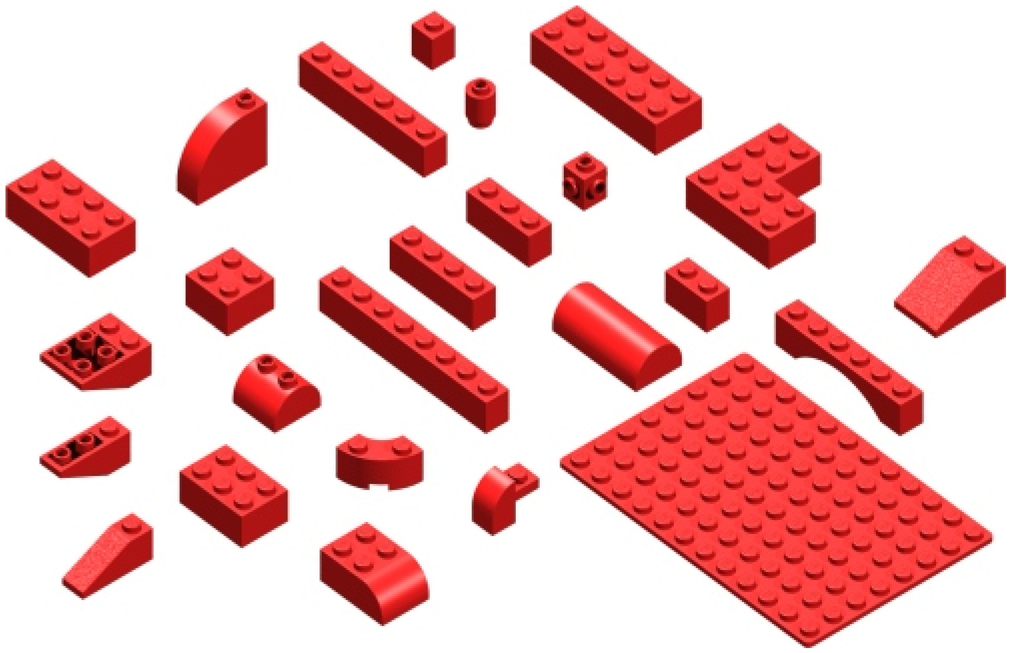
The Lego Process Manifesto



The Software Industry is in a crisis. According to the Standish Group initial CHAOS report[1] only 16 percent of all Software Projects finish on schedule. A staggering 31 percent of all software Projects are completely canceled. An amazing 53 percent of these projects exceed their budgets, each by an average of 189 percent. In addition, research by Fred Brooks[2], has shown several decades ago, that adding more developers to a late Software Projects will only marginally reduce the time it takes to deliver that same project.

Another problem, according to Boehm[3], is that the best developers are up to as much as a thousand times more productive than the worst. The worst also outnumber the best by the same amount. While at the same time actual demand for Software seems to increase exponentially.

---

1. The Standish Group. *The Chaos Report.* http://www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf
2. Fred Brooks. the Mythical Man Month. ISBN; 0-201-83595-9
3. B Boehm. Software Engineering Economics. Prentice Hall PTR, 1981

Couple this with almost a complete lack of re-usability of anything but infrastructure software, libraries and controls, and the scene is set for a massive inability to meet the needs for software in the world. In fact, the entire Software Industry today can be compared to the way the conventional industry was organized in the 18th Century.

The Objective of the Lego Process

The objective of the *Lego Process* is to fix these problems by bringing together existing best practices, design patterns, methodologies and processes, in addition, also adding a couple of its own ideas. The end result and goal being the industrialization of the Software Sector[4]as a whole.

Note that although the Lego Process might be seen as a description for how to implement a framework, it is not tied to a specific framework. The Lego Process is 100% agnostic of all technologies, platforms and languages, but the Lego Process can definitely - and probably should be - implemented in the forms of a Framework that implements the ideas given in this document.

4. http://www.softwarefactories.com/

The Solution

Most everyone in the Software Industry today can recognize that the industry is in a crisis. In fact, many skilled developers have a deep understanding of the problem. Some even know how to partially fix these problems, yet no real (and more importantly; **no complete**) solution has ever been proposed. A lot of great minds have created design patterns and frameworks to reduce these problems, but so far, no proposal exists that can cope with all aspects of these problems.

All of these existing solutions can be thought of as, spikes in a wheel, they're almost right, but without having them all collected together to a complete circle there is no way the wheel is going to roll effortless.

The *Lego Process* includes references to many existing ideas, processes, design patterns and methodologies. In addition to that it also brings in a couple of unique solutions of its own. When all these ideas are working together the reader will see that this solution has formed that complete circle, and the problems that have plagued the software industry will be eliminated.

A productivity increase of at least 2000% can be expected, and is also virtually inevitable within your company by applying these principles and ideas throughout all your Software Projects.

Agile Development

Ever since the *Agile Manifesto*[5], developers have been flocking to Agile Software Development[6] and methodologies. Few have been successful in implementing Agile, though, Agile is the right way to implement Software, and few are able to dispute today. When you start implementing Agile development in your organization you will soon notice that there are several problems you need to solve before you can reap the benefits of being truly Agile. Without solving these problems Agile methodologies are impossible to implement. Those who have successfully been able to implement Agile methodologies are either intuitively or consciously implementing some of the ideas given here.

Customer Driven Development

Customer needs and wants are in the front seat of most popular Agile processes. This is referred to as *Customer Driven Development* in Agile processes. This means that whenever the customer wants to have a new feature or another bug fixed, this is a *direct order* to the developers on the project. This is obviously the right way to create Software. The situation today becomes more that of *The Inmates are Running the Asylum*. Software is not a goal, it is the means to accomplish other goals, the Software Industry is mainly a helper for other industries.

However in most systems today, the system parts are so intertwined that when a developer starts creating the features the customer wants, some other parts of the system breaks down. Many Agile advocates claim this to be a problem of lack of Unit Tests[7]. However, some parts of the system might be very difficult to create Unit Tests for, like for instance in the UI[8] layer of the application. In fact rather than being a problem with lack of Unit Tests, the Lego Process recognizes this problem in the context of that the systems being built today are way too Closely Coupled - or **Monolithic**[9] in their structure.

Or in other words; Just because one follows Agile methodologies and

5. http://agilemanifesto.org/
6. http://en.wikipedia.org/wiki/Agile_software_development
7. http://en.wikipedia.org/wiki/Agile_software_development
8. User Interface. The Graphical parts of the system that interacts with the end user.
9. http://en.wikipedia.org/wiki/Monolithic_system

processes, doesn't mean that one has Agile capability, nor does it mean that one has tools that enables the Agile Process. Without the tools that enable Agile development you might just as well try to carve a new car out of the mountain with a hammer and a chisel. The creation of the Lego Process - or something similar - has also been postponed by the Agile Movement's focus on letting people know that; *"Agile is not about the tools"*. So all the best Agile Evangelists have explicitly encouraged people to run **away** from tools that Enables or Facilitates the Agile Process.

The problem of *Monolithic Applications* can be solved easily by tools that make sure that all parts of the system are completely loosely coupled and are constructed with a *Modular Design*, which doesn't construct dependencies for other parts of the system at all. So instead of making one system, you actually end up creating several sub-systems, or *Modules,* which are completely independent upon each other, and can later be *tied together* to form a complete system. This is a known technique in other industries and is often referred to as *Divide and Conquer*[10], or *Modularization*[11].

In fact the technique of Divide and Conquer has been successfully applied to many other areas through Human History, ranging from the organization of Military Units, to how to distribute First Aid after earth quakes. Divide and Conquer has also been used when building bridges and railroads for Centuries. Many Software Companies and individuals within the sector will dispute here and claim that they are already following this practice. This is simply not true! Unless you're able to reuse 95% of all your code in 95% of all your projects, then you are not using a truly Modular Design. You are also far away from reaping the full potential of *Divide and Conquer* unless you are able to - without changing one line of code - exchange any part of any systems in your portfolio. All while the system is online and **in production**.

Do not feel sad however, you are in good company. I have still not seen one Software Company being able to be neither fully Modular nor being able to follow Divide and Conquer at any significant level. This paper will help you become 100% Modular in your Software Designs, which again enables you to perfectly execute on the Divide and Conquer technique within your organization.

This *Modular Design* makes the system extremely flexible during both development and maintenance. If the customer at any time is not satisfied with any one particular part of his system, then that part can be updated without having any negative side-effects on any other parts of the system.

---

10. http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm
11. http://en.wikipedia.org/wiki/Modular_programming

Or, as an alternative the Module can be completely replaced by another Module. This also creates the additional benefit of ending up with a whole range of Modules which can later be reused on new projects, and consequently, kick-start those projects when initiated. Notice that this a a prerequisite for the Lego Process, unless you can take any existing module and put it into any existing system you have, then your systems are **not** constructed within the Lego Process boundaries. This also facilitates the reuse of large portions of your applications, not only libraries and controls.

