

**WP Translation plugin**

Compatible with all Theme & Plugins (incl. WooCommerce).  
SEO optimized. Join 20,000 users.



Advertisement

CODE &gt; TOOLS &amp; TIPS

# 3 Key Software Principles You Must Understand

by [Chris Peters](#) 7 Sep 2012Difficulty: Intermediate Length: Long Languages: English ▼

Tools &amp; Tips

Web Development

Software



If you're in software development, new techniques, languages and concepts pop up all of the time. We all feel those nagging doubts every now and then: "can I keep up with the changes and stay competitive?" Take a moment, and sum a line from my favourite movie, Casablanca: "The fundamental things apply, as time goes by."

## Republished Tutorial

Every few weeks, we revisit some of our reader's favorite posts from throughout the history of the site. This tutorial was first published in April, 2012.

*What's true for love, is true for code.*

What's true for love, is true for code. The fundamental things will always apply. If you have an understanding of the underlying ideas of software development, you will quickly adjust to

new techniques. In this tutorial, we will discuss three basic principles and mix them with many more. They provide a powerful way of managing the complexity of software. I'll share some of my personal opinions and thoughts, which, hopefully, will prove useful when it comes to applying them to code and real-world projects.

## Principle - Don't Repeat Yourself

*A basic strategy for reducing complexity to manageable units is to divide a system into pieces.*

This principle is so important to understand, that I won't write it twice! It's commonly referred to by the acronym, DRY, and came up in the book [The Pragmatic Programmer](#), by Andy Hunt and Dave Thomas, but the concept, itself, has been known for a long time. It refers to the smallest parts of your software.

When you are building a large software project, you will usually be overwhelmed by the overall complexity. Humans are not good at managing complexity; they're good at finding creative solutions for problems of a specific scope. A basic strategy for reducing complexity to manageable units is to divide a system into parts that are more handy. At first, you may want to divide your system into components, where each component represents its own subsystem that contains everything needed to accomplish a specific functionality.

For example, if you're building a content management system, the part that is responsible for user management will be a component. This component can be divided into further subcomponents, like role management, and it may communicate with other components, such as the security component.

As you divide systems into components, and, further, components into subcomponents, you will arrive at a level, where the complexity is reduced to a single responsibility. These responsibilities can be implemented in a class (we assume that we're building an object-oriented application). Classes contain methods and properties. Methods implement algorithms. Algorithms and - depending on how obsessive we want to get - subparts of algorithms are calculating or containing the smallest pieces that build your business logic.

***The DRY principle states that these small pieces of knowledge may only occur exactly once in your entire system.***

They must have a single representation within it.

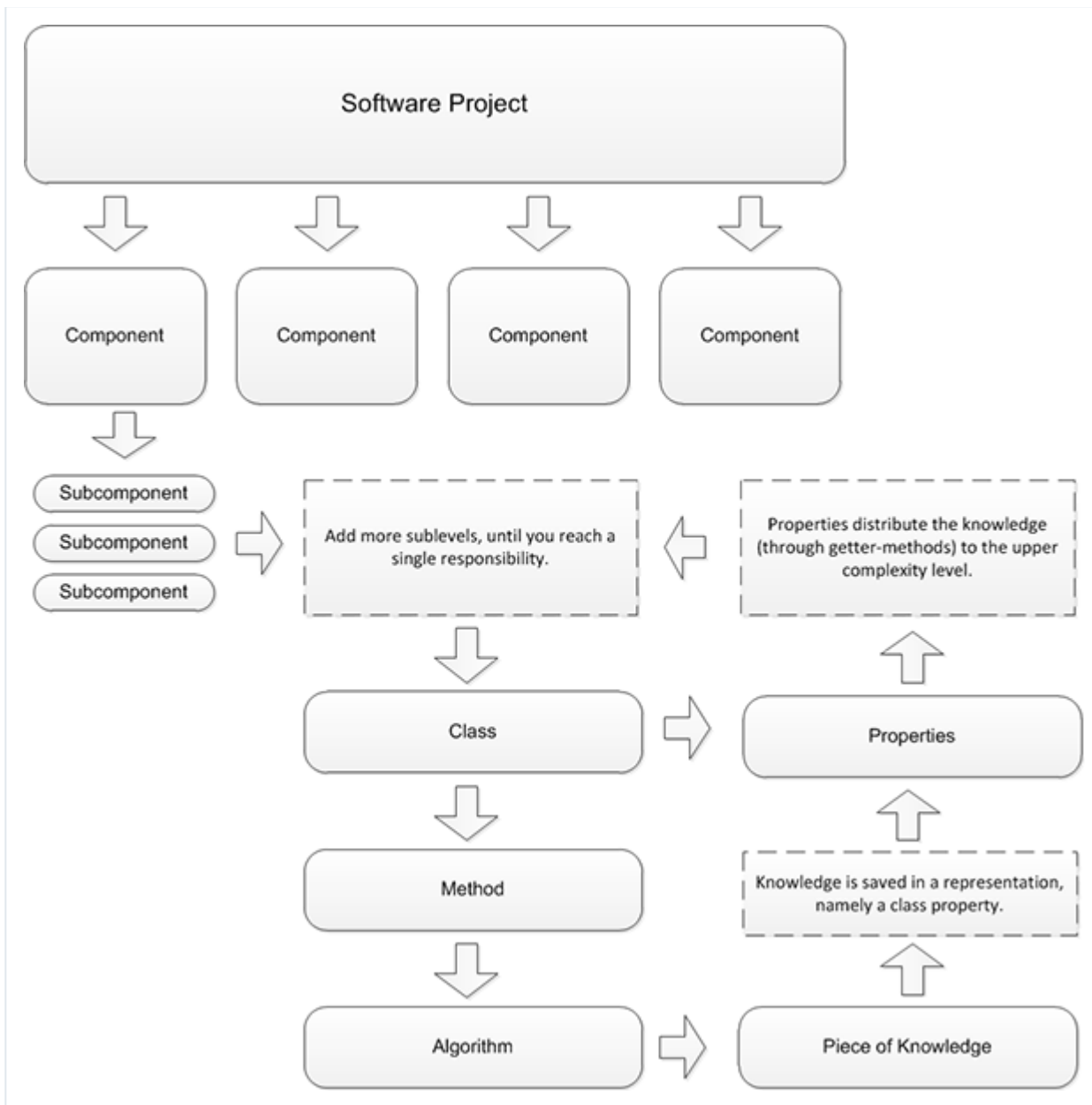
***Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.***

