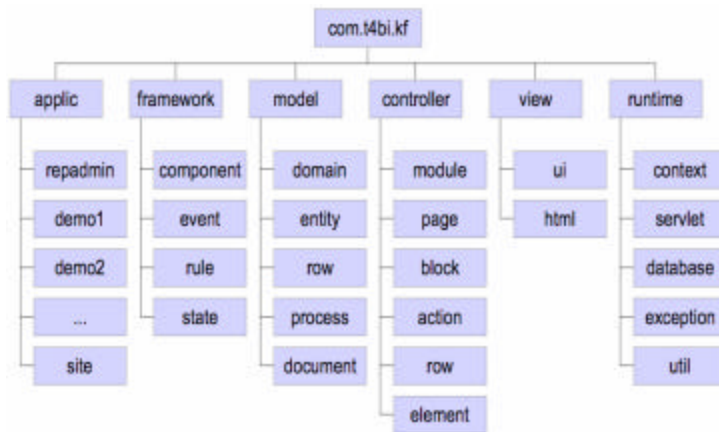
The reference server platform for deployment and testing is Tomcat 5.0 (which is the official J2EE reference platform for Java servlets), and an Oracle thin JDBC driver certified by Oracle on the target JVM and Oracle database version. Most of today’s servlet/JSP engines are highly compatible with Tomcat, and the portability

of applications is excellent. KF5 applications are currently deployed in production or training scenarios on Tomcat, Oracle iAS (or OC4J) and BEA WebLogic, and have been tested on IBM WebSphere.

3. Java Packages and Classes

This section presents a summary and organization of KnowledgeFrame Java packages and classes – mainly as a navigation aid to the following sections. This diagram and text explains the KF5 packages as available in the application framework as provided by T4Bi. While the developers of an application can follow their own package and class naming standards, T4Bi recommends mirroring the structure of KF5 in the hierarchy of application packages and classes as well. For example, if an application class inherits from (or meets the same purpose as) a class in **com.t4bi.kf.applic**, then it is recommended to put that class into a package

<company_name>.<application_name>.<application_component_name>.applic. For overall readability, it is also helpful to use the base KF5 class name as suffix in the application class name – for example, an application-specific class inherited from **com.t4bi.kf.model.entity.Entity** should be called **<Application_Specific_name>Entity** .



3.1. Site and Application Definition

This is the key package for defining Application code. All Application Objects and Business Objects for the application should be defined here, and use the functionality of the other packages.

- Implemented as **com.t4bi.kf.applic**
- Application definition. Default KnowledgeFrame distribution includes only utility applications (e.g. the module to edit the menu system and assign security) and demo applications (e.g. “demo0X”).
- The sub-package **site** contains modules. Menus, and configuration information needed to run the whole application site, especially the Login Module and Home Module (“Main Menu”).
- On a real project, the customer will mirror this directory with a project-specific directory (e.g. com.acme.sales.applic), containing sub-packages for actual application modules.

- If the project needs to override the default framework functionality, it will mirror the KF directory structure for overriding classes (e.g. com.acme.sales.view.ui)

3.2. Framework

This package contains definitions of key concepts, semantic classes and meta-meta-data classes, to provide a common baseline and “vocabulary” for the whole framework and application. An important part of Framework is the definition of User Actions, Business Events and Business Rules, and their connection to Web, HTML and Java runtime technology. The application will rarely need to extend or implement its own conceptual and semantic classes; however, it may need to add application-specific types of Actions, Events and Rules to those supplied with KnowledgeFrame.

- Implemented as **com.t4bi.kf.framework**
- Main entry point to the KF framework; contains definitions of the main objects and meta-classes. All other packages in the framework use these definitions.
- Package **component** – basic and common KF definitions, especially Component Type, Component and Component Allowed Value; the essential Components supported by KnowledgeFrame are Business Components (data-centric objects bound to SQL database objects) and Application Components (functional components visible to the users).
- Package **event** – object or application Event definition; the main purpose of Events is to enable binding of Rules to Components.
- Package **rule** – a common superclass for all KF rules.
- Package **state**– support for the definition of object states, and generic implementation of a state machine. *(Not used in the current KF version).*

3.3. Model

This package contains the persistent database layer of the MVC paradigm. The application may require encapsulation of certain data access patterns (e.g. decomposing a database record into a Name-Value structure, or implementing a custom-coded join algorithm).

- Implemented as **com.t4bi.kf.model**
- Implements the concept of Business Components. Handles mapping between the KF runtime instance and persistent database data. Supports both business object level functions and instance (row) level functions
- Package **domain** – common domains (“data types”) definition. In the most recent KF version, the Attributes of Entities (“columns” of “tables”) are actually strongly typed using Java classes in the package **attribute**, and **domain** provides common type and attribute behaviors to **attribute** classes.
- Package **entity** – business object level functions (meta-data definition, physical database mapping, binding of data and rules) – Entity, Attribute, Association. This is the core mapping of the persistent data model, with Entity being the middle-tier

representation of one or more database tables (or equivalent data structure), Attribute is the mapping of a database column (or similar field), and Association represents a logical data link between Entities, including not only Foreign Keys, but also more complex associations (e.g. many-to-many).

- Package **row** –row-level functions (Queries, cursors, individual rows and fields access, CRUD operations). SQL physical data mapped by classes Data Row and Data Element in this package into Java memory are used by the corresponding package row in the Controller to build application and presentation behaviors for that data.
- Package **document** – dynamic business documents. *(Not used in the current KF version).*
- Package **process** – business processes and workflows. The application “Main Menu” is modeled as a hierarchy of Application Processes.

3.4. Controller

This package contains the application logic layer of the MVC paradigm. This package will be extended by most non-trivial applications, mostly implementing reusable or self-contained application logic, such as application-specific Business Rule classes.

- Implemented as **com.t4bi.kf.controller**
- Implements the concept of Application Components. Handles application functionality and control flow.
- Package **module** – definition of main application building blocks, with support for full lifecycle, similar to Oracle Forms FMB file, or Jakarta Struts Application, and Application Component Associations
- Package **page** – definition of a display or print page
- Package **block** – definition of one application functional block, which is mapped to one model Entity at any point of time – similar to Oracle Forms Block, or Jakarta Struts Form
- Package **row** – has the main class Row, and controls how 1 row of data (provided by Data Row class in Model) is presented in a block (e.g. an Edit Block has 1 Row, while a List Block or Lookup Block have many Rows). This package also contains the class Element, which provides an application support for one element (“attribute”, “column”) of a Row, based on data provided by the corresponding Data Element object in Model.
- Package **action** – support for User Interface elements, such as buttons, URLs, programmatic controls etc., which initiate application activities and state changes. Actions are typically bound to visible Application Components (e.g. Pages, Blocks, Rows etc.).

3.5. View

View is the user interface layer of the MVC paradigm. If the application requires highly customized layouts, browser features, support for plug-ins etc., the necessary functionality will be extended in this package.

- Implemented as **com.t4bi.kf.view**
- Handles user interface, presentation functions and behaviors
- Package **ui** represents a “User Interface Engine”, a common definition of presentation and user interaction functions
- Package **html** is the implementation of the User Inter-face Engine for the Web browser (HTTP, HTML and JavaScript) interface

3.6. Runtime

This package contains runtime technical components. An application may have specific technology dependencies, such as access to legacy data source via custom code. These technology components must be encapsulated to be transparent for the rest of KnowledgeFrame code.

- Implemented as **com.t4bi.kf.runtime**
- The runtime engine of KnowledgeFrame, handling all system and technology dependencies.
- Package **context** – all data specific to a runtime session for 1 user, starting with the SCX object (based on the HTTP Session standard)
- Package **servlet** – support for J2EE “servlet” and “HTTP servlet” interface, including Session Manager
- Package **database** – database connection and cursor handling, built around the JDBC standard
- Package **exception** – common KF exception definitions and handling, both KF-specific and implementation-specific

3.7. Rule Engine Framework

The KnowledgeFrame Business Rule framework implements a context-and-event-based rule concept, for all major components of the framework. This means that the definition and execution of each rule is controlled by the following:

- Rule Context - the Components for which the Rule is defined, associations of this Component (e.g. parent Components), and runtime context for this Component (status, user’s roles etc.)