This might not seem to be a new idea, and in fact it isn't, Modularization of Software is an old idea, starting out in the late 60s in fact. However, what is new with the Lego Process is that it gives you an exact recipe of how to actually accomplish Modularization by describing the key elements of the processes and mechanisms needed to become Modular. So instead of having a vague idea of mixing something brown together with something white to create something that tastes good, you'll end up with a complete recipe and the groceries needed to bake your own chocolate cake, instantly.

The Global Ownership Problem

Global Ownership[12]of code is a great initiative and many have tried to implement this for beneficial reasons. First of all, it enables - together with *Pair Programming*[13] - that your most skilled developers are able to scale their skills to other less skilled developers on the project. Secondly, it reduces risk, since you're now no longer relying on any one developer to be able to maintain "portion X" of the system. However in a tightly coupled system it can be very difficult to understand the implications of changing a part of the system, unless one has initially implemented it. This is true because a change in "portion X" of the system might have a negative consequences for "portion Y" in the same system. This makes *Global Ownership* - in traditional Software Projects - almost **impossible** to achieve. Yet again the *Modular Design* of the Lego Process will solve this problem.

There are many reasons why the Modular Design solves this problem, but one of the most important is that no Module is dependent upon any other Module in your system. Another important fact is that the Modular Design also implicitly creates very small pieces of code that can be easily understood without having access to the original developer who implemented this Module. This makes Global Ownership of code not only easy, but in fact also to some extent almost impossible to avoid. Global

12. http://www.extremeprogramming.org/rules/collective.html
13. http://www.extremeprogramming.org/rules/pair.html

Ownership is **implicitly given** by the Lego Process. When you can change any parts of your applications without breaking anything, then the size of the context you need to acquire to maintain your applications dramatically shrinks. As a result, Modular Systems will become easier to understand, especially for those who did not originally create them.

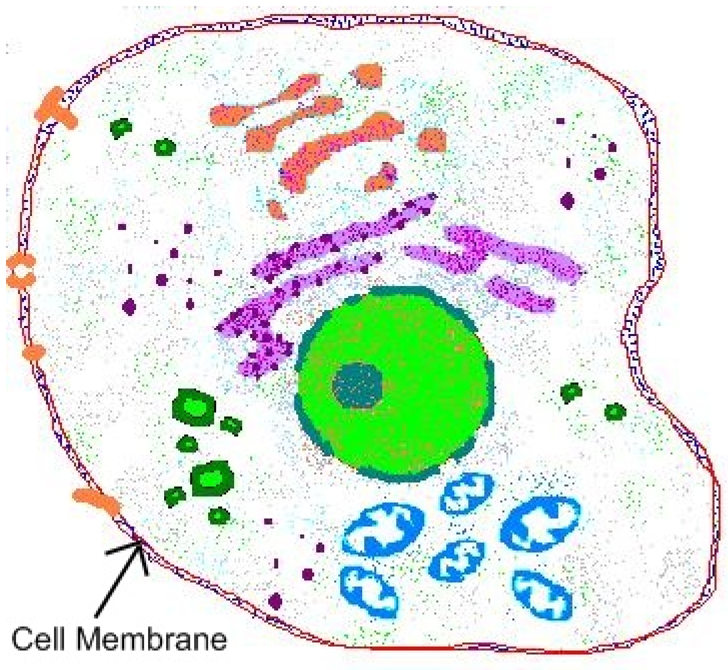
When working on the Alto Machine at Xerox PARC[14] Alan Kay[15]discovered that novices and children could easily create advanced software if the number of lines of code did not exceed ~200. But as the programs became longer they would fail in being able to understand both how to create them, and also how to maintain them. The probable cause was believed to be that the context needed to understand these programs would grow too large for mere mortals. Trained developers have dramatically increased this number, just like professional soccer players can run faster for a longer period of time then you and I. However no human being can put the entire context of an average-sized complete application into his brain at the same time. 100,000 lines of code, and sometimes even as much as 1,000,000 lines of lines of code is equivalent to asking our previously mentioned soccer player to run 40KM per hour around the Galaxy.

When working with a Modular Design you will seldom experience Modules or other pieces of code larger then ~200 lines of code. This means that it will be far easier to train new developers, and also far easier [for anyone] to maintain existing Software. And more often then not you can come away with ~20 smaller Modules each with ~200 lines of code solving the same problems a Monolithic System with more then 1,000,000 lines of code solves. So the total size of the system has gone down from 1,000,000 lines of code to 4,000 lines of code. **A reduction of your code base by 99,6%!**

Alan Kay[16] suggested at OOPSLA in 1997[17] that we should start to build Software Systems more organically, or think of our systems like collections of cells. One important fact about cells that Alan Kay emphasized is that there is no way for one cell to look into the internals of another cell, still they could perfectly collaborate to create extremely advanced structures. The cell was a perfect Black Box[18], purely doing its task without letting anyone look at its internals. This was because of the cell's membrane. He also spoke about how he had noticed that cell based structures was able to scale into trillions of individual cells together creating a whole, like for instance a

14. http://en.wikipedia.org/wiki/PARC_%28company%29
15. http://en.wikipedia.org/wiki/Alan_Kay
16. http://en.wikipedia.org/wiki/Alan_Kay
17. http://video.google.com/videoplay?docid=-2950949730059754521&hl=en
18. http://en.wikipedia.org/wiki/Black_box

human being or a fish etc. He further went on to explain how this is the actual design of the most successful Software Project through History; the ARPANET[19] - which later became the Internet.



The Refactoring Problem

Agile Advocates often say that a prerequisite for being able to successfully *Refactor*[20]code is to have a wide range of Unit Tests. This is true, but the same problem that applies to Unit Tests in regards to Customer Driven Development also applies to Refactoring. It is virtually impossible to have Unit Tests for every part of your application, like for instance the UI[21] layer. This makes it harder to successfully implement *Refactoring*. As it is, time constraints and deadlines, also come into the mix and ends up creating financial problems in regards to creating enough Unit Tests and to do enough Refactoring. The deadlines of your project are often way to constraining to enable a full suite of Unit Tests and the needed Refactoring.

---

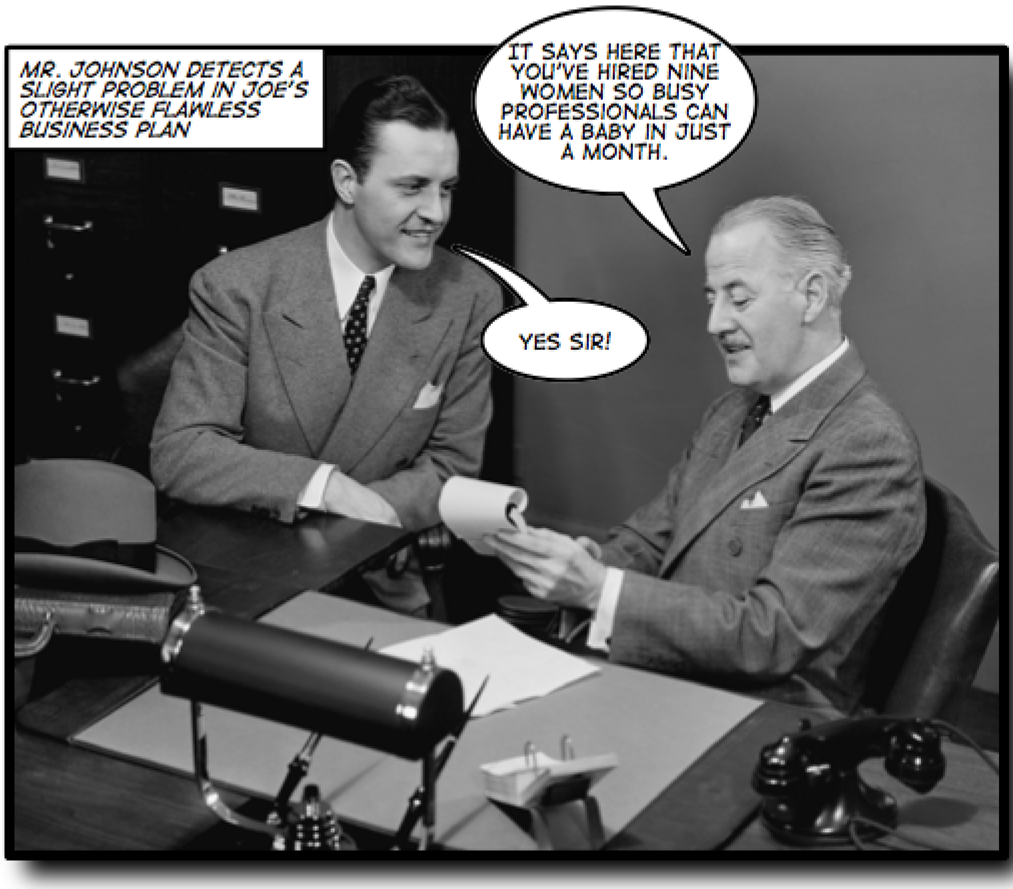
19. http://en.wikipedia.org/wiki/ARPANET
20. http://www.extremeprogramming.org/rules/refactor.html
21. UI means User Interface and is what the user sees. Some solutions exists to simulate users interacting with your system through some sort of macro or scripting techniques, but they are very difficult to implement and they are very difficult to maintain. In addition there will always be a border-case that's not accounted for.

