4

The Genesys Framework

The Genesys code generation framework is a reference implementation of the ideas that constitute the Genesys approach presented in this book. As mentioned in Sect. 3, the jABC framework and its underlying XMDD approach form the technical and conceptual basis for this implementation. At the same time, jABC is also an appropriate domain for applying the Genesys framework in case studies (see Chap. 5).

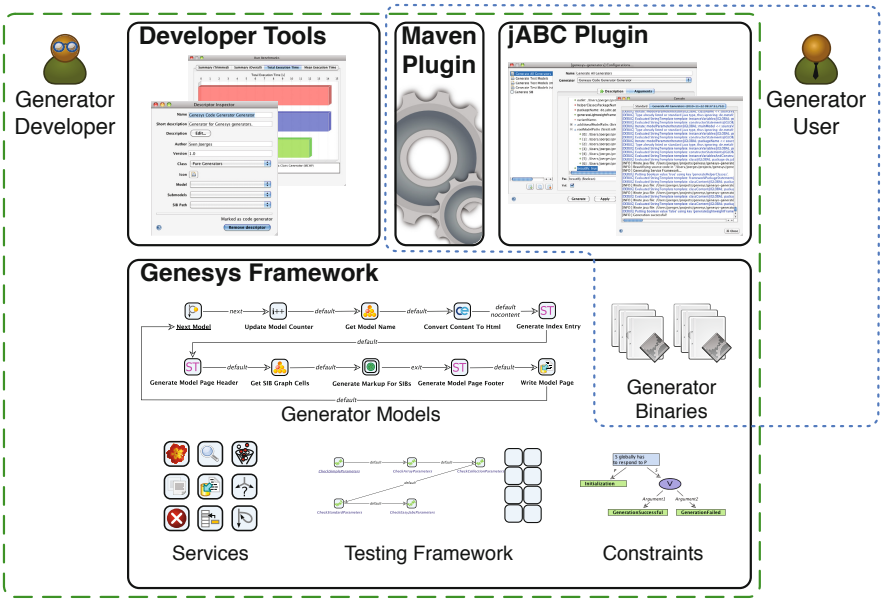

Fig. 4.1. Genesys architecture and involved roles

Fig. 4.1 shows how the reference implementation is organized, the central part being the actual *framework*, which consists of the following components:

S. Jörges: Construction and Evolution of Code Generators, LNCS 7747, pp. 75–100, 2013.

Services: The framework provides a library of services that cover typical functionality required for most code generators, such as type conversion, identifier generation, model transformations and code beautification (*Requirement G2 - Reusability and Adaptability*). These services are available as SIBs (cf. Sect. 3.2), so that they can be used as atomic building blocks for code generator models built with jABC. Sect. 4.1 further elaborates on this service library.

Generator Models: As described in Sect. 3.2, XMDD does not only support the reuse of services, but also of entire models (*Requirement G2 - Reusability and Adaptability*). Consequently, the framework contains a library of code generator models which realize further typical functionality such as loading and traversing input models, e.g., jABC's SLGs or EMF models (cf. Chap. 7). Just like the atomic services mentioned above, these models can be directly reused as macros when building a new code generator (thus representing ready-made *features* in the sense of XMDD, see Sect. 3.1). They can also serve as patterns which are instantiated or adapted for new code generators. Furthermore, the model library contains most code generators created in the case studies which will be presented in Chap. 5–8. The rationale behind this is that each new code generator contributes to this library of models, so that the available repertoire and the potential for reuse is growing continuously (*Requirement G2 - Reusability and Adaptability*).

Generator Binaries: In order to be accessible by tools and users, the framework also includes all code generators as compiled Java classes. For this purpose, each modeled code generator is translated to an appropriate Java class via the *Genesys Code Generator Generator* (see Sect. 5.2.6).

Testing Framework: As an addition to manually testing a code generator by executing its models with the Tracer (cf. Sect. 3.3), the Genesys framework also includes an approach for the automated model-driven testing of code generators. In this approach, test cases as well as the corresponding test inputs are also modeled as SLGs. Subsequently, dedicated code generators (again developed with Genesys) translate these SLGs into test scripts or test programs running on a desired test platform. Sect. 6.3 elaborates on the details of this approach, which is currently realized for jABC code generators, i.e., those which support SLGs as input models (see Chap. 5). Accordingly, the provided facilities include a library of SIBs for building test cases and test data models, a collection of standard test cases covering the domain of jABC models as well as a testing strategy for checking whether the execution semantics of a model is retained by the code generation (cf. Sect. 6.3.1).

Constraints: For verifying code generators via model checking, the framework provides a library of global constraints which specify required properties such as the complete processing of the input data or proper handling of errors. These constraints are specified graphically as *formula graphs* and then translated to temporal logics by the FormulaBuilder (cf. Sect. 6.2.1), which is also an application of jABC and Genesys. Sect. 6.2 presents examples of such global constraints for code generators and shows how they are specified.

Above the framework, Fig. 4.1 shows *tools* that support the usage and the development of code generators. The *jABC plugin* allows the configuration and invocation of code generators within the jABC editor, and the *Maven Plugin* enables the integration into a Maven-based project setup. Finally, the *developer tools* bundle utilities that support the creation of code generators. All tools will be presented in more detail in Sect. 4.3.

Fig. 4.1 also indicates the roles that are targeted by Genesys. First, Genesys addresses *generator developers* (left hand side of Fig. 4.1) by providing facilities that support creating, adapting, verifying and testing code generators. For this purpose, they use a specific jABC bundle tailored to the domain "code generation", containing all services, models and tools depicted in Fig. 4.1. In combination, the ready-made services and models form a *DSL for building code generators*. Second, Genesys offers a library of ready-made code generators which can be used without deeper knowledge of the Genesys approach or the internals of the framework. *Generator users* (right hand side of Fig. 4.1) may access these generators by means of the tools outlined above (jABC plugin, Maven plugin), or integrate the generator binaries into their own applications via API.

When Genesys is applied to jABC itself, i.e., used for developing a code generator whose source language is given by SLGs, the above two roles can be related to the jABC roles introduced in Sect. 3.2. In this specific scenario, which is examined in greater detail in Chap. 5, the generator developer equals the domain expert and the generator user is congruent with the application expert. When creating a domain-specific jABC variant for the application expert, the domain expert also has to define how SLGs are translated to code in the target domain by creating new code generators with Genesys or selecting existing ones. The application expert then uses the jABC variant customized by the domain expert to model and to generate an application for the target domain.

The following sections will elaborate on Genesys' constituent parts, starting with the atomic services for creating code generator models (Sect. 4.1). In order to give an idea of modeling a code generator with Genesys, Sect. 4.2 describes a complete example of a simple code generator. Finally, Sect. 4.3 presents all tools provided by Genesys.

4.1 Services for Building Code Generators

The Genesys framework is equipped with many ready-made atomic services which are made available as SIBs, the most important building blocks for creating code generator models. Following the principles of service orientation, these services are black boxes with very simple interfaces, and, in consequence, easy to use without any knowledge of their actual implementation (*Requirement G3 - Simplicity*). The availability of such ready-to-use services saves the generator developer from having to start from scratch, thus speeding up the overall development (*Requirement G2 - Reusability and Adaptability*). The following sections present services which are especially relevant to the code generation domain. As the description of all available SIBs (more than 200 Common SIBs and around 40 Genesys-specific SIBs) would go beyond the scope of this book, there is a focus on those services which are essential to most code generators. For an exhaustive list of all available SIBs please refer to the documentation of the Common SIBs [TU10] and the Genesys SIBs [Jör10].