- Rule Event – the Event that takes place for the Component for which the Rule is defined, and which triggers execution of this Rule
- Sequence – a predefined “firing sequence” within the list of Rules applicable for a particular Component and Event
- Rule Type – a definition, or “algorithm” which implements the rule; it can be predefined (standard) or custom-written for this application
- Timing – whether the rule executes before or after processing the Event

The Rule Engine is composed of the Rule Framework and the Rules themselves, and exists in 2 implementations – a **declarative approach** (suitable for building common libraries of pre-built rules, called sequentially upon execution of an event), and a **programmatic approach** (suitable for complex business scenarios with a nested or non-linear set of dependencies).

3.7.1. Declarative Approach

KF contains a basic library of pre-built business rules, directly useable for the declarative approach. These rules typically deal with rule pre-conditions based on attribute values (comparing 2 attributes, or an attribute as a constant, using basic operations, like “equal”, “greater/less than”, “like”, etc.), plus a resulting action (setting a data or meta-data value, generating a warning or an error etc.). The application can develop more specific rules by extending the pre-built Rule classes, or can use the programmatic approach to implement essentially any Java expression or algorithm.

In the Declarative Approach, KnowledgeFrame separates the definition of Rules from their association to Components. The Framework is therefore independent of patterns used to define particular classes of Rules.

All major KnowledgeFrame Components (including Entities, Attributes, Associations, Modules, Pages, Blocks, sys-tem objects) are built to recognize sets of events meaningful for their particular Component type. The application designers and developers can predefine and raise application-specific events (business events, or system events) for each Component. The runtime engine notifies the Component when one of the registered Events takes place for the Component, which allows the Rule Framework to “fire” all Rules defined for this Component and Event, in their proper sequence.

When the Rule Framework triggers the execution of the Rule, it passes the execution context information to the Rule in the form of Rule Parameters. Rule Parameters can originate in the Component for which the Rule is defined, or in associated Components, or as a part of the runtime session information (e.g. the list of Business Roles defined for the current user in the application Party Registry).

The rule execution is allowed to read any data identified by its parameters (execution context), modify the Component it is attached to, or to modify data that will not affect processing of this Event (such as inserting a record into an audit table). Once complete, the Rule will return a result code, from a set of predefined values (“severity levels”, described below).

The current implementation of the rule execution flow is strictly linear (all rules execute in sequence, until either a rule signals a failure, or until all rules are satisfied). No forward-chaining or backward-chaining concepts are used at present.

The rule definition starts from a Java class that implements the rule algorithm, called Rule Type in Knowledge-Frame repository. This Java class must be derived from a common rule definition recognized by the Knowledge-Frame Rule Framework – it must be created as a subclass of the “rule pattern” class `com.t4bi.kf.framework.rule.Rule`. This abstract class defines the main interface elements for the Rule implementation:

- A unique name
- The instance of the Component to which the Rule is attached
- The Event of the Component that triggers the execution of the Rule
- The severity level signaled back to the Rule Framework if the rule fails
- The standard error message if the rule fails
- The number and types of Rule arguments

If the rule succeeds, it returns the severity code 0 (“SE-VERITY_NONE”) as an indication of success. If the rule fails, it returns a severity code from the following list:

- SEVERITY_NONE; the rule never fails (or, any results of rule execution must be ignored)
- SEVERITY_INFO; the rule execution may result in an informational message shown next time a HTML page is generated for the user after processing this Event; the Rule Framework proceeds to the next Rule in the list for this Event
- SEVERITY_WARNING; the rule execution may result in a warning message shown next time an HTML page is generated for the user after processing this Event; the Rule Framework proceeds to the next Rule in the list for this Event
- SEVERITY_ERROR; the Rule Framework aborts the processing of the Event, and displays the error message to the user
- SEVERITY_FATAL; not expected to take place under ordinary circumstances; typically should prompt the user to contact Technical Support

The Java coding standards for the Rule classes are no different from other Java classes; the mandatory inheritance of the Rule class from the Rule super-class appropriate for the Component Type will enforce the data dependencies between the Rule algorithm and the corresponding Component and Event.

The Rule Framework does not have any dependency on (or knowledge of) the Rules it will execute, other than that they are:

- Bound to a Component
- Fire before or after an Event

- Have mandatory and/or optional parameters, which the Rule Framework must resolve
- Have outcomes which the Rule Framework must process
- Rule Classification does not have any knowledge of (or dependency on) the content of any Rules.
- The KF Design Utilities provide interfaces for Rules to be defined and assigned to Components, and all their Parameters and data bindings specified.
- All Rules are written in Java and conform to rigid code and behavioral requirements (including inheritance of an appropriate KF Rule Class). Rules can invoke database stored procedures where database level enforcement is required.
- Several classes of Rule are “built-in” to the Rule Framework and enforced transparently. These include fundamental referential integrity, data-type and format rules amongst others. The base meta-data for these Rules is generally extracted from the Oracle data dictionary and then extended through the Knowledge-Frame Repository Administration tool.
- The KnowledgeFrame Rule Framework is responsible for executing each Rule and providing it with all of the parameters it requires (including resolving all references). The Rule itself can be completely data-driven (i.e. the logic is derived at runtime) or contain hard-coded logic or a combination.
- Rule Timings (e.g. Pre, Post, other) are also defined for the Rule Framework.
- New Events can be simply defined, including Timed Events, System Events and Business Triggers based on state changes or external actions.
- New Events and Rules are attached to the Entity, Do-main, Attribute, Process or Application Component (Module, Page, Block, Item) as appropriate.

Rules are attached to an extensible Event Model. Base Classes of Event are:

- Database Events (New Data, Insert, Update, Delete)
- UI Events (Paint, Process)
- Navigation Events (Paint, Process, Enter Page, Exit Page, Enter Module, Exit Module, Enter Lookup, Exit Lookup)

New Rule Classes and Events can be defined, as required, by sub-classing the KF Rule class. These Rule Classes are added to the Rule Framework’s Rule Class definitions and instances can then be assigned for individual Rules.

3.7.2. Programmatic Approach

In a Programmatic Approach, the activation points for each Event, together with their timing, are defined as method stubs in existing Component classes. For example, to support a “Database Insert” Event on an Entity, and Entity class definition now provides 2 common

methods – **beforeInsert()**, and **afterInsert()**. As implemented in the common superclass (Entity), these methods only invoke a set of Rules defined via the Declarative Approach (if any). Any application Entity can override these methods, and put any Java application logic into their bodies. The methods' interface provide complete access to the data, meta-data and session context of the application objects (current Data Row, Entity, SCX etc.), and returns a standard Severity Code when completed.

The developer can implement an arbitrary algorithm here, and can control the rest of the execution via the returned Severity Code. If the algorithm wants to also support rules defined via the Declarative Approach in this situation (which, under ordinary circumstances, it should), it needs to call a corresponding method in the generic super-class. For example, method **beforeInsert()** must call **super.beforeInsert()** to execute declarative rules (if any).

4. Using Standard Application Components

A majority of development effort in KF applications is spent defining and refining the application “image” of the Data Model via Entities, Attributes and Associations, and presenting them to the user via Modules, Pages and Blocks. Even when the developer implements custom processes, rules and transformations, it is typically done in the context of these KF components.

KF allows the components to be either generated from a repository (SQL, XML, proprietary to the design tool etc.), maintained in Java as declarations and program code, or coexist as a combination of both approaches. While a full coexistence between a generated approach and a Java-maintained approach can be applied to any KF component, real projects have used this capability only for the Entities, Attributes and Associations. For Modules, Pages and Blocks, the basic module definition structure is relatively simple – even complex multi-page modules with all pages, blocks and non-standard actions are typically expressed as one relatively small Java file – and coexistence between a design tool (and repository) and Java code was seen by developers as excessive overhead. However, should a future project require this capability, the principles used for Entities, Attributes and Associations can be extended to Modules, Pages and Blocks as well (*and potentially even for the Main Menu, Rule definitions etc.*).

