



**Atom-Based Software Engineering**

# ABSE and Software Engineering

## ABSE and Software Engineering

This section is devoted to describe several aspects of software engineering and how they relate with ABSE. Conversely, you'll learn how ABSE aligns with or supports software engineering techniques or approaches.

ABSE is an universal modeling approach that can support and combine most aspects of software engineering.

---

### Model-Driven Development

Model-Driven Development (MDD) is also known as Model-Driven Software Development (MDSD) or Model-Driven Engineering (MDE).

If this is your first contact with Model-Driven Development, or if you have been recently introduced to it, you are strongly advised to learn about these techniques. ABSE uses new methodologies and concepts that may not be familiar to you. By learning about MDD, you build a solid foundation on the principles that drive ABSE.

---

### Models, Metamodels and Meta-metamodels

In computer terms, when we create a "thing" that describes another "thing", we add the term *meta*. For example, when we describe data, we are creating *metadata*.

Similarly in ABSE, [Atom Instances](#) make up your model, so they are *models*. There are many different models in a project.

To describe how Atom Instances would work and behave, you must describe them through a uniform specification. You do that through a specific form of model, called a *metamodel*. [Atom Templates](#) are ABSE's metamodels.

But there must also be a specification of the general composition and behavior of a metamodel. Like we did in the last step, we must describe it with a uniform spec model. That model is the [Atom](#). The Atom is ABSE's *meta-metamodel*.

In a sentence, the meta-metamodel describes how metamodels define your models that will be translated into your code.

---

### Domains

A domain is a set of common requirements, terminology, and functionality for any software program constructed to solve a problem in that field.

If that field will tell you *how* to solve a problem, we can further call it a **solution domain**.  
If that field will tell you *what* problem to solve, we can further call it a **problem domain**.

Examples of solution domains : Database, Java, AJAX, .NET  
Examples of problem domains : ERP, CRM, Automotive, Traffic Control

With ABSE, you can work on any domain belonging to the problem or solution spaces. Atom Libraries are closer to a domain concept. You will create solution-domain Atom Libraries, and then create problem-domain Atom Libraries using Libraries from the solution domain.

---

## Model Transformation

In other MDSD approaches, you'll find the notion of model transformation. For instance, the MDA approach allows you to create a Platform-Independent Model (PIM) in the problem domain, that can result in one or more Platform-Specific Models (PSMs) in the solution domain, through model transformation.

This mechanism is not needed in ABSE because you can use Atom Inheritance: Problem domain Atoms can be composed of or inherit solution domain Atoms.

The only model transformation you need to care of in ABSE is the one that transforms your project's Model into generated artifacts. To achieve this you need to write Lua programs in one or more of the Atom Template's Create, Pre, Exec or Post sections.

---

## Domain-Specific Modeling

Domain-Specific Modeling (DSM) uses graphical models (also called a graphical Domain-Specific Language) to abstract notions, objects and other aspects of a particular domain. DSM tools also allow code and other artifacts to be directly generated from the model.

ABSE is a DSM-based methodology, so it shares a lot of principles with it. ABSE does not rely on graphical models. It does rely on a tree that can be considered a graph, but AtomWeaver represents and manipulates this tree in a non-graphical way.

You can obtain a useful background in modeling for ABSE if you familiarize yourself with DSM.

---

## Domain-Specific Languages

A Domain-Specific Language (DSL) is a programming or specification language targeted at a particular domain. With a DSL you can express your solution to a given problem using direct and familiar concepts, built directly into the language.

To generate code, ABSE needs a transformation language but makes no assumption on which language you need to use. However, AtomWeaver has been implemented to use a pre-defined language called Lua, which is not a DSL but a general purpose language. You can learn more about Lua at its [official site](#).

With ABSE you don't need DSLs. However, since you can generate code in any language, you can generate programs written in a DSL.

---

## Software Product Lines

The key difference between traditional, single system development and software product line engineering is a fundamental change of focus: from the individual system to the product line. This shift also implies a shift in strategy: from the next-contract vision to a continuous, strategic view of the company's business domain.

Contrary to many traditional reuse approaches that focus on simple code assets, the product line infrastructure encompasses all assets that are relevant throughout the complete software development life-cycle, from the requirements stage over architecture and implementation to testing. This group of assets together defines the *product line infrastructure*. A key difference of software product line engineering from other reuse approaches is that the various assets themselves contain explicit variability.

