Editor:
Office of the CTO
Architecture Excellence

Authors:
Bernhard Gröne
Wolfram Kleis

SAP Architecture Bluebook

ABAP Enhancement and Switch Framework

Version 1.0
March 2008

History

Date	Version	Name	Change/Enhancement
2007/10/19	0.9	Andreas Fahle	Initial Version
2007/11/22	0.95	Bernhard Gröne	New structure, additional diagrams, Glossary
2007/12/18	0.96	Wolfram Kleis	Structure changed, rewritten parts on enhancement framework, added diagrams and details on enhancement frameworks, added BAdI chapter
2008/02/14	0.97	Bernhard Gröne, Wolfram Kleis	Re-worked many parts, filled glossary, clarification of terms and concepts
2008/03/05	0.98	Bernhard Gröne, Wolfram Kleis	Incorporate first review
2008/03/12	1.0	Bernhard Gröne, Wolfram Kleis	Incorporate second and third review, first version published

Acknowledgement

The authors want to thank the colleagues who contributed to this bluebook in various ways: Michael Acker, Eveline Crasovan, Torsten Kamenz, Andreas Kemmler, Roland Krämer, Jürgen Remmel, Michael Schneider, and Thomas Weiss. A special thanks goes to the student Andreas Fahle who prepared the initial version of this document.

Table of Contents

Table of Figures

1 Introduction

1.1 Scope and Target Group

The purpose of this architecture bluebook is to describe the concepts of two frameworks provided by SAP NetWeaver for ABAP development: the switch framework and the enhancement framework of NetWeaver release 7.0. The key concepts and features of these two frameworks are explained from the perspective of application development. The internal implementation of the frameworks is not in the scope of this document.

Target group are SAP internal readers who need an introduction to the concepts of the ABAP switch and enhancement frameworks.

1.2 Overview

1.2.1 Enhancing SAP Products

SAP delivers standard products that have to be adapted to the individual needs of specific industries or of single customers. Therefore SAP provides several adaptation possibilities that can be used by customers, partners or by in-house development.

In general, there are three typical means to adapt SAP applications:

- Configuration: Set parameters that change the behavior of the product.

  In some cases, the required flexibility can be achieved with configuration and with declarative approaches, but in many cases it is still required to add new code or to replace existing parts with own implementations.

- Modification: Directly change the development objects (such as program sources) delivered by SAP

  This approach has many disadvantages for software lifecycle management and is not recommended by SAP. The challenge is to keep the system working with all adaptations after an upgrade, that is after SAP has delivered a newer version of the product. This leads to severe problems with modifications. Adapting a standard product without modifications lowers the TCO of a customer and the cost of industry-specific development at SAP as well.

- Extension: Extend the application by developing new parts

  There are two categories of extensions: External extensions use interfaces (such as enterprise services) to build on top of the business platform provided by SAP. Internal extensions enhance the applications internally for example by adding fields to structures or by plugging in new behavior or changing behavior inside the application. These internal extensions are called enhancements.

Typically the same product is enhanced by different parties along the chain “SAP standard release – SAP enhancement package - SAP industry solution – Partner or Customer”. This requires an hierarchical (multilayer) approach for enhancements.

The enhancement framework offers a unified approach for different enhancement technologies that exist for ABAP-based applications. An enhancement technology allows to change or extend the behavior of SAP applications without code modifications.[1] The enhancement framework supports defining and organizing enhancement possibilities as well as implementing enhancements. With the enhancement framework,

[1] Nevertheless, after an upgrade of the standard there is still often the need for adjustments to make the enhancements work again with the changed standard.

extensions of standard applications can be made in an organized way with good integration into SAP development tools and software lifecycle management. The enhancement framework is explained in chapter 1.

1.2.2 Switching Features In SAP Applications

An important feature of the enhancement framework is the connection with the switch framework: With switches, enhancements can be installed in a system in an inoperative mode, without affecting the system behavior. Only when the switches are turned on, the corresponding enhancements become effective. This approach can be used to import mutually exclusive enhancements into the same system, as long as only one is switched on afterwards. Examples are industry solutions that are developed as enhancements of the core application. Typically only one industry solution can be effective in the same system, but with the switch technology they can be installed into the same system in an inoperative state.

Another feature of the switch and enhancement framework is the good performance, since in most cases enhancements and switching are handled by the ABAP compiler that generates corresponding code and thus avoids interpretation of meta data at runtime.

The switch framework is also used to implement the concept of non-disruptive innovation in the SAP Business Suite: New functions and features are delivered as enhancements that are initially switched off. These enhancements can be installed by the customer without any risk for existing functionality and processes. Customers can then selectively switch on the functionality they actually need. Here the enhancement framework and the switch framework work hand in hand to support this new strategy for software delivery. The switch framework is covered in chapter 3.

2 The Enhancement Framework

2.1 Overview

Experience shows that modifications are not a recommendable option for adapting standard applications. When the standard software is upgraded to a new version, it is extremely hard to maintain the modifications. The modifications done to implement some feature may be scattered across many development objects, some of which may be affected by the upgrade in different ways. Adjusting all the modifications and establishing a consistent system is a difficult and costly effort. This may even keep customers from upgrading their products at all. Therefore direct code modifications are not recommended.

ABAP user exits and their object-oriented successors, the BAdIs (Business Add-Ins), as well as enhancement points and sections in ABAP code provide predefined sockets into which extensions can be plugged. With this approach, the original objects and the enhancements are different repository objects, so upgrades of the original objects will not lead to clashes on development object level.[2]

However, with these concepts enhancements can only be done in a predefined way and in predefined locations. Enhancements that were not planned by the developer of the original objects are not possible. To solve this problem, the concept of implicit enhancement options was introduced. Implicit enhancement options need not be defined by the original developer. While BAdIs and user exits require call statements in the code, implicit enhancements can be injected without an explicitly defined call. Examples are field definitions or method definitions that can be added to an existing ABAP Objects class, or adding pre- and post methods that are executed before or after an existing method.

Compatibility of released enhancement possibilities is an important goal, but it is not always achieved in reality. Therefore, a technology is needed that helps to manage the incompatibilities and to fix the resulting problems. To avoid multiple efforts for SAP and customers, a unified approach to enhancements and their lifecycle is required for all enhancement technologies. Definitions of enhancement options and their implementations must be described, stored, propagated (by source code transports) and delivered in a uniform way. Furthermore, a common way is required to detect and resolve problems after an import or an upgrade.

Having one unified framework for different enhancement technologies also allows using a common set of tools for defining, browsing, and implementing enhancements. This saves effort for both the provider and the implementer of enhancement possibilities. With a common framework, application developers do not have the need to invent their own mechanism to define, register, and execute extensions. This way the framework helps to improve conceptual consistency of the application architecture.

With the enhancement framework, SAP addresses the requirements we derived above:

- A unified framework for various enhancement technologies such as BAdIs and source code enhancements. The framework is open and it can be extended by new enhancement technologies.
- Life cycle support for managing incompatibilities after imports or upgrades

Enhancements can be switched. This is useful for handling multiple industries, for implementing the non-disruptive innovation strategy of SAP Business Suite, and – to some extent – for testing a system with and without enhancements. Here the enhancement framework is connected with the switch framework, which is explained in chapter 3.

[2] Nevertheless, syntax errors or dependency problems with other development objects are still possible. A set of tools (for example, transaction SPAU_ENH) and life cycle support as described in chapter 2.2.4 help to resolve these problems.

2.2 Enhancement Framework Concepts

2.2.1 Classification of Enhancement Possibilities

SAP provides several methods to enhance ABAP programs without modifying existing code. These methods have been introduced in different versions of the platform, and they use different technology and meta data. The enhancement framework provides a unified handling of the different enhancement technologies. In the enhancement framework, each location where an application can be enhanced is called **enhancement option**. Enhancement options either have to be defined explicitly by the developers of the original code, or they are implicitly provided (for example at specific locations in ABAP Objects classes). Figure 2-1 gives an overview of the various enhancement options provided by SAP.

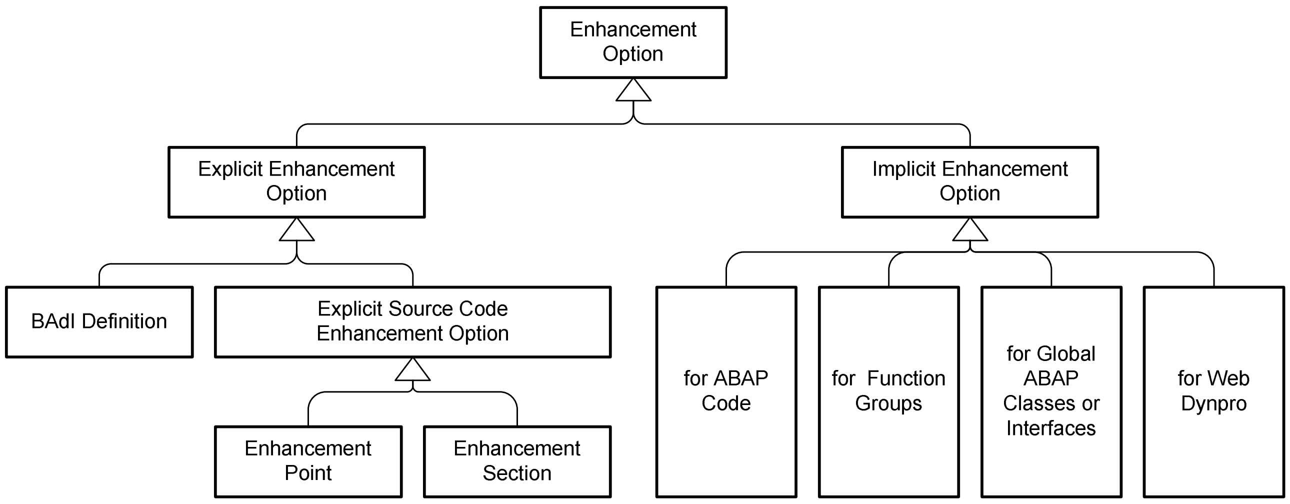
Enhancement Option
Explicit Enhancement Option
Implicit Enhancement Option
BAdI Definition
Explicit Source Code Enhancement Option
Enhancement Point
Enhancement Section
for ABAP Code
for Function Groups
for Global ABAP Classes or Interfaces
for Web Dynpro

Figure 2-1 Classification of enhancement options

2.2.1.1 Explicit Enhancement Options

Explicit enhancement options are like sockets that were defined by the application developer. Into these sockets the extensions can be plugged-in. The extensions can replace or extend standard behavior at application-specific points. Explicit enhancement options are defined with metadata and documentation that define and describe how they are to be used. The documentation can be used to describe the intention of the enhancement option and the semantic contract between application and enhancement, for example when it is called, what it is supposed to do, what the rules for enhancement providers are, or which semantic pre and post conditions of the enhancement calls exist.

The recommended technology for explicit enhancement options is SAP's object plug-in technology, the **Business Add-In (BAdI)**. BAdIs allow controlled and encapsulated extensions based on well-defined object-oriented interfaces. BAdIs are covered in detail in section 2.3.

In addition to BAdIs there are the enhancement points and enhancement options, which are low level source code plug-in mechanisms. Source code plug-ins should not be used for new development. Source code plug-ins are typically used as a fast and lightweight approach in situations where an alternative for existing source code modifications is needed, but an object oriented design with BAdIs would cause too much effort.

With source code plug-ins, the enhancement implementation is injected directly into the surrounding code at generation time. There is no isolation or encapsulation between the implementation of a source code plug-in and the surrounding code. When a source code plug-in is located in an ABAP objects method, the implementation has access to local variables defined in that method as well as to private fields and methods of the class. This establishes a tight coupling without a clear interface. With a source code plug-in in a method, all internals of a class become part of the contract with the extension provider. Compatibility is hard to achieve in this situation: Removing a local variable or a private field may break the extension. Because of these problems, object oriented BAdIs are to be preferred over enhancement points and sections whenever possible.

Enhancement points are locations in the application code where custom code can be injected. They are defined with a special ABAP statement. An enhancement point is implemented by writing ABAP code which is locally (but not physically) inserted into the original code at the position of the enhancement point. The implementation code is stored in separate repository objects, but at compilation time it is inserted into the application code. The implementation is compiled and executed in the scope of the surrounding code (similar to a text include). Multiple implementations may exist for the same enhancement point. They are included at the enhancement point in a sequence chosen by the framework.

An **enhancement section** is similar to an enhancement point, but it provides a default implementation that can be replaced by custom code. For an enhancement section only one implementation can be effective in a system.

2.2.1.2 Static and Dynamic Source Code Plug-ins

With the switch framework, implementation of enhancements can be switched. In SAP internal systems the switching can be done separately for each client. Client-dependent switching is internally implemented by generating code that encloses the implementation into an `if` statement. This leads to problems if the enhancement implementation contains declarations that cannot be put inside if statements (for example the definition of a FORM). Therefore, for both enhancement point and section, you have to specify whether their implementation may contain declarations, that is whether they are static.[3] **Static source code plug-ins** may contain declarations, while **dynamic source code-plug-ins** only contain executable code. Static source code plug-ins should be used with care for two reasons: changing or overwriting declarations may lead to errors in code that uses the original declarations and static enhancements cannot be switched in a client dependent way inside SAP[4].

2.2.1.3 Implicit Enhancement Options

Implicit enhancement options are automatically available, they do not have to be defined by the developers of the original code. They are built into the language and are available without further declarations. The following enhancements can be done via implicit enhancements options.