Also if your application is Monolithic, then the smallest "unit" it consists of is itself, and you can not create Unit Tests to verify the correctness of any smaller Module then the entire Application. Imagine if the only way to test the individual parts of a car was to drive the car for a long period of time, instead of the way we can test individual components of the car today like for instance the "pressure the tires can tolerate" and so on.

In a Modular Design though, you are able to reuse Unit Tests the same way you are able to reuse a Module itself. This makes it much less expensive covering all the aspects of your code with an extensive amount of Unit Tests. And in fact, in a Modular Design you're also able to reuse Refactoring, which might sound like - for the experienced developer - reusing drinking water, or some other purely *"one way resource"*. However, in a Modular Design, when you Refactor, you Refactor *building blocks*, which are being used in several different Applications. So even an apparently *"impossible resource to reuse"*, like for instance Refactoring is also easy to reuse - several times - in a Modular Design.

Refactoring, scales with a Modular Design!

The Mythical Man Month


MR. JOHNSON DETECTS A SLIGHT PROBLEM IN JOE'S OTHERWISE FLAWLESS BUSINESS PLAN
IT SAYS HERE THAT YOU'VE HIRED NINE WOMEN SO BUSY PROFESSIONALS CAN HAVE A BABY IN JUST A MONTH.
YES SIR!

How come the Egyptians 5000 years ago could easily organize 20,000 people, working on the same project, meeting their deadlines, and create structures that are among the greatest ever created and are still standing today? Easy; Modularization of tasks and problems enabled Divide and Conquer and made one large problem become several smaller ones. Then each of these problems were small enough to be solved by a handful of people.

> *"When it comes to working as a team, bees and ants are smarter than humans, they have created systems, so easy to understand, that thinking is not a prerequisite for solving their problems."*

Fred Brooks went into the *Mythical Man Month Problem* in details in his book *called; "The Mythical Man Month*" - so an exhaustive explanation is beyond the scope of this paper. But the basic theory goes like this.

Many Software projects, when faced with the fact that they are too late, will try to add more developers to their teams. This almost never has the expected result, and the reason as pointed out in Brooks book has a lot to do with the *added need for communication*. This is what Brooks refers to as *The Mythical Man Month*. This sentence refers to the mythical belief in that, if you add 5 more *Man Months*[22] to your project around the original deadline of the project, then you should experience an increase in the speed by 5 Man Months of development. However, you will seldom experience any improvement at all, and often the exact opposite will happen. When you think about what it takes to get a new person up to speed, the project ends up becoming even more late. So the additional manpower seldom gives any effect at all, and often if there are any effects, these effects might even be negative.

> *"Adding more developers to a late project only makes it later."*
>
> *-- Brooks*

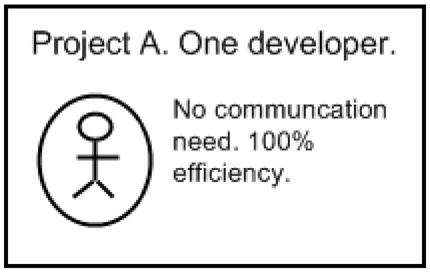
The reasons to this are very easy to understand once explained, and it goes like this; When you have a Monolithic Software project with many developers, then every time any one of these developers need to make a change, they must talk with all the other developers on the project to hear their opinion about this change, and also if the change might have negative side-effects for the other developers on the same project. This translates into; The more developers on the project, the larger the need for communication becomes.

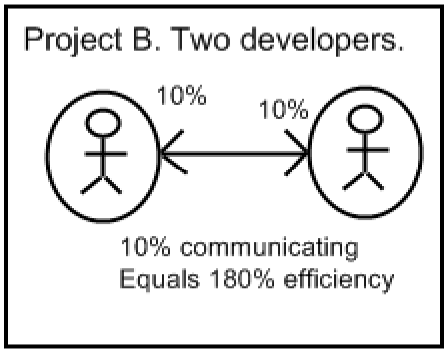
> *"If 1 woman can have a baby in 9 months, how fast can 9 women have a baby then?"*
>
> *-- Brooks*

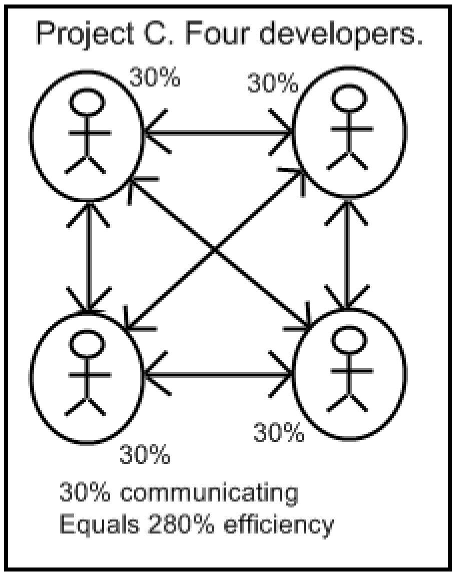
---

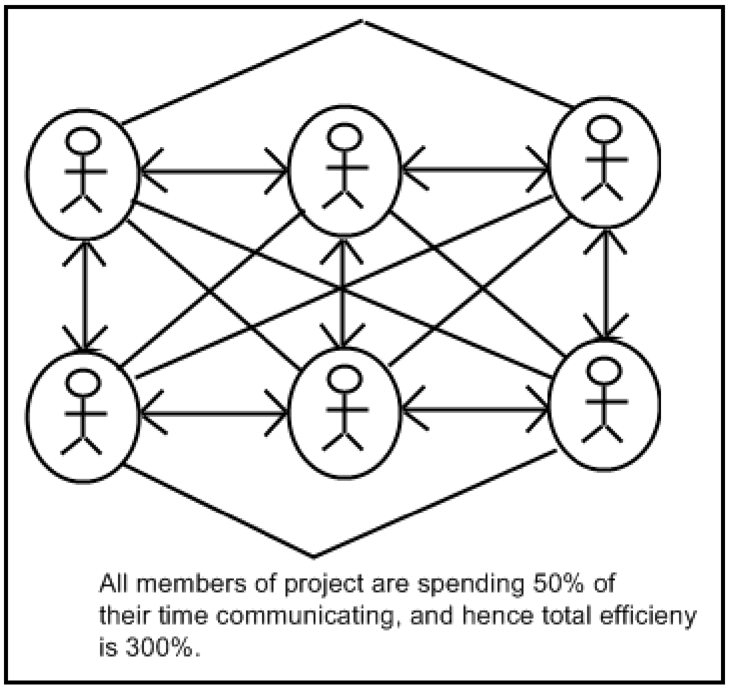
22. Can be 5 men working for one month each, or 1 man working for 5 months. Man Months are originally a phrase conceived by IBM in the 60s.

In the image below we have illustrated four different projects, each showing the need to communicate with the other developers on the project. Notice how developer working alone does not need to communicate at all and can spend 100% of his time purely on developing the system, while the largest project with 6 developers with the added manpower is barely gaining any momentum at all. And in fact, as you add more than 6 developers you will start to **loose** productivity for every new person you add to the team.


Project A. One developer.
No communcation need. 100% efficiency.

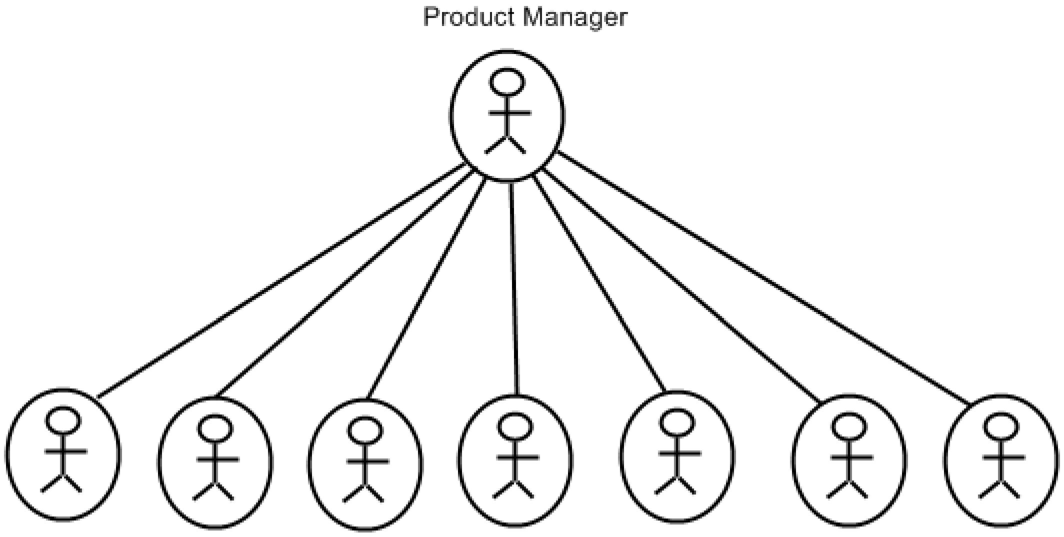





All members of project are spending 50% of their time communicating, and hence total efficieny is 300%.

