The Handly Core Framework: An Overview

By Vladimir Piskarev, 1C
June 2018

Introduction

The core framework of the Eclipse Handly project is a class library[1] that provides basic building blocks for handle-based models, with an emphasis on language-specific source code models of the Eclipse workspace. It allows creating highly scalable, robust, and thoroughly optimized models similar in design principles to the tried-and-tested Java model of Eclipse Java development tools while reducing programming effort, fostering software reuse, and enabling interoperability.

The framework is designed for flexibility and can be used to create a high-level source code model for practically any programming language, whether general-purpose or domain-specific; it is compatible with any parsing technology. The model implementor has complete control over the model's base-level API, including the ability to implement a preexisting handle-based model API. At the same time, the uniform meta-level API provided by the framework establishes a common language and makes it possible for IDE components to work in a generic way with any Handly-based model.

Background

Handle-based models employ a variation of the handle/body idiom, where clients have direct access only to ‘handles’ that act like a key to a model element and have the following principal characteristics:

- Immutable, equal by value
- Can define behavior of the element, but don't keep any element state beyond the key information. The element state beyond the key information is stored separately in an internal ‘body’
- Can refer to non-existing elements

Such design has a number of important properties:

- Handles are ‘stable’, you can freely keep references to them
- Handles are lightweight but can be rich in behavior
- Bodies can be ‘virtualized’ and computed on demand

This makes handle-based models highly scalable and perfectly suited to presenting in Eclipse IDE views such as Project Explorer, Search, Outline, etc.

It is surely not a coincidence that handle-based models are an important ingredient in the Eclipse IDE. The handle-based resource model of Eclipse workspace provides a common low-level foundation for language-specific development tools. The handle-based Java model, which wraps the workspace resource model and renders it from the Java language’s angle, is one of the pillars of Eclipse Java development tools (JDT). To a great extent, it is the Java model that makes possible seamless tool integration and unified user experience in

[1] Written in Java and made available under the Eclipse Public License 2.0

JDT. Meanwhile, models with design properties similar to those of the Java model can play an equally important role in Eclipse Platform-based development tools for other languages, as illustrated by the C model of Eclipse C/C++ Development Tooling (CDT) or the Script model of Eclipse Dynamic Languages Toolkit (DLTK). It is this class of models that is the main concern of the Handly core framework.

Design Motivation

Traditionally, handle-based models such as the JDT Java model or the CDT C model were built either entirely from scratch or by copying and modifying, with due consideration of possible licensing issues, the source code of a preexisting model. The traditional process required much effort, was tedious and error-prone. The resulting models were effectively silos with a completely isolated API, which prevented a possibility of developing reusable IDE components around those models, although the models did seem to have certain traits in common. This method, albeit deep-rooted, doesn't appear to be an attractive proposition, especially when dealing with languages on a large scale.

The Handly core framework aims to reduce programming effort, foster software reuse, and enable interoperability while retaining much of flexibility of the traditional approach. It provides building blocks for implementing handle-based models for language-specific Eclipse Platform-based development tools and also provides a uniform meta-level API that establishes a common language and makes the models accessible in a generic way to reusable IDE components. The provided implementation allows creating source code models on a par with the JDT Java model in quality for virtually any language or group of languages. It is compatible with any parsing technology and gives the implementor complete control over the model's base-level API.

Implementation Overview

The Handly core framework is based on a few fundamental concepts, with the 'conceptual parsimony' being an important design goal. For the distilled core abstractions the framework provides partial implementations that implementors of Eclipse Platform-based development tools can use to create, in a systematic way and with reduced effort, complete handle-based source code models similar in design principles to the JDT Java model, the CDT C model, and the like.

Fig.1 depicts main elements of the framework as an inheritance hierarchy (programmatic interfaces are on the left, corresponding partial implementations are on the right). It is these elements that are used (specialized) by the model implementor to define a specific model. As we'll see, there are almost no restrictions on the shape of attainable models; the framework can be used to create source code models for practically any programming language, whether general-purpose or domain-specific.

Model API

Typically, the model implementor would start by defining the API interfaces for model elements.

Consider a contrived programming language named Foo. The Foo *model* (a handle-based model of the workspace rendered from the Foo language's angle) might consist of Foo *projects* containing Foo source *files* that contain declarations of *variables* and *functions*.

The implementor of the Foo model would define `IFooElement` (the base interface for all Foo elements), `IFooModel`, `IFooProject`, `IFooFile`, `IVariable`, and `IFunction` as the API interfaces for *handles* to model elements (Fig.2). The interfaces defined by the model implementor not only provide static typing for model elements, but may contain methods specific to the corresponding element. For instance, the interface `IFooFile` might contain methods such as `getVariables()`, `getFunctions()`, etc.