KF terminology for the combination of generated and Java-maintained approach is to distinguish between **structural** and **behavioral** features of each component. The approach which allows **structural** features to be generated, while the **behavioral** features are maintained in Java, is covered in section “Entities, Attributes, and Associations”. The approach where most of the code is maintained in Java is covered in section “Modules, Pages and Blocks”.

When considering these 2 alternatives, it is important to realize that ***maintaining meta-data information in Java does not mean giving up the declarative approach to software development***. Java, like many object-oriented languages, has strong declarative capabilities, enhanced by strong dynamic typing, and it is quite feasible to specify many properties and behaviors in a tabular, declarative format. The benefit of this approach is the ability to maintain both declarative aspects of the application, and procedural code (algorithms, extensions, customizations) from the same development environment, and leveraging Java’s compile-time-checking capabilities.

4.1. Entities, Attributes and Associations

4.1.1. Overview

The basic structure of Entities, Attributes and Associations in Java is reasonably simple, and Java classes are relatively easy to write by hand to generate *once* from an SQL repository or from another tool (several generation scripts are included with KF). The problems with generating or writing Entities and other data-related components are:

- The scope of information on a typical enterprise project is huge – major projects may need to support hundreds of Entities and Associations, and thousands of Attributes

- The data model typically keeps changing after the initial version of Entities has been generated, and after they have been modified by developers (to add specific application rules and behaviors)

The challenge is to be able to re-generate the database-related information, without losing or invalidating the application code changes. To help in this task, KF distinguishes between **structural properties**, which are expected to be re-generated, and **behavioral properties**, maintained by programmers.

While the project may choose to re-align which properties it considers structural and which are behavioral, a certain basic set of properties is so essential to interpretation of the relational data structures that it is always supposed to be structural. This includes:

- Entity names and their mapping to SQL tables
- Entity parent class (*some classes are not built directly from Entity, but from more specialized sub-classes, which re-implement certain methods, e.g. a Name Value Pair Entity, which “pivots” row data into “vertical” name-value pairs instead of literal rows*)
- Definition of Attributes and their SQL mappings (column, SQL and Java data types)
- Definition of Primary Keys and Descriptors
- Definition of Associations, and Attributes forming “Association Ends” (on both parent and child side)

The default KF version also adds the following properties into the “structural” category (mostly based on actual project experience):

- Title
- Essential presentation and data input properties (Mandatory, Visible, Visible In List, Query-able)
- Type-specific properties (length, precision, format etc.)

These additional “structural” properties are easy to override at runtime by behavioral methods.

In terms of actual implementation, under the KF naming conventions, the structural information is expected to be stored in a class called ***initStructuralProperties*** described in detail in the next section. The behavioral properties are coded in the method ***init***. This means that the behavioral method can override any structural definitions.

4.1.2. Defining Structural Properties

The important part of the definition of structural properties in KF 5.2 is the fact that most data in the entity class is defined as **static** or **static final**. These definitions are loaded into memory once, and shared by all users and sessions (and, if an Entity is used more than once in a Module) all usages. That means, if any property is set or modified once, it is modified everywhere.

The rationale for making the main structural static is that in Java, static structures can be resolved at compilation time, and thus can be used for compile-time checks. The Entity classes have a special method, **initStructuralProperties()**, which is called **once** for each Entity when the system is started. All changes in this method are applied to shared data. *(That's why in this method, session-specific data from SCX and dependent objects is not accessible).* The method `initStructuralProperties()` in `XXXEntity` is called first, and controls the whole execution (although, as a "good citizen", it is expected to call the method `super.initStructuralProperties()`).

Method `initStructuralProperties()` typically sets up properties like the primary key, descriptor etc. In addition, this method sets up titles, display types and properties which are expected to be shared between all instances.

When the Entities are actually used, the static data is copied in specific in-memory instances, which can then be modified on a per-session or even per-instance basis, via the **init()** method (or even later, via rules).

When defining structural properties, the attributes are defined via the `Attribute` class (same as the one used at runtime). Association is defined via the class `Association`, which at runtime is actually copied into 2 classes (*Entity Association, Attribute Association – this is to be able to support composite associations*).

All Attributes and Associations defined as static can then be used in other definitions in the same Entity, in other Entities (e.g. as "Association Ends") or in application code. Should the definition be removed or redefined, the Java compiler will alert the developer about any dependencies.

Anything that is not set via **static** Attribute and Association must be defined via `setXXX()` methods defined for Entities, Attributes and Associations, listed in this document and in the classes themselves.

The following code snippets are from a definition of a simple structural class, "Account Types". Please note that the method `initStructuralProperties()` allows the application programmer access to even structural properties.

```
/**
 * Class EssAccountTypesEntity is generated from the KF5 repository,
 * entity ESSACCOUNTTYPES.
 */
public class EssAccountTypesEntityDef
    extends Entity {
    //~ Static fields/initializers -----

    public static final RandomSKAttr attrId = new RandomSKAttr(0, "AT.ID","AT",
        "ID", "Id", MANDATORY, NOT_VISIBLE, NOT_VISIBLE_IN_LIST,
        NOT_QUERYABLE, 20, true);

    public static final StringAttr attrTypeName = new StringAttr(1,
        "AT.TYPE_NAME", "AT", "TYPE_NAME", "Name", MANDATORY, VISIBLE,
        VISIBLE_IN_LIST, QUERYABLE, 50);

    public static final TextBoxStringAttr attrDescription = new
        TextBoxStringAttr(2,"AT.DESCRPTION", "AT", "DESCRIPTION",
```

```

        "Description",NOT_MANDATORY, VISIBLE, NOT_VISIBLE_IN_LIST,
        NOT_QUERYABLE, 2000,50, 3);

public static final StringAttr attrAbbreviation = new StringAttr(3,
    "AT.ABBREVIATION", "AT", "ABBREVIATION",
    "Abbreviation",NOT_MANDATORY, VISIBLE, NOT_VISIBLE_IN_LIST,
    NOT_QUERYABLE, 10);

public static final StringAttr attrPositiveBalanceSign = new
    StringAttr(4,AT.POSITIVE_BALANCE_SIGN", "AT",
    "POSITIVE_BALANCE_SIGN",          "Positive Balance Sign",
    NOT_MANDATORY, VISIBLE,NOT_VISIBLE_IN_LIST, NOT_QUERYABLE, 1);

public static final IntAttr attrOrderSeq = new IntAttr(5, "AT.ORDER_SEQ",
    "AT", "ORDER_SEQ", "Order Sequence", NOT_MANDATORY, VISIBLE,
    NOT_VISIBLE_IN_LIST, NOT_QUERYABLE, 22);

public static final IntAttr attrParentId = new IntAttr(6,
    "AT.PARENT_ID","AT", "PARENT_ID", "Parent Account Type",
    NOT_MANDATORY, VISIBLE, NOT_VISIBLE_IN_LIST, NOT_QUERYABLE, 22);

public static final SpellCheckedStringAttr attrChangeHistory = new
    SpellCheckedStringAttr(7, "AT.CHANGE_HISTORY", "AT",
    "CHANGE_HISTORY", "Change History",NOT_MANDATORY, VISIBLE,
    NOT_VISIBLE_IN_LIST, NOT_QUERYABLE, 4000, -1, -1);

    . . .

    // Keys

public static final EntityKey PK = new
    EntityKey(EssAccountTypesEntity.class,EntityKey.PRIMARY_KEY, attrId);

public static final EntityKey DESCR = new
    EntityKey(EssAccountTypesEntity.class,EntityKey.DESCRPTOR, attrTypeName);

    // Associations

public static final Association assocAtParentFk = new
    Association("AT_PARENT_FK",EssAccountTypesEntity.attrParentId,
    EssAccountTypesEntity.PK, null);

    . . .

protected synchronized void initStructuralProperties()
    throws KfMetaDataException {

    // Attribute "AT.ID"
    attrId.setDisplayType(Attribute.DISPLAY_TYPE_TEXT);

    // Attribute "AT.TYPE_NAME"
    attrTypeName.setDisplayType(Attribute.DISPLAY_TYPE_TEXT);
    attrTypeName.setUpperCase(true);
    attrTypeName.setUpdateable(false);

