empros gmbh
process & information management services

Some Thoughts on Better Software

By Sascha Frick, empros gmbh

Contents

Revision History

03.09.2005 Revision 1.0 sf

Developing Software is Hard

Today, writing a runable program is easy – developing software on the other hand is extremely challenging, despite all the progress of recent years in languages, design patterns, tools, technologies and processes. Still, overrun budgets, high defect rates, difficult to use applications, security holes and dissatisfied users plague our industry more than 20 years after the first software crisis has been proclaimed. So what does it take to change this, how can we get out of this mess?

In this paper, I will try to find some answers to the question, how we can improve the way software is developed and how we can devise better software. To make it clear from the beginning: For me, the answer is not one of better tools or processes. I firmly believe that most of the tools and processes we need to achieve better software are available today, what's really missing is the right attitude. We have to understand that software development (SD) is a collaborative effort of solving complex problems. To be successful, we need skilled and dedicated people, a trusting and supportive environment that fosters constant learning and that allows for an open communication. But we also need to shift our focus away from efficiency towards effectiveness. To put it mildly, our industry got a little carried away with efficiency, almost completely forgetting the simple truth that solving the right problem effectively is more important than solving some problem efficiently. It is more important to know the right problem than having a right solution. If we know the right problem, we stand a chance of finding an appropriate solution. Only by solving the relevant problems, can we hope to find elegant solutions that are as simple as possible but not simpler, solutions that benefit their users by helping them achieve their goals.

Programmers are Nerds

Let's come back to the point of software being a collaborative effort. Many people imagine software developers as technic-savvy, ingenious but introverted geeks who are a little engrossed and slightly removed from reality. And of course, they all wear cool t-shirts, refuse to drink coffee from anything but a mug adorned with the logo of their favorite tool or programming language, and their diet almost exclusively consists of pizza and soft-drinks. They are not very talkative and rather like to hide behind their screens and keyboards. Whether this stereotype is accurate, there is a grain of truth to it: most software developers rather focus on technical aspects than the intricacies of a specific problem domain.

The Heart of Software

As Eric Evans observes: "The heart of software is its ability to solve domain-related problems for its user. All other features, vital though they may be, support this basic purpose." Software must help the user to achieve his or her goals, it must provide value. Of course, software developers know this, but all too often, they get carried away doing technical stuff: they go on to work on elaborate frameworks, trying to solve domain problems with technology. To a

certain degree, this is understandable; working on the business logic of a complex domain is difficult, and some even wonder, why they call it business logic in the first place, when there are so many conflicting rules, and quite often logic seems to be completely absent.

Solving complex domain problems is difficult, calling for the dedicated effort of talented and skilled people. Developers must hone and cultivate their domain modeling and technical skills alike. They have to become sensitive to domain problems, empathic to the user and his or her needs and finally masters of multi-level communication; a communication that ranges from talking to and learning from domain experts, to shaping supple designs and clear cut intention revealing abstractions in code. Lest we ever forget that software developers are first class problem solvers, it is their job to tackle the complexity at the heart of a problem domain and it's their job to find a clear, simple and concise solution in software.

Don't get this wrong: emphasizing the importance of the problem domain, does by no means imply that technical expertise is not important. The opposite is true: technical skill is absolutely necessary for any software developer. A thorough understanding of design patterns and principles, algorithms and the programming language in use are a must have, you simply cannot be successful without it. So, software developers, who are still having trouble implementing an Observer or Visitor from the top of their heads, or who are still thinking the DRY-principle refers to alcoholic beverages and information hiding is about privacy in their e-mail communication, should definitely think about either getting educated or finding a new job. And while we are at it: No development process can compensate for lack of skill, knowledge and dedication; no process can make average or below-average people become outstanding.

Developing Software is an On-Going Process

Better software helps users achieve their goals. All requirements should therefore be goal-driven. And since goals change, we should expect our software to change as well. Developing software is an on-going process, a learning experience that never stops. Software should be an asset that can be easily changed to always offer value to its user in the best possible way. Software is never finished, merely fitting a purpose. So developing better software is an attitude of "good-enough" that strives for excellence not by aspiring some mystic form of perfection, but by providing concrete value in given circumstances. Therefore, better software is an attitude, a specific way to look at SD: It's about change, collaboration and problem solving based on explorative learning.

Successful Collaborations

"Successful collaborations are dreams with deadlines."

- Bennis & Biedermann

"When we set out to write software, we never know enough."

- Eric Evans

Domain experts and software developers must actively work together to add value; they must engage in successful collaborations, they must speculate and learn from each other and their mistakes. They must build trust and support; they must learn to welcome the fact that planning in a complex environment is speculation and that following a plan at best produces the results you intended, but just not the product you need. Solving complex problems means accepting the fact that deviations from plan are not mistakes that must be corrected but guides towards the correct solution. Solving complex problems needs rigor and discipline, but it also requires the courage to deviate and make mistakes; problem solving is iterative and incremental.