Note the difference between the *piece of knowledge*, and its *representation*. If we're implementing the database connection in our CMS, we will have a code snippet that will initialize the database driver, pass the credentials, and save a reference to the connection in a variable. The code snippet is part of the knowledge, it's about *how something is achieved*. The variable with the reference to the connection is the representation of that knowledge - and this can be used by other parties. If the database credentials change, we will have to change the snippet - not its representation.

In a perfect application, every small piece of business logic encapsulates its knowledge in a representation, namely a variable or a class property.

This variable itself is encapsulated in a class that can be described as a representation of a responsibility. The class is encapsulated in a component that can be described as a representation of functionality.

This can be proceeded until we reach the top level of our software project - that is, a stack of representations with increasing complexity. This way of looking at the complexity of software is called modular architecture, and DRY is an important part of it.



Software architecture is about managing complexity.

## Achieving DRYness

***DRY is a philosophy that packages logic into representations.***

There are many ways of achieving DRYness. Hunt and Thomas suggested (among other things) code generators and data transforming. But, essentially, DRY is a philosophy that packages logic into representations.

As every part of your application can be seen as representation, every part exposes specific fragments of your underlying logic: The user management exposes access to registered

users of the CMS, the user class represents a single user and exposes his properties (like the username). It retrieves the properties, via the representation of the database.

DRY and modular architecture require good planning. To achieve a representational hierarchy from bottom-up, divide your application in a hierarchy of logically separated smaller parts and let them communicate with each other. If you have to manage larger projects, organizing them into components and using DRY within the components is a good idea. Try to apply the following rules:

- Make a visual hierarchy of your software application and map the main components to it. Complex projects may require a dedicated map for each component.
- If you're arriving at a level of connected responsibilities, you may want to switch to UML diagrams (or similar).
- Before writing a chunk of code, name its hierarchy in your software project. Define what it's representing, and be sure you know its role in the surrounding component.
- Define what the representation should expose to other parties (like functions to execute SQL in a database driver) and what it should hide (like the database credentials).
- Ensure that representations do not rely on representations of another complexity level (like a component that relies on a class in another component).

The database driver is a simplified example, as there are many more layers involved in the real world (such as a specific database abstraction layer), and there is much more you can do to encapsulate logic - especially diving into design patterns. But even if you've just started with coding, there's one thing to keep in mind:

***When you find yourself writing code that is similar or equal to something you've written before, take a moment to think about what you're doing and don't repeat yourself.***

In the real world, applications that are a 100% DRY are hard, if not impossible, to achieve. However, applications that are unDRY to an unacceptable degree - and therefore hard to maintain - are quite common. Hence, it's not surprising to learn that more than 50% of all software projects fail - if you're taking a look at the code.

Many people tend to think that bad code is produced by bad coders. In my experience, this is very much an exception. More often than not, bad code is produced by bad account managers and an overall misconfiguration of process management in companies.