```

```
// Attribute "AT.DESCRPTION"
attrDescription.setDisplayType(Attribute.DISPLAY_TYPE_TEXTBOX);

// Attribute "AT.ABBREVIATION"
attrAbbreviation.setDisplayType(Attribute.DISPLAY_TYPE_TEXT);

// Attribute "AT.POSITIVE_BALANCE_SIGN"
attrPositiveBalanceSign.addAllowedValue("+", "+");
attrPositiveBalanceSign.addAllowedValue("-", "-");

attrPositiveBalanceSign.setDisplayType
    (Attribute.DISPLAY_TYPE_POPLIST);

super.initStructuralProperties();
}
```

4.1.3. Defining Keys

Each entity key can have one or more keys defined as static structures. The two most common keys are primary key and a descriptor key. In KnowledgeFrame, each entity must have a primary key defined. The primary key is used to uniquely identify each instance of the entity. To define the primary key, you use the EntityKey method and attach the number of attributes required to make a unique instance. Typically, the primary key will be used in creating associations between entities, see Defining Associations.

If an entity is used as a lookup, you can define a descriptor key that will be used as the presentation value. A descriptor key typically is a user defined description, such as a name. To define a descriptor, you use the EntityKey method and attach the number of attributes that is to be displayed in lieu of a key value. If more than one attribute is specified, KnowledgeFrame will separate each attribute with a “,”.

Here is the “defining of keys” for the previous example:

```
// Keys

public static final EntityKey PK = new
EntityKey(EssAccountTypesEntity.class,EntityKey.PRIMARY_KEY, attrId);

public static final EntityKey DESCR = new
EntityKey(EssAccountTypesEntity.class,EntityKey.DESCRIPTOR, attrTypeName);
```

Entity Keys must be defined after the attributes have been defined.

4.1.4. Defining Associations

Entity Associations are typically based on database Foreign Keys, but can be created even where there are no underlying Foreign Keys, or to extend the Relational Integrity constraint with additional where Clauses; this simplifies the navigation of the underlying physical data and avoids the need for the Application components to build in dependencies on the data they use.

Associations are defined on the child entity (i.e. the entity that has the foreign key reference). These Associations can be Primary Key/Foreign Key joins or other types of joins, and can be dependent on conditions that could be expressed through where clauses.

If an entity is referred in another entity, then an association is required to have KnowledgeFrame handle the referencing. An association is setup by referencing the class and the entity key of the corresponding (parent) entity with the entity key within this entity.

Here is the “defining of associations” for the previous example:

```
// Associations

public static final Association assocAtParentFk = new
Association("AT_PARENT_FK", EssAccountTypesEntity.attrParentId,
EssAccountTypesEntity.PK, null);
```

Association(s) are defined after the entity keys have been defined.

4.1.5. Defining Behavioral Properties

Most of the behavioral modifications are done in the **init()** method once the Entity’s contents has been copied into a private instance, and in the context of a particular session or instance. This means that the Session Context (SCX) for the session has been already established, and the behavioral settings can be made based on user’s ID, Roles, information from a specific database connection etc. If later a Rule or application logic overrides anything in the Entity, it is expected to do it only in the private instances, and not touch the **static** data shared by all sessions.

Here is the “behavioral part” of the previous example classes:

```
public void init() throws KfMetaDataException {
    this.setHeadingPlural("Account Types");
    this.setHeadingSingular("Account Type");
    this.setAllowEditWhenLookup(false);
    this.setAllowNewWhenLookup(false);
    this.setOrderByClause("AT.ORDER_SEQ ASC, AT.TYPE_NAME");
    this.setValidationWhereClause("TRUNC(AT.EFFECTIVE_DATE) <=
        TRUNC(SYSDATE) AND TRUNC(NVL(AT.END_DATE, SYSDATE)) >=
        TRUNC(SYSDATE)");
    this.name = "ESSACCOUNTTYPES";
    this.setSqlTable("ESS_ACCOUNT_TYPES", "AT");
    EsisSecurityMatrix.setRoles(this);
    super.init();
}
```

4.1.6. Entity Property Methods Reference

Following the initial definition, the properties of all Data Access business objects can be controlled via a set of property methods. The following sections describe the most important ones.

The following property methods are applicable for the Entity:

- **addSqlTable(String _sqlName, String _sqlAlias, String _sqlJoinAlias, String _sqlJoinParentColumns, String _sqlJoinChildColumns)** ... this method adds an SQL table to the Entity definition. The parameter `_sqlName` is the physical table (or view, or synonym) name in the SQL database, and `_sqlJoinAlias` is the object alias. An alias can be the same as object name for simple scenarios; however, for more complex data models, it is possible that the same physical table name is used more than once in SQL statements (e.g. to model hierarchical relationships or self-associations); in that case, the alias must be used to distinguish between Entity instances. Each Entity has to be mapped to at least 1 physical table – if it is mapped to exactly one (which it is, in a majority of cases), then the values `_sqlJoinAlias`, `_sqlJoinParentColumns` and `_sqlJoinChildColumns` are all null; otherwise they express the join conditions.
- **setWhereClause(String _whereClause)** ... sets an optional WHERE clause which is applied to all queries against the current entity, for example to access only data with a certain status or to select subtype data from supertype tables
- **setValidationWhereClause(String _validationWhereClause)** ... in certain situations, the data selected from the database is different from new data allowed to be entered (for example, when a list of allowed codes has been changed, but records with old codes still reside in the database, and must be displayed correctly); this difference can be expressed as “WHERE clause” vs. “validation WHERE clause”.
- **setOrderByClause(String _orderByClause)** ... the default ORDER BY clause applied to the queries; individual queries and blocks can override it if they wish to
- **setHeadingSingular(String _headingSingular)** ... printable Entity heading, singular form
- **setHeadingPlural(String _headingPlural)** ... printable Entity heading, plural form
- **setInitialForcedFilter(boolean _initialForcedFilterFlag)** ... when data from an Entity are displayed in a list, report or lookup, it is possible to force the user to narrow down search criteria (“filter” the data first) – this is used for Entities that can return many rows on each typical query
- **setReadOnly(boolean _readOnlyFlag)** ... make sure that all Blocks using this Entity will make data accessible read-only
- **setInsertable(boolean _insertableFlag)** ... indicate whether this Entity allows INSERT (creating new database records)
- **setUpdateable(boolean _updateableFlag)** ... indicate whether this Entity allows UPDATE (changing existing database records)
- **setDeleteable(boolean _deleteableFlag)** ... indicate whether this Entity allows DELETE (deleting existing database records)
- **setQueryable(boolean _queryableFlag)** ... indicate whether this Entity allows QUERY (database record search) – please note that even if search is not allowed, the Entity may still allow individual record updates via block navigation (e.g. a user

can be allowed to see or modify his/her own account, but is not allowed to search for other people's accounts)

- **addCascadeDelete(String _expression)** ... SQL expression executed before DELETE, to perform a cascade-delete functionality
- **setEnforceMandatory(boolean _enforceMandatoryFlag)** ... if set to FALSE, the "mandatory column" rules are temporarily disabled – this is to support saving "work in progress" data for long-running transactions, and avoid data loss
- **setSelectOptimizerHint(String _selectOptimizerHint)** ... optional hint for the Oracle SQL optimizer, needed for large tables participating in complex queries (see Oracle Database Tuning Guide)
- **addAttrGroup(AttrGroup _attrGroup)** ... add an "Attribute Group", as documented in `com.t4bi.kf.model.attribute.AttrGroup` – this is used to group several attributes into a group for presentation purposes (e.g. to draw a labeled frame around related attributes, or to support tabbed interface etc.)
- **addEntityRole(EntityRole _role)** ... add a security role to this entity; more than one role can be added. If at least one role is added to the entity, only users who log in with one of the assigned roles can see or modify data controlled by this entity
- **setDisplaySeparator(String _sep)** ... when a Descriptor is printed in the output as a single string, individual attributes are typically separated by a comma; this method allows an alternative separator
- **setDocumentation(String[][] _doc)** ...sets the documentation for the entity. This documentation will be used when documentation is generated (if enabled). See "Generating Application Documentation".