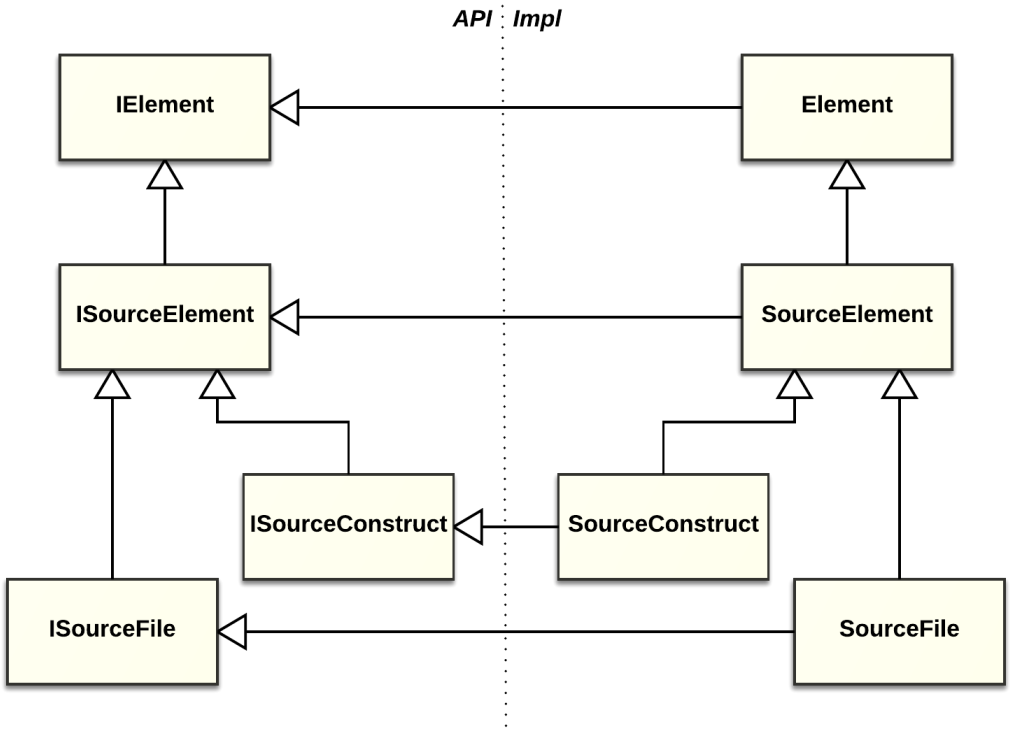

Fig.1. Main Elements of the Handly Core Framework

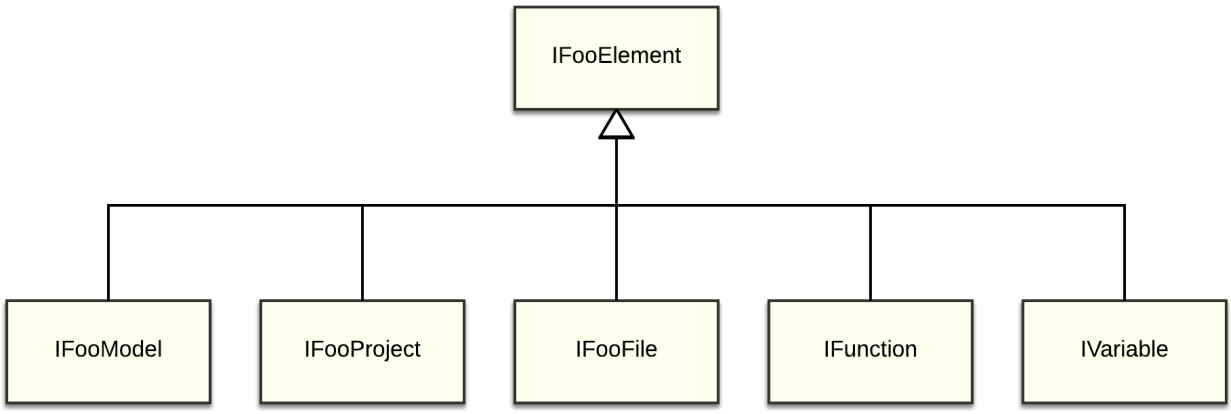

Fig.2. Handle Interfaces for Foo Elements

The implementor of a Handly-based model has a lot of freedom when defining the model API and may even try to implement a preexisting handle-based model API with the framework. In particular, the model interfaces are not required to extend any framework interfaces. However, it is usually a good idea for the model API to extend the relevant common interfaces for model elements, as Fig.3 shows, in order to make the model easier to use with APIs expressed in terms of the common element interfaces. Otherwise, explicit casts might be necessary when interacting with such APIs. Since each of the common interfaces for model elements is just a marker interface and contains no members, there is generally no drawback in extending them.