***Bad code is rarely produced by bad coders.***

## **An Example**

***DRYness is achieved by good planning.***

As an example, say you're hired as a technical consultant by a company that has problems with code quality and maintenance. You review the source and you see hacks and code duplication - the code is not DRY. This is a symptom of bad code quality, it's not the reason. If you take a look at the version control system - aka the history of the code - chances are that you may find hacks that were introduced at times near deadlines and milestones. Take the time to review what changes are made, and you will likely be confronted with a change in requirements.

As noted above, DRYness is achieved by good planning. Forced changes on a tough deadline are forcing developers to implement dirty solutions. Once the code is compromised, the principle of DRY is likely to be sacrificed completely upon further changes.

There's a reason why the most successful corporations in the IT business were founded by people with very good technical understanding - or even coders themselves: Bill Gates, Mark Zuckerberg, Steve Wozniak, Steve Jobs, Larry Page, Sergey Brin and Larry Ellison know (or knew) what efforts are needed to implement something. Contrary, many companies tend to lay the requirements for engineering into the hands of account managers, and the conceptual part in the hands of business consultants...**people who have never implemented anything.**

Hence, many technical concepts work only in Powerpoint, Photoshop, and on 27" widescreen displays. This may have been a successful approach in the days of, more or less, static websites, but it's not nowadays - with interactive applications on multiple devices. Because coders are the last in the line, they are the ones who have to apply quick

fixes on errors in the concept. If this is accompanied by an account manager, who can't stand up to a client that likes to make last-minute changes, plans are thrown in the garbage, and something quick and dirty is implemented. The code becomes unDRY.

This example is a bit extreme (nevertheless, I have witnessed such scenarios), but it demonstrates that DRY is a theoretical concept, which is challenged by various parties in the real world. If you're working in a company that forces you to work in this manner, you might suggest some changes to the process (like introducing technical expertise at an earlier stage of technical projects).

If you have a hands-off approach, keep reading! The *You ain't gonna need it* principle will come to the rescue.

## Principle - Keep it Simple Stupid

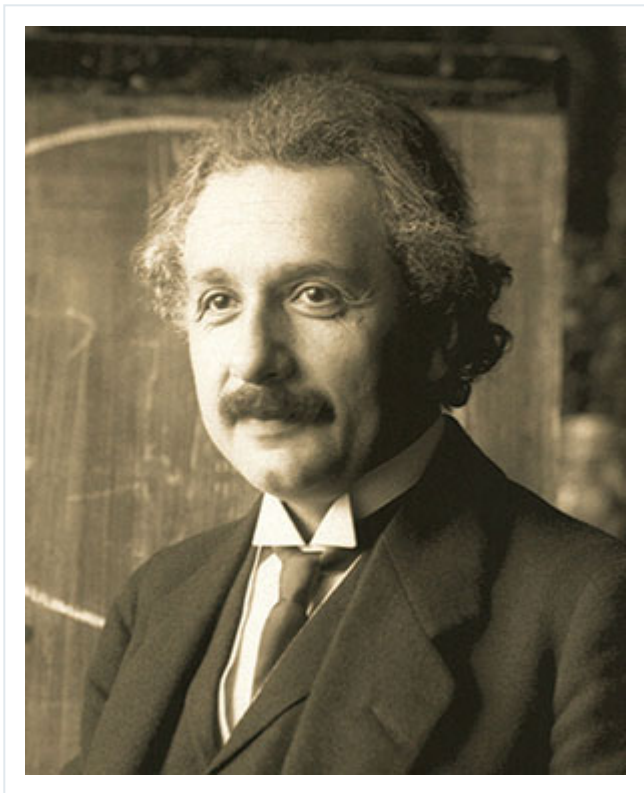
*The simplest explanation tends to be the right one.*

In the late 19th century, physicists struggled to explain how gravity, magnetism and optics interact, when it comes to large distances - like the distances in our solar system. Hence, a medium named aether was postulated. It was said, that light is traveling through this medium, and that it's responsible for effects that couldn't be explained otherwise. Through the years, the theory was expanded with assumptions that adjusted the aether postulate to the results of experiments. Some assumptions were arbitrary, some introduced other problems, and the whole theory was quite complex.

An employee of the swiss patent office, Albert Einstein, suggested to get rid of the whole aether theory when he introduced a simple, yet revolutionary idea: All the oddness in calculating with large distances would fade away if we'd accept that time is not a constant; it's relative. This incredibly of out-of-the-box thinking to come to the simplest explanation with the fewest assumptions to select between competing scenarios is referred to as *Ockhams's Razor*.

There are similar concepts in many areas. In software development (and others), we refer to it as KISS. There are many variants for this acronym, but they all mean that you should

strive for the simplest way of doing something.



