

# Fhlintstone Architectural Documentation

## Table of Contents

1. Introduction and Goals .....	2
1.1. Requirements Overview .....	2
1.2. Quality Goals .....	3
1.3. Stakeholders .....	3
2. Architecture Constraints .....	4
3. Context and Scope .....	5
3.1. Business Context .....	5
3.2. Technical Context .....	6
4. Solution Strategy .....	7
4.1. Technology Decisions .....	7
4.2. Top-level Decomposition .....	7
4.3. Quality Goals .....	7
5. Building Block View .....	8
5.1. Whitebox Overall System .....	8
5.2. Level 2 .....	10
5.3. Level 3 .....	14
6. Runtime View .....	15
6.1. Maven Integration .....	15
6.2. Command Line Execution .....	15
6.3. Core Processor Initialization .....	16
6.4. ValueSet Enum Generation .....	17
6.5. StructureDefinition Class Generation .....	18
7. Cross-cutting Concepts .....	21
7.1. Automated Tests .....	21
8. Architecture Decisions .....	22
9. Quality Requirements .....	23
10. Risks and Technical Debts .....	24
11. Glossary .....	25

---



# 1. Introduction and Goals

Fhlintstone is a Java code generator to support [HL7 FHIR](#) developers using the popular [HAPI FHIR](#) framework. Based on one or multiple [NPM Packages](#) containing the various definitions, it can be used to

- generate Java enums for [ValueSets](#)
- generate Java classes to implement [custom resource and extension classes](#) for [StructureDefinitions](#) and
- facilitate [handling profiles and extensions](#) by providing builders tailored to the specific FHIR adaptations described by the package contents.

The main objectives of Fhlintstone are

- to support developers by automating repetitive tasks as much as possible,
- to produce high-quality (i.e. easily human-readable) code to support debugging,
- to support [FHIR profiles](#) and [packages](#) out-of-the-box (i.e. without local adaptation) as much as possible and
- to integrate itself into both manual and automated development workflows easily.

## 1.1. Requirements Overview

ID	Requirement	Explanation
R1	FHIR Package Support	MUST be able to work with <a href="#">FHIR resources</a> provided as <a href="#">NPM packages</a> .
R2	FHIR Release Support	MUST support FHIR release R4. SHOULD support releases R4B, R5 and ongoing; SHOULD NOT preclude support of earlier releases.
R3	Code Generation	MUST provide developer support by generating code as specified by sub-requirements.
R3.1	ValueSet Enumerations	MUST be able to generate a Java enum that represents a <a href="#">FHIR ValueSet</a> .
R3.2	Custom Resource Implementation	SHOULD be able to generate a Java class that represents a custom <a href="#">FHIR resource</a> .
R3.3	Complex Extension Implementation	MUST be able to generate a Java class that represents a custom complex <a href="#">Extension</a> .
R3.4	Builder Generation	SHOULD be able to provide builders to generate instances of FHIR objects compliant with the package contents.
R4	Java Version Support	MUST be able to generate Java compliant with Version 21.



ID	Requirement	Explanation
R5	Integration Capabilities	MUST be able to integrate into automated build processes.
R5.1	Maven Integration	MUST provide an integration into Apache Maven processes.
R5.1	Command Line Interface	SHOULD provide a command line interface to invoke generation manually.
R6	Configurability	MUST provide support to configure output path, target namespace and other options.

## 1.2. Quality Goals

The main quality goals are

- **Functional Suitability:** must be able to provide valuable and meaningful support to Java developers
- **Compatibility:** should support as much of the FHIR standard in actual use as is reasonably possible
- **Maintainability:** should be easily extensible to cover new FHIR releases or as-of-yet unsupported FHIR structures.

Note: For product quality requirements, see [Quality Requirements](#).

## 1.3. Stakeholders

Role/Name	Expectations
Java / FHIR Developer	...



## 2. Architecture Constraints

Fhlintstone shall be

- platform-independent, i.e. executable on the major operating systems (Linux, Windows, mac OS)
- deterministic, i.e. produce the same output for each execution with the same input data (apart from generation timestamps)

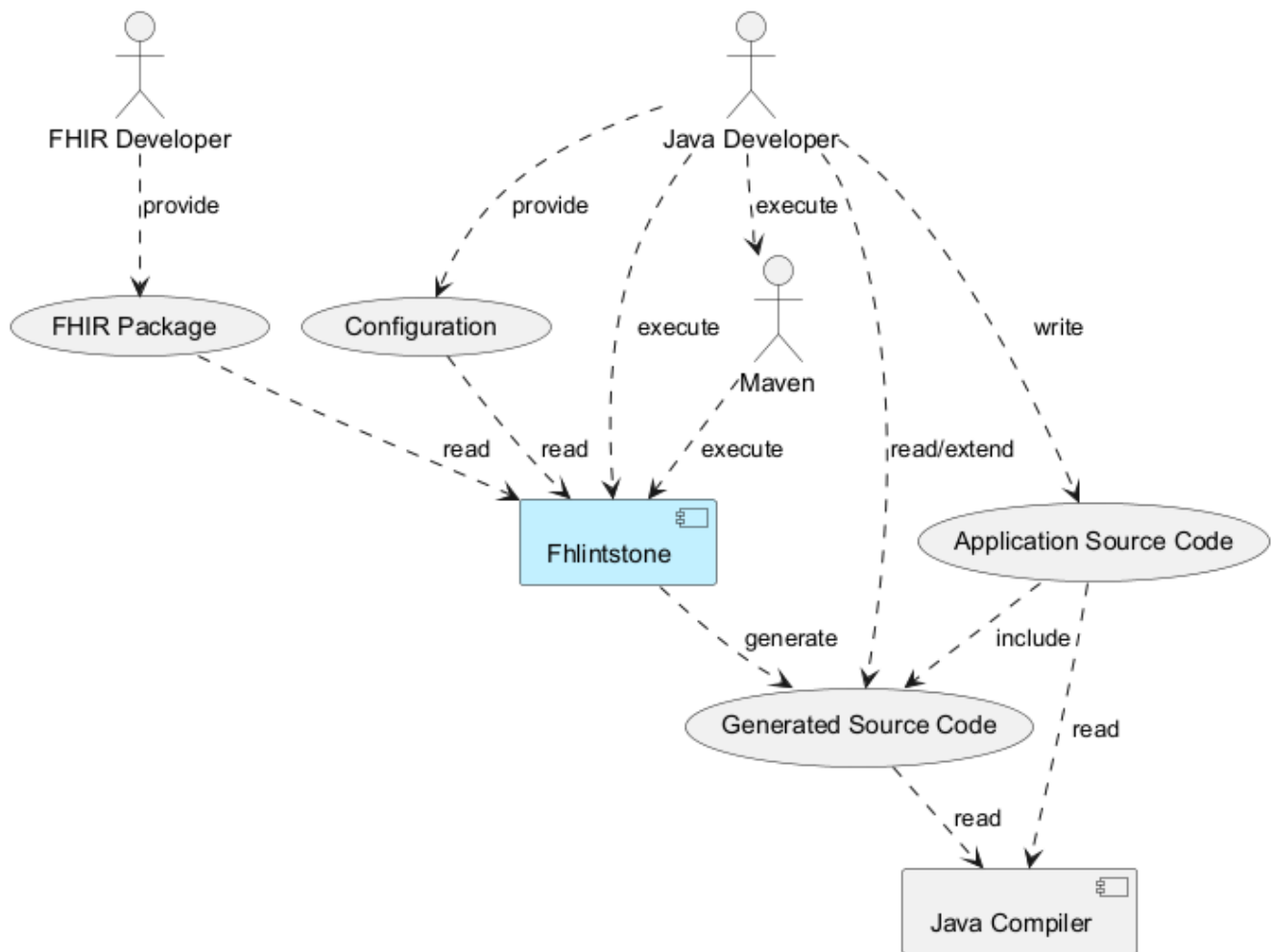
Fhlintstone has to conform to the following external constraints:

