Practical API Design

Confessions of a Java Framework Architect

Jaroslav Tulach

Apress®

Practical API Design: Confessions of a Java Framework Architect

ISBN-13: 978-1-4302-0973-7

ISBN-13 (electronic): 978-1-4302-0974-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Clay Andres
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Susannah Davidson Pfalzer
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Gina Rexrode
Proofreader: Liz Welch
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

Contents

About the Author

JAROSLAV TULACH is the founder and initial architect of NetBeans, which was later acquired by Sun. As creator of the technology behind NetBeans, he is still with the project to find ways to improve the design skills among all the programmers who contribute to the success of the NetBeans open source project.

Acknowledgments

This book could not have been written without the generous help of Geertjan and Patrick, the best editors I've ever met. Thank you and everyone else very much, guys. Visit `http://thanks.apidesign.org` to learn details.

Prologue: Yet Another Design Book?

"There are more than enough design books in the programming world already," you might think. In fact, there are so many that it makes sense to ask why I would write—and especially why you would read—yet another one. Particularly, there is the famous *Design Patterns: Elements of Reusable Object-Oriented Software*,[1] about design patterns in object-oriented systems, written by the so-called "Gang of Four," which is a must read for every developer making use of any object-oriented language. In addition, there are many specialized books describing design patterns, all of them useful when writing specific types of applications. Moreover, there is the unofficial Java programmer's bible, *Effective Java*.[2] In light of these facts, is there really a need for yet another design book?

I believe the need exists. I've been designing NetBeans APIs—that is, application programming interfaces—since 1997. I've passed through almost all the possible stages a person designing a framework or a shared library can pass. In the early days, I slowly gained a feel for the Java language while trying to apply coding styles that I knew worked well in other languages. Later, I became fluent in Java. At that point, applying various known patterns to my code written in Java seemed so simple, although after a while I realized that things are not always as easy as they seem. I realized that traditional patterns are not appropriate for an object-oriented application framework such as NetBeans, and that you need a completely different set of skills.

The oldest NetBeans APIs were designed in 1997. Some of them are still in use and working adequately even after ten years of service, although to be honest, these are not exactly the same APIs as they once were. Over the years, we needed to accommodate new requirements, extend library functionality, and fix beginners' mistakes. Despite that, the API clients that compiled their code then are still able to execute their code, even with today's latest versions of those libraries. This is possible because we always tried, as far as reasonably possible, to maintain backward compatibility. As a result, the programs written against our decade-old libraries are likely to work even in their current versions. This preservation of investment—that is, our decision to let our libraries evolve in a backward-compatible way—is something not seen in common design books, at least not in the ones I've read so far. It's not that all NetBeans APIs were evolved without problems, but I've come to believe that the NetBeans team has now mastered this skill to a high degree, and also that this skill is widely needed among other groups of programmers. That is why a large part of this book talks about retaining backward compatibility and about special API design patterns that produce code suitable for maintaining in a backward-compatible way.

---

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Upper Saddle River, NJ: Addison-Wesley, 1995).
2. Joshua Bloch, *Effective Java* (Upper Saddle River, NJ: Prentice Hall, 2001).

The other challenge we faced when working on the NetBeans project was scalability of teamwork. In those early days, back in 1997, I wrote the APIs on my own. The other NetBeans engineers "just" wrote the code; that is, they provided user interfaces and implementations for various parts of the NetBeans IDE, while continually making use of the APIs that I provided. Unsurprisingly, this created a bottleneck. I came to realize that the number of people working on various NetBeans IDE features had grown to a capacity where one "architect" was unable to handle the demand for APIs. Over time, change was urgently needed. We needed a majority of the NetBeans development team to be able to design their own APIs. At the same time, we also wanted to maintain a certain level of consistency between the APIs created by different people. This turned into the biggest problem, not because developers didn't want to be consistent, but because I wasn't able to explain to them what I meant by consistency. Maybe you know the feeling of knowing how to do something, without knowing how to explain it coherently. That was my situation: I thought I knew how to design APIs, but it took many months before I managed to formulate the most important constraints that I wanted others to follow.

A LATE NIGHT API TALK

I've been interested in API design and API publishing for a long time. I've given various presentations inside Sun Microsystems and for NetBeans partners on this topic. However, my first public talk that related to this topic took place during JavaOne 2005 in San Francisco. My friend Tim Boudreau and I had submitted a proposal entitled "How to Write an API That Will Stand the Test of Time." It was scheduled to start at 10.30 p.m., probably because the abstract didn't contain buzzwords such as "Ajax" and "Web 2.0." That time of night is not ideal for a presentation, as we had to compete with parties, free beer, and other late-night distractions. We were pessimistic and expected one or two friends to show up to console us. Our mood got even worse when we arrived at the venue and saw that right next door a JDBC driver presentation was going to be held. The corridor was filled with people. Our assumption was that everyone there was interested in the database stuff. However, to our surprise, most of the people there were waiting for our talk instead! Our room filled quickly. All the seats were taken. People started sitting on the floor or standing up against the walls; they even had to keep the doors open, and several had to listen from outside in the hallway. In the end, it was the most exciting presentation I've ever been involved in.

Since then, I've known the need for information relating to API design is real. The memory of that presentation encouraged me whenever I began losing motivation while writing this book. It helped to remind me that the rules for proper API design that we had discovered needed to be documented. These rules, though based on the knowledge spread across common design books, are highlighted and expanded upon in this book, because designing APIs has its own specific demands.