Substantial progress in the history of mankind was achieved by lateral thinkers.

## HTTP

The *Hypertext Transfer Protocol* is widely considered to be a perfect example for a simple solution: designed to transfer hypertext based documents, it is the backbone of highly interactive and desktop-esque applications nowadays. Maybe we have to find solutions for limitations in the protocol, and maybe we have to replace it someday. However, status quo is: based on a few request methods (like GET and POST), status codes and plain text arguments, HTTP has proved to be flexible and robust. That's why HTTP has been repeatedly pushed to the limits by web developers - and is still standing.

We take this approach for granted, but the history of software development and standardization is full of overly complex and half-baked solutions. There's even a dedicated made-up word for it: bloatware. Software like this is also described to be *DOD*, dead on arrival. I have a theory that is very similar to my theory of unDRY code, when it comes to bloatware ... However, the success of the internet can be described as a success of simple, yet efficient solutions.



So what's required to come to the simplest solution possible? It all comes down to maintainability and comprehensibility in software development. Hence, KISS kicks in during the phase of requirements engineering. When you think about how to transform a client's requirements to implementable components, try to identify the following parts:

- Functionality that has an inappropriate ratio between benefit and efforts.
- Functionality that is highly dependent on other functionality.
- Functionality that is likely to grow in complexity.

***There are many people involved in the conceptual process, who do not have the technical expertise to make a reliable cost-benefit analysis***

I was once working on a project, where the client wanted to import Excel spreadsheets into his crew management software. This was a clear match. Excel is a proprietary software with a complex document format. The format is complex, because it's feature-rich: You can add graphs and other things to it - features that were not needed by the client. He was simply interested in the numbers. Thus, implementing the Excel import would require the implementation of a lot of unnecessary functionality. On top of that, there are multiple versions of Excel versions, and Microsoft fires off another release each year. This would have been hard to maintain, and it comes with additional costs in the future.

We ended up implementing a comma-separated-value import. This was done with a few lines of code. The overhead of the data was really small (compare an Excel sheet to its CSV equivalent) and the solution was maintainable and future-proofed. Excel was ready to export CSV anyway (as well as other programs that the client might want to use in the future). Since the solution was low-priced as well, it was a good application of the KISS principle.

To sum up: try to think out-of-the box if a task looks complicated to you. If someone is explaining to you his requirements, and you're thinking that it'll be tough and complex to implement, you're right under almost any circumstances. While some things are just that - hard to implement - overcomplicated solutions are quite usual. This is the case because there are many people involved in the conceptual process, who do not have the technical expertise to make a reliable cost-benefit analysis. Hence, they don't see the problem.

Double-check the requirements whether they are really stripped down to the essence that the client needs. Take the time to discuss critical points and explain why other solutions might be more suitable.

## Principle - You "Ain't Gonna Need It

*Coding is about building things.*

When Google+ launched, Mark Zuckerberg - founder of Facebook - was one of the first who created an account in the social network that was aiming to take his own down. He added just one line to the *About me* section: »I'm building things.«. I honestly think that this is a brilliant sentence, because it describes the pure essence of coding in a few simple words. Why did you decide to become a coder? Enthusiasm for technical solutions? The beauty of efficiency? Whatever your answer is, it may not be »*building the 1.000.001th corporate website with standard functionality*«. However, most of us are making money that way. No matter where you are working, you'll likely be confronted with boring and repetitive tasks every now and then.

***80% of the time spent on a software project is invested in 20% of the functionality.***

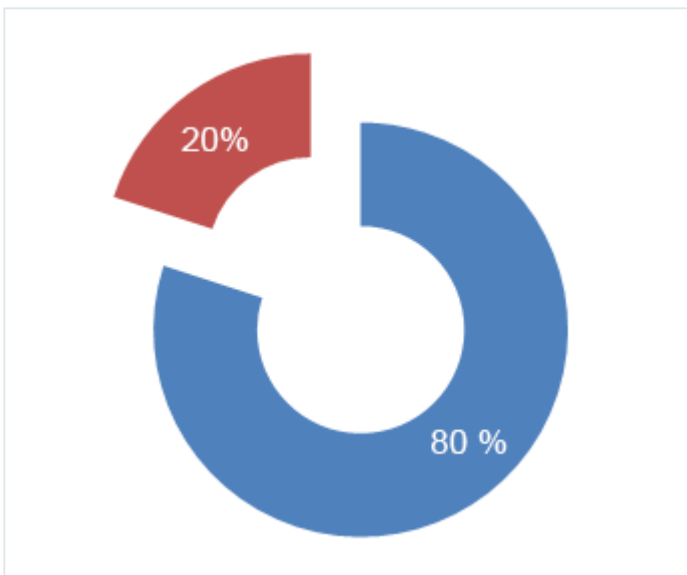
The *You ain't gonna need it* principle (YAGNI) deals with these tasks. It basically translates to: If it's not in the concept, it's not in the code. For example, it's a common practice to abstract the database access in a layer that handles the differences between various drivers, like MySQL, PostgreSQL and Oracle. If you're working on a corporate website that is hosted on a LAMP stack, on a shared host, how likely is it that they will change the database? Remember that the concept was written with budget in mind.