Please note that the following descriptions focus on the parameters when explaining the structure of the SIBs. This is due to the fact that all presented SIBs provide the same two branches: `default`, meaning that the service has been executed successfully, and `error`, indicating that the execution of the service has failed.

4.1.1 Contributions to the Common SIBs

During the development of Genesys, many SIBs had to be created, in particular at the very beginning of the project. At this time, Genesys significantly pushed the development of jABC's Common SIBs (cf. Sect. 3.2.1), thus contributing to the availability of general jABC services. Any created SIB which was suitable to be used with a broader scope than code generation has been added to the Common SIBs.

The following SIB bundles have been heavily influenced by Genesys and thus are used in nearly any code generator:

Basic SIBs: The SIBs contained in this bundle provide very basic functionality which is required by nearly any application modeled in jABC (especially by any Genesys code generator). This mainly includes building blocks for manipulating the execution context (see Sect. 3.3.2), e.g., creating, modifying, removing or copying context entries, as well as for thread-safe access, and locking or unlocking context entries. Furthermore, the bundle provides SIBs that realize control flow patterns such as conditional constructs ("if" and "switch") and loops. There are also building blocks for the basic support of application-level logging and performance measurement. Due to their elementary character, most SIBs in this bundle are very generic and represent rather fine-grained functionality.

Graph Model SIBs: This bundle provides building blocks for handling SLGs and hence is relevant to code generators that are built for jABC (cf. Chap. 5). It includes SIBs for loading and traversing hierarchical SLGs as well as for retrieving information from contained SIBs (e.g., SIB labels, parameters, branches, outgoing edges, successors in the SLG, or user objects). Due to this reflective character, the bundle can be considered a metamodel API for accessing SLGs and their constituent parts. The more than 60 SIBs of this bundle were originally a part of Genesys, but have been contributed to the Common SIBs in order to enable SLG accessibility for other jABC applications. Please note that in order for a code generator to be able to deal with another modeling language than SLGs, another SIB bundle suitable for the new modeling language has to be employed. This is exemplified by the case study presented in Chap. 7, which required the development of a SIB bundle for accessing EMF models.

IO SIBs: These SIBs mainly support dealing with files and directories, e.g., creating and scanning directories or writing text files. It is used by Genesys for writing generated code to actual files. Furthermore, the bundle contains simple building blocks for exception and error display on the console.

Script SIBs: This is a set of more specialized SIBs that enable the execution of scripts as well as the integration of template engines, the latter being particularly relevant to Genesys. Currently, the template engines Velocity [Apa10], StringTemplate [Par04][1] and FreeMarker [Fre11b] are supported. Consequently, the generator developer can freely decide upon an appropriate template engine by just using the corresponding SIB (*Requirement G1 - Platform Independence*). It is also possible to mix several template engines in one code generator model, e.g., in order to benefit from a specific feature of a template language without having to use the corresponding engine for the entire code generator.

Apart from those very general SIBs, which can be used in any jABC application and mostly realize small, fine-grained tasks, there are also services specifically designed for the code generation domain. Several examples of those SIBs, which are part of the Genesys framework (cf. Fig. 4.1) and form a bundle called "Genesys SIBs" [Jör10], are presented in the following sections.

4.1.2 Type Mapping

An essential task of a code generator is mapping the data types found in the source language to corresponding data types of the target language. In middleware techniques like CORBA and Web Services [Pap08], data type mapping usually includes the use of intermediate exchange formats such as the Common Data Representation (CDR) for CORBA or XML dialects like

[1] The `RunStringTemplate` SIB described in Sect. 3.2 is also contained in the "Script SIBs".

SOAP [W3C07] for Web Services. Thus marshalling and unmarshalling is required, i.e., the conversion of the data types into the exchange format and vise versa. In the case of Web Services, e.g., this conversion is performed by XML binding libraries like the Java Architecture for XML Binding (JAXB) [Jav09a].

Type Mapping Scenarios:

For code generators, this task is usually much simpler, as it involves a direct unidirectional translation of the data type to a corresponding text-based representation (e.g., an initializer) in the target language. The overall complexity of this task strongly depends on the given combination of source and target language. For this book, the following cases are distinguished:

1. *Identity:* The exact same data type exists in the source language as well as in the target language. For instance, when translating jABC's SLGs to Java, the data type `java.lang.String` exists in both languages, due to the fact that SIBs are realized as Java classes.
2. *Direct Mapping:* The data type can be mapped to an equivalent type in the target language without any loss or omission of information. For instance, when generating Java code from EMF models, the data type `EString` in Ecore (cf. Sect. 7.2) can be directly mapped to the Java data type `java.lang.String`. Such a mapping does not necessarily have to be injective, as it is possible to map several data types which are different in the source language to one single data type in the target language.
3. *Reductive Mapping:* If the data type has no direct equivalent in the target language, it possibly can be reduced, provided that the source data type contains information that is not required for the generated result. For instance, the built-in data type `ContextKey` in jABC's SLGs (cf. Sect. 3.2.1) has no direct counterpart in any target language. However, it contains a context scope, which might be omitted if the target language does not support a stacked execution context. In this case, `ContextKey` can be reduced to the name of the context key, which can be easily represented by a simple string in the target language. This method only works if the omitted information is irrelevant, otherwise a solution without information loss is preferable. An additive counterpart of the reductive mapping is imaginable, but as no practically relevant examples have been found in the context of Genesys, this case is not considered here.
4. *New Data Type:* There is no equivalent counterpart for a source data type in the target language, neither for a direct nor for a reductive mapping. In this case, a possible solution is the introduction of a new data type in the target language. The corresponding code of the new data type has to be emitted by the code generator. As will be presented in Sect. 5.2, this solution is applied by the code generator which translates SLGs to plain Java code. For every complex jABC data type (cf. Sect. 3.2.1), such as `ContextKey` or `ListBox` which are not built-in Java types, the code generator emits a corresponding counterpart.

5. *Data Type Exclusion:* If neither of the above cases is applicable, a source data type may be excluded from the mapping, at the expense of no longer being able to translate any inputs containing that data type. In most cases, this solution is only a compromise and thus should be avoided. For instance, the `ContextExpression` data type in jABC's SLGs (cf. Sect. 3.3.2) requires an implementation for resolving EL expressions, which is not available for non-Java languages such as Objective-C [Koc09]. Instead of spending effort on providing such an implementation, the exclusion of the `ContextExpression` data type might be a pragmatic alternative.

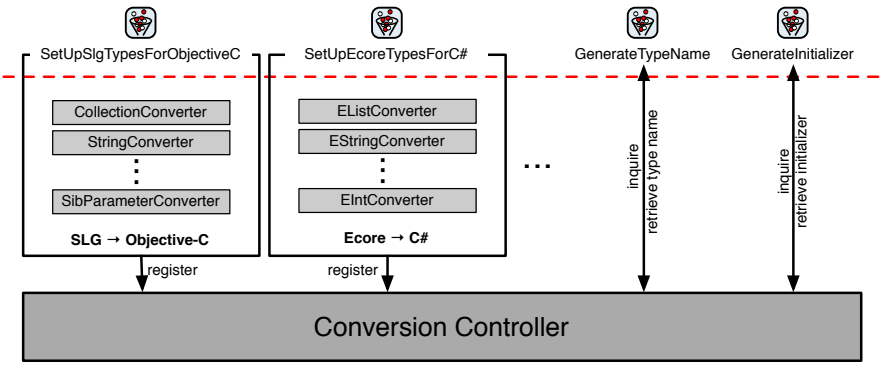

Fig. 4.2. Data type mapping infrastructure in Genesys

Type Mapping Services:

Genesys provides a simple infrastructure for establishing data type mappings, which is depicted in Fig. 4.2. For each combination of source and target language, a mapping has to be specified by means of a set of *converters*. A converter is responsible for one or more source data types and is able to produce corresponding type names, initializers etc. for the target language. For instance, Fig. 4.2 shows a data type mapping from SLGs to Objective-C, containing a *String Converter*. For a given `String` instance found in an SLG, this converter is, among other things, able to produce a corresponding type name (`NSString`) or initializer (`@"myStringValue"`[2]), which is used by the code generator.

