Applying Code Generation Approach in Fabrique

Kirill Kalishev, JetBrains

This paper discusses ideas on applying the code generation approach to help the developer to focus on high-level models rather than on routine implementation tasks. This approach is exemplified by persistence level generation performed by JetBrains Fabrique™. An overall overview of Fabrique is also presented here.

Fabrique – a code generator

Briefly, Fabrique is a tool for rapid development of applications with rich client-side web-based interfaces and a database on the backend. Although there are lots of such tools around, all of them represent a big family of "libraries" that provide some additional service but force you to conform to some rules.

For instance, the EJB specification promises to take care of managing persistence for your objects but requires you to follow rules for describing that with code and deployment descriptors. On top of that, different EJB vendors have their own specifics.

The point is that after we choose a platform like EJB, we become dependent on it. That kind of dependency may last for years regardless of whether we are still happy with it. Eventually we may come to regret the decision to go with EJB at all, but after months (or years) of development it's unlikely that we would devote time and resources to switch from EJB to, say, Hibernate. Even if we did, we're still in the same boat. Who says that we will be happier with the new platform, or if we are now, for how long?

However, there is good news. As in our example with persistence, and in many other cases, an actual layer between our code and a target library is pretty formal and can be created automatically. There is no actual need for a human to do this, since following specifications is a routine task. So why not describe *what we want* with some high-level and platform-neutral language and then let a computer to do the rest? This is how Fabrique is designed.

The main idea of Fabrique is that when the creation of some code becomes a formal routine, it gets generated. The developer only needs to create high-level business models built with a mix of Java and Fabrique- specific languages, and choose which target platform to deploy them to. In this way, Fabrique helps developers focus *more on real business solutions* and *less on technologies* that implement them.

The picture below shows this basic principle:

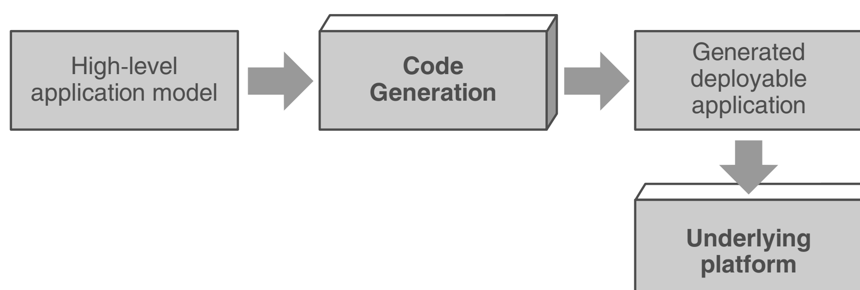

Figure 1: Basic principle of code generation

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

With this approach we are free to decide on an actual platform now or later. Moreover, the right to change our mind is always reserved for us. If we start with EJB and later want to switch Hibernate, we just change the code generator. The application level model does not know anything about the actual platform, so it doesn't require being touched during this process.

Of course, things are not quite *that* simple. Even with this approach we have a couple of major issues.

First, generated code is likely to be used in manually written code. It's just not possible to remove such dependency since not everything is so formal and routine that it can be generated.

For example, if we have lots of generated EJB classes, we are supposed to use them somehow, right? But what we should depend on from client code? If we depend on generated EJB classes, there's not so much fun with this generative approach. Yes, it's cool that somebody generated those classes for us but the client code is still locked into EJB. So the question is, how can high-level code from an application model use its generated counterparts? If we change the target platform, what happens to our business code that uses some code generated earlier for the previous platform?

The next issue is even tougher: how do we perform refactorings when both generated and hand-written code are affected?

Fortunately, both these issues are solvable. For the solution we have to introduce some more players:

- Generated code should implement some *platform-independent interfaces* on which hand-written code can safely depend.
- Make refactoring a problem for a *refactoring tool* rather than the user.

So it's time to have a more detailed look at Fabrique's design and major component parts to see how they solve these issues. Refer to the following diagram.

Visual Fabrique – an IDE for creating, editing, and refactoring the Fabrique Application Model. Visual Fabrique is based on IntelliJ IDEA™. In addition to IDEA's intelligent capabilities for Java code editing, Fabrique provides visual editors for model languages. Of course, it fully supports specific its own languages with code search across the project, code completion, error highlighting. etc.