- New data fields can be added to existing structures in the definition of data, types or constants.
- Arbitrary code can be injected at specific locations, such as start or end of a procedure (form subroutine, function module or method).
- Function Group Enhancements allow adding parameters to the interface of function modules.
- ABAP objects classes or interfaces can be enhanced by adding new fields or methods. Methods can also be enhanced with parameters, and pre or post methods that are executed before or after the original method. It is also possible to enhance an existing class by replacing an existing method implementation with a new one (overwrite). For class extensions see also 2.6.2.

Implicit enhancement options have the advantage that enhancements are possible without having to be planned by SAP. This offers great flexibility, but also has a disadvantage: SAP cannot be expected to guarantee that implementations of implicit enhancement options will still work in future versions of the product – otherwise SAP development would be paralyzed: Almost no change would be allowed any more

[3] This will become obsolete when the client-specific load is introduced in the ABAP kernel. This feature allows to use different compiled code (load) for different clients in a system and then the generated IF statement will not be required anymore. Note also that with the current implementation, client-dependent switching is not available for customers (can be used only internally at SAP).

[4] Therefore, enhancement sections declared as “static” lead to prio 1 errors in the extended syntax check (checkman).

because each data class, function, form etc may have been extended. Therefore SAP does not assure any compatibility for implicit enhancement options: Implementing an implicit enhancement option follows the same rules as modifications: Stability and compatibility is not guaranteed by SAP and there is no support for problems related to implementations of implicit enhancement options after an upgrade.

2.2.1.4 Other Concepts For Extending SAP Applications

The ABAP extension framework currently covers mechanisms that are related to ABAP programming, i.e. the definition of program elements, interfaces and the execution of code. There are further options for adaptation or extension options that cover other development entities, such as Dynpro, DDIC structures or IMG structures. They are not handled by the enhancement framework, but can be switched with the switch framework. Section 3.2.2 lists these entities.

Another type of explicit enhancements is the customer exit, which should not be used in new development since R/3 release 4.6. There are different types of exists, such as screen exits for Dynpros (by including a subscreen), menu exits to add entries to the application menu, and function module exits which are called via `CALL CUSTOMER-FUNCTION '001'`. In the Sales application, the so-called user exits were offered, which are simple subroutine calls to an include that the customer can modify, but which is located inside the application's module pool.

2.2.2 Classification of Enhancement Related Activities

In the life cycle of an explicit enhancement option, there are three different activities:

- **Definition of the plug-in types**

  Enhancement possibilities can be described in analogy to an electronic circuit board that contains sockets for exchangeable devices or slots for plugging extensions cards into a communication bus. The first activity is to specify the bus protocol or the socket type. In the extension framework this maps for example to the definition of a new BAdI (as explained in section 2.3, BAdIs are object oriented plug-ins, called by standard applications and implemented by industry solutions, partners, customers, and so on). A BAdI definition defines a type of extension possibilities. This definition specifies the name of the BAdI, the object-oriented interface by which BAdI instances can be called, and also filter parameters that allow to select the appropriate implementation from a set of alternatives. The BAdI definition alone describes only a plug-in type.

- **Definition of the locations where plug-ins are called**

  To really offer the ability to extend the application, the plug-in type must be instantiated and called in the standard application. In our analogy with sockets and plugs, the creation of the application code that instantiates and calls the plug-in corresponds to the installation of different sockets of a certain type on a circuit board. Defining of the plug-in type and writing the code that instantiates and calls it in the application are the tasks of the application provider, for example SAP core application development.

- **Implementation of Plug-Ins**

  To extend the application, the partner or customer has to provide an implementation for the plug-in definition. Implementing a plug-in corresponds to constructing a device that fits into a certain type of socket. The execution of the BAdI instantiation and the BAdI call at runtime finally corresponds to actually putting a specific instance of a device into a socket of the corresponding socket type.

Please note that the definition of the types and the call in the applications are done by the provider of the original application. The implementation of plug-ins is done by the party who provides the actual extension.

2.2.3 Organizing Enhancements and their Implementations

This section introduces the most important concepts and the terminology that are used in the enhancement framework to manage explicit enhancement options and their implementation. The abstractions and their relationships are shown in Figure 2-2. The light grey boxes indicate which role is responsible for which items. The dark grey boxes indicate during which of the phases shown in 2.2.2 the different items are defined.

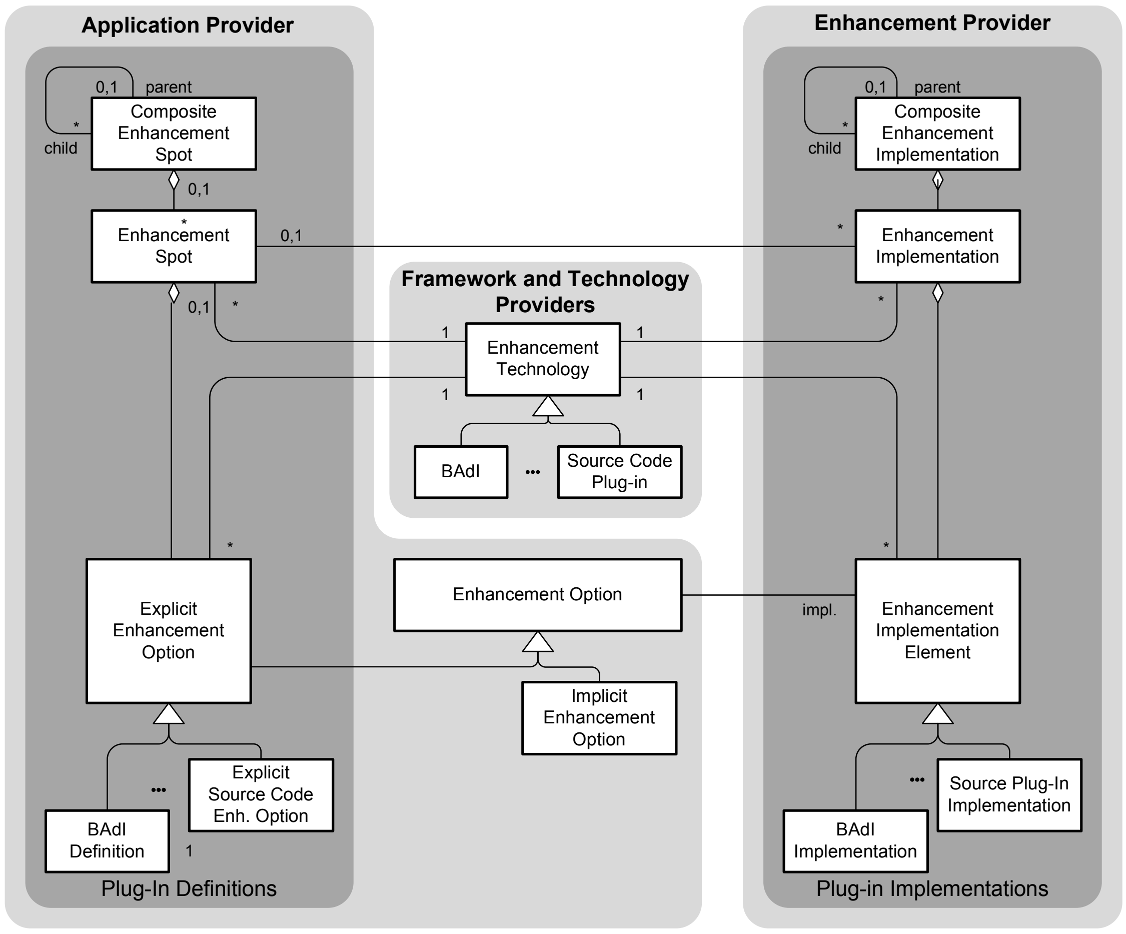
Application Provider
Enhancement Provider
Framework and Technology Providers
Composite Enhancement Spot
Enhancement Spot
Composite Enhancement Implementation
Enhancement Implementation
Enhancement Technology
BAdI
Source Code Plug-in
Enhancement Option
Explicit Enhancement Option
Implicit Enhancement Option
Enhancement Implementation Element
BAdI Definition
Explicit Source Code Enh. Option
BAdI Implementation
Source Plug-In Implementation
Plug-In Definitions
Plug-in Implementations
parent
child
impl.

Figure 2-2 Organizing Enhancement Spots and Implementations

Enhancing a certain aspect of an application may require a number of changes in different locations in the application. Replacing a business strategy with an own implementation may for example require the definition of several plug-in types, for example several BAdIs. The enhancement framework allows to bundle these definitions by assigning them to the same enhancement spot. An **enhancement spot** is a collection of semantically related explicit enhancement options (BAdI definition, enhancement section or enhancement point). Enhancement spots are generic concepts that are independent of the enhancement technology. However, there are two restrictions:

- All enhancement options that are assigned to one enhancement spot must be of the same enhancement technology (for example only BAdIs)
- All explicit source code enhancement options (enhancement points and sections) that are assigned to one enhancement spot must be located in the same development object (for example in the same program source)

Note that there is no separate definition for source code enhancement options such as enhancement points and enhancement sections. Unlike BAdIs which are defined once and then called multiple times, enhancement points and enhancement sections are defined and called at the same time: When an enhancement point is defined in a line of code, this line defines and “calls” the enhancement element at the same time.

The implementations of the different enhancement possibilities are organized in a similar way like the enhancement options: To implement an enhancement option (such as a BAdI definition), an **enhancement**

implementation element has to be created. Like the enhancement options, the enhancement implementation elements are grouped into containers for better organization and overview in the development environment. The containers for enhancement implementation elements are called **enhancement implementations**. An enhancement implementation is assigned to one enhancement spot (or none, as explained below). An enhancement implementation which is assigned to an enhancement spot contains only implementation elements for the elements of that spot. The enhancement implementation need not provide implementation elements for all elements of the corresponding enhancement spot. It also cannot bundle implementations of elements from different enhancement spots. Implicit enhancement options are not assigned to enhancement spots. Therefore the implementations and implementation elements for implicit enhancement options are not referring to enhancement spots or spot elements (this is indicated by “n to zero-or-one” relationships in Figure 2-2). Like the enhancement options, the enhancement implementations contain only elements of the one enhancement technology, for example only BAdI implementations.

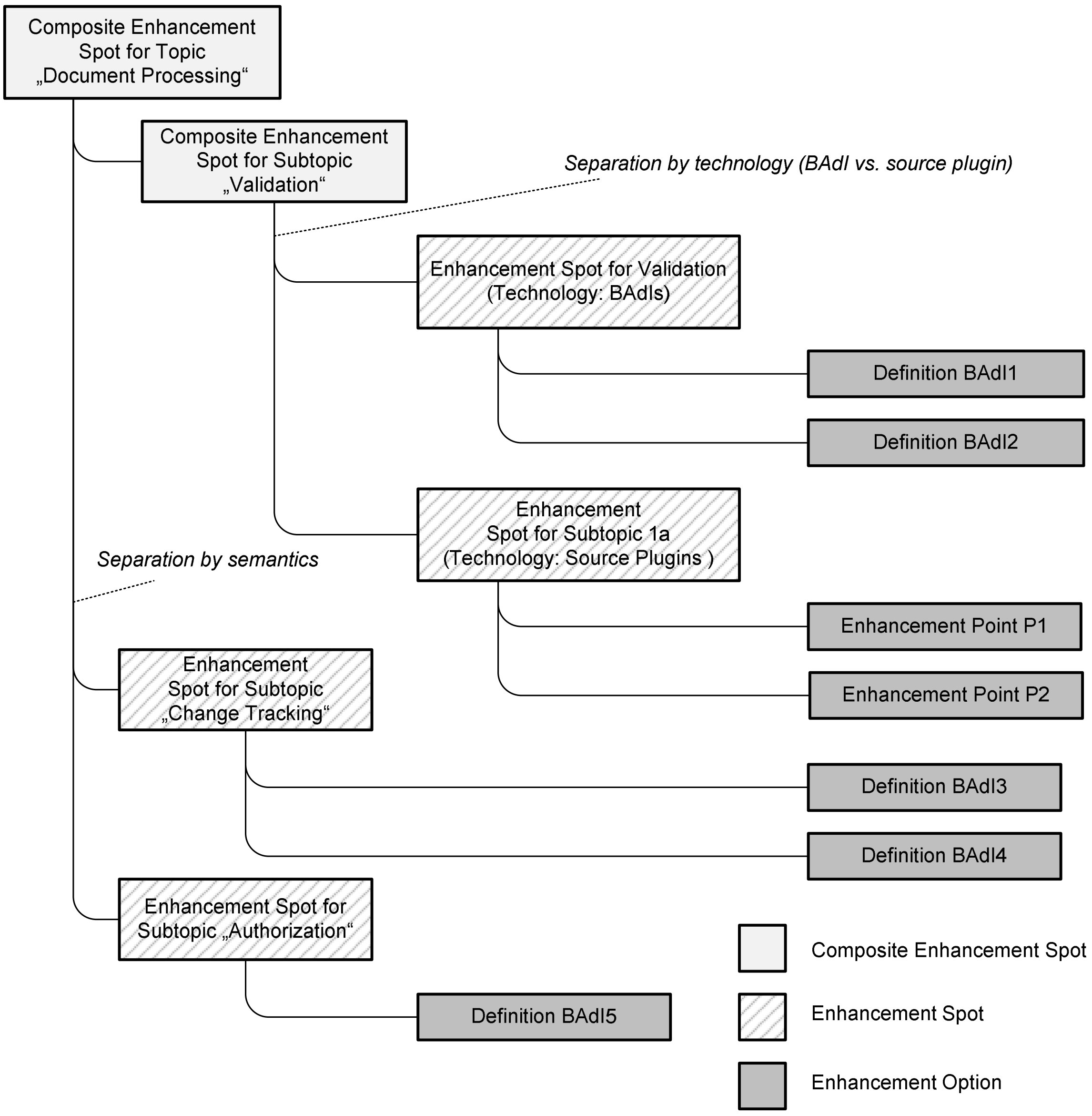
Composite Enhancement Spot for Topic „Document Processing“
Composite Enhancement Spot for Subtopic „Validation“
Separation by technology (BAdI vs. source plugin)
Enhancement Spot for Validation (Technology: BAdIs)
Definition BAdI1
Definition BAdI2
Enhancement Spot for Subtopic 1a (Technology: Source Plugins)
Separation by semantics
Enhancement Point P1
Enhancement Point P2
Enhancement Spot for Subtopic „Change Tracking“
Definition BAdI3
Definition BAdI4
Enhancement Spot for Subtopic „Authorization“
Definition BAdI5
Composite Enhancement Spot
Enhancement Spot
Enhancement Option

Figure 2-3 Enhancement Options Example

In big enterprise applications one hierarchy level is not enough for organizing enhancement possibilities. The enhancement framework allows to organize enhancement spots in hierarchies of composite enhancement spots. **Composite enhancement spots** may contain simple enhancement spots or recursively other composite enhancement spots. A composite enhancement spot may contain simple enhancement spots for different enhancement technologies. The example in Figure 2-3 illustrates the hierarchical grouping of enhancement spot elements.

