Introduction to Generative Computing

Look at the resource section in the overview material for links etc.

Introduction to Generative Computing

Goals

1. **Learn how and when to use generative technologies.**
2. **Lern to build code generators.**
3. **Learn about model, meta-model and code.**

What is Generative Computing?

Generative Programming is about manufacturing software products out of components in an automated way, that is, the way other industries have been producing mechanical, electronic, and other goods for decades. The transition to automated manufacturing in software requires two steps. First, we need to move our focus from engineering single systems to engineering families of systems - this will allow us to come up with the "right" implementation components. Second we need to automate the assembly of the implementation components using generators.

(James Coplien in the forword to "Generative Programming by Eisenecker/Czarnecki)

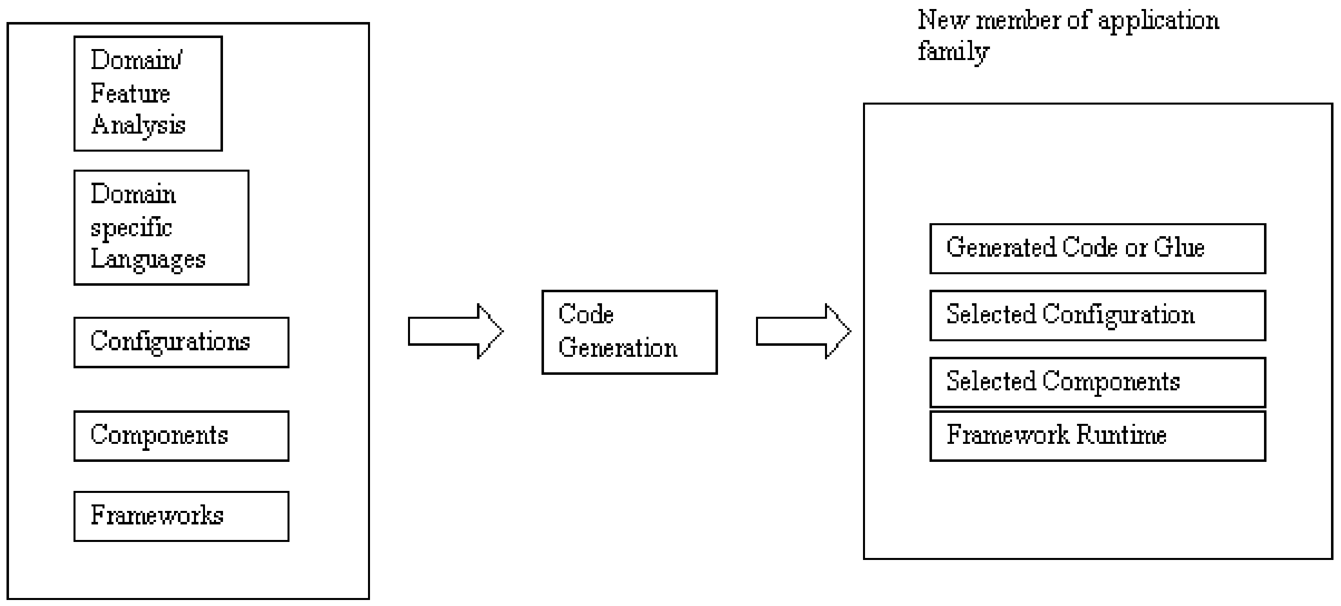
Technologies comprising "Generative Computing"

1. Domain Engineering: finding commonalities and variations in software families. Defining hot/cold spots of software families. The result is configuration know-how for a business domain.
2. Metaprogramming: building programs with programs. Includes meta-modelling, reflection and other techniques.
3. Code generators: Building engines which take templates, models and configuratins and produce some output.

Jim mentions in his forword something “ dedidedly unobject-oriented behind generative computing ”. He even sees a time “ beyond objects ” coming up. There could be some truth to it because current development paradigms show a multitude of technologies mixed (look at the J2EE web programming and EJB model which combines templates, scripts, generated database code etc.). Even more, some concepts like components do not have a representation in OO languages at all. And what can we say about all the meta-data now held in XML formats outside OO-languages?

How generative computing works

How generative computing works (Continued)