Fabrique Application Model – a high-level platform-independent description of a Fabrique Application. It's described with various declarative as well as imperative languages (which we will consider further later on).

Fabrique Compiler – a code generator. It generates all necessary Java source code ready for compiling into a deployable application, and runs the Java compiler for those sources. The generator always generates two kinds of source code:

- Platform-independent code which is safe to use from the application model
- Platform-dependent code which extends the above, and is invisible to the application developer

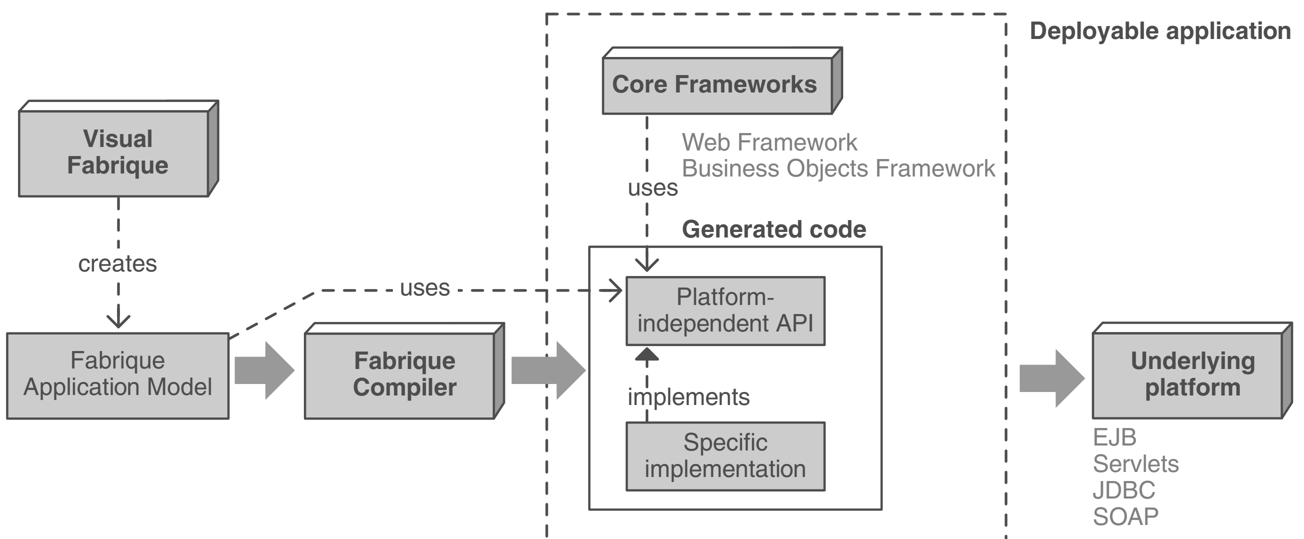

Figure 2: Fabrique code generation architecture

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

Fabrique Core Frameworks – contains runtime code which drives the built and deployed Fabrique application. It contains the following parts:

- *Web Framework* – a collection of ready-for-use web controls and an event system
- *Business Objects Framework* – a set of technologies for transaction and persistence management as well as the application's business logic

So the two extra players mentioned previously are:

Platform-independent API, generated irrespective of any platform so that hand-written code may depend on that API rather than on something that changes when we change a code generator (and the respective platform)

Visual Fabrique, which handles all refactoring issues including changes affecting both hand-written code and its generated counterpart

Fabrique Application Model

The point of application model is that it's a platform-independent description of an application. A Fabrique application model can consist of:

- *FabScript* – a simple Java-based language for describing business logic. FabScript is designed so that scripts are compact for easy inlining to other parts of the model.
- *Java code* – can also be used for describing business logic, but cannot be inlined to the model
- *FabQL* - a query language for Business Object Model. FabQL is an extension of EJB QL with some improvements that make it simpler in use in Fabrique
- *Business Object Model* - describes persistent objects, their attributes, methods and relationships. Business Object Model is described in an XML-based language where the following languages are inlined:
  - FabScript is used for describing bodies of methods
  - FabQL is used for defining finder methods for business objects.
- *Web Page Model* – describes the application's web pages, their structure and reactions to events. FabScript can be inlined in many places on the page.
- *Business Service Model* – describes business logic, which is exposed via a global access point.

It's the job of Fabrique Compiler to generate all of the above into compliable Java code, to run the Java compiler against that generated code, and to prepare a deployable application.

