Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Programmer's Activity Acquisition and Persistence in Eclipse

Semestral Project

Authors: Samuel Molnár and Pavol Zbell

Field: Information Systems

Course: Object-Oriented Analysis and Design of Software Systems

Supervisor: Ing. Ivan Polášek, PhD.

May, 2014

Contents

List of Figures

List of Tables

List of Examples

1 Introduction

In this work, we analyze an existing software system for acquisition and persistence of programmer's activities in an integrated development environment – *PerConIK Eclipse Intergration*[1]. As the name indicates, this software is a part of the *PerConIK*[2] (Personalized Conveying of Information and Knowledge) [2] research project and built as an extension atop the *Eclipse Platform*[3] focusing mostly on *Java*[4] programmers. It provides robust and extensible architecture for listening to and processing native *Eclipse* events.

1.1 Domain Model

For better understanding of the analyzed software we briefly describe selected entities:

Activator	Activates *Eclipse* plug-in, Eclipse class.
Core Service	Holds a *Provider* and a *Manager*.
Executor	Executes a *Runnable* in a thread, standard Java class.
Listener	Listens to events propagated by *Resources*.
Lookup	Reflective utility used to instantiate *Registrables*.
Manager	Manages registration and unregistration of *Registrables*.
Native Listener	Listens to events on *Native Objects*, Eclipse interface.
Native Object	Produces events, Eclipse object.
Provider	Resolves *Registrable* implementation types and instances.
Registrable	Either a *Resource* or a *Listener*.
Resource	Wrapper around a *Native Object*, propagates events to *Listeners*.
Runnable	Executable fragment, standard Java interface.
Services Loader	Loads *Core Services* at startup.
Services Snapshot	Snapshots *Core Services* state.
Watcher Service	Responsible for event persistence.
Watcher Operation	Notifies *Watcher Service* about an event.

[1] PerConIK Eclipse Integration: http://perconik.github.io
[2] PerConIK Research Project: http://perconik.fiit.stuba.sk
[3] Eclipse Platform: http://eclipse.org
[4] Java Programming Language: http://oracle.com/technetwork/java

2 Model Diagrams

We selected several interesting parts of the system and visualized their workings in modeling diagrams – use case, sequence, activity and state – spread across whis chapter.

2.1 Use Case Diagrams

2.1.1 Eclipse Startup

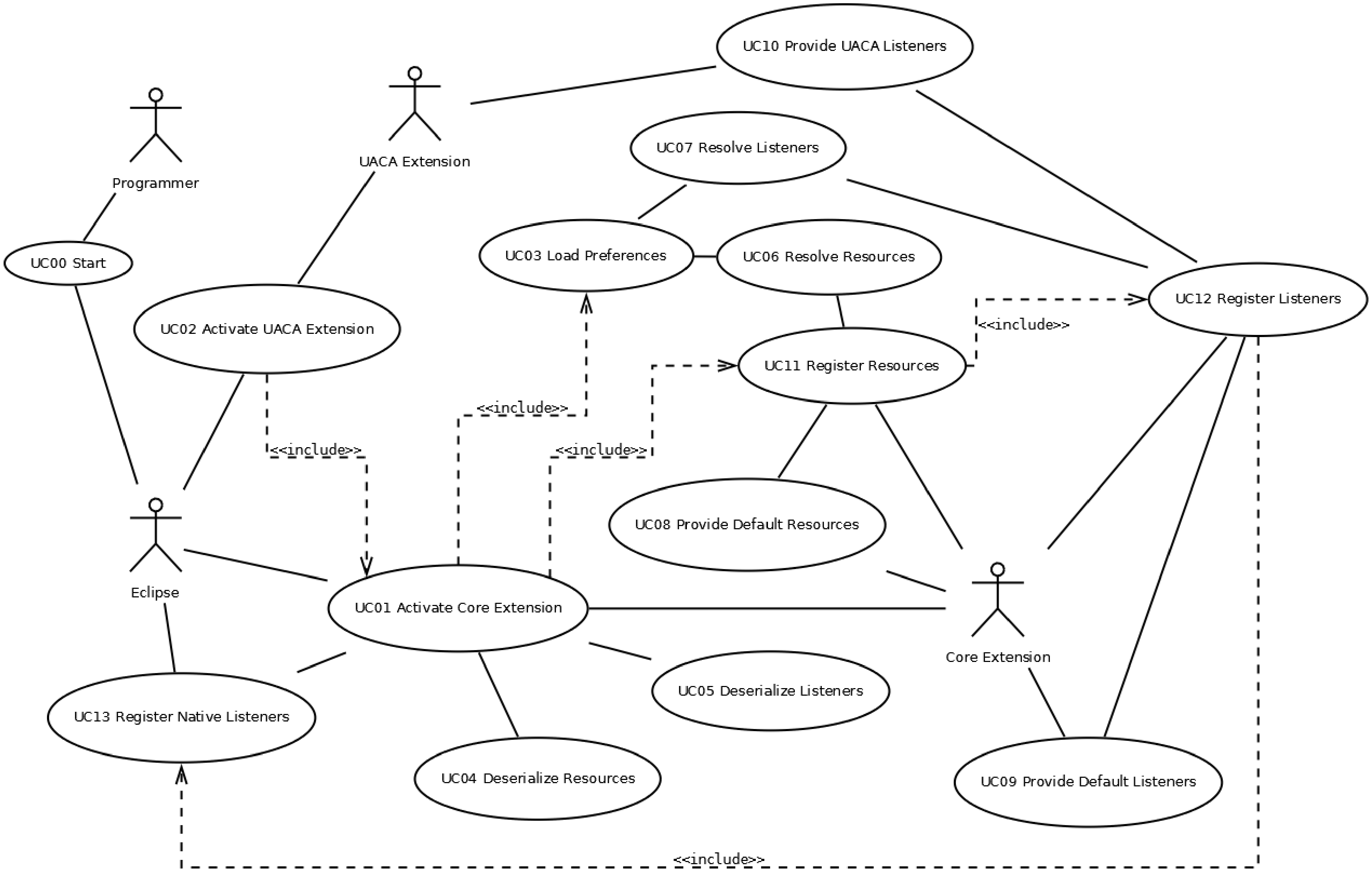
Eclipse startup use case diagram in figure 2.1 represents processes that handle initialization and registration of resources and their respectful listeners. It also represents interaction between *User Activity Central Application*[1] (UACA) Extension and Eclipse APIs for registering native listeners and loading user's preferences for accessing serialized listeners and resources from previous Eclipse session. List of use cases is summarized in table 2.1.

Table 2.1: *Use cases at Eclipse Startup*

Id.	Name	Description
UC00	*Eclipse Startup*	Programmer starts Eclipse IDE
UC01	*Activate Core Extension*	Eclipse activates PerConIK Core plug-ins
UC02	*Activate UACA Extension*	Eclipse activates PerConIK UACA plug-ins
UC03	*Load Preferences*	Core plug-in loads preferences
UC04	*Deserialize Resources*	Core plug-in deserializes resources
UC05	*Deserialize Listeners*	Core plug-in deserializes listeners
UC06	*Resolve Resources*	Core plug-in resolves active resources
UC07	*Resolve Listeners*	Core plug-in resolves active listeners
UC08	*Provide Default Resources*	Core plug-in obtains known resources
UC09	*Provide Default Listeners*	Core plug-in obtains known listeners
UC10	*Provide UACA Listeners*	UACA plug-in obtains knows listeners
UC11	*Register Resources*	Core plug-in registers active resources
UC12	*Register Listeners*	Core plug-in registers active listeners
UC13	*Register Native Listeners*	Core plug-in registers native listeners via Eclipse Platform APIs

[1] User Activity Central Application: http://github.com/perconik/uaca

Figure 2.1: *Use Case Diagram – Eclipse Startup*



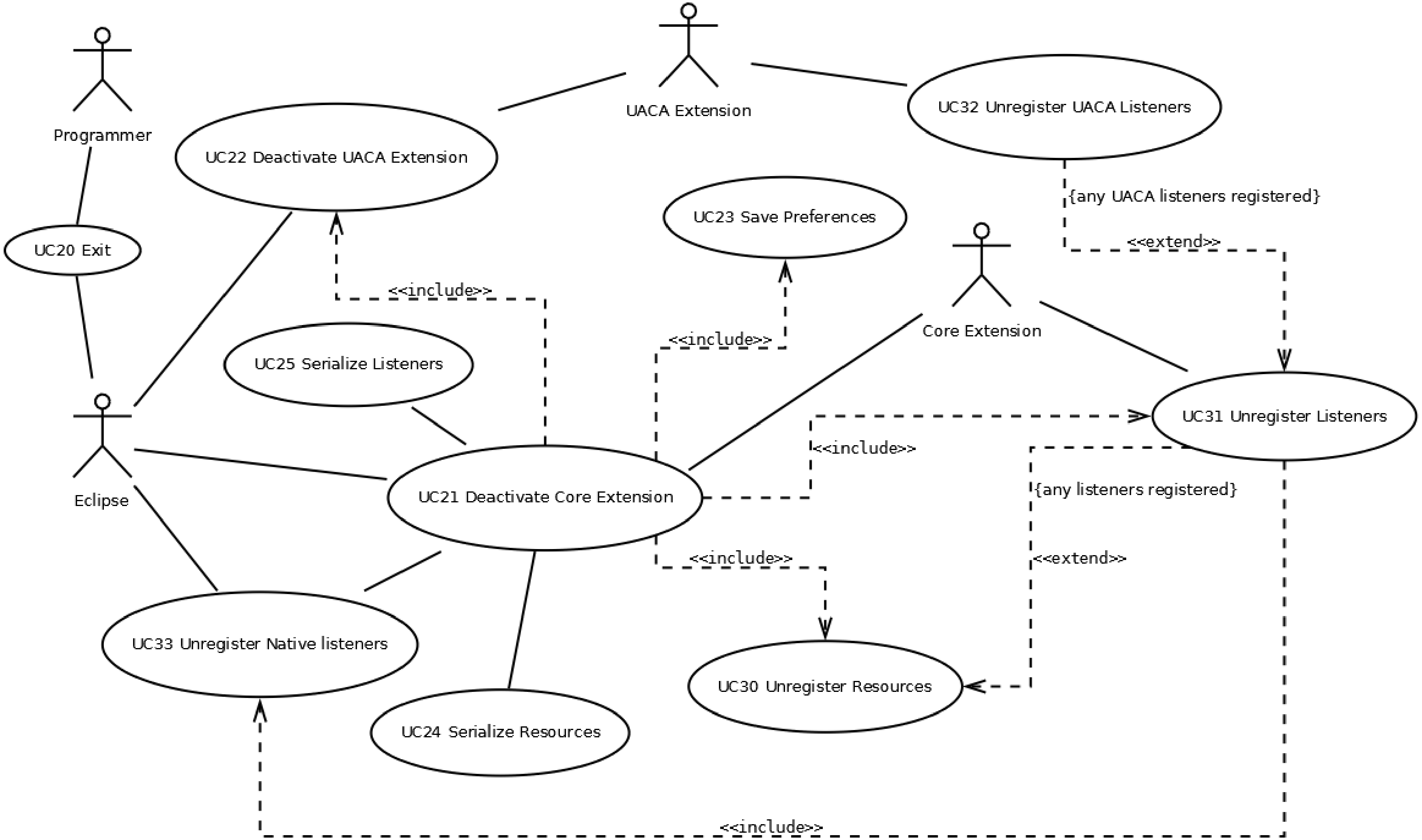
2.1.2 Eclipse Shutdown

Use case diagram for Eclipse shutdown in figure 2.2 depicts interaction between UACA Extension and Core Extension for Eclipse APIs. When user decides to exit Eclipse, the Core Extension has to deactivate itself and unregister every registered listener and it's respectful resources. Before exiting Eclipse completely, Core Extension needs to save preferences in order to reinitialize environment when Eclipse starts next time and reload serialized resources and listeners. List of all use cases for Eclipse Shutdown is summarized in table 2.2.

Table 2.2: *Use cases at Eclipse Shutdown*

Id.	Name	Description
UC20	*Eclipse Shutdown*	Programmer exits Eclipse
UC21	*Deactivate Core Extension*	Eclipse deactivates PerConIK Core plug-ins
UC22	*Deactivate UACA Extension*	Eclipse deactivates PerConIK Core plug-ins
UC23	*Save Preferences*	Core plug-in saves preferences
UC24	*Serialize Resources*	Core plug-in serializes resources
UC25	*Serialize Listeners*	Core plug-in serializes listeners
UC30	*Unregister Resources*	Core plug-in unregisters active resources
UC31	*Unregister Listeners*	Core plug-in unregisters active internal listeners
UC32	*Unregister UACA Listeners*	Core plug-in unregisters active UACA listeners
UC33	*Unregister Native Listeners*	Core plug-in registers active native listeners via Eclipse APIs

Figure 2.2: *Use Case Diagram – Eclipse Shutdown*



2.1.3 Event Processing

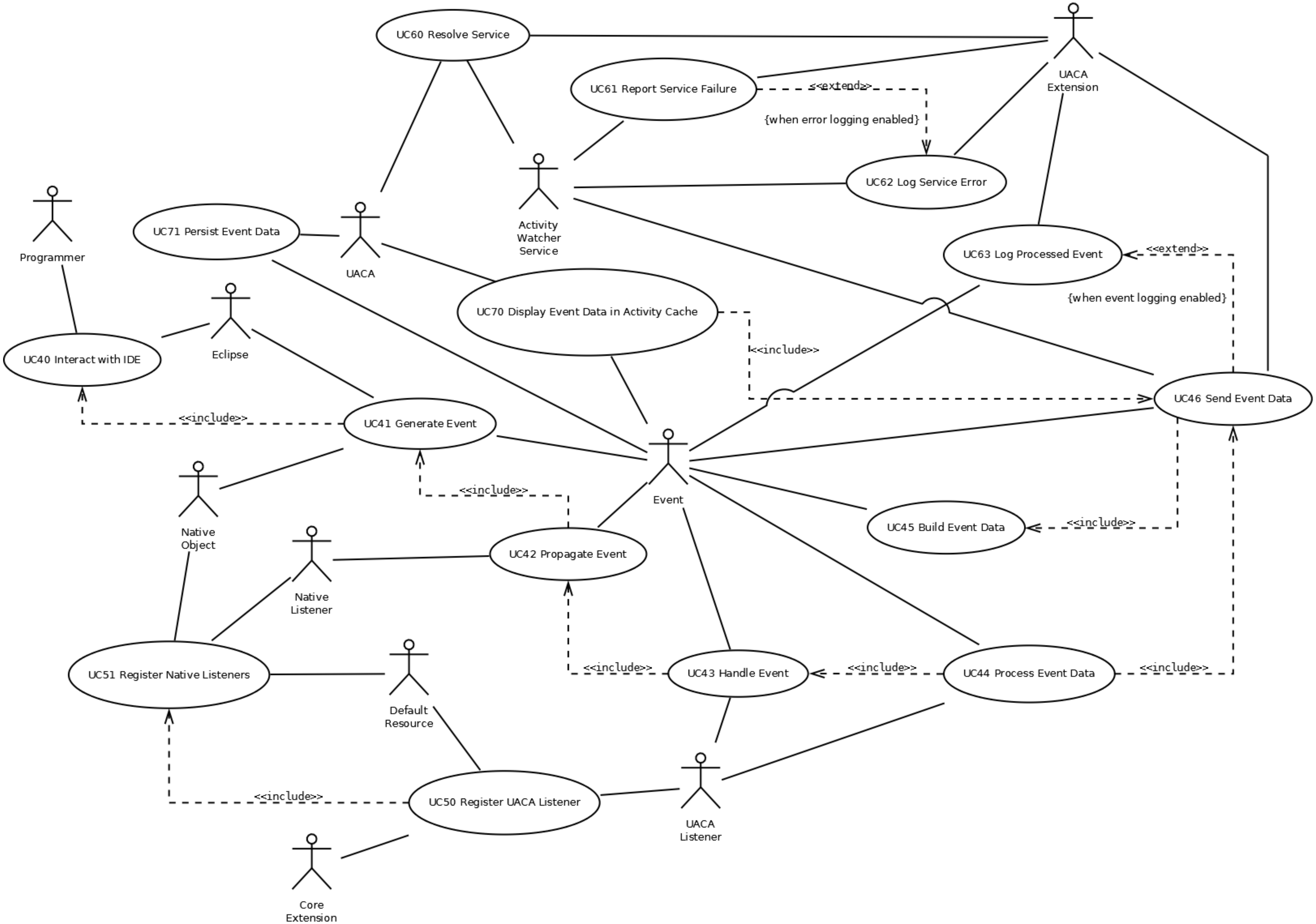
Programmer's activity and interaction with Eclipse resources (e.g documents or source trees) generates events on those resources. Every resource provides a way of registering listeners for such events. Use case diagram for Event Processing in figure 2.3 represents how are these events processed by the Core and UACA Extensions and shows a real example of a registered generic UACA Listener.