4.1.7. Attribute Property Methods

The following property methods are applicable for the Attribute:

- **setPrimaryKey(boolean _primaryKeyFlag)** ... indicate whether this Attribute is a part of the primary key (*typically set only during initialization*)
- **setDescriptor(boolean _descriptorFlag)** ... indicate whether this column is a part of the Descriptor (*typically set only during initialization*). The Descriptor is a list of columns displayed in labels, lists, pop-lists etc., and meant to be meaningful to the business user (as opposed to showing internal values of surrogate keys)
- **setSqlName(String _sqlName)** ... set the physical SQL column name in the database for this Attribute (*typically set only during initialization*)
- **setAllowedValues(String[][] _allowedValues)** or **addAllowedValue(String _value, String _label)** ... set the list of allowed values and their labels, either one-by-one or via an array (*typically set only during initialization*)

- **setInSql(boolean _sqlFlag)** ... indicate whether this Attribute has a database value (**true**), or whether it is a derived Attribute (**false**)
- **setDisplayType(int _displayType)** ... select a display type for this Attribute (plain text, pop-list, radio group, popup window etc.)
- **setDataLength(int _dataLength)** ... set the maximum length of input data
- **setDisplayLength(int _displayLength)** ... set the display length of data on screen (may be less than data length, and the Web browser may restrict it even further)
- **setAlignment(int _alignment)** ... set the data alignment (left / right / centered)
- **setFormat(String _format)** ... set the display format for numeric, date and timestamp values, using formatting masks similar to Oracle TO_CHAR function
- **setDefaultValue(String _defaultValue)** ... if the value is null in a new or empty record, it is auto-filled to the specified value
- **setForceValue(String _forceValue)** ... force this value to this field before each write operation
- **setNullReplacementValue(String _nullReplacementValue)** ... if the value is null immediately before the record is written to the database, it is auto-filled to the specified value
- **setMinAllowedValue(String _minAllowedValue)** ... if not null, checks that the entered value is not lower than the specified number / string / date
- **setMaxAllowedValue(String _maxAllowedValue)** ... if not null, checks that the entered value is not higher than the specified number / string / date
- **setEditHeading(String _editHeading)** ... set heading for the Edit block
- **setListHeading(String _listHeading)** ... set heading for the List block
- **setQueryHeading(String _queryHeading)** ... set heading for the Query block (mode)
- **setReportHeading(String _reportHeading)** ... set heading for the Report
- **setMandatory(boolean _mandatoryFlag)** ... indicates whether this Attribute is Mandatory
- **setVisible(boolean _visibleFlag)** ... indicates whether this Attribute is visible in any Block
- **setVisibleInList(boolean _visibleInListFlag)** ... indicates whether this Attribute is visible in List, List Edit or derived Blocks
- **setReadOnly(boolean _readOnlyFlag)** ... indicates whether this Attribute is read-only

- **setUpdateable(boolean _flag)** ... indicates whether this Attribute is updateable, after it has been initially saved in the database
- **setUpperCase(boolean _upperCaseFlag)** ... indicates whether this Attribute is shown in uppercase, and forced into uppercase on data entry
- **setQueryable(boolean _queryableFlag)** ... indicates whether this Attribute is included in Query / Filter screens
- **setQueryCaseSensitive(boolean _queryCaseSensitiveFlag)** ... indicates whether this Attribute, when included in Query, generates case-sensitive query clauses. **Note that mixed-case Attributes that are NOT case-sensitive in queries can result in extremely bad database performance on large tables.**
- **setQueryMandatory(boolean _queryMandatoryFlag)** ... indicates whether a user, when filling in the Query criteria, must specify a Query / Filter value for this Attribute
- **setQueryRange(boolean _queryRangeFlag)** ... indicates whether this Attribute, when included in Query, can be specified as a range of values
- **setDuplicate(boolean _duplicateFlag)** ... indicates whether this Attribute is copied to a new record during a "record duplicate" operation
- **setImmediatelyRefreshed(boolean _immediatelyRefreshedFlag)** indicates whether this Attribute, when changed in the browser screen, triggers an immediate refresh of the browser page from the application server (*currently supported only for POP-LIST presentation style, via Javascript*)
- **setHelpText(String _helpText)** ... set an ASCII or HTML help text for this Attribute, for context-sensitive help
- **setPopUpWindowWidth(int _popUpWindowWidth)** ... if implemented as POPUP WINDOW, allows to override the default pop-up window width
- **setPopUpWindowHeight(int _popUpWindowHeight)** ... if implemented as POPUP WINDOW, allows to override the default pop-up window height
- **addAttrGroup(AttrGroup _attrGroup)** ... include this Attribute in a particular Attribute Group (see Entity Property Methods)
- **setExternalLabel(String _externalLabel)** ... set a customer-specific label, numbering scheme etc. for this Attribute (used typically for applications developed to replace legacy paper forms, where business users are used to fill in "field 32A")
- **setDocumentation(String[][] _doc)** ...sets the documentation for the module. This documentation will be used when documentation is generated (if enabled). See "Generating Application Documentation".

4.1.8. Association Property Methods

The only 2 property methods for Association (class EntityAssociation) used actively today are:

- **setWhereClause(String _whereClause)** ... restrict the values visible in the target table (typically parent table), by adding additional WHERE clause(s) to the SELECT statement
- **setLookupWhereClause(String _lookupWhereClause)** ... similar to setWhereClause, used for Lookup (pop-list, pop-up window) scenarios, where additional restrictions may be requested by the application

Additional clauses have been defined for future use, but are not yet supported by code, or not actively used by many applications.

- **setAssocType(int _assocType)** ... will allow support for other, more complex, associations, not just Foreign Key based
- **setCardinalityType(int _cardinalityType)** ... the currently supported cardinalities are 1:1 and 1:many, not enforced by code; additional cardinalities and additional checks may be supported later
- **setActive(boolean _activeFlag)** ... will allow to enable or disable the association dynamically at runtime
- **setCascadingDelete(boolean _cascadingDeleteFlag)** ... currently supported via Entity.setCascadeDelete(), which gives us more flexibility (e.g. the ability to delete data not mapped via Entities); using Association. setCascadingDelete() is a more appropriate model-centric approach

4.1.9. Event Activation Methods for Programmatic Rules

The class Entity currently provides the following Activation Methods for the Programmatic Approach to Business Rules:

- **beforeDelete(), afterDelete()** – code executed before or after a record is deleted from the database
- **beforeDuplicate(), afterDuplicate()** code executed before or after the application creates a new in-memory record as a copy of an existing record
- **beforeInsert(), afterInsert()** code executed before or after a record is inserted into the database
- **afterNew()** - code executed after a new, empty record has been created in memory
- **beforePaint()** – code executed before the current record is painted on screen (browser) via a corresponding Block

- **beforePostLookup(), afterPostLookup()** – code executed before or after a pop-up window lookup executes an HTTP POST to copy the lookup result into the record in the base block
- **afterProcess()** – code executed after user-modified record has been received back from the screen (browser)
- **beforeQuery, afterQuery()** – code executed before or after a database Query is executed for this entity. The beforeQuery() method can further modify the Query (e.g. add a WHERE or ORDER BY clause). The afterQuery() method is called once for each record and can modify that record's data (e.g. calculate derived values)
- **beforeSave(), afterSave()** – this is just a convenient shortcut for beforeInsert() + beforeUpdate(), or afterInsert() + afterUpdate(). Many data-quality Rules must be applied on both INSERT and UPDATE, and it is safer for developers not to have to duplicate the code.
- **beforeUpdate(), afterUpdate()** – code executed before or after a record is updated in the database
- **beforeExitRow()** – code executed before navigating to another row. Data-quality rules such as ensuring mandatory requirements are met before leaving a row.