Active Libraries

Needless to say, extensibility of an application is an important thing. In Fabrique there's a notion of Active Library, which is the way to contribute to Fabrique. We can contribute lots of things: from the runtime appearance of controls to a new editor for Visual Fabrique. An independent vendor can, for example, provide a new language for defining whatever it wants, a visual editor for that language, and a code generator to make it sensible at runtime.

Separation of development aspects

So, the whole point of Fabrique is clear separation of several development aspects:

- High-level description of an application (Fabrique Model). You don't have to decide on a platform before development, and you are free to change platforms later.
- Visual editing for Fabrique Model (with Visual Fabrique). Since Fabrique Model is a high-level description, the editors have the ability to be more intelligent because they know about much more than just plain Java classes.
- A pluggable code generator that generates target platform-dependent implementations of things described in Fabrique Model (Fabrique Compiler). We only need to select and easy configure the code generator for the platform we want our application to run on.
- A runtime library (Core Frameworks) that handles all difficult, routine (and boring!) tasks such as web controls rendering and behavior, transaction management, and so forth.

The rest of this article is devoted to an overview of Fabrique's Business Object Framework (BOF). With this example I will try to show benefits of this separation of development aspects as well as the code generation approach in general.

Business Objects Framework showcase

Here we proceed with a simple showcase to highlight cool features of code generation.

Building the model

With Visual Fabrique we draw a simple diagram.

Here we defined two business objects: Project and Developer, and a relationship between them that says

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

for each project there can be zero or many developers and each developer always participates in one project.

For each business object there is a predefined finder method Find All () (which we call “query”) that returns all objects for a given type. In Visual Fabrique we can simply add a new query for Developer that would return all developers who participate in a given project. We code this in FabQL.

Then, we add some business logic to the Developer object by writing a method in FabScript that would say whether a given developer participates in a specified project.