Implicit enhancement options exist automatically for every class or function module without further definition. There is nothing to be grouped in enhancements spots or elements on the definition side. Therefore there are no enhancements spots or element definitions for implicit enhancement options. On the implementation side, however, there is no difference between implementations of explicit and implicit options: In both cases the implementation elements are organized in enhancement implementations that can be organized in hierarchies of composite implementations.

Figure 2-4 is an extension of Figure 2-1: It shows the categories of enhancement options with additional information about the different types of implementation elements.

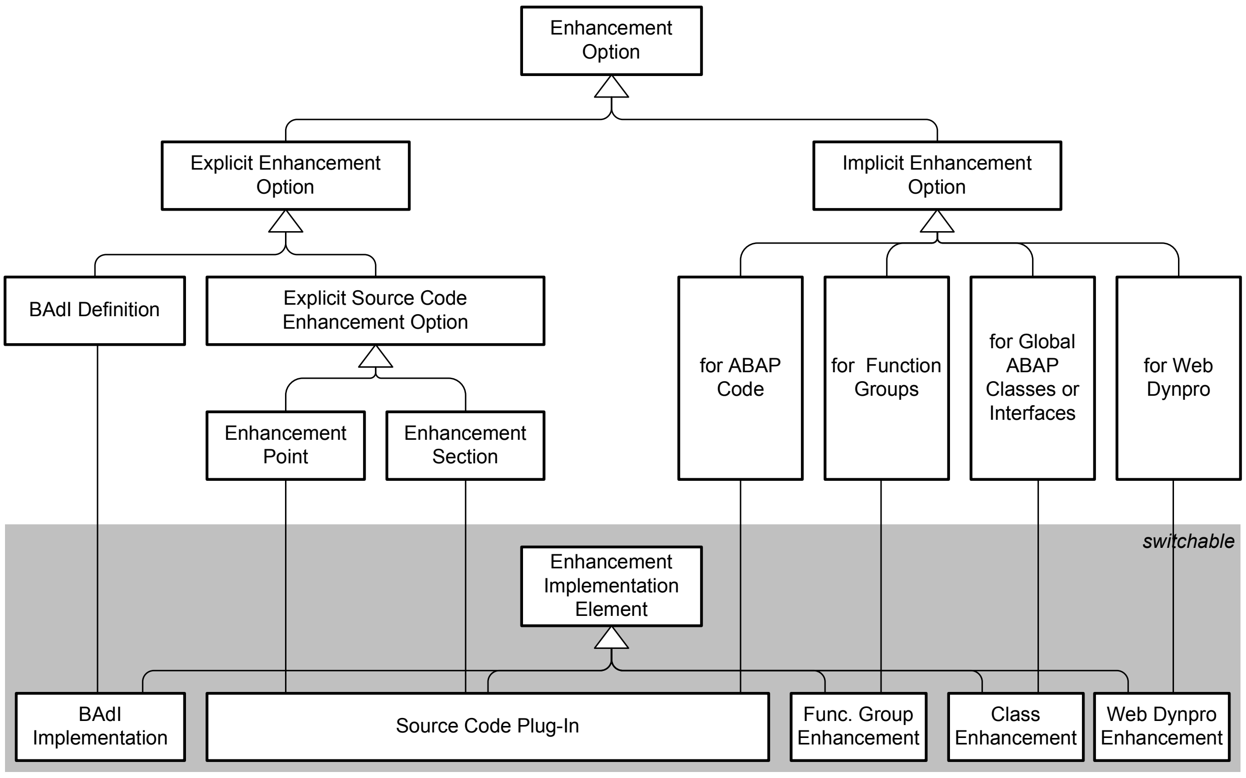

Figure 2-4 Categories of Enhancement Options and Implementations

2.2.4 Life Cycle Support for Enhancements: Conflict Detection and Contracts

Enhancement spots, composite enhancement spots, as well as enhancement implementations and composite enhancement implementations are development objects that are managed by the ABAP change and transport system. This ensures that the metadata describing the hierarchy of spots and their contained elements, the element attributes and the association between definitions and implementing objects is not lost when the software is delivered into some other system.

The enhancement options (or the objects that can be enhanced via implicit options) and the enhancement implementations are created and maintained by different parties in different locations (for example SAP core application release → SAP core application enhancements → SAP industry, SAP → partner, SAP → customer, Partner → Customer etc). In such a distributed development situation it cannot always be prevented that conflicts or compatibility issues arise between new versions of the enhancement option and the enhancement implementation. Here are some examples:

- A version of a BAdI interface is imported into a system where implementations exist that were based on a different version of the BAdI definition. Problems arise for example if the new version of the BAdI interface contains new methods that are not implemented by the current implementation.
- A source code enhancement exists inside a function module (or method), and now a changed version of the development object is imported. In this case there is a notification that the environment of the enhancement has changed. Based on the notification, the developer of the enhancement implementation has to review the enhanced code to make sure that the enhancement still makes sense or to test the enhancement again.

SAP tries to keep typed enhancement options, such as BAdIs, compatible, but there may always be exceptions with good reasons for incompatible changes. Furthermore, partners or customers may use the enhancement framework to define their own enhancement options, and they may be less restrictive with respect to changes. In addition, there are the source code enhancements that have no defined interface, and the implicit options that were not introduced by the developers and for which therefore no specific documentation exists. Here any change in the environment of the enhancement may affect the correctness of the enhancement. Therefore formal and semantic conflicts cannot be completely avoided for this type of enhancements. It is required to accept the fact that conflicts will arise, and that you need to manage them as good as possible.

All problems described above follow the same pattern: Due to a life cycle operation (upgrade, transport), an enhancement implementation is now in an environment that is different from the one where the implementation was created. To detect this type of situation, the enhancement framework has to know the relevant pieces of the environment that existed when the enhancement implementation was developed. This information is called the **contract.** In this technical context, the contract is a snapshot of the enhanced object at the time when the enhancement implementation was activated. The contract consists of different pieces of information for different enhancement technologies: For BAdIs, it is the metadata of the BAdI definition and the associated ABAP object interface. For source code enhancements, it is a copy of the enhanced source code object. The enhancement framework stores this information with the enhancement implementation as the contract against which it was developed. Inside a development landscape, the contract information is transported together with the enhancement implementation. This allows to detect conflicts even in a system where neither the enhancement nor the implementation were developed.

To be able to check for lifecycle problems, the enhancement framework has implemented its own import post processing which is executed after each import (during upgrade or when transporting inside a landscape). When an enhancement-relevant object is imported (such as enhancement options with implementation in the system, implicitly enhanced objects), the import post processing checks for conflicts or inconsistencies. If problems are found during an upgrade, the problem is listed in the enhancement adjustment (transaction SPAU_ENH) and the developer is directed to the tool that assists with fixing the problem. If the problem occurs during a normal transport inside a landscape, warnings are displayed in the transport log with the recommendation to start transaction SPAU_ENH manually.

2.2.5 Enhancement Technologies and Framework Extensions

The enhancement framework supports different enhancement concepts (BAdI, class extension, source code enhancement etc, see 2.2.1). In the terminology of the enhancement framework these are called enhancement technologies.

Inside the enhancement framework there is a plug-in mechanism that allows extending the framework itself; this allows to add support for additional enhancement technologies easily. This would not be done by customers or ISVs, but by SAP development teams who invent a new programming model or development object. In order to support a new enhancement technology, a new **technology plug-in** has to be implemented. A technology plug-in manages technology-specific data, for example, it offers a user interface for the where-used-list to resolve conflicts after a transport or upgrade, and it communicates with the enhancement registry.

At design time and at runtime, the enhancement framework keeps a registry with metadata about the existing enhancement spots and their elements, and of the existing implementations. This registry is constructed in a modular and extensible way: There is one **enhancement registry pool** which contains several type specific **enhancement registries** for the different enhancement technologies such as BAdI, source code plug-in etc.

2.3 BAdIs: Object Oriented Enhancements With Business Add-Ins

2.3.1 The Business Add-In Concept

The Business Add-In (BAdI) concept is SAP's object-oriented plug-in concept for ABAP[5]. BAdIs are a mechanism for planned extensibility. Planned means that the developer of the standard software already anticipates that others may want to change or enhance the standard behavior at certain points in the application. BAdIs are used to plug in custom behavior either in an additive way or by replacing the standard behavior. The purpose can be to implement a custom variant of some calculation or to override a default strategy.

2.3.2 Sockets versus Buses: Single Use and Multiple Use BAdIs

The socket / plug analogy fits only for one type of BAdI: the **single-use BAdI**. A single-use BAdI really corresponds to a socket on an electronic circuit board into which some device must be plugged to get a functional system. For a single use BAdI there must be exactly one effective implementation in the system. If there is none or more than one effective implementation, a runtime error occurs. To avoid problems in cases when no implementation exists, a fallback implementation can be shipped with the application. This default implementation is only used when no other implementation is effective.

The single-use BAdI type is suitable for situations where the computation of some business calculation with a single result has been deferred to the enhancement implementation provider. As there can be only one result, there can be only one implementation. The different implementations of such a BAdI are mutually exclusive alternatives for an algorithm, strategy or calculation of which only one can be effective at the same time.

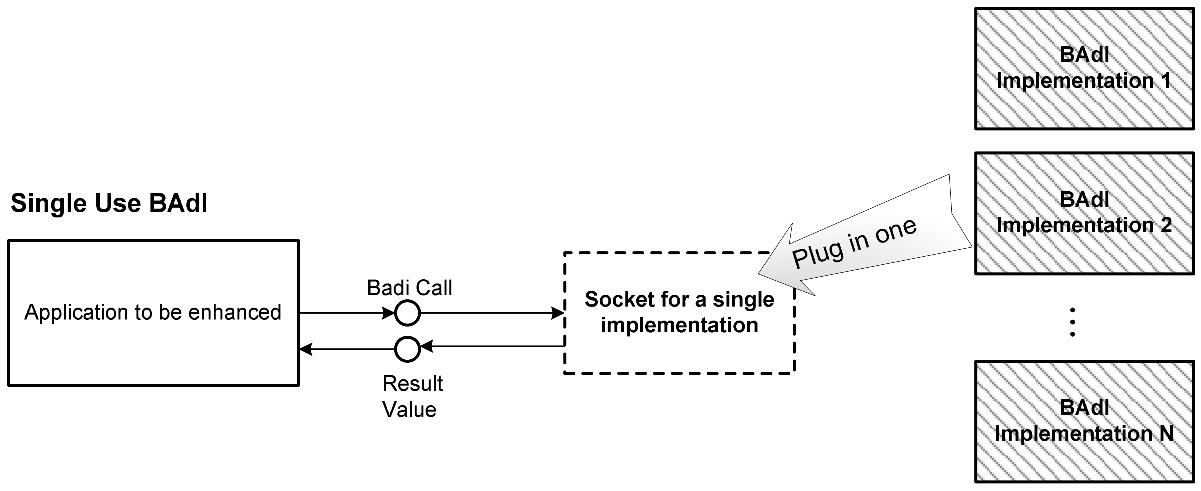

Figure 2-5 Single-use BAdI

[5] In this document we cover what is called the "new" BAdI. The older "classic BAdI" is obsolete and not explained in this document. Please refer to http://help.sap.com/saphelp_nw70/helpdata/en/e6/d54d3c596f0b26e10000000a11402f/frameset.htm for details.

In addition there is also a type of BAdI that rather corresponds to a bus or a broadcast mechanism, such as event notification.[6] This is the **multiple-use BAdI**. For this type of BAdI, multiple implementations may exist in parallel and it is also allowed to have no implementation at all. With multiple implementations in place there cannot be a single return value; therefore the ABAP Objects methods of the interface of a multi use BAdI must not have "returning" or "exporting" parameters. This type of BAdI is not used to compute results; instead, it allows to execute one or more activities at a certain point in time. This can be used to implement an event mechanism that allows multiple implementers to specify their individual reaction to some event. The difference between the two types of BAdIs is depicted in the block diagrams figure 2-5 and figure 2-6.