API Design Is Different

The reason why the existing design books are not enough lies in the fact that designing a framework or a shared library is a more complicated task than designing an in-house system. Building a closed system, such as a web application running on your own server with access to a private database, feels like building a house. Some houses are small, some are big, sometimes they're skyscrapers. However, in all cases, a house has one owner at a time and the owner is in charge of making changes. If necessary, the owner can change the roof, replace windows with new ones, build new walls inside rooms, pull down existing ones, and so on.

Of course, certain changes are easier to make than others. Replacing a roof is likely to cause no great harm to the floors. Changing windows for different ones of the same size is unlikely to influence other parts of the house. However, trying to replace windows for larger ones might not be that simple; doubling the size of an elevator is normally an almost impossible task. Except in rare situations, nobody is seriously going to experiment with inserting a new first floor while moving the existing first floor and the floors above it upwards. Doing so would cause so many problems that the benefits are unlikely to outweigh the costs. On the other hand, all this seems technologically possible. If the owner has the need, it can be done.

In-house software systems seem to exhibit similar behavior. There is typically a single owner and the owner normally has full control. If there is a need to upload a new version of some part of the system, it can be done. If you need to change the database schema, it can be done as well. Clearly, certain changes can be more complicated than others. A change to a database schema is likely to have a much bigger impact than a change to a one-line bug fix that prevents a `NullPointerException` somewhere. Still, any change is conceivably possible. The owner has full control, and if there is a real need for a major upgrade of the system, the owner can even shut the system down for a while. In addition, there are a lot of design principles to help us manage changes to in-house systems. There are books about design patterns that help developers structure their code. There are methodologies for designing systems and for testing their correctness. There are books describing how to organize and lead people's work. At the end of the day, maintaining an in-house system appears to be a pretty well-understood and documented process.

However, writing APIs is different. As an analogy, consider the universe. Though not as straightforward as the "house" analogy, this train of thought will prove useful. Let's start with recalling the "known" universe. I explicitly call it "known," as no human knows the whole of it—that is, all the existing stars, galaxies, other objects, and to provide an immaterial example, all the physical laws. Humans can only see just a small fraction of the universe, so far anyway. Our horizon defines all that is seen of the universe. In other words, what is before the horizon is the "known" universe. It contains numerous objects and effects, but our expectation and experience tells us that there are other stars and galaxies behind the horizon, and that these are thus far unknown to mankind. This experience is based on the fact that from time to time people manage to shift the horizon further and discover new objects or new rules, through building better equipment or by recognizing and understanding new laws of nature.

The universe is not constant; it's always changing. However, it doesn't change completely arbitrarily. Some rules guide what happens with planets, stars, and other objects. For example, if someone shifts the horizon and discovers a new star, it's no surprise that the star is going to be there tomorrow, and the day after tomorrow, and the day after that. Indeed, it can move, it can rotate, and it can even explode. However, the laws of nature guide all this. Nobody seems to be releasing Universe Milestone X every other week, where a star would appear, disappear, or move randomly. If the world behaved like that, it would clash with our understanding of the universe as we perceive it today. We simply know that *once a star is discovered, it is going to be with us forever*. We even believe that it stays there when nobody is watching. Well, obviously. The star can be observed by someone on earth, someone from another place in the solar system, potentially by some other creature in the universe, or by nobody at all. However, the star itself doesn't know if it's being observed, and the only thing it can do is to follow the laws of nature, and therefore, *once discovered, stay with us forever*.

Good APIs are similar. Once a shared library introduces a new function in some version, it's like discovering a new star. Anyone who gets the new version can see the function and can use it. They can, but don't have to, which depends on the programmer's horizon. It's possible

to add functionality that is almost invisible for most API users. However, you cannot rely on that. My experience tells me that API users are really creative. Sometimes the API user's horizon is farther than that of the API designer. If there is a way to misuse something, users are likely to do so. As a result, it's likely that neither the function itself nor its author know if it's being used and how often. There might be many users in the world, or there might be none, but unless you want to break the laws of good API design—that is, break backward compatibility—you have to assume someone is observing, and therefore, the function has to be kept and maintained. "APIs are like stars; once introduced, they stay with us forever."

There is yet another analogy between the universe and API design. It involves the way we improve our understanding of the universe and the way we evolve our libraries. Ancient Greeks could identify and observe the movements of the planets, all the way to Saturn and Jupiter, and thus define the planets of their "known" universe. However, although they tried to explain the reasons behind the planetary movements, they weren't successful according to our current standards. Laws causing the planets to move were still beyond the horizon. This continued during the Renaissance, when Nicolaus Copernicus proposed the heliocentric system and Johannes Kepler discovered his three laws describing the trajectories and speed of planets on their path around the sun. This discovery enriched the known universe by providing a very precise explanation of "what." However, nobody knew "why"! It took until 1687, when Isaac Newton provided an explanation of Kepler's laws, and introduced the notion of a physical force. This not only explained why Kepler's laws held true, but also started a magnificent expansion of the known universe, because physics could explain nearly everything happening between objects of the known universe, thanks to Newton's laws.