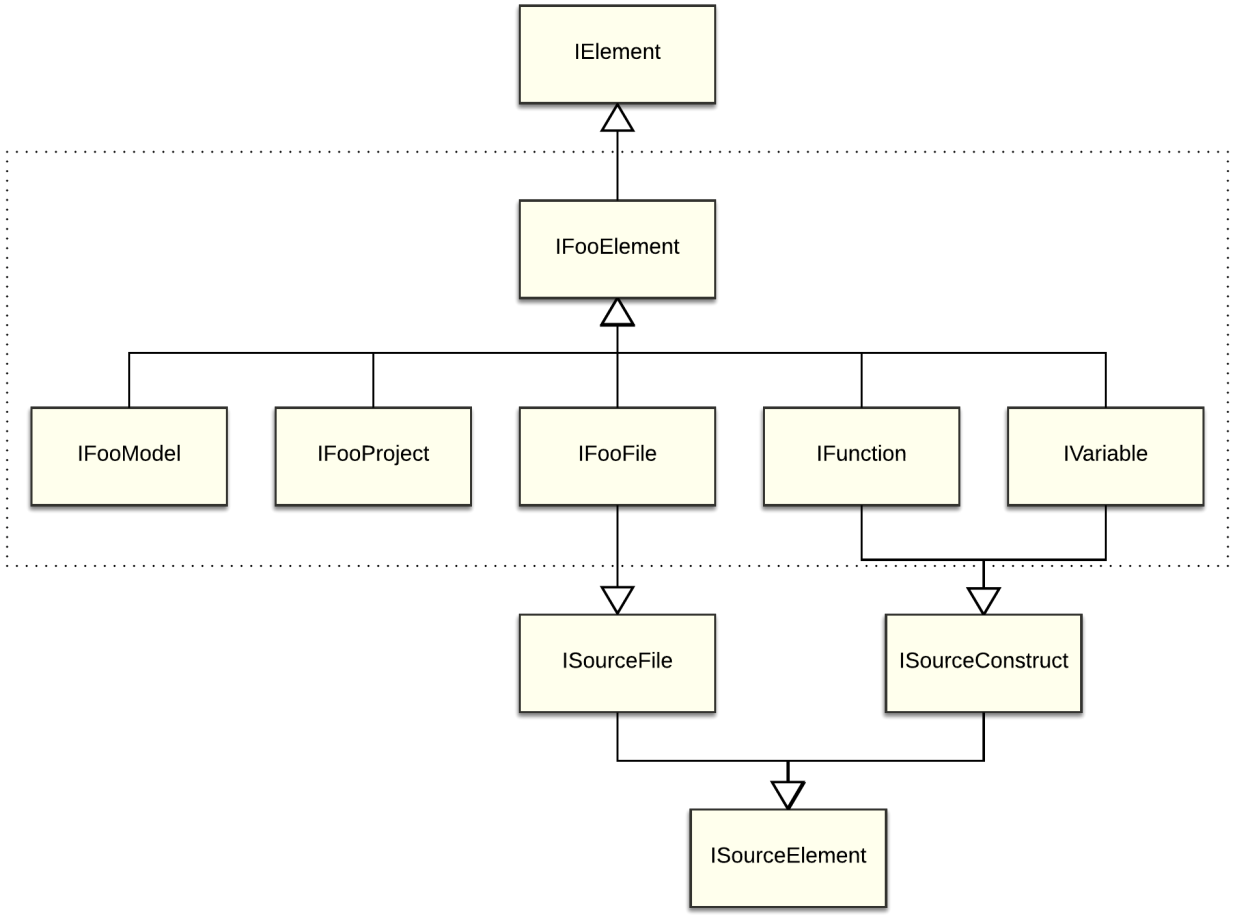

Fig.3. Using Common Interfaces as a Base for Foo Element Interfaces (Optional)

The framework also provides a set of ‘extension interfaces’ that extend the corresponding common interfaces and introduce a number of generally useful default methods for model elements, effectively acting like mix-ins. The model implementor may choose to mix the relevant extension interfaces into model element interfaces, as Fig.4 illustrates. The implementor always retains control over picking and choosing desirable extensions.

An important API expressed in terms of the common interfaces for model elements is the class `Elements` that provides static methods for generic access to elements of any Handly-based model. Taken together with the common element interfaces, this class serves as a uniform meta-level API to the models created with the framework and makes them accessible in a generic way to reusable IDE components.