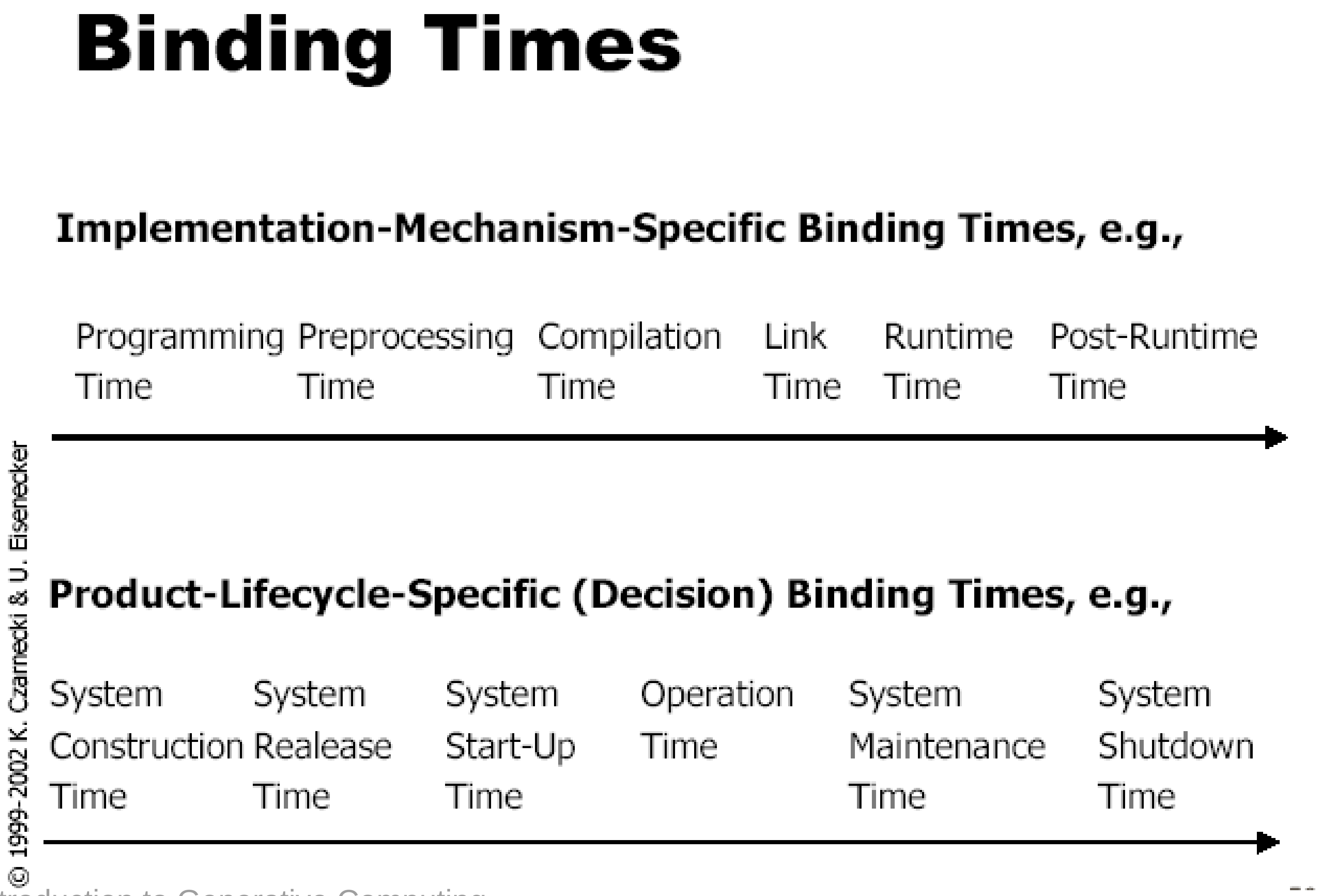
How generative computing works (Continued)

It is important to notice that generative computing includes much more than just code generation. The most important part of it - finding commonalities and variations - is actually completely independent of code generation.

Generation phases and binding

Generative techniques can be used at different times in the development process.

Generation phases and binding (Continued)



Binding Times

Implementation-Mechanism-Specific Binding Times, e.g.,

Programming Time → Preprocessing Time → Compilation Time → Link Time → Runtime Time → Post-Runtime Time

Product-Lifecycle-Specific (Decision) Binding Times, e.g.,

System Construction Time → System Realease Time → System Start-Up Time → Operation Time → System Maintenance Time → System Shutdown Time

Generation phases and binding (Continued)

Static vs. dynamic approaches

Reflection is a highly dynamic approach for flexibility - at the price of high complexity and slower runtime performance. Successful generative computing approaches try to push the decision making phase into the compile phase. The generated code is highly specialized for one purpose but runs fast. If the purpose changes the code needs to be re-generated.

Extremely powerful and flexible systems try to make the model information even available to the program at runtime.

Generic vs. Generative Computing

Generic computing means to treat different things the same way. E.g. by using an abstract framework type interface one can handle many differnt sub-types in a program.

Generative usually means that something specialized is created which usually reduces flexibility at runtime - but that in those case the flexibility is not needed because the generated code has been customized to fit exactly.

Generic types in programming languages like C++ and now since 1.5 also in Java seem to me to include both elements. The way they are defined looks generic to me. The way they are implemented and used seems to be generative. We will take a closer look at Java generics.

Stumbling into generative programming: A document

Frequently softwarec companies almost accidentially end up using generative techniques, driven by the need to create different versions for different customers or by the sheer size of their - perhaps international - applications which are harder to develop with every release.

A company has created an application for forms processing - both electronic and paper based. Customers are banks, insurances etc.

The appliation needs to become configurable. A configuration file is created which allows some processing steps to be parameterized per customer.

The configuration file grows and soon contains many different things.

The software is modularized and processing functions are now tied to configuration items. The validity of the forms processing workflow is now dependent on the validity of the configuration file

The configuration file is parsed at application startup. Errors in the file are extremely hard to find. Parsing is hard coded and no explicit grammar exists.

The system grows and needs to support different scanning hardware. This is done by changing and adapting the configuration file. A customer with the same workflow but a different scanner gets a different configuration file which was adapted using copy and paste techniques.

The company realizes that they cannot perform in field updates to the software because all

Stumbling into generative programming: A document

configuration files are different and contain both company defaults and individual customizations done by customers.

The product sells well and creates a big problem exactly because it sells well: Every new installation at a customer site needs service and support which is increasingly difficult to provide. Just thinking about new releases causes nightmares for the management. The service team grows...

This is the moment when the CTO starts planning a re-engineered product. Goals are not only a better servicable and maintainable software product but also to support new business areas, e.g. document processing.

Several types of analysis are performed: the new domain is analysed with respect to commonalities and variations, core features etc. This results in a new business conceptual model which is expressed in a switch in terminology from "form" to "document". From this, hot spots and cold spots are derived and turned into requirements for the new software.

A software internal analysis uncovers weak spots (e.g. no grammar for config file). The central configuration file is cut into pieces reflecting different aspects of the software: workflow (clearing functions), document structures, hardware. SGML is used to describe the structure of the configuration file. SGML dtd's turn out to be too weak to express the semantics.

processing framework (Continued)

OO-based modularization of the software results in a CORBA based framework which dynamically loads all necessary classes at startup. The software is now structured in core/branch/customer specific areas which are both reflected in the source code control system as well as in the configuration files. The configuration system slowly turns into a repository for implementation classes and workflow commands.