All registered converters are managed by the *Conversion Controller* (bottom of Fig. 4.2). Given a concrete primitive value or object instance found in the source language, the controller determines the responsible converter, applies it and returns the result. In order to keep the determination mechanism

[2] `@"myStringValue"` is Objective-C's way of creating a new constant string object with the value `myStringValue`.

simple, each possible data type in the source language has to be assigned to exactly one converter. While multiple data types may be handled by one converter, it is not allowed to assign more than one converter to a data type.

From the perspective of the generator developer, the data type mapping infrastructure is accessed via simple services shown at the top of Fig. 4.2. In the initialization phase of the code generator, the system has to be set up by registering required converters, which is performed by a dedicated, usually parameterless, SIB (e.g., `SetUpSlgTypesForObjectiveC`). Such a SIB and its corresponding converters have to be created for every new combination of a source and a target language, either by the generator developer himself or by a SIB expert. Technically (and transparent to the generator developer), the SIB registers all required converters with an instance of the conversion controller, which is then stored in the execution context (cf. Sect. 3.3.2) so that it can be accessed by other SIBs.

After this initialization step, the generator developer may use the SIBs `GenerateTypeName` and `GenerateInitializer` to apply the data type mappings for the code generation. Table 4.1 shows the parameters of those services that are available to the generator developer. Besides context keys for the input and output, especially the `GenerateTypeName` SIB provides several additional configuration flags that, e.g., allow the omission of any package or namespace information in the resulting type name. Except for the case in which the generator developer writes converters by himself, everything below the dashed line in Fig. 4.2 is transparent to him and virtualized by the SIBs.

Table 4.1. Services for data type mapping

GenerateInitializer		Generates an initializer string for the given object/primitive value.
Parameters	object	Context key for reading the object.
	initializer	Context key for storing the generated initializer.
GenerateTypeName		Generates a type name string for the given object/primitive value.
Parameters	object	Context key for reading the object.
	typeName	Context key for storing the generated type name.
	generateSimple-TypeName	If this flag is set to true, the simple type name (without the package or namespace information) will be generated.
	preferInterface	If this flag is set to true, the type name for a preferred interface will be generated (e.g., `java.util.List` instead of `java.util.ArrayList`), which is useful for declarations.

4.1.3 Identifier Generation

Another important task of code generators is the output of valid identifiers. Identifiers are tokens that name elements in programming languages such as classes, variables, labels or methods. The generation of appropriate identifiers depends on the given target language, as each programming language defines its own syntactic restrictions for correct identifiers. For instance, a valid identifier in Java is composed of characters such as lowercase or uppercase ASCII Latin letters, digits, underscores or dollar signs, but must not begin with a digit, contain any blanks or have the same spelling as a reserved language keyword such as "`public`" [Gos+05]. Furthermore, some programming languages even assign semantics to single characters in identifiers. In Perl, identifiers beginning with a dollar sign (`$`) indicate scalars, and those starting with the percent sign (`%`) denote hashes [Wal00]. As another example, in Ruby, variables with identifiers that start with an upper case letter are considered immutable [FM08]. Thus with each new target language, identifier generation needs to be specified appropriately.

Apart from the syntactic restrictions, a central characteristic of identifiers is their uniqueness. For example, in most programming languages it is forbidden to declare two variables with the same identifier within the same scope. When generating code from models, identifiers are often produced from the names of model elements, which usually are subject to various syntactic restrictions, or which are not regulated at all. For instance, the labels of SIB instances in jABC's SLGs are not restricted and thus may contain blanks, or even may equal reserved keywords of a target language. Simply using such names as identifiers and ignoring the rules of the target languages may lead to faulty generation results, such as code which is not compilable or which yields unexpected execution behavior.

In order to cope with this recurring task, the Genesys framework provides a backend for identifier generation, which can be accessed by corresponding services. In essence, this backend keeps track of a blacklist of reserved and previously used identifiers. When a new identifier is given, this blacklist is checked: If the new identifier is already on the blacklist, then it is *unified*, e.g., by adding a suffix like "`_<serial number>`", and finally added to the blacklist. Otherwise, the unmodified identifier is added to the blacklist. For instance, if "`public`" is given as an identifier and it is contained in the blacklist due to being a reserved keyword, it will be unified to "`public_1`", which can be safely used in generated code.

In the initialization phase of a code generator, reserved identifiers such as keywords of the target language can be added to the blacklist via the SIB `RegisterReservedKeywords`. At this point it is necessary to distinguish between two kinds of identifiers that occur in code generation: those which are derived from the code generator's input (*generated identifiers*) and those which are fixed, e.g., because they are hard-coded in a template (*fixed identifiers*). In contrast to generated identifiers, fixed identifiers are static at generation time

Table 4.2. Services for identifier generation

RegisterReservedKeywords		Creates a blacklist containing keywords and prefixes which are forbidden to be used for identifiers in generated code.
Parameters	reservedKeywords	Set containing the reserved keywords.
	reservedPrefixes	Set containing prefixes for reserved keywords. All words starting with one of these prefixes are not allowed.
UnifyString		Unifies a given string.
Parameters	string	Context key for reading the string that should be unified.
	uniqueString	Context key for storing the unified string.
GenerateJavaIdentifier		Converts a given string into a valid Java identifier.
Parameters	string	Context key for reading the string that should be converted.
	identifier	Context key for storing the resulting identifier.
GenerateUniqueJavaIdentifierForSlg		Generates a unique Java identifier for the given SLG, derived from its name.
Parameters	model	Context key for reading the SLG.
	uniqueIdentifier	Context key for storing the resulting unique identifier.

and are specified by the generator developer when modeling a code generator. In order to avoid clashes between such generated and fixed identifiers, the generator developer might add each fixed identifier to the blacklist as a separate reserved keyword. As this is rather uncomfortable, it is, apart from prohibiting entire words, also possible to define reserved prefixes. Each identifier starting with a reserved prefix is treated as if it would already be contained in the blacklist. Consequently, as a convention for the generator developer, any fixed identifier has to start with such a reserved prefix (e.g., "`cg_`").

After initializing the blacklist, any given string may be unified by using the SIB `UnifyString`. This SIB expects a string in the execution context, invokes the backend for identifier generation to unify it, and finally stores the resulting unique string in the execution context for further usage. However, before a string can be unified, it has to be converted to a valid identifier, adhering to the rules of the target language for which the code is generated. Consequently, for each different target language, a corresponding specific service is required.