This has given rise to the [quite justified] belief that when skilled developers read about a Software Project in the news exceeding its budget with millions, and sometime even billions that they could themselves implement the whole thing in "a month".

This problem comes from the fact that the more developers you add to an existing project, the more these people need to communicate about their technical problems, solutions, suggestions and such. Simply because there are more people involved who need to be kept in the loop. However, by Modularizing these problems and by implementing a *Product Manager*[23] - which solely has the responsibility to be the communication link between the customer and the developers - you get a totally different picture.



All 7 developers are now only spending 10% of their time communicating and hence the productivity of the entire team is now at 630%.

Many Agile Advocates will say that they have always had a *Product Manager*, a *Product Owner*, a *Customer* or a similar role that is a perfect substitute of the Product Manager Role above. However when it comes to a *Closely Coupled System* - or a *Monolithic System* - all the developers on the system still need to talk with all the other developers on the same project because that Monolithic Structure makes it impossible to implement changes without knowing the side-effects of what that change will imply for all other parts of the system. Every time a developer wants to create a new feature or fix a bug he needs to verify his ideas about how to fix that bug with - way too often - with **all** the other developers on the project.

Research have shown that every time a developer is being interrupted while debugging he wastes ~40 minutes because of the Context Switch. This

23. A Product Manager is not a Project Manager. The Product Manager has the role of being the communication link between the customer(s) and the developers. While a Project Manager is managing the project as a whole, allocating resources and helps everyone get access to the resources they need to fulfill their jobs.

comes in addition to the time spent responding to who actually interrupted him. The context needed for debugging is huge because of the large amount of data needed to store in the conscious parts of the brain, therefore whenever a programmer is being interrupted while debugging he needs to spend ~40 minutes to come back into the Flow[24], if he is brought entirely out of it. To help another developer with a problem that needs conscious thinking, moves focus away from the debugging the developer was pursuing, and will definitely bring the entire context of the debugging out of the conscious parts of the brain for the developer being interrupted. This means that only a minor question from another developer could loose ~40 minutes of productive time for the developer being asked the question.

This has also added a lot to the perception of developers as asocial beings, and as being "more in love with computers then humans".

Many Agile methodologies recognize the Communication Problem and tries to fix it by putting constraints on how meetings should be attended, how often one should meet or some other *"Meeting Constraints"*. eXtreme Programming for instance uses the concept of *Stand Up Meetings*[25]. Although these ideas will partially solve the problems, it is still fixing the **wrong** problem. Imagine if everyone building the pyramids 5000 years ago had to meet once every day telling their opinions about every new rock carved out of the mountain. Even if they all stood on top of each other - every last one of them - it still wouldn't solve anything. In fact they would probably all instantly suffocate! And the Egyptians would basically experience the same problems we are facing in the Software Industry today.

The only way to truly fix this problem is to eliminate the need for communicating with the other developers on the team through Modularization and Divide and Conquer.

> *"Far too often we as intelligent beings, while sitting in our cars are confronted with a tree in our path. So we stop our car and step outside, grab our axe, and chop down the tree. Then after driving our car about 10 more meters we see another tree and need to repeat the process. Happily living like this we force our cars through the thickest areas of the forest believing that we are great problem solvers, while completely failing to see the highway five meters away from us which would have made the process of getting from A to B so much simpler. The first thing we should do when discovering a problem is to see if it is the* ***right*** *problem. Fixing the* ***wrong*** *problem leads us all into the forest."*

---

24. http://en.wikipedia.org/wiki/Flow_%28psychology%29
25. http://www.extremeprogramming.org/rules/standupmeeting.html

-- Erik Naggum - slightly modified to make a point

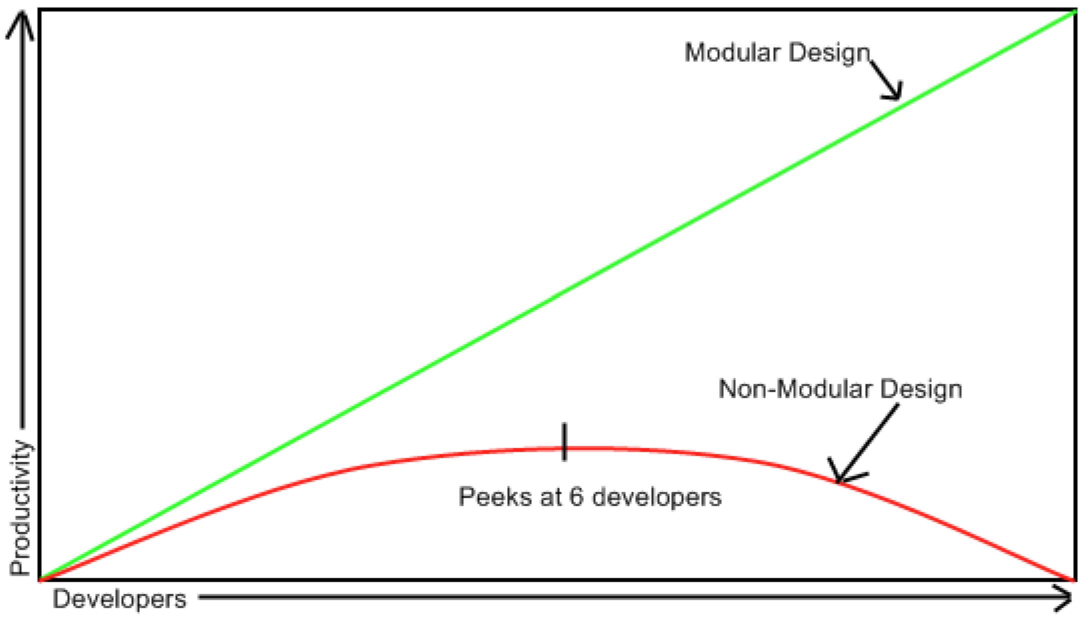
"If Henry Ford in the early 1900s had discovered the wrong problem, instead of making cars, he would have made travel easier by breeding really fast horses!"

Notice though, that having Stand Up Meetings might still be a good idea, it just doesn't solve the communication problem.

In a *Loosely Coupled System*, with a *Modular Design*, any one developer can implement **any change** in any part of the system without introducing negative side-effects for other parts of the system. This almost completely eliminates the need for communication. Or at the very least, the developer implementing the change does not need to talk with any other **developer** about his change, thereby increasing the amount of noise and bring other developers out of the flow by destroying their context. Instead he needs only to talk to the *customer* or in this case the *Product Manager*. This reduces the amount of *Muda* in the process.[26]

26. http://en.wikipedia.org/wiki/Muda_%28Japanese_term%29

The difference in scaling is obvious, as seen below.



So while a Monolithic Design peeks at about 6 developers with 300% efficiency, and after that point does not have higher potentials in regards to productivity at all, and in fact, reaches zero productivity with 11 developers. The Modular Design already at two developers outperforms the same sized Monolithic Design team and are able to scale linearly upwards. And at 6 developers, instead of having 300% productivity they have **540% productivity**. This means that in a Modular Design you can actually add developers to a late project to make it be released earlier. So Brooks *Mythical Man Month Law* **does no longer apply**. At 7 developers the Modular Design Team is more then twice as productive as the Monolithic Design Team. This is because you can in a Modular Design add up any amount of developers, and they will never have to spend more then 10% of their time communicating.

Notice that the most productivity you will ever get out of a Monolithic project is maximum 300%, regardless of how many developers you add to your team. In fact, at 7 the productivity is reduced to 280%. This translates to 2.8 times as productive as a single developer would have been doing the entire project himself. So while the costs are 7 times as high at 7 developers, the productivity only increases by 2.8 times. Thus the absurdity in adding more developers to a late Monolithic Project becomes pretty apparent.

Notice also, that the constant of 10% of communication needed per additional developer is probably quite low. With a higher, more realistic constant, the peek will be reached even earlier. And the maximum productivity and momentum for the project will be lower.

It might be possible for highly skilled project managers to reduce the constant of communication-need increase per developer added. Maybe some really skilled project managers are also able to make this number increase logarithmically, according to how many developers are added to the same project. It is also true that highly skilled developers will reduce the communication-need overhead. But still, this important fact remains; The costs grow **linearly**, while the productivity grows in a **Sinus Curve**.

