

SoftFluent

How Model-Driven Software Development
can help your projects succeed

CodeFluent Entities White Paper

Daniel COHEN-ZARDI

Version 2.0

June 2011

Executive Summary

The objective of this white paper is to describe the software development challenge, clarify its root causes and show how SoftFluent addresses it through its CodeFluent Entities model-driven software factory and associated methodology.

The first half of the document explains the market challenge and why this is a tough business issue:

- In chapter 1, we explain the structural nature of the software development challenge,
- In chapter 2, we comment on the common approaches that exist on the market,
- In chapter 3, we detail our field observations of project failures, particularly frequent with new technology (whose openness bear more risks than traditional constrained legacy),
- In chapter 4, we propose a "Success Equation" as a model for summarizing these elements.

We would like to point out that this part is widely applicable by anyone interested in software development and is not dependent on our offering.

In the second half of the document we explain the solution we have built to structurally address the challenge:

- In chapter 5, we present our tenets that have been guiding the design of our solution to overcome the limits of past CASE approaches,
- In chapter 6, we detail the different steps of our methodology using the CodeFluent Entities approach for modeling, developing and industrializing,
- In chapter 7, we list typical situations to illustrate the concrete benefits that one will get by using the CodeFluent Entities approach.

In the end, CodeFluent Entities aims to be a pragmatic recipe to achieve successful software development projects. Possibly, this recipe is not unique, but this one has proved to be of great value to our customers so far.

We conclude in chapter 8 by exposing why model-driven is so structurally important in finding the solution to the evolution challenge detailed in this white paper.

© SoftFluent, 2010-2011 - Do not duplicate without permission.

CodeFluent Entities is a registered trademark of SoftFluent. All other company and product names are trademarks or registered trademarks of their respective holders.

The material presented in this document is summary in nature, subject to change, not contractual and intended for general information only and does not constitute a representation.

Table of Content

I.	The software development challenge	4
II.	Common approaches	5
A.	RAD Tools	5
B.	CASE Tools	5
C.	Off-shore	5
III.	Our field experience of failures.....	7
A.	Lack of skills	7
B.	Lack of experience and “geek attitude”	7
C.	Over-engineering and frameworks	8
IV.	A proposed model for the “Success Equation”	10
V.	SoftFluent Tenets	11
VI.	CodeFluent Entities Agile Methodology	12
A.	CodeFluent Entities principles	12
B.	CodeFluent Entities quick overview	13
C.	Agility with CodeFluent Entities	14
D.	Recommended steps for basic usage	15
E.	Recommended steps for advanced usage	19
VII.	CodeFluent Entities benefits through 12 scenarios	22
VIII.	Conclusion.....	24
IX.	Appendix	26

I. The software development challenge

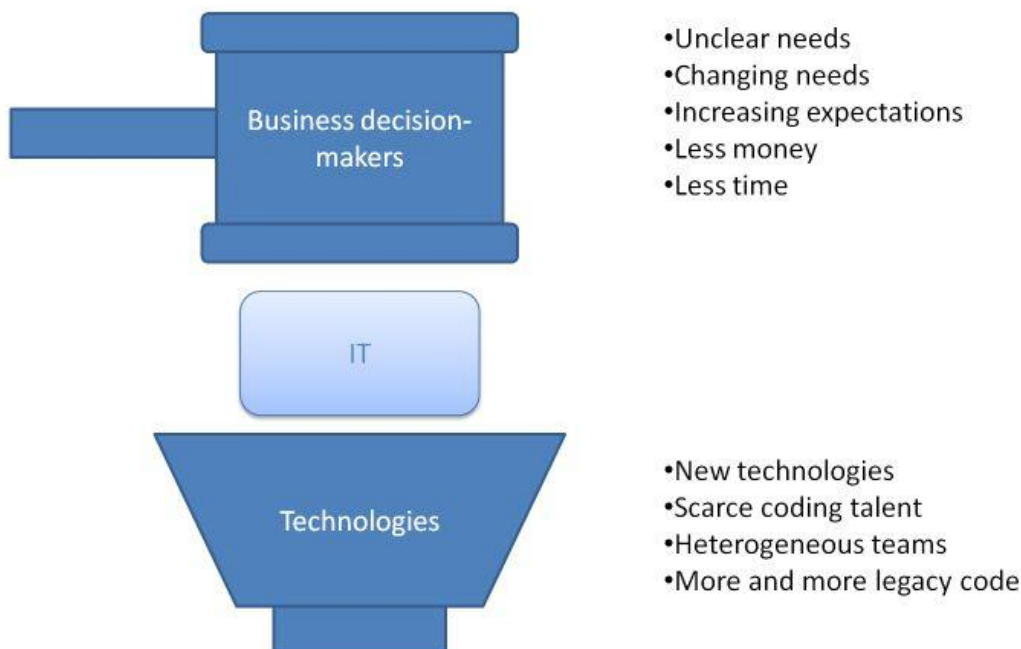
Being in charge of a business software development team in the 2010 decade is not the most enviable job in the world. On the one hand, user expectations keep inflating, whereas means are put under increasing constraints.

As each business sector is making progress, this unavoidably translates into **more and more sophisticated functional requirements**. Meanwhile, business users have **higher and higher technology and usability expectations**, consistently with what they experience at home as individual consumers. The fact that packaged software they use in their personal life is produced for millions of users with significant investment does not help much an IT team to argue about lowering those expectations for their business users at work.

As a matter of fact, the development team must **digest technology innovation**, and provide ready-to-use solutions for its users. The challenge is getting harder and harder as the pace of technology innovation is accelerating, with an **increasing complexity** which becomes hardly human-manageable.

Completing the picture by the additional pressure most IT teams face in terms of **reduced budgets and timeframes**, everyone can quickly understand that developers are **structurally squeezed** between growing functional expectations and evolving technology.

Challenges faced by developers



Coming to that point, you might be tempted to embrace another career path, and we might recommend you do so if you can! But if this is too late for you as this is for us, we invite you to continue the reading.

II. Common approaches

Software development is a 50 year-old discipline, so one can wonder why solutions do not exist today. Let us comment on the different common approaches that have been tried and highlight their main limits.

A. RAD Tools

As developer time reveals costly, there is a long history of software engineering tooling focused at accelerating this process through various means, especially the one known as code generation.

Some popular RAD tools, such as PC Soft Windev or 4D, have actually provided a good way to deliver applications within quite short timeframes. Different prototyping or RAD tools can quickly demonstrate results that look appealing with a simple process.