All seemed well until the end of the 19th century, where various measurements showed that there are behaviors, especially of objects with great velocities, unexplained by the use of Newton's laws. The accumulating evidence that something was not quite right helped Albert Einstein to discover his theory of relativity, which provided an enhanced understanding of the universe, including objects moving with very high velocities. In fact, Einstein's theory is an extension of Newton's: when objects move reasonably slowly, both theories yield the same result.

What does this physical and historical excursion have to do with API design? Let's suppose, for the next few paragraphs, that some god communicates with people through an API library. The library gives mankind an interface to the "known" universe. Ancient Greeks would be using version 0.1 of the library, which would only enumerate the planets and their names. It's clearly not a very rich API, but for some users and for some time, it may be enough. For example, it's enough to let us look at the planets and name them. Regardless of the state of an existing library, there are always going to be a few people suggesting improvements. Similarly, the universe 0.1 library was found insufficient because Kepler really wanted to understand the rules for the motion of planets. Therefore, the imaginary god of this paragraph gave him an update called universe 1.0. This version of the library could provide the space coordinates for each planet at a specified time, while the original functionality provided by the Greeks would stay the same and would continue to work.

However, users are never satisfied, and the physicists weren't either. That is why our imaginary god had to help Newton release a new major version called universe 2.0. Not only did this version provide information about the actual force of gravity between the sun and each planet, but also a handy set of subroutines to calculate forces, acceleration, and speed of the objects in space, not just limited to planets. Needless to say, all the functionality of the previous versions, provided by the Greeks and then by Kepler, would continue to work as in the previous versions.

Up to this point, all the additions were straightforward. The imaginary god of the previous paragraph simply added new features. But what to do at the points in history where physicists claim that all the laws of the universe are known and physics itself is seen as having nothing left to explain? Let's tease mankind! The imaginary god invents the Michelson experiment, which leads Einstein to formulate his theory of relativity. The latest version of the universe library now really faces the problem of no longer being easily backward compatible, because the new idea introduced into it is that everything the previous physicists did, including Newton, was slightly wrong! However, even such a radical change is manageable in backward-compatible mode. Only very high velocities are in danger of incorrect results. These velocities are much higher than Newton and his predecessors could measure, and therefore, although there was an incompatibility, nobody was able to prove an inconsistency in the previously performed measurements, or to prove that the behavior of the universe library had changed.

The moral of this absurd physics fable lies in the observation that our understanding of the universe is continually evolving. This is also the case with the API of our libraries. Although optimists might disagree, I am afraid that mankind will never understand the whole universe. Yet, my guess is that we'll continue to learn more about it. Although some developers might think differently, I am sure that the APIs of almost all our libraries in active use will never be final. They'll always evolve. We must be ready for that. We must be prepared to modify our understanding of the universe and we must be ready to enhance and improve our library APIs.

Different from building a house or an in-house software system, this requires developers to think about the future while coding the current version of their API. As far as I can tell, this is not a common approach taken with API design so far. Also, the current books and their suggestions don't help much with this kind of thinking. Their design patterns are mostly used to describe a single version. People who use them only think in the context of the current version. They often only minimally need to refer to older versions or only marginally think about what will happen in future releases. Still, these skills are needed when writing shared libraries and frameworks. We need to stop designing a house, and learn how to design a universe. We need to learn that "once an API star is discovered, it is going to stay with us forever."

Who Should Read This Book?

If you are holding this book while standing in a bookstore and deciding whether to buy it or not, you might be wondering if this book is for you. I cannot answer that question because I don't know you. However, I can explain why I needed this book myself and why I decided to write it. When I was designing the NetBeans APIs, I was learning on the fly. In the beginning, I was guided by instinct, and I thought that writing APIs was some kind of art. Well, that might be true, because you need to be creative. However, it's not just an artistic discipline. Over time, I started to identify a structure behind all the work I had done and I formulated measurable standards that turn an ordinary API into a *good* API.

This book describes the standards the NetBeans team adheres to when measuring the quality of our APIs. It also explains why we adhere to them. It took us several years of trial and error to get where we are now. Since reinventing the wheel is not the most productive expenditure of your time, I recommend this book to every API architect who prefers a bit more engineering design over a purely artistic one. In the beginning of NetBeans, I was the only person who wrote the APIs. At that time, we even believed that "a good API cannot be designed by committee." One designer is able to maintain consistency without any formal rules. However, one designer simply doesn't scale. We discovered this in the context of NetBeans, too. So, my

task was to find a way to let a broader set of people design our APIs, while maintaining overall consistency. At that time I started to write this book, to describe the theory behind API design, the motivation that leads us to write APIs, and the rules that we must adhere to when evaluating whether an API is good or not. Then I passed my approach to the developers working on NetBeans, I let them write APIs, and then I monitored and mentored them at the beginning and end of their design task. As far as I can tell, this has worked out well enough. Given that they've evolved for ten years and we've learned on the fly, our APIs are relatively consistent and satisfy most of our requirements. If you are in a position where you need to monitor the design of APIs, you may find this book's suggestions useful too.