For instance, the SIB `GenerateJavaIdentifier` converts a given string into a Java identifier according to the rules outlined above. Just as with the converters for data types described in Sect. 4.1.2, such a specific SIB has to be created if it does not exist yet, either by the generator developer himself, or by a SIB expert. However, once the SIB has been created, it can

be reused for every code generator that targets the corresponding language (*Requirement G2 - Reusability and Adaptability*), which is a major benefit of service orientation.

Optionally, even more specific SIBs may be added as desired, e.g., to provide more convenient identifier generation for specific combinations of source and target languages (*Requirement S1 - Domain-Specificity*). As an example, the SIB `GenerateUniqueJavaIdentifierForSlg` creates an identifier from a given SLG by extracting the name of the SLG, converting it to a Java identifier and then unifying it in one step. Furthermore, it additionally caches all identifiers that have already been generated for SLGs, so that executing the service repeatedly for the same SLG always yields the same identifier. This is very useful if the identifier occurs at multiple places in the generated code.

Table 4.2 sums up the services presented in this section.

4.1.4 Variant Management

When reusing existing code generators as drafts or templates for building new ones, it is an obvious approach to start with a code generator that is as similar as possible to the one that should be developed. For instance, in the context of code generators for jABC, it was easy to derive a code generator for Java Servlets [Jav11b] from an existing code generator for ordinary Java classes with only few modifications (cf. Sect. 5.4.1), as the structure and generated output of both is very similar.

Genesys is designed to facilitate software product line engineering [CN01; PBL05] (*Requirement S3 - Variant Management and Product Lines*). Using the terminology established in this realm, the models and services contained in the Genesys framework are *core assets* that form the basis for building *product lines* and deriving *variants*.

Accordingly, appropriate tool support is required that enables specifying and managing variability, e.g., via the definition of *variation points* [PBL05, p. 62]. The Genesys framework enables this by offering a dedicated service. Please note that as its implementation was driven by the demands raised during the realization of Genesys, this service does not exploit the full potential of applying product line engineering in jABC yet – Sect. 10 elaborates on further prospects and possibilities.

The service enables the specification of variants on the basis of aspect orientation (AO) [Kic+97] provided by jABC's hierarchical modeling facilities. As described in Sect. 3.2.2, hierarchical modeling is performed in jABC via macros, which enable embedding SLGs into each other. On this basis, models can act as reusable aspects (*Requirement G4 - Separation of Concerns*): For managing variability, macros are used as variation points (or *joinpoints* in AO nomenclature) to which multiple submodels may be assigned, each of them representing a single variant. This is an extension of jABC's standard

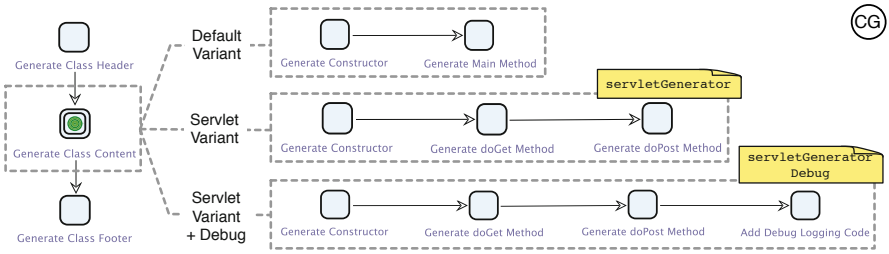

Fig. 4.3. Specifying variants via hierarchical modeling

semantics for hierarchical models, which normally only allows assigning exactly one submodel to a macro.

Fig. 4.3 shows a simplified and schematic example of this concept. On the left hand side, there is a simple model that represents the generation of a Java class: First, the header of the class is generated, followed by the content, and finally the remainder of the class is added. The SIB for generating the class content is a macro and thus a potential variation point.

In order to obtain a complete model that is well-formed in terms of executability, at least one submodel, which represents the default behavior, has to be assigned to the macro. In Fig. 4.3, this submodel is labeled "Default Variant" and generates the content of an ordinary Java class, consisting of a constructor and a `main` method[3].

The modeler may now specify further variants, which are kept apart by unique names. The example in Fig. 4.3 shows two variants. The first ("Servlet Variant") with the unique name `servletGenerator` produces the content of a Servlet class by generating a `doGet` and a `doPost` method[4] instead of the `main` method defined in the default behavior. The second ("Servlet Variant + Debug") with the unique name `servletGeneratorDebug` extends the Servlet variant by adding extra code for debugging.

Please note that the unique name of a variant does not necessarily refer to one single variant model only. Instead it *globally* identifies a variant (or product line) of an entire SLG hierarchy (or even several hierarchies). For instance, the variant name `servletGeneratorDebug` in Fig. 4.3 may refer to a set of variant models that are associated with variation points distributed over the SLGs hierarchy.

As the assignment of multiple submodels to a macro is not supported by jABC, this feature is added by the Genesys jABC plugin (see Sect. 4.3.2). Please note that technically, only the submodel that represents the default behavior is a physical part of the model hierarchy. All other variants assigned to a macro are just attached to it as additional information, but are not incorporated into the model hierarchy.

[3] `main` is a dedicated method used for starting the execution of Java programs.

[4] See the Servlet specification [Jav11b] for details on the `doGet` and `doPost` methods.

Table 4.3. Service for the generation of variants

BuildVariant		Builds the variant identified by the given name for the given models.
Parameters	models	Context key for reading the models that will be transformed.
	variantName	The unique name of the variant that will be built.
	transformedModels	Context key for storing the transformed models.

For achieving this, the specified variants are automatically generated by means of an accordingly equipped generator generator. To this end, the generator generator needs to incorporate the `BuildVariant` SIB (see Table 4.3), which builds a variant via a simple model-to-model transformation. As its input, the service requires the unique name of the variant that will be built along with the list of models to be transformed. For each macro contained in each of the given models, the service checks whether a variant with the given unique name is assigned to it. If so, the detected variant is set as the default submodel of the macro (thus now representing the new default behavior). Otherwise the macro is not modified, leaving the default behavior unchanged. The role of this model-to-model transformation is very much comparable to the one of aspect weavers [Kic+97] in AO, except that the weaving is performed on the model level (*model weaving* [Béz+04]). After this transformation step, the generator generator translates the models, which now represent the desired variant of a code generator, to code.

The current implementation of the variant management service imposes one restriction on models in order to be suitable as variants: They have to provide the same model interface (i.e., model parameters and branches) as the default variant. This is due to the fact that there is currently no support for separately parametrizing variants – the implementation of a corresponding GUI will remove this restriction (cf. Sect. 10 for details).

A more complex example showing the application of variant management is the Java Class Generator for jABC, which will be presented in detail in Sect. 5.2. This generator offers different generation strategies, each of them realized by a corresponding variant.

4.2 Simple Example: Documentation Generator

In order to give the reader an idea of modeling a code generator with Genesys, the following sections present a complete example of a simple code generator called the “Documentation Generator”. For the most part, this

example is based on a tutorial introduction which originally has been published in [JSM10]. The Documentation Generator is designed to be used in jABC, i.e., SLGs are its source language. The generator's task is to produce an HTML documentation website (comparable to the output of Java's Javadoc Tool) from those models according to the following requirements:

1. The generator should process all models in a given directory.
2. For each model, a separate HTML page should be generated, containing the following information:
   - the documentation of the model and
   - a list of all SIB instances contained in that model. Each list entry should display the corresponding SIB's label. Furthermore, each entry should be linked to a detail page (described in 3) containing the documentation of the particular SIB instance, as well as to the corresponding online SIB documentation (e.g., [TU10]).
3. For each SIB instance in each SLG, a detail HTML page will be generated, displaying the SIB instance's documentation. This page should be linked to the corresponding model page.
4. An index page should be generated, listing all processed models along with links to their respective model pages.
5. Each generated HTML page should contain a timestamp in order to retain the time of the last generation.