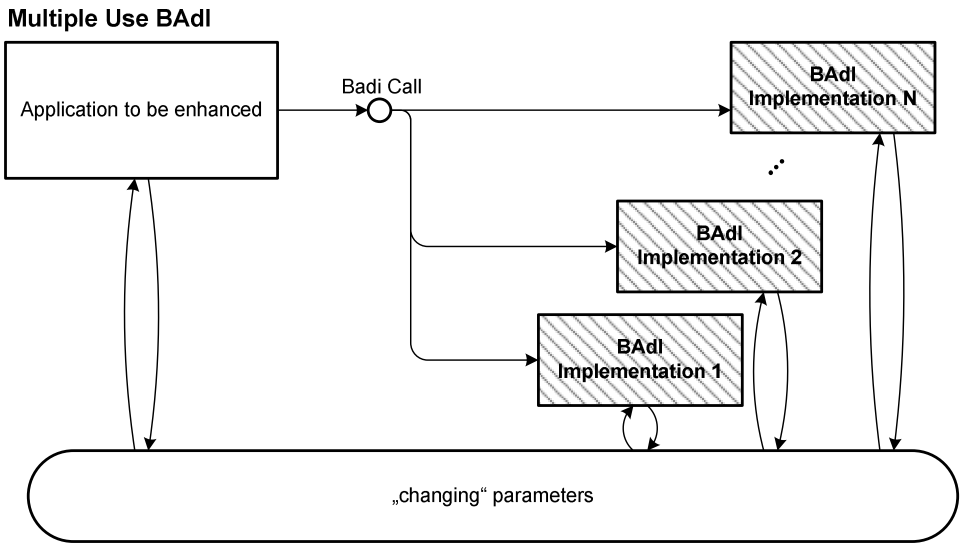

Figure 2-6 Multiple-use BAdI

2.3.3 Selecting a BAdI Implementation by Specifying Filters

To support parameterized instantiation, a BAdI definition may specify a **filter**. Those filters can be used, at instantiation time, to select the matching implementation from a set of alternatives. Different filters can be defined for different selection criteria. For example: A BAdI that allows the implementation of a certain calculation might have a filter that defines the requested precision category of the calculation (such as "high", "medium", "low"). Another BAdI implementation may be different depending on country or size of business. Corresponding filters for country and size could be used to choose the right implementation.

With filters, the restriction of single-use BAdIs has to be refined: For a single-use BAdI, exactly one effective implementation must exist at instantiation time *which matches the filter condition*.

2.3.4 BAdI Instantiation Mode

BAdI implementations are objects that may contain state. Therefore it is important whether subsequent similar requests for BAdI instances return the same or different instances of the BAdI implementation. For the instantiation of BAdIs, three different modes are supported:

Stateless: Whenever a BAdI instance is requested, a new instance is created by the framework.

[6] Multiple-use BAdIs offer some advanced features compared to ABAP events. One is the filter concept that allows a selection on caller side (see 2.3.3), another is a tool to manage possible incompatibilities after an upgrade (transaction SPAU_ENH).

Singleton: The framework creates at most one instance for each BAdI implementation. Whenever a BAdI of a particular type is requested, the framework returns the same instance.

Context Based: When the application asks for a BAdI instance, it uses an optional parameter which tells the framework which existing instance shall be returned or whether a new instance shall be created. This additional parameter identifies a **context**. Each context may contain one instance of each BAdI implementation class, but different contexts will contain different instances.

When the application asks for a BAdI instance the first time using a specific context C1, the enhancement framework creates the required implementation instances for that context. When the application asks again for an instance of the same BAdI for context C1, the already existing instances are used. This is useful in case the BAdI instances have an internal state, and the application needs to access the same instances with their already existing state information.

Technically, the context is implemented as an instance of an interface defined by the enhancement framework. The context object contains a collection (inherited from the framework interface) which contains references to the implementation objects already created for that context. Conceptually this can be seen as a distributed implementation of a mapping from contexts to the list of already created BAdI instances.

2.3.5 BAdI Instantiation and Call: Conceptual Model

The block diagram in Figure 2-7 and the corresponding activity diagram in Figure 2-8 show a conceptual model of an application that can be enhanced by implementing a BAdI.

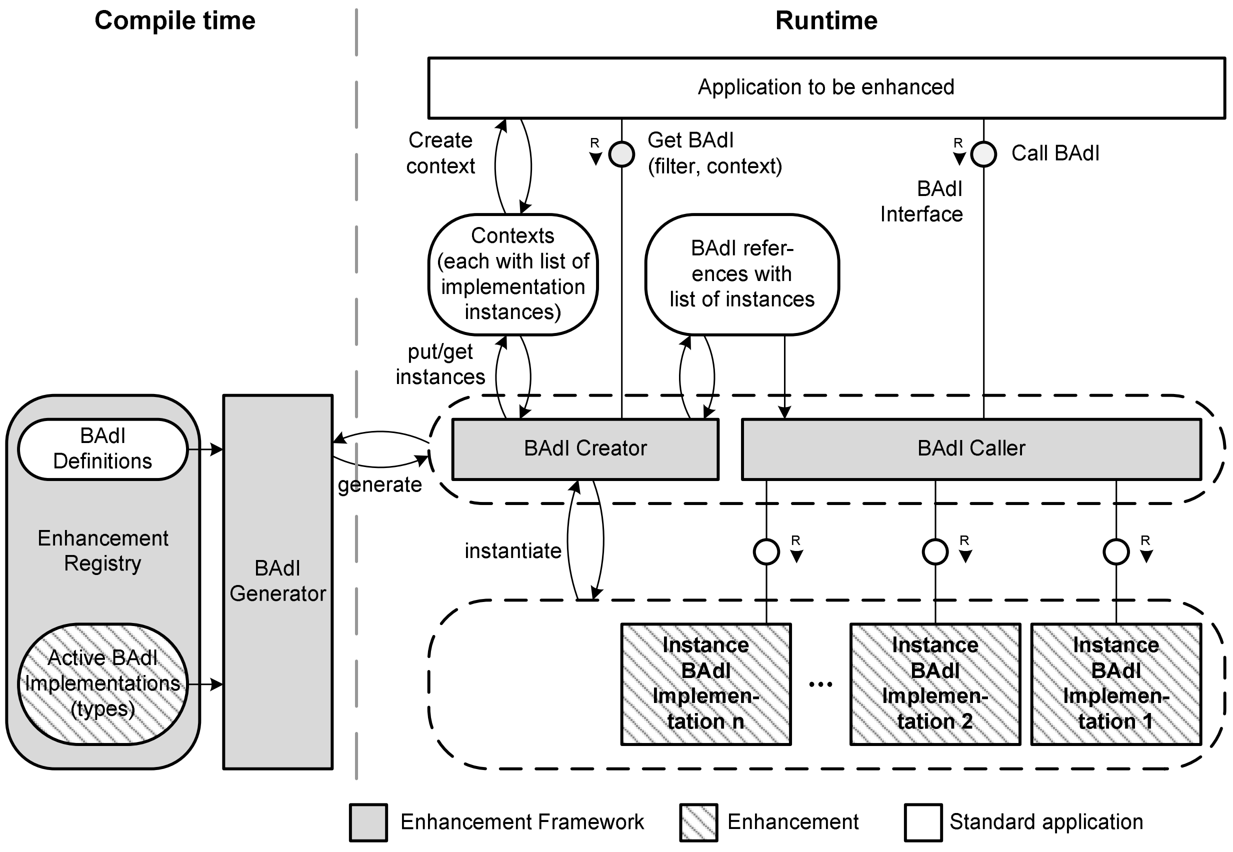
Compile time
Runtime
Application to be enhanced
Create context
Get BAdI (filter, context)
Call BAdI
BAdI Interface
R
Contexts (each with list of implementation instances)
BAdI references with list of instances
put/get instances
BAdI Definitions
Enhancement Registry
Active BAdI Implementations (types)
BAdI Generator
generate
BAdI Creator
BAdI Caller
instantiate
Instance BAdI Implementation n
...
Instance BAdI Implementation 2
Instance BAdI Implementation 1
Enhancement Framework
Enhancement
Standard application

Figure 2-7 Block Diagram: BAdI Call and Instantiation

Before an application can call methods of a BAdI, it first has to obtain a BAdI instance using the ABAP statement `get badi`. The BAdI reference is then used to invoke the BAdI implementations using the ABAP statement `call badi`. The enhancement framework has to determine the implementations to be invoked. To achieve maximum performance, this determination is done as far as possible at compile time: When the application is compiled, the framework already determines the BAdI implementations that are effective in the system and generates the corresponding instructions, which at runtime will create the instances of the found implementation classes. The references to those instances are stored with the BAdI reference for later use when the BAdI is called. For the `call badi` statement, the compiler creates code that calls the corresponding method on each implementation stored with the BAdI reference.

This situation is modeled in Figure 2-7 by introducing an agent named BAdI Generator. The BAdI generator works only at compile time. The BAdI generator has access to an enhancement registry which contains information about the BAdI definitions and also the information about the BAdI implementations that are effective in the system. The BAdI Generator is invoked whenever the compiler encounters a `get badi` or a `call badi` statement. In the model, the BAdI Generator generates agents named BAdI Creator and BAdI Caller which represent the implementation of the `get badi` and `call badi` statements. When the application requests a BAdI reference at runtime (`get badi`), this request is executed by the BAdI Creator. When the application calls a BAdI, the request is executed by the BAdI Caller.

The behavior of the BAdI Creator and the BAdI Caller is shown in the activity diagram in Figure 2-8. For each effective implementation (determined at compile time), the BAdI creator checks whether there is a filter provided by the application and if yes, whether it matches the filter values defined for the BAdI implementation. If yes (or if no filter is supplied), the BAdI Creator continues processing that implementation.

In context-based or in singleton instantiation mode, the BAdI Creator checks whether the context (see 2.3.4) already contains a reference to an instance of that class. If not, the BAdI Creator instantiates the class and stores a reference to the instance in the context. In singleton instantiation mode, a framework-internal global system context is used to ensure that the same implementation is used for every `get badi` request.

In stateless mode, the BAdI Creator always creates a new instance. The BAdI Creator associates the references to the implementations (that were found or created) with the BAdI reference that is returned to the application. When a BAdI method is called for a specific BAdI reference, the BAdI Caller invokes the corresponding method for each implementation associated with the BAdI reference.

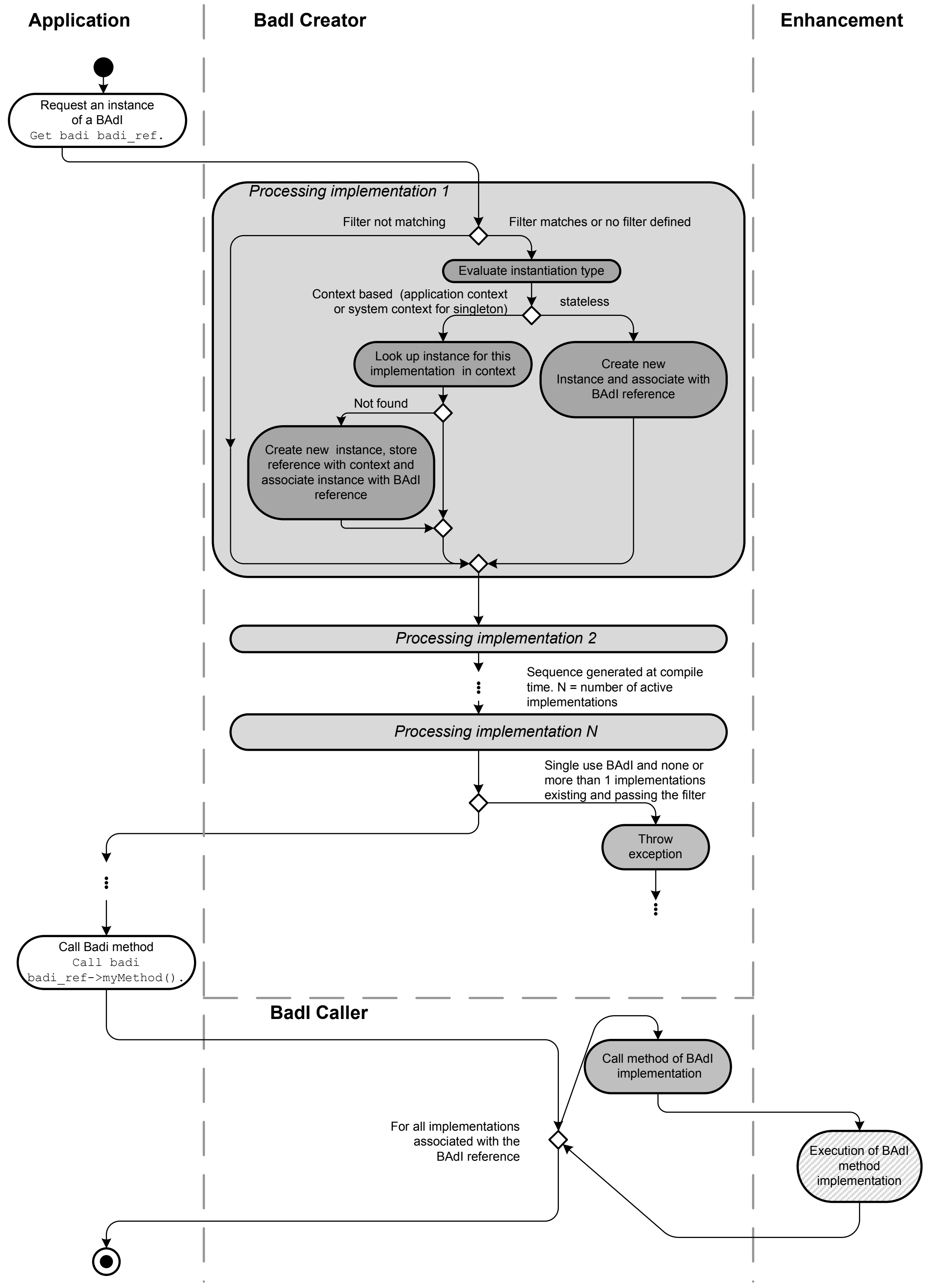
Application
BadI Creator
Enhancement
Request an instance of a BAdI
Get badi badi_ref.
Processing implementation 1
Filter not matching
Filter matches or no filter defined
Evaluate instantiation type
Context based (application context or system context for singleton)
stateless
Look up instance for this implementation in context
Create new Instance and associate with BAdI reference
Not found
Create new instance, store reference with context and associate instance with BAdI reference
Processing implementation 2
Sequence generated at compile time. N = number of active implementations
Processing implementation N
Single use BAdI and none or more than 1 implementations existing and passing the filter
Throw exception
Call Badi method
Call badi badi_ref->myMethod().
BadI Caller
Call method of BAdI implementation
For all implementations associated with the BAdI reference
Execution of BAdI method implementation