When I was defining the meaning of the term "API" for myself, I found that it is in fact very broad. You don't need to write a framework or a shared library to write an API. Even if you just write a single class that is consumed by your colleague in the next office, you are in fact writing an API. Why? Because the developer who has to use your class isn't going to be very happy if you delete or rename methods it used to have, or if you change the behavior of the methods in your class. Exactly the same problems arise when writing an API for a shared library. You probably have more than one user of your class, and requiring all of them to do a rewrite when you change the class can turn into a nightmare of inefficiency. That nightmare should be completely unnecessary. Treat your class as an API and you'll have many fewer headaches. Moreover, it's not hard to think about it in that way. It means you need to design your class more carefully, evolve it in a compatible way, and apply other good API design practices. From this point of view, nearly every developer is or should be in the API design business.

An essential part of an API is the way it works. Testing plays an important role in describing how things work. It's nearly impossible to write a good API without properly testing it. Several chapters of this book outline testing patterns; that is, ways to test externally visible aspects of a library so that they hold true over multiple releases. I'll mention various kinds of tests, including signatures, unit tests, and compatibility kits. So, this book has value for people who need to check API compatibility too.

Last but not least, having a library that is in wide use can be a good asset for the person who creates it. Increasing this asset means satisfying your existing users, while attracting new ones to join and use it as well. Only with a sufficiently rich user base can you really monetize the work dedicated to creating and maintaining a library. This book discusses this too, and therefore can also be of interest to people examining software development from a more business-oriented perspective.

Is This Book Useful Only for Java?

NetBeans is an integrated development environment (IDE) and framework written in Java, and as most of my API design knowledge is based on working on the NetBeans project, it's correct to ask whether this book can be useful beyond Java development. My answer to this question is *yes*. In this book I discuss generic guidelines for good API design. These guidelines and principles are applicable to any API in any programming language. Discussions in this book include reasons why you would create an API at all, the rules and motivation for writing and structuring good API documentation, and the principles of backward compatibility. Such principles can be applied to a wide variety of languages, including C, FORTRAN, Perl, Python, and Haskell.

Of course, when it comes to more detailed descriptions, I cannot avoid mentioning features specific to the Java-like languages. First of all, Java is an object-oriented language. Designing an API for object-oriented languages has its own concerns due to all the support for inheritance, virtual methods, and encapsulation. So, some of the principles discussed in this book are more applicable to specific kinds of object-oriented languages, such as C++, Python, or Java, than to the "old but good" non-object-oriented languages, such as C or FORTRAN.

Also, Java belongs to the camp of newer languages that use a garbage collector. In fact, the widespread industry acceptance of Java has proven that it's possible and beneficial to use a garbage collector in production applications. In the pre-Java age, the industry preferred classical memory management, provided by common languages such as C, C++, and so on, where the developer explicitly controls allocation and deallocation. Languages with garbage collectors existing at that time, such as Smalltalk or Ada, were generally seen as being experimental or at least adventurous to use by most of the software industry. Java has changed this completely. Currently, the majority of software engineers no longer laugh at the notion of a high-performance programming language with an automatic memory management system, and programmers are no longer afraid to use one. However, an automatic memory management system has implications on the API you produce. For example, in contrast to C, Java requires only `malloc`-like constructs to allocate new objects, but there is no need for paired deallocation APIs. You get those for free. That means certain approaches in this book are more applicable to languages with a garbage collector—in other words, the newer languages that use the memory approach popularized by Java.

Java has also popularized the use of the virtual machine and dynamic compilation. During static compilation, Java source code is transferred to many class files. These are then distributed and linked together, but only during program execution. Moreover, these class files are in a format that is independent of the actual processor architecture on which the final application is executed.

This is achieved by a runtime environment that not only links individual class files together, but also converts their instructions into those for a real processor. During Java's early days, this was yet another area where Java deviated. Everyone knew that a well-performing program could not be interpreted by a virtual machine, that it needed to be written in FORTRAN and directly compiled to use the most optimized features of the assembly language on the operating system that runs it. Some, myself included, admitted that it was possible to write in C or C++ and produce fast programs as well, but again, the approach taken by Java was seen by many as unlikely to succeed.

However, time has shown that languages based on virtual machines have certain advantages. For example, all numeric types have the same length, regardless of the platform the program runs on, which greatly simplifies the need to understand the underlying architecture. Also, Java programs don't crash with a segmentation fault when something goes wrong. The virtual machine guarantees that memory won't become corrupted by improper C pointer arithmetic, and that variables will always have the correct type. Still, performance problems persisted in early Java implementations. However, over time even the interpreters sped up and were replaced with just-in-time compilers that produced code that was fast enough to attract additional new languages to try this "virtual" path too. As a result, currently the term "virtual machine" is accepted and has widespread use. Virtual machines are covered in this book, to some extent, although I spend more time concentrating on the format of class files, since that is the *lingua franca* of the virtual machine.

It's important to understand the format of class files to correctly grasp what a Java language construct means to the virtual machine itself. It's handy to speak its language and see the class file using the virtual machine's eyes. Though other programming languages, such as C, have their abstract binary interfaces (ABI) models, the one used by Java is special in at least two respects. First, it's naturally object-oriented. Second, it allows late linkage; that is, it contains far more information than a plain C object file would. As a result, the knowledge gained from studying the virtual machine is less applicable to the old but good non-object-oriented languages, although it can be useful for other modern languages that have a virtual machine mimicking Java's.