Based on these requirements, the code generator is modeled by a generator developer who has to have knowledge of using jABC and Genesys, of SLGs (the source and implementation language) along with their associated concepts such as SIBs and branches, and of the HTML format (the target language). The following sections will show how to model the complete Documentation Generator, which is built almost entirely on the basis of jABC's Common SIB library. For the sake of simplicity, not all the parameterizations of the employed SIBs will be explained in detail, but instead the descriptions will focus on which SIBs are used to solve the task and how they are connected to each other. For each employed SIB, the corresponding class will be named, so that it can be easily related to the online documentation [TU10], which contains detailed information for all SIBs. If no class is given, then the name of the SIB class equals the SIB label displayed in the model.

4.2.1 Structuring the Generation Process

Modeling a code generator with Genesys usually proceeds top-down. Accordingly, as the first step of modeling the Documentation Generator, the generation process is divided into two abstract coarse-grained phases: the *initialization phase* and the *generation phase*. In the initialization phase, the code generator will set up the generation by verifying the input parameters and loading the input SLGs, and in the generation phase the actual HTML website is produced.

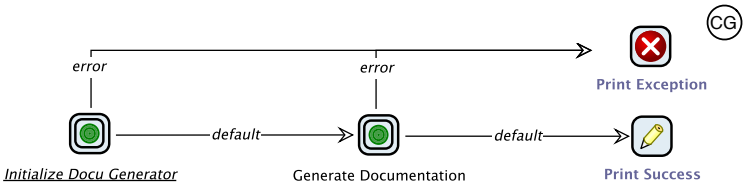

Fig. 4.4. The `Docu Generator` main model (topmost hierarchy level)

Fig. 4.4 shows the resulting model, containing the SIB `Initialize Docu Generator` for the initialization phase and `Generate Documentation` for the generation phase. Both SIBs are macros (SIB class `MacroSIB`), and both phases will be refined and concretized in the following. Please note that all models of the Documentation Generator only use `MacroSIB`s (cf. Sect. 3.3.3) as macros, i.e., a flat execution context is employed (cf. Sect. 3.3.2). Along with the two macros, the model contains two other SIBs emitting either a success message (`Print Success`, SIB class `PrintConsoleMessage`) when the two phases have been finished successfully, or an error message (`Print Exception`) if anything failed during the execution of the code generator.

Note that in this example, the error handling is in most cases delegated to the main model depicted in Fig. 4.4 (*error delegation*). All SIBs used in the Documentation Generator's models have "error" branches, most of which lead to the SIB `Print Exception`, either as direct edges in the main model, or as model branches in all the other models. Consequently, `Print Exception` is the central (though very simple) error handling step for the entire generator. Another (but more rare) way of handling errors is *in-place handling*, whereby the handling is usually performed by subsequent SIBs (or even models) in the same model. This distinction between in-place handling and error delegation is very much comparable to exception handling in Java, where a caught exception may either be handled directly or thrown again in order to be handled somewhere else.

4.2.2 The Initialization Phase

Subsequently, the initialization phase is concretized. Basically, this phase has to verify the input parameters provided by the user and to set up the generation process. The Documentation Generator will have two input parameters:

outputFolder, the absolute path to the output directory for the generated HTML files, and

modelPath, a list of absolute paths to directories containing the jABC models for which the documentation should be generated.

Fig. 4.5 shows the refined model for the initialization phase.

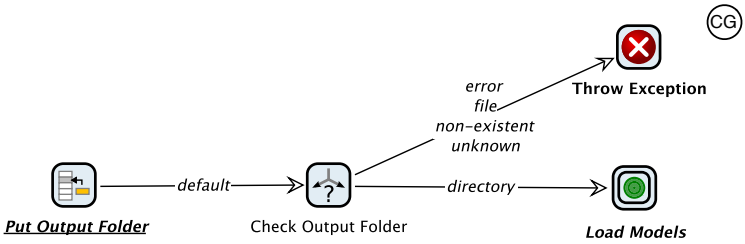

Fig. 4.5. The `Initialize Docu Generator` model (second hierarchy level)

The generator starts by processing the parameter "outputFolder", whose value is first put into the execution context (`Put Output Folder`, SIB class `PutFile`) in order to be accessible by the following SIBs. Afterwards, the SIB `Check Output Folder` (SIB class `CheckPath`) verifies this value: If it does not denote a proper (i.e., existent and writeable) directory, the following step `Throw Exception` issues an error. Otherwise, the initialization phase continues with handling the second input parameter "modelPath" (macro `Load Models`), which is again performed in a submodel.

Referring to the different error handling techniques described above, the SIB `Throw Exception` is an example for in-place handling of errors. In this case, the exit branch of a preceding step (`Check Output Folder` in Fig. 4.5) reflects an undesired result. When such an error is detected, it is directly handled by `Throw Exception`, which performs the error handling in-place at the service level, rather than delegating it to a higher model hierarchy level.

The submodel referenced by `Load Models` is displayed in Fig. 4.6. As loading SLGs is a standard task for many code generators (at least for those dealing with SLGs as their source language), this model is, among many others, contained in the Genesys framework and thus can be entirely reused for the Documentation Generator without any changes. A detailed discussion of this loading process is not required at this point: The generator developer does not have to know the technical details of loading SLGs anyway, as from his perspective, the model is used just like a ready-made service in a black box fashion.

4.2.3 The Generation Phase

For modeling the generation phase, the macro `Generate Documentation` in the main model (Fig. 4.4) is refined. Fig. 4.7 shows the resulting model. The generation process starts with producing the static header of the index page (`Generate Index Header`) using StringTemplate (see Sect. 2.4.2). Please note that all SIBs with "ST" on their icon are instances of the SIB class `RunStringTemplate` described in Sect. 3.2.1. The header of the index page only consists of static text, for instance, containing the opening `html` and `body` tags for the document. Afterwards, a time stamp is generated (`Generate Time Stamp`, SIB class `GetTimeStamp`), which is inserted into the footer of

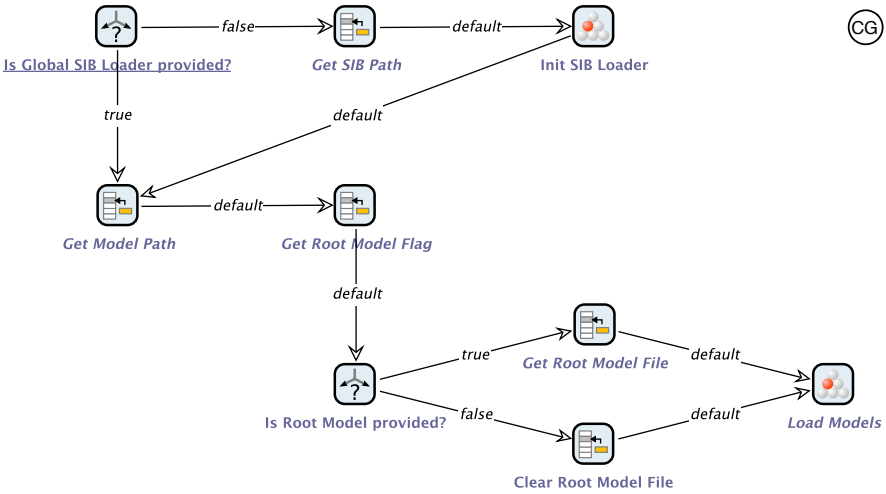

Fig. 4.6. The `Load Models` model (third hierarchy level)

each generated HTML page. The generation of the index page content and the detail pages for the models and the contained SIBs is again modeled in a submodel (referenced by the macro `Generate Model Pages`). After the detail pages are produced, the generator finalizes the index page. For this purpose, the SIB `Generate Index Footer` is parameterized with the following simple template:

```
    </ul>
    <hr>
    <i>Generated: $timeStamp$</i>
  </body>
</html>
```

Besides some closing tags, this template contains a placeholder called "timeStamp", which is enclosed by dollar signs. When the generator is executed, StringTemplate replaces this placeholder by the timestamp produced by the step `Generate Time Stamp`. The generation phase finishes with writing the index page to a file (`Write Index Page`, SIB class `WriteTextFile`).