Doing it right the first time simply doesn't cut it for complex tasks. Developing software that provides value usually requires solving complex problems. Developing better software for a new problem domain is always risky; it is an exploration into the unknown. If you can't afford to take risks, don't embark on that journey, better stay home and hope for someone else to do the job. But if you have the guts, you must be willing to *get it wrong the first time in order to get it right the last time*; you must be willing to take risks. By aiming at doing it right the last time, you can actively manage risk, you can acknowledge uncertainty, you can allow for experiments, you can worship error as an opportunity for learning, you can expect to deviate from plan and let deviations guide you toward the right solution.

Tracer Bullet Development

"Ready, fire, aim…"

- Hunt & Thomas

One way to support this sort of concentrated successful collaboration is Tracer Bullet Development (TBD). The idea of Tracer Bullets was first introduced by Hunt & Thomas in their landmark book "The Pragmatic Programmer": When firing a machine gun in the dark, tracer bullets are a very effective means to find out if you hit the target. When a tracer bullet is fired it leaves a pyrotechnic trail from the gun to whatever it hits. If the tracer hits the target, then so do the regular bullets. Of course, you could also do various calculations in order to find out, where your target is. And if you have made no mistakes and the circumstances haven't changed between the time you started the calculations and

the time you actually aim and fire, you my even hit the target. Using a tracer bullet is the preferred way to aim in the dark, since it is easy, cost-effective and provides immediate feedback. TBD allows us to write code that glows in the dark.

TBD doesn't try to change the way you work; it wraps around the way you are already working. This is important, since any process must fit your team and environment. TBD is very light-weight, it doesn't prevent any best-practice from being used, being as non-invasive as it can be. TBD allows you to create an end-to-end system as quickly as possible, so you can have early and constant feedback on where you are and where you are going. It ideally supports the adaptive cycle of speculation, collaboration and learning. At first, all the components of your systems are merely hollow objects: you write code for the major parts of your system, but the objects aren't doing any work.

While a detailed description of TBD is beyond the scope of this paper, let's briefly look at the key elements of this approach:

You start by identifying the major parts of your system, and divide your product into blocks of related functionality, the so called system objects. Then you flesh out the interfaces that are needed for the blocks to interact with each other. Of course, you don't expect to get it right the first time. Now write just enough code to make everything look as if it works. Basically, you are writing an entire application of mock objects. With this thin, skeletal framework in mind, you can fill in the real logic inside each block as part of iterative and incremental collaborations.

TBD is consistent with the notion that software is never finished: there will always be changes and new functions to be incorporated into the product. TBD helps us to *think applications and not projects*. This is important, because projects are disruptions that need to come to an end as soon as possible, while applications are long-lived assets, that constantly need to evolve; applications are never finished, only retired. Closely related to this are test-driven development, refactoring and continuous integration: they are vital and indispensable aids for the software craftsman that sees his or her job in providing ongoing value to users.

User-Centered Interaction Design

"Don't make me think!"

- Steve Krug

Providing value to users is not only about features. Feature-rich software seldom offers its users the best possible value – a sad fact to which many of today's shrink-wrapped applications give a living testimony. In order to provide value to the user, software developers not only need to understand the problem domain and find ways of designing accurate solutions. Software developers also must come to understand how users think about specific domain-problems, they must get a clear understanding of the users' mental models.

Any model is an abstraction of reality, it is based on selective ignorance and it governs the way we look at things. In software there are two important models: For one, there is the user's mental model, his or her way of thinking about the problem and its possible solutions. Furthermore, there is the implementation model, the actual technical solution to the user's problem. Usually, the two models are quite different. That is why a domain model that closely

matches the user's mental model is vital to better software, since it offers the necessary grounds for successful collaborations that involve users and software developers on equal grounds.

Basically, domain-driven design tries to move the developer closer to the problem domain. But this is not enough; we must also find a way to see the application through the eyes of its specific users. That is, we have to do user-centered interaction design. Interaction design approaches the design of software products with a goal-directed perspective. It is a synthesis of traditional design, usability, cognitive science and software design. In short, it is about "humane software" that takes into account the human strengths and weaknesses to help develop software that adds value.

A detailed treatment of user-centered interaction design is beyond the scope of this paper. Suffice to say that domain-driven design and user-centered interaction design are complementary aids. Since most developers neither are trained interaction designers nor have the time to become experts in this field, it pays to have an interaction designer on your team, at least if you develop software that requires end-user interaction. Anyway, it helps to remember two important rules in interaction design: First, don't make the user think! Second, imagine your user intelligent, but very busy! So, whenever you need to decide on a particular solution that is directly or indirectly revealed to users of your software, just remind yourself that users have their own mental models that you need to understand and that users do not care about – nor should they have to – the technical details of your solution.