Processing of every event requires that events is, as first, propagated to other listeners. Every listener then processes data of the native event object – it builds an event data structure and send it for further processing. Service resolution reporting while UACA persists data is a necessary step for managing full-stack cooperation between UACA Extension and Core Extension for Eclipse APIs. List of use case diagrams for Event Processing is summarized in table 2.3.

Table 2.3: *Use cases of Event Processing*

Id.	Name	Description
UC40	*Interact with IDE*	Programmer's interacts with Eclipse IDE
UC41	*Generate Event*	Eclipse Native Object generates an event
UC42	*Propagate Event*	Eclipse Native Listener propagates an event to UACA Listener
UC43	*Handle Event*	UACA Listener handles propagated event
UC44	*Process Event Data*	UACA Listener processes event data
UC45	*Build Event Data*	UACA Listener builds event data transfer object
UC46	*Send Event Data*	UACA Listener sends event data
UC50	*Register UACA Listener*	Core plug-in registers UACA Listener
UC51	*Register Native Listeners*	Core plug-in registers a native listener for the UACA Listener
UC60	*Resolve Service*	UACA plug-in resolves Activity Watcher Service
UC61	*Report Service Failure*	UACA plug-in reports Activity Watcher Service failure
UC62	*Log Service Error*	UACA plug-in logs Activity Watcher Service error
UC63	*Log Processed Event*	UACA plug-in logs processed event
UC70	*Display Event Data in Activity Cache*	UACA displays event data in Activity Cache
UC71	*Persist Event Data*	UACA persists event data into remote storage

Figure 2.3: *Use Case Diagram – Event Processing*

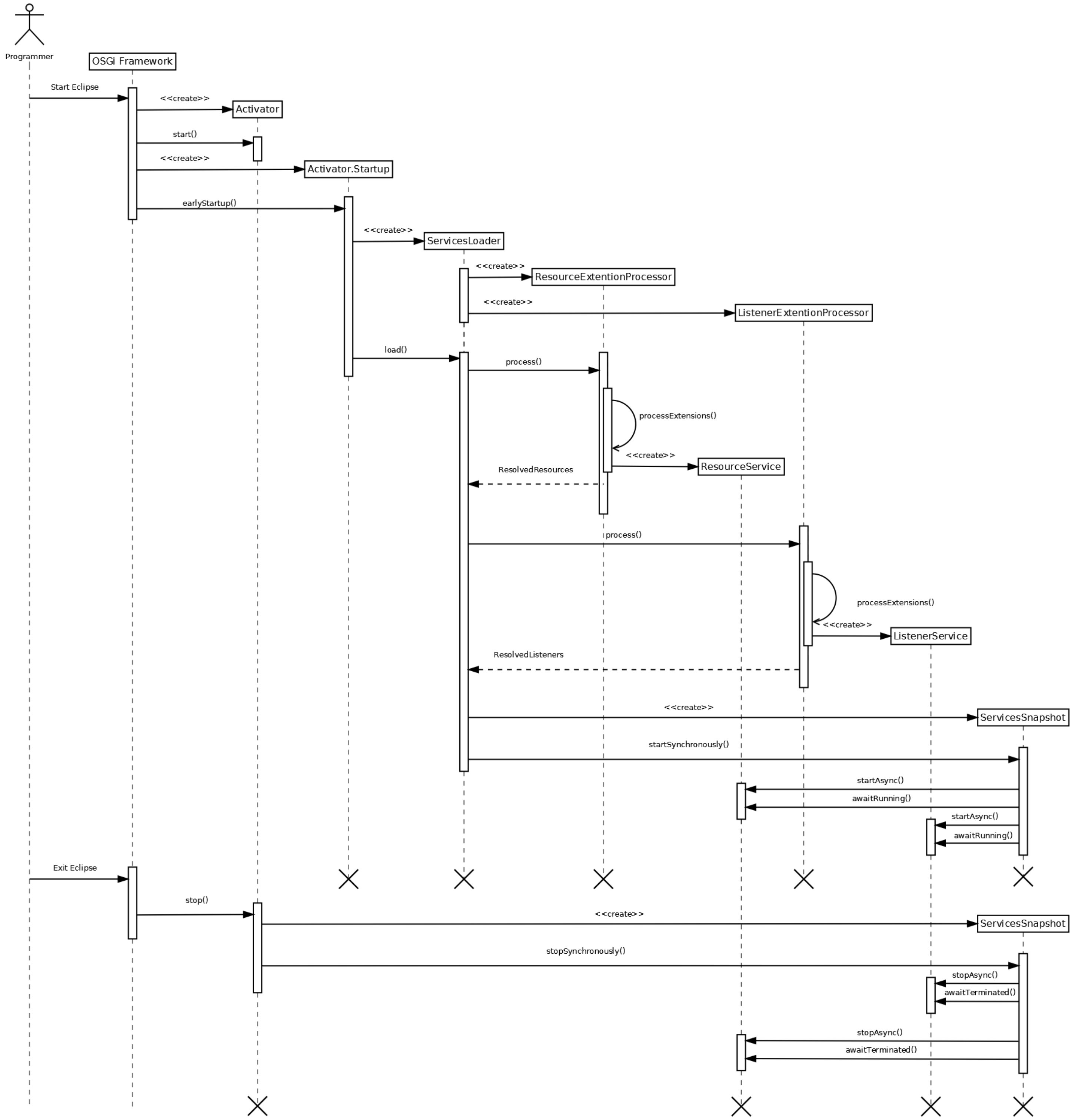


2.2 Sequence Diagrams

2.2.1 Eclipse Startup

Sequence diagram for Eclipse Startup in figure 2.4 represents loading and resolution of Core Services – Resource Service and Listener Service. Both services are started (or stopped on exit) asynchronously in another thread utilizing a Services Snapshot instance.

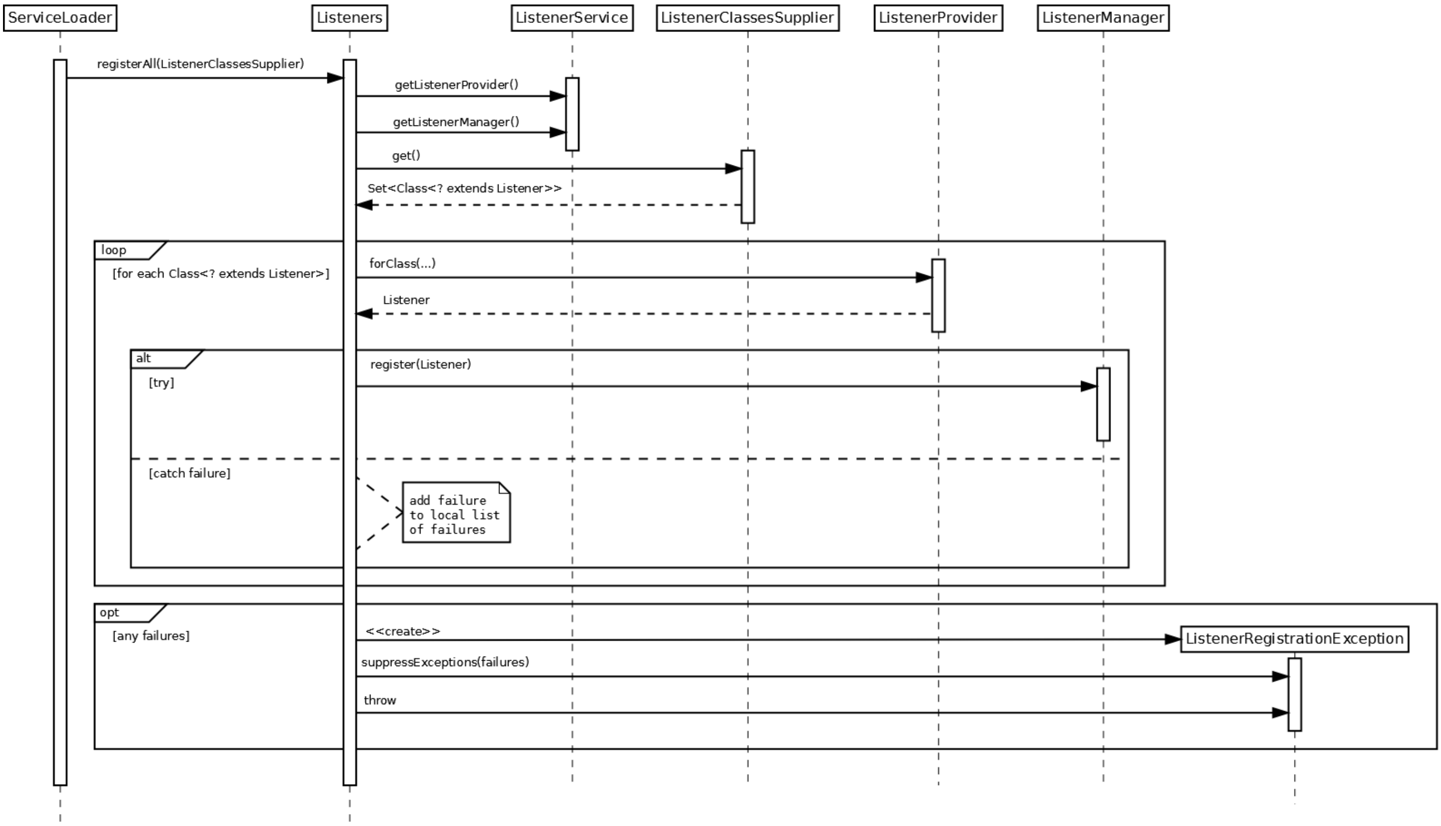
Figure 2.4: *Sequence Diagram – Eclipse Startup*



2.2.2 Registering Listeners on Startup

Sequence diagram for Registering Listeners on Startup in figure 2.5 shows how each listener is provided and then registered to it's respectful resources utilizing listener provider and manager instances. The provider provides a listener implementation class and manager handles the registration of a listener instance.

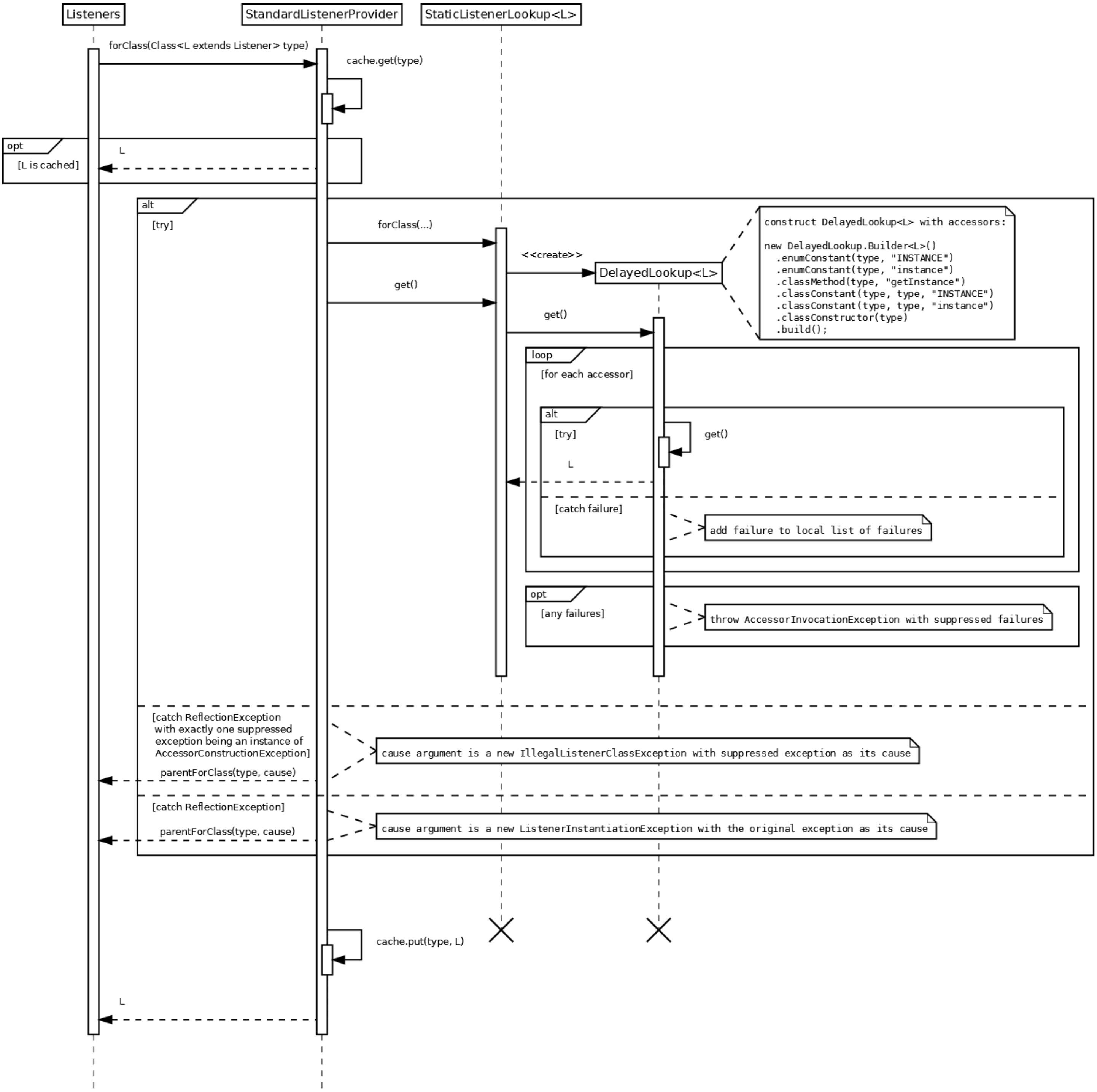
Figure 2.5: *Sequence Diagram – Registering Listeners on Startup*



2.2.3 Listener Resolution

Sequence diagram for Listener Resolution in figure 2.6 represents how a listener instance is resolved for a particular listener type. Listener provider provides the listener instance similarly to a standard Flyweight. Therefore, if a listener implementation class and the actual instance for the listener type are already resolved, it provides them from cache. Otherwise, as we can see in *alt* block in figure 2.6, the listener is resolved using delayed lookup based on reflection for resolving class instances using reflective accessors at runtime on the listener implementation class.

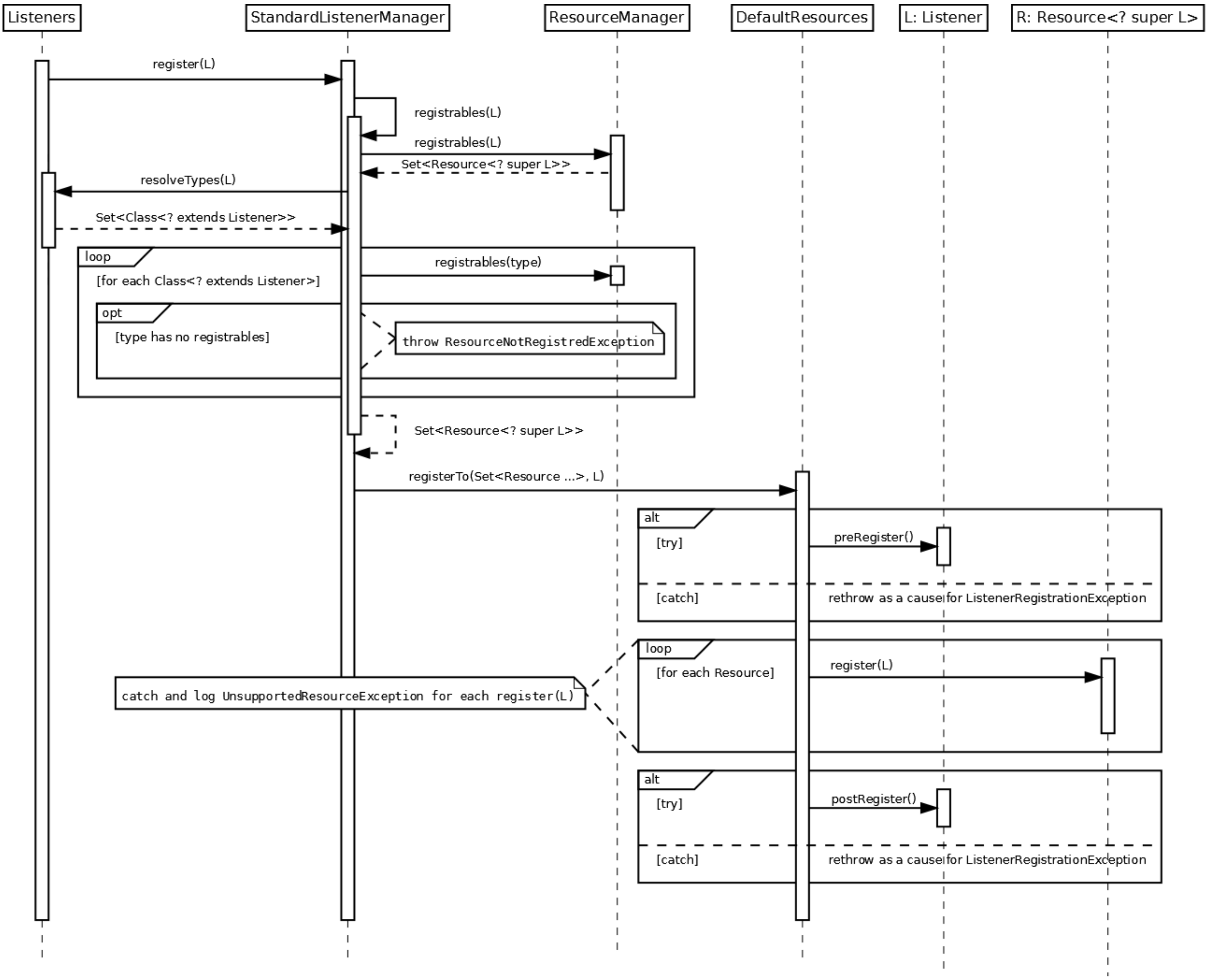
Figure 2.6: *Sequence Diagram – Listener Resolution*



2.2.4 Listener Registration

Sequence diagram for Listener Registration in figure 2.7 represents a process of how a listener is registered. Listener manager is responsible for registering the listener for every registrable (suitable) resource. Listener itself provides standard hooks for handling logic before and after registering on the registrable resource.