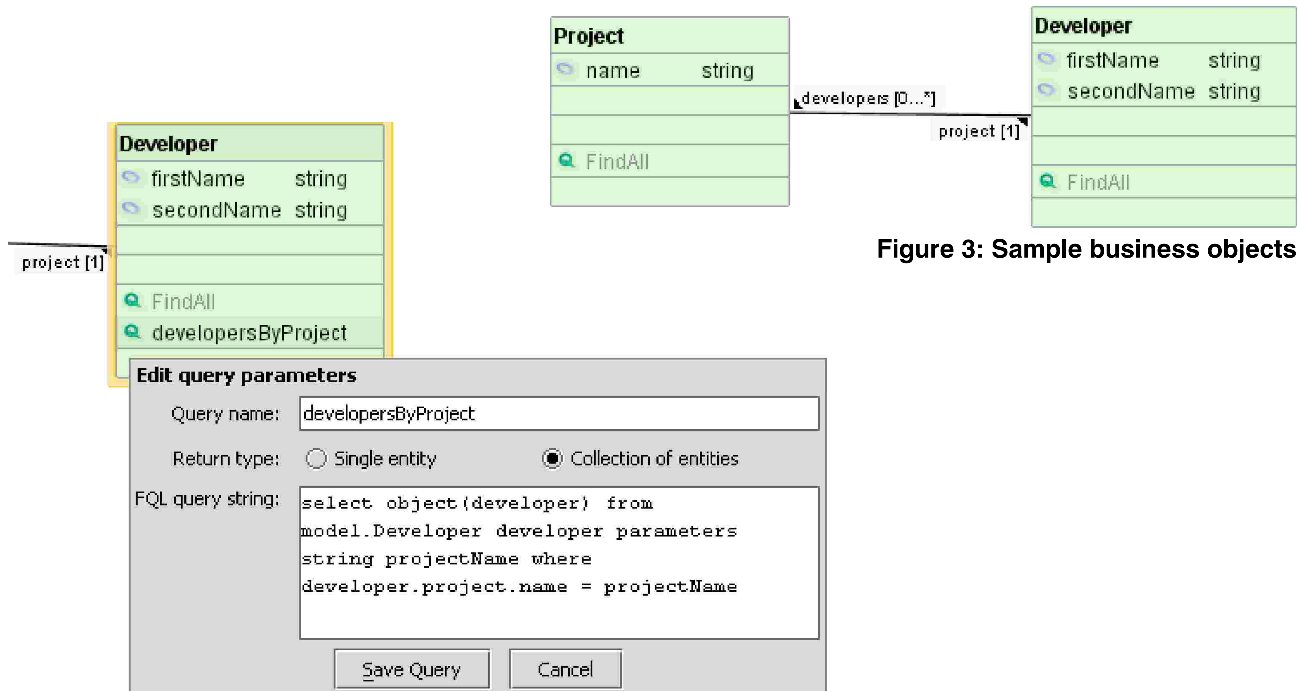

Figure 3: Sample business objects

Figure 4: Specification of a query method

Figure 5: Specification of an entity method

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

Running the Code Generator

Up to this point we have been modifying Business Object Model. Now it's time to use it. We can now run Fabrique Compiler to generate an actual implementation for the persistence level. But first, we should decide on target platform. There're two options currently available:

- EJB CMP container (WebLogic/JBoss/Orion)
- Hibernate (JBoss/Orion/Tomcat)

In either case, the complier will generate all necessary classes and deployment descriptors and prepare them for deployment. However, regardless of the target platform, there is always part which is the same (what we called "Platform independent code" above). For each business object, two interfaces are generated:

1. *Business Object interface* – with methods that expose the object's attributes, relationships and business methods. The name of the interface is the same as the name of the business object. Here is such an interface from our sample, saved in a file Developer.java.
2. *Business Object Manager interface* – with methods that manage the object's lifecycle and perform fetching of objects by means of the queries we created. For our sample, here is a corresponding manager for the Developer object saved in a file DeveloperManager.java.

It's safe to depend on these two interfaces from any part of the model. We will consider use cases in the next chapter.

So, what has the compiler done for us? If we run the complier against the EJB/Orion configuration the following things will be generated:

- Implementations of local, remote and home EJB interfaces
- All necessary deployment descriptors

```
package model;

public interface Developer
➥ extends jetbrains.fabrique.bof.rt.BusinessObject {
   java.lang.Long getPrimaryKey();
   java.lang.String getFirstName();
   void setFirstName(java.lang.String firstName);
   java.lang.String getSecondName();
   void setSecondName(java.lang.String secondName);
   model.Project getProject();
   void setProject(model.Project project);
   public boolean isParticipant(java.lang.String projectName);
}
```

```
package model;

public interface DeveloperManager extends
jetbrains.fabrique.bof.rt.BusinessObjectManager {

public model.Developer create();
public model.Developer create(jetbrains.fabrique.bof.rt.UnitOfWork
➥ unitOfWork);
public model.Developer find(java.lang.Long primaryKey);
public java.util.Collection findAll();
public java.util.Collection findDevelopersByProject(java.lang.String
➥ projectName);
}
```

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

- All necessary code that provides a bridge between Fabrique's Business Object framework and the EJB container

If we run the compiler against the Hibernate/Tomcat configuration it will result in the following output:

- Implementations for Hibernate beans
- Hibernate mapping model
- All necessary code that provides a bridge between Fabrique's Business Object framework and Hibernate

Usage from hand-written code

Usage of business objects is very simple. Inside Fabrique Model you can be sure that there's always a visible variable for each manager object. You don't care how it's initialized, you just write something like:

```
Collection fabriquePeople =
➥ developerManager.findDevelopersBy
➥ Project("Fabrique");
```

Coding assistance

Complete coding assistance is provided by Visual Fabrique:

The cool thing here is that you work with interfaces that have nothing to do with the target persistence platform. You don't have to write redundant casts when working with business objects and you don't see any unrelated methods there since their interfaces are completely generated.

Refactoring it all together

One of the reasons people are conservative about applying code generation is the fact that generated code is not easy to maintain. What happens, if I want to change something, say – to rename a business object?

Fabrique was designed with the idea that you should never have to hack generated code. If you want to rename something, Visual Fabrique will take care of renaming all references in your model and then regenerate the affected code.