> *"In a Monolithic Project - as you add more developers - the production costs will grow linearly while the productivity will be a Sinus Curve. In a Modular Project the costs and the productivity will both grow linearly proportionally to the other."*

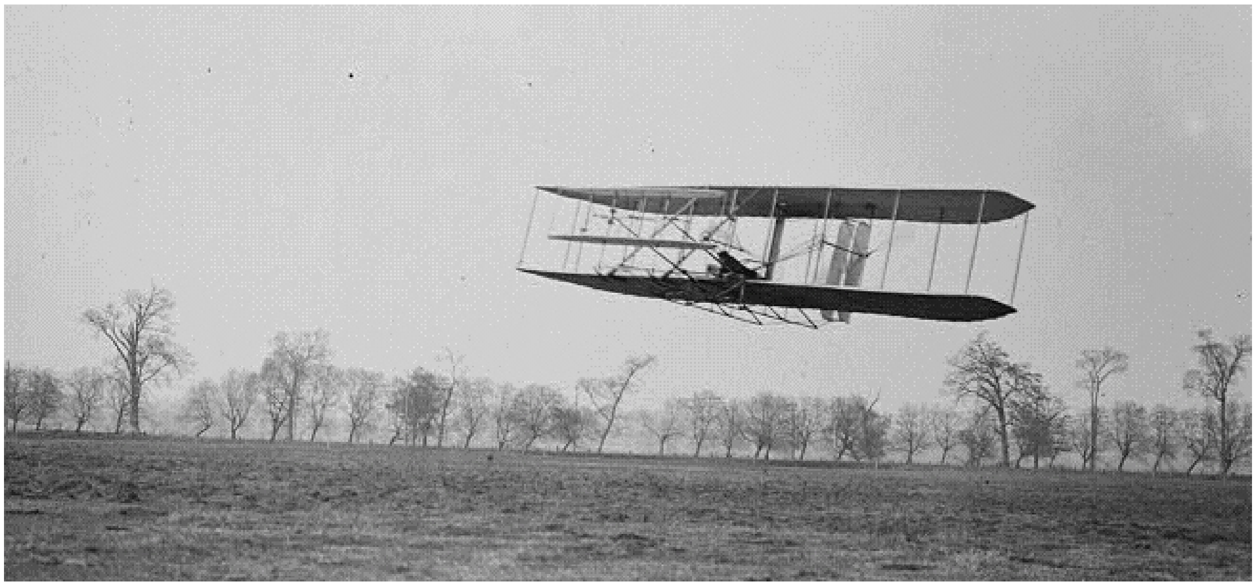
Defying the Laws of Gravity

The Software Law of Gravity; "Adding more developers to a late project only makes it later."

-- Brooks

When the Wright brothers soared into the air in the early 20th Century, it must have seemed for the spectators as if they were defying the very Law of Gravity. And, it will feel the same way for you when you defy the "Law of Gravity" in your Software Projects. However, what the Wright brothers understood, and also the Lego Process realizes is how the laws of Gravity work, and instead of working against them you make them work **for** you.

Instead of standing on the ground jumping up and down, waving your hands trying to fly like a bird, you realize that the pressure upwards can exceed the pressure downwards created by Gravity if you have enough speed forward and some mechanism that makes forward speed transform into upwards pressure. Or in plain English; You have Wings...



...it doesn't sound that difficult when you understand what actually happens...

Resource Management

In conventional, Monolithic Projects, there is also the problem of *Resource Management*. Seldom will you have many developers in your staff that you are able to move around to different projects. This is because of the Monolithic Design and the complexity it dictates on your systems. All your projects will be too difficult to understand - for most of your developers - to be able to make even small changes within. Not to mention become productive team members. This is because of, among other things, Monolithic System are *Close Coupled*, where one needs to understand the **whole** system before being able to modify any significant parts of the system.

"Looks delicious on a plate, but not in your code!"

This concept is known as *Big Ball of Mud*[27] or *Spaghetti Code* and most developers will claim that they don't create this kind of code, however most developers are wrong. All Monolithic Systems with a slightly higher degree of complexity than a naive *Hello World*[28]implementation will experience Spaghetti Code - or the Big Ball of Mud. This is because it is impossible to create systems which are not *Big Balls of Mud* unless you have intuitively [or consciously] already adopted the ideas of the Lego Process. This is true because *Big Ball of Mud* is a synonym to the word Monolithic.

> *"A synonym for the Big Ball of Mud is Monolithic, however few understand this before it is written like it is here. A Monolithic Design is the opposite of a Modular Design. Hence all Monolithic Projects are Big Balls of Mud, and all Modular Designs are not!"*

---

27. http://en.wikipedia.org/wiki/Big_ball_of_mud
28. http://en.wikipedia.org/wiki/Hello_world_program

With several Modular Design Projects running at the same time you can easily move resources around. Normally in a Monolithic Design it takes months for an average developer to understand the system he is supposed to start working on. A Modular Design can be understood as the code is being read, and there is no need for more than a couple of minutes before a new developer is productive. This is because of, in a Modular System every Module will be very easy to understand, even for a novice. First of all, because the code will not be coupled with other parts of the system, and secondly, because the code will be highly less complex and smaller in size.

> *"Debugging is twice as hard as writing the code correct in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*
>
> *-- Brian Kernighan*

Most people read the above statement and are satisfied if their system can be debugged, but how about the complexity needed to maintain their systems? Maintenance is probably at least twice as hard as debugging the system. How about giving the entire code-base to another team for them to maintain, that is probably twice as hard as maintaining it yourself. This grows by the square every step upwards in the hierarchy.

> *"To maintain a system written by others is 8 times harder then writing the code in the first place."*

If your systems has less then 12.5% of the complexity needed to write the code, then this means that you can easily move resources and developers around as they are needed, instead of having too many developers working on one project and too few on another project. To achieve this will add extreme demands on your tools and processes. The Modular Design however, will make such a low degree of complexity, not only easy, but in fact implicitly happen as a natural process.

Reuse of resources - Software Factories

Reuse of Software is somewhat of the *Holy Grail* today, and while we are quite able to reuse large portions of our projects today through infrastructure software, libraries and components, we are still far away from being able to reuse *Application Building Blocks*. In a true Modularized system you can reuse any Module in all your projects. This means that every time you start a new Software Project, most of the Domain Problems[29] will already be implemented in prefabricated Modules.

How a typical Software Project begins

Typically a developer collects features through e.g. User Stories or something similar from his Product Manager or customer. A typical list for the first iteration can look something like this;

- Login Control
- User object associated with Roles
- Splash Screen

Now the developer has the information needed for starting out, he then begins to implement these features. The first thing he starts with might be creating the *User Entity Type*. Then he starts consuming this User Entity Type in a Login Form of his application. Finally in his iteration he will create the Splash Screen.

In a Modularized Design, all of these tasks will already be finished. Most systems need a Login Module, most systems also need a Role Based implementation, and many systems will need a Splash Screen. This means that the project team, or company, will likely have existing components for the exact same tasks from previous projects. This means that all the tasks from the above iteration would already be finished, and you could skip the entire iteration. In a typical Software Project where you have say 20 iterations, you can probably expect to be able to start the entire project out somewhere between iteration 15 and 19. Just how long depends on how long you have applied the Lego Process within your company and what existing Modules you have already created or acquired through some other mechanism.

This is because of the re-usability that a Modular Design gives you. So instead of only reusing Infrastructure Software[30], Libraries and Controls. You

29. http://en.wikipedia.org/wiki/Problem_domain

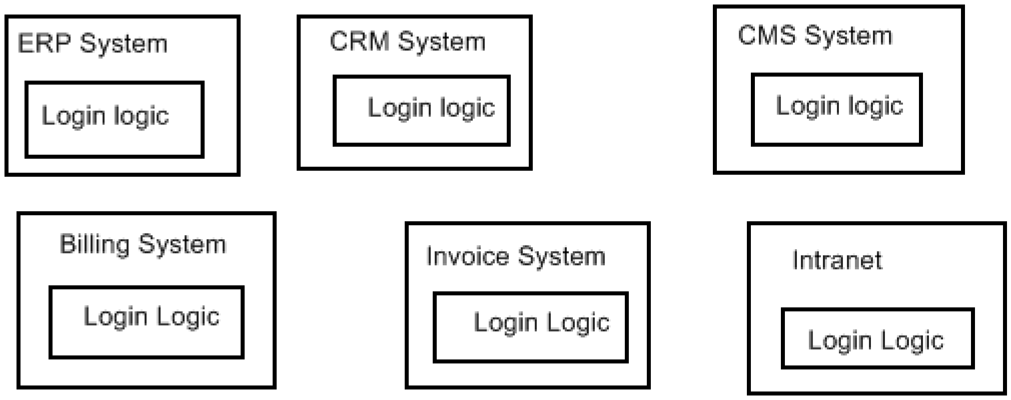
can actually **reuse large portions of your entire applications**. This also has the additional benefit of that whenever you're refactoring or adding features to an existing Module you will implicitly also refactor or add features to other systems, if you wish. This gives a productivity increase completely unmatched by anything else within the Software Industry. A productivity increase of at least 95 percent, or in fact also above 99% can be expected after having acquired a suite of pre-built Modules.