If there's no budget for database abstraction, there's no database abstraction. If the unlikely event of a database change does occur, it's a natural thing to charge for the change request.

You may have noticed the difference between *You ain't gonna need it* and *DRY-driven* modular architectures: The latter is reducing complexity by dividing a project into manageable components, while the former is reducing complexity by reducing the number

of components. YAGNI is similar to the *KISS* principle, as it strives for a simple solution. However, *KISS* strives for a simple solution by trying to implement something as easily as possible; YAGNI strives for simplicity by not implementing it at all!

*Theodore Sturgeon*, an American sci-fi author, stated the law: »*ninety percent of everything is crap*«. This is a very radical approach, and not overly helpful in real-world projects. But keep in mind that "crap" can be very time consuming. A good rule of thumb is: roughly 80% of the time spent on a software project is invested in 20% of the functionality. Think about your own projects! Everytime I do, I am surprised by the accuracy of the 80:20 rule.



80% of the time spend on a software project is invested in 20% of the functionality.

If you're in a company that is notorious for tight deadlines and imprecise concepts, this is a powerful strategy. You won't be rewarded for implementing a database abstraction layer. Chances are that your boss does not know what a database abstraction layer even is.

While this concept may sound simple, it can be hard to differ the necessary from the unnecessary parts. For example, if you're comfortable with a library or a framework that uses database abstraction, you won't save much time in dumping it. The key concept is another way of looking at software: we're trained to write future-proof and maintainable software. This means that we are trained to think ahead. What changes may occur in the future? This is critical for bigger projects, but overhead for smaller ones. Don't think into the future! If a small corporate website does fundamental changes, they may have to start from scratch. This is not a significant problem compared to the overall budget.



## Planning a Project

When you're preparing your to-do list for a project, consider the following thoughts:

- Achieve lower complexity by reducing the level of abstraction.
- Separate functionality from features.
- Assume moderate non-functional requirements.
- Identify time consuming tasks and get rid of them.

Let's go a little bit into detail! I already provided an example for the first item in the list: don't wrap a database driver around a database abstraction layer. Be suspicious of everything that adds complexity to your software stack. Notice that abstraction is often provided by third party libraries. For example - depending on your programming language -, a persistence layer, like Hibernate (Java), Doctrine (PHP) or Active Record (Ruby) comes with database abstraction and object-relational mapping. Each library adds complexity. It has to be maintained. Updates, patches and security fixes have to be applied.

We implement features everyday, because we anticipate them to be useful. Hence, we think ahead and implement too much. For example, many clients want to have a mobile website. Mobile is a term of wide comprehension; it's not a design decision. It's a use case! People who are using a mobile website are, well, mobile. That means they may want to access other information or functionality than a user who visits the site laid back at his desktop. Think of a cinema site: Users on the bus will likely want to access the starting time of upcoming movies, not the 50 MB trailer.

## ***Bad concepts can often be identified by the lack of non-functional requirements.***

With an appropriate budget, you would perform a dedicated analysis of the requirements for mobile. Without this analysis, you will simply provide the same information as is on the desktop site. This will be just fine for many circumstances! Because mobile browsers are very clever in adjusting desktop sites to their display, a radical YAGNI approach might be to not write a mobile site at all!

Non-functional requirements do not describe behaviour of a software, they describe additional properties that can be used to judge the quality of software. Since describing software quality presumes knowledge about software, bad concepts can often be identified by the lack of non-functional requirements. Maintainability, level of documentation, and ease of integration are examples for non-functional requirements. Non-functional requirements should be measurable. Hence, »*The page should load fast.*« is too inconcrete, »*The page should load in two seconds max during an average performance test.*« is very concrete and measurable. If you want to apply the YAGNI principle, assume moderate non-functional requirements if they are not mentioned in the concept (or if they are mentioned, but inconcrete). If you are writing the non-functional requirements yourself, be realistic: A small corporation with 20-50 page visits a day does not require three days of performance tweaking - as the page should load fast enough because the server is not busy. If the corporation can increase the number of daily visits, a better server or hosting package shouldn't be too expensive.