- [FHIR resources](#) are usually provided in [NPM packages](#). The Java developer frequently has very little, if any, influence on the contents of these packages. Fhlintstone has to be able to work with a wide range of valid FHIR NPM packages.
- Multiple FHIR resources may be required to produce a single output type. Fhlintstone must be able to resolve the dependencies and relationships between the various resources.
- FHIR resources may be distributed over several NPM packages which may or may not be connected with explicit dependency relationships. Fhlintstone must be able to locate resources in multiple input packages irrespective of an existing package-level dependency. Missing package-level dependencies should be recognized and logged though.



## 3. Context and Scope

### 3.1. Business Context

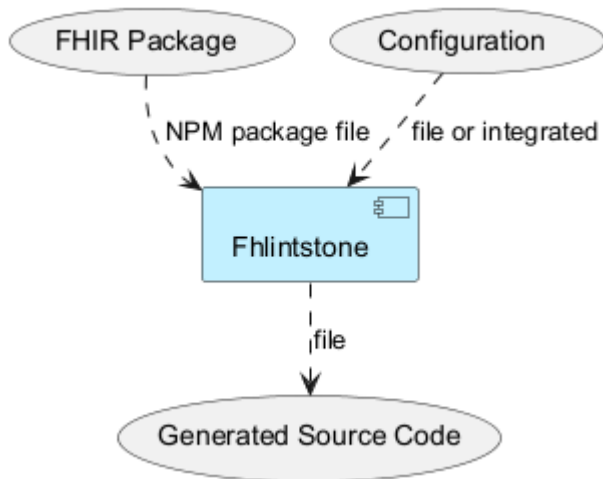


Neighbor	Description
FHIR Developer	Provides <a href="#">FHIR packages</a> . May be located outside the development organization.
<a href="#">FHIR Package</a>	Contain <a href="#">ValueSets</a> , <a href="#">StructureDefinitions</a> and other FHIR Resources that may be relevant to the developer of a FHIR application.
Java Developer	Configures Fhlintstone to read the FHIR Packages and (optionally) executes Fhlintstone to produce the Generated Source Code, either manually or via a build tool like Maven. Develops the actual Application Source Code by extending the Generated Source Code.
Configuration	Specifies which FHIR packages to use, which target structures to generate and where to place the Generated Source Code.
Fhlintstone	Reads the Configuration and the FHIR Package(s) and produces the Generated Source Code.
Generated Source Code	Output of the Fhlintstone generation process. Can be re-generated at any time.



Neighbor	Description
Application Source Code	Uses the Generated Source Code to provide the actual functionality of the target application.
Java Compiler	Reads both the Generated and the Application Source Code to produce the deliverable artifacts.

## 3.2. Technical Context



Interface	Description
FHIR Package	FHIR Packages are read from ,NPM package files (in .tgz format).
Configuration	The Configuration is either provided as a separate configuration file (for command-line execution) or as part of the build system (e.g. Maven) configuration.
Generated Source Code	The Generated Source Code is written to the file system for maximum compatibility with both the Java compiler and other build tools as well as the Development IDE.



## 4. Solution Strategy

### 4.1. Technology Decisions

**Java** code is to be generated, and the target framework for which code is to be generated is also written in Java. For this reason, Java is also used for the implementation of Fhlintstone. An LTS version current at the time of development is used.

At the start of development, **Maven** integration was planned as the primary use case. For this reason—and due to the high maturity of this build framework—Maven is used to support the software lifecycle.

Fhlintstone does not maintain its own persistent state. This ensures that a call always produces a reproducible result that depends only on the explicitly specified input data. This ensures the stability of the host build.

### 4.2. Top-level Decomposition

Since Fhlintstone is to be executed both from the command line and from a Maven build, the **core logic** for analyzing FHIR structures and code generation is separated from the command line application and the Maven integration. This also facilitates later integration into other build systems.

### 4.3. Quality Goals

In order to best support the [,quality goals](#), the following measures are planned:

**Functional suitability** is ensured by maintaining a realistic example scenario in parallel with the development of the tool. This consists of a hypothetical FHIR package and a simple but realistic example program that demonstrates the use of the code generator and the generated classes and ensures the usability of the generated components.