Java is also one of the first progenitors of associating documentation for APIs with the actual code. Java has popularized commenting of code for public use through the Javadoc. This makes the actual behavior of the APIs and their documentation much closer, allowing it more simply to stay up to date. Even though every other language allows commenting, the Javadoc actually produces browsable documentation from those comments and forms the basic skeleton that gives consistency to every API documentation in Java. On the other hand, this is no longer Java specific. This has proven so useful that almost every language created after Java includes a concept similar to the Javadoc. For other, already existing languages, additional tools retrofitting this association of code and documentation are being created. That is why, when analyzing the usefulness of the Javadoc and the pros and cons of its format for simplifying the understanding of an API's users, this book will make conclusions applicable to almost any programming language.

Java 5 has changed the Java language to provide support for generics. While this book doesn't want to be an ultimate source of information about this language construct, it cannot ignore it. Generics form an important new phenomenon in API design. Why new? Because traditionally object-oriented languages encouraged reuse by inheritance. The second most common form of code reuse—reuse by composition—was possible, but only as a second-class citizen. One of the most important reasons for this was that inheritance was built as a language construct, while composition could only be coded by hand and it was difficult to type correctly. At the same time, a stream of languages was produced that preferred reuse by composition and made inheritance a second-class citizen, especially in the cases of modern functional languages such as Haskell. Some people feel that both approaches have their benefits and spend a lot of time trying to marry object-oriented languages with polymorphically typed functional languages.

Generics in Java are the result of this kind of marriage. Some criticize them for being too complex, though my own research in 1997 implied that this could hardly be done in a better way. I like what the language team managed to achieve, as now inheritance and composition are more or less on par. That is why I'll talk about generics in this book as well. Doing so will bring parts of this book closer to languages such as Haskell.

There is another reason why this book can be applicable to other languages: it accepts Java as it is. It doesn't try to invent a new language more suitable to handle the API design problem. No, throughout the book we work with Java as we know it. All principles and recommendations are about specific coding styles, not about adding new keywords, pre- or post-conditions, or invariant checks. This is needed in software engineering, as often the language is a given, and the goal—for example, to produce a library for general usage—needs to be achieved within that constraint. This is not really surprising. Learning new APIs might require a bit of work, but it's nothing compared to learning a new language.

Since the language to be used is almost always a given, API design principles have to be expressed in that language. If it's possible to write a good API in C, there should be no reason not to write a good API in Java. That is why plain Java is good enough for this book. In short, this book has general parts applicable to any programming language. Other parts talk more about object-oriented concepts, and whenever we need to dive deeper, we demonstrate the case in Java.

Learning to Write APIs

Without a doubt, there are people who develop APIs correctly; otherwise there would not be as many great and useful software products out there as there currently are. However, sometimes it seems that the design principles, the *main rules* of API design, are acquired subconsciously. Designers tend to follow rules without actually knowing or understanding the original motivation leading to the choices they make. As a result, the subconscious knowledge of good API design is built by trial and error, which obviously takes time. Moreover, the result of this process is typically a loose collection of tips on how to do things "right." Though this is a useful step forward, such a collection often suffers from two problems. First, tips of this nature are often tightly tied to a particular operational area. For example, they might work fine for one project or for a dedicated group of people, but the usefulness to other teams or the applicability to other projects is not guaranteed at all.

Second, in these cases it's difficult to transfer knowledge to people with a different way of thinking. If your experience shows that Java classes are preferable to Java interfaces, while this experience is gained from working on a specific problem, the experience becomes difficult to transfer to a different scenario. You can try to convince others that this is the correct approach, but in the end, without an appropriate explanation of the related reasons, you can only hope for adoption on the basis of faith. Faith can only create believers and rejectors, which is not the intention of knowledge transfer.

SUBCONSCIOUS NETBEANS API DESIGN

It's fair to admit that the members of the NetBeans project went through such a period too. We went through many different experiences doing API design where we somehow "felt" what worked and what didn't. However, this knowledge wasn't built from the bottom up. That is, it wasn't built using serious reasoning, or an understanding of the reasons for such design decisions. It was more a feeling, and the reasons that created such feelings lay undiscovered in our subconscious. This formed a problem when trying to transfer knowledge to other people, because they simply lacked our experience and had no reason to trust us. This forced us to think much more deeply about the reasons and the actual experiences that helped us formulate the measurables of good API design. This book is a result of such thinking. We believe that our experiences exposed us to information that helped us to uncover the hidden logic behind the assumptions we'd been making. We turned this logic into something conscious, something that we have become aware of, and something we can reasonably explain to everyone who wants to listen.

The foremost questions that deserve an answer are, "Why create an API?" and, in fact, "What is an API?" This book discusses these questions in detail.

Our experience shows that even without reading, understanding, or even agreeing with everything we advise here, it's useful for everyone participating in the development of software products to understand our basic motivations and terms. This will lead to an increased awareness and understanding of the problem and bring its complexity out into the open. When all members of a development team can see the "API design" with their own eyes, communication is simplified and decisions need no explanation, because they become part of the shared knowledge base. In turn, this improves cooperation between members of a development team, as well as between the team and its distributed partners, which leads to better quality software.

That is why this book is intended for everyone. It explains the basic motivations to anyone who wants to listen, it provides examples and tricks useful for developers, it describes the aspects of good architecture for those who design them, and it provides measurable principles for assessing API quality.

If you are still asking yourself whether you should read the book or not, here is a much shorter answer: "Yes, you should read this book!"

Is This Book Really a Notebook?

