
KnowledgeFrame 5 Technical Overview

This document provides an overview of the KnowledgeFrame Release 5 Application Framework and development approaches using KnowledgeFrame.

KnowledgeFrame is a standards-based Application Framework designed and developed by Templates 4 Business, Inc. (T4Bi), for reducing the costs, complexities and time-to-market of sophisticated business application systems.

For more information about KnowledgeFrame or the services offered by T4Bi contact knowledgeframe@t4bi.com

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

Office: 604.681.4228
Fax: 604.681.4256

<http://www.t4bi.com>



**KnowledgeFrame
reduces costs,
complexities and
time-to-market of
sophisticated
business application
systems.**

Application Frameworks

Framework Definition

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.

Framework Benefits

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

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



Framework Architecture Policies

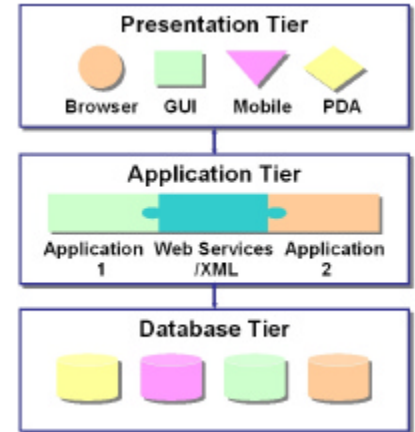
The following policies ensure that the framework delivers these desired benefits:

- Clearly identify the class of problem the framework addresses.
- Follow best software design and build practices; follow an architecture of “loose-coupling” between component tiers, minimizing the costs and risks of change.
- Scalability is achieved by enabling components to (appropriately) exploit the following:
 - data and rule caching,
 - distribution across multiple (middle-tier) servers (e.g. with load-balancers).
 - tuned database operations.
 - tuned network operations.
 - multi-threading
- Robustness is achieved through:
 - comprehensive exception handling
 - component interface design
 - component reuse
 - thorough testing
- Provide a high-level framework structure which communicates the behaviour of the framework and provide a starting point for designing framework extensions and interactions.
- Provide a design and development methodology - standards, approach, tools and techniques - to enable designers and developers to follow strict guidelines when extending the framework.
- Avoid dependence on vendor-specific technologies and approaches. All interactions between components should follow industry accepted standards. Vendor specific code and designs should be appropriately abstracted to minimize the impacts of adopting different technologies.

Architecture Vision

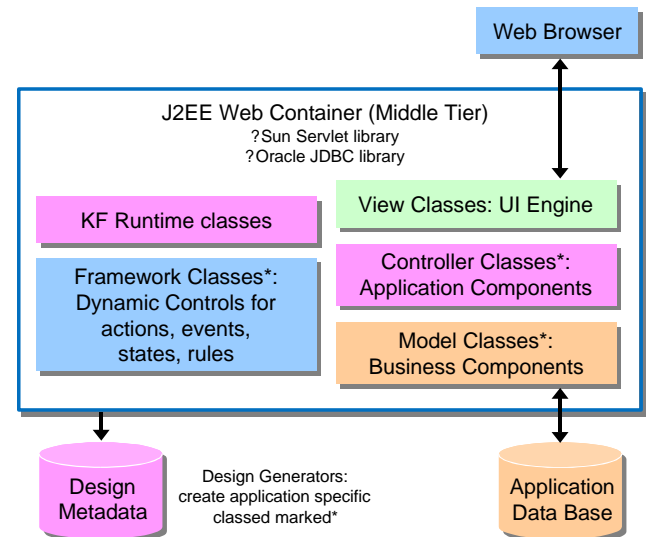
The following schematic emphasises the separation of the logical application tiers; presentation, application logic (including rules) and data. The abstraction of components in these tiers is accepted across the IT industry as a best practice to achieve the benefits previously outlined.

Within each tier components are appropriately abstracted and provide consistent interfaces to also avoid excessive interdependence.



Logical Component Architecture

The following schematic shows the organization of the core KnowledgeFrame Components. The schematic also shows that these components are generated from the design environment. T4Bi KnowledgeFrame can support dynamic meta-data instantiation, where the meta-data Repository is deployed to the Production environment.



KnowledgeFrame Application Design Standards and Guidelines

Design and Build Approach

Design and Build using KnowledgeFrame follows a declarative development paradigm. This emphasizes reusing existing KnowledgeFrame components (as building blocks) to address all of the Application's System Requirements. The Application logic is divided into 2 logical tiers; Business Components and Application Components. The Application Components are responsible for the business functions that enable and support user interaction and the Business Components are responsible for the data. (These components are detailed in the following section).

From a Design and Build perspective the other significant components are the Rule components, which can be bound to Application and Business Components as required.