However, if the application reaches a certain level of complexity, it is often found that this kind of tools is not able to deal with feature sophistication, as the tool is very efficient in delivering what it has been designed for, but totally closed to additional added-value if it has not been pre-designed for it.

The main issue for customers is then to be stuck with **not being able to answer the functional requirements** for their applications. This is usually not an acceptable risk for customers.

B. CASE Tools

Computer Aided Software Engineering tools have existed for long, and have been numerous popularized during the client/server wave of the 1990s.

Tools such as Powerbuilder, Progress, Magic, NS-DK or Centura gained a lot of momentum at that time. These tools had invested a lot in providing a programming model for client/server applications, and encountered real success in terms of both market reach and productivity gains. In particular, these tools hid the complexity of languages such as C++ which was a standard lower-level programming language at that time.

But those CASE tools lost popularity, mostly because of the emergence of a totally new programming model with the emergence of the web. As a consequence, numerous customers refocused on lower level languages such as Java or C#, pushed by platform software vendors, losing in the process the gains that had been achieved so far by these higher abstraction tools.

Another reason is the fear of some customers of being locked in the vendor environment, as these tools usually provide their own programming language.

In the end, the main issue for customers is the fact of **not being able to keep up-to-date with technology**, as the footprint of these offerings make them very slow to evolve. As a matter of fact, there is no real efficient CASE tool to date for web applications. This, again, is usually not an acceptable situation for customers.

C. Off-shore

Another approach that many companies have tried especially during the 2000s decade is to address the issue by lowering the cost of resources through moving development off-shore.

Though it might look economically appealing initially, with labor costs varying of a 1 to 10 factor across geographies, this “solution” has been far from delivering on its promises.

In fact - when it works - it does not fundamentally address the challenge, as it just lowers perceived costs on development resources, while adding a lot of new difficulties and risks on the project.

Complete books could be (and have been) written on off-shore development, but, to make a long story short, putting the team further away does not increase its know-how and is not the answer to our challenging equation.

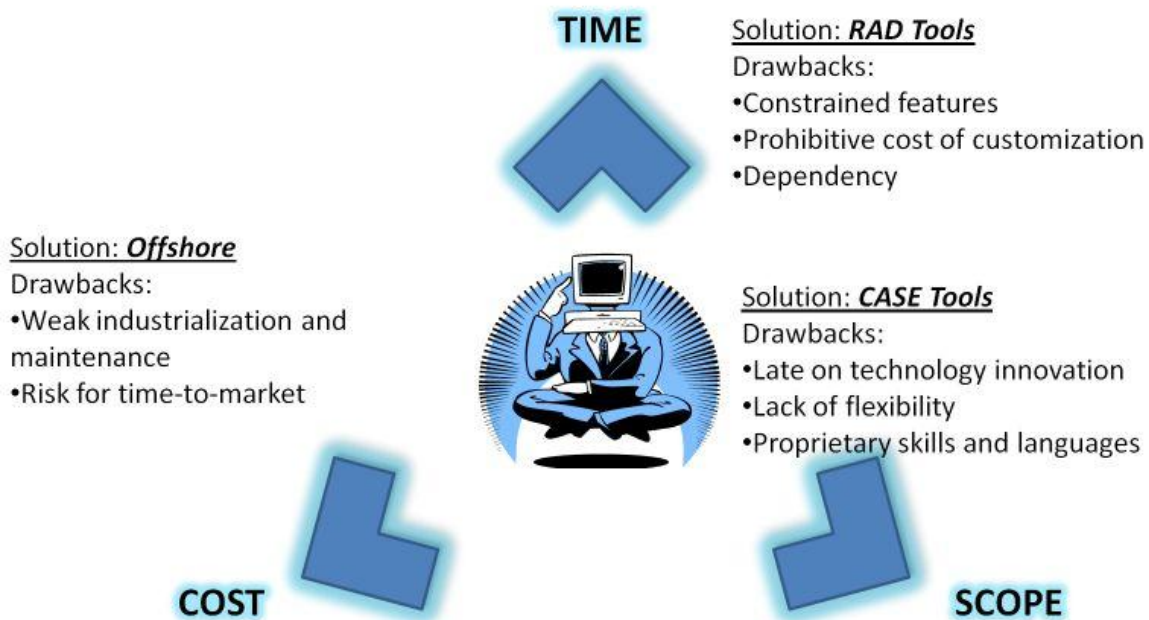
On the contrary, experienced people in software development know that the total cost of an application will encompass the costs incurred during the maintenance phase. This phase will usually last longer than the initial development phase.

And what might look like savings in the short term often turns out to be additional costs and lost agility over the long term. Any experienced developer knows that the evolution capability of a poorly designed application is weak and each minor evolution becomes more and more costly.

The main issue with off-shore is that it is often driven by cost reduction, with a project delivered **at the expense of industrialization and maintenance**.

In our views, those 3 approaches bear intrinsic limits putting an excessive emphasis respectively on timeframe, scope and cost.

Common approaches : focusing on 1 of the previous ambition component



All those three parameters need to be balanced appropriately, and this is why we favor another methodology to address the software development challenge.

III. Our field experience of failures

Because we are experienced in software development, we are often asked to lead audit missions on .NET developments to analyze the quality level of an application, its evolution capability, its scalability, its manageability in production.

First of all, it is worth mentioning that we are amazed by the great proportion of project failures and the amount of inefficiencies. We have often met failures in the several million dollar range and it is quite common to conclude that at least half of the produced code could have been avoided by using the appropriate libraries, most of the time in the .NET framework itself!

An interesting point to note is also the existence of typical syndromes we have observed for development teams, that we sum up hereunder.

A. Lack of skills

The first reason for failure is the lack of skills.

In some cases, even experienced developers may lack skills in new technology. As the complexity of technical layers used to be properly hidden in CASE tools, it is not the case any longer, with the additional challenge of a great openness of environments, leading to many more potential pitfalls.

In this situation, we observe that the code usually works... until the project gets more complex in perimeter or in capacity. Since code is produced manually, with heterogeneous styles changing over time, maintenance is getting harder and harder and the cost of incremental change inflates until the team eventually meets a wall.

In the mentioned failure case, this failure is due to the first factor: **skills**.