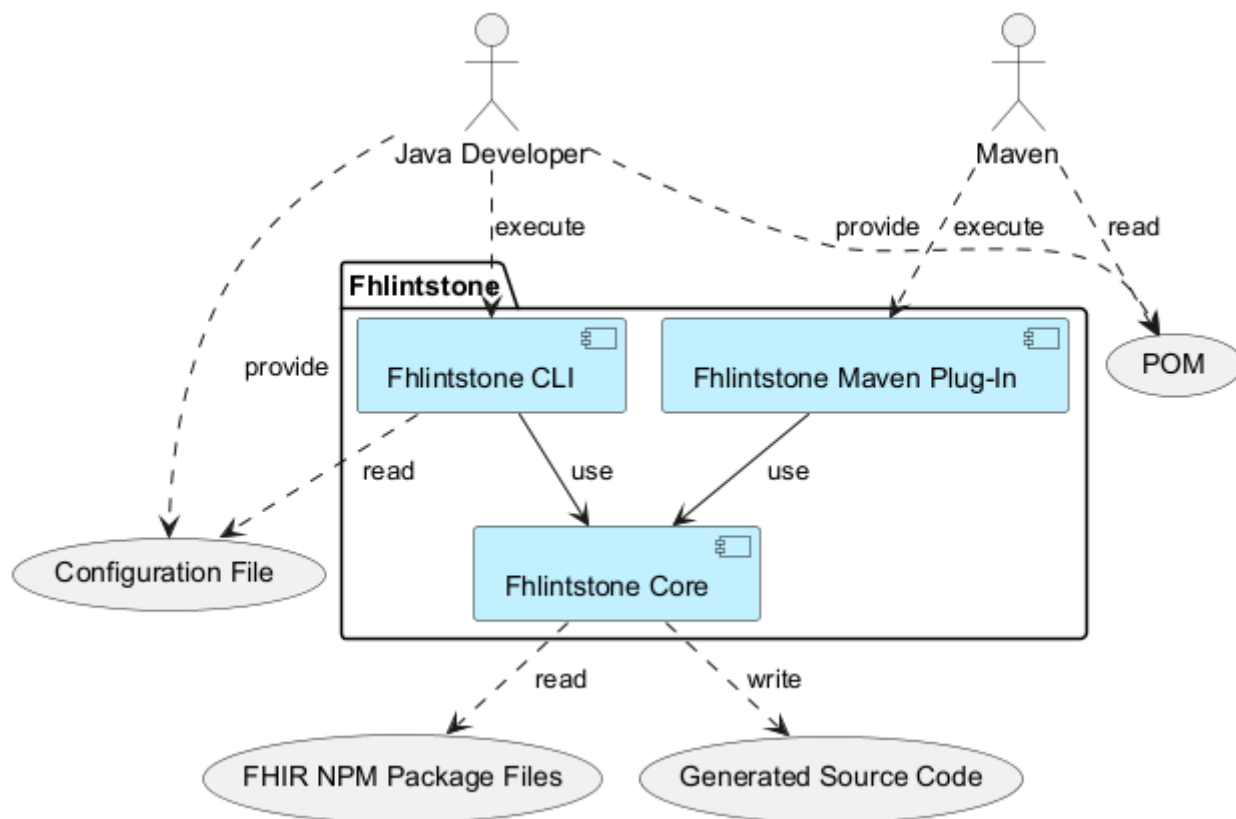
The high complexity of the FHIR standard makes it difficult to ensure comprehensive **compatibility**. For this reason, we are striving to achieve a high level of compatibility while optimizing resource requirements by continuously adapting both the automated tests and the sample project to observations from practical use.

In addition to extensive automated testing, structural measures are planned to improve **maintainability**. The release-specific aspects of the FHIR framework are separated as far as possible from the structural analysis and code generation by an abstraction layer. Ideally, this will allow a new FHIR version to be supported without changing the code generator. To facilitate support for new or previously unsupported FHIR features, a multi-stage approach is chosen for implementation: First, an internal intermediate model is generated from the FHIR structures, from which a model of the code to be generated is then created. This model in turn controls the code generator. This approach makes it possible to keep changes to support new features local as far as possible. Where this is not possible, this approach offers clear traceability of the effects through the structure of the application and better test support.



# 5. Building Block View

## 5.1. Whitebox Overall System



### Motivation

The core of Fhlintstone contains the parts of the application that reads and evaluates the contents of the FHIR package and generates the source code. To achieve the integration into various calling contexts like the command line interface (CLI) or build tools like Maven, specific adapters are provided that map the input and configuration to a context-independent configuration.

### Contained Building Blocks

Name	Description
<a href="#">Core</a>	Core logic that contains package handling, FHIR resource interpretation and code generation.
<a href="#">Command Line Interface</a>	Adapter to access core logic from the command line.
<a href="#">Maven Plug-In</a>	Adapter to integrate core logic into Maven build processes.

### Important Interfaces

Name	Description
<a href="#">FHIR NPM Package Files</a>	Access to the FHIR resources that describe the structures for which code has to be generated.



Name	Description
<a href="#">Configuration</a>	Control of the generation process.
<a href="#">Generated Source Code</a>	Output of the application.

### 5.1.1. Black Box: Core

This component contains the essential components of the application: access to the contents of the FHIR NPM packages and the configuration-controlled generation of the Java sources. It fulfills all of the requirements except R5 (Integration Capabilities). It accesses the NPM packages and the source code to be generated directly. Configuration and invocation are performed by Java objects and method calls at runtime.

This component is implemented as a Maven submodule `fhlintstone-core` of the main project.

### 5.1.2. Black Box: Command Line Interface

This component makes the functions of the core module available to the user for calling from the command line. It has a compatible configuration model that can read the configuration from an XML source file and transfer it to the internal configuration model.

This component is implemented as a Maven submodule `fhlintstone-cli` of the main project.

Since this is a very simply structured component, it is not broken down further in Level 2.

### 5.1.3. Black Box: Maven Plug-In

This component makes the functions of the core module available within a Maven build process. The plug-in allows the configuration to be specified directly in the Maven POM. It is able to transfer the specified settings to the internal configuration model.

This component is implemented as a Maven submodule `fhlintstone-maven-plugin` of the main project.

Since this is a very simply structured component, it is not broken down further in Level 2.

### 5.1.4. Interface: FHIR NPM Package Files

The source resources for code generation are available in the form of [FHIR NPM Packages](#). Both the package format and the relevant resources ([CodeSystems](#), [ValueSets](#), [StructureDefinitions](#) and others) are specified by the HL7 FHIR standard. The relevant standard sections are linked in the glossary.

### 5.1.5. Interface: Configuration

For simplicity, the structure of the configuration data is kept identical across all components of the application. The configuration is described in detail in the User Manual.



### 5.1.6. Interface: Generated Source Code

The application must generate valid Java code in accordance with the relevant specifications. For further use, the code is written to a configurable output directory. The configured package structure is observed.