Framework classes are full with system internal special coding/naming conventions etc. A tool is created to generate class templates for the different software domains.

The software is ported from OS/2 to Unix and NT. Many adaptions are done by using the C++ preprocessor to generate the platform dependent code.

More and more meta-data are extracted from the software and put into the meta-data layer. Slowly developers understand that extracting those data is also a step toward description and abstraction. There is no need that the syntax of the meta-data needs to resemble C++.

At this point the following ideas and problems come up:

1. The software is written in C++ which has almost no runtime type information. After the experience with some simple forms of code generation the development teams thinks about generating an extension to the C++ system to provide runtime type information.
2. The developers realize that they have a complete model of the documents but they don't use it to generate the database structures from it. Instead, the whole information is replicated and the tables are manually created. This results in possible mismatches

Stumbling into generative programming: A document

between DB and the data layer.

3. While they are at it, the developers realize that they could also generate a lot of the end-user GUI layout from their workflow and document model. And on top of that: a document editor could be turned into a document structure editor as well.
4. It is unclear whether they should put more effort into making the runtime system more dynamic or whether they should generate more customer specific code from the beginning.
5. The software becomes more and more complex. A logging sub-framework is implemented and tied to the class generator. Still, developers need to put in the calls to the logging system manually.
6. The developers realize that their software is composed of many different aspects which are somehow mingled in their code. Wouldn't it be nice to generate logging calls completely automatic? Call parameters and returns are know to the system so why can't it do that? Which again ends in complaining about the lack of meta-programming features in C++.
7. The model and meta-data give reasons for headaches as well: A lot of things are not expressed in meta-models. Source code needs to understand many details of the model. Should the model and meta-model information be also available at runtime or just be used at generation time? Should the configuration turn into a Domain Specific Language?

processing framework (Continued)

This course will try to make the options and problems clear. At the end you should know when to use what kind of generative technologies.

Code Generation

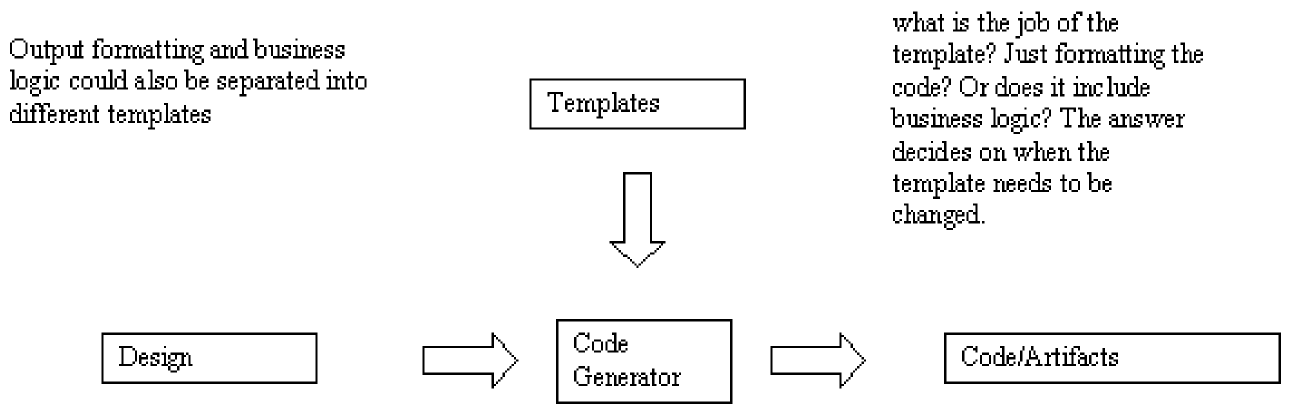
Reasons for code generation

listed with increasing importance:

1. Save time by avoiding repetitious work
2. Improve quality by automating error-prone tasks
3. Further inprove quality by tracking dependencies automatically (e.g. the multi-tier problem)
4. Save and re-use know-how by using descriptive technologies to capture domain knowledge. This conserves know-how across changes in programming languages and technologies and at the same time lets business users use their own language.

Where light is there is shadow as well: increased complexity is one price to pay. Participating developers need to understand and accept the value of abstractions and descriptions. Management needs to understand that progress will take time. For the reasons see also: Jack Herrington, Code-Generation techniques in Java (Resources)

Code Generation Process


Output formatting and business logic could also be separated into different templates
Templates
what is the job of the template? Just formatting the code? Or does it include business logic? The answer decides on when the template needs to be changed.
Design
Code Generator
Code/Artifacts

