When agile fails, a hotfix is not enough

"Proposing the MAD model"

by

Spyridon Antakis
spyros@loonydesk.com

September 11, 2013

Abstract

It is incredibly confusing how sometimes companies try to apply *Agile Development* and how they terribly fail to do so. There is no comprehension of what they define as their *"own agile development"* and certainly the chosen structure is at best unstable. This paper analyses the situation with a pure engineering insight, leaving out of the picture all the *micro* and *macro management* processes. Finally, it challenges the fixed model that exists in most of the companies, which sets *Agile development as the one and only panacea for everybody to follow"*. In that attempt, it proposes the *Manager, Architect and Developers (MAD)* model, purely inspired from different aspects of real life examples.

1 Introduction

When in June 2001, a group of developers came up with a lightweight development method, they were just trying to solve their own problems in the area of software development. They defined some very basic and important *principles* that should be followed during development and they published the so called:

Manifesto for Agile Software Development [1] [2]

Individuals & Interactions
over
Processes & Tools

Working Software
over
Comprehensive Documentation

Customer Collaboration
over
Contract Negotiation

Responding to Change
over
Following a Plan

The *manifesto* states that while there is value in the items appear on the **bottom** of the above list, the items that appear on the **top** are considered to be more valuable. There is no doubt that the *agile principles* amplify a more flexible culture around software lifecycle and *prioritize business reasons* against *time consuming* processes. But a quote from the past, wisely recalls:

"In theory, there is no difference between theory and practice. In practice there is."

Unfortunately, the *agile model* a decade after its first appearance, became another heavy industry. *Expensive commercial tools*, *strict unnecessary processes*, *companies offering consultancy* to a model that **paradoxically** enough, was targeting to simplify things. Victims of a system that is hard to change, in a period that software engineering *explodes*. Unfortunately, billion dollar companies are left with no time to reconsider their approaches from inside out and adjust a productive *bottom-up* development model based on their needs. Creating extremely frustrating environments that they are desperately trying to **be agile**, like there is no tomorrow if they choose **not to be**.

2 Agile as a bottleneck

Unavoidably, every company needs to have a process in place for a successful software development lifecycle. It seems that the majority generally acknowledges that *Agile Development* could be very efficient. However, only the minority seems to acknowledge that their environment **might not** be suitable for applying the *agile process*. Environments are not all the same and when you start treating them in such a way, then you eventually hit **a bottleneck of inefficiency**.

Different teams inside the same environment behave and function in their own unique way and that's a fact. Teams are formed by individuals with *diverse experiences*, *habits*, *opinions* and *skills* and ideally all together should come in harmony. Only then, you could talk about real *productivity* and *efficiency*. Imagine now when on top of an already very complex task (e.g. build team's chemistry), you try to apply a model called *agile*, **the wrong way**. Disaster!

The question that logically arises now is what does **the wrong way** mean in every occasion. Obviously, there is no definite answer to that question and it simply depends. There are so many *unique factors* for each occasion, that would be at least naive to generalize the answer. Nevertheless, there are *common mistakes* made while applying agile, which as a result lead to the misusage of a model that was only meant to *help* and *simplify*. It is at this point that someone should reconsider the truth behind the *"quote"* about theory and practice, as it becomes priceless:

1. Unnecessary tools & Processes

As the manifesto states, agile is supposed to favour *personal interaction* against *tools* and *processes*. Most of the companies invest in very expensive tools that are heavily used daily. A very time consuming process that has no real benefits (only capturing context), and it extremely slows down development. Serious side-effects of the usage of such tools: *days or even weeks* of continuous planning for reviewing the captured context, *several hours spend* for keeping context up-to-date, *multiple tools* for the same purpose and *so on ...*

2. Lack of architecture

Avoiding to document in thousand of pages every single detail of the product's architecture is desired. It completely aligns with the agile manifesto, which it instead proposes working software. However, the manifesto does not state: **neglect documentation**! A properly structured document for capturing clearly the product design and all the architectural requirements is a fundamental ingredient if you want to build any **complex system** that is *scalable*, *extendable*, *testable*, *easy-to-understand* and offers maximum levels of *quality*. Otherwise, you are just looking at **pieces of code** that were **forced** to co-operate successfully.

3. Defined roles & Hierarchies

Although agile sets the principles of *customer collaboration* and *responding to changes*, wisely avoids to explicitly define the roles and the hierarchies, which will be required to achieve these two goals. Someone could argue that the process has already a defined structure for roles and hierarchies, such as small *scrum teams*, *scrums-masters* and *tech-leads*, but that is at best inaccurate. These are just suggestions for achieving certain goals and if they are proved that they are indeed working on your environment, then you probably should adopt them. If not, you need to define your **own** successful model of *roles* and *hierarchies*, which will allow you to use agile as a concept, rather as a strict process that you have to blindly follow. Using the suggested roles, hierarchies, tools and processes for being agile, **does not** make you agile. You need to respect the agile manifesto and find your way into it.

4. The illusion of control

There is the perception that applying the agile model gives better control over other software development processes and makes things easier to manage. That is only partially true, as it assumes that agile process is *not broken* or *misused*. Therefore, most of the time it gives **the illusion** of a fine grained control, which only adds an incredible overhead on the development lifecycle, with no meaningful value. Certainly you get all those nice to have, but **no product life threatening** control metrics. At this stage someone should really wonder, if the *Fibonacci numbers for sizing*, the *burn down charts*, the *special templates for the definition of done*, the *time consuming planning sessions* and having *8 different states* for a defect, are the mechanisms for **controlling better** the development. I doubt!