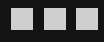
When I began thinking about the right style for this book, I examined a wide range of approaches available between two extremes. On one extreme, I could write a strict scientific description of the motivations, reasons, and processes required when practicing API design. This would produce a set of suggestions and rules applicable to any project. Of course, this is a specific goal of this book. It has to be generally applicable and not simply a description of what we did on the NetBeans project during the past ten years. On the other extreme, I strongly believe that advice without proper explanation is useless. I really dislike following the "what" while not being able to understand the "why." I always want to understand the context, evaluate various solutions by myself, and then choose the one that appears to be the best under the circumstances. That is why I also want to share with you the context that motivated us to accept our design rules. The best way to provide this context is to describe the real problems the NetBeans project faced at the time. As a result, this book is very close to a notebook.

Also, the lab journal format pretty much follows the process of creation of this book. It was not written in one go; its topics have been added over several years. Whenever we needed to solve a problem that looked general enough, I added a new topic to the book's table of contents, thought about the solution, and then later wrote it down. This was the most effective way of recording our rules, and indeed as a result, the final product resembles a lab journal: a lab journal where a note is not written per day, but per problem!

To get the best of both approaches, each topic analyzed in this book contains a note describing the real situation that the NetBeans project had to solve. The problem is then converted into a general recommendation applicable to any framework or shared library project. This resembles the "thought path" we used: first there was a problem, then we analyzed it and came up with rules to overcome it. As well, this gives the reader a chance to verify our "thought path" and check that the advice is really applicable in other situations and that our generalization is really correct. In all cases you can start with morphing the state described in the "note" into your own project, and then applying the same thought steps and checking whether they really result in our advice.

The world of API design is beautiful, and so far, mostly unexplored. Yet its knowledge is needed. The software systems being built today are becoming extremely large and we need to apply the best engineering practices to build them properly and make them reliable. API design is one such practice. Let this book be your guide for 21st century software development! Let our NetBeans API design adventures be your learning samples, and let the general advice extrapolated from them help you to eliminate similar mistakes. It's my hope that this book will help you pass through your API design phases smoothly and without reinventing the rules that we discovered on our journey, starting all the way back in 1997.

PART 1



Theory and Justification

The process of inventing, designing, and writing application programming interfaces (APIs) can either be seen as an artistic or as a scientific pursuit. Depending on your point of view, the API architect is an artist trying to change worlds or an engineer building bridges between them. Most people I know would rather be artists, because artists are associated with creativity, spontaneity, and beauty. However, a purely artistic approach has one significant problem: emotions are not transferable. They are extremely subjective. Explaining them to others is fraught with complexity. Although it is possible to create work that generates an emotion, ensuring that two people respond identically is beyond the capabilities, and probably also the goals, of art. As a result, an architect writing an API, just like an artist producing a painting, risks the possibility that the work will be misunderstood. In that case, developers using it will create a feeling very different from the architect's intentions.

This might not be all bad. However, problems arise when the API architect decides, or is forced, to develop an API in a group. Immediately, various attributes of an API are at risk. For example, one of the most important attributes of an API is consistency, which avoids unpleasantly surprising the API users. For an API to be consistent, there needs to be significant coordination between members of the designing group. Coordination is difficult to achieve when each member approaches the work as an individual artist. There needs to be a shared vision. There needs to be a vocabulary that the group can use to describe the vision. And there needs to be a methodology that can be used to create a result that fulfills the vision. As soon as this is recognized, the API architect probably begins praying for the API design process to become an engineering discipline, rather than an artistic one.

Anyone who has ever tried to organize the work of 20 artists, and compared it to the work of 20 engineers, can easily understand why.

DESIGN BY COMMITTEE

I designed and wrote most of NetBeans' APIs from its earliest days. At the time, we strongly believed that *a good API cannot be designed by committee*. That is why the goal for the other developers was mainly to use the designed API, write implementations of their features against it, and potentially comment on and improve what already existed. But gradually some of the other developers started to create their own APIs too.

I monitored such attempts closely. I commented on them, occasionally telling the others to do something differently or to remove pieces that looked odd to me. I sometimes even rewrote them myself and insisted that they use my version. However, I found myself in an increasingly uncomfortable position. Although I knew how I wanted the APIs to look, I was unable to explain my vision properly. Nor could I explain why they should consider my advice useful, because my colleagues did not always want to listen and follow my advice. I could use my right to veto their work, but that did not solve the problem either. I felt that something was wrong. However, I was unable to explain what or why. We lacked a shared set of terms, which limited our communication. This situation was inevitable because the APIs had been designed by means of a purely artistic approach.

I found proof of this a year later. One of the APIs developed further and, as NetBeans is an open source project, it attracted an external developer. At first, the external contributor worked on bug fixes. Next, he started his own subproject and designed its API. The original owner of the API came to me and asked me whether I could help him find reasons why the subproject's API was turning out badly: he did not like the newly created API and wanted to prevent it from being integrated into his project. However, he was not able to formulate what was wrong with it. At most, he managed to say that it did not adhere to his *original vision*. That was absolute déjà vu for me. I used to tell him that his work did not fit into the NetBeans API's vision—or at least my own interpretation of it!

Although it took me a while, after a few years of designing APIs and working with others to design APIs that I liked, I came to realize that API design can be approached as an engineering discipline. Although it's not obvious at first, this discipline does have a lot of objective attributes. As a consequence, it's possible to turn the *API design* process into something that engineers can understand—a kind of science.