## ***Last, but not least, remember the 80:20 rule-of-thumb!***

Last, but not least, remember the 80:20 rule-of-thumb! We have to identify the time consuming parts. If a part is absolutely necessary, you have to implement it. The question should be: how will you implement it? Does it have to be the latest framework with a small community? Do you need to switch to the just-released version of a library if the documentation is not up to date? Should you use the new CMS, when not all extensions are available? How much research will be necessary to do so? »*That's the way we have always done it.*« is not an exciting approach, but it'll get the job done without surprises.

It's important to understand that all of this does not mean that you can start writing dirty code with hacks along the way! You're writing a lightweight application, not a messy one! However, *You ain't gonna need it* is a practical approach. If it would cause many lines of code to reduce a few lines of code duplicates, I personally think that you may relate efforts to budget and some unDRYness is ok. It's a small application. Hence, the added maintenance complexity is acceptable. We're in the real-world.

Let's come back to the initial thought: we like building things. When Beethoven wrote the *Diabelli Variations*, it was contract work. I don't think he made compromises on budget. He ran the extra mile, because he did not want to write average music; he wanted to write a perfect composition.

I'm certainly not implying that we're all geniuses, and that our brilliance should shine through every line of code, but I like to think of software architecture as compositions. I'm a passionate developer, because I want to build perfect compositions, and I want to be proud of the things I'm building.

***If you want to be an experienced and business-proofed developer, you have to master the You ain't gonna need it principle. If you want to keep your passion, you have to fight against it every now and then.***

## Summary

Software principles are a way of looking at software. To me, a good principle should be based on a simple concept, but it should evolve to a complex construct of ideas when confronted with other techniques and philosophies. What are your favourite software principles?



Weglot

## WP Translation plugin

Compatible with all Theme & Plugins (incl. WooCommerce). SEO optimized. Join 20,000 users.

Advertisement



Chris Peters


N/A

FEED LIKE FOLLOW FOLLOW

### Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Update me weekly



**Python  
Developers  
Survey 2017  
Results**

- General Python Usage
- Types of Development
- Python 3 Adoption
- Technologies
- Tools and Features
- Developer Profile
- Raw Data
- Key Takeaways

[View results](#)

Advertisement

### Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by  **native**



61 Comments

Nettuts+

 Login ▾ Recommend 4 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS **Banhawi** • 6 years ago

"Keep it Simple Stupid" that how should we all code, simple stupid :D, no need for complexities.

4 ^ | ▾ • Reply • Share ›

**Maria** • 6 years ago

Principle #1 "Don't Repeat Yourself" and #2 You "Ain't Gonna Need it", were pretty easy for me to adapt from the get go. The KISS principle is one I learned over time. It's funny how you go back and look at code you wrote 6 months to 1 year back and can plainly see how you could have simplified it! Good article, we can all use reminders like these

2 ^ | ▾ • Reply • Share ›

**Alan** • 6 years ago

"In the real world, applications that are a 100% DRY are hard, if not impossible, to achieve."

This is a funny statement. It makes it sound like there's something implicit in DRYness that makes it hard to achieve, when in fact, there's almost nothing easier. I've never had any trouble reducing repetition of any kind in real systems, provided we're using a sufficiently high-level language.

And that's the catch, which you omitted. Most "real world" (ha) applications are written at companies that have policies dictating what languages you can use, and these languages are too low-level to allow for fully DRY code. The problem is not DRY, but programming language choice, and hence, company politics.

If your language doesn't have closures and quoting, DRY is going to be impossible to achieve without writing your own runtime to simulate these features. If your language has those but doesn't have syntactic abstraction, DRY is going to be possible but will be so verbose nobody is going to bother (as per your third principle, YAGNI).

That's why XML is so popular with Java/C# programmers: it's the path of least resistance to your own parser. (Except now you've written your own mini-language in XML, which is 90% redundant with every other XML mini-language.)

[see more](#)

1 ^ | ▾ • Reply • Share ›

**Matthieu** • 4 months ago



Nice article! Indeed we need, as developers, take these good principles and apply them as much as we can.

About the DRY principle I would precise that code duplication is not necessarily a violation of the principle. I see a bunch of comment on the article where people mix both (knowledge duplication / code duplication).

I wrote an article about it: <http://web-techno.net/dry-p...>

^ | v • Reply • Share ›



**Kamil** • 5 months ago

I followed DRY didn't know about principle, just while writing code i realised how many places i have to update when i repeating code in a few places.

^ | v • Reply • Share ›



**Hussein Terek** • a year ago

Check out this article which defines the most commonly used design principles which makes you software maintainable and extendable.

<http://programmergate.com/s...>

^ | v • Reply • Share ›



**José Henrique Targino** • a year ago

I liked it. Good text. It is on my favourites.