B. Lack of experience and “geek attitude”

In other cases, we have observed young developers that are rather clever and comfortable with new technologies. But they may lack enterprise experience and many of them have a strong tendency to adopt a “geek attitude”, a behavior where people focus on each and every new piece of technology.

In those situations, developers have a strong technology bias and concentrate only on the code and its beauty, like writing a complex instruction within a single line of code, which satisfies the ego of the developer but does not bear any intrinsic value. In fact, it even holds potential negative value as it may increase maintenance costs.

These profiles often lose sight of the functional objectives, because users are not their primary point of interest, and they do not have the proper level of experience to understand the challenges of manageability and the necessity for code instrumentation.

Because of this lack of experience, they commonly underestimate some challenges of scalability and applications they design perform poorly with high volumes of data or numerous users.

Applications developed this way are usually hard to maintain and require targeted optimization or redesign in areas where performance is needed.

This case of failure is in fact a lack of a proven **method**.

C. Over-engineering and frameworks

The last case often leads to the most costly situations. And in fact, it is the most observed situation on large accounts.

In this third syndrome, a genius so-called architect (or several) has been hired to invent a customized solution for the development team. Usually, this architect is smart, and this is part of the issue, because he often believes he will be able to invent all relevant solutions for the project. He then starts writing - potentially with a team of developers - a framework that is supposed to be the optimal recipe for the project but that often goes too far.

Setting the appropriate abstraction level for a project requires placing the cursor at the right place on the schema hereunder:

Setting the appropriate abstraction level



The “Where to stop?” on the left is a critical question, as in our experience, this is where things turn wrong, even though the idea of building a framework could look great.

Experience shows that:

- Building a flexible framework that can face all needed situations with new and evolving technology is hard and risky,
- The risk of simply failing with a framework that does not work is huge, with potentially a tunnel effect during framework development, a phase that can last forever,
- In most favorable cases, the framework will provide value for a certain technology wave and will not resist the next wave, with the necessity of big reinvestment to keep up with new technology, and another risk of failure,
- In almost all cases, the framework cannot be maintained by someone else than the architect.

Though not always admitted, this situation leads to significant economic failures.

If sometimes CASE Tools were perceived as a problem because of vendor lock-in, the **architect lock-in** syndrome described here is much worse. The risk of depending on an individual should be perceived as much more worrisome than depending on any third-party.

This case of failure is an over-investment in **tooling**, a factor which is quite hard to address, and whose complexity proves to always be under-estimated.

IV. A proposed model for the “Success Equation”

Many studies about software development confirm a high degree of failures in software development projects. In a recent international study quoted by AFDEL, it is recognized that only 30% of software development projects are considered as successes, and only 13% are delivered on time.

In fact, we could put the success probability of a project into an equation that we would simplify by a simple formula such as:

$$\% \text{ Success} = \frac{\textit{Know how}}{\textit{Ambition}}$$

If the global “Know-how” is above the “Ambition”, the project will be successful. This sounds pretty simple to understand.

Additionally, introducing the elements mentioned above about expectations in the introduction, we could consider that:

$$\textit{Ambition} = \frac{\textit{Scope}}{\textit{Budget} \times \textit{Timeframe}}$$

Noting that each component is evolving in an unfavorable direction, this means the ambition is **growing faster and faster by a cubic factor!**

Simplified this way, it gets crystal clear that software development projects cannot succeed without an approach that allows “Know-how” to grow fast enough to keep up with the ambition.

As a matter of fact, without a methodological shift, **there is not even a chance for development teams to succeed over the long-term.**

This is consistent with our observations in the field, where – in reality although sometimes hidden - costs are often exploding and the volume of “legacy obsolete code” keeps growing, increasing the maintenance burden on development teams.

So is there any hope to keep up with the ambition or is this a lost battle?

Let us consider the know-how as being the combination of 3 key components, the ones that appear to us to be the major success factors for development projects:

$$\textit{Know how} = \textit{Skills} \times \textit{Method} \times \textit{Tools}$$

Our equation becomes sustainable, at the imperative condition of **perpetually improving the skills, the method and the tools.**

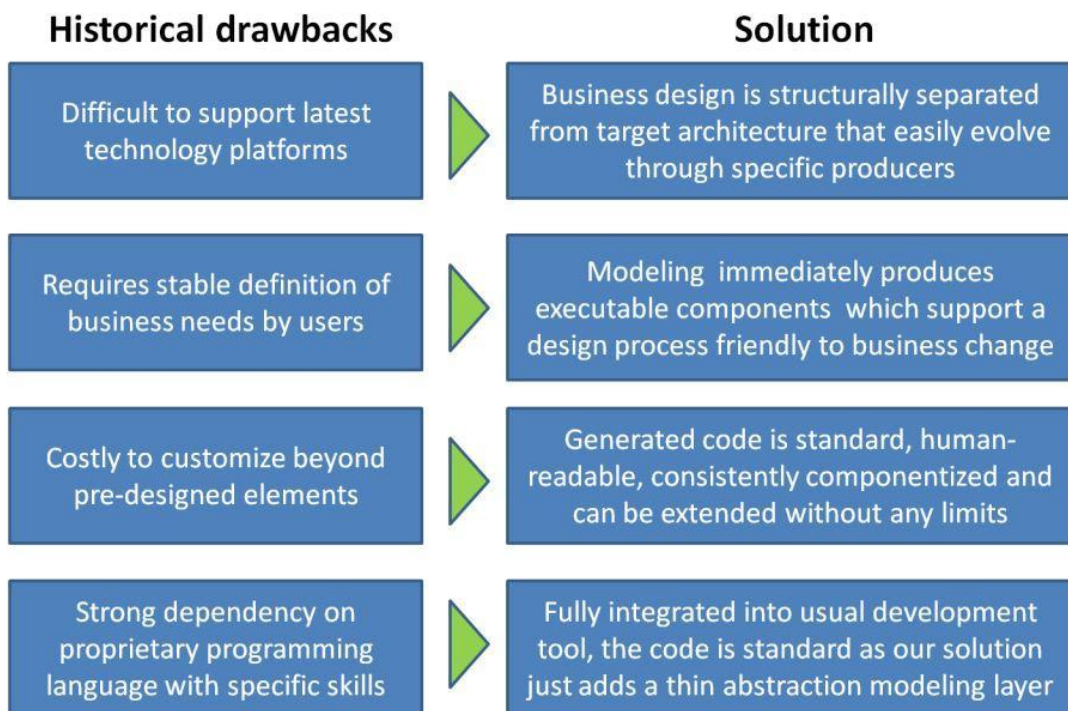
This is basically what we propose with our software development approach, based on an iterative model-driven method, supported by a tool that improves the development process itself. Development team responsibility is then to focus on maintaining the skills, a simplified process in our approach, as part of the complexity of underlying technologies is hidden.