You can implement a SPL in AtomWeaver by using Atom [Execution Conditions](#). You can create one or more configuration Atoms that will then drive the creation of others. This way you can easily activate or deactivate the inclusion of specific features in your final product.

The individual assets in the product line infrastructure are linked together. For example, traceability is defined among the individual assets, enabling you to take a requirement and identify all related implementation code and test cases. Feature traceability is easy in ABSE: All Atom Instances in a branch belong to the same feature.

The following are the fundamental principles of software product line engineering:

- **Variability management:** Individual systems are considered as variations of a common base system. This variability is explicit and must be continuously managed.
  - **Business-centric:** Software product line engineering thoroughly connects the engineering of the product line with the long-term strategy of the business.
  - **Architecture-centric:** Software must be implemented in a way that allows taking advantage of similarities among the individual systems.
  - **Two-life-cycle approach:** As said, the individual systems are variations of a common base system. These products – as well as the base system – have their individual life-cycles.
- 

## Software Factories

A Software Factory is an organization that develops software systems in a systematic and "industrialized" way. Instances of those systems (or their parts) share features, functionality, and architecture. All phases of the application's life cycle are clearly defined and automated. Building a Software Factory is the ultimate dream of any software vendor.

The term *Software Factory* was introduced by R. Bremer of General Electric and M. McIlroy of AT&T in 1968. While Bremer proposed an approach of standardized tools and controls, McIlroy

emphasized the systematical reuse of code when creating new software systems. Both approaches can be seen as the ancestors of the Software Factory concept.

A Software Factory is a thoughtfully selected and packaged collection of core assets and instructions for developing instances of a Software Product Line: applications that have a considerable amount of common features, functionality, and architecture. In addition, a Software Factory defines a development process. Application developers have to follow that process when building product line members. A facilitator of this process can be implemented through contextual guidance.

From the developer's perspective, a Software Factory becomes a part of his development environment that can be used to develop members of the Software Product Line in a more effective and predictable fashion.

To build a Software Factory, you need to organize the software development process and environment around the following four concepts:

- Software Product Line development
- Reusable software assets and extensible architecture frameworks
- Contextual and automated guidance
- Use of higher-level abstractions for code generation (like Model-Driven Development)

ABSE and AtomWeaver provide the above four pillars so that you can start building your own Software Factory. The multiple-tree paradigm of ABSE allows you to model all your processes, from requirements to maintenance. Atoms allow you to organize and automate all these tasks.

---

## Compositional Software Engineering

Compositional Software Engineering aims to minimize process-driven coordination and instead rely on architectural rules, constraints and decoupling mechanisms to maximize the ability of teams to operate independently.

Compositional Software Engineering relies on some clear principles:

- Independent teams can be inside and outside the organization, and are able to release their components frequently, whenever they want
- Teams can set their own road maps
- System architecture focuses on facilitating development by composition
- Customers compose their products by selecting from the available functionality (software families)

ABSE implements a Compositional Software Engineering system.

### Assembling applications

In ABSE, the Atom has the same application and can be used in the same way as real-world atoms: It's the basic construction unit in a project. You can combine several Atoms of different types to "assemble" an application. You can consider an Atom as a building block of your larger systems.

---

## Software Building Blocks

Most of us developers have dreamed of building software applications by just snapping small parts together, just like the Lego bricks we all have played with in our childhood.

However, albeit constantly sought for, this goal has never been quite reached. The extremely broad application of software engineering (almost everything in our lives today) makes it quite difficult to create infinite possibilities from components that have a finite, small number of “connections” or options.

But with the emergence of software components, MDD and DSM, we can narrow those “infinite possibilities” into a manageable “large quantity” of possibilities, making it possible now to model a system that is complex, but within a known domain, using those aforementioned building blocks.

Atom Templates (ABSE’s metamodels) can be made of other Atom Templates, creating larger components. And through Atom dependency constraints, you can define to which other “blocks” can a “block” snap to. These constraints implement model construction guidance.

ABSE can be a pure “snap together” modeling system. It does have its costs, naturally. Metamodels conforming to your domain have to be built and tested. But after having a well-developed metamodel schema in place, building applications with ABSE will be highly productive.

---