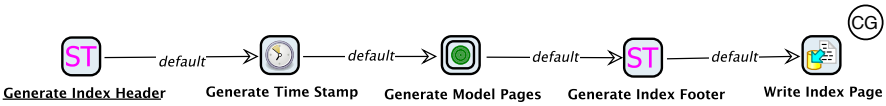

Fig. 4.7. The `Generate Documentation` model (second hierarchy level)

The submodel that refines the macro `Generate Model Pages` is depicted in Fig. 4.8. It starts by iterating over all input SLGs that have been loaded in the initialization phase (`Next Model`, SIB class `IterateElements`). Please

note that the Documentation Generator produces HTML pages for *all* SLGs in a given directory, which particularly includes all referenced submodels. Consequently, we do not need to expand the macros or to use any recursion - a simple iteration of the models is sufficient. As long as there are still models left to be processed, the "next" branch of the SIB will be used. Otherwise, the execution proceeds with the parent model (Fig. 4.7), which is connected via a model branch (i.e., the "exit" branch of `Next Model` is exported as a model branch that leads to the parent model). The following step `Update Model Counter` (SIB class `UpdateCounter`) keeps track of a model number that is incremented each time the SIB is executed. This number is required to construct the names for the model detail pages. Then the generator extracts some information from the current model. The SIB `Get Model Name` stores its name in the execution context, and the SIB `Convert Content to Html` reads annotated documentation and converts it to proper HTML markup, which is also stored in the execution context. Now the generator has collected enough information for generating an index page entry for the current model (basically the model name, linked to the model detail page, step `Generate Index Entry`) as well as the header of the model detail page containing the model's documentation and name.

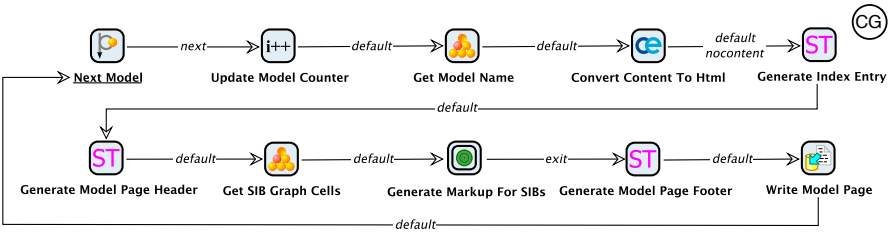

Fig. 4.8. The `Generate Model Pages` model (third hierarchy level)

In order to generate the list of SIBs in the current model along with the SIB detail pages, the generator retrieves all contained SIB instances (`Get SIB Graph Cells`) and then again delegates the production of all SIB-specific HTML markup to a submodel (macro `Generate Markup for SIBs`). Finally, the footer of the detail page is generated (`Generate Model Page Footer`) and the entire page is written to a file (`Write Model Page`).

The last model required for the Documentation Generator refines the macro `Generate Markup for SIBs` and is depicted in Fig. 4.9. In the first step, it iterates over all SIB instances contained in the current model (`Next SIB Graph Cell`, SIB class `IterateElements`), which works just like the `Next Model` step in Fig. 4.8. Furthermore, analogous to the model detail pages, the SIB `Update SIB Counter` (SIB class `UpdateCounter`) keeps track of a SIB counter that is used for the file names of the resulting SIB detail pages. Then again, some information is collected from the current SIB found

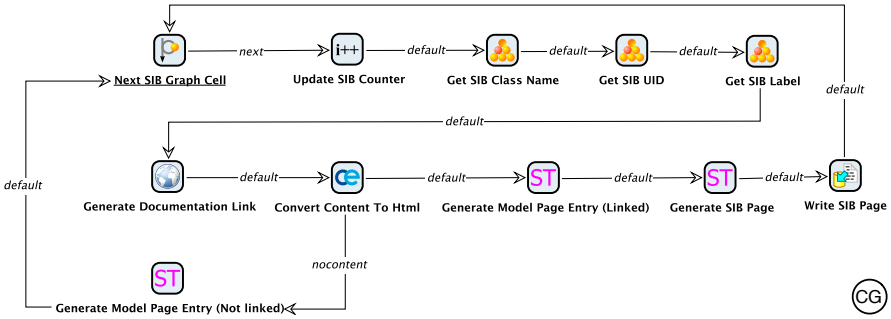

Fig. 4.9. The `Generate Markup for SIBs` model (fourth hierarchy level)

in the execution context: its class name (`Get SIB Class Name`), unique identifier (`Get SIB Class Name`) and instance label (`Get SIB Label`).

The following SIB `Generate Documentation Link` differs from the other SIBs used in the Documentation Generator, as it is the only one that calls a remote functionality, in this case a Web Service. This Web Service takes a SIB's class name and UID (cf. Sect. 3.2.1) as input and uses this information to construct the URL of the online documentation [TU10] that describes the SIB class. As such a SIB was not provided by the Common SIBs, it had to be created by implementing an appropriate service adapter (cf. Sect. 3.2.1), that realizes the communication with the already existing Web Service. This implementation was an easy one-time task.

In the following step, the code generator reads the documentation annotated to the current SIB (`Convert Content To Html`). Depending on whether such a documentation could be found, a list entry for the current SIB on the model page is generated. In case a documentation exists, this entry is linked to a SIB detail page which is generated in the step `Generate SIB Page`. This SIB is parameterized with the following template:

```
<html>
  <body>
    $sibDoc$
    <a href="model_$modelCounter$.html">back to "$modelName$"</a>
    <hr>
    <i>Generated: $timeStamp$</i>
  </body>
</html>
```

Again, the static text contains placeholders that are replaced by StringTemplate, using information collected by the code generator:

sibDoc: The current SIB's HTML documentation retrieved by the `Convert Content To Html` step in Fig. 4.9.

modelCounter: The number of the current model assigned by the SIB `Update Model Counter` in Fig. 4.8.

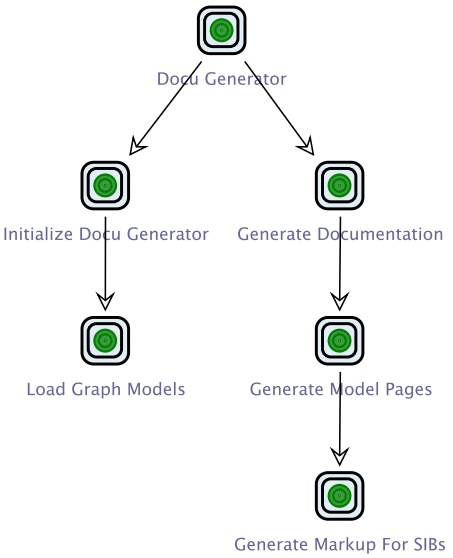

Fig. 4.10. The model hierarchy of the Documentation Generator

modelName: The name of the current model retrieved by the SIB `Get Model Name` in Fig. 4.8.

timeStamp: The time stamp produced by the SIB `Generate Time Stamp` in Fig. 4.7.