Figure 2-8 Activity Diagram: BAdI Call and Instantiation

2.3.6 Embedding of BAdIs in the Enhancement Framework

The diagram in Figure 2-9 shows an example how BAdIs are embedded in the enhancement framework. In the example, an enhancement option was added to a piece of application code by instantiating a BAdI `MyBADIDefinition` and calling one of the methods contained in the corresponding BAdI interface `IF_MY_BADI_INTERFACE`. This is an ABAP Objects interface which - like every BAdI interface – is derived from the marker interface `IF_BADI_INTERFACE` which is defined as a part of the framework. If the application specifies a context when requesting a BAdI instance, this must be an instance of the interface `IF_BADI_CONTEXT`.

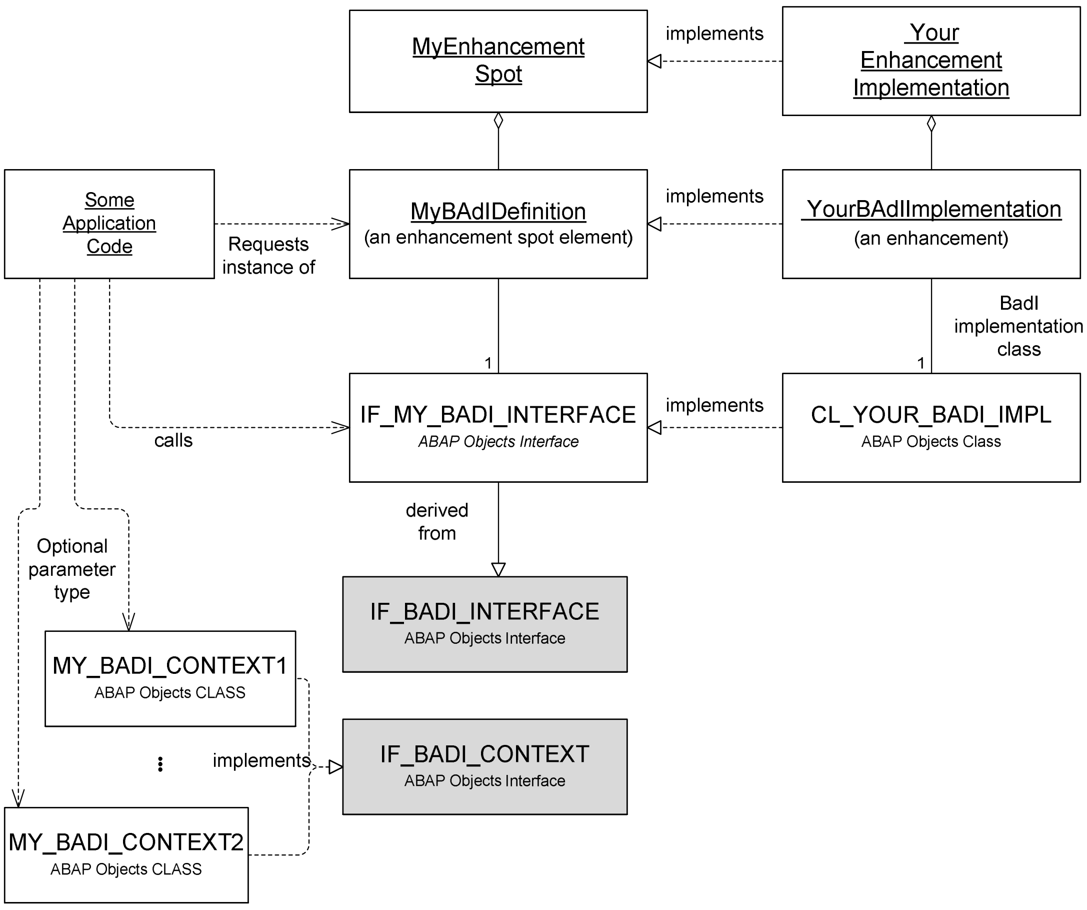

Figure 2-9 Embedding BAdIs in the Enhancement Framework

Assuming the BAdI interface contains a method `m1` with no parameters, the call code in the application would look like this:

```
DATA badiref type ref to MyBAdIDefinition.
GET BADI badiref filters filter1 = value1 context context1.
CALL BADI badiref->m1( ).
```

Because the BAdI concept is built into ABAP language, the application developer does not have to declare a reference to the BAdI interface and get an instance from some factory object. Instead the BAdI reference appears like a reference to an instance of the BAdI interface, through which the interface methods can be called.

The BAdI definition is an enhancement spot element which is assigned to the enhancement spot `MyEnhancementSpot`.

2.4 Enhancements and Switches

As indicated in Figure 2-4, all enhancement implementations can be switched using the switch framework.

With switches, implementations of enhancement options can be shipped in an inoperative state. Those enhancement implementations can be installed in a system, but they will be ignored by the enhancement framework unless they are explicitly switched on. The switch framework is introduced in chapter 3.

2.5 Nested Enhancements

It is not allowed to modify or delete an existing enhancement, for example an enhancement created by SAP industry solution development. Instead, an enhancement for this enhancement implementation has to be implemented. By this, a hierarchy of enhancements is possible. The enhancement framework supports nested enhancements for BAdIs. Putting enhancement sections inside enhancement sections is not possible at the time this document was written, but is planned for future releases.

In addition, you can create Overwrite Enhancements that replace existing enhancement implementations. Whenever an upgrade changes the replaced enhancement, an entry indicating that an adjustment may be necessary will appear in transaction SPAU_ENH.

2.6 Implicit Enhancements

2.6.1 Implicit Source Code Enhancement Options

For ABAP programs, a number of implicit enhancement options exist, for example:

- At the end of an include,
- At the end of a structure definition (`types, data, constants, statics`)
- At the start and at the end of a method or function module

These options provide very powerful means to alter standard code. In some cases, there are no other ways to enhance, for example when adding a type or data definition. In other cases, SAP strongly recommends to use BAdIs instead, since they provide a defined interface.

2.6.2 Enhancements of Classes, Interfaces and Function Modules

There are many ways to enhance ABAP classes and function modules. On the one hand, there are enhancements of the class or function module definition, for example additional attributes or methods, which are implicit by definition. On the other hand, there are of course enhancements of the existing code. Here, the recommended way is to use BAdIs, but there are of course many implicit enhancement options as well.

In **function modules**, you can

- add optional parameters,
- insert code at the start of the function module implementation,
- insert code at the end of the function module implementation.

In **local classes**, you can:

- insert code at the end of the `PUBLIC`, `PROTECTED` and `PRIVATE` sections of the class definition,
- insert code at the end of an interface definition,
- add parameters (`CHANGING, IMPORTING, EXPORTING`) to a method definition,
- insert code at the end of the class implementation section, for example to add method implementations,
- Insert code at the start and end of a method implementation.

In **global classes** and interfaces, you can

- add methods,
- add attributes,
- add events,
- add interfaces,
- add types (release 7.10),
- add optional parameters to methods,
- insert pre-, or post-methods to existing methods,
- replace method implementations by overwrite-methods.

A frequently asked question is why those enhancements are not done in the object oriented way, that is using inheritance for adding or overwriting methods. Inheritance cannot be used for two reasons: One is that sub classes do not help if you cannot control the instantiation of the objects. This is the case when the objects are created as instances of some standard class by code that can not be overwritten by extensions. But even if there would be another extension option allowing to replace the creation of objects instances, the other problem remains: Parallel development of extensions by independent extension providers would hardly be possible with inheritance. Objects must be instances of some concrete class, and if one extension provider would overwrite the instantiation to create objects of his subclass, all other extensions would be locked out. And finally there is trouble given the class to be enhanced already uses inheritance.

Most enhancements are treated like an in-place modification, for example an added attribute or method appears in the class builder like all others, with an additional reference to the enhancement implementation object. For example, you can add an attribute `new_attr` and a method `new_method`, and within the method implementation you can access private attributes, be it original or enhanced attributes.

The enhancement implementation of pre- / post- / overwrite methods is currently located in separate local classes. The enhanced class is automatically extended by an attribute referencing to the instance of the local class that also has an attribute `core_object` pointing to the object to be enhanced. This results in a restriction of the enhancement coding: private attributes of the enhanced class are not accessible. Furthermore, if you add an attribute `new_attr` to a class and try to access it from a pre- / post- / or overwrite method, you have to declare `new_attr` as public and access it via the reference `core_object->new_attr`. This restriction is removed in release 7.10 by using the ABAP Objects `friends` mechanism.

2.6.3 Web Dynpro Enhancements

Web Dynpro technology offers many possibilities for enhancements.

- Layout:
  Add or remove user interface elements in Views
- Contexts (of Controllers, Views, Windows):
  Add nodes and attributes
- Navigation:
  - Add navigation links,
  - change navigation targets in Windows,
  - add inbound / outbound plugs in Views and Windows,
  - add actions to views
- Controller (component / window / view):
  - Add methods,
  - insert pre- / post-methods,
  - add attributes

The typical use case is to add fields or buttons to a screen and propagate data to be displayed or entered via contexts and controller methods to the backend.

In contrast to Web Dynpro, the (old) Dynpro technology is not covered by the enhancement framework, although screen elements and module calls can be switched, as explained in section 3.2.2.

2.7 Future Development of the Enhancement Framework

The enhancement framework offers many options to enhance existing applications. Nevertheless, with implicit enhancement options, similar problems have to be expected like with the modifications. Therefore, some measures are planned for the next release to get more control of possible enhancements.

One option is the new package concept [15]. Currently, only the extended syntax check – checkman – controls whether only package interfaces are used from outside a package. The new package concept will give a warning immediately when a developer tries to bypass package interfaces. Furthermore, a developer can then define the enhancability level of a package, for example

- Not enhancable,
- Only the implementation of *explicit* enhancement options is allowed, or
- Fully enhancable (everything allowed).

Enhancement implementations will be located in a separate enhancement append package that is assigned to the original package so that the package check treats the content like it was part of the original package.

In addition there will be the option to allow or deny enhancements for global ABAP objects classes (like with dictionary objects). Implicit enhancements of a class will then only be allowed if the class is marked as "enhanceable".

Other features are nested enhancements (see section 2.5), a new edit control in the workbench that integrates enhancements in a more intuitive way, and multiple loads, allowing to switch all code enhancements depending on the client.

The feedback from the current users of the enhancement framework will result in more specific requirements for the framework development in the future.

3 The ABAP Switch Framework

3.1 Goal and Scope

The ABAP Switch Framework offers an organized way for defining and managing optional features that can be switched on demand. In case a feature is switched off, the corresponding software objects are not effective although they are installed in the system. There are multiple usage scenarios for the switch framework:

Developing and Delivering Industry Solutions / Industry Extensions

Aside from the standard business suite, SAP delivers industry-specific solutions, for example for Oil & Gas, or for automotive industries, complemented by industry extensions. In former releases, the development teams had to modify the standard release, and apply and test these modifications with every new release of the SAP standard business suite. One result was that some industry solutions were based on older versions of the suite, which lead to problems when a customer had to operate different systems.

With the switch framework and the enhancement framework, a consolidation was possible with the result that industry solutions are now developed as enhancements of the standard applications. The advantage for a customer is the fact that now it is possible to have ERP standard and an industry solution within one system. Before, the customer had to install updates on the ERP system immediately, but had to wait until the industry solution had been ported and tested with these updates. The ERP changes would have overwritten the industry modifications. The enhancement framework can now prevent this.

With the switch framework, conflicting enhancements can be installed into the same system as long as only one is switched on, or when a conflict switch (see section 3.2.6) resolves the conflict. This is used for delivering SAP industry solutions. With the Switch framework, many industry solutions can be installed in an inoperative state. By doing so, multiple mutually exclusive industry solutions can be delivered as part of the same installation. The customers can then switch on the industry solution they actually need.