Declarative design and development allow Analysts and Developers to specify their requirements in meta-data (through the KF Design Utilities and scripting) rather than by writing code. This meta-data is stored in the KnowledgeFrame Repository and used to construct the Application and Business Components (including the binding of Rule 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.

The partitioning (or abstraction) between Business, Application and Rule Components enables significant levels of reuse significantly improving productivity. This Architecture also makes the application more flexible and resilient, allowing changes to be isolated to just the impacted component(s) and inherited through interfaces. For example changes to System Requirements resulting in database changes and additional Rules, can generally be achieved declaratively and isolated to just those Java Classes implementing the new or modified Rules.

Custom code is protected during a re-generation cycle and, should any dependencies become obsolete, these can generally be identified by the Java compilers.

The goal of this approach is to restrict the need for custom coding to just the Business Rules and Action Items which are not already supported "natively" by KnowledgeFrame. Custom code is written in the Java IDE (e.g. JDeveloper) and inherits from its appropriate KnowledgeFrame Class, extending or overriding behaviour as appropriate. The custom code is linked into the Application declaratively through the KF Design Utilities.

◆ The core design patterns are therefore provided by KnowledgeFrame and the Design and Build extends these by adding Rule and Action Classes. This level of reuse is fundamental to achieving a flexible and extensible application within the Project's time and budget constraints.

The KnowledgeFrame Repository can be deployed to the Production environment and components built dynamically by the runtime framework. A generation methodology generally has productivity, robustness, performance and configuration management advantages.

Business Rules can themselves be code *shells* with all their principal behaviours and parameters controlled declaratively. For these cases the Rule can be deployed as a combination of code and meta-data (in SQL tables) or simply as code with the meta-data embedded in the Java Class. Deploying meta-data in

SQL tables is only appropriate where that meta-data must be changed in the Production environment by an Application Role. These design decisions are made during the detailed design phase.

Rules are used to implement data quality and presentation requirements.

Business Components

- Business Components are middle-tier Business objects, instantiated in the JVM. These components may be instantiated from definitions in compiled Java Class definitions or definitions in meta-data.
- Business Components access data from one or more database objects and apply Business Rules to these data on the appropriate Events.
- Business Components encapsulate the Object-Relational mapping needed to integrate Java Objects with SQL Objects and also hide SQL complexities from developers.
- Business Components are first defined in the KnowledgeFrame Repository. This is generally achieved by loading definitions from the analysis repository (e.g. a

CASE tool like Oracle Designer) and then extending these through the KF Design Utilities. The following artefacts are generally transferred:

- Entities, Attributes, Relationships, Keys,
- Domains,
- Function Hierarchies.

Projects with limited scope can skip the Design Repository and work directly in Java IDE's.

- Strategies for the initial population of the KnowledgeFrame Repository include creating Entity, Attribute, Relationship, Allowed Value and Key definitions from scripts executed against the database data-dictionary.
- Business Component Design tasks include setting the properties for each of the above artefacts. This Design is generally done through the KF Design Utilities. These allow new Entities, Attributes and Associations (between Entities) to be defined.
- Business Components generally map to one SQL Table or View and provide a 'logical' (business functional) view of the underlying physical data structure; avoiding the need for Application Components to further manipulate the data, other than through context related Rules.
- Entity Associations can be created even where there are no underlying Foreign Keys, or to extend the Relational Integrity constraint with an additional Where Clause; 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.
- The KF Business Component definitions therefore abstract data complexities in the physical data model, hiding these from other components and removing the need to for developers to write SQL for data operations. As discussed this also minimizes change impacts, achieving a key architectural objective for the application.

Entities and Associations

Entities and Associations are based on SQL objects (usually tables, views and referential integrity constraints). These represent the business view of the data and, as such, their design generally closely follows the original System Model (Entity Relationship Model) as opposed to the Physical model.

Entities have meta-data which includes how they are mapped to a SQL object and how select, insert, update and delete are performed. The developer does not need to

write SQL directly when working with Business Components. The Entity's meta-data includes its Associations to other Entities (parent and child associations). 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.

The Entity's meta-data can be retrieved and modified at runtime by Rules, allowing its behaviour to be reconfigured. This technique avoids creating Entities for each possible context of their use or each possible sub-type.

Most important Business Entities are sub-typed on the System Model. These sub-types inherit some Attributes, Rules and Associations from their parents but they generally have Attributes, Rules and Associations of their own.

Projects with limited scope can skip the Design Repository and work directly in Java IDE's.

KnowledgeFrame allows these sub-types to be created as literal Entities or for the super-type Entity to be implemented with Rules governing the behaviour of each sub-type. This design decision usually depends on how distinct each sub-type is and whether these sub-types are generally used by

very different processes. Similar criteria are used for the physical database implementation.