Domain-Driven Design and Domain-Specific Languages

As mentioned before, domain-driven design tries to move the developer closer to the problem domain. Domain-driven design requires and fosters the use of a common language – Eric Evans calls it the ubiquitous language - that is based on a simple metaphor, deeply rooted in the subject matter domain. Ideally, we receive a supple domain model that is illustrative and concise and that can be implemented using a general purpose programming language like Java. The better our understanding of the problem domain, the better we can express our intensions in code. And since code is the most detailed specification - the only description of the domain model that accurately describes the actual system behavior - expressive code plays an important role in the development of better software: Programming by intension, unit tests that not only assure our code's correct functioning but that also document its correct use, its capabilities and limitations, and the ability to easily refactor code as need be, are indispensable elements of success. But even if you carefully craft your solutions, using the correct mix of design patterns, intention revealing interfaces that are easy to use right; even if you provide side-effect-free functions, loosely coupled components, and design protocols in a "tell, don't ask"-style, you encounter a subtle but critical problem: the mix of subject matter problems and technical aspects. So, even if you "write it shy, make it DRY, and tell the other guy", you are mixing domain and technical problems. At first, this doesn't seem like a big thing, but the more complex the problem is, the greater the risk becomes that we end up with additional, inadvertent complexity that results from the intermingling of domain knowledge and technical aspects.

To solve this problem, we need higher level abstractions to describe solutions for a specific problem domain; we need executable models that move us even closer to the problem domain. We need Domain-specific Languages that abstract away from the underlying technical details by using concepts and terms directly related to the problem domain. We want to be able to more directly deal with subject matter problems. This way, we are able to ignore all the technical details that are related to programming in a General Purpose Language (GPL) like Java. By avoiding the intermingling of subject matter problems and technical problems, we can increase the expressiveness of our domain models, while keeping the necessary flexibility on the technical level.

It is important to note that DSLs are not meant to replace GPLs; they complement each other. DSLs allow us to be more abstract and concrete at the same time: more abstract with regard to the technical intricacies and more concrete with respect to the domain problem at hand. GPLs provide us with the means to lay the technical foundation by providing the necessary frameworks and behind-the-scene implementations.

Domain-driven design using DSLs is not to be mistaken by model-driven development as proposed by the OMG in their Model-Driven Architecture (MDA). Note especially that we are talking about the use of DSLs to work closer to the problem domain. We do not suggest, as many MDA-proponents do, that we should or even could model on a higher level of abstraction simply by using a more abstract GPL. UML and xUML simply are a golden hammer that requires you to turn every problem into a nail.

It is really important to stress that a DSL is not a full blown programming language, its purpose and scope are limited to a specific problem domain, e.g. insurance, credit-loan handling, machine control software, etc. As noted before, a DSL allows us to describe solutions more concrete and more abstract at the same time by focusing on domain problems and ignoring the gory technical details. Those technical details are still relevant, but their solution should not “pollute” the domain model, they are simply part of the technical implementation. This implementation is a combination of GPL-code, actively generated from DSL-based domain models, framework and library code, augmented with hand-written GPL-code that deals with specific aspects that simply cannot be solved effectively and efficiently using a DSL. Some applications may also include DSL-interpreters that are capable of executing models at application runtime.

A note on code generation: It is an absolute must that all code generation be active, i.e. any generated artifact is read-only. It is absolutely forbidden to hand-change any generated element. If changes are necessary, do them in the model from which the generated artifact originates. It is crucial that there is one and only one authoritative source for any change. For this reason generated artifacts are normally not put under version control, your domain models, on the other hand, definitely must.

The higher the complexity of a software system, the more likely we are going to need several DSLs to address cross-cutting concerns. We, therefore, need a Multi-DSL development platform that allows for the seamless integration of all relevant artifacts. At the time of this writing, no such platform exists, though there are some promising projects out there. But even without such an integration platform, domain-driven design is a viable and worthwhile step in the right direction. So, even if using DSLs might not be on your short-term agenda, domain-driven design definitely should be.

In parting…

This article touches on many subjects and much more could and should be said about each of them. Problem solving, communication, domain-driven design, interaction design, DSLs, Test-driven development, Tracer Bullet Development, Active Code Generation and Multi-Level Domain-Integration, all deserve a more thorough treatment and articles of their own. I am well aware that my treatment is more than incomplete. Therefore, I have put together a list of further reading that might set you off in the right direction.

If you have any questions or suggestions, please do not hesitate to contact me; I am looking forward to hearing from you.

Resources

- **Domain-Driven Design:** Eric Evans, Domain-Driven Design; Tackling Complexity in the Heart of Software, Addison Wesley, 2004.
- **Tracer Bullet Development:** Jared Richardson & William Gwaltney Jr, Ship it! A Practical Guide to Successful Software Projects, Pragmatic Bookshelf, 2005.
- **Interaction Design:** Alan Cooper & Robert Reimann, About Face 2.0 – The Essentials of Interaction Desgin, Wiley, 2003.
- **Various information** in German on Domain-Driven Design, the use of Domain Languages, Test-Driven Development, etc.: http://www.empros.ch

About the Author

***Sascha Frick**, born in 1966, is founder and owner of empros gmbh. He has been a software developer, trainer and coach for more than 17 years. He has written various articles about and is a regular speaker on the subject of test-driven object-oriented software development. Under the nom de plume "nemo" he also maintains a weblog (http://www.empros.ch/nemoslog) where he critically looks into the trade of software developers.*