V. SoftFluent Tenets

Because at SoftFluent, we have spent many years on development projects¹, we have leveraged our experience to define a new recipe that will ensure project success, even with evolving technology.

Our development approach is built on the success of past recipes, such as CASE tools, but with a couple of structural differences targeted at addressing the limitations observed with those tools.

Improving on the CASE tool approach



The way we address the key historical drawbacks of CASE tools are formalized into our keys tenets, at the heart of our model-driven development methodology:

1. Business entities have longer lifecycles than technology and should be defined in formats that will **survive technology shifts** and allow supporting them without reengineering the application.
2. Business rules and process changes need to occur through **short cycles** that can fit into a lean continuous maintenance scheme.
3. Efficient **coupling** of data, model, and presentation layers including customized code parts must be **guaranteed by design**.
4. Coding **patterns** need to be standardized and their implementation **automated** to ensure maintenance can be performed by **standardized skills**.

¹ SoftFluent founders are former Microsoft employees with a long track record in software development.

VI. CodeFluent Entities Agile Methodology

Now that we have introduced our tenets and the key drivers that we have built our offering upon, we will explain in this chapter how one works with CodeFluent Entities to design and deliver software.

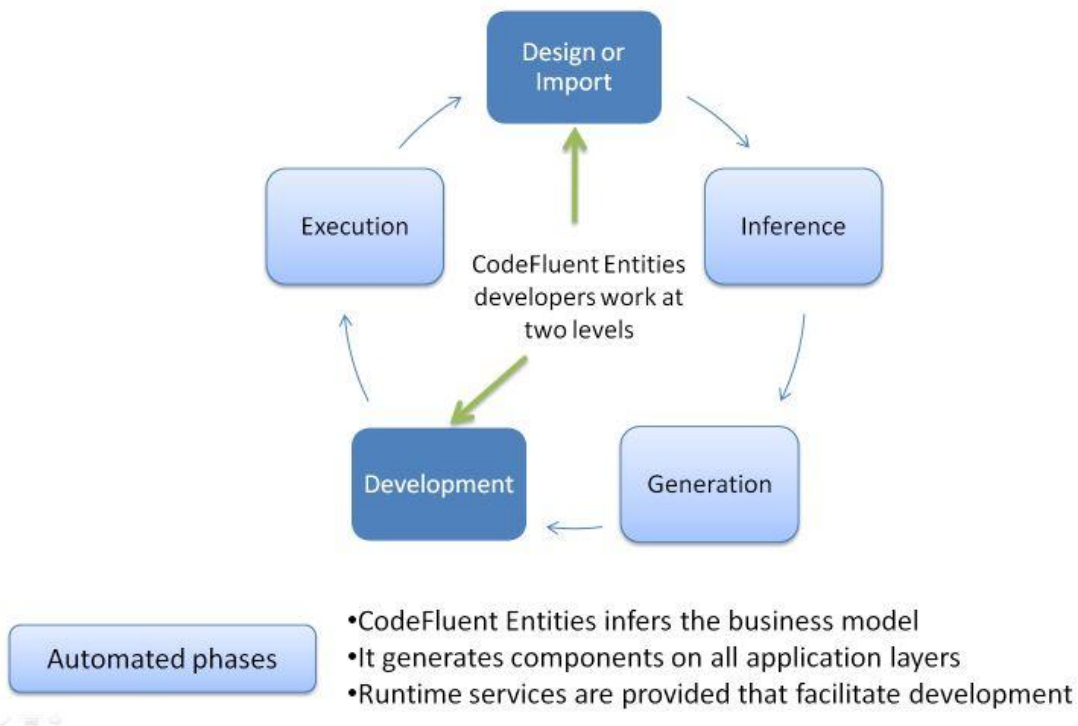
A. CodeFluent Entities principles

CodeFluent Entities builds on SoftFluent tenets by proposing to concentrate the effort on two areas:

- **Design of functional elements** in a manner that is mostly independent from technology,
- Development of **customized code**, mainly in the business layer for business code and in template customization for generated UIs or reports.

As described in the schema hereunder, all other phases of the application life cycle are automated:

CodeFluent Entities Application Life Cycle



Beyond the process itself, which is concentrating the effort into two focused areas, the added-value of CodeFluent Entities automated phases is embedded at 3 different levels:

- **Model analysis and inference:** though not visible, this automated phase is extremely powerful as this is what allows the extremely advanced generation process that follows and the combination of complex behaviors,
- **Generation:** the producers generate ready-to-use components on all layers chosen, with a total consistency in the way those components integrate with each other,
- **Execution:** the runtime libraries provide advanced services that are leveraged by generated components to make the development process more productive for the developer.

B. CodeFluent Entities quick overview

Though the goal of the white paper is not to give a detailed view of the product, we suspect that many readers familiar with software development would like to get a sense of what it exactly does and how it differentiates from traditional offerings through a minimal demo.

A complete demo would last hours as the product encompasses many development areas and brings a very complete set of features, but for persons who really would like to see the product "in action", we recommend going through this [product overview](#) (22 minutes).

It is also possible to directly jump to the desired topic using the following table:

Video topic	URL
Define Entities	http://www.youtube.com/watch?v=CV5py9C1laQ
Define Properties	http://www.youtube.com/watch?v=L-4CQyEBBFc
Define Enumerations	http://www.youtube.com/watch?v=nI5QKxpJHlo
Collaborate	http://www.youtube.com/watch?v=lo2K8gXeB4Y
Meta-Model	http://www.youtube.com/watch?v=WEh17S0tiYA
Generate	http://www.youtube.com/watch?v=SXAeJlpIKHQ
Instances	http://www.youtube.com/watch?v=Hp4a90QlcgU
Continuous Generation	http://www.youtube.com/watch?v=lqZHn3AwSqE
Methods	http://www.youtube.com/watch?v=0Z4Ju4-5FD4
Define Rules	http://www.youtube.com/watch?v=5oG6KqXRBES
Extend Generated Code	http://www.youtube.com/watch?v=On3GkLAYIMg
XML View	http://www.youtube.com/watch?v=K1E4baWe-lQ
WPF Smart Client Producer	http://www.youtube.com/watch?v=uw_n11aTNqo
SharePoint Web Parts Producer	http://www.youtube.com/watch?v=l7vudNA5fcl
Develop Custom Uis	http://www.youtube.com/watch?v=wsZa2I0zCPA
Import	http://www.youtube.com/watch?v=VoLCj2uaJws
Form Editor	http://www.youtube.com/watch?v=cORcfjRuEgo
Navigating in Models	http://www.youtube.com/watch?v=LcNoEWZqucE
Customizing Namespaces	http://www.youtube.com/watch?v=daSyJjj9JZ8
Adding Notes	http://www.youtube.com/watch?v=Lio7FVEba_0
Model Grid	http://www.youtube.com/watch?v=3mS84lkJAMs

These videos illustrate a subset of product features and if you want to know more about the product, we recommend you to follow the [CodeFluent Entities product blog](#) for a more complete and regular vision of its possibilities.