Every real discipline needs a theory behind it: a theory that defines the world that forms the subject of its domain. Such a world needs to be sharp and clearly defined. The less sharp it is, the less rigorous the science is, and the more it becomes an art. This chapter defines the world of API design. It identifies the various objects you have to deal with in API design. It analyzes situations that you have to face during the process of designing an API. Moreover, it begins to build a common vocabulary that people knowing the same theory can use to better identify the objects of the API world and their relations. Based on

these fundamentals, we can later build in more complicated and less obvious conclusions about the subject of API design theory.

It's hard to write good APIs that can be consumed by a wide audience of users, especially an international one. Everyone has their own style of understanding and their own way of viewing problems. Satisfying all of them is hard. Satisfying all of them at once is usually impossible. Moreover, if your API is targeted to an international audience, it needs to deal with various cultural differences. That's why writing good, widely approachable APIs is hard.

Writing a book for an international audience is hard as well. Again, the matter of personal and cultural preferences influences the way people want to read. Some would like to understand the background first. Others would like to jump to examples directly to find out if the whole thing is useful to them or not. Satisfying both these camps at once seems impossible. Ultimately, as Edsger W. Dijkstra wonderfully describes in his text, "On the fact that the Atlantic Ocean has two sides,"[1] some people believe that a theoretical approach is difficult and boring and that, by extension, practical examples are much more interesting. They are probably right, at least in their cultural context. On the other hand, another group of people would rather spend time building a common vocabulary and increase their understanding of the world being explored step by step. This is difficult without a thorough introduction. Without it, terms can have dual meanings, often leading to confusion. Obviously, it is hard to satisfy both camps at once. However, I will do my best to make everybody happy.

A STABLE API

One of the basic terms in the world that our discipline wants to explore is *stable API*. I had been using this term quite a lot when talking to various people in the NetBeans project, without expecting any problem. All along, it seemed to me to be a word with a clear and obvious meaning.

Then I listened to an explanation of the term by one of my colleagues. He said that the *API is stable if and only if it will never undergo change*! Although a stable API is likely to have some kind of "stability," it will still need to change sometimes.

I face this kind of misinterpretation all the time, where a term from the basic API vocabulary has different meanings for different people. Of course, this ruins the whole conversation. If people think they understand each other, when in fact they do not, then it is better not to talk at all. The images that crystallize while talking are completely divergent, and as such are absolutely useless for communication.

That's why I believe it's good to spend some time defining basic terms.

---

1. Edsger Dijkstra, "On the fact that the Atlantic Ocean has two sides" (1976), `http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD611.html`.

To avoid confusion about terms, this part is dedicated to vocabulary building and analysis of the general aspects of API design. It builds a vocabulary of basic terms, describes the motivations of the whole API design effort, and outlines the main goals of the design process. If this isn't your preferred style of learning and if you believe you don't need this elementary material, feel free to jump directly to Part 2, which contains many more code samples, tips, tricks, and various hacks. Don't be surprised if the purpose of some of them seems strange. That might be a sign that you're missing the background material discussed in this part. Also, if you're eager to acquire more practical advice on tools, compilers, and so on, feel free to start reading Part 3. However, again, keep the previous warning in mind: if the advice given doesn't make sense, that might be due to a missing piece of understanding relating to the API design theory behind it.

Without further ado, let's dive into the theory. First we'll go over the basics, which will help us all to get up to speed. To get us in the mood for proper API design, let's start with the most basic questions that address the why, what, and how.

CHAPTER 1

The Art of Building Modern Software

The history of software development is short. It has been less than one hundred years since people wrote and managed to execute the first computer program. Although brief, this history is reminiscent of the history of any other intellectual invention. The most interesting parallel I've heard so far is the comparison of the history of computer science with the way people tried to understand the real world. One result of this comparison also yields an explanation of why a good application programming interface (API) is needed. Let's walk down that road now.

Rationalism, Empiricism, and Cluelessness

The renaissance of modern science seemed to create two major, yet extreme, philosophical approaches. Rationalism treated reason to be the primary source of information and postulated that using just a pure thought, it is possible to understand and describe the real world. The set of philosophers supporting this idea includes the progenitors of modern science Rene Descartes (1596–1650) and Gottfried Wilhelm Leibniz (1646–1716), as well as Benedict Spinoza (1632–1677), the creator of pantheism.

The initial impulse for this kind of understanding came from Galileo Galilei's law of falling objects, which postulates that two objects regardless of their weight always fall with the same acceleration. This goes completely against natural expectations, as anyone who has tried to drop a brick and a piece of paper knows that they are unlikely to reach the ground at the same time. The brilliance of Galileo and other modern scientists was that he attributed that to mutual cooperation of various natural laws, where the law of free fall was just one of them. How did Galileo discover his law? He did a mental experiment. He imagined two solid balls of the same size and weight being dropped. Indeed, they would reach the ground at the same time. Then he imagined the same experiment but with one solid ball and one ball cut in the middle into two parts, but with both parts being closely attached to each other. The result of this experiment ends up exactly the same as the first one—both objects fall synchronously. Now what happens if we slowly start to separate the two pieces of the ball? We can even keep them connected with a small wire to still form one body. Indeed, regardless of the two halves of the ball being a centimeter, meter, or even further apart, this object would continue to fall with the same speed as the full solid ball. And last but not least, the same result would be obtained even if we remove the wire! This result is completely against natural experience.