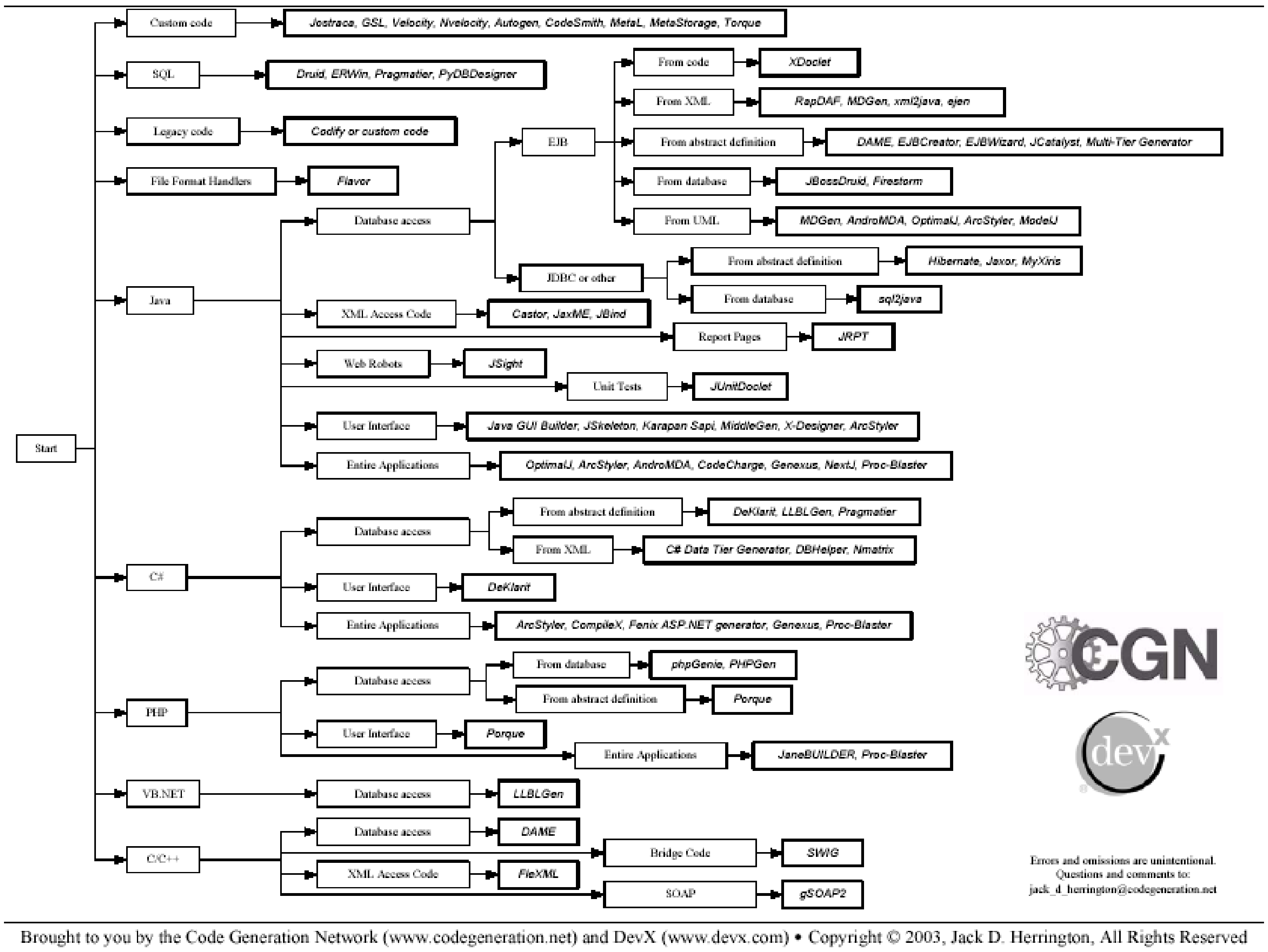
Code Generation Process (Continued)

This is the process code generation frequently uses. (Taken from Harrington, Code generation Techniques...). The reasons for code generation can be much simpler than a generative computing approach. They can be completely confined to IT-internal problems like conversion between different but equivalent syntax etc.

Code Generation Decision Tree

Jack Harrington did assemble an overview of application areas and code generation tools. See http://www.codegeneration.net or his book for more info.

Code Generation Decision Tree (Continued)


Start
Custom code
Jostraca, GSL, Velocity, Nvelocity, Autogen, CodeSmith, MetaL, MetaStorage, Torque
SQL
Druid, ERWin, Pragmatier, PyDBDesigner
Legacy code
Codify or custom code
File Format Handlers
Flavor
Java
Database access
EJB
From code
XDoclet
From XML
RapDAF, MDGen, xml2java, ejen
From abstract definition
DAME, EJBCreator, EJBWizard, JCatalyst, Multi-Tier Generator
From database
JBossDruid, Firestorm
From UML
MDGen, AndroMDA, OptimalJ, ArcStyler, ModelJ
JDBC or other
From abstract definition
Hibernate, Jaxor, MyXiris
From database
sql2java
XML Access Code
Castor, JaxME, JBind
Report Pages
JRPT
Web Robots
JSight
Unit Tests
JUnitDoclet
User Interface
Java GUI Builder, JSkeleton, Karapan Sapi, MiddleGen, X-Designer, ArcStyler
Entire Applications
OptimalJ, ArcStyler, AndroMDA, CodeCharge, Genexus, NextJ, Proc-Blaster
C#
Database access
From abstract definition
DeKlarit, LLBLGen, Pragmatier
From XML
C# Data Tier Generator, DBHelper, Nmatrix
User Interface
DeKlarit
Entire Applications
ArcStyler, CompileX, Fenix ASP.NET generator, Genexus, Proc-Blaster
PHP
Database access
From database
phpGenie, PHPGen
From abstract definition
Porque
User Interface
Porque
Entire Applications
JaneBUILDER, Proc-Blaster
VB.NET
Database access
LLBLGen
C/C++
Database access
DAME
Bridge Code
SWIG
XML Access Code
FleXML
SOAP
gSOAP2
CGN
devx
Errors and omissions are unintentional.
Questions and comments to:
jack_d_herrington@codegeneration.net
Brought to you by the Code Generation Network (www.codegeneration.net) and DevX (www.devx.com) • Copyright © 2003, Jack D. Herrington, All Rights Reserved

Code Generation Decision Tree (Continued)