Enhancement Package Concept

The switch framework allows to separate the installation of enhancements from their activation. This is used by the SAP Business Suite for the enhancement package concept. With this concept, customers initially install the core, or “Go to” release of the product. The customers can rely on the rule that the features of the core release remain stable for the lifetime of the product. Customers who do not need new features get support packages for the core that are guaranteed to contain fixes, but no new functionality. New features are delivered as enhancement packages, containing many new features which are initially installed in an inoperative state.

The installation of an enhancement package should be fast and without any risk. It must not cause any change in the visible behavior of the system and not disrupt any existing business processes. The customers can then selectively switch on the features they need. To achieve a similar granularity of control without the switch concept, a complicated installation process with much smaller installation units would be required. For details about the concept of a stable core with enhancement packages, see [13].

Managing Conflicting Enhancements

Industry specific development is one example for a scenario with multiple parties contributing conflicting enhancements for the same product. The switch framework helps here because it allows to *install* the conflicting enhancements in the same system. For enhancement technologies like single-use BAdIs and enhancement sections, multiple implementations must not coexist. Without switches, this rule would cause additional complexity for software logistics: Installations, upgrades or imports would leave the system in a bad state if they contained additional implementations of an already implemented “exclusive” enhancement option. Administrators would have to be very careful before deciding which installations or imports are allowed. With switches, all enhancement implementations can be imported in an inoperative state. As long as only one implementation is switched on, there will be no conflicts. Unless the providers of the conflicting implementations are completely unrelated, the conflicts can even be described and managed using the concept of conflict switches (see 3.2.6) or dependencies of business functions (see 3.3.2).

Testing Effects of Enhancements

In some situations switches can be used to compare the behavior of a system with and without the enhancements done by some partner or customer. This can be useful to determine whether a certain problem was caused by the standard application or by an enhancement. However, this is not an option when the enhancement is needed for correct system function. This would be the case if the system already contained data that can only be processed correctly by the enhancement logic.

3.2 Switch Framework Concepts

3.2.1 Switches for Development Entities

A switch determines whether an associated switchable development object is effective or inoperative. Switches are repository objects.

The difference between a switch and a field in a customizing (IMG) table is that a switch can define which parts of the source code will be compiled – inoperative code will not be compiled and therefore will not be available at all.

3.2.2 Switchable Units

A switch determines

- whether DDIC objects such as types are available in the system,
- whether a development object, for example an enhancement implementation, is effective and can be used in the system,
- the visibility of maintenance views within view cluster and view fields in maintenance views,
- the visibility of screen fields and which modules are called in dynpro flow logic.

There are various types of development objects that can be switched, but the possibilities to declare them so at design-time differ slightly: Most types of objects, for example all enhancement implementations for ABAP as shown in chapter 1, are assigned to a switch indirectly by associating their package with a switch. Some development objects do not belong to a package; they can be associated with one or multiple switches, depending on the object type. Furthermore, you can specify switchable GUI elements whether they should be visible when the switch is turned on, or whether they are hidden. Figure 3-1 illustrates the applicable different possibilities of switch assignments.

Although area menus belong to packages, their package switch is not used but switches can be defined per node. Thus, granularity of switching is increased, and switch definition is aligned with IMG nodes which use the same hierarchy tool.

Business Configuration Sets (BC Sets) are containers of business configuration data. They contain table-like data, such as views, view clusters, customizing data, logical transport object, database tables, or individual transport objects. A BC set is a repository object that can be transported to other systems.

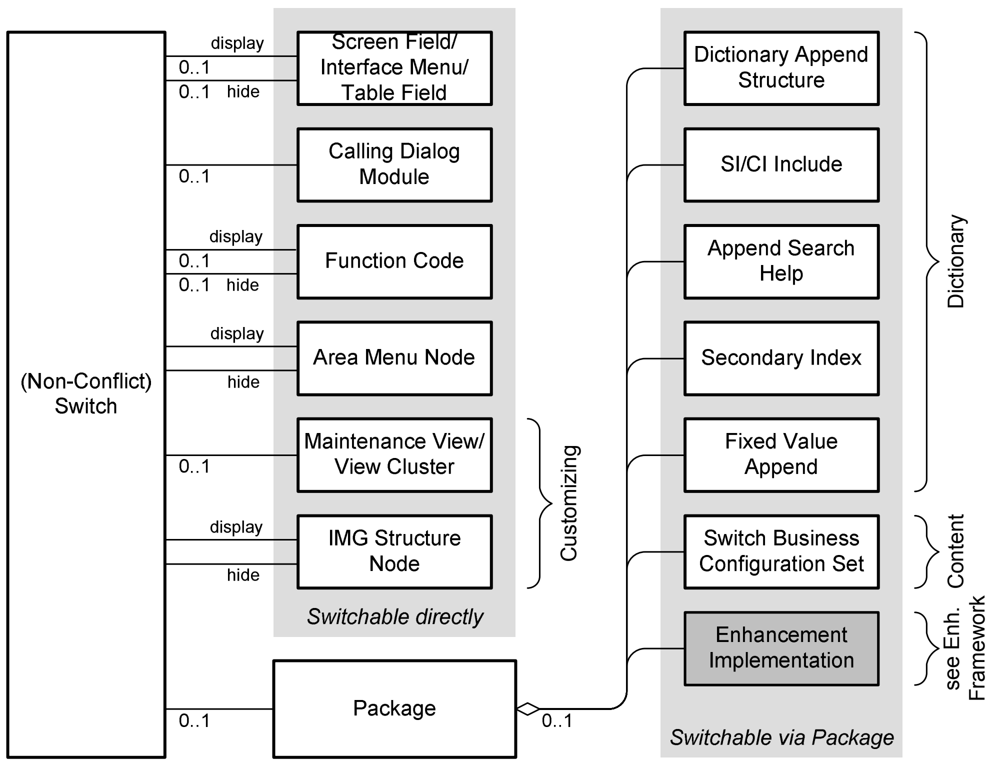

Figure 3-1 Switchable Objects

A **Switch BC Set** is a Business Configuration Set that is assigned to a switch via its package affiliaton, see figure 3-2. Since the content of a switch BC set will be copied to the database only when setting the switch on, switch BC sets are a good means to transport additional table data with an enhancement package.

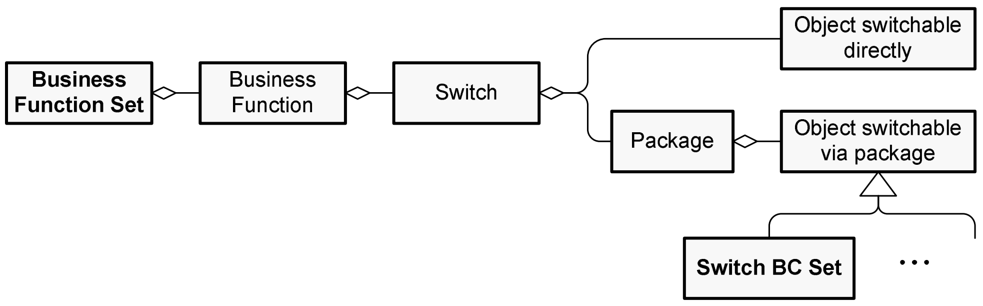

Figure 3-2 From Business Function Set to Switch BC Set

When activating a switch BC set, its content is copied to the corresponding database tables. A switch BC set is also activated when it is transported to a target system where the corresponding business function set is already effective (automatic activation). In contrast to other switchable objects, a switch BC set can be activated directly, for example when errors occurred during automatic activation.

To fill switch BC sets with content, one way is to use switch BC set recording during customizing. This recording uses standard workbench transport requests used during customizing and converts them to switch BC set content, which needs some additional work before the switch BC set is transportable (record completion).

3.2.3 Switches and package hierarchy

A package can be assigned to exactly one switch; a switch, on the other hand, can be assigned to multiple packages. All switchable repository objects are affected by the package's switch state.

Packages can be nested. The state of a switchable object depends on the switch assignment of its package; if there is none, then on the assigment of its super package, and so on. Therefore, a subpackage switch overrides a super package switch.

3.2.4 Switch states

Not all development entities can be switched in the same way: While it is simple to either compile or not compile a source code section, switching table definitions and other DDIC structures have to be treated carefully. Therefore, SAP switches provide three states:

- Off

  All associated objects are switched off. The system behaves as if the objects did not exist.[7]

- Standby (enabled, but not effective)

  All declarative repository objects such as DDIC structures, tables, and static source code enhancements (that can only contain declarative statements) are switched on. All other repository objects are switched off.
  Standby has system-wide consequences.

- On

  All associated objects are switched on. They are available in the system like non-switched objects.
  At least in SAP-internal systems, this activation can be client- or user-specific.

At a given point in time, a switch has one of three states: OFF, STANDBY, or ON. Simply spoken, an object is not visible in the system if the switch is OFF, only visible at development time if the switch is STANDBY, and visible at runtime if the switch is ON. Except for DDIC switches and Business Configuration Sets, switch states can be client specific.[8]

A use case for standby switch state is when an industry solution requires some data types of an industry extension without having to activate the coding.

In case a switchable object is not assigned to a switch (for example because neither its package nor super packages are assigned to a switch), then it is regarded as switched on.

3.2.5 DDIC switches

A "DDIC" flag can be used to document whether a switch affects DDIC objects. This kind of switches is sometimes also called DDIC switch.

3.2.6 Conflict Switches

Sometimes switches are in conflict because different enhancement implementations are associated with the switches, but the enhancement only allows for one implementation, such as a single-use BadI or an enhancement section. In this case, the conflicting switches are assigned to a special conflict switch.

[7] The consequence is that the coding does not exist for the syntax check as well!

[8] For example, the ABAP compiler will embrace a source code plugin that depends on a client-specific switch by `IF/ENDIF` statements automatically. Client-specific switches are available only SAP-internally.

A conflict switch usually should resolve the conflicts of the assigned switches. The enhancement implementation provided by the conflict switch overrides the conflicting implementations provided by the switches, as shown in figure 3-3. If all assigned switches are on, the conflict switch has the state on, else off.

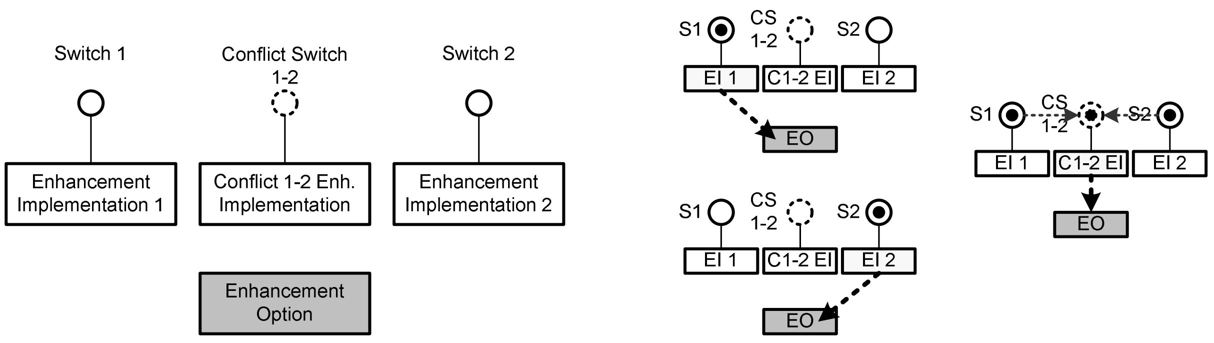
Switch 1
Conflict Switch 1-2
Switch 2
Enhancement Implementation 1
Conflict 1-2 Enh. Implementation
Enhancement Implementation 2
Enhancement Option
S1
CS 1-2
S2
EI 1
C1-2 EI
EI 2
EO

Figure 3-3 Conflict switches

As a consequence, you have to provide such conflict resolving implementations for all problematic combinations of the assigned switches. For example, to cover all possible combinations of three conflicting switches S1, S2, and S3, you would have to provide four conflict switches CS1-2, CS1-3, CS2-3, and CS1-2-3 together with the corresponding enhancement implementations.[9]

In contrast to normal switches, a conflict switch is not assigned to a business function. Instead, you have to assign the conflicting switches to the conflict switch. It is set to ON if all conflicting switches are set to ON. Since the enhancement implementations are switched via the package that is assigned to the conflict switch, as a consequence you need one additional package for the implementation of each conflict switch.

The typical use case for conflict switches is that one provider offers two business functions that contain conflicting implementations, and provides the corresponding conflict switch implementations as well.

3.3 Organizing Switches

Since there may be many switches that have to be switched as a group to enable a new functionality, it is not possible to change the state of a switch directly. Instead, the concept of business functions has been introduced that map one specific business function to a set of technical switches.