The detailed Design deliverable will include a definition of all Application Entities and their default meta-data definitions and values. High-level Entity Design is covered in section 4 of this document.

Application Components

The KF Repository is used to define all Application Components; Modules, Pages, Blocks and Actions.

KnowledgeFrame includes Java Classes for most common paradigms for representing data and navigating through structured data. Module implementations generally use these base KF classes to implement their required behaviours. These base classes can be extended where exceptions are discovered.

Conformity of design, behaviour and functionality is enforced by the KF Classes. Changes to look and feel, behaviour or universal functionality, must be done by extending or overriding the appropriate base KF class.

Application Components are also first defined in the KF Repository. These components are:

- Application Menu.
- Modules; consisting of Pages, Blocks and Actions
- Utilities; such as Audit Functions, associated to Business Components.

Modules are designed according to the following criteria:

- Reusability; reduce the amount of code required. This is often a result of common patterns of data usage. However functional patterns (independent of data) are also modeled and implemented for reuse.
- Flexibility; ability to support new/modified requirements with minimal or no code changes.
- Data usage patterns;
- User Workflow; how the system's users will interact with the Modules through all possible business process contexts. This is generally controlled by Rules based on the user's Role(s), privileges of that Role, the current Process being performed and the state of the data being operated on.
- Productivity and Usability; ensuring the system's users can perform their tasks in appropriate time-frames and with minimal learning curve requirements.

Business Rules

- Business Rules are defined in the KnowledgeFrame Repository and associated to a Business or Application Component and the Event on which they will be fired.
- Business Rules must be classified in the KnowledgeFrame Repository and each of these Rule Classes is implemented as a Java Class instantiated by the KF Rule Engine at runtime (with its required parameters).
- A rich Class hierarchy forms the basis for reuse and flexibility in the business application; most Business Functions can be abstracted to patterns (Classes) which perform exactly the same types of Create, Read, Update and Delete (CRUD) operations on one or more Entities. Rules control all aspects of these operations. Separating these CRUD mechanisms from the Rules avoids the need for custom Functions for most cases, and so enables Modules to be built using standard classes (design patterns).

- The following is the (high-level) KnowledgeFrame Rule Classification:

- Entity Rules
- Domain Rules
- Attribute Rules
- Association Rules
- Process Rules

- Authorization Rules
- Document Rules
- Presentation Rules
- The Rule Classification hierarchy is typically 3 or more levels deep in each of these branches.
- Detailed Design will map all defined Rules to the appropriate Classification in the KnowledgeFrame Repository.
- These classes are easily extendable through subclassing or even new branches to the hierarchy (provided they inherit from the parent KF Rule class).
- KnowledgeFrame Rule Classes are generic patterns of Rule logic independent of the business model.
- Rules have access to all of the Application Component and Business Component meta-data and can change the properties of these components dynamically at runtime (e.g. visual properties, WHERE clauses, allowed values and so on).
- Business Component Reports include physical table and key mappings and mappings to "originating" Business Requirement definitions.

A rich Class hierarchy forms the basis for reuse and flexibility in the business application;

KnowledgeFrame Application Build Standards and Guidelines

All Application Classes extend an appropriate KnowledgeFrame base Class. In most cases these Classes will be generated and will not need to override or modify any inherited functionality.

If custom behaviour is required the generated class can be sub-classed, so that re-generation does not overwrite custom extensions. Most custom behaviours will be achieved by adding Rules to reconfigure a Component or for custom Actions.

Should it become necessary to extend the core KnowledgeFrame functionality to achieve a specific behaviour, the standard procedure is as follows:

- Extend one of the existing KnowledgeFrame classes using Java inheritance.
- Implement new or modified behaviour in the new Java subclass (or, for complex requirements, call all the required classes and packages from the new Java subclass).

- Use the customized Java subclass instead of the built-in class in the appropriate location.

An example of a custom-written class is an implementation of a Block that displays data with a complex or non-standard layout (e.g. to generate a very specific report or “form letter”), or a Block that is expected to interact with a desktop application (e.g. Microsoft Word or Excel) instead of a standard HTML browser output.

Business Rule Development

A Rule is integrated into the application by attaching it to a particular Component and Event, as a part of the Component definition.

The KF Component generator will attach Rules to the Component class and to all appropriate child classes.

The sequence in which Rules are attached to a Component for specific Events determines the sequence in which the Rules are executed (when that particular Event is fired).

Design and Build Prototyping

This section is included to show how the Application Design is expected to evolve through a structured iterative design and build process. This does not change the scope of the System Requirements, however it allows for user involvement in how these requirements are implemented, as well as early confirmation that these requirements are complete and correct.