C. Agility with CodeFluent Entities

CodeFluent Entities is not strictly dependent on following an agile methodology, as one can easily leverage the product using a classical V-Cycle methodology. As a matter of fact, several projects have been delivered with CodeFluent Entities using traditional approaches based on a preliminary specification phase.

However, CodeFluent Entities is particularly efficient in supporting projects driven through an agile methodology. Agile methodologies are based on the principle of iterative development, which requires delivering software that works at the end of each iteration. One iteration after the other, the scope is extended until the software presented to users is satisfactory enough to be put into production or released to the market.

CodeFluent Entities proposes several attributes that prove to be very interesting when used in conjunction with an agile methodology:

- Functional elements designed into CodeFluent Entities **immediately generate executable components**,
- Specifications do not need to be defined upfront because the model can be **enriched and regenerated as necessary** without losing any customized code,
- Technical target architectures can be chosen **later in the cycle** by simply changing the producers and layer-specific customized code.

By design, CodeFluent Entities offers an iterative process for development because of its continuous modeling and development integrated process.

At many customers, business modeling is started directly with users and quickly transforms into a first version of model thanks to the opportunity of showing them generated user interfaces.

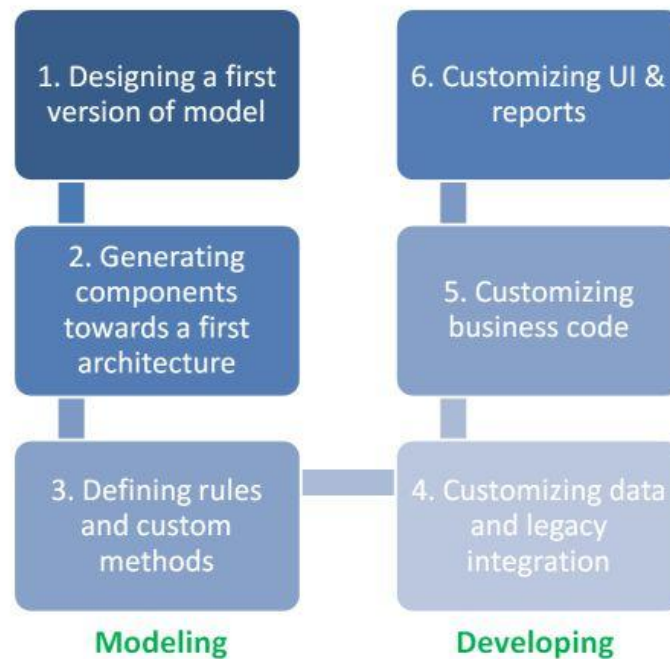
D. Recommended steps for basic usage

In this chapter, we describe the usual steps that we recommend for delivering best class applications using CodeFluent Entities.

The first three steps are focused on working at model level, allowing generation of prototypes that can be quickly presented to users for validation.

The second series of steps is more focused on developing custom code that will require a little more work, and which might be more dependent on target platform.

Basic usage implementation steps *(iterative)*



Important note: As explained in the previous paragraph, CodeFluent Entities is very friendly to agile methodologies, which means those steps are not intended to be sequential at the overall project level. Instead, these steps are usually sequential within an iteration, with the first iterations concentrating mostly on the ‘Modeling’ steps.

1. Designing a first version of model

The first step with CodeFluent Entities consists in building an initial version of the functional model. In this initial version, **most entities and major relevant properties**, including enumerations, are identified.

This step formalizes the key business entities relevant for your domain, their relationships and the major structural elements for your application. Using a classical Entity-Relationship model and inheritance, one can describe any kind of business with simple concepts.

It is also worth noting that even **data instances** can be defined in the model. This proves to be very useful as it is often needed to have some specific records in the database.

Usually, these elements are **very stable** over periods of time that can last more than a decade.

2. Generating components towards a first architecture

Once business entities have been modeled in CodeFluent Entities, the architect usually chooses the specific architecture relevant to the business: web, smart client, Office, SharePoint, traditional client-server or even cloud.

Thanks to the CodeFluent Entities Model-Driven approach, the architect can immediately generate a **first executable version of the application** that will implement the functional model.

This version can then be presented to users so as to **validate the functional elements** of the model.

Additionally, as CodeFluent Entities allows maximum flexibility with target architectures, it is also possible to use these early prototypes to **guide the architecture choice** (or choices).

It is quite often that in customer situations, we see a relatively important challenge in choosing the relevant target architecture, especially when those applications are intended to replace a legacy version.

Doing pure web is the natural trend, but usually the consequences are not fully measured, in particular the investment that is needed to reach the same level of ergonomics than a real Windows application. So going for a web version without measuring the readiness of users, may translate into a failure of acceptance test by users, which is the last thing a development team needs.

When fully investigating the expectations of our customers, we often find that **optimal target may combine**:

- A full web version broadly proposed to occasional users with the merit of minimizing deployment challenges,
- A smart client application for advanced users who need to work every day on some business data and manipulate them with maximum flexibility and visual tools,
- A mobile application with a specific reduced feature set within identified scenarios.

This is why we see very high value in being able to show results quickly with the CodeFluent Entities model-driven approach.

Note: Similarly, a storage system can be chosen at this stage and possibly changed later in the cycle.

3. Defining rules and custom methods

Once the major elements of the application have been validated by users, the second agile iteration can be used to define the major rules and custom methods of the model.