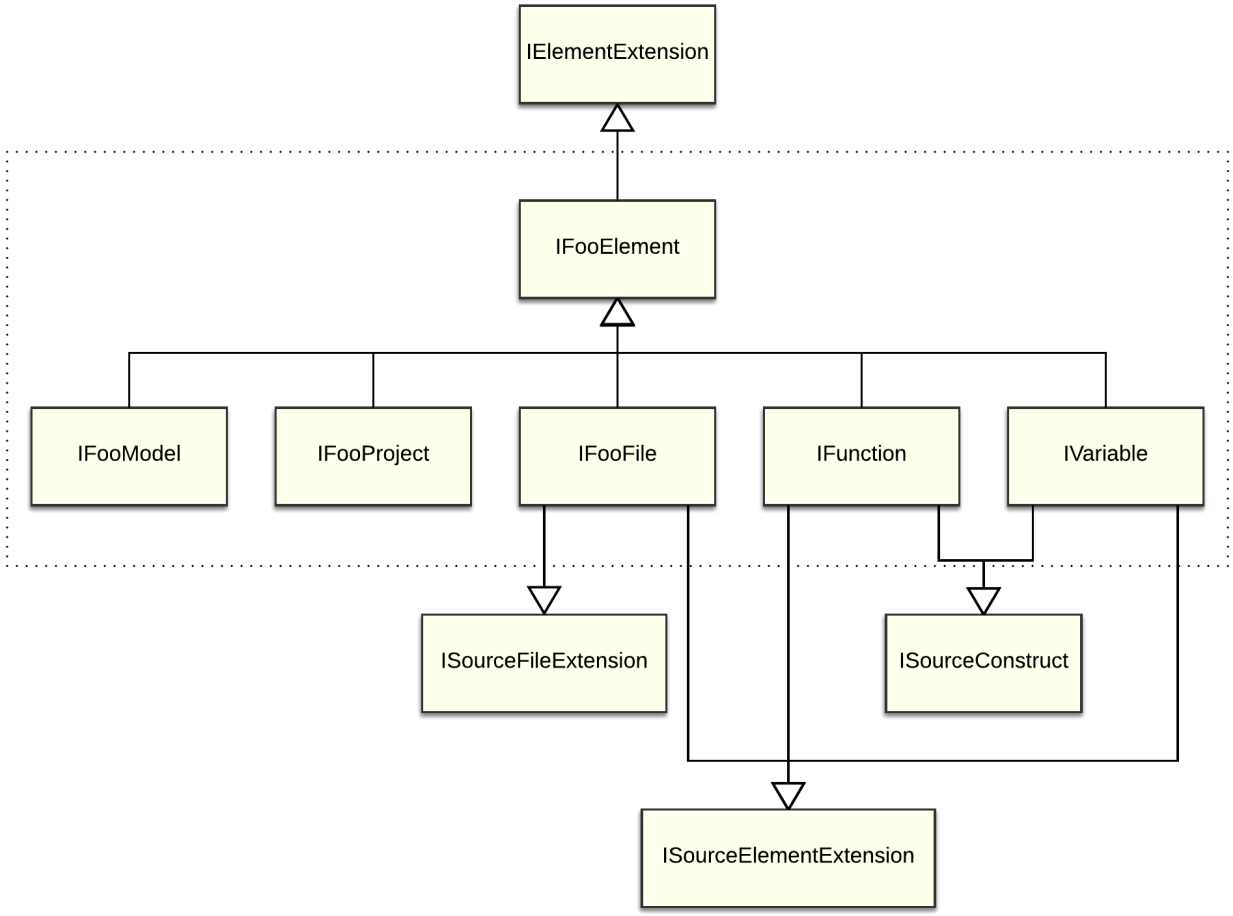

Fig.4. Mixing Extension Interfaces into Foo Element Interfaces (Optional)

Model Implementation

After the model API is defined, it is time to implement it using the core framework.

Continuing with our Foo model example, the classes `FooModel` (extends `Element` and implements `IFooModel`), `FooProject` (extends `Element` and implements `IFooProject`), `FooFile` (extends `SourceFile` and implements `IFooFile`), `Variable` (extends `SourceConstruct` and implements `IVariable`), and `Function` (extends `SourceConstruct` and implements `IFunction`) would be defined as the implementation classes for handles to model elements (Fig.5).

Sometimes, single inheritance is not expressive enough to allow the implementor to factor out all the common features shared by elements of a specific model while using skeletal implementation classes provided by Handly. For example, the implementor could factor out the common features of Foo elements into the class `FooElement`, which would extend `Element`. However, the class `FooFile` would then need to extend *both* `FooElement` and `SourceFile`, which is not possible with single inheritance. There are workarounds, such as explicitly duplicating features from `FooElement` in `FooFile`, or introducing an internal 'trait-like' interface with default methods (say, `IFooElementInternal`) in place of or in addition to the class `FooElement`. Since Handly 0.7, the core framework provides skeletal implementations for elements in a more flexible way with a number of 'trait-like' `*ImplSupport` interfaces extracted from the `Element` hierarchy. The new approach, based on Java 8 default methods, enables a form of multiple inheritance of behavior. The implementor may mix in default implementations from `*ImplSupport` interfaces directly or extend classes in the `Element` hierarchy. For example, the class `FooElement` could just extend `Element`, while the class `FooFile` could extend `FooElement` and implement (mix in) `ISourceFileImplSupport`.