On-the-fly generation

Visual Fabrique doesn't have to run the compiler to provide assistance such as code completion. To tell the truth, it does run it, but only to generate platform-independent code, which is a pretty quick thing. You don't sit around waiting for Visual Fabrique to regenerate

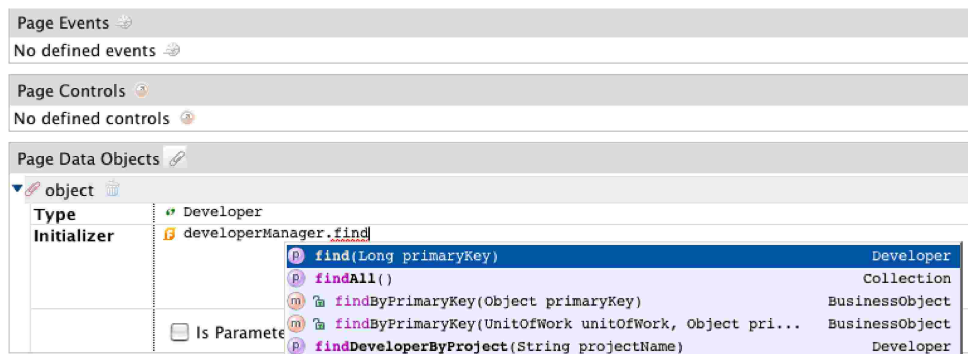

Figure 6: Code completion

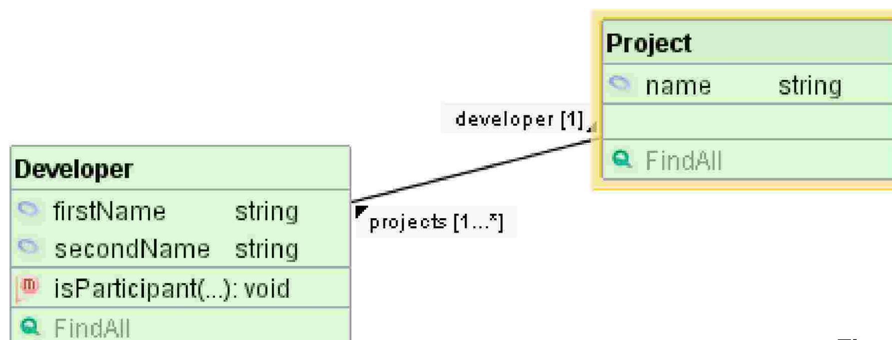

Figure 7: Find usages

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

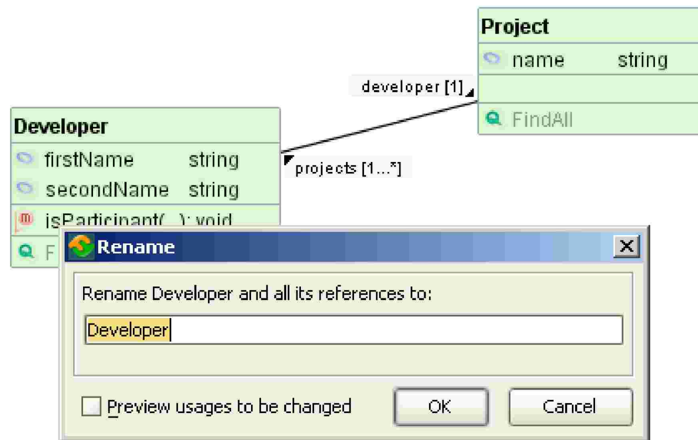

Figure 8: Rename refactoring

all the code on every change. Full code generation happens only when it's time to deploy. Even then, Visual Fabrique tries to make it as incremental and time-efficient as possible

Managing transactions

In Fabrique, there is something more than just making development abstract from concrete persistence platform. Fabrique provides a platform-independent implementation for transaction management specifically designed for web applications. This simply means that some typical work that web developers do has already been done for them.

For example, each web page instance always lives inside a transaction. Any changes to business objects will be local to the transaction until the transaction gets closed. Here is how it's done.

Logic inside the web page always works with copies of business objects. Managers of business objects are accessible as we described above, and we can be sure that each time we use, say, the "developerManager" variable, it points to a local copy of the DeveloperManager object, and all objects the manager creates are local for the transaction.

As the page gets closed, all local objects are either dropped or saved to the database. It's managed by a simple API that the logic inside the page can use:

- markForCommit() – when the transaction gets closed, commit local data to database
- markForRollback() – when the transaction gets closed, drop all local data

In addition, it's possible to create nested transactions and either commit or roll them back as a whole.