This mainly includes the ones that can be declaratively designed. CodeFluent Entities offers an advanced set of options to describe:

- Existence rules,
- Basic format rules,
- More advanced format rules such as 'email' for example,
- Integrity rules with options like 'Cascading updates or delete',
- Computed properties.

At the end of this step, the **business model**, formalized as a CodeFluent Entities model, already includes an important part of its **core value**, while being totally independent from technology.

Because the format used for storing this model is plain and simple XML, this makes it very open and resistant to future technology trends.

4. Customizing data and legacy integration

Beyond the modeling of methods, CodeFluent Entities proposes advanced mechanisms to **customize data-access methods** through:

- Simple and intuitive CodeFluent Query Language,
- Custom views,
- Possibility to use native stored procedure language where necessary.

Optionally, it is also possible to leverage particular options to:

- Map naming to preserve existing legacy database schemas,
- Connect to legacy components instead of the generated database.

5. Customizing business code

CodeFluent Entities embeds numerous modeling concepts that simplify software development and allows a significant level of definition for custom methods and rules.

Still, the philosophy of the product is that it is not possible (or should we say economically relevant) to introduce over-complicated modeling concepts to deal with the most advanced use cases. This is why CodeFluent Entities is designed to **allow** the "classical" implementation of **customized code** in the different layers. In particular, CodeFluent Entities based applications usually concentrate many custom rules or behaviors in the "business object model", the central layer, to ensure consistency.

So this key step in implementing applications consists in developing the **partial classes to complete the generated business object model**, in order to ensure the fulfillment of business requirements.

6. Customizing UI and reports

Though CodeFluent Entities comes with standard producers that directly produce operational user interfaces that can be used to manipulate data, business applications usually require custom-tailored screens or pages that optimize the business processes.

CodeFluent Entities facilitates this process by building a business layer and objects that are **easily pluggable to user interface controls available** on the market. This approach works well with standard controls delivered by Microsoft in the platform as well as all classical third-parties such as Infragistics, DevExpress, Telerik or ComponentArt.

Designing an optimal user interface for frequently used business processes requires maximum **flexibility**, and this is why CodeFluent Entities developers usually assemble the generated components, while leveraging a set of **useful runtime services** provided by CodeFluent Entities for the different supported architectures (Web, rich-client, Silverlight, SharePoint).

Though customized user interfaces are generally needed, in many projects we see, usually a significant portion of the screens only require standard behavior and structure. This is why, in "advanced usage scenarios", we will describe how one can automate the creation of user interfaces using templates.

CodeFluent Entities comes with an **integrated template engine** that allows the production of any text file based on the principle of mixing the "text output" with instructions that are executed (like Active Server Pages used to work before .NET). Though being a relatively small part of CodeFluent Entities, this element is comparable to many code generators that actually provide only this mechanism.

But it is even more powerful because the template engine is aware of the CodeFluent Entities object model available as an API, allowing an **easy leverage of all the elements modeled** by the developer in the graphical designer.

Additionally, the **CodeFluent Entities template engine is RTF-friendly**, as it deals correctly with the escape sequences, allowing an easy strategy for Word reporting in your projects with a high level of performance.

The Microsoft Office list producer also provides a solution for integrated user interfaces in classical Microsoft Excel and Microsoft Access.

E. Recommended steps for advanced usage

Though easy to start with, CodeFluent Entities actually includes very advanced features for the most demanding development projects. As it is a complete software factory, it offers solutions to industrialize the production of code at an unprecedented level.

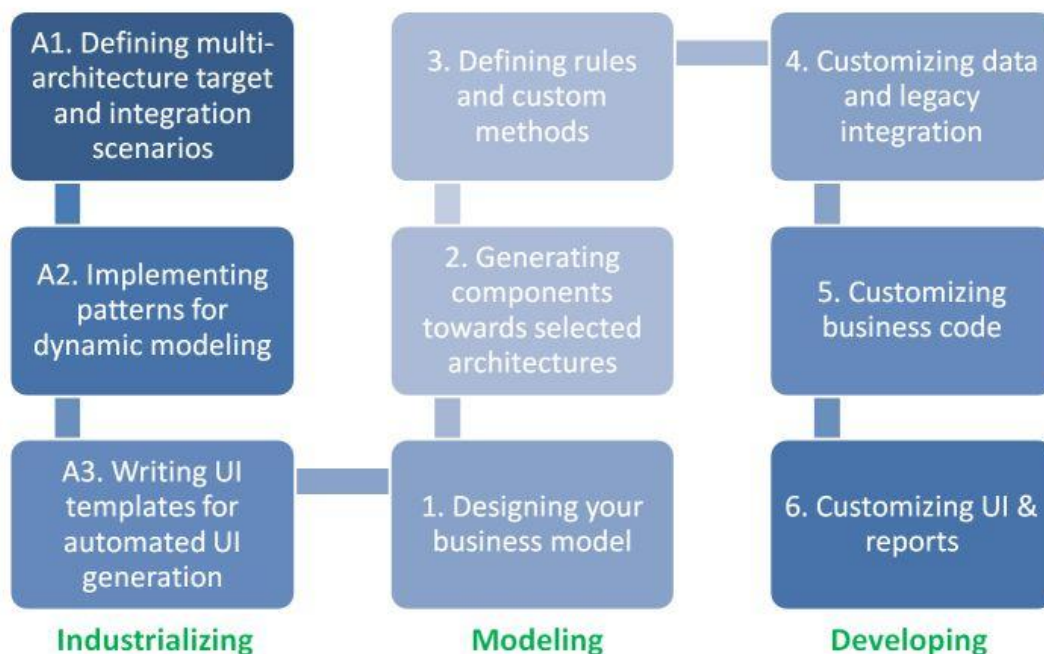
Though we recommend to implement the first project through a basic usage scenario to get familiar with the model-driven approach and the product, on large projects, a **huge leverage effect** can be achieved through advanced usage.

Most advanced usage scenarios include:

- Multi-architecture scenarios,
- Dynamic modeling and patterns,
- User Interface automated generation.

Advanced usage implementation steps

(iterative)



1. Defining multi-architecture target

Today, applications are usually designed for a specific architecture, such as client-server, rich client, web or rich web. Sometimes, there are specific extensions to support a particular scenario, such as mobility. It is then often seen as another version of the application.

In our view of software development, and also because it is made possible at a reasonable cost through our model-driven approach, future successful applications may often include different architectures, at least for part of them.