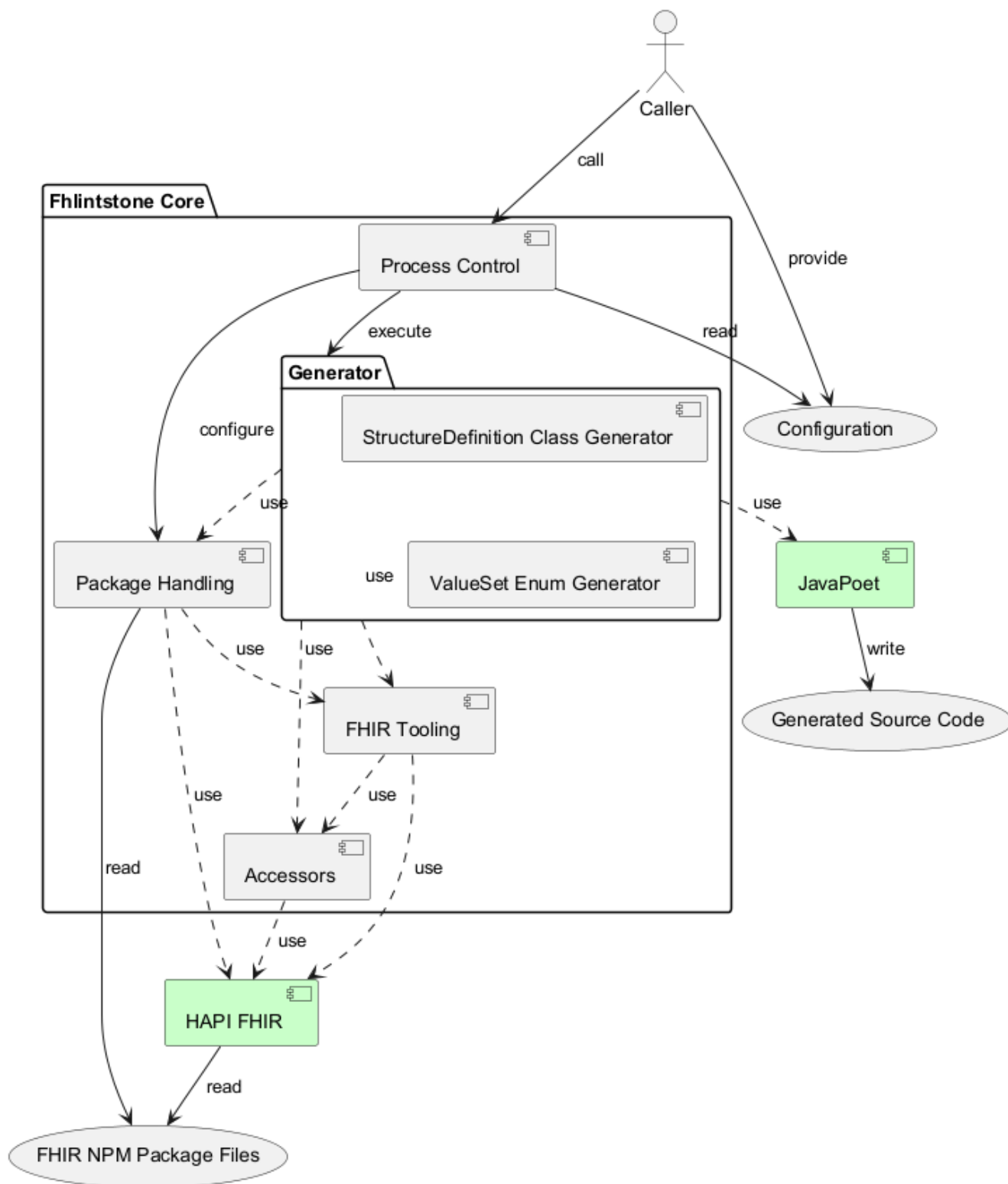
The entire code is regenerated each time the application is executed. Existing code is not updated; existing files are completely overwritten. The generated code is not intended to be adapted manually; in particular, there are no protected regions in the generated code.

Files from previous executions are **not** automatically deleted when they are no longer needed.

## 5.2. Level 2

### 5.2.1. White Box: Core





## Motivation

The core of the application consists of the components shown above. It uses two external libraries that are crucial for its operation: [HAPI FHIR](#) is used to work with FHIR packages and resources, and [JavaPoet](#) is used to generate Java source code. In addition to the actual generators, the core contains further components to facilitate access to the FHIR structures, to manage the FHIR NPM packages, and to control the generation process.

## Contained Building Blocks



Name	Description
<a href="#">Accessors</a>	Adapters to shield application from version-specific HAPI implementations.
<a href="#">FHIR Tooling</a>	Additional algorithm implementations to work with FHIR resources.
<a href="#">Package Handling</a>	Handling of FHIR NPM Packages and access to package contents.
<a href="#">Generator</a>	Modular generator subsystem.
<a href="#">StructureDefinition Class Generator</a>	Generator module to create Java classes from <a href="#">StructureDefinitions</a> .
<a href="#">ValueSet Enum Generator</a>	Generator module to create Java enums from <a href="#">ValueSets</a> .
<a href="#">Process Control</a>	Configuration handling and overall control of the generation process.

### Important Interfaces

Name	Description
<a href="#">FHIR NPM Package Files</a>	Access to the FHIR resources that describe the structures for which code has to be generated.
<a href="#">Configuration</a>	Control of the generation process.
<a href="#">Generated Source Code</a>	Output of the application.

### 5.2.2. Black Box: Accessors

Fhlintstone is designed to support multiple FHIR versions (releases). For versions up to and including R5, [HAPI FHIR](#) uses parallel inheritance structures to implement each FHIR version separately from every other version. The accessors are used to "shield" the Fhlintstone core application from these redundant structures. The goal is that the only components that access the release-dependent HAPI structures directly should be the accessors contained in this component.

This component is contained in the Java package hierarchy below `de.fhlintstone.accessors`.

This component contains additional functional implementations that are release-dependent as well:

#### Resource-Level Dependency Determination

FHIR elements frequently make use of other elements. When generating code or other derived structures, one needs to be able to identify these dependencies. This sub-component provides a (release-dependent) visitor that is able to extract the dependency information of selected resource types. Because this implementation needs to access the underlying HAPI classes directly, it is part of the accessor component.

#### HAPI Implementation Type Access

When generating code for a specific FHIR release, the generator needs information about the HAPI structures extended by the the generated code. For example, when producing an extended version of the Patient resource, the generator needs to know the actual HAPI class that



represents the base Patient for the targeted FHIR release. The contents of this sub-component provide this kind of access. Because this implementation needs to access the underlying HAPI classes directly, it is part of the accessor component.

### 5.2.3. Black Box: FHIR Tooling

This package contains implementations of various functions that are used by the code generator to examine FHIR structures. These functions include the generation of a dependency tree and the conversion of the flat list of ElementDefinitions contained in a StructureDefinition into a tree structure. This component uses the accessors and may not access the version-specific HAPI implementations directly.

This component is contained in the Java package hierarchy below `de.fhlintstone.fhir`.

### 5.2.4. Black Box: Package Handling

[FHIR profiles](#) are provided as [NPM Packages](#). This component contains a central registry that maintains an index of all packages used and provides access to the resources contained within these packages.

This component is contained in the Java package hierarchy below `de.fhlintstone.packages`.

### 5.2.5. Black Box: Generator

This component contains the code generators. Fhlintstone uses a modularized approach with separate generators for each resource type and target structure type.

This component is contained in the Java package hierarchy below `de.fhlintstone.generator`.

### 5.2.6. Black Box: StructureDefinition Class Generator

This component contains the generator module that produces Java classes corresponding to FHIR [StructureDefinitions](#). The component is split into two main parts: The first part of the generator converts the FHIR structure information into a model that can be used to produce the code generator control model. The second part uses [JavaPoet](#) to produce Java classes corresponding to FHIR [StructureDefinitions](#). This part contains the actual code generator and its control model.

This component use the accessors and may not access the version-specific HAPI implementations directly.

This component is contained in the Java package hierarchy below `de.fhlintstone.generator.structuredefinition`.

### 5.2.7. Black Box: ValueSet Enum Generator

This component contains the generator module that produces Java enums corresponding to FHIR [ValueSets](#). It converts the FHIR value information into a model that can be used to produce the code generator control model. It then uses [JavaPoet](#) to produce Java classes corresponding to FHIR [ValueSets](#).