Examples for code generation

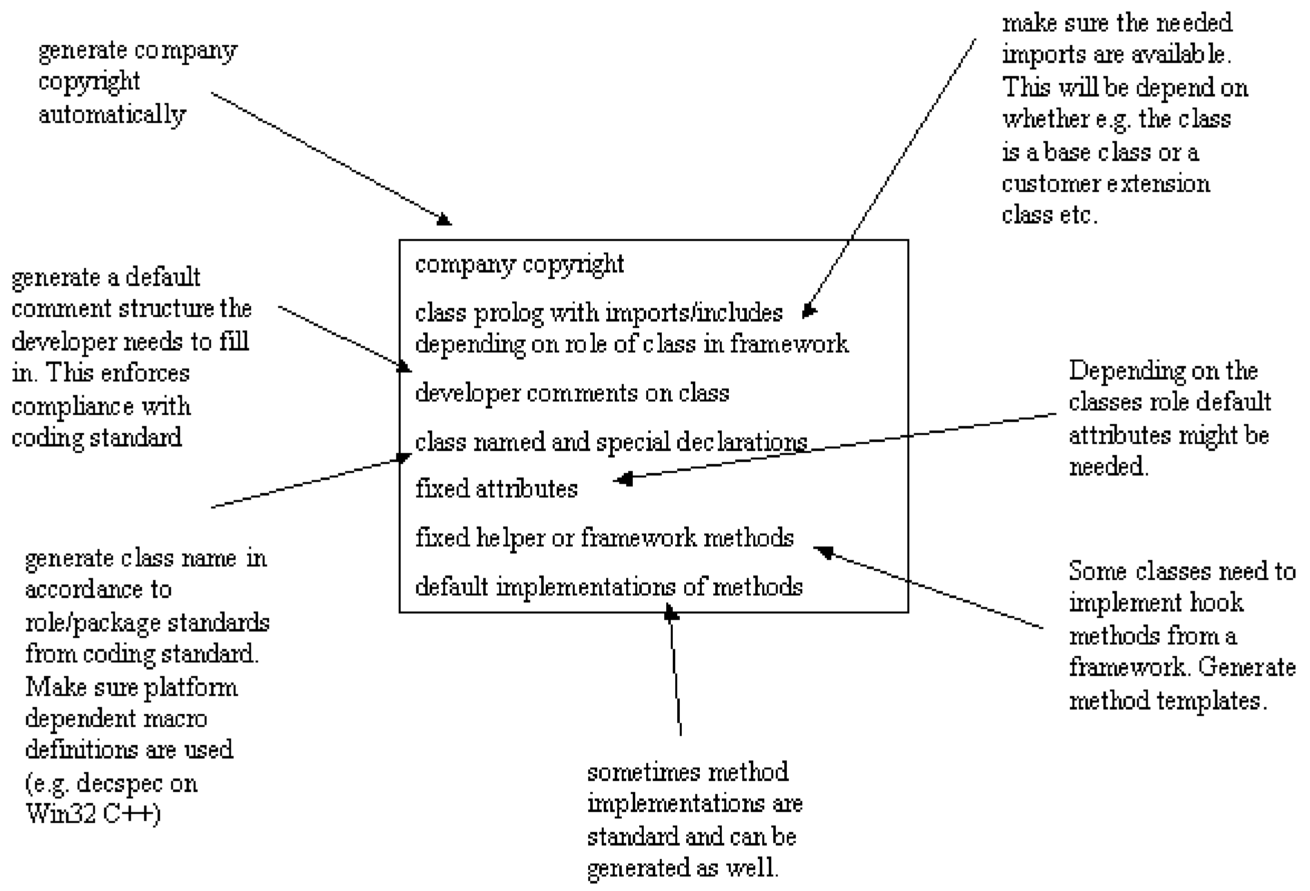
Surprisingly often IT uses different languages to express the same thing. Surprisingly many code pieces are commpletely determined through typed information an can be generated easily. Simple examples for code generation are e.g.:

1. the conversion of type information between different language
2. Marshaling of parameters for distributed computing. Or the generation of automatic proxy/stub code
3. The generation of API documentation from source code. This keeps doc and source in sync.
4. Generation of startup-code, e.g. a template for a class definition which follows a complicated coding standard used in a framework (naming conventions, helper functions etc.)

A simple class template

This example shows how much code in a framework design can be generated. Imagine how much easier it is for a developer to fill in such a template instead of memorizing all the rules from architecture standards and team coding standards. The time savings are enormous as has been proved in "Frameworking" (see Resources)

A simple class template (Continued)