Figure 2.7: *Sequence Diagram – Listener Registration*

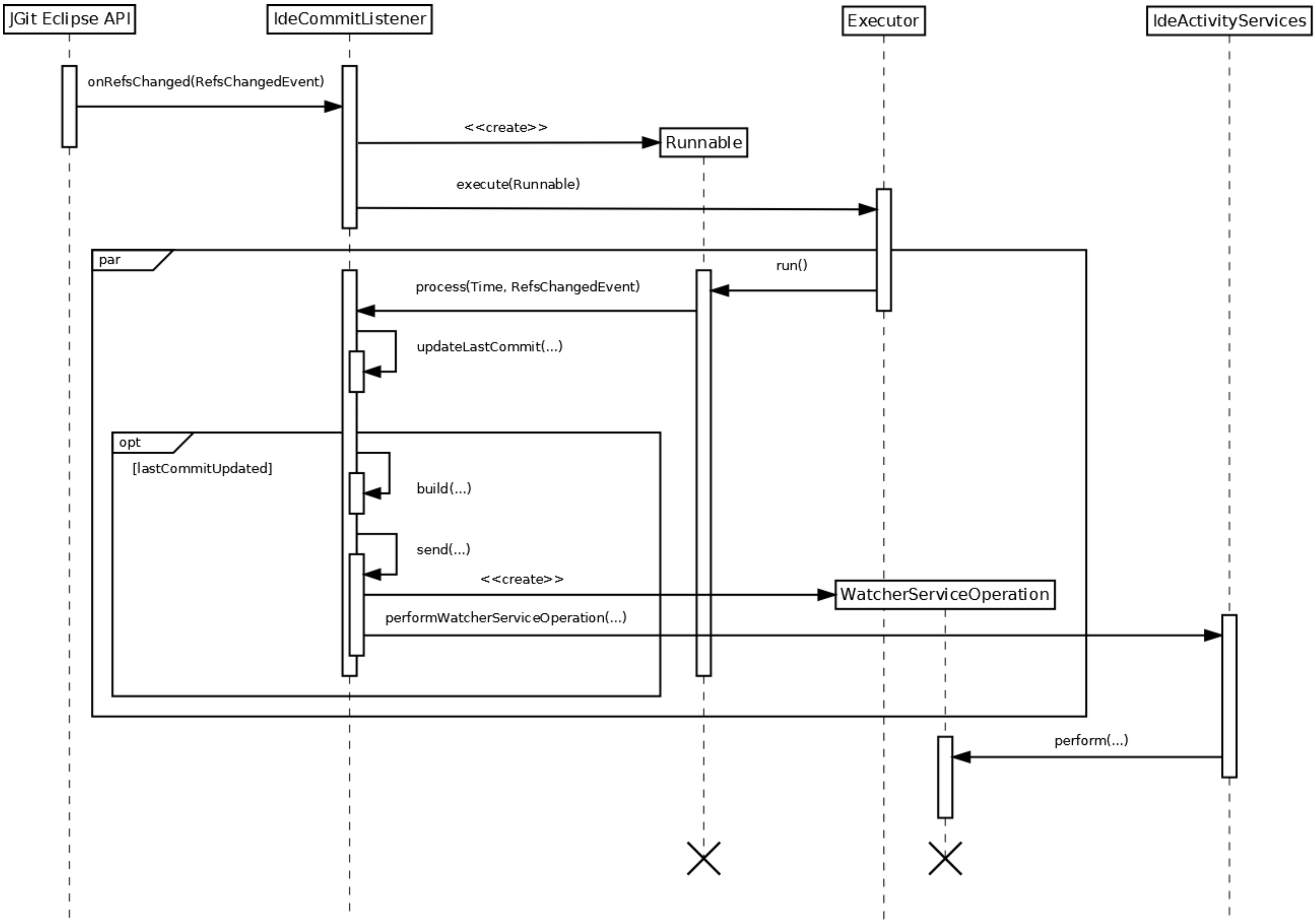


2.2.5 Commit Event Processing

Sequence diagram for Commit Event Processing in figure 2.8 depicts a real example of a registered listener in action. Commit listener hooks for every event that is produced by changed or newly created reference on a Git[2] repository. The listener filters and processes events, builds event data transfer objects and finally sends these data objects wrapped as a Watcher Service Operation to be persisted by the Watcher Service.

[2] Git: http://git-scm.org

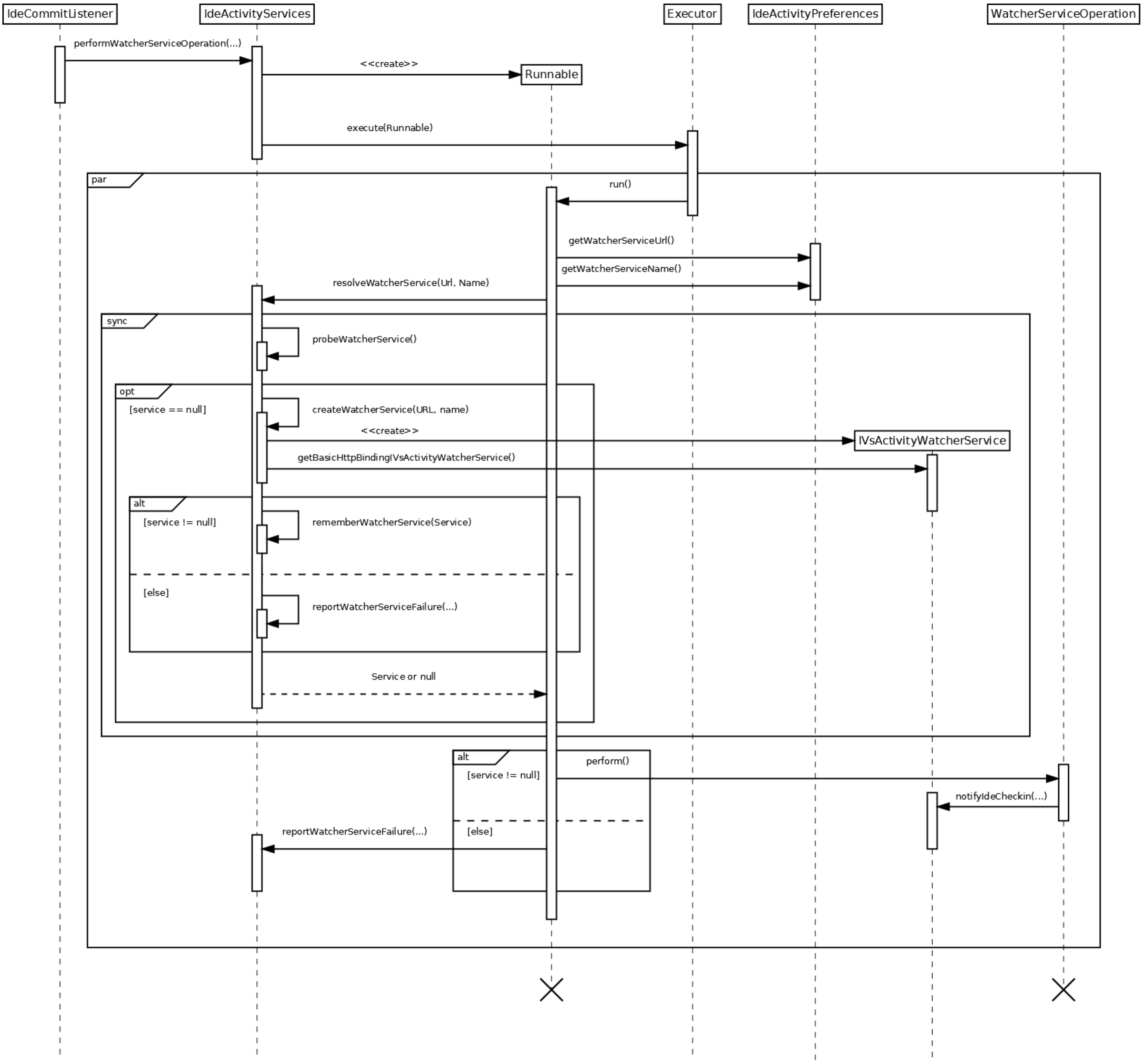
Figure 2.8: *Sequence Diagram – Commit Event Processing*



2.2.6 Watcher Service Operation Execution

Sequence diagram for Watcher Service Operation Execution in figure 2.9 shows how is the Watcher Service instance resolved (and remembered for subsequent use) and used to perform the operation to send event data to UACA in another thread provided by an executor.

Figure 2.9: *Sequence Diagram – Watcher Service Operation Execution*



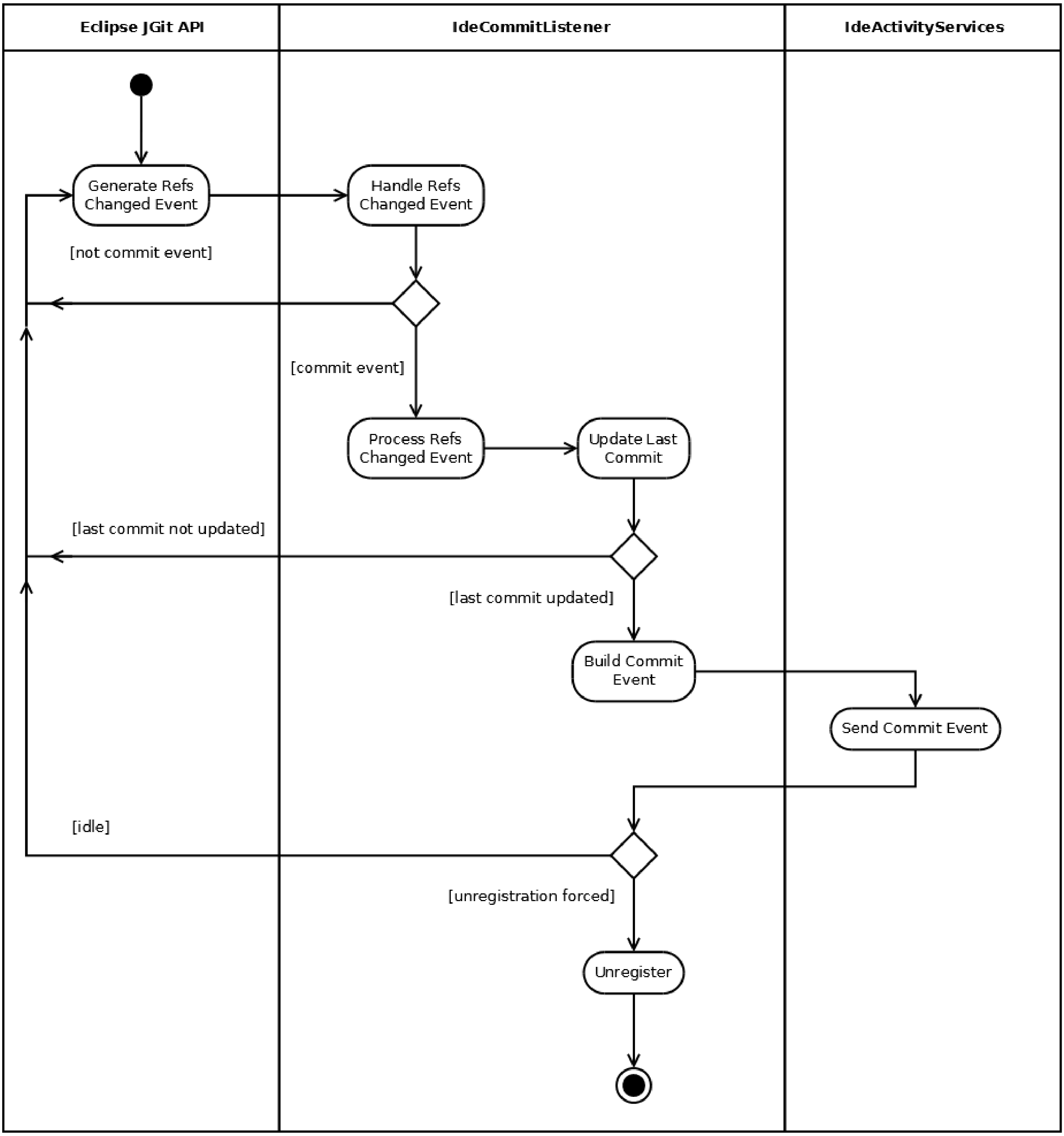
2.3 Activity Diagrams

2.3.1 Git Committing

Activity diagram for Git Commiting in figure 2.10 shows how events concerning reference changes on a Git repository are handled by Commit Listener. API for JGit[3] in Eclipse generates an event and Commit Listener handles the event further by processing reference change event, building and sending the event data. The listener ca also be unregistered if requested to.