This component use the accessors and may not access the version-specific HAPI implementations directly.

This component is contained in the Java package hierarchy below `de.fhlintstone.generator.valueset`.

### 5.2.8. Black Box: Process Control

This component contains the main process control of the Fhlintstone core logic. It prepares the environment and delegates the actual code generation to the individual generator components.

For the time being, the Fhlintstone Core has its own configuration model - contained in this component - that is mirrored by each calling component. (See [issue 87](#) for a discussion on how these redundant configuration models might be combined in a future version.)

### 5.2.9. Interface: FHIR NPM Package Files

The source resources for code generation are available in the form of [FHIR NPM Packages](#). Both the package format and the relevant resources ([CodeSystems](#), [ValueSets](#), [StructureDefinitions](#) and others) are specified by the HL7 FHIR standard. The relevant standard sections are linked in the glossary.

### 5.2.10. Interface: Configuration

The configuration is represented at this level by a Java class model, which is instantiated at runtime by the caller and passed to the processing logic. (See [issue 87](#) for a discussion on how these redundant configuration models might be combined in a future version.)

### 5.2.11. Interface: Generated Source Code

The generated code is written to the output directories specified by the configuration using the output method supplied by [JavaPoet](#).

## 5.3. Level 3

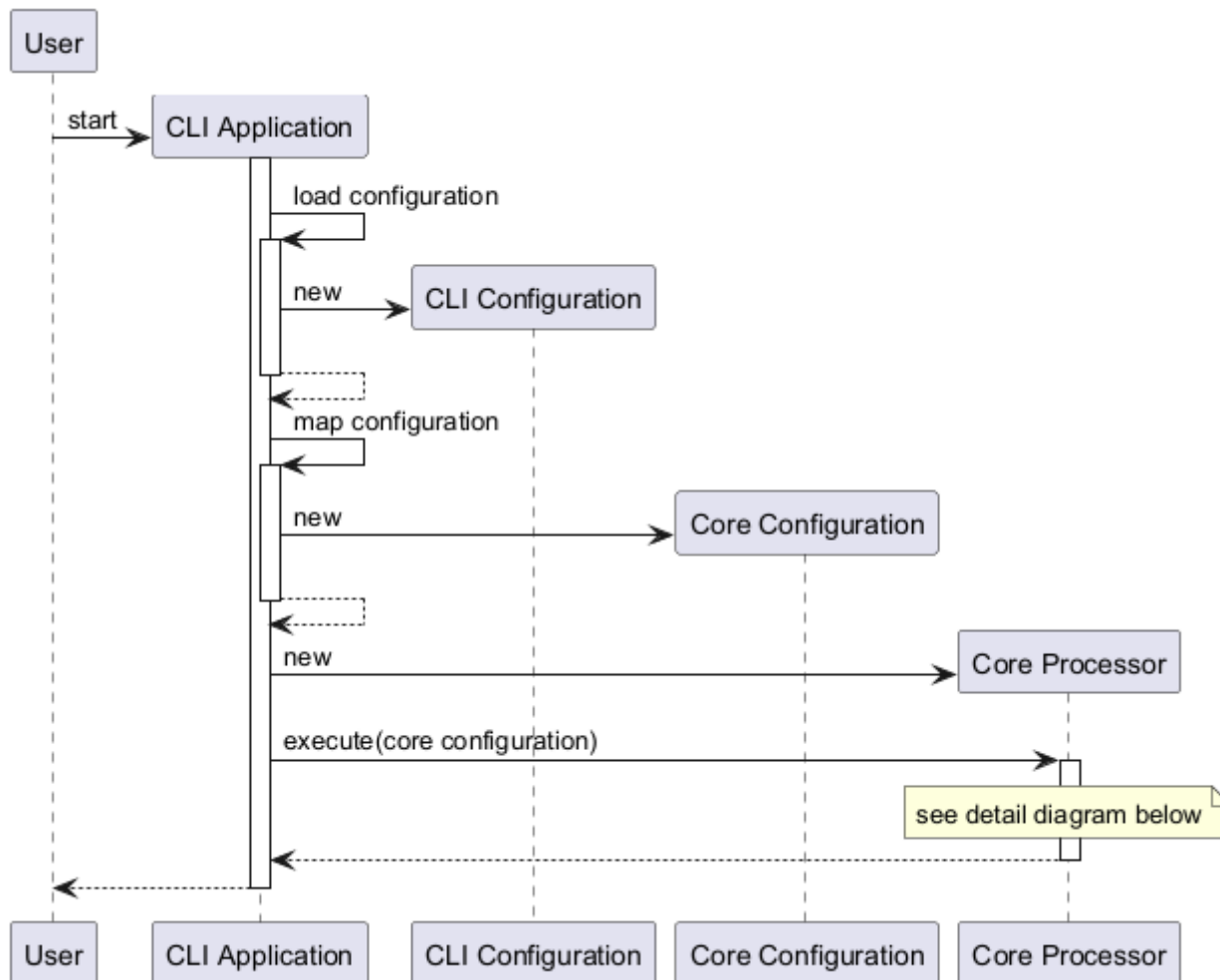
No further breakdown of the components at level 3 has been carried out to date.



## 6. Runtime View

### 6.1. Maven Integration

This scenario describes the use of Fhlintstone when integrated into a Maven build process.