So the same Login Module might be in use in many different systems, and as a result the cost of developing and maintaining that Login Module will be spread out over several different projects. This will linearly reduce the costs of developing and maintaining a Module according to how many projects are using that same Module. This also scales outside of your organization.

---

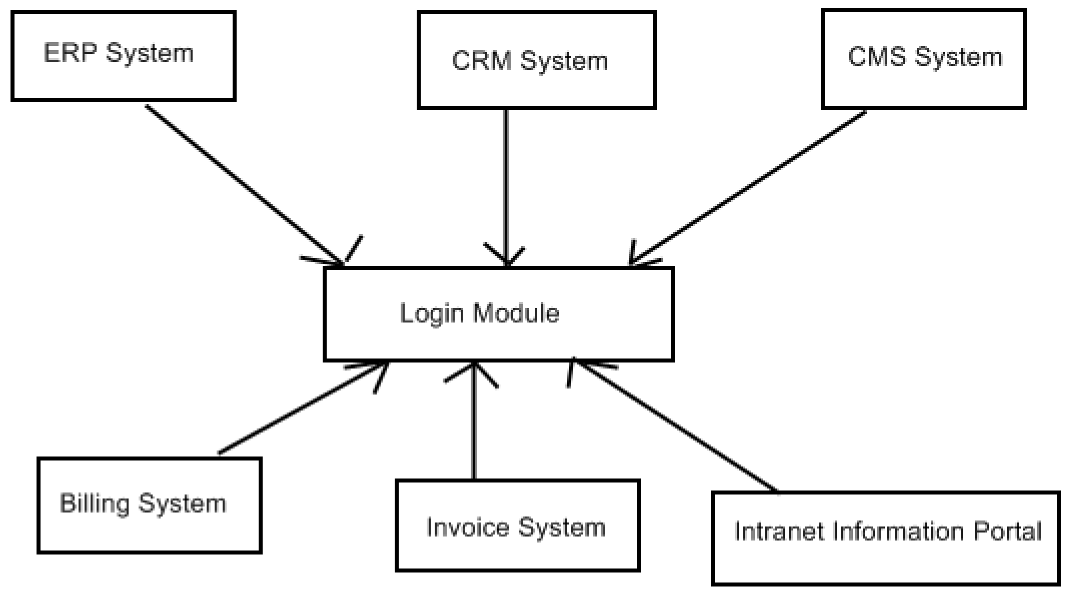
30. Refers to things like the Operating System, Databases, Run-times and such.

Below are six typical systems created with the Monolithic (traditional) Design;



As you can see here the developer(s) have duplicated logic for the Login Control in every application.

The same systems created with a Modular Design;



Here the developer(s) are using the ***same*** *Login Control across multiple applications.*

Of course when displayed like this, the madness in conventional Monolithic system development seems obvious, but the first image is actually a 100% accurate description of how virtually all systems are being developed today. And in fact, you would probably have a lot more duplicated/similar code in your Monolithic Systems than only the Login Control. A qualified guess would be around 50% commonalities, but these will grow beyond 50% when getting some experience with the Lego Process, probably as high as 95% for most systems. Some examples are given here;

- Roles Management
- Customer Components
- Contacts Components
- Address Components
- Email Handling
- etc, etc, etc

The more one practices the Lego Process, the more one is able to find commonalities in seemingly unrelated problems. Not to mention that all the things that should be commodities[31] in your applications could be implemented in the Application Pool, and also reused in all applications, some samples here are;

- Security
- Scalability
- Infrastructure Software Connection points and abstractions
- Email handling
- etc, etc, etc

> *"A Modular Design makes it easy to reuse large portions of your applications. And not only libraries and tools."*

According to several international studies, at least 80% of the costs of a system is in maintaining the system, and not the initial development. However when you have a Modular Design, then whenever you are maintaining one of your systems, you are actually maintaining **all** your systems. Unless you explicitly chooses not to maintain more then one Application by *Forking*[32] whichever Module you are currently working on.

This can also be scaled outside of your company between several different companies sharing their Modules with each other, and entirely split the cost on maybe as much as thousands of projects and millions of end users.

---

31. http://en.wikipedia.org/wiki/Commodity
32. http://en.wikipedia.org/wiki/Fork_%28software_development%29

The "Barry" Problem

Many Software Companies have their "Barry". Barry is a person which is insanely skilled in System Development. Often "Barry" is several orders of magnitudes more productive then the average developer on his team. And he is considered a *Magician* in regards to Software Development. He represents that 1 out of a 1000 best developers on the planet and everyone looks to him when they are stuck. You could probably put 200 randomly chosen developers together without getting the production capacity of "Barry".

This is a fact seen over and over again in the industry. For instance when Linus Torvalds[33] started out creating GIT[34]there had been hundreds of developers working on Revision Control Systems[35], ever since the late 1960s. Still, Linus alone could outperform all the work done so far in the Problem Domain in less then 10 months.

The Lego Process makes your developers productivity perform to the level of "Barry". Before you implement the Lego Process it would take maybe as much as several hundred developers to outperform "Barry" on both quality and productivity. And in fact, most of the times it wouldn't be possible at all to outperform "Barry", regardless of how many developers you put into a project. This fact is thoroughly explained by many in the Software Industry, Paul Graham[36] being one of them. After you have implemented the Lego Process all your developers will have risen up several orders of magnitudes, both in regards to productivity and quality. The difference between "Barry" and "John Doe Average Developer" will be no more then 1 in 5, while previously it was 1 in 1000. As it is, "Barry" has also improved in his productivity.

In fact to a large extent the Lego Process recognizes what these remarkably skilled developers [Barry] are subconsciously doing, writes it down, and implements a process and tools to allow the rest of the developers to be able to consciously follow these same set of principles.

"Barry" might, within the Lego Process have specialized tasks, like creating that "really advanced Module" communicating with that really hard-to-get system that only he understands. This will mean that "Barry" is still a "special resource" for your team, but he will not be completely impossible to replace. Or with other words; The company will not be entirely dependent

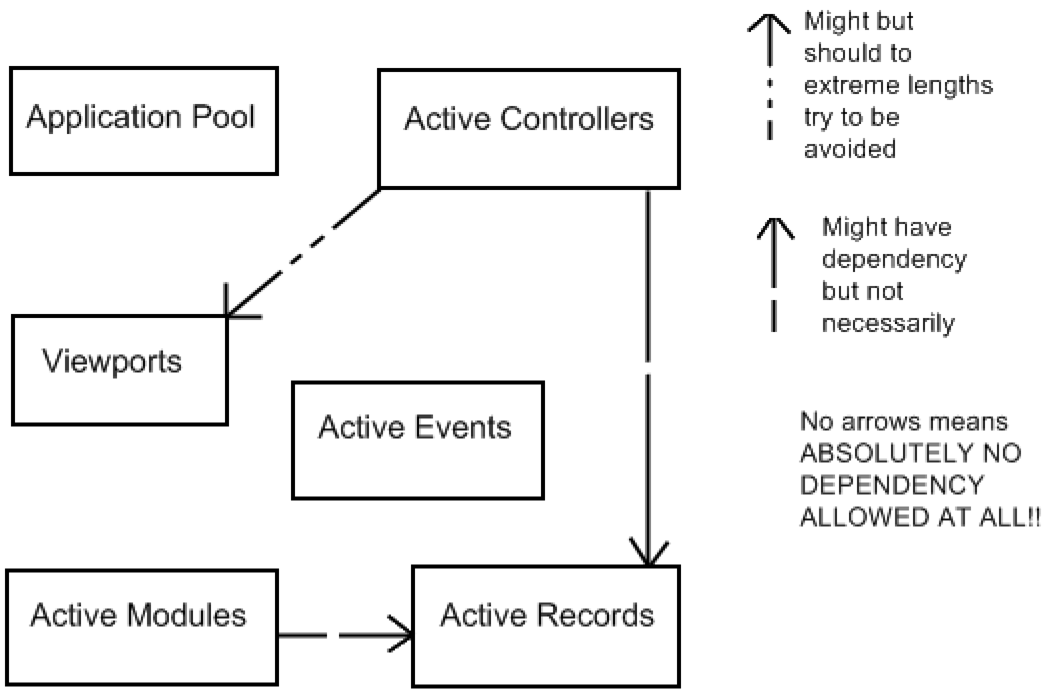
33. http://en.wikipedia.org/wiki/Linus_Torvalds
34. http://en.wikipedia.org/wiki/Git_%28software%29
35. http://en.wikipedia.org/wiki/Revision_control
36. http://paulgraham.com

upon Barry's un-willingness to have vacations.