For example, one could imagine for an HR ERP that:

- Specialized business users will use a **smart-client** to manipulate data with maximum flexibility,
- All users in the enterprise will access their data through a pure **web-based** self-service portal,
- Some scenarios will be directly integrated on **mobile devices** where the relevant features will be deployed.

This kind of scenarios will be more and more frequent, as well as their combination with **cloud** versions if we anticipate a bit the future.

Using CodeFluent Entities, **adding those new architecture scenarios** is just a matter of:

- Adding **new producers** to generate relevant components,
- **Completing** only the code parts concerned, especially **customized UI**.

2. Implementing dynamic modeling through patterns

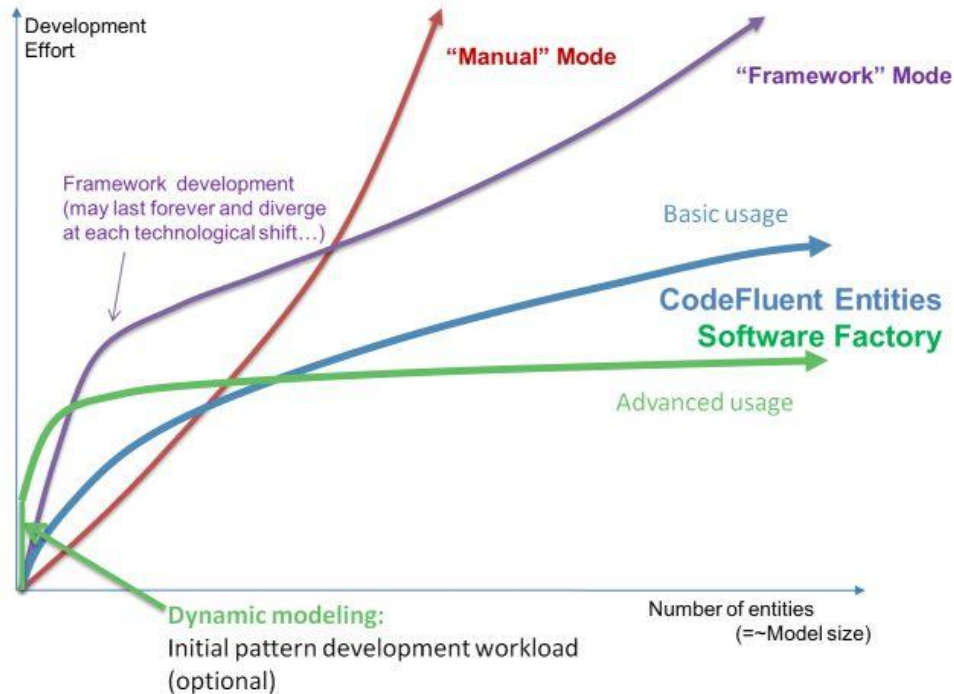
As explained in chapter 3, setting the appropriate abstraction level requires to find the right balance. The good news with CodeFluent Entities is that you can **start very simple** while getting the benefits of powerful concepts at a later stage.

The product being **completely designed as an API**, giving developers maximum flexibility, the template-based approach explained earlier can be applied to the model itself, bringing a powerful leverage effect: **dynamic modeling**.

Applying the template principles to the model itself, CodeFluent Entities allows you to define **patterns**, an industrial way of adding behaviors to your entities. This includes behaviors such as: localization, tracking, text-based search or any other custom need.

Whereas purely template-based generators are unable to combine these behaviors without leading to templates of such a level of complexity that they are not maintainable (well-known as the spaghetti code effect in 'Active Server Pages'), the different approach proposed by CodeFluent Entities for the production of the business object model makes it industrial, as represented hereunder.

Industrializing software production



Basic usage of the product already provides an economically viable solution over the long-term (which is not the case of ‘manual’ and ‘framework’ mode that we observe on the field) and advanced usage with dynamic modeling provide an **even greater ROI for large projects** that are intended to work with large models that comprise hundreds of entities.

3. Defining UI templates for automated UI generation

In many projects we see, though customized user interfaces are generally needed, usually a significant portion of the screens only require standard behavior and structure. This is particularly true for back-office user interfaces or management of reference data.

In that case, it is relevant to invest in a **user interface template** that will factorize the structure of the UI and the common behaviors. This template will serve as a basis for the generation of user interfaces as the one delivered in the product.

This way, the **development and maintenance costs will not be correlated to the number of entities**. For projects dealing with hundreds of entities, this can provide major savings, especially over the whole application cycle.

Thanks to the great flexibility of CodeFluent Entities architecture, this approach can also be applied for all user interfaces, using advanced product features, custom renderers and custom code in templates. However, for complex screens, it might be harder to maintain and this is why we propose both approaches, the architect being able to **balance** the two at the optimal level for the project.

VII. CodeFluent Entities benefits through 12 scenarios

Now that you understand why we build CodeFluent Entities and how you can deliver applications using CodeFluent Entities, let us give some classical example scenarios that will illustrate how you will benefit from it.

Scenario	Without CodeFluent Entities	With CodeFluent Entities	Keywords
Product management or users have forgotten an important property in an entity	You will need to change the database schema, the stored procedures, modify the business, services and user interface layers.	You just add the property to the model and you regenerate. If your user interface is custom, you simply add the property.	Model-Driven Consistency Layers
Your application has become successful. But the issue is that you now need to support 10,000 concurrent users with billions of records	You will need to implement advanced paging and sorting mechanisms on relevant screens and figure out how to propagate the approach across the layers.	Leveraging built-in paging and sorting mechanisms of CodeFluent Entities, your work is limited to configuring them appropriately on relevant parts of the application.	Scalability Best practices Consistency
Legal department now requires you to track all data changes made by the application	You will need to hook specific processing at various places in the application while evolving the database schema to store the relevant data.	You just need to model the tracking data you need and implement a pattern that will be hooked in the business layer for all components once and for all.	Patterns Dynamic modeling Consistency
Users want to enter data in a list mode directly inside Excel	You will need to develop a specific import process to integrate Microsoft Excel.	Using the Excel producer, you can immediately generate updateable lists that include checking business rules.	Office producer Ergonomics
All of your objects now need to respect certain standards such as property names starting by the entity name	You will need to rename all elements manually, or build a tool to automate the process and test it. It is likely that your work will not be 100% right.	You can directly use naming conventions proposed by CodeFluent Entities and you will get 100% compliance "by design".	Consistency Convention Quality
Your application has been sold abroad, you need to support 10 languages	You will need to check the respect of internationalization rules and find ways to localizing data in the database.	Using Unicode types, messages for resources and the CodeFluent Entities embedded localization pattern, you will have a solution for all internationalization requirements.	Localization Best practices Patterns