Classes Attribute and Association are accessed via the Entity's Activation Methods.

4.2. Modules, Pages and Blocks

4.2.1. Overview

Modules, Pages and Blocks are typically defined in a single Module file. The Module file consists of a simple constructor, and an **init()** method. All Page and Block objects are defined and linked to each other inside the **init()** method under full programmer's control. In addition to Pages and Blocks, the code in the **init()** method also creates instances of Entities for individual Blocks. For more complex Modules, custom Actions or other programmatic behaviors are called and attached (almost always to Blocks or their Entities).

4.2.2. Defining Pages and Blocks in a Module

A Page is defined via a local variable in the **init()** method, with a specific page constructor (typically Base Layout Page or Navigation Layout Page) building a page definition. Pages can be defined in any order.

Blocks are defined in a similar way, via a local variable and a constructor. Blocks have more dependencies than pages, and all these dependencies must be set in the constructor or later via **setXXX()** methods. All Blocks must be assigned to one Page in the Module.

If a Block is linked to database data (that means, it is a subclass of an Entity Based Block), then its entity must be defined. The Entity is either build specifically for the Block (again, via a local variable and a constructor), or it can be shared with a parent Block. Also, a Block may have a relationship with its parent block – either by sharing data (e.g.

when drilling down from a List Block to an Edit Block), or by traversing an association. In the latter case, the Block Association must be specified in a Block constructor (because Entities the Blocks are based on can have more than 1 association between them).

For navigation purposes, one of the Blocks in the Module must be defined as a starting block. When the user enters a Module, by default, the Page containing the starting Block is displayed first.

Following the initial definition, the properties of all Application objects can be controlled via a set of property methods. The following sections describe the most important ones.

This example shows the definition of a simple module with 2 pages and 2 blocks – the first page displays a list of master records, and the second page is a drill-down to list of detail record, linked to the master table via a Foreign Key (Association):

```
public class ActivityWorkloadManagementModule
    extends Module {
    ...

    /**
     * Method init() contains most of the Module definition data,
     * and completes the Module initialization.
     *
     * @throws KfMetaDataException ;
     */
    public void init()
        throws KfMetaDataException {
        setName("ESISFRMWKLD020");

        // =====
        // ---- Page[1]: ESIS_WKLD020_SPA01
        BaseLayoutPage p1 = new BaseLayoutPage(scx, "ESIS_WKLD020_SPA01",
            "Activity Workload Management", this);

        p1.setPageNumbering(true);
        // -----
        // ---- Block[1]: ESIS_WKLD020_LEB10
        // Entity ESSWKLDACTIVITYPARTICIPANTS
        EssVwkldActivityParticipantsEntity ble1 = new
            EssVwkldActivityParticipantsEntity(scx);
        ListBlock b1 = new
            ListBlock("ESIS_WKLD020_LEB10",
                "List Activity Participants", ble1, p1, scx);
        b1.addAction(
            new GenerateDocumentAction(
                b1, b1.getName() +
                GenerateDocumentAction.GENERATE_XLS_ACTION,
                "Export to Excel", "excel_template.xml"));

        // =====
        // ---- Page[2]: ESIS_WKLD020_SPA02
        BaseLayoutPage p2 = new BaseLayoutPage(scx, "ESIS_WKLD020_SPA02",
            "Activity Participants", this);

        EssActivityParticipantsEntity b2e2 = new
            EssActivityParticipantsEntity(scx);
```

```

EntityAssociation asb2e2_ble1 = b2e2.buildAssociation(
    "AP_VWAP_FK", ble1);
ListEditBlock b2 = new ListEditBlock("ESIS_WKLD020_EBL10",
    "Edit Activity Participants", b2e2, p2, scx, b1,
    Block.PARENT_BLOCK_TYPE_VIA_ASSOCIATION, asb2e2_ble1);
this.activeBlock = b2;
this.activeBlock = b1;
super.init();
    }
}

```

4.2.3. Module Property Methods

The following property methods are applicable for the Module:

- **setTitle(String _title)** ... set the module title; the module title is not explicitly displayed on screen, but it is used for session tracking in the database, via DBMS_APPLICATION_INFO. Also, Pages and Blocks can use this text to display it as a part of their titles.
- **setShowErrorIcon(boolean _showErrorIconFlag)** ... indicator whether an error icon should be displayed in the error/warning message.
- **setDocumentation(String[][] _doc)** ...sets the documentation for the module. This documentation will be used when documentation is generated (if enabled). See "Generating Application Documentation".

4.2.4. Page Property Methods

The following property methods are applicable for the Page:

- **setTitle(String _title)** ... set the page title, displayed with the page-level toolbar.
- **setHeaderFooterDisplayable(boolean b)** ... indicator whether this page will display the application-standard header and footer
- **setDocumentation(String[][] _doc)** ...sets the documentation for the page. This documentation will be used when documentation is generated (if enabled). See "Generating Application Documentation".

The following property methods are applicable only for the NavigationMenuPage:

- **setRootNode(boolean isInit)** ... define the root block of the navigation tree
- **setUsingScrollableFrames(boolean _flag)** ... true means that the "frames" (tree, and data) are separately scrollable
- **setAutoExpandChildCount(int _count)** ... set the maximum number of rows expanded automatically on this page

- **addPeerBlock(String _label, EditBlock _block, EditBlock _parentBlock) ...**
add a new Block to the Navigation Menu, as a peer to an existing Block. In this case, the existing Block must be a parent to the new block (in terms of Block parent/child relationship in the module), and both Blocks must share the Entity & comp. (PARENT_BLOCK_TYPE_SHARED_DATA).
- **addGroupBlock(String _label, EditBlock _block, EditBlock _parentBlock) ...**
- **addChildBlock(String _label, EditBlock _block, EditBlock _parentBlock, [boolean _useListEditBlock]) ...**
- **addRecursiveBlock(String _label, EditBlock _block, EditBlock _parentBlock, EntityAssociation _recursiveAssociation, int _levels, boolean _useListEditBlock) ...**
- **setWidthPercent(int _widthPercent) ...** define the width of the navigation tree, in percent of HTML page width
- **setWidthPix(int _widthPix) ...** define the width of the navigation tree, in pixels
- **setLabelDisplayableCharacters(int _labelDisplayableCharacters) ...** limit the number of characters displayed on tree nodes (labels), to limit text wrapping
- **setNavigationMenuBlockTitle(String _title) ...** set the title for the navigation menu block
- **setRelativeHomeOption(boolean _option) ...** enables an option, through which a user of a long and complex navigation tree, can “pin” an expanded sub-tree to the root of the tree, to work with a subset of the tree data for a while. Recommended for advanced and computer-savvy users only.
- **setNavigationPageTitle (String _title) ...** set the title of the navigation page
- **setNavigationTitle (String _title) ...** set the title of the navigation toolbar
- **setAutoSave(boolean _autoSave) ...** force auto-save for all blocks on the page

4.2.5. Block Property Methods

The following property methods are applicable for the Block and EntityBasedBlock (which is what we are mostly interested in):

- **setTitle(String _title) ...** set the block title, displayed with the block-level toolbar
- **setNavigationMandatoryIndicator(boolean _navigationMandatoryIndicatorFlag) ...** set an indicator whether this block should be marked as mandatory in the navigation tab/menu
- **setVisible(boolean _visibleFlag) ...** make this block visible (default) or invisible
- **setAutoSave(boolean _autoSaveFlag) ...** force auto-save for this block