The difference in productivity between your worst and your best developer has changed from 1 in 1000 to 1 in 5. Leveling out the differences in regards to productivity and quality between your employees.

The Entire Process in Details

Probably the most important thing to understand in the Lego Process is the way Components are constrained from having dependencies upon each other. **The almost complete lack of dependencies in the Lego Process is its absolutely most important feature**. Below is a chart showing the dependencies of Component Types in Lego. You might want to print out this specific image and hang it over your computer on your desk, at least until you have gotten used to how to implement Lego Components.


Application Pool
Active Controllers
Viewports
Active Events
Active Modules
Active Records
Might but should to extreme lengths try to be avoided
Might have dependency but not necessarily
No arrows means ABSOLUTELY NO DEPENDENCY ALLOWED AT ALL!!

If there are no lines between Components in Lego, then it is **absolutely not allowed** to exist any dependencies between those components at all! If you take shortcuts here, you're really not doing anything but building old style [Monolithic] applications!

In fact if you manage to limit yourself to the above diagram in regards to dependencies, you will experience, that you too can easily become almost as productive and "skilled" as Linus Torvalds.

Active Modules will likely be dependent upon your Active Records. This is because Active Modules are the ones who display the data that consists within your Active Records. Active Controllers will be responsible for loading Modules into your Viewports. This might create a dependency upon the Viewports for your Active Controllers. The dependency your Active

Controllers have upon your Viewports should however at all costs be as small as possible, and if possible completely eliminated. If this dependency is too big then you can not exchange your viewport with another viewport, and consequently you loose a lot of your Modular Design.

Some people might ask why the Controllers are not dependent upon your Modules, especially considering that the Controllers are the ones actually loading these Modules. This is simply because, even though the Controllers are responsible for loading your modules, it is a requirement that they do this *virtually*. This means that you must be able to somehow easily configure which Modules your Controllers are physically loading. This can be done by using *keys* when loading Modules from Controllers, and then make it possible to map these keys to point to any Module you wish. Think of these keys like pointers, where whatever they point to can be changed.

So when some Controller requests the *ViewCustomer Module* to be loaded, the actual Module that will be loaded can be overridden - and so is not known to the Controller. But the same Module must be loaded every time a Controller requests a Module with the same key to be loaded.

Components

All the building blocks in the Application Pool are referred to by the common name of *Components*, and there are five different types of Components. These are;

- Active Viewports
- Active Records
- Active Modules
- Active Controllers
- Active Events

The Application Pool

The Application Pool is where your entire application lives. It can be seen as a "pool" where all the Components swim around like fish, completely independently of the other Components, without making troubles for other inhabitants of the pool. Sharks[37] must be avoided in this pool. Sharks will only exist if individual components are created in such a way that they are dependent upon each other. Whenever one *Component* starts creating dependencies upon other Components, for almost any reason, this is

37. Components with dependencies upon each other.

referred to as a *"Shark"*. Note though that you can still compose one or more Modules together to create new Modules that are using other Modules through Composition[38]. These can be thought of as a group of fish that together in the group forms a whole. But Composition is the **exception** to the rule and should **not** be misused. If your application consists of one, two or three large components composed out of several Modules each, then you have an unhealthy eco-system!

> *"No Component, should at any time, know about the internals of any other component! This is a rule* ***without*** *exceptions! This is the foundation of the Lego Process!"*

To have a healthy pool you must at any time be able to put any new Component into the pool, or remove or replace any existing Component in the pool without negatively affecting the eco-system as a whole. This is possible even for Components that are using Composition of other components to form themselves, but only as long as no components are aware about the internals of other components.

If the pool becomes too small because of too many visitors [users] or too many fish [components], you must be able to scale the system easily by making new pools next to existing ones. This translates to the scalability of your system. The system must be able to *Scale Out*[39] perfectly with zero changes to the internals of you pool.

Active Modules

As previously mentioned, the Lego Process is a Modular Process where all the visible Components of the Software System are created as Modules. These Modules are 100% loosely coupled, and will together form the Application. A prerequisite for the Lego Process is that these Modules are completely interchangeable, and does not create any dependencies upon each other. Another prerequisite is that these Modules are built 100% self-contained, bringing no dependencies upon other Components or the Application itself. But the most important fact, as stated previously, is that no component knows anything about the internals of any other component in the Application Pool. It is also imperative that the Application developer can easily override these Modules and easily exchange one Module for another, either by configuration or by physically removing the Component

38. http://en.wikipedia.org/wiki/Object_composition
39. "Scale Out" refers to the concept of adding more servers to an existing park, as opposed to "Scale Up" which means to replace existing (slow) servers with newer (and faster) servers.

that implements the Module and replace it with another Component that implements another, but logically interchangeable Module.

Active Viewports

A *Viewport* is a *Container* for your Modules, these Viewports can be thought of as the place where your currently visible Modules can be seen. The Lego Process lends heavily from the *Model View Controller Pattern*[40] as described by Trygve Reenskaug in the late 1970s. While the Modules can be thought of as the Views and the *Active Records* are the Model and the *Active Controllers* are the Controller, the *Viewport* is almost unique to the Lego Process. It is imperative that you are able to load multiple *Modules* into your entire *Application Surface*[41] at the same time, the Viewports accomplishes this in the Lego Process. This is what makes your application become composed out of *Building Blocks* and *Orchestrated together* instead of created as a *Big Ball of Mud* - or a Monolithic System.

A nice way to think about the Active Viewports is that they are like clothes. Just like we humans can change clothes, your application should be able to change clothes without requiring a liver transplant, which translates into no changes to any of the Components inside of your Application Pool.

It is also important that your Active Viewports also are Active Modules, and can be threated as such. This means that your Active Viewports might contains Modules which themselves are Active Viewports.

Active Events

All the Components of your Application Pool must be able to communicate with each other, without having dependencies between them. A Component might for instance be interested in subscribing to an event which will be triggered when a customer is deleted in any other Component. While at the same time it is imperative that if any one of those two endpoints does not exist in your system, no part of your Application will break or stop working. This means that seemingly unrelated Components might be installed into the *Application Pool,* and they will instantly start communicating with the other Components in the Application Pool - and vice versa. Although they have no

40. http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
41. Means the entire visible UI parts of your application. At any time the Application Surface must be able to have multiple Modules shown at the same time.

actual knowledge of the existence of each other. You must also be able to remove Modules **out** of the *Application Pool* without having any side-effects, or breaking other parts of the system itself.

At first sight the Active Events might seem to be a Broadcasting Design, it really isn't, but sometimes it helps to picture them as such. Active Event Handlers must be able to define which Active Events they would like to subscribe to. This to reduce the noise in the system. Think of the types of Active Events as a Hashtable[42] containing a list of Event Handlers interested in knowing about when that particular Event occurs. This way any Component can register interest in a new or an existing type of Event, and any Component can raise any type of Event which will notify all those interested in knowing about that type of Event - which might be a list with zero entries.

Another way to look at it is as a distribution central for magazines, anyone can create subscriptions to any magazines. Then whenever new magazines are published, all subscribers of that magazine will get their copy. If someone creates a subscription to a magazine that doesn't exist in the central's portfolio, nothing will happen, but if that magazine later becomes a part of the portfolio of the central, these subscribers will get their magazine when new magazines of that type are published. Here the magazine subscription is Active Event Handlers, and the publishing of new magazines are the equivalent of raising new Active Events. So it's a loosely coupling between functions [Event Handlers] and those calling those functions [Raising Events].

This is the key-stone behind being able to create large and complex *Application Pools*, which are able to interact with each other, but at the same time does not create dependencies between inhabitants [Components] of the pool.

> *"The Application Pool is the village, the Components are the people in the village, and the Active Events their language used for communicating with each other."*

Note that this can be further expanded upon, and made into a "communication protocol" which could be as rich as the English Language, but for simplicity - at least within single systems - such advanced implementations are probably not required. Cells communicate with chemical signals, humans need much more "complex" language to communicate. Components here are cells, while systems are humans. The higher up in the hierarchy you move, the more rich the communication protocol (Active Event

42. http://en.wikipedia.org/wiki/Hash_table

Implementation) must be. But if you implement the Active Event pattern to cross application boundaries, you'd probably have to extend the protocol to something resembling linguistic rules, having grammar, and become much richer then what's described above.

Active Record

The *Active Record Design Pattern*[43]is thoroughly defined by Martin Fowler and others in the computing History. There is one significant difference in regards to how the Lego Process implements the Active Record design pattern. It is an absolutely **must** that the Active Record implementation used in combination with the Lego Process has to be completely abstracted away from the physical model of the database. This means that the *"One class per table"* mantra from the original Active Record design pattern, does not apply in the Lego Active Record Design Pattern. This is because you need to be able to install new and un-install old Modules and all their related *Active Records* in your application without having to go through complex installation processes and such. In fact dropping a new Component into the Application Pool is all you should have to do. If you don't have this capability, somehow with your Active Record implementation, then you loose many - in fact most - of the benefits by applying the Lego Process. This means that the concept of a unique addressable *database table* will within the Lego Process probably be rendered completely obsolete. There are other reasons to this also, but that's the most important reason.