The usage of this information in the template shows how SIB instances in submodels can easily access information left in the execution context by SIB instances at arbitrary levels of the model hierarchy, which is due to the flat nature of the execution context.

Finally, if a SIB detail page has been generated, it is also written to a file (`Write SIB Page`).

4.2.4 Finalizing the Generator

In summary, the demonstrated models constitute a complete code generator according to the requirements listed above. The resulting generator consists of six models (five new, one could be reused from Genesys' model library), containing 43 instances of 23 different SIBs. Only one SIB had to be implemented, as the rest of the required functionality could be covered with existing ones. The resulting model hierarchy (see Fig. 4.10) spans four levels.

While modeling a code generator, it is possible at any time to execute, debug and test it using jABC's Tracer (Sect. 3.3). However, for productive use of the code generator, it should be translated to code itself, for instance in order to be able to use it via the Genesys jABC Plugin or to integrate it into a Maven-based tool-chain (both options will be described in more detail

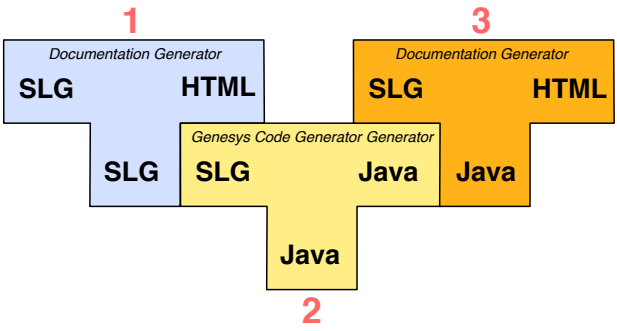

Fig. 4.11. Translating the SLGs of the Documentation Generator to Java code

in Sect. 4.3.2). This finalization of the code generator usually consists of two steps: *editing the generator's metadata* and finally *generating the generator*. The former includes metadata such as the generator's name, version, author and usage documentation, and is edited via a corresponding GUI that is part of Genesys' developer tools presented in Sect. 4.3.1.

In the second step, a corresponding generator generator is responsible for translating the generator models into code. The T-diagram depicted in Fig. 4.11 shows that for the Documentation Generator, this translation is performed by means of the *Genesys Code Generator Generator* (see Sect. 5.2.6). It translates a set of given generator models to Java code that contains all necessary information (e.g., metadata and corresponding interface implementations) to be useable with the appropriate tools named above.

4.2.5 General Remarks on the Example

The example above demonstrated how a code generator is modeled with Genesys. Several aspects of the example are particularly noteworthy as they illustrate some characteristics of the general approach. In particular, the SLGs of the Documentation Generator show that there is a strict separation between the generation logic and the output description (cf. Sect. 2.4), as demanded by *Requirement S4 - Clean Code Generator Specification*. The generation logic is given by the code generator SLGs, and the output description is given by the templates, which are parameters of dedicated SIBs such as `RunStringTemplate`.

In order to achieve this strict separation, it is not advisable to describe parts of the generation logic in the templates. This would be easily possible, as most template languages also support control structures like conditionals, loops or function calls (cf. Sect. 2.4.2). However, describing parts of the generation logic in the code generator SLG and other parts in the templates has several serious drawbacks. First, the models are more difficult to understand as the generation logic of the code generator cannot be fully grasped by just

looking at the flow of actions in the SLGs. Second, as parts of the generation logic are hidden in templates and not modeled explicitly, they are not considered by tools such as a model checker (see Sect. 3.4), thus impeding a proper verification of the code generator.

Consequently, the use of templates in Genesys is mostly restricted to those facilities of a template language that allow accessing data (e.g., placeholders or simple expressions). Employing advanced control structures is discouraged, as they most likely would be used to describe logic which should rather be modeled explicitly by means of corresponding SIBs provided by Genesys. Likewise, templates should not be misapplied for making function calls. Instead either an existing SIB realizing the function should be used or, if no such SIB exists yet, a new one should be created in order enable the reusability of the function.

Accordingly, the templates shown in the previous sections are not particularly simple examples. Instead they are characteristic of how templates are generally used in Genesys.

The Documentation Generator also exemplifies the mixed application of source-driven and target-driven transformation. The generator is target-driven as, apart from the actual templates, its generation logic roughly follows the structure of the output. For instance, the headers of the single HTML pages are always generated before the footers (see, e.g., Fig. 4.7 and 4.8). At the same time, the generator can be considered source-driven, because its generation logic avoids multiple traversals: Each model and each contained SIB is only visited once, and when processing a model or SIB, all required output is produced at once, so that no additional visit is necessary. This is, e.g., visible in the SLG shown in Fig. 4.9, which generates an entry for the model detail page as well as the SIB detail page for a particular SIB instance.

Furthermore, this flexibility of structuring the transformation allows the Documentation Generator to produce multiple output files without any problems, thus overcoming the typical inefficiencies attributed to template-based code generators (cf. Sect. 2.4.2).

4.3 Genesys Tooling

As outlined at the beginning of this chapter and depicted in Fig. 4.1, Genesys provides tools supporting generator developers as well as generator users. The following sections briefly introduce those tools.

4.3.1 Developer Tools

The developer tools provide facilities that assist the generator developer in building code generators. They are realized as jABC plugins, but are not necessarily restricted to code generators for jABC (i.e., having SLGs as their source language).

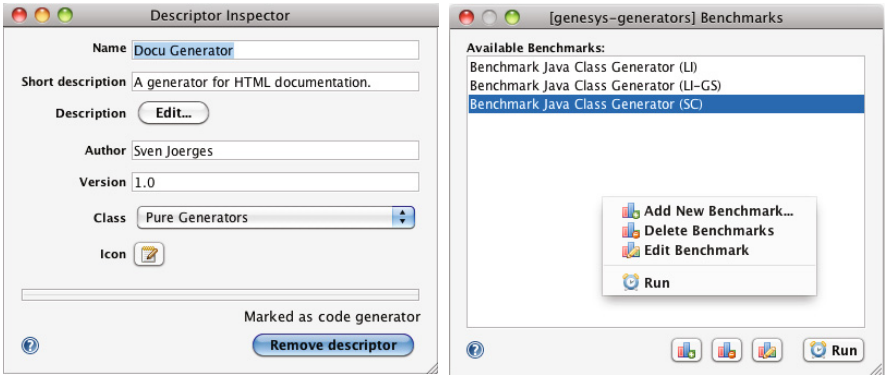

Fig. 4.12. Left hand side: Inspector for editing a code generator's metadata, Right hand side: Setting up a benchmark

Descriptor Inspector:

The *Descriptor Inspector* allows the generator developer to add metadata to a code generator. This includes information such as the name of the code generator, a short and a long description, the author's name, a version, the category of the code generator as well as an icon. This metadata is attached to the topmost model of the code generator as a user object (cf. Sect. 3.2.2) and serves multiple purposes. First, generator generators may incorporate the metadata into a generated version of the code generator and thus may allow tools using the code generator to display the information to users. For instance, when translating a code generator for jABC using the *Genesys Code Generator Generator* (see Sect. 5.2.6), any metadata is added to the resulting Java class. When using a code generator translated this way in jABC, the information is displayed to the user by means of the Genesys jABC Plugin presented in the next section. Second, the metadata may be used to automatically generate documentation (e.g., an HTML website) for the code generator. Fig. 4.12 (left hand side) shows the contents of the Descriptor Inspector for the Documentation Generator presented in Sect. 4.2.