Prototyping starts by using pre-existing building blocks (KnowledgeFrame Classes) and then, once basic scope is confirmed, adds custom code. Prototyping is enabled by reuse.

The following classes of prototypes will be created during Design and Build phase iterations:

- Data Prototype; Used to validate core data structures and relationships through screens and reports. No custom code is introduced at this point and Rules are restricted to built-in declarative Rules for data quality and presentation.
- Design Prototype; Used to establish navigational, functional and presentation standards for common simple and complex scenarios. (This prototype is planned for delivery at the start of the Functional Prototype). This prototype allows the base Application classes to be configured for the appropriate user preferences. Should this phase establish that the existing KnowledgeFrame Application Components will not meet all standard requirements they will be extended at this point.
- Functional Prototype; Used to validate the implementation of the core business functional requirements and

establish gaps (e.g. missing rules, navigation and presentation features). This prototype introduces Entity, Attribute and Association Rules and more advanced Presentation Rules.

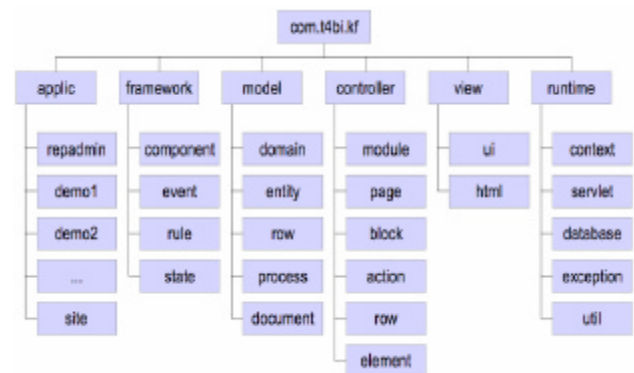
- Workflow Prototype; Used to validate the implementation of the workflow processes from productivity and usability standpoints. The Workflow Prototype introduces Process Rules.
- Application Prototype; Completion of core business functionality, rules and navigation requirements. Identification of gaps and advanced productivity and usability requirements.

The Workflow Prototype will introduce the new Rule Classes required to achieve the more advanced scope requirements. The Functional Prototypes focus on ensuring the fundamental data and functional scopes of the Module.

KnowledgeFrame Application Architecture

Packages and Classes

The following Package diagram shows the high-level KnowledgeFrame Architecture.



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 needed to run the whole application site, especially the Login Module and Main Menu (or Home Page).
- 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)

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 area of the Framework Package 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 ComponentType, Component and ComponentAllowedValue; 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.

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
- Package entity – business object level functions (meta-data definition, physical database mapping, binding of data and rules) – Entity, Attribute, Domain, Business Component Association
- Package row – instance (row) level functions (queries, cursors, individual rows and fields access, CRUD operations). SQL physical data mapped by classes DataRow and DataElement in this package into Java memory are used by the corresponding package row in the Controller to build application and presentation behaviours for that data.
- Package document – dynamic business documents
- Package process – business processes and workflows

An important Aspect of the Framework Package is the definition of User Actions, Business Events and Business Rules, and their connection to Web, HTML and Java runtime technology.

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 (or DynaForm)
- Package row – has the main class **Row.java**, and controls how 1 row of data (provided by DataRow class in Model) is presented in a block (e.g. an EditBlock has 1

Row, while a ListBlock or LookupBlock have many Rows).

- Package element – has the main class Element.java and provides an application support for one element (“attribute”, “column”) of a Row, based on data provided by the corresponding DataElement 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.).

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 behaviours
- Package ui represents a “User Interface Engine”, a common definition of presentation and user interaction functions
- Package html is the implementation of the User Interface Engine for the Web browser (HTTP, HTML and JavaScript) interface

Runtime

This package contains runtime technical components. An application may have specific technology dependencies, such as access to legacy data source via custom code, or biometric user authentication technology etc. 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

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 Framework is independent of patterns used to define particular classes of Rules.

The Rule Engine is composed of the Rule Framework and the Rules themselves. 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, system 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 Compo-

nent 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 KnowledgeFrame repository. This Java class must be derived from a common rule definition recognized by the KnowledgeFrame 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 signalled 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 (“SEVERITY_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.

New Rule Classes and Events can be defined, as required, by subclassing.

- 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 behavioural 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 KnowledgeFrame 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.

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

- New
 - Insert
 - Update
 - Delete
- UI Events
 - Paint
 - Process
 - Enter Page
 - Exit Page
 - Enter Module
 - Exit Module
 - Enter Lookup
 - Exit Lookup

- 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, Domain, Attribute, Process or Application Component (Module, Page, Block, Item) as appropriate.

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.