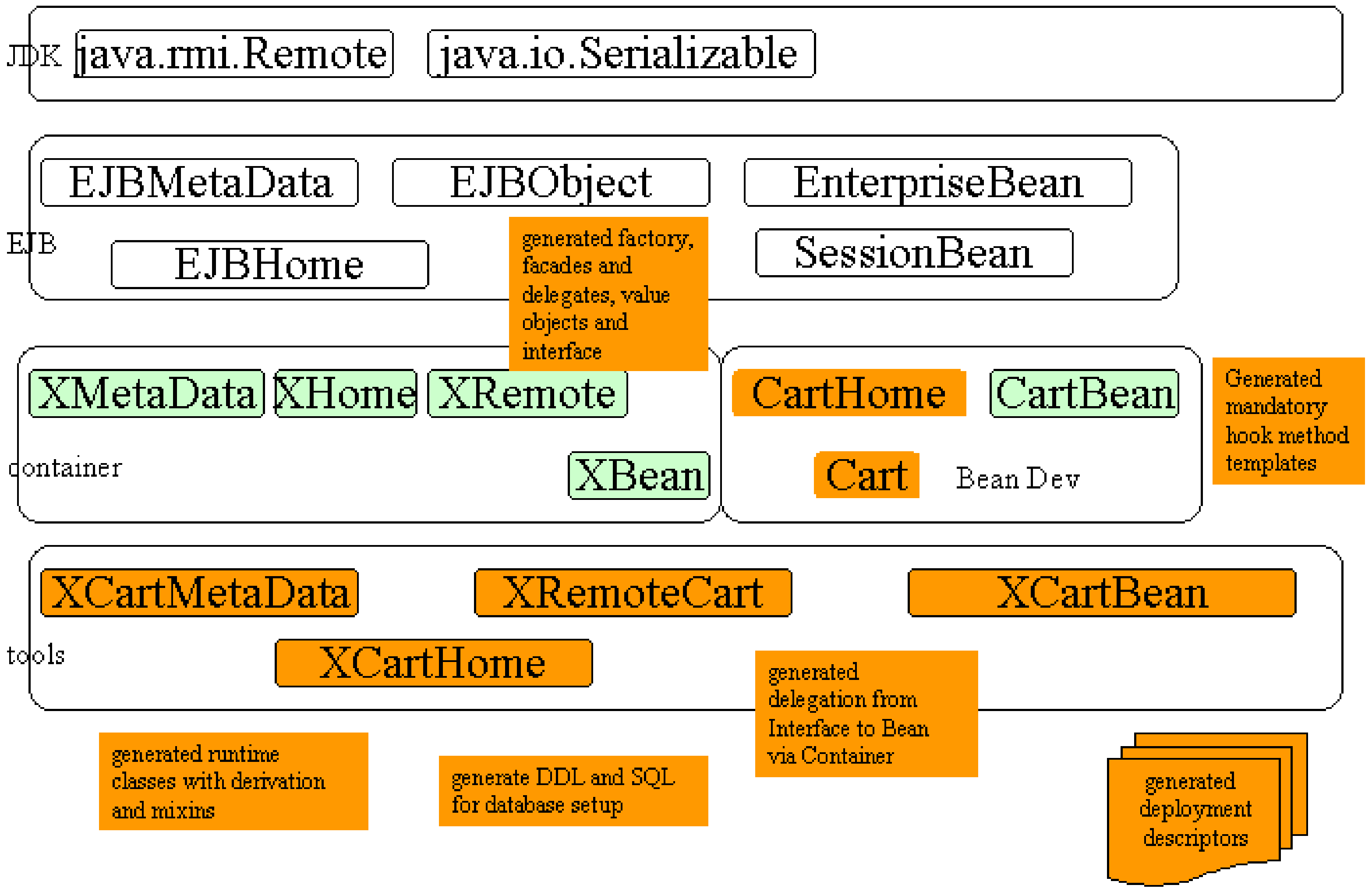

generate company copyright automatically
make sure the needed imports are available. This will be depend on whether e.g. the class is a base class or a customer extension class etc.
company copyright
class prolog with imports/includes depending on role of class in framework
developer comments on class
class named and special declarations
fixed attributes
fixed helper or framework methods
default implementations of methods
generate a default comment structure the developer needs to fill in. This enforces compliance with coding standard
Depending on the classes role default attributes might be needed.
generate class name in accordance to role/package standards from coding standard. Make sure platform dependent macro definitions are used (e.g. decspec on Win32 C++)
Some classes need to implement hook methods from a framework. Generate method templates.
sometimes method implementations are standard and can be generated as well.

A simple class template (Continued)

Generating Glue Code for Frameworks

This example shows how much code can be generated in modern container (application server) architectures. We will investigate this further with XDoclet. The same is true for .NET development.

Generating Glue Code for Frameworks (Continued)


JDK
java.rmi.Remote
java.io.Serializable
EJB
EJBMetaData
EJBObject
EnterpriseBean
EJBHome
SessionBean
generated factory, facades and delegates, value objects and interface
container
XMetaData
XHome
XRemote
XBean
CartHome
CartBean
Cart
Bean Dev
Generated mandatory hook method templates
tools
XCartMetaData
XRemoteCart
XCartBean
XCartHome
generated delegation from Interface to Bean via Container
generated runtime classes with derivation and mixins
generate DDL and SQL for database setup
generated deployment descriptors

Generating Glue Code for Frameworks (Continued)

Active vs. passive Generators

A passive generator (Wizard) can be used only once to create some artifact. When the generation is done the developer takes over the result and improves/completes it. The class template example from above represents passive generation. There is usual no real model behind passive generation.

An active generator is e.g. a compiler. It is a permanent part of the development process, takes a model and transforms it into a different artifact. When the model changes the compiler needs to run again. Typical for active generators is that the generated code is not changes by the developer. Or an elaborate system is put in place to distinguish generated code from hand-written code (checksums, at-generated tags in EMF). Do not use a passive generator when the artifacts need to change after model changes or you will drive your developers nuts.

Behind all this lies the ugly problem of "roundtripping". Is it possible to re-import changes made by developers after generation in every case? What does this mean for the level of abstraction between model and generated artifacts? Doesn't this imply that only a transformation into a different but equivalent syntax happened which works basically at the same abstraction level?

Code driven vs. Model driven Generators

A code driven generator contains both code and generation commands. Typical examples are Javadoc or XDoclets. The advantage is that this type of generator is simple. The disadvantage is that commands are tied to a specific language used e.g. in the code pieces. This make platform-independence impossible.

Model-driven generators keep a separate model which drives generation. Therefore artifact in different programming languages or runtime platforms (J2EE and .NET e.g.) can be generated.

Example of code-driven generation

This is an example from the XDoclet project which uses code attribution (that's why they also call this attribute oriented programming) to generate EJB helper classes and functions.

```
/**
 * This is the Account entity bean. It is an example of how to use the
 * EJBDoclet tags.
 *
 * @see Customer
 *
 * @ejb.bean
 *     name="bank/Account"
 *     type="CMP"
 *     jndi-name="ejb/bank/Account"
 *     local-jndi-name="ejb/bank/LocalAccount"
 *     primkey-field="id"
 *
 * @ejb.finder
 *     signature="java.util.Collection findAll()"
 *     unchecked="true"
 *
 * @ejb.transaction
 *     type="Required"
 *
 * @ejb.interface
 *     remote-class="test.interfaces.Account"
```

Example of code-driven generation (Continued)

```
 *
 * @ejb.value-object
 *     match="*"
 *
 * @version 1.5
 */
```

XDoclet will use this information to create the proper deployment descriptors etc.

Code Generation vs. Compilation

Is this the same? Very bold Model-driven architecture evangelists use examples from compiler technology to show that MDA is just as possible. They may be overextending the similarities because compilation creates executable instructioins directly while in most cases code generation performs only a transformation into a different - not directly executable - format. Careful practicioners like Harrington differentiate clearly between both.