[3] JGit: http://eclipse.org/jgit

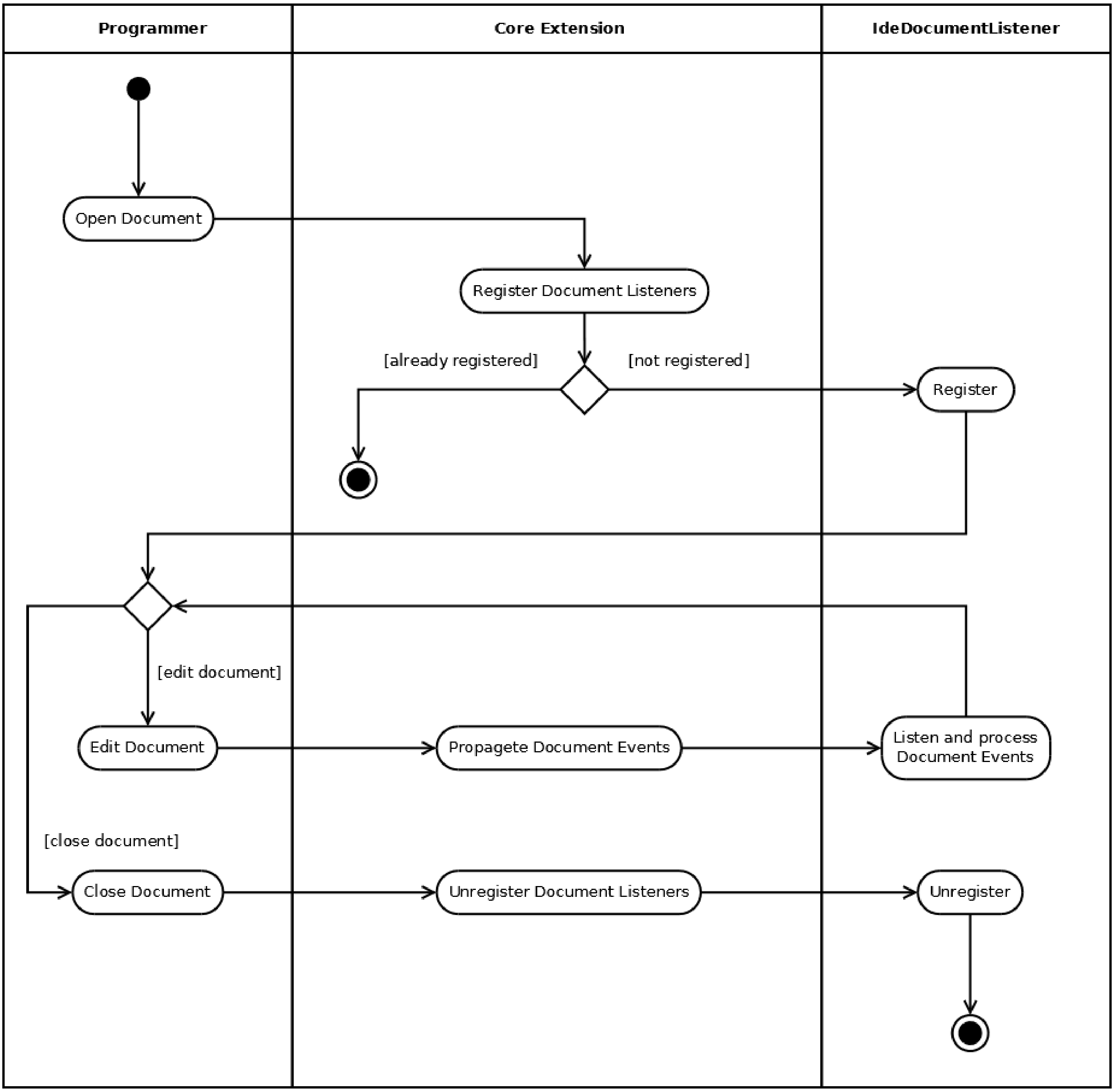
Figure 2.10: *Activity Diagram – Git Committing*



2.3.2 Document Editing

Activity diagram for Document Editing in figure 2.11 represents a flow of events processing that are fired when the programmer interacts with a source code document. When programmer opens a document, the Core Extension registers respectful document listeners. These document listeners then listen for every change on the document and when the programmer closes the document, the Core Extension unregisters all these listeners.

Figure 2.11: *Activity Diagram – Document Editing*

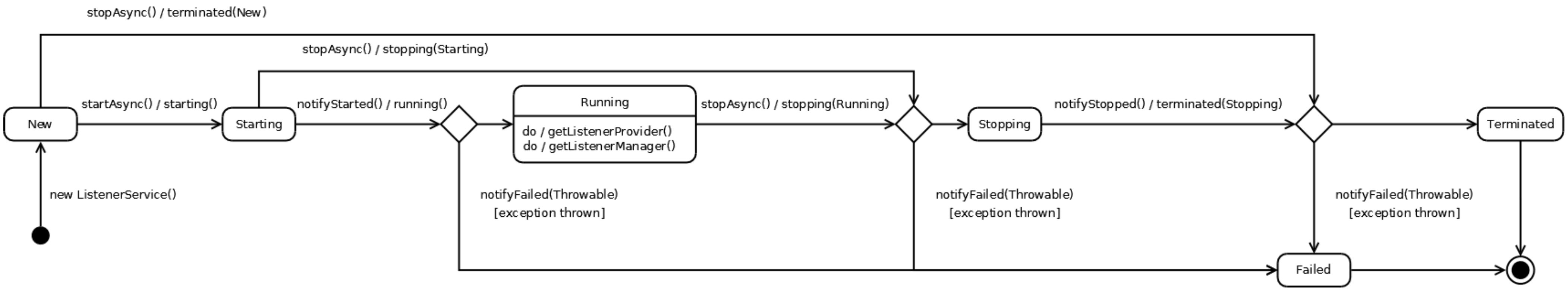


2.4 State Diagrams

2.4.1 Listener Service Lifecycle

State diagram for Listener Service Lifecycle in figure 2.12 depicts a complete life cycle of a listener service instance. It is notable that the service runs in a separate thread and therefore any access or requests to it will surely fail if not running, hence there are no operations available on states other than Running.

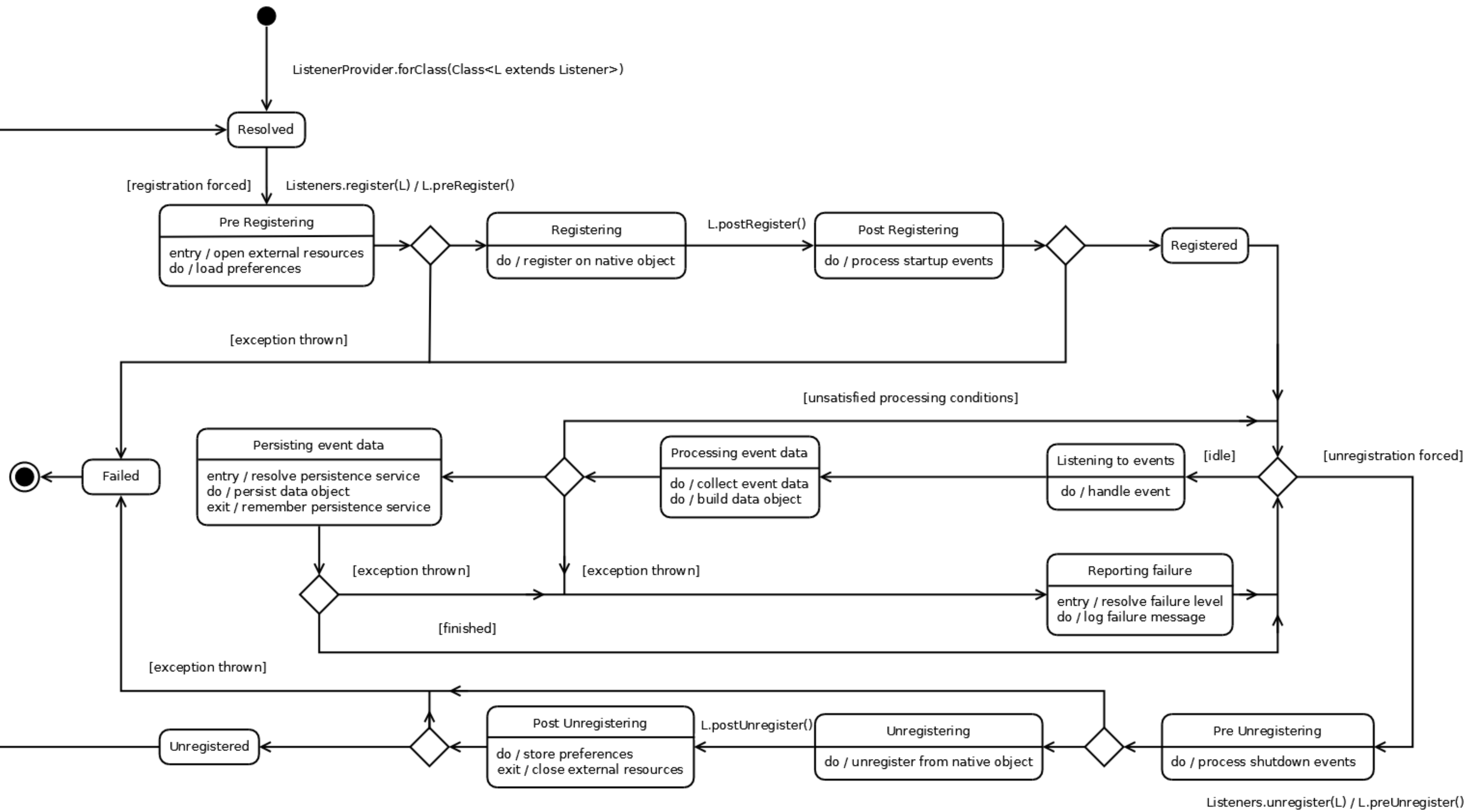
Figure 2.12: *State Diagram – Listener Service Lifecycle*



2.4.2 Listener Lifecycle

State diagram for Listener Lifecycle in figure 2.13 shows a complete life cycle of a listener instance. As soon as the listener instance is resolved it can be registered. Listener registration is a bit complicated process covered by a sequence of important states with appropriate actions. After successful registration the listener enters a state where it listens (and further processes) to events or is requested for unregistration. Listener unregistration is a very similar process as registration.

Figure 2.13: *State Diagram – Listener Lifecycle*



3 Design Patterns

In our analysis we identified and carefully selected several design patterns and visualized them in diagrams across this chapter. Most of them are recognized as classic design patterns [4], others may be Java specific, e.g. *Enum Singleton* [3], or adopted from other languages like *Optional* which is a very common feature of functional languages.

Patterns in this chapter are structured by features of the analyzed software system and it is common to depict more than one pattern on a single diagram. Therefore there is a complete catalog of recognized patterns is in section 3.1 for easier orientation in this document.

We also visualized an almost complete overview of the software system in a component digram in figure 3.2 for even easier orientation.

3.1 Pattern Catalog

List of recognized design patterns according to [4]:

Pattern	Mention
Builder	3.3, 3.13
Abstract Factory	3.4, 3.8, 3.9
Facade	3.5
Flyweight	3.7
Composite	3.7, 3.14
Memento	3.6
Proxy	3.8
Singleton	3.8, 3.15
Strategy	3.10, 3.11, 3.12

List of recognized Java specific patterns as proposed in [3] and some other well known patterns:

Pattern	Mention
Enum Singleton	3.9, 3.12, 3.14
Optional	3.15
Null Object	3.15
Serialization Proxy	3.6

3.2 Component Overview

Accompanying component diagrams depict usage connection among Core APIs in figure 3.1, and UACA APIs and Core Java DOM APIs (actually part of Core, but currently not completely utilized) in figure 3.2.

Figure 3.1: *Component Diagram – Core Overview*

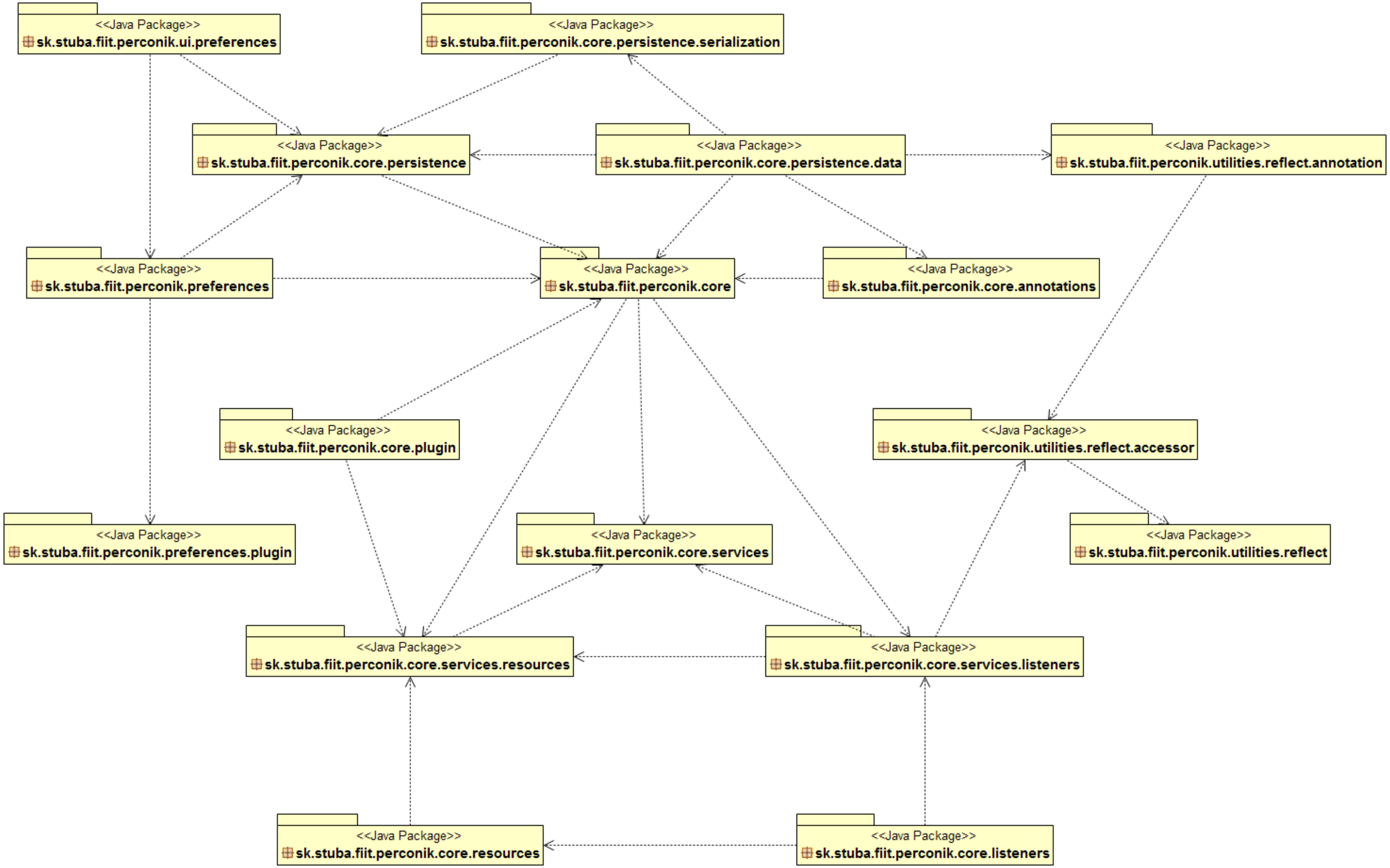
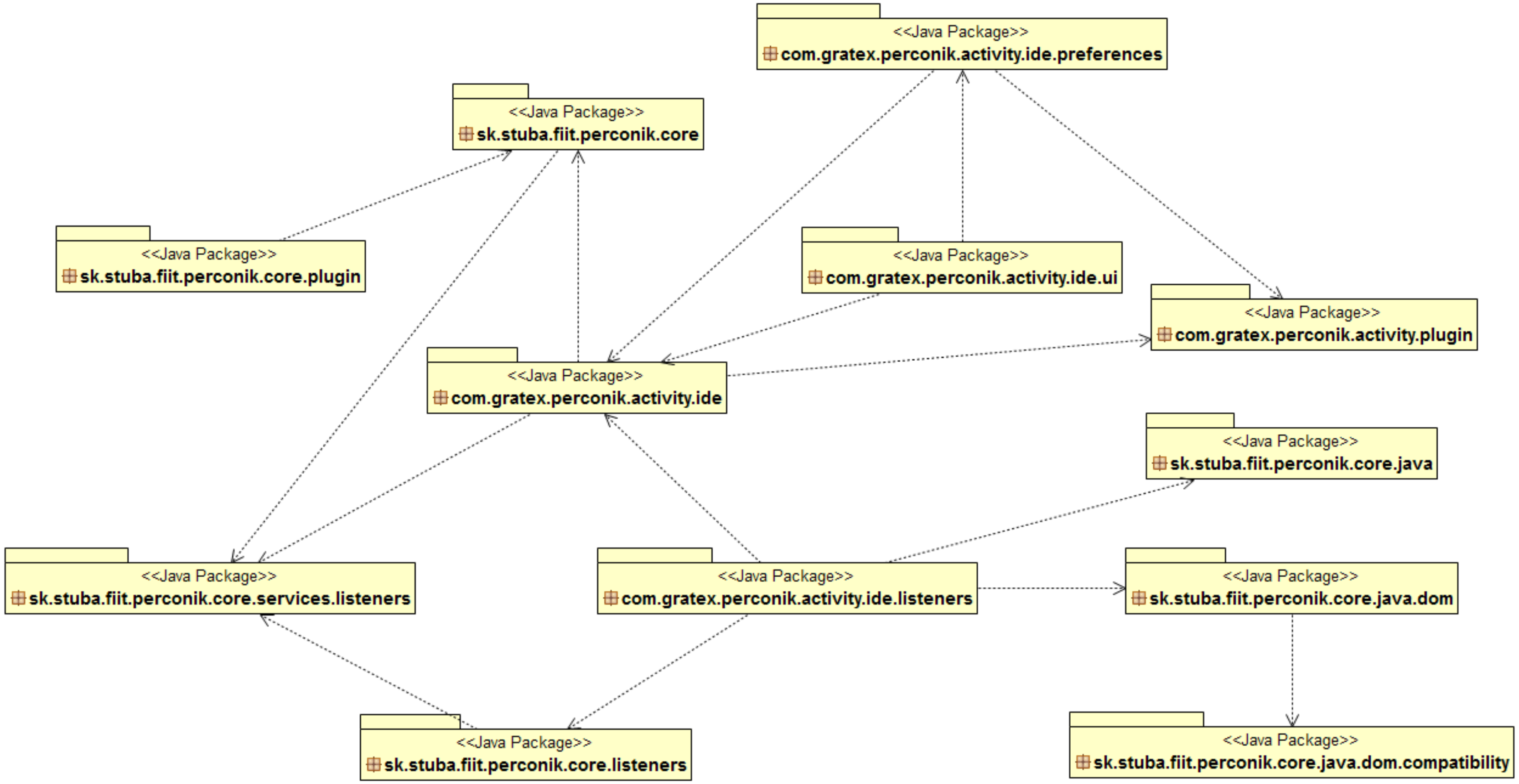


Figure 3.2: *Component Diagram – UACA and Java DOM Overview*



3.3 Core Services

Overview of supplied Core Services type hierarchy.