^ | v • Reply • Share ›



**cm4573r Hacker** • 2 years ago

I like your clearness of mind when explaining things Chris. I've never seen someone relate those principles side by side like that! Congratulations!

^ | v • Reply • Share ›



**Syed Muhammad Kamruzzaman** • 2 years ago

Good article

^ | v • Reply • Share ›



**khan** → Syed Muhammad Kamruzzaman • 2 years ago

what software engineering principle were used before oop.

^ | v • Reply • Share ›



**Alex Aloysius** • 5 years ago

"When you find yourself writing code that is similar or equal to something you've written before, take a moment to think about what you're doing and don't repeat yourself." - I like that and it is very important.

^ | v • Reply • Share ›



**Jack** • 5 years ago

Very nice!



^ | v • Reply • Share ›



**Bhagwant** • 6 years ago

Nice article, Thanks a lot :)

^ | v • Reply • Share ›



**Wel Marquez** • 6 years ago

I like how you referenced Albert Einstein and Beethoven to give emphasis on some points there. Great read.

^ | v • Reply • Share ›



**Bogdan Günther** • 6 years ago

Great Article! I especially had to think about the »ninety percent of everything is crap« quote :-)

^ | v • Reply • Share ›



**Richard** • 6 years ago

I really liked this article. In fact, I would suggest Chris to change the title to "3 Key Software Principles Every Developer Needs to Know". I strongly believe in these principles, however, I have to say sometimes due to time constraints or improper project management, we tend to forget about this. We get too caught up in the "we got to get this done and release this product" mentality.

^ | v • Reply • Share ›



**Vance** • 6 years ago

Hi, I do think this is an excellent site. I stumbled upon it ;) I'm going to return yet again since I book-marked it. Money and freedom is the best way to change, may you be rich and continue to help others.

^ | v • Reply • Share ›



**kanuj bhatnagar** • 6 years ago

I'll completely agree on the YAGNI method. For years I've built CMS sites for clients, complete with an admin section to allow them to edit their pages as they see fit, only to have them email me every other day to change the font, add a one liner, add a new line etc. Yes, it becomes frustrating after a while, to the point you consider why you used a server side language in the first place. To which I think that the only reason we even make CMS sites these days is to make our job easier, and to not open up a FTP and edit HTML pages for a client's each demand. I'd rather use the WYSIWYG editor I put in, than to edit each file by hand.

^ | v • Reply • Share ›



**Decor dakheli** • 6 years ago

thanks for an awesome update and This will help a lot

^ | v • Reply • Share ›



**Sando** • 6 years ago

Great article and it covers the main and most important points of software development.

Often problem is when clients don't know what they want and usually they start throwing new features in already started project with agreed development plan. Hence that's good place to use the YAGNI approach and just fight back your client but explain why this feature shouldn't be developed which could be, mainly, impact over budget or bloating the application which could drive back the users.

And most important keep balance above all of it (Client, Budget, Users)

^ | v • Reply • Share ›



**Alex** • 6 years ago

Thank you! This will help a lot in my upcoming project :)

^ | v • Reply • Share ›



**Daquan Wright** • 6 years ago

If you're running a business, it's only logical that you need good business practices.

Same goes with good programming practices. I like these tips!

^ | v • Reply • Share ›



**Raitis Stengrevics** • 6 years ago

Awesome read! No comments.

^ | v • Reply • Share ›



**steve graham** • 6 years ago

The problem with DRY is that should you have to change that database extraction class that is called by 20 other functions then you have to test each and every function. If however you split the database extraction class into 4 parts then you only have to test 5 functions.

As an example we have separate extraction methods depending on how the data should be sorted rather than passing the sort type into one method.

As you say, DRY can cause complexity.

^ | v • Reply • Share ›



**Angelo** ➔ **steve graham** • 6 years ago

DRY makes sense in some cases, NOT in all cases.

The whole point of DRY is to have less maintenance and less potential for bugs by having less code. When you start over complicating your code to accommodate DRY, that's when it makes more sense to duplicate logic. Less code does not automatically mean simpler and easier to maintain code. In many cases it can make sense to ignore DRY. Code that is intuitive and easy to understand trumps more efficient, lightweight or elegant code in many cases. It all depends on what your priorities are.

Use a philosophy/methodology/design pattern only to the point where it's beneficial. Don't ever stick with something that isn't adding value simply because it's the "right" way to do it or for the sake of being 100% consistent throughout your application.

Those are BS, idealistic justifications that have no place in a successful business. Good is always better than perfect if you want to make money.