Notice that this has a dramatic effect on Modularizing your Applications compared to the classic Active Record Design Pattern as suggested by Martin Fowler, simply because your Domain Model is actually Loosely Coupled from your Data Storage.

NOTE!
Even though the Application Pool must be completely agnostic in regards to which Components are installed, some Active Modules and probably also some Active Controllers must be installed side by side with their associated Active Record Components.

> *"The only rule without exception is that there are no rules without exception."*

About O/RM and Active Record

43. http://en.wikipedia.org/wiki/Active_record_pattern

The Lego Active Record Design Pattern is actually not what the Software industry today refers to as an O/RM tool. An O/RM tool might, although, be useful to implement the Lego Active Record Design Pattern. An O/RM tool's most important responsibility is to map Objects to Relational Database Tables, and while the Active Records might be stored in tables within a Database, it is certainly not a requirement. In addition the Lego Active Record design pattern is 100% Loosely Coupled from the physical appearance of tables and such within a database. This is a requirement of the Lego Active Record Design Pattern. In fact you could probably easily implement the Active Record without a Database at all. And, you could also probably implement the Active Record Design Pattern by using only one table. But, the most important reason to why the Lego Active Record Design Pattern is not an O/RM tool, is because instead of expecting a mapping between tables and types, it expects the data-storage to be 100% loosely coupled and not be dependent upon the physical implementation or appearance of your database at all.

A typical implementation of the Lego AR Pattern might be able to put an infinite number of AR types into no more then 10 different physical tables - as the means to abstract away the data-storage.

Active Controllers

These are the objects or types within your system that controls the *flow* of the system. The Active Controllers are the Components whom are responsible for *"mapping"* Modules together and also to a large extent are responsible for signalizing to the Active Modules, Active Records and such Active Events that occurs in the Application Pool. The Active Controllers are also the ones responsible for loading your Modules up into specific Active Viewports. They can be thought of as the Lego Process counterparts to Controllers in the MVC pattern, although the Lego Active Controllers are significantly extending the concept of the Controller from the MVC Pattern.

It is imperative that the application developer is easily able to install and un-install Active Controllers in the Application pool. Either by configuration or by physically removing the Component that implements the Active Controller.

Notice that the Active Controllers are dependent upon knowing about the existence of some of your Active Records Components, but it is imperative that none of your components in your Application Pool are dependent upon your Active Controllers in any ways. The Active Controllers should easily be interchangeable with any other Active Controllers in your Application Pool. It should be possible to remove any Active Controller without having any other parts of your system break down. The Active Controllers might also seem to be dependent upon your Active Modules, since the Active Controllers are the ones actually loading your Modules and injecting them into your Active Viewports, but this is not true. The Modules being loaded by the Active Controllers **must** be possible to override through configuration or some other mechanism.

Flow of Active Events

This is an important concept to understand since the Active Events are the parts that ties your Applications together. One Module might for instance indicate that a Customer deletion is being requested. Then who can possibly handle that event is of high importance. When an Active Event is raised, these Components must be able to handle the event;

- Instance Module Event Handlers (modules loaded into the Viewport somehow) since they might need to update their UI as a consequence
- Static Module Event Handlers (for modules not loaded into the Viewport) since you may want to implement code which logically belongs to a Module when some Active Event is raised.
- Instance Controller Event Handlers (default behavior for handling events in Active Controllers)
- Static Active Record Event Handlers (for logic that typically belongs to Active Record types)
- Instance Viewport Event Handlers
- Instance Application Pool Event Handlers

As you can see, it is not a requirement to be able to handle Active Events in instance event handlers of your Active Record objects. This is mostly because of, that to register all Active Record Components created, fetched or otherwise in existence somehow as Active Event Handlers will probably be too expensive in regards to resources, and make it difficult to scale your Applications. But, if it can be done in your Lego Process tools somehow it might be a bonus. Also since Viewports, Controllers and Application Pools are per definition only existing as *"Singletons"* [kind of] there is hardly any use for trapping Active Events in static event handlers.

MVC and the Lego Process

As you can see the Lego Process can, to some extent, be seen as a *Super-set* of the MVC pattern. But, it also adds a whole range of new, unique ideas which again brings in a whole new range of flexibility. One example of such uniqueness is the concept of a Module being a *"Plugin"*[44] and the Active Events doing the communication between the extremely loosely coupled Active Modules, Active Records and Active Controllers. Another example, is the additional constraint of having the Active Record design pattern being the Model, in addition to the Viewports being the Containers of your Modules. Also, some parts of the Lego Process has been strongly influenced by the works of the people behind the *Composite UI Application Block Framework*[45] from the Pattern and Practices group at Microsoft. Also many ideas have been inspired by the works of the team behind the *Ruby on Rails*[46] Framework.

Agile Methodologies

The Lego Process lives in symbiosis with Agile as a methodology or process. In fact, none of them can exist without the other, at least not successfully. It is a prerequisite for being Agile to implement the Lego Process, and it is also a prerequisite for being able to implement the Lego Process that one is Agile. Which Agile Process one chooses to implement is of less importance, and this is up to the individual implementer of the Lego Process to choose for themselves. The Lego Process is *Agile Neutral*. However, you need at least one Agile development process to [successfully] implement it. Some Agile Processes are mentioned here for reference purposes.

- eXtreme Programming (XP)
- Scrum
- Crystal Clear
- Adaptive Software Development
- Feature Driven Development
- DSDM

There is one place where the Lego Process to some extent disagrees with the Agile Manifesto[47], or at least puts higher emphasizing on a point less

44. Plugin means a component being able to "plug into" an existing system.
45. http://msdn.microsoft.com/en-us/library/aa480450.aspx
46. http://rubyonrails.org/
47. http://agilemanifesto.org/

important in the Agile Manifesto, which is in regards to tools. The Lego Process - although not tied to any particular framework, tool-chain or implementation - still emphasizes that it is imperative to have a tool that enables the Lego Process to be implemented. Often this will come in the forms of a tool, several libraries or a *Lego Enabling Framework* if you wish. Still the goal is to achieve loosely coupling of Modules and not merely to have a tool. If you just *"got a tool that implements Lego"* you will probably be increasing the quality of your software and productivity, but it is still imperative that you understand how to use that tool. Which sums up to that the tools are less important then an understanding and knowledge of how to either use existing Lego Process Enabling tools, or to create (good) Lego Process Enabling tools of your own.

Another prerequisite of the Lego Process is that the tools you choose to use are really simple and easy to understand and doesn't end up piling features on top of features to accommodate any instant need or request from its users. Children below the age of two need instant gratification to become motivated beings, we are grown ups and can handle an extra day or two while the Lego Process owners figure out the best way to implement a feature, or hopefully, in fact an alternative for implementing it at all. The Lego Process is a **brave** process, and the tool-creators creating frameworks that implements the Lego Process must also be brave. Some like to define this as *"Opinionated Software"*, the Lego Process fosters *"Opinionated Design"*. Most importantly, any implementation of the Lego Process must be as simple as possible.

> *"Any feature not described in this document is baggage, and not a part of the Lego Process! The goal of Software Design is to have a system that by the push of a large green button - filling your entire screen - solves all the problems in the world, fills your fridge with groceries, cleans your clothes and makes you more appealing to humans of the opposite sex."*

Since it is really impossible to implement any Agile process without also implementing the Lego Process, this can probably be thought of as an addition to the Agile Manifesto. Or a prerequisite for being able to be Agile. If the Agile Process was completely tool agnostic, it wouldn't be a System Development Process since there are probably few places you need so many tools as in System Development. [Computer, Compiler, IDE, Runtime, Operating System, Data Storage Back-End, etc, etc, etc]

Also, remember that while the Lego Process eliminates the **need** for communication between developers, this does not mean that one should stop the sharing of information and knowledge between developers. If you implement eXtreme Programming for instance, then by all means, implement Pair Programming and Stand Up Meetings every morning also. This will give you - just like in conventional Monolithic development - a boost, and make sure knowledge is shared and distributed among your developers. This makes everyone on your team more productive and happy.

> *"If your team is not orders of magnitudes happier going to work after implementing the Lego Process, then you have failed! The Lego Process is about eliminating obstacles, removing uncertainty and making sure everyone on your team are more proud of their work. Because of this, an increase in happiness and general well-being is inevitable. If happiness doesn't occur automatically, you've done something wrong. Go back to basics and start all over again."*

The Lego Process also has many similarities with the *Lean Manufacturing*[48] which is to a large extent responsible for the entire production capacity increase in conventional factories during the 20th Century. *Lean* is often quoted together with the Agile Software processes.

48. http://en.wikipedia.org/wiki/Lean_manufacturing

One for all, all for one!