- **setIndicateMandatory(boolean _indicateMandatoryFlag)** ... if set to **false**, the Mandatory Attribute rules are not shown on screen, or enforced (this is to allow saving work-in-progress data e.g. in temporary work areas)
- **setReadOnly(boolean _readOnlyFlag)** ... **true** makes all data in the Block read-only
- **setInsertable(boolean _insertableFlag)** ... **false** means that this Block does not allow new data records to be created
- **setUpdateable(boolean _updateableFlag)** ... **false** means that this Block does not permit updates of existing data
- **setDeleteable(boolean _deleteableFlag)** ... **false** means that this Block does not permit data to be deleted
- **setInitialForcedFilter(boolean _initialForcedFilterFlag)** ... **true** means that before any data is displayed, the user has to enter search / filter criteria, to reduce the number of records typically returned from the database
- **setWhereClause(String _whereClause)** ... set an additional SQL WHERE clause to all queries
- **setFilterWhereClause(String _whereClause)** ... set an additional SQL WHERE clause to filter statements
- **setOrderByClause(String _orderByClause)** ... set an SQL ORDER BY clause to all queries
- **setColumnsInBlock(int _howManyColumns)** ... allows data in Edit Block to be displayed in more than one column (default = 1, max. = 4)
- **setMultiColumnPresentation(int _presentation)** ... if Edit Block data is displayed in more than one column, control the row-wise ("Z-order") or column-wise ("N-order") presentation
- **setDocumentation(String[][] _doc)** ...sets the documentation for the block. This documentation will be used when documentation is generated (if enabled). See "Generating Application Documentation".

4.3. Using Symbolic Expressions and Bind Variables

As KF uses a declarative approach to defining the relationships between Java objects at runtime, it needs a symbolic declarative form to refer to other runtime objects, which exist (or contain relevant values) only at runtime. This is done in the form of symbolic expressions and bind variables. The bind variables (modeled after JDBC or Oracle bind variables) are the key building block for resolving the runtime access to other data and application objects. The bind variables and expressions can be used in many places where string values can be used. The general syntax is in the form of **:type(values)** – that string (starting with the colon, and ending with the closing parenthesis) is replaced with the value to which the bind variable evaluates. Currently, the following bind variables are supported:

- **:attr(*name*)** or **:attr(*entity, name*)** ... replaces the bind variable with the value of attribute name in the current entity. If an attribute with that name does not exist, the parent entity (the entity linked to the parent block in the same module) is used instead, and this process is repeated until the attribute is found, or until the topmost block in the module is reached (in which case the system generates an exception). The second syntax form, with entity name, is used to resolve cases when an attribute with the same name (e.g. "ID") exists in several entities in the module, and we need to select a specific case.
- **:descr(*name*)** or **:descr(*entity, name*)** is similar to :attr – only, if the attribute is either a List-of-Values (LOV) field or a Foreign Key, it displays the "business label" (or "descriptor", in data-modeling speak) of that field, instead of the internal database value (e.g. "John Smith" instead of "employee# = 6234").
- **:session(*code*)** replaces the bind variable with the value generated from the current session, controlled by **code** . The code supported in this version include **ID** (a unique id of the KF session), **PTY.ID** (an internal, typically numeric, identifier of the person or "party" that has been authenticated for this session), **USERNAME** (a human-readable form of PTY.ID), **APP_SYS_NAME** (a string identifying this application and version – used on sites and servers running multiple application and sharing functionality and data among them), **DATE_NOW** (current date), **TIME_NOW** (current time of day), **TIMESTAMP_NOW** (= DATE_NOW +TIME_NOW), **YEAR_NOW** (current year). More code can be added when needed.
- **:sql(*expression*)** evaluates an SQL expression (*which can contain other bind variables, e g. :attr or :session or :user, in its WHERE clauses*) and replaces the bind variable with the result. The developer must ensure that this expression always returns a single value (1 row, 1 column), which can be fetched from JDBC as a character string.
- **:user(*name*)** replaces the bind variable with the value set by a business rule or custom application code earlier in the application, in the hash map (name / value) of user-defined bind variables

5. Development Environment Configuration

5.1. Directory Naming Conventions

Many developers prefer their own directory structure, and system dependencies (e.g. using shared drives, common component installations etc.) typically force a certain directory structure as well. On top of that, it is common practice to install major releases with a release number in the directory name, to ensure that the original installation remains intact when a new release is being installed and tested. This document uses a number of symbolic names to identify roots of directory sub-trees. The developers can choose their own preferred directories for the sub-trees, but the code underneath is expected to follow conventions shown in this document (or in the documentation of vendors of specific tools). Also, the developers can decide whether they wish to define actual environment variables for these values, or just use them during the configuration process. The variable `$DEV_HOME` is used by some ANT scripts, and Oracle recommends setting up `$ORACLE_BASE` and `$ORACLE_HOME` (and including `$ORACLE_HOME/bin` in `$PATH`) in its documentation. This document will use UNIX notation for environment (variable names prefixed with “\$”, directories separated with “/”) – obviously, in Windows, Microsoft notation must be used instead (e.g. `$ORACLE_HOME/bin` is translated to `%ORACLE_HOME%\bin`).

The following symbolic names are used in this document:

- `DEV_HOME` ... the common parent of the directories where the source from the configuration management system is extracted; ideally, it should match the root of the Eclipse workspace
- `DEV_LOCAL_HOME` ... in case local deployment differs from the central copy
- `JAVA_HOME` ... the Java JDK installation directory; it is good practice to put `$JAVA_HOME/bin` into the machine's `$PATH`
- `ORACLE_BASE` ... as per Oracle documentation – the parent directory for all Oracle software versions, and the parent directory for database configuration files
- `ORACLE_HOME` ... as per Oracle documentation – the root directory for the installation of the currently active version of Oracle software
- `ECLIPSE_HOME` ... Eclipse installation directory
- `TOMCAT_HOME` ... Tomcat installation directory
- ...

5.2. KF Directory Structure

In all Configuration Management systems (CVS, StarTeam etc.), each project has own directory structure. The name of the project is usually a name of customer/application. This is the KF-standard directory structure within a typical project:

-
- **ctl** – SQL*Loader control files.
 - **data** – Data initialization and loading scripts. The standard here is that there is a schema.sql script in this directory which calls X *.load scripts to physically load the required initialization data.
 - **ddo** – Data Dependent Objects – Basically, objects that require the data in the data directory to be loaded before they can be loaded.
 - **ddo/data** – Data that is dependent on other data load scripts in upper level data directory.
 - **ddo/idx** – Functional indexes that are dependent on data load scripts in upper level data directory or local data directory.
 - **ddo/pkg** – Packages that are dependent on data load scripts in upper level data directory or local data directory.
 - **ddo/trg** – Triggers that are dependent on data load scripts in upper level data directory or local data directory.
 - **ddo/vdf** – View definitions that are dependent on data load scripts in upper level data directory or local data directory.
 - **doc** – Project Documentation
 - **doc/designer** – Designer generated PDF E/R and Report files.
 - **fdx** – Functional Indexes.
 - **idx** – Global indexes not generated by designer (those that are not automatically logically partitioned). This would be empty for all application not using Sergio's logical partitioning strategy. This is dependent on the configuration data being in our CFG though.
 - **info** – The version.txt file resides in here that identifies the script version level.
 - **install** – All build scripts go here as follows:
 - **install/build** – All scripts generated by Designer go here. The naming standard is to have the files named the same as their schemas as defined in Designer (from the example above, t4bi_cfg, t4bi_sys, platinum_lis, platinum_pub). The schema_inst script uses these to build the database objects in a non-production build.
 - **install/patches** – All patch scripts, standard is to have a directory below here for each patch. Each patch directory is named patchXXXX which is a sequential number. The -U option of schema_inst uses files in this directory to apply patches (also requires schema_patches and schema_information tables in database).
 - **install/prod** – All production build scripts. These are generated from any schema using the gen_prod_ddl scripts which basically go into the data dictionary and the CFG schema and determine how to really build tables for production use.

Production tablespaces are assumed to be: SCHEMA and SCHEMA_IDX with the allowance for a separate LOB tablespace of your choice.