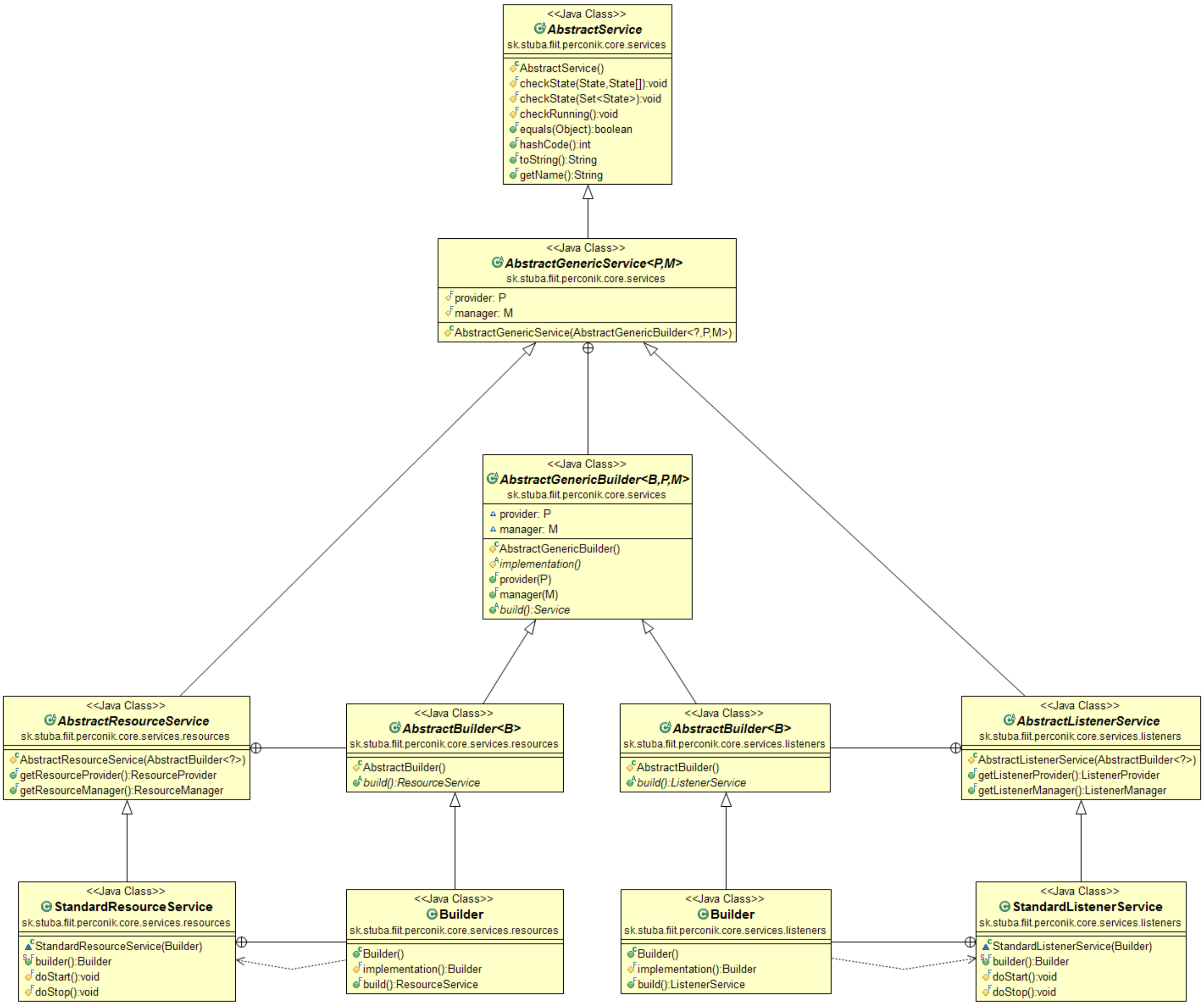
3.3.1 Builder

According to [4] the abstract builders in figure 3.3 are represented by *AbstractGenericBuilder* and its two direct descendants named *AbstractBuilder* contained in *AbstractResourceService* and *AbstractListenerService* as inner classes. Concrete builders are two implementations named *Builder*, both inner classes of *StandardResourceService* and *StandardListenerService* and thus maintaining similar type hierarchy and structure.

Abstract product is *AbstractGenericService* – the direct descendant of the *AbstractService*. Concrete products are *StandardResourceService* and *StandardListenerService*. The director is missing in this diagram, but we can safely assume that it is an instance of the *ServiceLoader* class.

Note that this builder patter design is effectively used to preserve immutability amongst its products [3].

Figure 3.3: *Class Diagram – Core Services Builders Type Hierarchy*



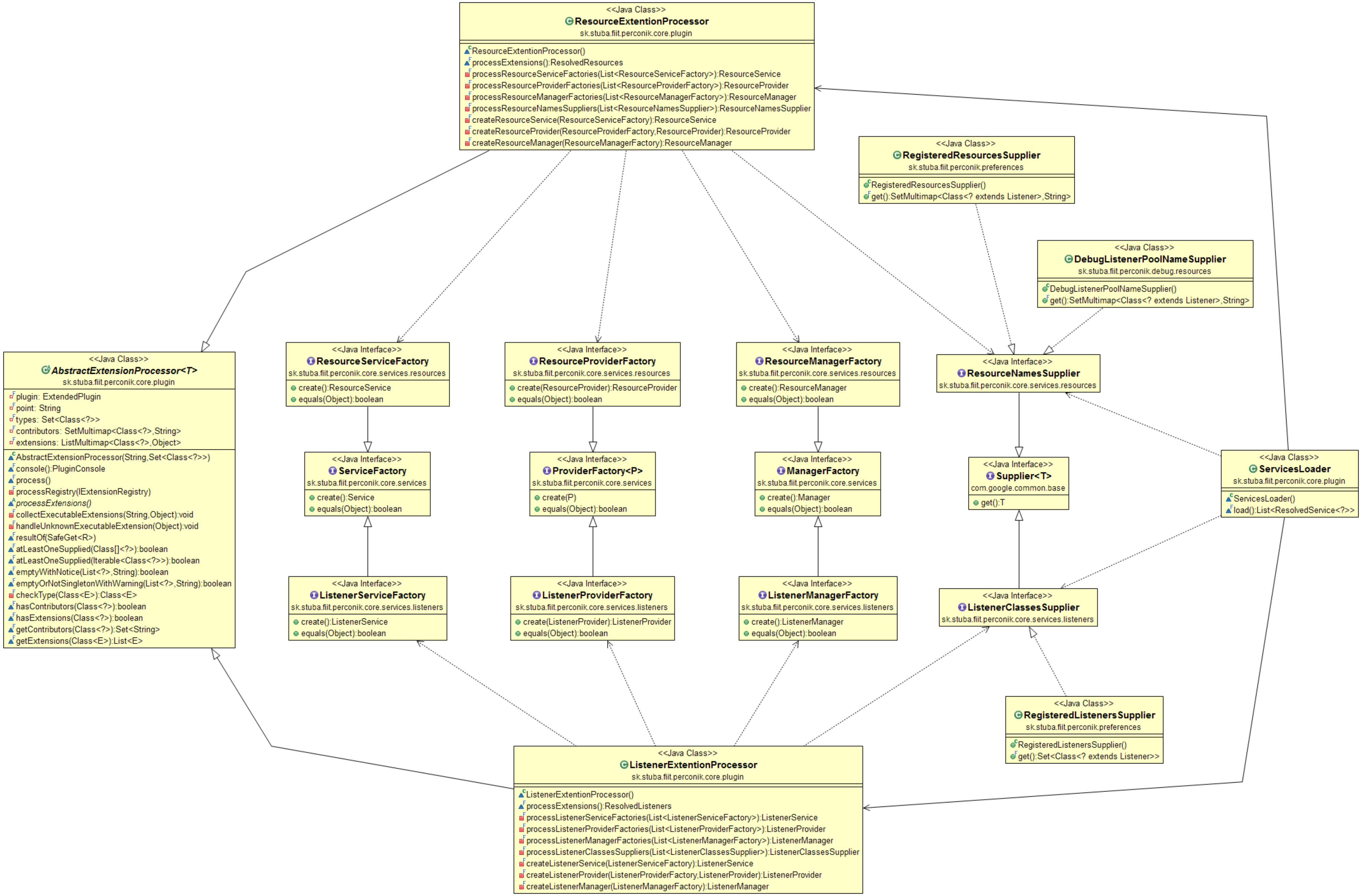
3.4 Core Factories

Overview of the extensive Core architecture – pluggable factories.

3.4.1 Abstract Factory

We provided a complex type hierarchy of abstract factories – interfaces as in the diagram center in figure 3.4. As specified in [4] we further introduced concrete factory implementations such as *RegisteredResourcesSupplier*. Abstract products and concrete products are not shown in the diagram but examples include *ListenerProvider*, *ListenerManager*, *ListenerService* and more. The client is naturally represented by the *ServiceLoader* class which utilizes concrete implementations of *AbstractExtensionProcessor* as the real clients.

Figure 3.4: *Class Diagram – Core Factories*



3.5 Core Facades

Main Core API overview.

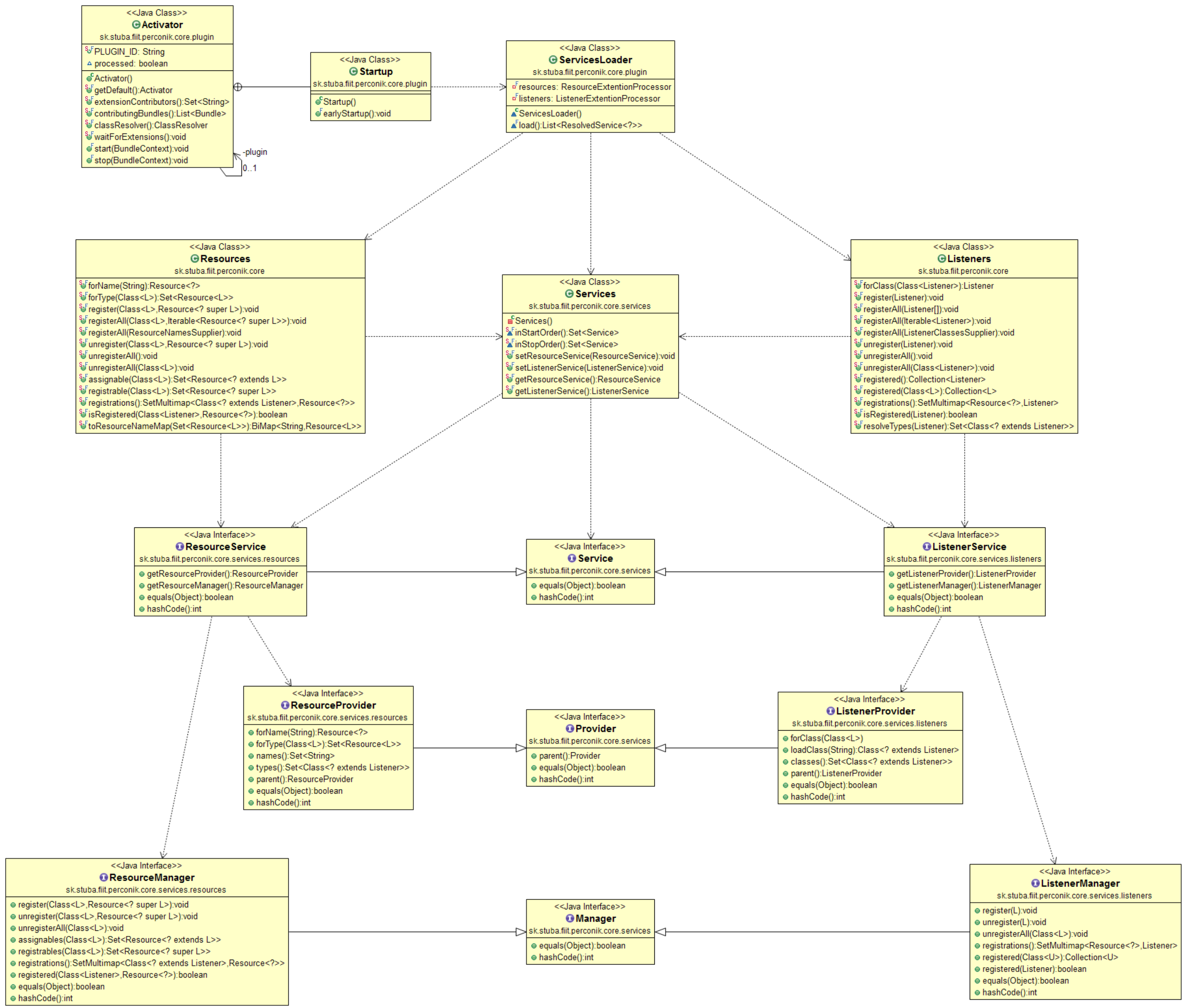
3.5.1 Facade

According to [4], *Resources*, *Services* and *Listeners* are facades, everything in the diagram in figure 3.5 above them are client classes and everything below them are subsystem interfaces (actual implementations are omitted). See example 3.1 showing sample parts the *Listeners* class.

Example 3.1: *Listeners facade as an access to core services*

```java
public final class Listeners {
  private Listeners() {}

  static ListenerService service() {
    return Services.getListenerService();
  }

  static ListenerProvider provider() {
    return service().getListenerProvider();
  }

  static ListenerManager manager() {
    return service().getListenerManager();
  }

  public static Listener forClass(final Class<? extends Listener> type) {
    return provider().forClass(type);
  }

  public static void register(final Listener listener) {
    manager().register(listener);
  }

  public static void registerAll(final Listener ... listeners) {
    registerAll(Arrays.asList(listeners));
  }

  ...

  public static void unregister(final Listener listener) {
    manager().unregister(listener);
  }

  public static void unregisterAll() {
    unregisterAll(Listener.class);
  }

  public static void unregisterAll(final Class<? extends Listener> type) {
    manager().unregisterAll(type);
  }

  public static Collection<Listener> registered() {
    return registered(Listener.class);
  }

  public static <L extends Listener> Collection<L> registered(final Class<L> type) {
    return manager().registered(type);
  }

  public static SetMultimap<Resource<?>, Listener> registrations() {
    return manager().registrations();
  }

  ...
}
```

Figure 3.5: *Class Diagram – Core Facades*



3.6 Core Persistence

3.6.1 Memento

As described in [4] a memento in figure 3.6 is *ListenerPersistenceData*. Originator is *RegisteredListenersSupplier* and caretaker is an impelmentation of *IPreferenceStore* (an interface from Eclipse Preference API), both originator and caretaker are not shown in class the diagram. There is a paralel hierarchy for resources (with some interface and implementation differences).