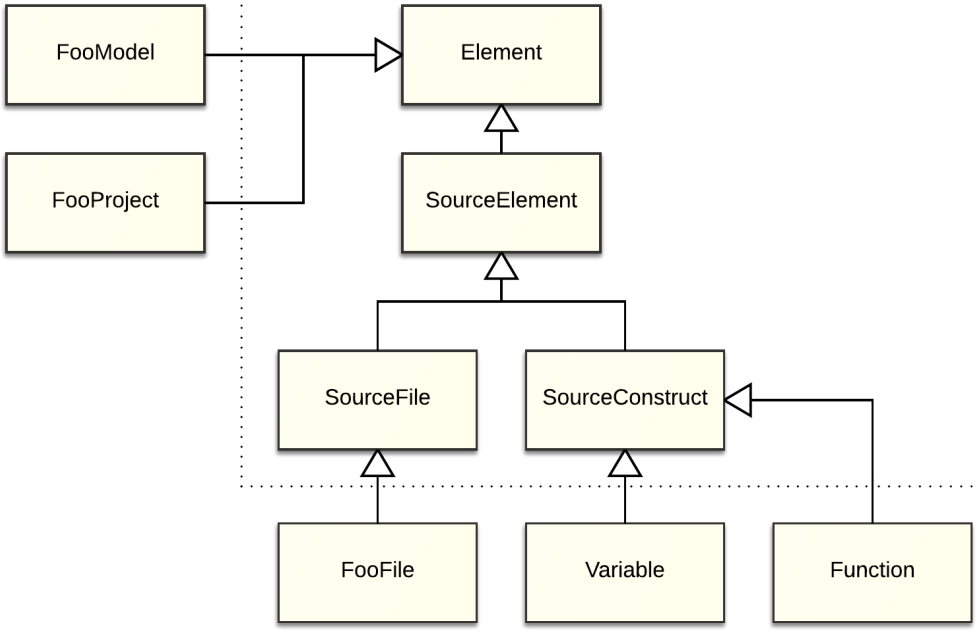

Fig.5. Handle Implementation Classes for Foo Elements

The skeletal implementations provided by the framework contain a few abstract methods that must be implemented by the model implementor. Those methods relate to a generalized implementation of the ‘handle/body’ idiom (Fig.6).

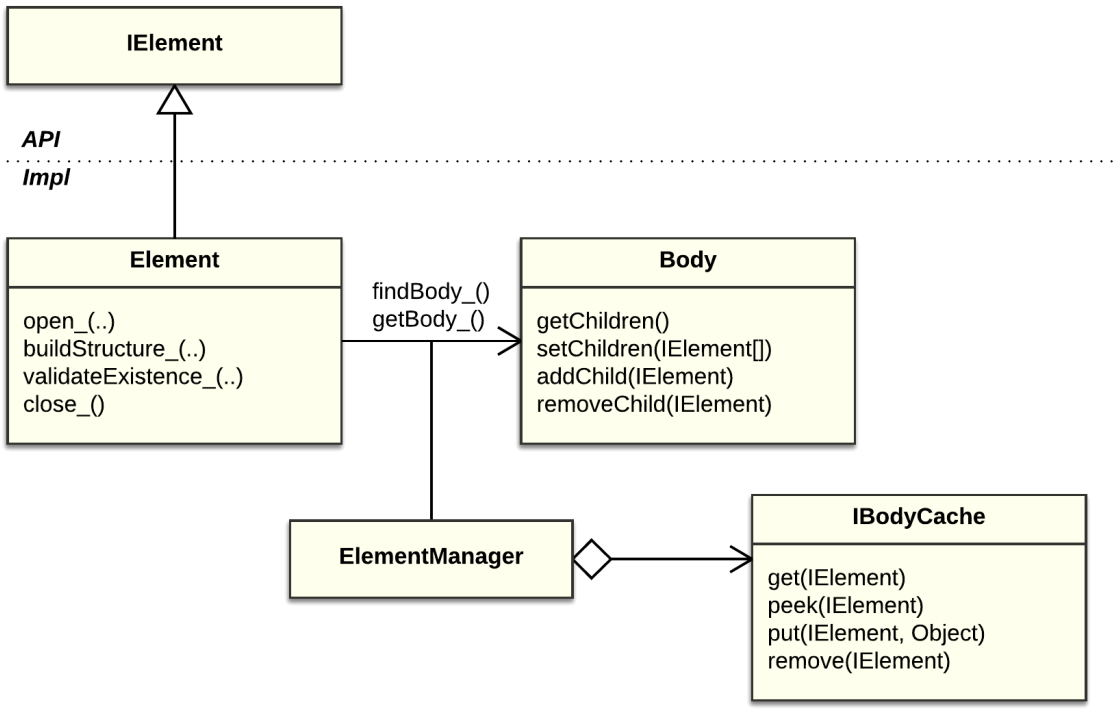

Fig.6. Generalized Implementation of Handle/Body

Objects of type `IElement` are ‘handles’ which contain only immutable ‘key’ information that is necessary and sufficient for unambiguous identification of the corresponding model element. All the other information about the element state (e.g., a list of the child elements) is stored in an instance of ‘body’. The class `Body` provides a

default implementation for element bodies, although any `Object` can be used as a body if necessary. Some methods pertaining to model elements — such as `getName_()` and `getParent_()` — are 'handle-only' in that they can be implemented based only on information contained in the handle itself, without accessing the element's body. Other methods — such as `getChildren_(IContext, IProgressMonitor)` — require accessing the body. The class `ElementManager` is responsible for maintaining the association of a model element's handle with the corresponding body. As a rule, each model has its own instance of the `ElementManager`. The `ElementManager` depends on an implementation of `IBodyCache` that must be supplied by the model implementor.

The framework provides deferred initialization of the model by computing element bodies on demand. When a body is needed, an element requests it from the corresponding `ElementManager`, and if the body is not found in the `IBodyCache` (the element is not 'open'), the framework invokes the element's `buildStructure_` method to create and initialize the body, and then asks the `ElementManager` to put the initialized body into `IBodyChache`. The abstract `buildStructure_` method is the core method that the model implementor needs to implement. For example, an implementation of this method in the class `FooModel` would create a new `Body` and initialize it with a list of the currently existing Foo projects as its children.