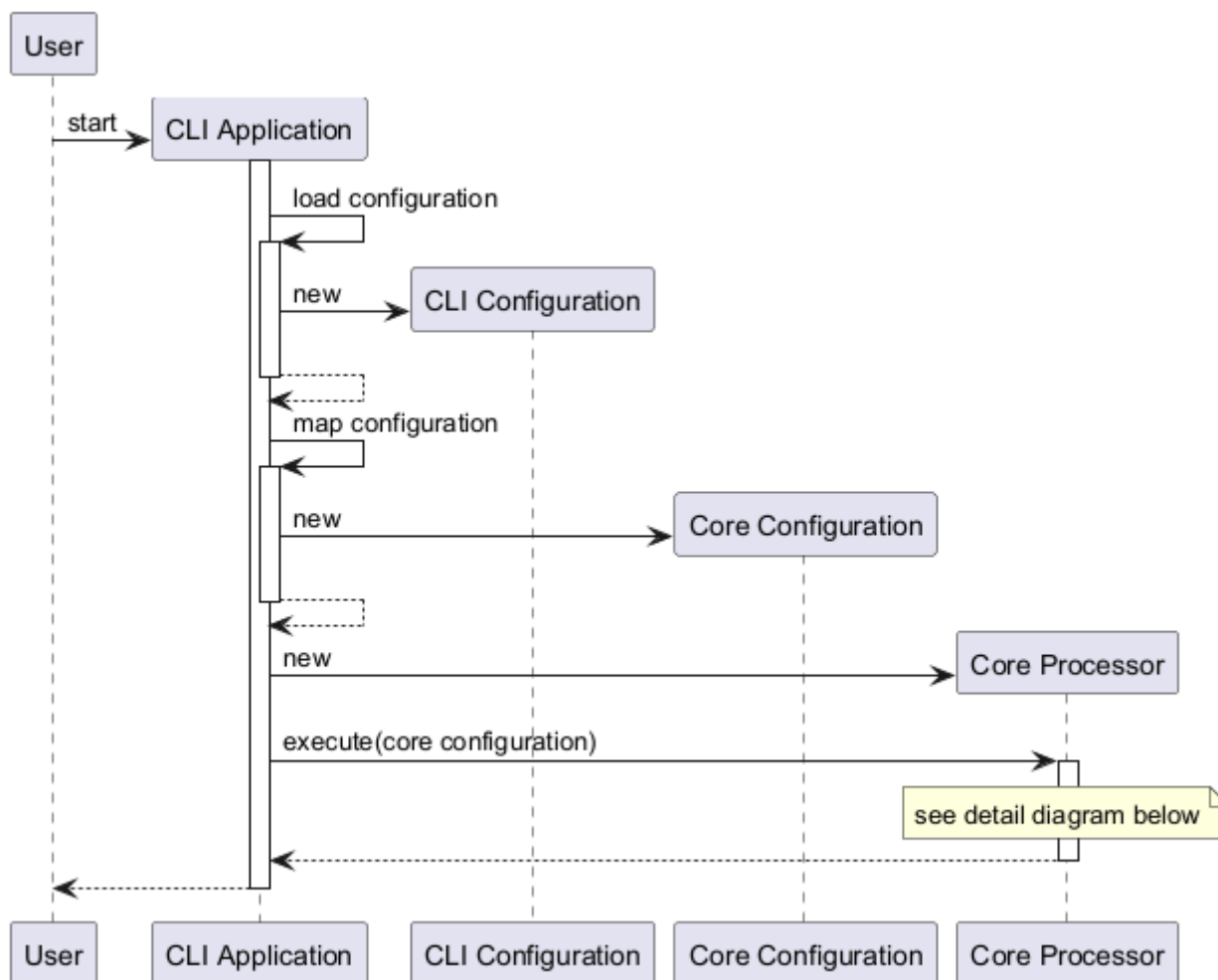


The plug-in specific configuration is injected by Maven and mapped to the core configuration. It is then handed to the core processor where it is used to generate the output files as described below.

### 6.2. Command Line Execution

This scenario describes the use of Fhlintstone when called from a command line.



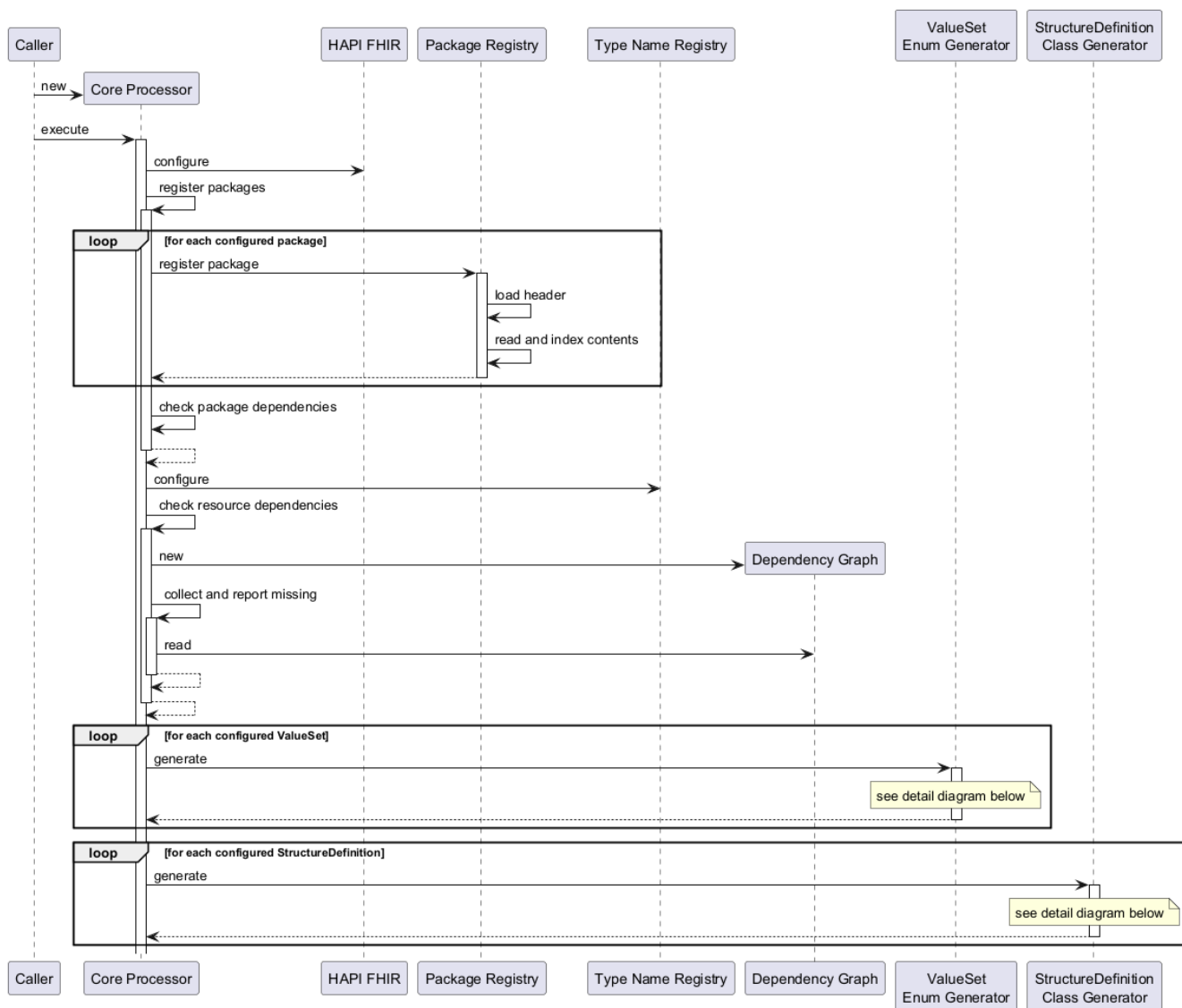


In this case, the CLI specific configuration is loaded from an XML file and mapped to the core configuration. It is then handed to the core processor where it is used to generate the output files as described below.

## 6.3. Core Processor Initialization

When the core processor is called, it initializes a number components before performing the actual generation steps. In this phase, the configured [NPM packages](#) are read and their contents are indexed for later access.