Benchmark Framework:

By means of the *benchmark framework*, a generator developer is able to compare different code generators or generator variants in terms of the performance of their generated results. For instance, one could compare the two Servlet Generator variants exemplified in Sect. 4.1.4 in order to examine whether the addition of extra debugging code negatively influences the performance of generated results. The benchmark framework performs this comparison by

1. using each participating code generator to translate a set of input models which act as objects of investigation,

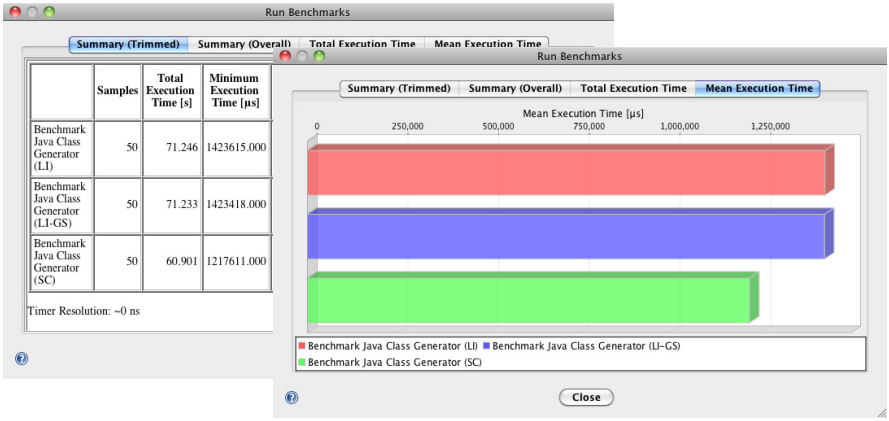

Fig. 4.13. Benchmark results visualized in tabular or bar chart form

2. executing the result of each generation (after eventually compiling it, if necessary),
3. measuring the time duration of each execution and
4. finally visualizing the measurements.

For setting up a benchmark, the generator developer first creates a configuration for each participating code generator. Such a configuration includes information such as the code generator that will be used, the input models which are translated to code as well as a so-called *execution runner* that specifies how the generated code is executed. The latter strongly depends on the code generator that is used, as, e.g., a Java class with a main method is executed in a different way than a Servlet. Technically, an execution runner is a Java class following a simple interface, which has to be implemented for every generation result that should be supported by the benchmark framework.

Furthermore, the generator developer may also specify the number of runs for a configuration. This avoids benchmarks which execute so fast that they are hardly observable, e.g., due to small input models or very high-end host machines. Another motivation for repeated executions is dealing with statistical deviations which might prevent reliable comparison of results, e.g., resulting from possible effects of memory caching, just-in-time compilation or hot-spot optimizations. Fig. 4.12 (right hand side) shows the graphical interface for setting up a benchmark, which contains prepared configurations for different variants of a code generator for Java classes (cf. Sect. 5.3).

After this preparatory configuration, the benchmark can be executed. The benchmark framework follows the procedure described above and then displays the results. As shown in Fig. 4.13, the measurements can be viewed in either tabular form or as bar charts.

Please note that although only the execution time of the generated artifacts is measured, it is nevertheless possible to similarly benchmark the execution

performance of the code generators. For setting up such a benchmark, the used code generator has to be a generator generator, that is then applied to the models of the code generators whose performance is to be measured.

VTL Editor:

As many Genesys users choose Velocity (cf. Sect. 2.4.2) as their template engine, the Developer Tools provide a textual editor for templates written in the Velocity Template Language (VTL). This editor allows in-place editing of Velocity templates in jABC with line numbering and syntax highlighting. Please note that although Velocity is specifically supported that way, Genesys is not restricted to any particular template engine.

4.3.2 User Tools

jABC Plugin:

The Genesys Plugin for jABC provides a graphical interface, depicted in Fig. 4.14, for configuring and executing code generators in order to translate SLGs to code. For an existing jABC project, a user may create an arbitrary number of code generator configurations. After selecting the desired code generator, the metadata specified by the generator developer (see Sect. 4.3.1) is displayed and the generator can be configured (e.g., its input models and output directory). The code generation can then be started via this configuration. An integrated console informs the user about the generation progress and about any errors.

Furthermore, apart from generator users in jABC, this jABC plugin is also useful for generator developers. The graphical interface allows them to conveniently generate their code generator models by creating a configuration for an appropriate generator generator. In order for a code generator to be usable with the Genesys Plugin for jABC, it has to be generated with the *Genesys Code Generator Generator*, which is described in more detail in Sect. 5.2.6.

Maven Plugin:

Apache Maven [Apa11b] is a popular and powerful tool for creating build environments. It supports, among other things, building and deploying artifacts, dependency management, automatic testing and release management. All these activities are realized as plugins. In a special XML file, the so-called Project Object Model (POM), such plugins are configured and assigned to particular phases of a build lifecycle managed by Maven.

As code generators are often integral parts of such build environments they need to be compatible with corresponding management tools (*Requirement S6 - Tool-Chain Integration*). For being able to integrate a code generator developed with Genesys into a Maven-based tool-chain, Genesys provides a

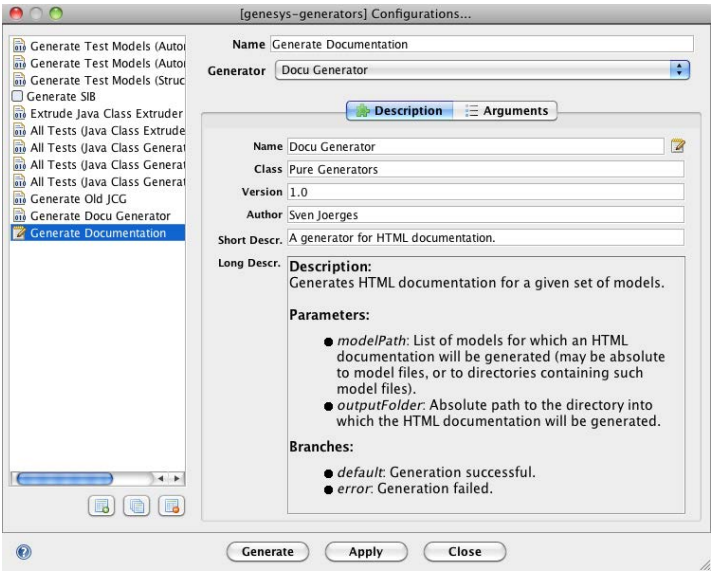

Fig. 4.14. Creating a configuration for using a code generator in jABC

corresponding Maven plugin. In analogy to the Genesys jABC plugin, this Maven plugin is able to execute any code generator that has been translated with the *Genesys Code Generator Generator*. Like any other Maven plugin, it is configured and added to a build environment via a project's POM.

As code generators produced by the *Genesys Code Generator Generator* are just plain Java classes, it is moreover easy to use them with other build management tools such as Apache Ant [Apa11a] or GNU make [Fre11a].