Is Code also a model?

How come many good programmers seem to be able to write code without e.g. an explicit UML diagram? Let's look at the following piece of code from David Flanagans Thread programming examples:

```
public class Deadlock {
    public static void main(String[] args) {
        // These are the two resource objects we'll try to get locks for
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        // Here's the first thread.  It tries to lock resource1 then resource2
        Thread t1 = new Thread() {
                public void run() {
                    // Lock resource 1
                    synchronized(resource1) {
                              System.out.println("Thread 1: locked resource 1");

                          // Now wait 'till we can get a lock on resource 2
                             synchronized(resource2) {
                                System.out.println("Thread 1: locked resource 2");
                             }
        }
.....
```

The important point is that an experienced programmer (experienced BOTH in Java and transactions) will recognize that the code inside the first synchronized statement

Is Code also a model? (Continued)

(including the second) represents a UNIT OF WORK which is supposed to perform uninterrupted to guarantee the business semantics. The problem is that neither the concept UNIT OF WORK nor the business meaning behind is explicitly visible in the code. An experienced business programmer but new to Java e.g. might not catch the problem behind "synchronized".

So program code represents a model as well - it is just often not made explicit or in a syntax which is not understood by non-specialists.

EJB e.g. decided to make such semantics explicit through declarative statements and hide the imperative aspect from the business programmers.

Imperative vs. declarative Code

Imperative code describes how things should be done. In many cases code generation creates imperative code artifacts. Those artifacts are by neccessity bound to certain runtime platforms or languages and are usually not highly portable.

Declarative Code - in our days mostly represented through XML sytax - does only express WHAT should happen and not HOW. The level of abstraction is certainly higher here. The downside is that while the WHAT is portable it usually needs some additional "HOW" pieces added during the transformation into imperative code or even executable code.

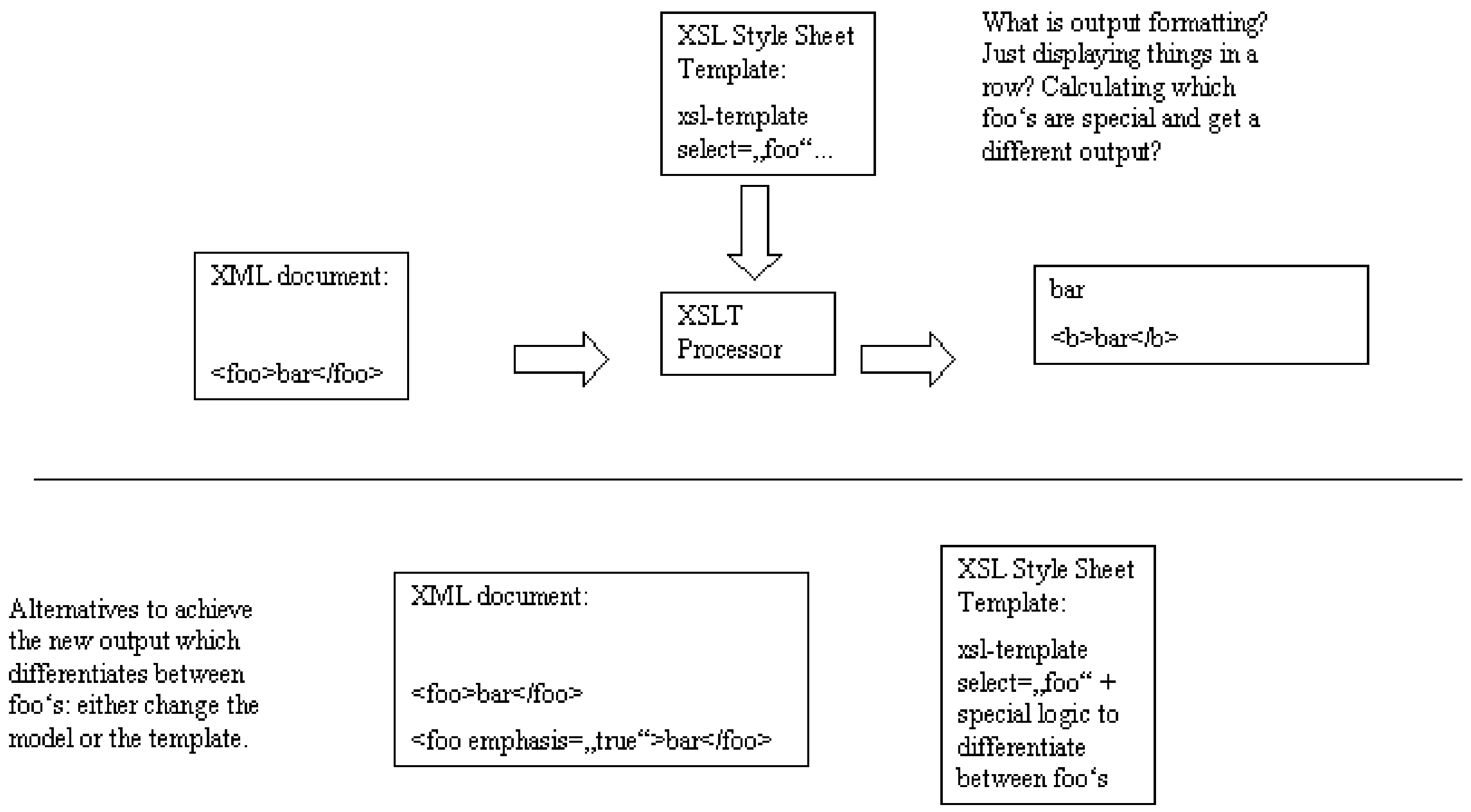
End-to-end vs. Platform Generation

A hotly debated discussion circles around the question whether code generation that tries to generate executable code directly is easier than generating code which needs to run on a specific platform like EJB or .NET. Experienced code generators sometimes claim that end-to-end generation which avoids complex runtime platforms is easier. The reason behind this - surprising - statement could be that once you have created a powerful model and meta-model, adding behavior e.g. transaction support is not so hard anymore if it is done on top of a simple and primitive platform. It gets harder once you have to comply to highly complex runtime platforms like EJB or .NET with your generated code.