In this way, the framework 'opens' model elements (computes their bodies) automatically, as the need arises. Here, of significance is the notion of an 'openable' element, capable of opening itself on demand by providing its own implementation of `buildStructure_`. When opening any model element, the framework first opens its 'openable parent' if it is not already open. However, child elements are never opened by the framework automatically when opening an element. These rules ensure the correct order and the deferred nature of the model's initialization.

The `close_()` method closes an element together with its descendants by removing the relevant information from the body cache. An element that has been closed can be re-opened on demand if it still exists. The `close_()` method, along with methods `findBody_()` and `Body.addChild/removeChild` for incremental structure updates, provide the basis for updating stale information in the model, which needs to be done, for instance, in response to resource change notifications. (The actual mechanism for updating the model when the underlying resources change is model-specific and cannot be effectively generalized in the core framework. It is responsibility of the model implementor to produce such a mechanism.)

As noted above, the model implementor is responsible for supplying an implementation of the interface `IBodyCache`. In that way, a caching strategy optimal for a particular model can be implemented. Advanced implementations of this interface might be based on a bounded LRU cache that automatically removes least recently used elements when a certain memory threshold has been exceeded. Such caching strategy is possible because, as discussed above, if the requested body is not found in the cache, the model can always re-compute it afresh. This permits to put a constraint on the amount of memory consumed by the model.

Source Elements and Source Element Info

The framework introduces the concept of 'source elements' to represent elements that may have associated source code. The interface `ISourceElement` extends `IElement` and is further specialized by interfaces `ISourceFile` and `ISourceConstruct` (Fig.1). The interface `ISourceFile` represents a source file, whereas `ISourceConstruct` represents an inner source element. Characteristics that are specific to source elements are represented by the interface `ISourceElementInfo` (Fig.7).

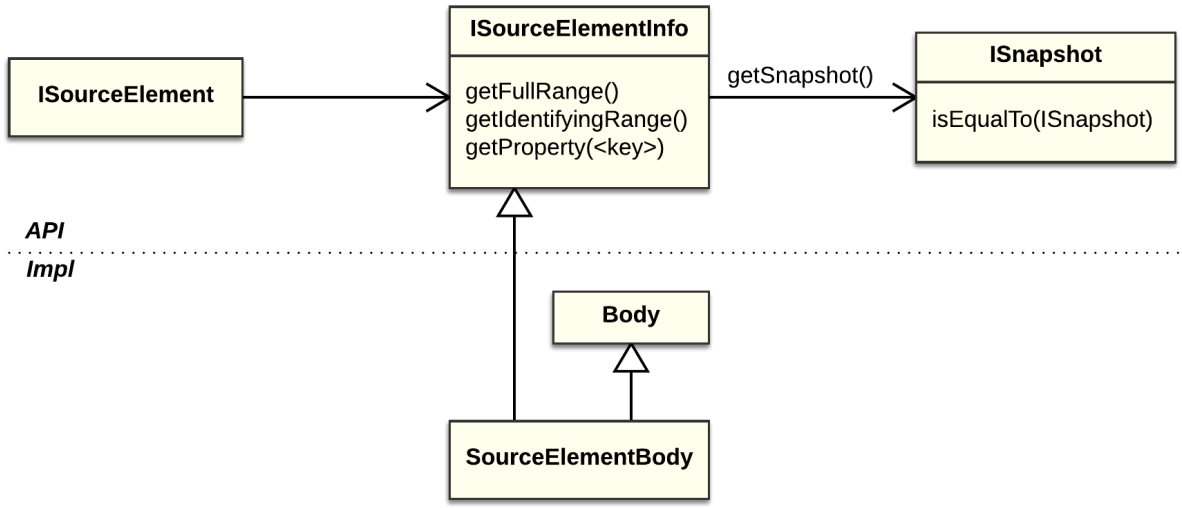

Fig.7. Source Element Info

An instance of source element info corresponds to a snapshot of the source file's text contents. This snapshot is available as an `ISnapshot` object associated with source element info. `ISourceElementInfo` provides methods for accessing the text range and other properties of the source element in the associated snapshot. The class `SourceElementBody` (a subclass of `Body`) provides a default implementation of source element info, which can be extended by the model implementor if needed.

Source Files, Buffers, and Working Copy Support

One of the key features of the framework is a generalized support for model elements representing source files (Fig.8).