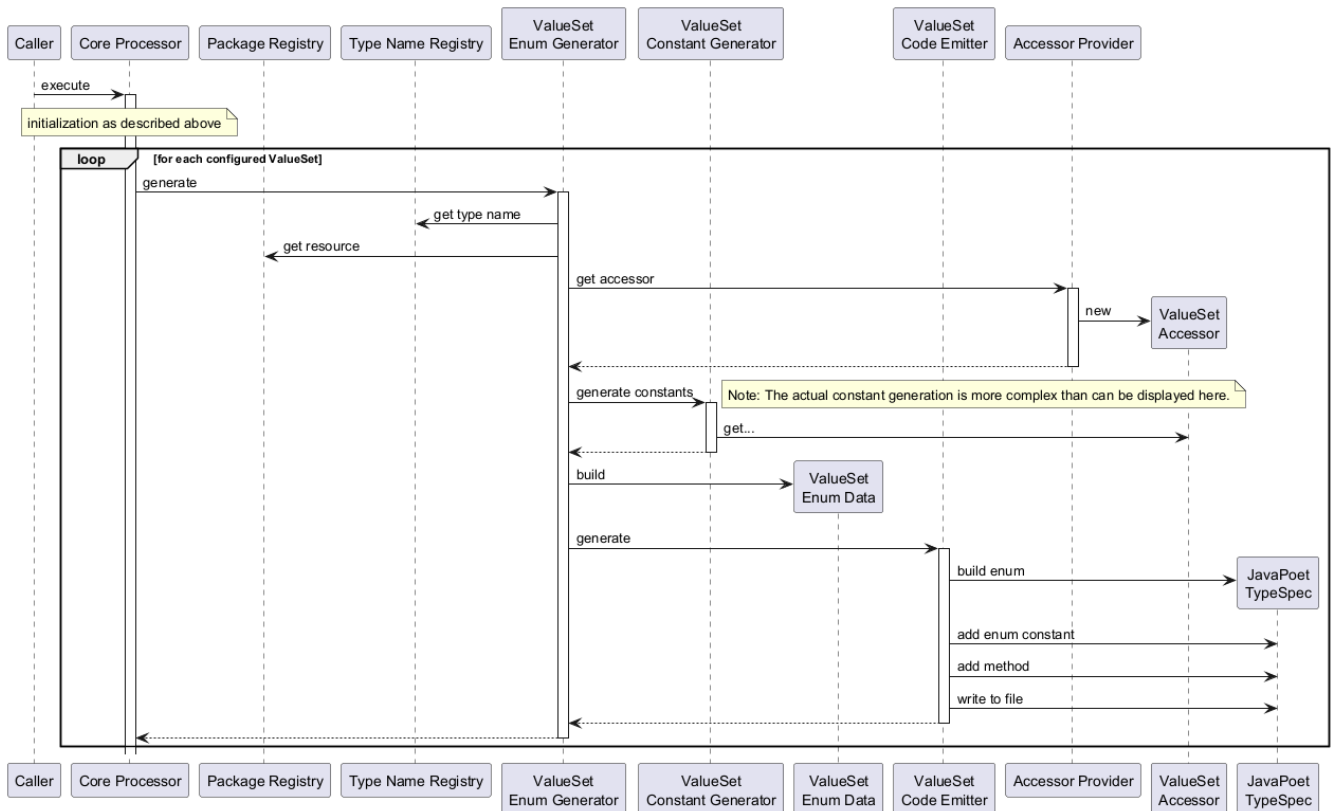




## 6.4. ValueSet Enum Generation

After initialization, the core processor executes the ValueSet enum generator once for each configured ValueSet. From the FHIR resources, the constants to be incorporated into the enum are determined. The collected information is then passed to a code emitter that controls the actual code generation using [JavaPoet](#).



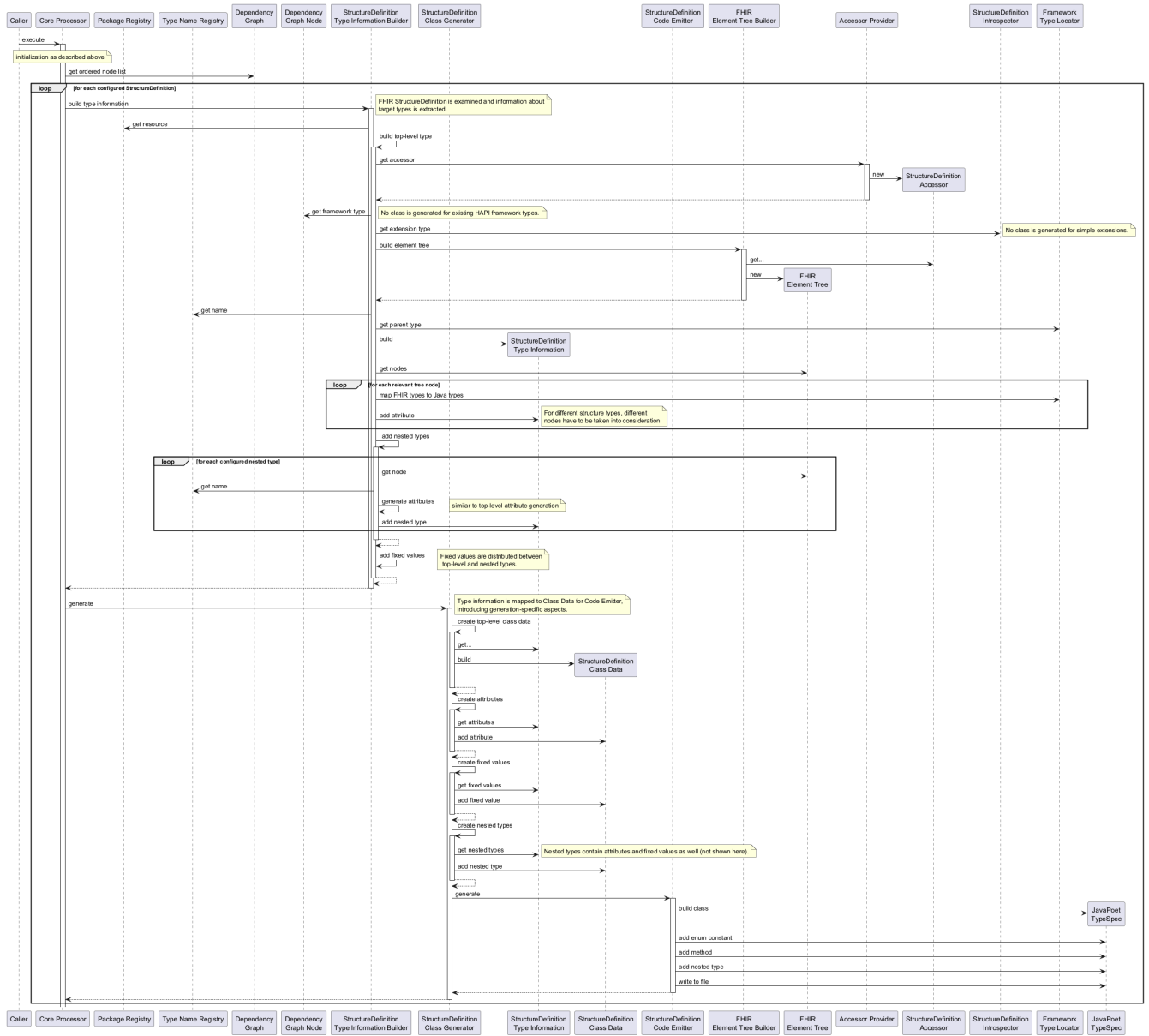


## 6.5. StructureDefinition Class Generation

After initialization, the core processor executes the StructureDefinition class generator once for each configured StructureDefinition. From the FHIR resources, the attributes to be incorporated into the class are determined. From the configuration, additional information about nested classes that need to be generated is also observed here. The collected information is then passed to a code emitter that controls the actual code generation using [JavaPoet](#).