3.6.2 Serialization Proxy

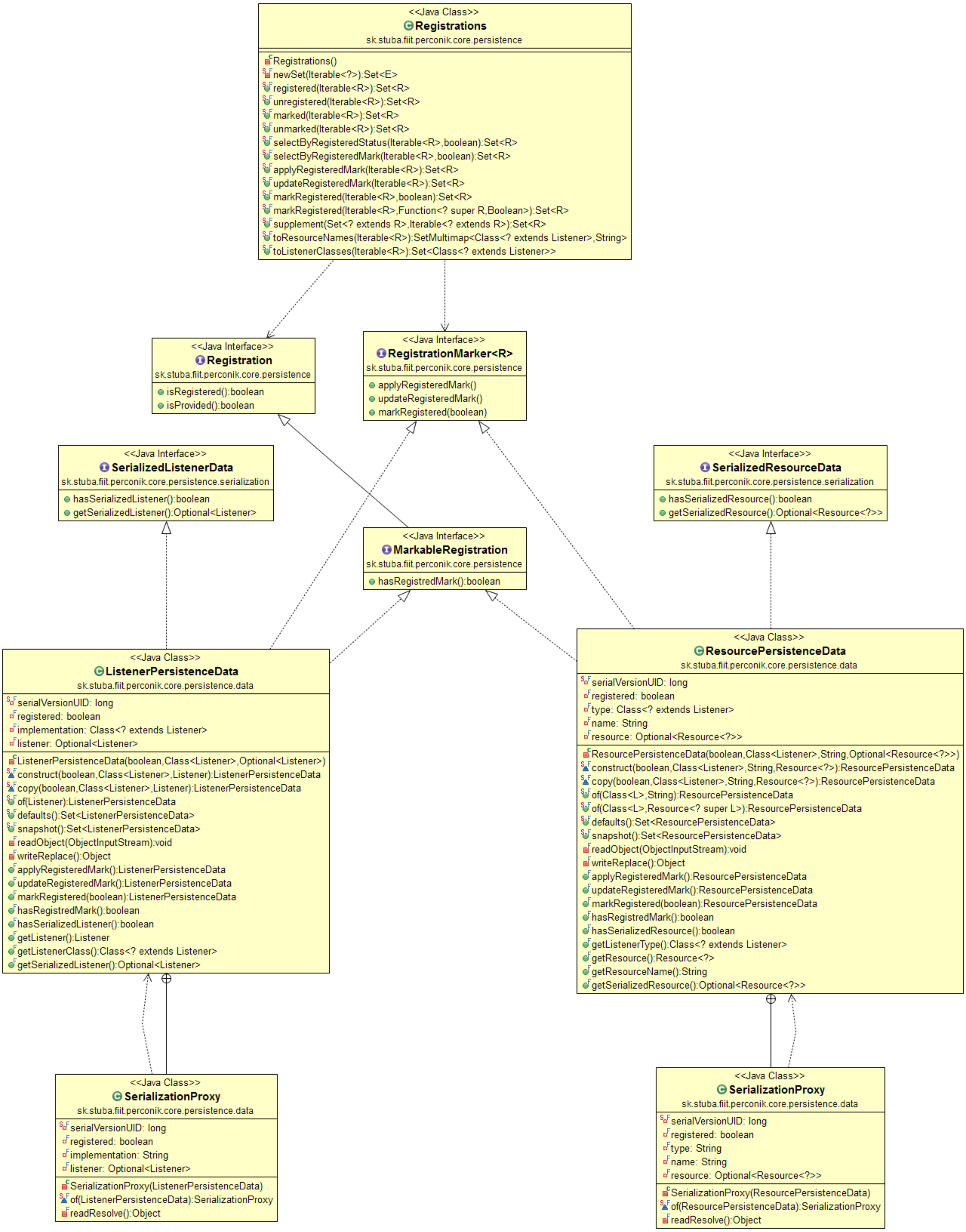
Both *SerializationProxy* classes in figure 3.6 are serialization proxies as popularized by [3]. Serialization proxy helps to maintain real immutability, forced by *final* modifier on fields of a real subject, in terms of deserialization (deserialization mechanism constructs objects in Java by avoiding constructors and hence the class can not perform precondition checks on deserialized values at instantiation). See an implementation of a *SerializationProxy* in example 3.2.

According to [4] a serialization proxy is a protection proxy as it helps protecting real subjects from malicious deserialization attacks. Real subject is *ListenerPersistenceData* and subject is SerializedListenerData, similarly with resources.

Example 3.2: *SerializationProxy as a protection for ListenerPersistenceData*

```
private static final class SerializationProxy implements Serializable {
  private static final long serialVersionUID = -6638506142325802066L;

  private final boolean registered;

  private final String implementation;

  private final Optional<Listener> listener;

  private SerializationProxy(final ListenerPersistenceData data) {
    this.registered = data.hasRegistredMark();
    this.implementation = data.getListenerClass().getName();
    this.listener  = data.getSerializedListener();
  }

  static SerializationProxy of(final ListenerPersistenceData data) {
    return new SerializationProxy(data);
  }

  private Object readResolve() throws InvalidObjectException {
    try {
      return construct(this.registered, Utilities.resolveClassAsSubclass(this.implementation, Listener.class),
    } catch (Exception e) {
      throw new InvalidListenerException("Unknown deserialization error", e);
    }
  }
}
```

Figure 3.6: *Class Diagram – Code Persistence*



3.7 Core Listener Provider

A view of standard listener provider and UACA listener implementations.

3.7.1 Composite

In figure 3.7 *ListenerProvider* is a component, *StandardListenerProvider* is composite, *SystemListenerProvider* is a leaf (not shown in diagram), and *ListenerService* is the client as defined in [4]. Each composite can hold up to one component accessible via the *parent()* method.

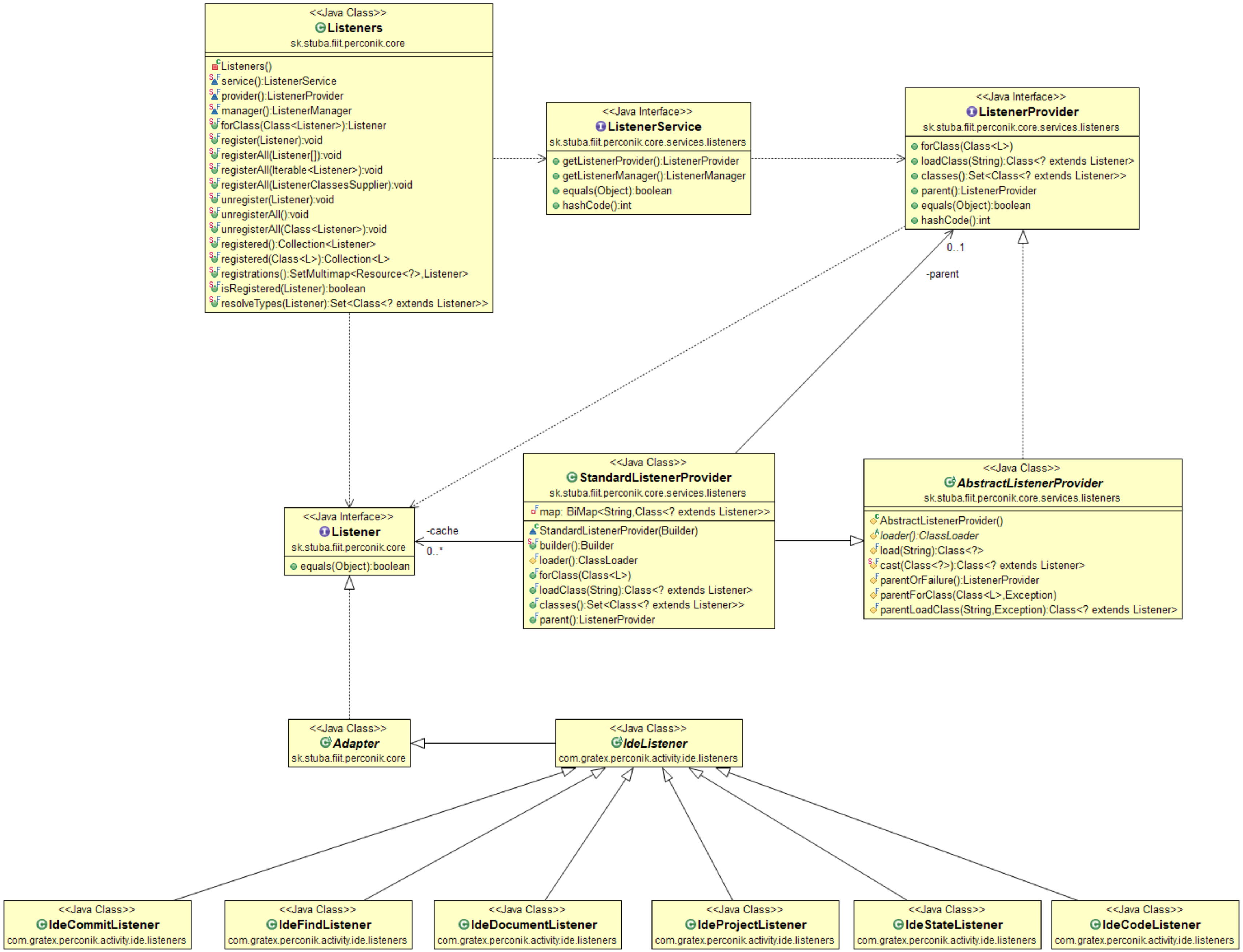
3.7.2 Flyweight

According to [4] as seen in figure 3.7, *Listener* is a flyweight, *ListenerService* is a client, *StandardListenerProvider* is a flyweight factory, and subclasses of *IdeListener* (depicted at the bottom of the diagram) are concrete flyweights. See example 3.3 for an implementation of flyweight factory.

Example 3.3: *StandardListenerProvider as an implementation of flyweight factory*

```
final class StandardListenerProvider extends AbstractListenerProvider {
  ...

  StandardListenerProvider(final Builder builder) {
    this.map = HashBiMap.create(builder.map);
    this.cache = Maps.newConcurrentMap();
    this.parent = builder.parent.or(ListenerProviders.getSystemProvider());
  }

  ...

  public <L extends Listener> L forClass(final Class<L> type) {
    Listener listener = this.cache.get(cast(type));

    if (listener != null) {
      return type.cast(listener);
    }

    L instance;

    try {
      instance = StaticListenerLookup.forClass(type).get();
    } catch (ReflectionException e) {
      Throwable[] suppressions = e.getSuppressed();

      Exception cause;

      if (suppressions.length == 1 && suppressions[0] instanceof AccessorConstructionException) {
        cause = new IllegalListenerClassException(suppressions[0]);
      } else {
        cause = new ListenerInstantiationException(e);
      }

      return this.parentForClass(type, cause);
    }

    if (!this.map.containsValue(type)) {
      this.map.put(type.getName(), type);
    }

    this.cache.put(type, instance);

    return instance;
  }

  ...
}
```

Figure 3.7: *Class Diagram – Core Listener Provider*



3.8 Core Utilities

Several Core helpers.

3.8.1 Abstract Factory

In figure 3.8 *PluginConsoleFactory* and *DebugConsoleFactory* are abstract factories, *DebugConsole.Factory* is a concrete factory, *PluginConsole* is an abstract product and *DebugConsole* is a concrete product as specified in [4].

3.8.2 Enum Singleton

In figure 3.8 *DebugConsole.Factory* is an enum singleton as popularized by [3].

Note that enums in Java are actually objects, i.e. *State.RUNNING* is a real object. Also note that enums, such as *State*, can have abstract methods and their instances, e.g. *RUNNING*, are the respective implementations.

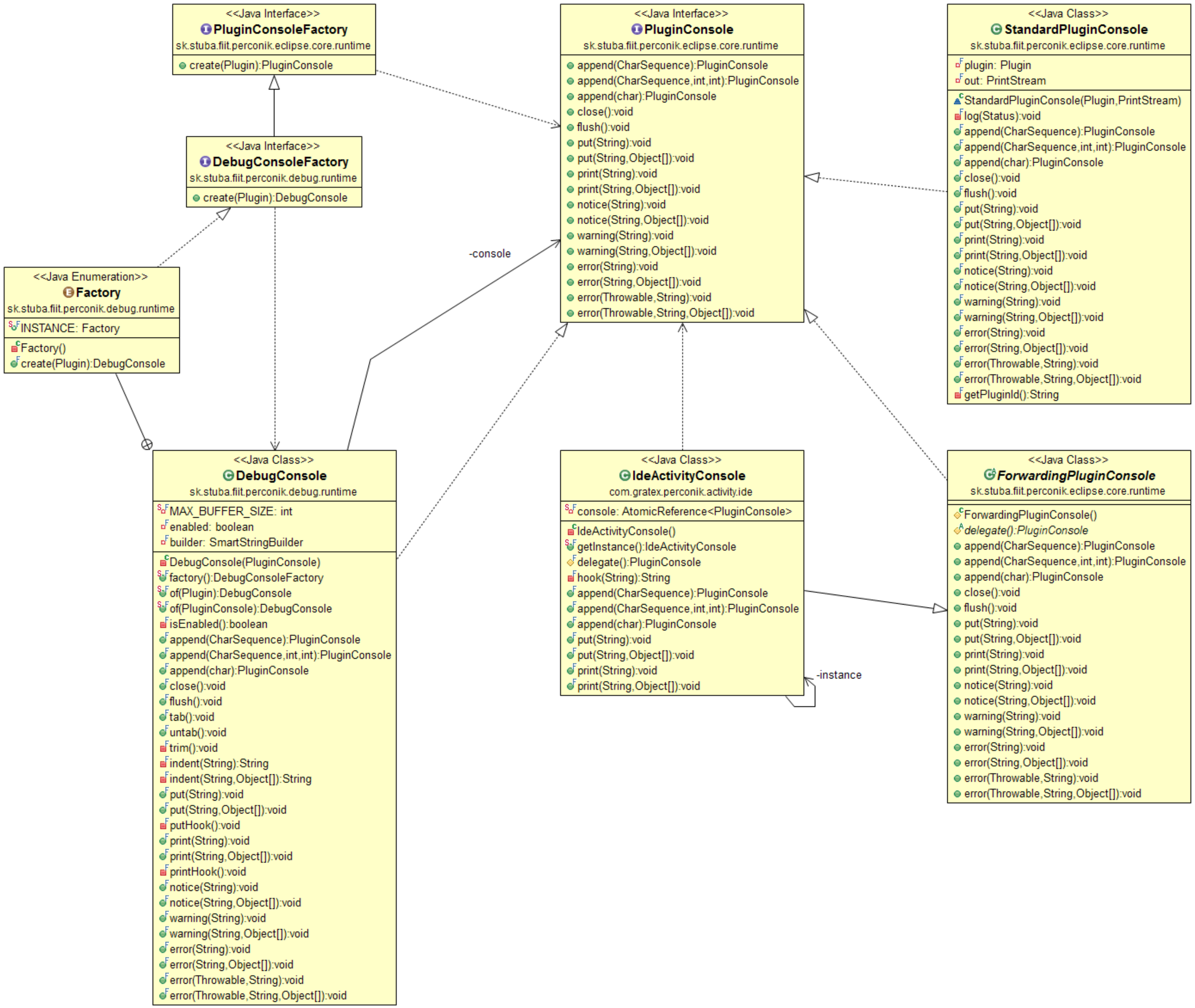
3.8.3 Proxy

In figure 3.8 *IdeActivityConsole* is a virtual proxy as described in [4], *PluginConsole* is subject and *StandardPluginConsole* is most likely the real subject.

3.8.4 Singleton

In figure 3.8 *IdeActityConsole* is a singleton held by a static class field and accessible via the *getInstance()* method as described in [4].

Figure 3.8: *Class Diagram – Core Utilities*



3.9 Java DOM Compatibility

Java DOM Compatibility API overview.