<p>As part of a partnership with Microsoft you now need to support Silverlight</p>	<p>You will need to reengineer your application for this new architecture, including hard points such as asynchronous calls.</p>	<p>Using CodeFluent Entities smart client object model, your work will be limited to configuring a new producer and adjusting UI elements.</p>	<p>Architecture Silverlight Innovation</p>
<p>The application now needs to be deployed on another database system</p>	<p>You will need to figure out what part of the data access code is specific to your current database system, reengineer those elements and potentially find a mechanism to support both (if needed).</p>	<p>CodeFluent Entities have been designed in a way to allow different database deployment scenarios, while optimizing coding for each RDBMS system. Only raw methods that can be used for optimizing specific processing need to be revisited.</p>	<p>Model-Driven Architecture Oracle/SQL</p>
<p>Users want to deliver letters to customers with data extracted from the applications</p>	<p>You will need to automate Microsoft Word or integrate a specific component to deliver this feature.</p>	<p>Using the Template producer with RTF templates, you have an immediate solution with customers being able to edit letter templates in Microsoft Word.</p>	<p>RTF producer Templates</p>
<p>A new policy changes the rule checking made on a property</p>	<p>You will need to change this rule wherever applicable, possibly both on the client and the server.</p>	<p>You add the constraint in the model and regenerate. It is possible to have rules checked both on client and server for better user experience.</p>	<p>Model-Driven Rule Architecture</p>
<p>Your last application has been so successful that you have been asked to reengineer the 20 year-old application to provide the same ergonomics standard</p>	<p>You will need to redesign the application from scratch, though the model has been built over a decade. This represents a huge amount of work.</p>	<p>Using CodeFluent Entities importer on the database used for the existing application, you immediately have a first model to start with, that you can generate to quickly provide modernized screens for your application. You can reengineer the application progressively, by improving the functional model and working on the new target architecture.</p>	<p>Legacy Importer Modernization</p>
<p>Investors have required your application to support a cloud architecture within the next weeks</p>	<p>You will need to figure out the different components of available cloud architecture and possibly write a wholly different version of your application.</p>	<p>Using CodeFluent Entities, you will be able to generate the model towards future platforms including cloud without writing a different version of the application.</p>	<p>Innovation Cloud Architecture</p>

VIII. Conclusion

Technology innovation is going fast, and this pace of change is accelerating as the world goes global and the competition gets fierce. A lot of announcements are made every day by technology providers and it does not look like this motion is going to slow down. The multi-layer style of software development has increased flexibility and capability, but has also **strongly increased the complexity of software development**.

With the multiplication of platforms and devices, such as cloud platforms or new mobile devices, and the growing needs to connect to collaborative social networks, we can be sure that this complexity will keep increasing in the future.

At the same time, for several decades now, businesses have developed significant operational IT systems, going further and deeper in features supporting their business and vertical sector. This means the legacy is way more important than it was years ago, and it is most of the time increasing. A [recent study by Gartner](#) showed that what they call the “IT Debt” is growing at a worrying rate.

In all businesses, there is a high pressure to lower costs, while providing new services to remain competitive in a more complex environment, and at the same time **bearing the increasing legacy burden mentioned above**, with its set of obsolescence constraints. Some systems even run totally unsupported which might become a critical issue that will force a reengineering at some stage.

For companies which need to maintain and improve the systems supporting their businesses, this is a quite unsolvable challenge, at least by keeping the same approach that was used in the past. It is not a solution to just apply past recipes as developing over applications at each new technology wave.

In fact, we hardly see customers managing to do it and except for very few exceptionally wealthy exceptions, no company can afford such a cost. And even if you were that wealthy, it would probably be a better option to spend your money on increasing your business.

At the same time, maintaining very old applications tends to be field reality, but one can legitimately wonder how long can this still be sustained without a major crisis. And such a “status quo” strategy, which is obviously less risky in the short term, is probably a sure path to death in the long term. Because once a competitor in the same vertical has been able to deal with it, it will be too late to start a reengineering which will take a long time to achieve.

This is why we think there are 2 critical areas that companies should be working on proactively:

- A. Set measurable objectives on the reduction of maintenance for existing legacy systems,
- B. Use the savings to implement new projects in a way as independent from technology as possible.

On point A, it is critical to precisely evaluate maintenance costs, in terms of hardware, software licenses and maintenance fees, and of course human expenses to run the system and correct bugs. For custom applications that are in maintenance mode, the cost is highly correlated to the number of lines of code and the quality of the application. Refactoring existing applications without any functional change can provide huge savings and is not necessarily a hard task, as long as we do not fall into the trap of changing features at the same time.

On point B, model-driven is exactly the promise of **capitalizing your work at a higher abstraction level than code**. We and our customers have been delivering projects that way for 6 years, and this evolution capability is obvious once you have done it.

Additionally, you get the significant advantage of getting very predictable costs and timeframes. Once used to the methodology, one can quite precisely estimate each project using factual elements such as the number of entities, properties, methods, business rules, user interfaces or reports. We will publish detailed numbers and the attached estimation methodology in the next version of this white paper, also allowing you to benchmark your current project against these numbers.

So stay tuned.

Daniel COHEN-ZARDI

CEO, SoftFluent

<http://blog.softfluent.com>

IX. Appendix

Here is a list of key resources to stay up-to-date with CodeFluent Entities and SoftFluent .NET software development best practices:

Url	Information
http://blog.softfluent.com	This blog is dedicated to sharing our .NET experience and the best practices on all aspects of software development.
http://blog.codefluententities.com	This blog is dedicated to giving you information on all features and aspects of CodeFluent Entities.
http://www.codefluententities.com/rss.aspx	This blog gives the list of new features and bug corrections for the product at each build.
http://forums.softfluent.com	This is the portal to get access to all SoftFluent forums.
http://www.youtube.com/softfluent	This is the SoftFluent channel that delivers regular video demonstrations about the product.
http://www.codefluententities.com/documentation	This the complete product documentation.
http://www.softfluent.com	This is SoftFluent Corporate web site.
http://www.codefluententities.com	This is CodeFluent Entities dedicated web site where one can download a free full-featured version of the software, unlimited for personal use.
http://store.softfluent.com	This is where you can buy the product on line.