1 ^ | v • Reply • Share ›



**Roan Bester** ➔ Angelo • 5 years ago

I agree. I've seen this happen where DRY is applied fanatically to mean 'do not re-type a single line of code', resulting in abstraction upon abstraction to the point where you have no idea what does what. Common sense and simplicity are always better in my mind than strict adherence to a specific philosophy for the sake of achieving DRY nirvana. If you spend more time developing 'frameworks' than actual coding you're setting yourself up for lots of pain.

1 ^ | v • Reply • Share ›



**Maicon Sobczak** • 6 years ago

Massive well explained concepts. I apply the KISS in every project and am learning the DRY.

^ | v • Reply • Share ›



**Corey** • 6 years ago

We should call unDRY code WET code Written Every Time

^ | v • Reply • Share ›



**Mika Andrianarijaona** • 6 years ago

Thanks for this post. It was really helpful and I have just discovered the YAGNI principle. It's 100% true, we generally implement things that are will likely not be used and the client won't even be aware either it was implemented or not. I will think twice now before implementing code ;)

^ | v • Reply • Share ›



**Think360studio** • 6 years ago

Hi chris. I totally agree with you. I also consider that a software should be simple so that it would be easy to use. Being a Designer i always prefer easy software so that any project of mine will consume less time.

^ | v • Reply • Share ›



**Guest** • 6 years ago

unDRY? wet!

^ | v • Reply • Share ›



**creage** • 6 years ago

Took a responsibility of translating this nice article in Russian here <http://habrahabr.ru/post/14....>

^ | v • Reply • Share ›



**Abe Tobing** • 6 years ago

This article is a true-quality !

^ | v • Reply • Share ›

**hansi** • 6 years ago

@Raptor: i thought the same thing. DRY and KISS are mutually exclusive sometimes.

^ | v • Reply • Share ›

**Stephen** • 6 years ago

I really enjoyed reading this article. very well laid out and great tips for newbie like me to the world of programming. :)

^ | v • Reply • Share ›

**Raptor** • 6 years ago

In my experience, DRY is not as much as a problem as people overdoing DRY. I have many times seen code factorisation that leads to more code or more complex code than if it wasn't factored in the first place. There is a balance to strike and when you want to share code between projects, you have to be sure it is worth it, and not making the code less maintainable.

^ | v • Reply • Share ›

**Chris M.** • 6 years ago

Good article for inspiring quality work, at least that's the effect it had me!

I did notice a couple typos under HTTP Section "...DOD, dead on arrival..." I think you mean "DOA." Also I don't believe Steve Jobs ever wrote a line of code in his life, he's might be considered more of an 'honorary programmer.'

^ | v • Reply • Share ›

**Dominick** ➔ Chris M. • 6 years ago

just noticed this too—hasn't been fixed yet :/

but enjoying the post so far; well written!

^ | v • Reply • Share ›

**Chris** ➔ Chris M. • 6 years ago

Yes, it's a typo. Should be DOA.

Steve Jobs is the reason I wrote "... have a very good technical understanding, or even coders themselves." He was definitely the first, but you're right, not a coder. That's why I listed the other Steve of Apple, Wozniak, too. An unbelievable talented programmer,

^ | v • Reply • Share ›

**Abhijit** ➔ Chris • 6 years ago

That goes on to show that good business sense is equally important to bring up those kind of organizations. (I know this is kind of off-topic, just felt like writing!)

^ | v • Reply • Share ›

**TJ** • 6 years ago

Excellent article. Very informative!

^ | v • Reply • Share ›

**Sergey** • 6 years ago

Great article! So glad to see this kind of content on nettuts:)

^ | v • Reply • Share ›

**Garrafote** • 6 years ago

Very nice article,

Thank you!

^ | v • Reply • Share ›

**Alex** • 6 years ago

This is a very inspiring article. I have constantly battled with delivering quality software/going over the top when delivering solutions according to a specific budget. The YAGNI principle helps tremendously.

^ | v • Reply • Share ›

**Avinash** • 6 years ago

Very True and Well written.

The structure and pattern of your application / site / program (whatever) defines most of it. If you write it crappily be ready to rewrite it SOON .

^ | v • Reply • Share ›

**Terry Rubio** • 6 years ago

Good article!thanks a lot for the information!

^ | v • Reply • Share ›

**Cesar** • 6 years ago

Fantastic article. Keep it up!

^ | v • Reply • Share ›

**Fritz Lekschas** • 6 years ago

Great article! Thanks :)

^ | v • Reply • Share ›

**BanoBatho** • 6 years ago

Advertisement

#### QUICK LINKS - Explore popular categories

ENVATO TUTORIALS



JOIN OUR COMMUNITY



HELP



tuts+

26,276

Tutorials

1,152

Courses

26,452

Translations

[Envato.com](#) [Our products](#) [Careers](#) [Sitemap](#)

© 2018 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

Follow Envato Tuts+