Experience says that a piece of paper falls more slowly than stone. This pure mental experiment explains that weight does not affect acceleration of falling objects.

UNCONSCIOUS MATH AND PHYSICS

You'll find, while reading the book, that I am obsessed with physics parables. Yes, I am, because, after reading Petr Vopěnka's book[2] about the importance of unconsciousness in the success of modern mathematics and physics, I just cannot get his philosophical explanations from my mind. From time to time I reuse some of his observations, however only in a very condensed form, as his book has more than 800 pages and carefully builds proper understanding of all the terms. That is not the case for this book. Deep explanation of all his concepts is beyond its capacity and purpose. As such, please excuse occasional simplifications.

People say that Galileo discovered his famous law by throwing stones from the leaning tower of Pisa. Maybe he really tried that, but it was the mental experiment that could explain the behavior without doubts. It was the first time that just plain thought could prove observation and experience wrong. Although in reality lighter objects fall more slowly than heavier ones, which is what we know from our experience, we now know that other laws interfering with gravity cause differences in falling speed. This was the experiment that showed the power of pure reason and that gave Leibniz and Descartes their impetus to favor reason over experience. It was the catalyst for the whole philosophical movement of rationalism. Indeed, this approach believes that the subject of research is and has to be reasonable. Indeed, if discoverable by reason, it needs to have a reasonable origin.

On the other side of the English Channel there was empiricism. Nearly at the same time, great British minds such as David Hume (1711–1776), John Locke (1632–1704), and George Berkeley (1685–1753) insisted that the primary source of understanding is experience. Without seeing, hearing, or feeling the world, the mind has no chances to "think it up." To understand means to experience—or, in a more scientific way, to do experiments. Even here we can trace the roots of Galileo, the first scientist who propagated scientific experiments as a source for verification that an idea or a hypothesis is valid. From the empiricist point of view, the world does not need to be reasonable. It might not be fully known, it might even not exist at all, and in fact doesn't matter. It isn't necessary to understand it all if the perception of the senses makes sense.

From today's point of view those two extreme ways to perceive the world are not in fact that far away. At least current science understands the value of an experiment to verify its theories. Also, Descartes understood the need for an experiment in science as well. So for us it shouldn't be a big problem to merge two opposing views into one. And yes, these days it's quite easy to do so. For most of our lives we don't care much about the philosophical aspects of our surroundings, we care more about the results. Life is supposed to be entertaining, not boring, and reasonable. However, things we use daily "just" need to work—we usually don't care *how* they work. For example, we're completely clueless about cars and mobile phones. We feel it's reasonable to use them, we just have no clue how they do what they do. We live in total cluelessness.

---

2. Petr Vopěnka, *Úhelný kámen evropské vzdelanosti a moci* (Prague: Práh, 1999).

RATIONAL APPROACH TO CLUELESSNESS

Writing APIs and books for an international audience is hard. Personal preferences and also cultural differences influence the way we approach problems that we face. Rationalists prefer to talk about theory—about the internal connections behind real objects—and only later they create real examples mapping the theory to the real world. Empiricists, on the other hand, would like to gain as much practical experience as possible, and only later, if ever, make judgments about the relation between objects of the world.

This book explains API design from the viewpoint of selective cluelessness. It sees APIs as a perfect tool to help us maximize cluelessness, while getting reliable results. It is essential to get a correct feeling for what cluelessness really is. However, we'll build our understanding of that term from a rationalist's point of view—we start with theory and not examples. This might not be the preferred approach for everyone; however, I cannot satisfy both camps at once. Anyway, do not despair—as soon as the theory is over, and we have a common vocabulary for the science of API design, there will be more than enough practical applications.

Cluelessness is a way of life for a majority of us. It is the result of the merging of rationalism and empiricism that applies these days. It is everywhere around us. It is present even in the way we program and do software engineering.

Evolution of Software So Far

In the 1940s and early '50s, programming was hard. People had to learn machine code to speak the computer's language, know the sizes and number of registers, and in worse cases even handle the screwdriver and connected wires that physically carried the signal between individual computing units. The ratio between the work needed to think up an algorithm and the slavery to turn it into an executable program was harshly tilted toward the boring, mechanical jobs.

FORTRAN was like heaven-sent simplification. Just like an empiricist, it allowed programmers to perceive the world of computation of mathematical formulas with just limited senses. Programmers no longer needed to understand assembly language or worry about the technical internals of computers. They could completely forget about these details and concentrate much more on the important thing—on converting a mathematical formula into algorithmic steps to compute it. FORTRAN simplified the software development process while only minimally limiting the things people could compute: a huge win for empiricism.

However, programming still was not an easy discipline and the appetite for simplicity continued to grow. New languages such as COBOL came up with visions such as "approachable by novice programmers" and "language readable by management," and simplified certain tasks associated with programming even more. Today nobody is seriously considering writing a new system in COBOL. However, in those days COBOL significantly reduced the amount of knowledge needed to access and manipulate a database compared to what was needed with plain assembler or even FORTRAN. Empiricism was on the rise.

However, not everyone liked this. There are and always have been people who believe in reason—who think that the world and things in it should be reasonable. Those people can be found all around us, even among programmers. In the '50s, rationalists such as John McCarthy