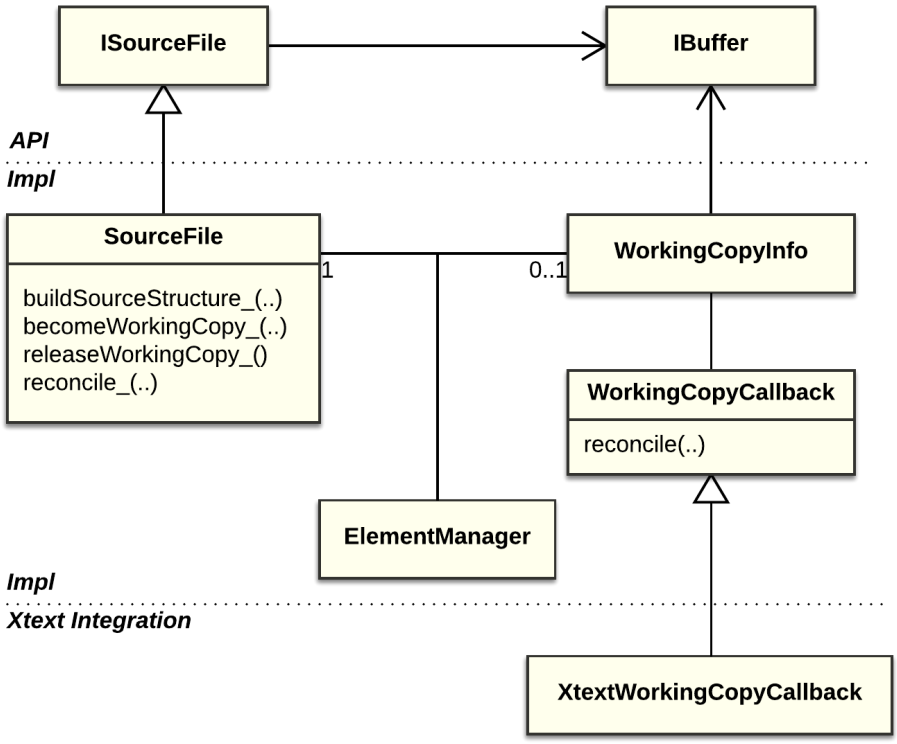

Fig.8. Source File, Buffer, and Working Copy Support

Elements inside a source file (source constructs) are not ‘openable’ (their `buildStructure_` method should never be called and always throws an error); instead, the source file itself (their ‘openable parent’) is responsible for building handle/body relationships for all elements of its inner structure in `buildSourceStructure_` method.

A shared buffer can be obtained for accessing and manipulating the text contents of the source file. The `IBuffer` interface provides methods for obtaining the current `ISnapshot` of the buffer, modifying the buffer’s text contents, and saving the buffer to the underlying file (Fig.9).

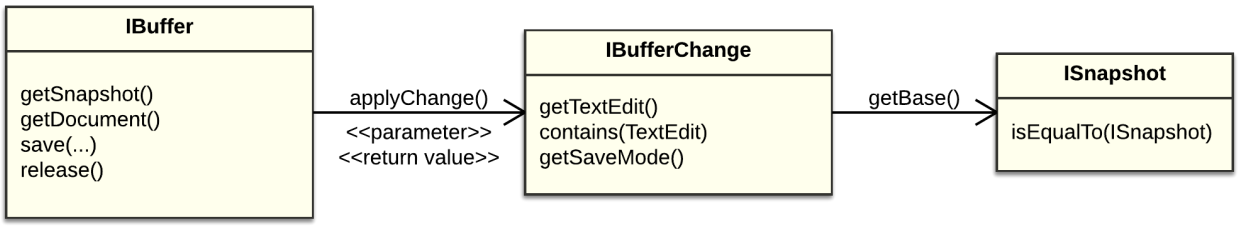

Fig.9. Buffer API

Modifications of the buffer’s contents are represented by `IBufferChange` objects. A `TextEdit` (from Eclipse Platform Text) associated with the `IBufferChange` describes the text manipulation operation. In addition, an `ISnapshot` may be associated with the `IBufferChange`. This is the snapshot on which the change was based. It serves as an ‘optimistic lock’ mechanism: if, when attempting to apply a change to the buffer (via the `applyChange` method), the current snapshot of the buffer is not equal to the snapshot on which the change was based, the change is rolled back with a `StaleSnapshotException`. If the change was applied successfully, an `IBufferChange` is returned that can be used to undo the applied change. A `SaveMode` (one of the `KEEP_SAVED_STATE`, `FORCE_SAVE`, or `LEAVE_UNSAVED` values) indicates whether the buffer has to be saved after the change was successfully applied to it.

The framework provides comprehensive ‘working copy’ support (Fig.8). Working copy is a special mode of the source file, where its state as a model element reflects the contents of a buffer explicitly associated with it, rather than the contents of the underlying file. In that way, the model can ‘see’ the unsaved contents of a source file being edited. Generic clients can reconcile a working copy with the current contents of its buffer at any time by invoking the static `reconcile` method of the class `Elements`.

The method `becomeWorkingCopy_` switches the source file to working copy mode by associating it with a `WorkingCopyInfo`, while the method `releaseWorkingCopy_` switches it back to ‘normal’ mode. An instance of `WorkingCopyInfo` holds information related to a working copy, such as a reference to the working copy buffer. Internally, the class `ElementManager` is responsible for maintaining the source file’s association with a `WorkingCopyInfo`.