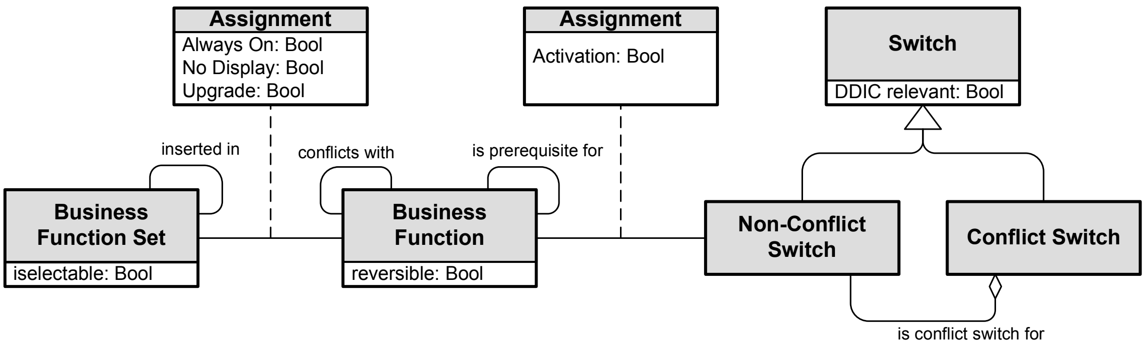

Figure 3-4 Switch Framework entities

[9] For *n* conflicting switches, there are $2^n - n - 1$ combinations.

Note that the sets of switches assigned to different business functions are not disjoint. The same switch can be assigned to multiple business functions. Even if some business function is switched off, some of the assigned switches may be on if they are controlled by other business functions as well. This might lead to surprising results if the switches and the switch assignment were not designed carefully.

3.3.1 Switching via Business Functions

Switches (but not conflict switches) are assigned to business functions either as an **activation switch** or as a **standby switch** (see figure 3-4). From an administrator's point of view, business functions (see 3.3.2) are the most fine-grained units that can be turned on or off, thus changing the state of assigned switches. Initially, all switches are in state OFF. When a business function is set to ON, activation switches are turned to ON, and standby switches to STANDBY. When a business function is switched OFF, the framework has to check all assigned business functions. If one of them is ON, then the switch will not be turned off (see Figure 3-5).

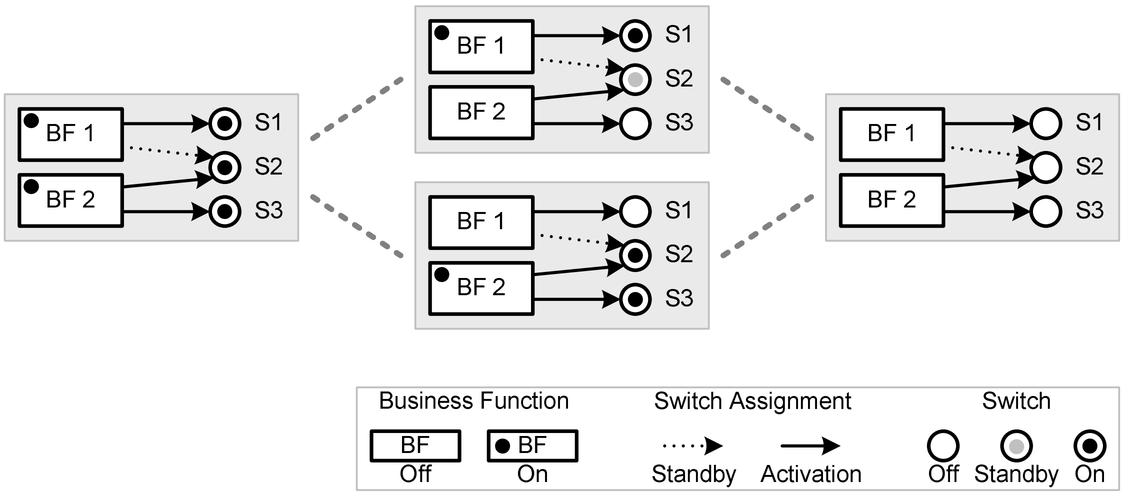

Figure 3-5 Switching via Business Functions: Determining the Switch state

Standby switch assignment is needed when packages with mixed content (type definitions and executable code) are needed by multiple business functions, and some of them only need the type definitions.

3.3.2 Business Functions

A business function is a self-contained function from a business perspective that refers to a set of switches[10]. A switch can be assigned to more than one business function (see figure 3-4).

By activating a business function, all assigned switches are turned on or to standby respectively, depending on the definition of the business function. Business Functions may be declared reversible if their switches don't affect the data dictionary or BC sets. Only **reversible business functions** can be turned off again. However, this is not recommended in production systems.

When assigning switches to business functions, you can choose whether a switch should be turned on or turned to standby when the business function is activated. This is explained in section 3.3.1.

[10] Conflict Switches are not assigned to Business Functions.

There are different types of Business Functions [14]:

- **Industry Business Functions** belong to a Business Function Set. After selecting a business function set, you can only turn on industry business functions assigned to this set, which relates to an industry solution.
- **Enterprise Business Functions** contain functions that you can turn on and use independently of any selected business function set. They usually contain enhancements of the standard.
- **Enterprise Extensions** (formerly Enterprise Enhancements) are first generation business functions that are available for full releases of SAP ECC (for example, SAP ECC 6.0). You use them to activate a number of industry-independent and industry-specific applications and business processes for each business function.

A Business Function may depend on another Business Function as a prerequisite, or be may be not allowed in conjunction with another Business Function. Those dependencies can be defined by developers and are checked before turning Business Functions on or off.

3.3.3 Business Function Sets

A business function set is a group of business functions that usually corresponds to an industry solution. A business function can be part of multiple business function sets. See figure 3-4 for an overview of the terms.

Business Function Sets are used to bundle Business Functions that should work together in an Industry Solution. Business Function Sets can be nested, for example to include one industry solution in another. Therefore, they can be “insertable”, that is to structure Business Function Sets only, and they can be “selectable”, that is they are able to be switched on.

You select a business function set using the switch framework customizing tool (see 3.4.2). Now you get a selection of industry business functions which you can turn on. You can change the selection of the business function set only to a set that also contains the industry business functions that already have been turned on.

3.4 Using the Switch Framework

As depicted in figure 3-6, the switch framework affects various phases in software lifecycle. Developers use the switch framework tools to define additional information about developed objects that should be switchable. These are, on the technical side, switches and their assignments to the objects, and on the business side, business functions and business function sets that bundle switches and have additional semantics.

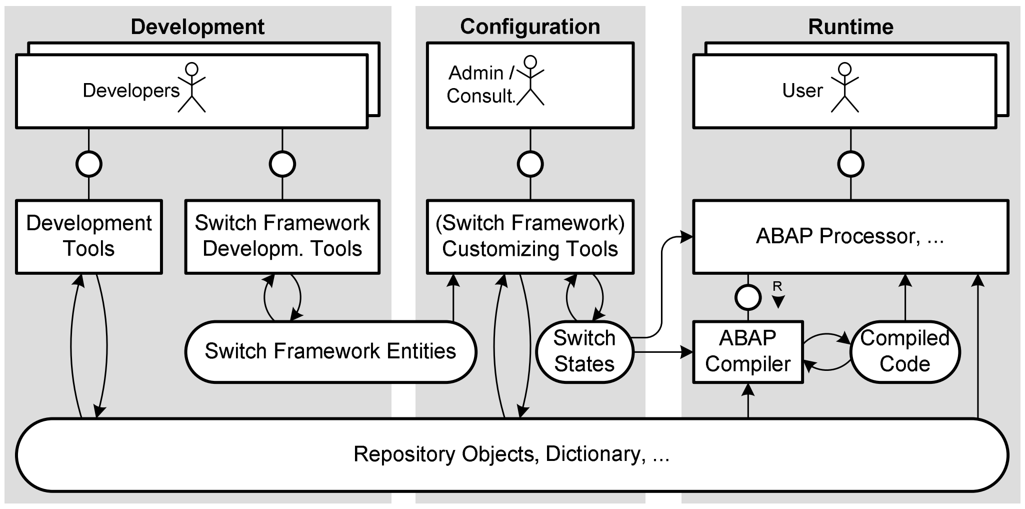

Figure 3-6 Architectural Overview

At configuration time, switches can be turned on in order to enable certain business functions. This results in changes to the switched objects, such as generation, activation and database table manipulation.

Turning on or off switches via business functions often has no direct effect until the development object is used first time at runtime. Switching a package just invalidates the source code, and the new switch state is evaluated during compilation.

3.4.1 Assigning Switches and Business Functions

Figure 3-7 provides more details about the switch framework definitions mentioned above. The Switch framework provides a dedicated tool that allows the manipulation of all major entities and their assignments, such as switches, business functions, business function sets.[11]

Many switchable objects are assigned to a switch indirectly via their package. Packages have to be assigned to a switch in the switch manager tool (SFW1). Other switchable objects are assigned to a switch in the development tool where they are developed.

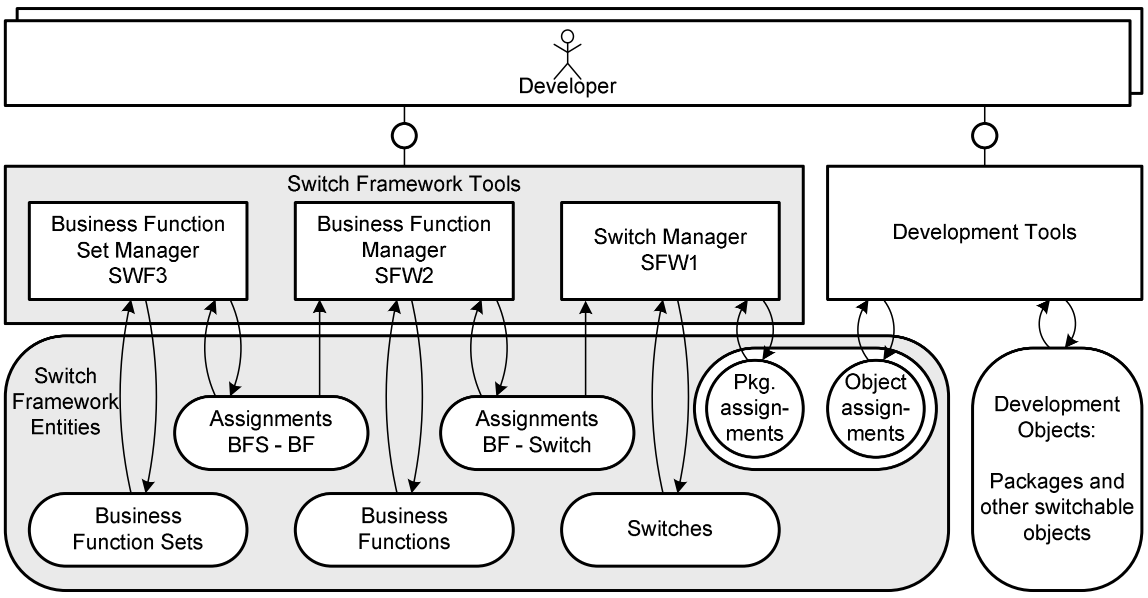

Figure 3-7 Switch Framework Entities and Tools for the Developer

3.4.2 Switching Behavior

It is the responsibility of a system administrator or consultant on the customer side to activate the required business functionality, which results in switching. There are two use cases: When the customer decides to activate a business function set, this is a decision for a certain Industry Solution that cannot be undone. However, the user can choose a subset of the associated business functions to be activated. The other use case is to activate standalone Business Functions, in particular enhancements to an existing solution or Enterprise Business Functions from custom development. Under no circumstances switches can be activated independently.

Switch activation results in manipulation of various objects such as ABAP code, ABAP Dictionary objects or customizing settings. By means of Switch BC Sets (see 3.2.2), also entries of database tables can be

[11] These tools are started using the transactions SFW1, SFW2 or SFW3, respectively

manipulated. Furthermore, it is possible to specify a BADI implementation that should run automatically after switch activation (Figure 3-8).[12]

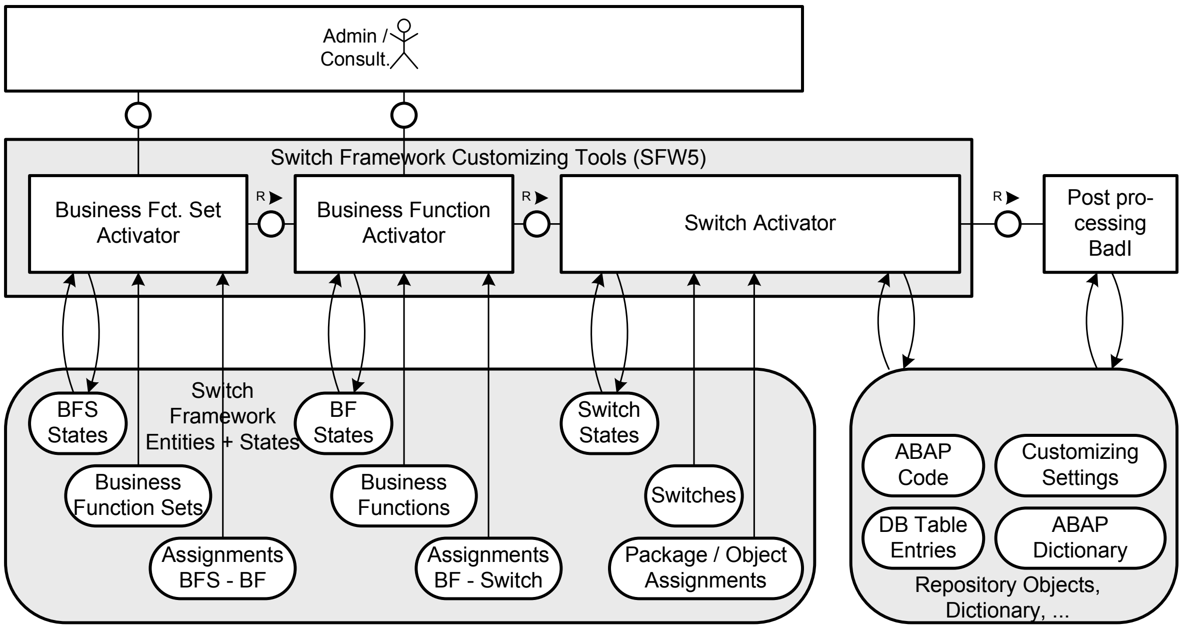

Figure 3-8 Configuration Scenario

The Switch Framework Customizing Tools (SFW5) also allow to store the current setting of all switches, business functions, and business function sets in a standard transport request that can be transported to a target system, for example from the test system to the production system. To avoid inconstistencies, it is not possible to transport a partition of the switch state.