Template Problems

Many code generation mechanisms use some form of template. Templates are also used in XSL transformations of XML into different output formats. The mechanism shows the same problems like any other template based generation: How much business logic should go into the templates driving the output formatting? A closer look at XSL/XSLT processing shows that the much claimed separation between content and presentation in XML is actually an illusion. Take this example: Certain output elements need a different formatting but there is nothing in the model to drive the translation process. Now you have two choices: either fix it in the stylesheet - thus including business logic there - or fix the model to include a new element or attribute that can drive the transformation. I guess now I understand why there is a thing like processing instructions in XML/SGML (;-).

Template Problems (Continued)


XSL Style Sheet Template:
xsl-template select=„foo“...
What is output formatting? Just displaying things in a row? Calculating which foo‘s are special and get a different output?
XML document:
<foo>bar</foo>
XSLT Processor
bar
<b>bar</b>
Alternatives to achieve the new output which differentiates between foo‘s: either change the model or the template.
XML document:
<foo>bar</foo>
<foo emphasis=„true“>bar</foo>
XSL Style Sheet Template:
xsl-template select=„foo“ + special logic to differentiate between foo‘s

Template Problems (Continued)

The template mechanism for code generation is so popular nowadays that it is worth taking a closer look at its problems. Read the critical paper by Terence Parr on keeping templates clean (see Resources)

I am working with a content management team which had used TCL based templates to render XML data. When they moved to an XSL/XSLT based approach they discovered that the effort to transform the TCL templates into XSL had little to do with the actual output formatting. Because TCL is a turing complete language the templates where also used for many quick fixes for output generation which where not based on model information (i.e. not in the XML data).

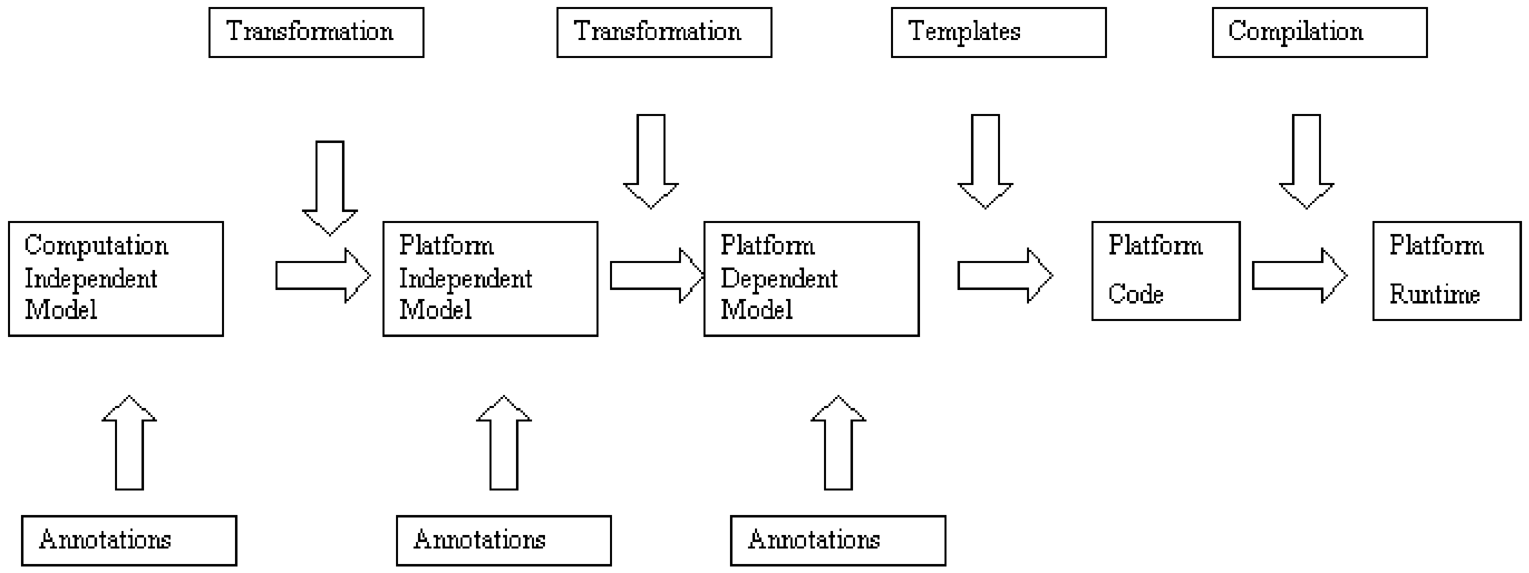
Limits to code generation

Most code generation technologies today focus on data modelling problems (e.g. insure referential integrity rules in object oriented code, generated database schemas for object/ relational persistence etc. They frequently (like EMF) do not support generation of business logic yet.

Model Driven Architecture (MDA)

The model driven architecture (MDA) defines a phased model for going from descriptive models via platform independent models to platform dependent models and then to the final platform itself.

Model Driven Architecture (MDA) (Continued)



Model Driven Architecture (MDA) (Continued)

Especially the left side of the MDA diagram is rather nebulous. Could it be that this could be the area of domain analysis (generative computing), conceptual domain analysis (M. Fowler, Introduction to UML) or Domain Specific Languages? Also interesting is what kind of information needs to be added to the various models to enable the transformations to levels with less abstraction. Will this system allow complete tracing - e.g. in case of errors at lower levels to go back through all the models? The annotations will play a critical role here if they are not part of the model itself.