Optionally, clients can pass an instance of `IWorkingCopyCallback` to the `becomeWorkingCopy_` method. The working copy callback receives notifications related to lifecycle of the working copy and intercepts calls related to reconciling of the working copy. The class `DefaultWorkingCopyCallback` provides a default implementation of the working copy callback. The class `XtextWorkingCopyCallback` is a specific implementation of `IWorkingCopyCallback` used in the Handly Xtext Integration Layer.

Model Change Notifications and Element Delta

The framework provides a generalized support for change notifications in a Handly-based model (Fig.10).

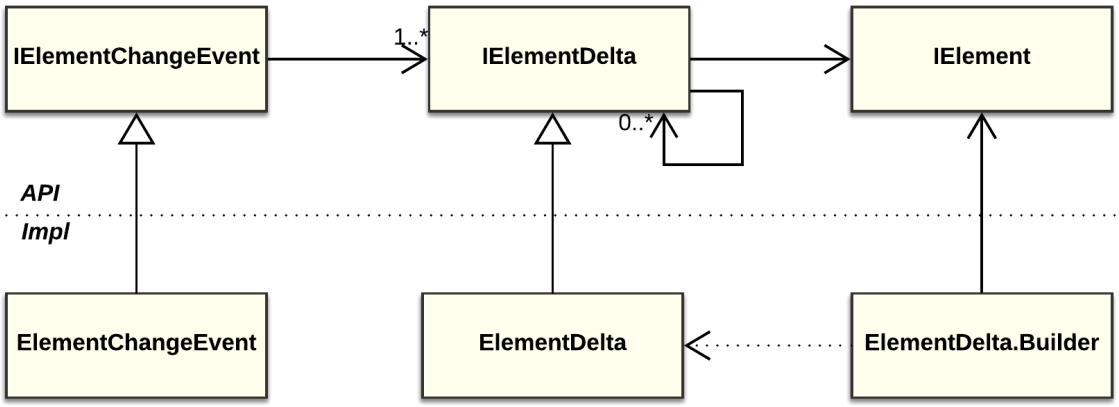

Fig.10. Generalized Representation of Model Change Notifications

The marker interface `IElementDelta` represents an elementary model change (addition, removal, or a change of a model element) that can be introspected in a generic way with static methods provided in the class `ElementDeltas`. Taken together, `IElementDelta` and `ElementDeltas` form part of the meta-level API provided by the framework. Element deltas support recursive composition (i.e., a delta for some element can contain deltas for its child elements); in this way, any model change can be represented as a 'delta tree'.

The class `ElementDelta` provides a default implementation of element delta, which can be extended by the model implementor as needed. In particular, the implementor might want to define a delta interface specific to the model. Just as in the case of model elements, the implementor has a lot of freedom in defining the model-specific delta interface; in particular, extending `IElementDelta` is recommended, but is not required. `IElementDeltaExtension` is an 'extension interface' that extends `IElementDelta` with a number of common default methods, which the implementor may opt to mix into the model-specific delta interface.

The class `ElementChangeRecorder` records changes in the state of an element tree between two discrete points in time and produces an element delta representing the changes. A delta tree can also be built from scratch with `ElementDelta.Builder`.

Conclusion

Using the Handly core framework, implementors of Eclipse Platform-based development tools can create handle-based source code models for practically any language while reducing programming effort, fostering software reuse, and enabling interoperability. In a nutshell, the process takes the following steps:

1. **Implement the 'handle part' of the model.** Using the core framework, the model implementor defines the API interfaces and the implementation classes for handles to model elements (Fig.1-5). There are almost no restrictions on the shape of attainable models. The implementor has complete control over the model's base-level API while the uniform meta-level API provided by the framework establishes a common language and makes it possible to work in a generic way with any Handly-based model.

2. **Implement the 'body part' of the model.** The implementor provides implementations for the inherited abstract methods (such as `buildStructure_`) and supplies an implementation of `IBodyCache` to complete partial implementation of the 'handle/body' idiom provided by the framework (Fig.6, 7). Any

suitable parsing technology can be used when building the structure of a source file. The body cache permits to put a constraint on the amount of memory consumed by the model.

3. **Implement a resource change listener for the model.** The implementor produces a model-specific mechanism for updating the model when the underlying resources change. The framework provides a number of methods (such as `close_`, `findBody_`, and `Body.addChild/removeChild`) that can be used for updating stale information in the model (Fig.6). Model changes can be accompanied by corresponding notifications (Fig.10).

4. **Implement integration of the model with the text editor(s) for source files.** Using the working copy facility provided by the core framework (Fig.8), the model implementor produces an editor-specific mechanism for reconciling the model with the current contents of a source editor. Integration with the Xtext editor is already implemented in Handly Xtext Integration Layer and requires only a simple binding in the language-specific Xtext UI module.

Forward Pointers

- The project's website, https://www.eclipse.org/handly/
- A step-by-step getting started guide, https://github.com/pisv/gethandly/wiki, made available under EPL-2.0
- The project's source code, https://projects.eclipse.org/projects/technology.handly/developer, including exemplary implementations, such as a basic Xtext-based example and a more advanced Java model example
- An experimental fork of Eclipse Java development tools based on Handly, https://github.com/pisv/jdt.core-handly