[12] Since many switched enhancements are generated in the code by the ABAP compiler, changing the state of a switch often just results in the invalidation of the corresponding ABAP load. Next time a switched object, for example a source code plug-in is called, the compiler will generate the new load that follows the new switch setting.

4 Summary

SAP software needs to be enhanced by different parties for different reasons: The release strategy for SAP Business Suite and SAP NetWeaver defines that innovations are shipped as enhancements based on a stable core release of the products. Industry extensions and industry solutions provide enhancements to core applications based on the requirements of a specific industry. ISVs and customers need to enhance SAP applications to adapt them to even more specific needs. In the past, industry solutions, partners, and customers often modified the SAP applications directly. However, this approach lead to problems and high efforts, for example for adjusting modifications after software upgrades. Several technologies such as user exits or the classic Business Add In (BAdI) were developed in the past to support modification-free enhancements, but each of them had different concepts and tools to manage the definition of enhancements and their lifecycle.

The enhancement framework provides a uniform approach for defining and managing different types of enhancements for ABAP applications. All the supported enhancement technologies are modification free, which means that the enhanced piece and the enhancement are always stored as two separate software objects. The enhancement framework replaces existing enhancement technologies such as user exits or the classic business add-in (BAdI). It supports different enhancement technologies such as the new BAdI, class enhancements, function module enhancements, Web Dynpro enhancements, or source code plug-ins. The framework has a modular internal structure that was designed for adding new enhancement technologies easily.

The enhancement framework allows application providers to define, document and organize the different options that are offered for enhancing the application. Enhancement options can be described with meta data, which for example control whether multiple plug-ins can be effective for the same enhancement option in the same system. Enhancement options that belong to the same topic or functionality can be grouped in a hierarchy. This makes it easier for enhancement providers to locate and understand the different enhancement options that are offered. Like the enhancement options, the actual enhancements can also be organized in semantic hierarchies, documented and attributed with meta data. An example for meta data are filter values that allow to select an enhancement implementation from different alternatives. One of the key features of the enhancement framework is the unified life cycle support. When a new version of an enhanced application is imported (during software transports or upgrade) into a system where enhancements exist, the framework detects potential inconsistencies and reports them to the enhancement developer.

The ABAP switch framework provides a structured approach for installing software in a system in an inoperative state, with the option to activate it later. In inoperative state the software exists in the system, but is not executed. The switch framework is connected with the enhancement framework because enhancements, among other types of software objects, can be switched. This allows installing enhancements in a system without activating them. It is even possible to install multiple conflicting enhancements – that provide different plug-ins for the same socket – as long as only one is activated. In addition to enhancements, there is a variety of other development objects that can be switched, for example user interface elements, IMG nodes, dictionary append structures or secondary indexes for tables.

Two important applications of the switch framework are enhancement packages and industry solutions. Enhancement packages are used to ship innovations in a non-disruptive way. With the switch framework, all enhancements can be installed in inoperative state, so that the customer will not encounter any noticeable change in behavior after the installation. Using functionality of the switch framework, the customers can then switch on the features they actually need. To make this process easier, the switches are grouped in business functions which represent a meaningful functionality and are the only units that can be switched by customers. Industry extensions and solutions are delivered as sets of switchable business functions. This allows installing many industry solutions in the same system, as long as only one of them is switched on. The switch framework allows to model dependencies as well as conflicts among both business functions and switches.

5 Further Reading

[1] *SAP Library: Switch Framework (NetWeaver 7.0)*
help.sap.com/saphelp_nw70/helpdata/en/af/e8b540afc87c2ae10000000a155106/frameset.htm
An introduction into the concepts and terminology of the switch framework

[2] *Software Requirements Specification: SFW Specification – Architecture for Switch Framework*
(https://portal.wdf.sap.corp/irj/go/km/docs/corporate_portal/WS%20PTU%20SAP%20NetWeaver/Product%20Information/Key%20Capabilities/Application%20Server/ABAP/ABAP%20Workbench/Switch%20Framework%20Architecture.doc)

[3] *Software Requirements Specification: SFW Specification – Switch Implementation*
(https://bis.wdf.sap.corp/twiki/pub/Sapinternal/SwitchFramework/sfw_design.doc in BIS TWiki)
Functional Requirements: ABAP und BAdIs (Alternatives)

[4] *Design Switch-Framework*
(https://bis.wdf.sap.corp/twiki/pub/Sapinternal/SwitchFramework/switch_framework_design.doc in BIS TWiki)
Development Objects, Tables, ABAP Language Extensions, BAdIs, Enhancement Framework…

[5] *Enhancement-/Switch-Framework (TechEd 06)*
(https://portal.wdf.sap.corp/irj/go/km/docs/room_info/cm_stores/documents/workspaces/21f50c17-5ce2-2810-6a99-a6c0b14559eb/Dresden/Enhancement%20and%20Switch%20Framework.ppt)
Introduction to the concepts of the Switch Framework and its relation to BAdIs

[6] *Switch Framework, Presales Workshop 7.6.05*
(https://portal.wdf.sap.corp/irj/go/km/docs/corporate_portal/WS%20PTU%20SAP%20NetWeaver/Product%20Information/Education_0_/Internal%20Knowledge%20Transfer/Presales%20Workshop%202005/Detailed%20Presentations/Switch%20Framework.ppt)
Switch Framework and how it affects coding

[7] *Industry Consolidation in mySAP ERP 2005* (http://smart.sap.corp:1080/smart/media.asp?v=41680)
Product centered view, planned Business Function Sets in ERP 2005

[8] *Introducing the switch and enhancement framework – consolidating industry solutions with mySAP ERP core*
(https://portal.wdf.sap.corp/irj/go/km/docs/corporate_portal/WS%20PTU%20SAP%20NetWeaver/Product%20Information/Key%20Capabilities/Application%20Server/ABAP/Programming/SAP%20Professional%20Journal/Switch%20%26%20Enhancement%20Framework.pdf)

[9] *Enhancement packages: Table Entries, Switch BC Sets*
(https://portal.wdf.sap.corp/irj/go/km/docs/corporate_portal/WS%20PTU%20Applications/Projects_1_/ERP%20Program/ERP_Enhancement_Packages/Development%20Documents/EHP_Table_Entry_Switch_BC_Sets_developer_version.ppt)
Description of Switch BC Sets

[10] *Switch Framework*
(https://portal.wdf.sap.corp/irj/go/km/docs/room_project/cm_stores/documents/workspaces/a1a8d45e-752b-2910-a38f-a77d3c0281f8/Presentations/SwitchFramework_AOF_06-Feb-06.ppt)
Short description of tables, infos concerning upgrade relevance and BC sets

[11] *SAP Library: Enhancement Framework (NetWeaver)*
help.sap.com/saphelp_nw70/helpdata/en/a0/47e94086087e7fe10000000a1550b0/frameset.htm

[12] *Revised Class Enhancement Concept*
Working Draft, Michael Acker, Markus Frick, Christoph Wedler, NW Foundation ABAP
\\ABAP\documents\Switch_Framework\ClassEnhancements\ClassEnhancements.doc

[13] *Stable Core and Enhancement Packages, Product Roadmap Document, SAP, 2007*
https://portal.wdf.sap.corp/irj/go/km/docs/corporate_portal/WS%20PTG/Product%20Architecture/Architecture%20Projects/Strategy%20and%20Guidelines/BS2008%20Stable%20Core%20and%20Enhancement%20Packages.pdf

[14] *SAP Library: Business Functions in ERP 2005 Enhancement Pack 3*
http://wbhelp.sap.com/ERP2005_EHP_03/helpdata/EN/97/9bddd4cebe423f9eb5767a275b2d78/content.htm

[15] *Advanced ABAP Package Concept in SAP Netweaver* (Working Draft), 2006
https://bis.wdf.sap.corp/twiki/pub/Sapinternal/PackageConceptDocs/PackageConcept_most_recent.doc

[16] SAP Developer Network Blog Series on Enhancement Framework, Thomas Weiss, SAP

- *Part 1: What the New Enhancement Framework Is For – Its Basic Structure and Elements* For Beginners, https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/3056
- *The new Enhancement Framework Part 2 - What Else You Need to Know Before Building* an Enhancement, https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/3336
- *How To Define a New BAdI Within the Enhancement Framework - Part 3 of the Series*, https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/3430
- *How to implement a BAdI And How to Use a Filter - Part 4 of the Series on the New Enhancement Framework*, https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/3506
- *Source Code Enhancements - Part 5 of the Series on the New Enhancement Framework*, https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/3595

6 Glossary

Term	Definition
BAdI (Business Add-In)	Object-oriented plug-in with well-defined interface and meta data that control number of allowed implementations, instantiation mode or selection criteria. (p.15)
BAdI Call	Invocation of a BAdI in the application to be enhanced. Establishes an Enhancement Option.
BAdI Definition	Definition of BAdI name, cardinality, instantiation type and ABAP interface.
BAdI Implementation	Implementation of a BAdI Definition in form of an Enhancement Implementation Element that is switchable via its package.
Business Configuration Set	Transportable container of table-like data, such as customizing settings (p.25)
Business Function	Self-contained function from a business perspective, technically a collection of switches (p.29)
Business Function Set	Group of Business Functions, usually corresponds to an industry solution (p.30)
Composite Enhancement Implementation	Serves as a means to hierarchically group enhancement implementations and thereby support an hierarchy of extensions following semantic groups.
Composite Enhancement Spot	Serves as a means to hierarchically group enhancement spots and thereby support an hierarchy of extensions following semantic groups (p.12)
Conflict Switch	A special switch that groups switches with potentially conflicting implementations. The implementation assigned to a conflict switch has the task to resolve conflicts (p.27)
Enhancement Implementation	defines an enhancement implementation for an enhancement spot or for an implicit enhancement option. The concrete development objects of the implementation are managed by Enhancement Implementation Elements. (p.12)
Enhancement Implementation Element	Used to manage concrete implementations of an enhancement, for example a BAdI implementation or a source plug-in, as part of an Enhancement Implementation (p.11)
Enhancement Option	Provides the possibility for changing or extending the behavior of applications (by providing a corresponding enhancement implementation). Can be explicit (pre-planned by application provider) or implicit. Only explicit enhancement options are managed as separate objects in the Enhancement Workbench.
Enhancement Package	Part of the delivery concept of the SAP Business Suite and SAP NetWeaver for “non-disruptive innovation”. Based on a stable core release, innovations are shipped as separate strictly optional enhancement packages. Features in enhancement packages must be explicitly activated by the customer using the switch framework. The enhancement framework is often used to implement new features (p.24)
Enhancement Point	ABAP Keyword at a place in ABAP code where implementing enhancement code can be inserted. For one enhancement point, multiple implementations are allowed (p.9)

Enhancement Registry	stores metadata about enhancements and spots for design- and runtime purposes. Depending on registry data, the platform decides which enhancement (object) to execute. The registry provides data about enhancement spots, enhancements, assigned switches and switch setting. For each technology, there may be a separate enhancement registry (p.15)
Enhancement Registry Pool	provides a unique access to the different enhancement registry types (p.15)
Enhancement Section	ABAP Keywords defining start and end of a code section with a given implementation that can be replaced by implementing enhancement code. For one enhancement section, only one enhancement implementation is allowed (p.9)
Enhancement Spot	group enhancement elements of the same technology (e.g. BAdI definitions). Enhancement spots are defined by the application provider and can be implemented by the enhancement provider. For some enhancement options such as DDIC structure appends, no enhancement spots exist. For implicit enhancement options there are also no enhancement spots. An enhancement spot is a transportable object (p.11)
Enterprise Business Function	Second generation Business Function that can be switched individually, usually part of an Enhancement Package (p.30)
Enterprise Extension	First generation of a Business Function that can be switched individually (p.30)
Industry Business Function	Business Function that belongs to an industry solution and can only be switched via its Business Function Set (p.30)
Industry Solution	Industry-specific extension of the core providing special functions. Starting from Business Suite 2005, all Industry solutions are shipped with ERP. One of them can be activated in a system using the switch framework, by activating the corresponding Business Function Set (p.24)
Package	Organizational unit that groups ABAP development objects, has defined package interfaces, declares usage dependencies to other packages and that can be nested. A package can be assigned to one switch.
Source Code Plug-In	Piece of ABAP code that is logically injected at a defined location in the source code to be enhanced. The location can be defined by an explicit or implicit enhancement option. The source code plug-in is an enhancement implementation that is stored and transported separately (p.9)
Switch	A repository object containing a state variable that determines whether an associated switchable development object is effective or inoperative. Switches are not individually operated by the user but assigned to one or more Business Functions that control the state of assigned switches (p.25)
Switch BC Set	A Business Configuration Set, containing table-like data, such as customizing settings, that is switched via package assignment (p.26)