- **java** - java related files and directories (in bold), where the tag is indicated, all files from that level are under the Source code management (StarTeam) and the <transient> tag is indicated, the entire directory is transient and may be deleted by ant or other scripts
- **java/antproperties.xml** - ant project properties files
- **java/build.xml** - ant build file
- **java/deleted_starteam_files.txt** files placed under source code management that are subsequently deleted
- **java/audit** - code audits using tools
- **java/bin** - any external binaries, possibly sorted by platform (e.g. java/bin/windows, java/bin/linux etc.)
- **java/build** - working output of the ant build process
- **java/conf** - configuration and properties information; subdirectories contain tool- or platform-dependent data
- **java/data** - mock files and other system data files
- **java/dist** - completed distribution for base target of ant
- **java/doc** - all java related documents and diagrams
- **java/doc/diagrams**
- **java/doc/javadoc**
- **java/doc/operations**
- **java/doc/uml** – UML documentation
- **java/lib** - all external library files
- **java/metadata** - ANT build metadata like build number and build user, META-INF
- **java/shipped/<x.x.x>** - actually shipped builds that are in production
- **java/src** - source code
- **java/src/javascript** – Javascript code and libraries
- **java/src/jsp** – java pages
- **java/src/static** – static elements

-
- **java/src/static/html**
 - **java/src/static/images** – includes all images sorted by format
 - **java/src/static/images/gif**
 - **java/src/static/images/jpg**
 - **java/src/static/images/svg**
 - **java/src/static/xml**
 - **java/test - junit tests**
 - **pkg** – PL/SQL package definitions, either as .pkg / .pkb files, or as common .sql files
 - **sh** – Shell scripts – typically, these should work with Bourne, Korn and BASH shells
 - **sql** – All general .sql scripts that do not fall into other directory categories
 - **test** – non-JUnit test scripts.
 - **test/data** – The .orig files to compare the new runs against.
 - **test/sql** – The SQL script that are used to generate the output data files to be compared.
 - **trg** – PL/SQL trigger scripts.
 - **vdf** - view definition scripts.

5.3. JDBC

While KF can theoretically support any database that provides a JDBC 2.x-compatible driver, the database used for most development work has been Oracle versions 8, 9 and 10 (as of today).

5.3.1. Oracle JDBC

An up-to-date version of Oracle JDBC driver, compatible with the target database, must be obtained from the Oracle Web site (<http://otn.oracle.com>). As of today, the recommended version is **ojdbc14.jar** .

6. Application Server Deployment

6.1. Create a WAR File for Deployment

6.1.1. Components of a WAR file

6.1.1.1. General

WAR files can store all the source files for a web based java application. This could be for JSP written applications, applications consisting of servlets only. So far the solutions developed by T4Bi are pure servlet solutions (no JSP's).

The types of files that exist in a WAR file are:

- static HTML files and JSPs stored in the top level directory
- Servlet and related Java class files must be stored in the `WEB-INF/classes` directory
- library JAR files must be stored in the `WEB-INF/lib` directory
- deployment descriptor is stored as a file named `web.xml` in the `WEB-INF` directory

6.1.1.2. HTML files

It's likely that there will not be any static HTML files, but if there are they can exist at the root level of the web application directory. For example if the application is called `mywar`, then there would be a directory somewhere called `mywar`, and the static HTML files would reside in that directory. Underneath the `mywar` directory would be a subdirectory called `WEB-INF` which would include the `web.xml` file and servlet files which are explained below.

6.1.1.3. Servlet files

Servlet files are java classes that are built using a JDK or IDE environment. For example after building a project in JDeveloper there will be at least one java class file generated for every java source file. Servlet class files reside in a directory called `classes` which lives underneath the `WEB-INF` directory. The full path in relation to the application directory in our `mywar` example is `mywar/WEB-INF/classes`.

6.1.1.4. Library files

Library files are usually Java Archive (JAR) files that are needed by the application. For example, the database connectivity would be in an archive (or library) file for

creating JDBC connections. This would be in a directory called lib. The full path for our example would be **mywar/WEB-INF/lib**.

6.1.1.5. web.xml file

The web.xml file must follow the strict rules of a valid XML file. Additionally, the name of the servlet class must correspond to a class file that is also in the archive file. The servlet mapping will also dictate which servlet is executed when a certain URL is entered into the web browser. Usually in the projects that T4Bi deliver, the servlet mapping will correspond to the name of the main servlet class.

Sample web.xml file:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>System3WLI</servlet-name>
    <servlet-class>System3WLI</servlet-class>
    <init-param>
      <param-name>appTitle</param-name>
      <param-value>WLI Maintenance</param-value>
    </init-param>
    <init-param>
      <param-name>dbType</param-name>
      <param-value>Oracle8</param-value>
    </init-param>
    <init-param>
      <param-name>dbConnectionSpecs</param-name>
      <param-value>t4bidb01:1526:T4BID02</param-value>
    </init-param>
    <init-param>
      <param-name>userId</param-name>
      <param-value>username_value</param-value>
    </init-param>
    <init-param>
      <param-name>password</param-name>
      <param-value>password_value</param-value>
    </init-param>
    <init-param>
      <param-name>expiration_milliseconds</param-name>
      <param-value>1800000</param-value>
    </init-param>
    <init-param>
      <param-name>applicUserId</param-name>
      <param-value>CMT</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>System3WLI</servlet-name>
    <url-pattern>/System3WLI</url-pattern>
  </servlet-mapping>
</web-app>
```

6.1.1.6. Creation of WAR file

After you have created your WEB-INF directory, the deployment descriptor, and have put the relevant files in the correct directories, you can use the "jar" utility from the Java Development Kit (JDK) distribution to create the WAR file. Check the tools documentation for the full syntax. The command you could use is (assumes you are at the top level of the directory structure in which you assembled the WAR contents):

```
jar cvf mywar.war WEB-INF {related top-level files or directories}
```

You can then deploy `mywar.war` using, for example, Apache Tomcat, Weblogic, or any J2EE compliant application server or Web container.

6.2. Deploying the WAR file to the Server

Once the WAR file is built, it is deployed by

- Copying the WAR file to the server
- Using the server's administration utility to load the WAR file into the server

6.3. Generating KF Documentation

Within the KnowledgeFrame framework there is the ability to generate documentation (in HTML pages) that will provide a developer and technical analyst a breakdown of the various components found in the application.

The generator "walks" through the application at startup and generates documentation of all objects that are used within the application.

The documentation that is generated interrogates each object and determines the various attributes of that object. If the attributes is not at the default setting the generator will generate information on this and place this in the documentation.

To further enhance the documentation capabilities, each component has the ability to have user defined documentation provided through the `setDocumentation(String[][] _doc)` method. This method provides the developer the ability to add documentation to assist in clarifying what the component is or used for.

The generator creates the following information:

Modules – each module will be created as a separate html page and contains the pages, blocks and entities. In addition any non-standard actions that are related to the pages and blocks are also shown. For each entity specified, a link to the entity's corresponding documentation will be provided.

Entities – each entity will be created as a separate html page. Each page will contain information related specifically to the entity, keys, associations and attributes.

Rule Cross reference – The page contains a cross reference to specific documented rules (using the setDocument method) and the component that it refers to. A link between the rule and the component is setup to allow the user to view the related information as required.

Business Function cross reference – The page contains a cross reference to specific documented business functions (using the setDocument method) and the component that it refers to. A link between the business function and the component is setup to allow the user to view the related information as required.

Entity cross reference – The page contains a cross reference for between an entity and all of the modules it is related in. A link between the entity and the module is setup to allow the user to view the related information as required.

To generate KF documentation, add the following to the web.xml file:

```
<context-param>
  <param-name>APP_DOCUMENTATION_DIRECTORY</param-name>
  <param-value>Documentation Root Directory</param-value>
  <description>Directory path for all report generator
    templates</description>
</context-param>
```

Once this line is added, start up the application and log into the application with a user id that has full access rights. The framework will look for this attribute being set and if set will generate the documentation.

Note: Remove the above line after the documentation has been generated as the documentation will be generated everytime someone logs into the application.