3.9.1 Abstract Factory

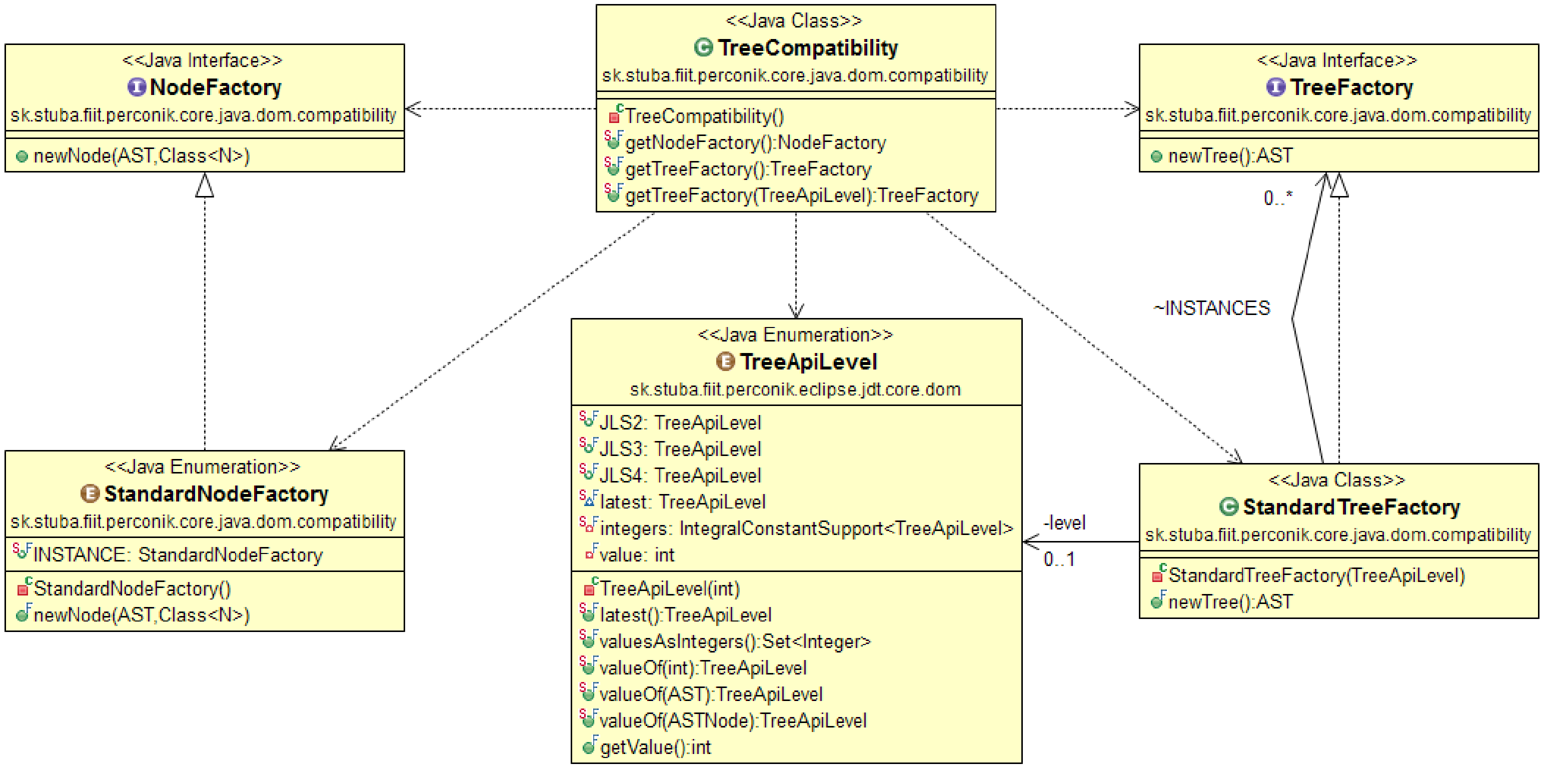
In figure 3.9 *NodeFactory* and *TreeFactory* are abstract factories, *StandardNodeFactory* and *StandardTreeFactory* are concrete factories, *ASTNode* and *AST* are products (both part of Eclipse JDT API, not shown on the diagram) as specified in [4].

3.9.2 Enum Singleton

In figure 3.9 *StandardNodeFactory* is an enum singleton as popularized by [3].

Note that enums in Java are actually objects, see section 3.8.2 for more details.

Figure 3.9: *Class Diagram – Java DOM Compatibility*



3.10 Java DOM Node Paths

Java abstract syntax tree node paths API illustration.

3.10.1 Strategy

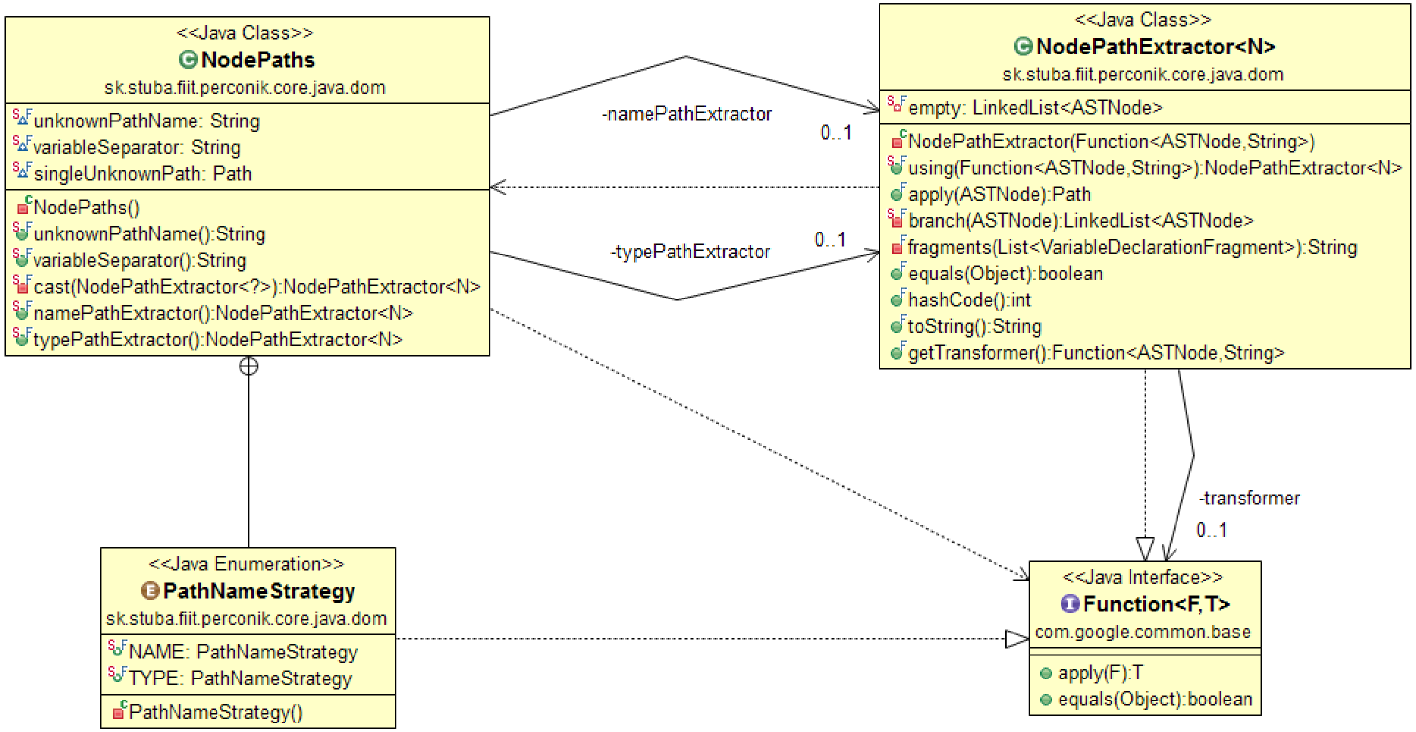
In figure 3.10 *Function* is a strategy, *PathNameStrategy.NAME* and *PathNameStrategy.TYPE* are concrete strategies, and *NodePathExtractor* is the context, as defined in [4]. See example 3.4 showing the implementation of mentioned DOM node path naming starategies.

Note that *PathNameStrategy.NAME* and *PathNameStrategy.TYPE* are objects, i.e. singleton-like implementations of the *Function* interface, see 3.8.2 for more details on Java enums.

Example 3.4: *PathNameStrategy as a namespace for node path naming strategies*

```java
private enum PathNameStrategy implements Function<ASTNode, String> {
  NAME {
    public String apply(final ASTNode node) {
      if (node == null) {
        return unknownPathName;
      }

      for (StructuralPropertyDescriptor descriptor: Nodes.structuralProperties(node)) {
        if (descriptor.getId().equals("name")) {
          return node.getStructuralProperty(descriptor).toString();
        }
      }

      return unknownPathName;
    }

    @Override
    public String toString() {
      return "name";
    }
  },

  TYPE {
    public String apply(final ASTNode node) {
      return node != null ? NodeType.valueOf(node).getName() : unknownPathName;
    }

    @Override
    public String toString() {
      return "type";
    }
  };
}
```

Figure 3.10: *Class Diagram – Java DOM Node Paths*



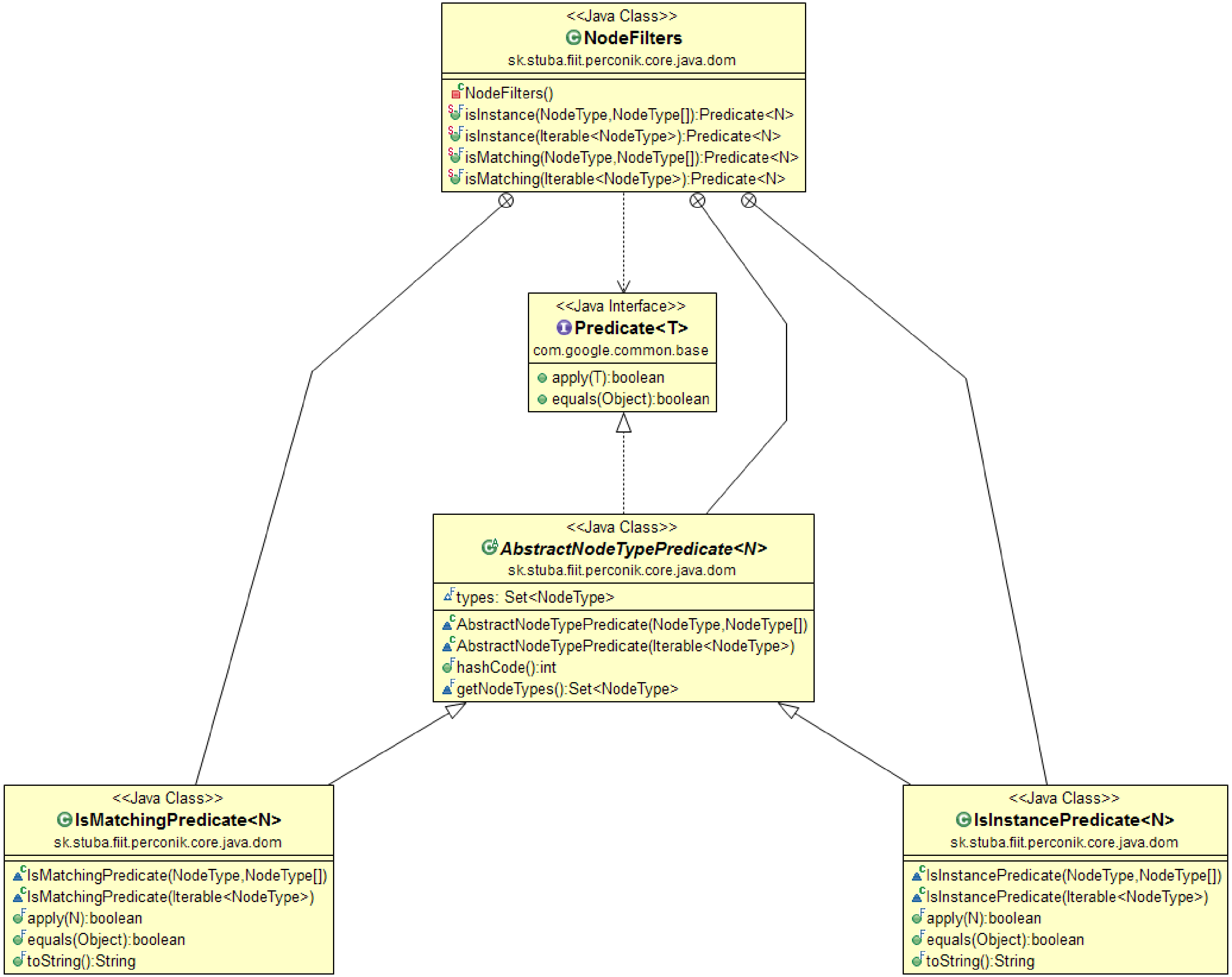
3.11 Java DOM Node Filters

Java abstract syntax tree node filters API illustration.

3.11.1 Strategy

In figure 3.11 *Predicate* is a strategy, and *IsMatchingPredicate* and *IsInstancePredicate* are concrete strategies and also themselves context objects, as defined in [4].

Figure 3.11: *Class Diagram – Java DOM Node Filters*



3.12 Java DOM Node Transformations

Java abstract syntax tree node transformations API illustration.

3.12.1 Enum Singleton

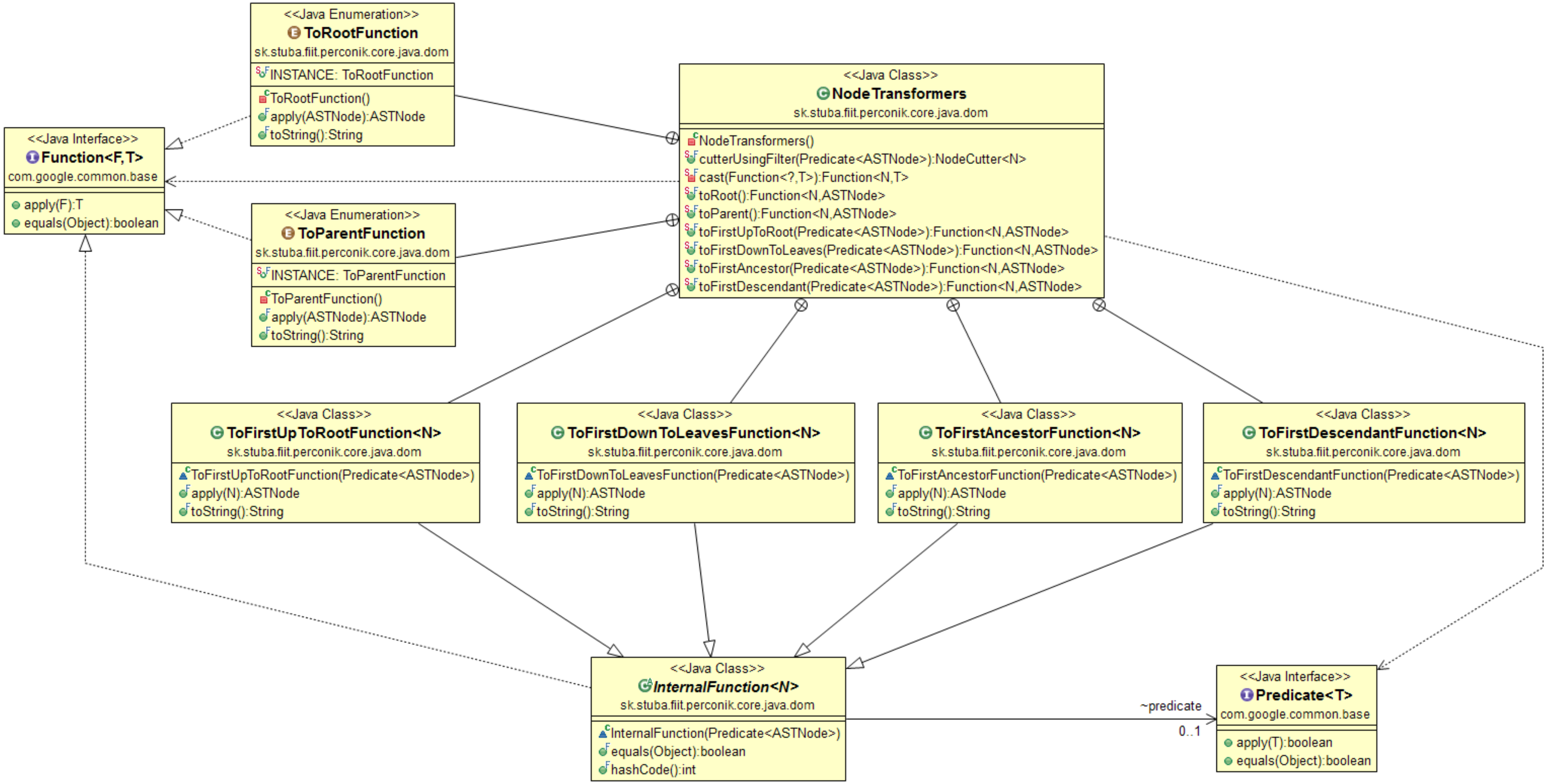
Both *ToRootFunction* and *ToParentFunction* in figure 3.12 are enum singletons as popularized by [3].

Note that enums in Java are actually objects, see section 3.8.2 for more details.

3.12.2 Strategy

Function and *Predicate* are strategies, direct descendants of *InternalFunction* and singletons *ToRootFunction* and *ToParentFunction* are concrete strategies, and *ASTNode* is the context obeject (part of the Eclipse JDT API, not shown on the diagram), as defined in [4].

Figure 3.12: *Class Diagram – Java DOM Node Transformations*



3.13 Reflective Lookup

Simple reflection utility effectively used to resolve instances at runtime.