So, lots of typical work is already done, but if we want something specific like the creation of nested transactions

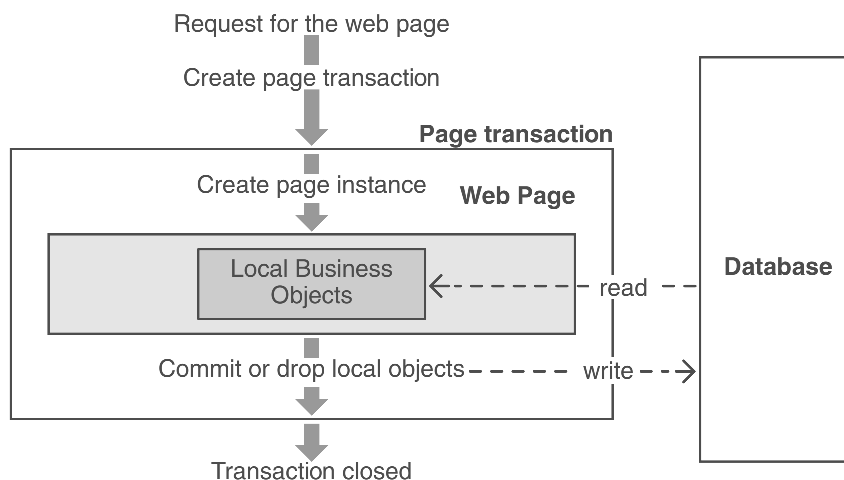

Figure 9: Web page lifecycle

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com

– a Fabrique transaction API is open to us. This is very simple. In Fabrique, almost in every place where you write FabScript, there is appropriate set of context variables. In a web page, among others (like “developerManager”, see above), there’s one called “transaction”.

The cool thing about Fabrique transactions is that they're platform independent like the things we considered before. The rules are the same – we don’t have to decide on a platform, and we let the complier generate all the implementation for us.

Wrap up

Here is a quick summary of neat things implemented with Business Object Framework:

- We can create the business model with visual editors – no need to specify any xml-deployment descriptors or implement any interfaces (or any other boring things!)
- Fabrique provides support for transactions which is extremely easy to use .
- Fabrique Complier generates platform-independent interfaces for business objects which are strictly typed and easy to use, and also specific implementations for a target platform.
- Visual Fabrique provides robust, intelligent coding assistance for those times when you do have to write some code.
- We can alter the complier settings so that it generates implementations for different specifications and application server vendors with no need to change the business model

Libraries vs. Generators

I think I've clearly outlined how Fabrique is different from typical web development platforms in general, but let me point out some specific things to make it even clearer. We can consider EJB CMP vendors or Hibernate to be typical representatives of such families of tools. So, what do we mean?

Libraries take care of some difficult things and provide a business-task oriented API (thus hopefully making programming simpler). However, we still have the following issues:

- Libraries don't change with the way we express our tasks. Even if we deal with a high-level API, we have to write code in the same low-level language which may not be well suited for expressing higher level things.
- A code editor cannot provide additional assistance to help the developer manage high level things (it still only knows about classes/packages/methods and has no idea that you use a library that can do something with persistence or user interface).
- It's very hard to build a library that would balance well between ease of use and functionality/flexibility. Type systems of most programming languages are not flexible enough to seamlessly integrate with libraries.
- Program becomes dependant on a library and, more importantly, the technology behind it.

Code generators, like Fabrique, ask us to write a task with some higher-level language and then generate the lower-level code. Having a high-level description of a task results in a number of benefits:

- it's simpler to read, understand and modify
- we are not stuck with only one code generator; we can choose/write/tune generators however we want
- we can generate new data types instead of using a generic mechanism
- we can make our IDE more intelligent and assistive to the developer since we know about high-level constructs, not just classes/packages/methods

As we outlined in the first chapter, there’s still a lot of work to address problems of communicating between hand-written code and generated counterparts and refactoring of the whole project. But much of this work can be delegated to an intelligent IDE for the user’s convenience. This is what Visual Fabrique does.

Fabrique status

Fabrique is now is open in the JetBrains Early Access Program (EAP) and can be downloaded and tried for free. It’s still in development and your feedback and participation in the EAP will make it a better product. Please visit http://www.jetbrains.com/fabrique for more information about Fabrique and participation in the EAP.

Copyright © 2004 JetBrains s.r.o. All rights reserved.
For permission to copy or redistribute this article please contact: editors@jetbrains.com