Due to the complexity of the FHIR model, the entire process is rather complex. The following diagram only gives a rough overview of the main processes.







Flintstone is used within the development lifecycle of the host project it is used for. It is not "deployed" in the sense that it requires explicit installation on any dedicated hardware or other similar resource. For this reason, no deployment documentation has as of yet been deemed necessary.



## 7. Cross-cutting Concepts

Note: Further information about "low-level" cross-cutting concerns may be found in the separate developer handbook.

### 7.1. Automated Tests

As is standard practice today, Fhlintstone also strives for a high level of functional correctness through extensive automated testing. Three different levels of testing are distinguished.

#### Unit Tests

Unit tests represent the lowest level of testing and check individual classes in isolation from their environment. The environment is simulated using a mocking framework ([Mockito](#)).

#### Integration Tests

Integration tests check the behavior of a class in conjunction with the implementations with which it will be used in the final product. Specially created test data sets are used for this purpose.

Background: It has become apparent that for many tests it is not feasible to adequately mock the entire environment of a class. This is especially true for classes that interact directly or indirectly with the (very complex) HAPI FHIR framework. Here, the effort required to create mocks that faithfully reproduce the behavior of the framework is very high. At the same time, the probability of errors in the mocks is high, which means that a lot of time is lost searching for phantom errors.

#### System Test

Unit tests and integration tests check the behavior of the Fhlintstone application in individual components. The integration tests extend this by testing the entire application in context and also testing the generated code. This ensures that not only the code generator runs without errors, but also that the generated code behaves as expected.



## 8. Architecture Decisions

No formal ADRs have been recorded so far.



## 9. Quality Requirements

No quality requirements beyond those already noted in the [introduction](#) have as of yet been documented.



## 10. Risks and Technical Debts

Known issues, limitations and implementation-level technical debt is tracked using the [issue tracker](#). The relevant sections in the code are marked with **TODO** or **FIXME** comments followed by the issue number.

Beyond this tracking, no further documentation of risks and technical debt currently exists.



# 11. Glossary

Term	Definition	Links
<i>CodeSystem</i>	<i>The CodeSystem resource is used to declare the existence of and describe a code system or code system supplement and its key properties, and optionally define a part or all of its content. Code systems define which codes (symbols and/or expressions) exist, and how they are understood.</i>	<a href="#">Resource CodeSystem - Content</a>
<i>Differential statement</i>	<i>Differential statements are one of two ways to describe the inner structure of a <a href="#">StructureDefinition</a>. Differential statements describe only the differences that they make relative to the base structure definition. In order to properly understand a differential structure, it must be applied to the structure definition on which it is based.</i>	<a href="#">Base Resource Definitions</a>
<i>Extension</i>	<i>Every element in a <a href="#">Resource</a> can have extension child elements to represent additional information that is not part of the basic definition of the resource. The use of extensions is what allows the FHIR specification to retain a core simplicity for everyone. To make the use of extensions safe and manageable, there is strict governance applied to the definition and use of extensions. Although any implementer can define and use extensions, there is a set of requirements that must be met as part of their use and definition.</i>	<a href="#">Extensibility</a>
<i>FHIR NPM Package</i>	<i>A set of <a href="#">Resources</a> bundled as a machine-readable archive file. Not to be confused with <a href="#">FHIR Package</a>.</i>	<a href="#">FHIR NPM Packages</a>
<i>FHIR Package</i>	<i>In the context of <a href="#">Profiling</a>, a group of related adaptations that are published as a group within an Implementation Guide. Not to be confused with <a href="#">FHIR NPM Package</a>.</i>	<a href="#">Profiling FHIR</a>
<i>FHIR Profile</i>	<i>A set of constraints on a <a href="#">Resource</a> represented as a <a href="#">StructureDefinition</a>.</i>	<a href="#">Profiling FHIR</a>
<i>HAPI FHIR</i>	<i>The HAPI FHIR library is an implementation of the <a href="#">HL7 FHIR</a> specification for Java.</i>	<a href="#">HAPI FHIR</a>
<i>HL7 FHIR</i>	<i>Fast Healthcare Interoperability Resources (FHIR) are a standard for health care data exchange, published by HL7®.</i>	<a href="#">HL7 FHIR</a>



Term	Definition	Links
<i>Implementation Guide</i>	<i>A coherent and bounded set of adaptations that are published as a single unit. Validation occurs within the context of the Implementation Guide.</i>	<a href="#">Profiling FHIR</a>
<i>JavaPoet</i>	<i>JavaPoet is a Java API for generating <code>.java</code> source files.</i>	<a href="#">JavaPoet (Github)</a>
<i>Mockito</i>	<i>Mockito is a mocking framework that provides a clean &amp; simple API, resulting in readable tests and clean verification errors.</i>	<a href="#">Mockito</a>
<i>Resource</i>	<i>A resource is an entity that has a known identity by which it can be addressed, identifies itself as one of the types of resource defined in the <a href="#">HL7 FHIR</a> specification, contains a set of structured data items as described by the definition of the resource type and has an identified version that changes if the contents of the resource change.</i>	<a href="#">Base Resource Definitions</a>
<i>Snapshot statement</i>	<i>Snapshot statements are one of two ways to describe the inner structure of a <a href="#">StructureDefinition</a>. Snapshot statements are a fully calculated form of the structure that is not dependent on any other structure.</i>	<a href="#">Base Resource Definitions</a>
<i>StructureDefinition</i>	<i>A definition of a FHIR structure. This resource is used to describe the underlying resources, data types defined in FHIR, and also for describing extensions and constraints on resources and data types.</i>	<a href="#">Resource StructureDefinition - Content</a>
<i>ValueSet</i>	<i>A ValueSet resource instance specifies a set of codes drawn from one or more code systems, intended for use in a particular context. ValueSets link between <a href="#">CodeSystems</a> definitions and their use in coded elements.</i>	<a href="#">Resource ValueSet - Content</a>