3.13.1 Builder

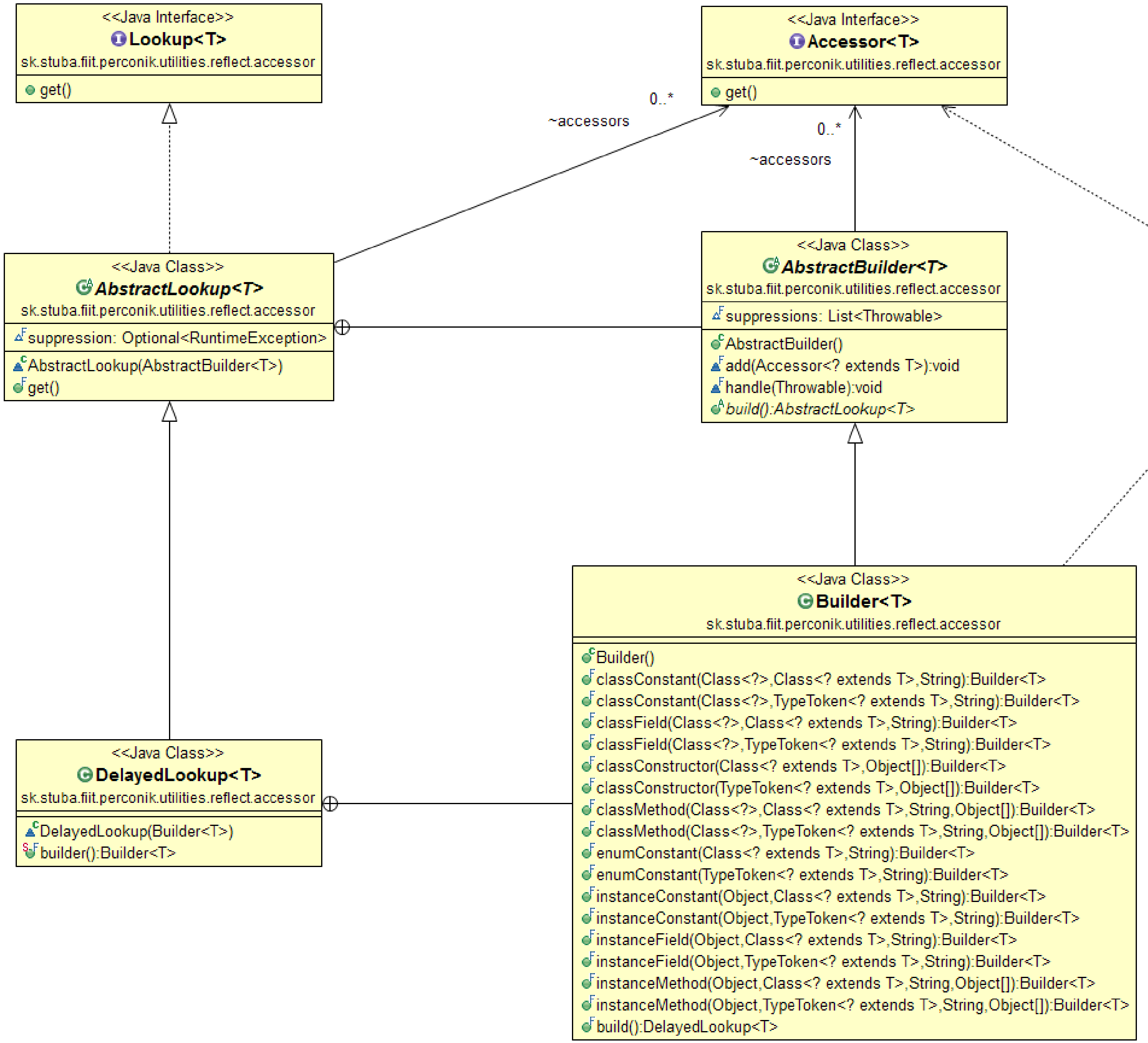
According to [4] we can immediately spot the abstract builder and concrete builder in figure 3.13. Abstract product is the *AbstractLookup* and concrete product is the *DelayedLookup*. See example 3.5 for an implementation of *AbstractBuilder*.

Note that this builder patter design is effectively used to preserve immutability amongst its products [3].

Example 3.5: *AbstractBuilder as a skeletal implementation for AbstractLookup builders*

```
static abstract class AbstractBuilder<T> {
  final List<Accessor<? extends T>> accessors;

  final List<Throwable> suppressions;

  public AbstractBuilder() {
    this.accessors = Lists.newArrayListWithExpectedSize(8);
    this.suppressions = Lists.newArrayListWithExpectedSize(8);
  }

  final void add(Accessor<? extends T> accessor) {
    this.accessors.add(Preconditions.checkNotNull(accessor));
  }

  final void handle(Throwable e) {
    Throwables.propagateIfInstanceOf(e, NullPointerException.class);

    this.suppressions.add(e);
  }

  public abstract AbstractLookup<T> build();
}
```

Figure 3.13: *Class Diagram – Reflective Lookup*



3.14 Class Resolvers

Wrapper around standard Java and Eclipse class loading mechanisms to unify their interfaces.

3.14.1 Composite

In figure 3.14 *ClassResolver* is a component, *CompositeClassResolver* is composite, *BundleClassResolver*, *DefaultClassResolver* and *LoadingClassResolver* are leafs, as defined in [4]. See example 3.6 for an implementation of composite class resolver.

Note that components can not be obtained from the composite.

Example 3.6: *CompositeClassResolver as a root of composable class resolving mechanism*

```java
final class CompositeClassResolver implements ClassResolver {
  private final List<ClassResolver> resolvers;

  CompositeClassResolver(Iterable<ClassResolver> resolvers) {
    this.resolvers = ImmutableList.copyOf(resolvers);

    Preconditions.checkArgument(!this.resolvers.isEmpty());
  }

  public Class<?> forName(String name) throws ClassNotFoundException {
    List<Throwable> suppressions = Lists.newLinkedList();

    for (ClassResolver resolver: this.resolvers) {
      try {
        return resolver.forName(name);
      } catch (Exception e) {
        suppressions.add(e);
      }
    }

    ClassNotFoundException failure = new ClassNotFoundException(name + " not found");

    throw MoreThrowables.initializeSuppressor(failure, Lists.reverse(suppressions));
  }
}
```

3.14.2 Enum Singleton

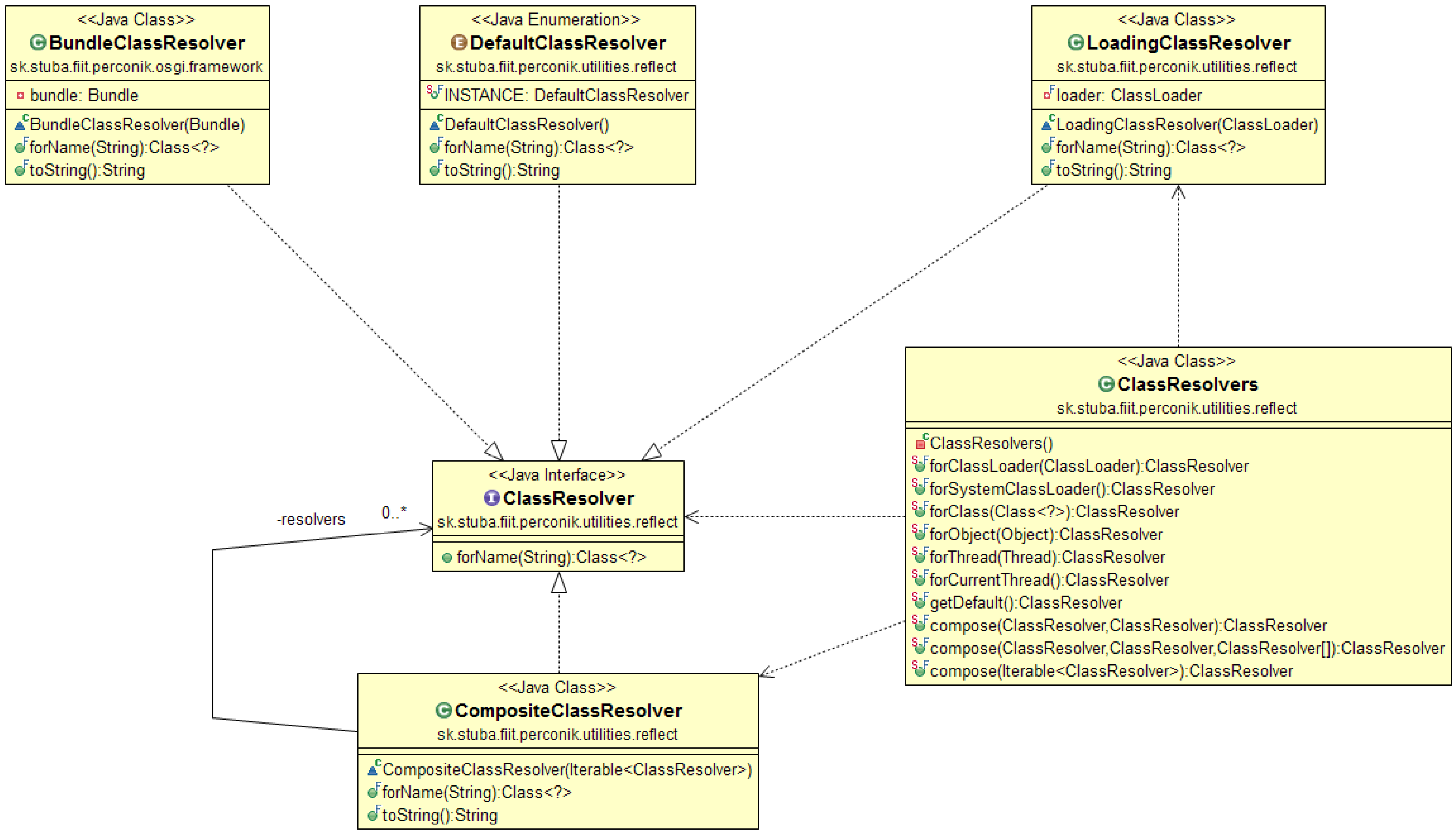
In figure 3.14 *DefaultClassResolver* is an enum singleton as popularized by [3]. See example 3.7 for an implementation of enum singleton.

Note that enums in Java are actually objects, see section 3.8.2 for more details.

Example 3.7: *DefaultClassResolver as an enum singleton*

```java
enum DefaultClassResolver implements ClassResolver {
  INSTANCE;

  public Class<?> forName(String name) throws ClassNotFoundException {
    return Class.forName(name);
  }

  @Override
  public String toString() {
    return "DefaultClassResolver";
  }
}
```

Figure 3.14: *Class Diagram – Class Resolvers*



3.15 Optionals

Very popular pattern mostly seen in functional languages and recently introduced as a part of Java 8 Standard Library. The class diagram in figure 3.15 shows an *Optional* from the Guava Library [1] and *Exceptional* which is our own approach. *Optional* holds a reference to an object on success or null on failure in comparison to *Exceptional* which holds a reference to an object on success or a reference to an exception on failure.

3.15.1 Null Object

In figure 3.15 *Optional.Absent* is a classic example of a null object pattern.

3.15.2 Optional

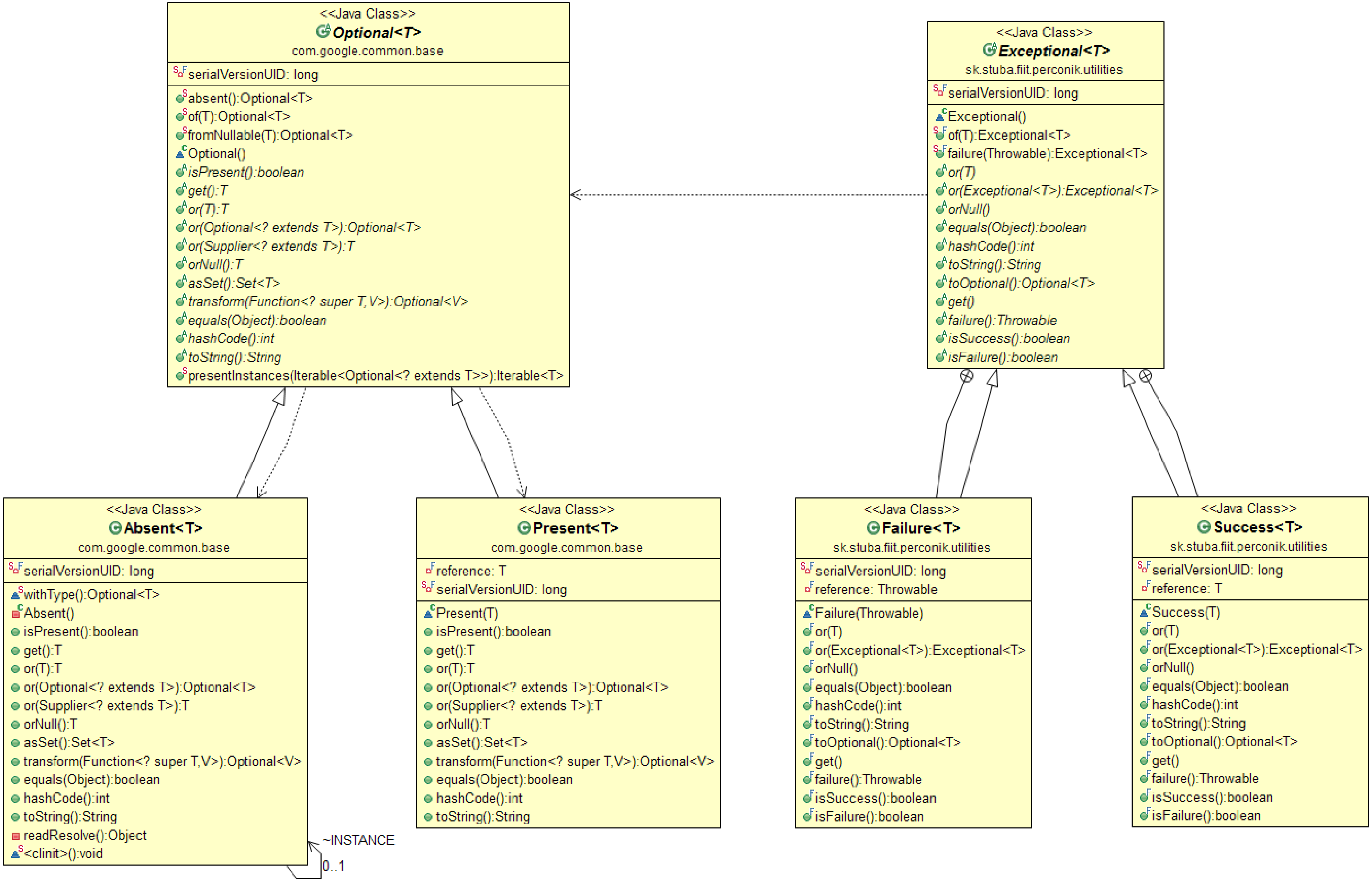
In figure 3.15 *Optional* and *Exceptionl* are design patterns themselves. Optionals are useful when there is an explicit need to substitute possible nulls with real references which is very useful when designing comprehensive APIs. Another sample usage for optionals is that they force API clients to always check whether they received or pass a reference or null.

3.15.3 Singleton

In figure 3.15 *Optional.Absent* is a singleton held by a static class field and accessible via the *whthType()* method in a package-private scope, similarly as described in [4].

[1] Guava Library: http://code.google.com/p/guava-libraries

Figure 3.15: *Class Diagram – Optionals*



4 Conclusion

In our work, we analyzed selected features of a system for acquisition and persistence of programmer's activity in Eclipse IDE. In use case modeling diagrams, we focused on three major processes in the system – startup and shutdown of Eclipse and event processing. In sequence modeling diagrams, we mainly analyze and model actions required for loading resources, their respectful listeners with focus on dynamic lookup of listener instances along with their registration on specific resources. Since listeners and services for their registration are essential part of the architecture, we provide two state diagrams modeling states of listener and listener service life cycle. Design of the system architecture is based on substantial amount of design patterns covering patterns such as Abstract Factory, Builder, Flyweight, Memento, and more.

This analysis is a part of the research project PerConIK at Faculty of Informatics and Information Technologies at Slovak University of Technology in Bratislava. Presented system is currently in production use as a component for programmer's activity acquisition included in a larger framework for complex platform independent software development monitoring [1].

Bibliography

[1] Mária Bieliková, Polášek Ivan, Michal Barla, Eduard Kuric, Rástočný Karol, Jozef Tvarožek, and Peter Lacko. Platform Independent Software Development Monitoring: Design of an Architecture. *SOFSEM 2014: Theory and Practice of Computer Science*, 8327:126–137, 2014.

[2] Mária Bieliková, Pavol Návrat, Daniela Chudá, Ivan Polášek, Michal Barla, Jozef Tvarožek, and Michal Tvarožek. Webification of Software Development: General Outline and the Case of Enterprise Application Development. *Global Journal on Technology, Vol 3 (2013): 3rd World Conference on Information Technology (WCIT-2012)*, 03:1157–1162, 2013.

[3] Joshua Bloch. *Effective Java (2nd Edition)*. The Java Series. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2008.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.