5. Decision making & Consequences

Agile manifesto does not mandate any specific

approach for *decision making*. Nevertheless, the *responding to changes* principle, requires someone to continuously determine the priorities and the absolute necessities for the product. If someone is desperately changing the decisions based on the customer's input, third party dependencies and commitments for new feature enhancements, it is likely to fail. Allowing a certain level of flexibility is desired, but it should be always inside the feasible boundaries of the team's capacity and what it can be delivered without accepting any significant trade-offs that they risk to harm the final product. Therefore, someone should have the authority to simply say **no to changes** when needed, while weighting all the potential consequences. A product *delivered on-time*, *fully working*, *feature complete* (initial commitments is the metric) is much better, than a product *overloaded*, *delayed*, with a higher risk for lower quality.

3 Non-agile real life examples

Agile is overrated most of the time, if not all of the time. It is treated **as the magician** that will transform a software development lifecycle, make things go faster and organize everything, while at the same time is expected to be the reason for delivering a great final product. Anyone that believes that, has already failed. It's like believing that a guitar will make you a musician, an exploit a hacker, a cooking book a chef, a treadmill a runner and so on ... It definitely **takes more than that**, and at the end you might discover that you do not need a guitar in order to be a musician or an exploit to be a hacker or a cooking book to be a chef or a treadmill to be a runner.

Facts are very simple. If someone can **respect** and **successfully adopt** the *agile manifesto principles* into any software development process, then the process is consider **by definition** *agile*. The ways that you achieve the agile objectives should not really matter and in fact they should be different across companies or even teams, if you want to talk about *an original* and *healthy* software development lifecycle.

It does not take much effort to look around you and realize that there are other systems that *serve similar purposes*, *target similar goals*, *face similar challenges*, but at the end of the day have **nothing to do** with software development. Let's take as an example **any team sport**. In all professional teams there is an absolute hierarchy of roles, including: *team owner (s)*, *managers*, *players*, *scouters*, *assistant managers*, *trainers*, *assistant-trainers*, *general staff*, *marketing people*, etc... By analysing their environments you could easily see a **very close resemblance**, that reflects into *roles* and *hierarchies* up to *goals*, *processes*, *customers* and *profits*.

There are other examples with certainly more structured models, such as *military* or *hospitals* units. Although these models have extremely strict policies due to the critical nature of the services that they provide, they also hide exceptional processes for *decision making* and *task prioritization*. Obviously, these environments are not directly comparable with a software development lifecycle, but they could certainly be considered as additional *food for thought* for improving models that are related to software development.

Going back to the **team sport** model analysis, by diving into the details we could easily highlight some quite important differences. The philosophy that underlies professional sports clubs (*for example a basketball team*) focuses on similar objectives, but the logic and the established authorities are distinct. *Communication*, *responsibilities*, *decision making*, *success*, *failure*, *consequences*, *feedback*, *training*, *tactics*, *recruitment*, *profit*, *marketing*, *budget*, *facilities*, *equipment*, *roles*, *hierarchies*, *evaluation*, and *competition* are **mirroring back** to software development, almost identically. However, there is *no agile* and these systems can still *deliver* the expected results. Interesting!

1. A single person for final decisions

Behind every successful team, there is almost always a great *coach*. He is the center of *all decision making*, *the single point of authority* and contradicting with him, simply **does not exist**. In return, he has to build a *team of winners*, *report back to the owners*, *inspire*, *communicate*, *reconsider*, *enforce* and at the end of the day *make the players believe*. He must become the *psychologist* for his team and it's owners. There is **no easy way out**. The coach will have to possess the *highest of technical depth* and *the type of background* that will allow him to easily gain *the respect of everybody* within the team, in a few days. Then, *trust* and *team chemistry* will follow, resulting into relationships with the *strongest baseline* and with *no unnecessary doubts*.

In return, he is the **first to blame** if something goes wrong and and the one that has to **tolerate** and **defend** all the judgemental press during his leadership. *Extreme pressure* and *responsibilities*, with only desire to be acknowledged as the main contributor for the team's success. Certainly not

a position that everyone is capable to handle and probably **the most critical role** in the entire team structure. A coach should have the ability to keep relationships at professional levels as appropriate and evaluate situations accordingly. Obviously, he does not have **a comfort zone**, he is *accountable* for the team's performance, he is the guardian of a *fragile balance* and it is usually an **all** or **nothing** deal.

2. Players are in control

Players can *win* or *lose* a game. They are the ones that *make the calls* inside the field and therefore, they are definitely considered to be **the ones in control**. *Players* are identified as the second most valuable asset of the product and together with the *fans*, which are indisputably the first asset, they create the sport. In an ideal system, someone could easily argue that these **two assets** are sufficient to complete a *profitable* system that targets sports entertainment. Interesting enough, the players form their *own* informal structure (e.g. team's captain), which allows them to function better under a light hierarchy model. The *right* or *charismatic* coach must be able to successfully integrate with that internal hierarchy and earn its respect, as he drives the entire team into success. **Anything less** would simply be an *average effort*!

3. Fans have the power

Without the fans, *professional* players, *profitable* sports or *team owners* would not exist. Sports entertainment is just another **heavy industry**, that depends on satisfied sports fans. The main priority is the fans and observing sometimes their **unified power** is simply amazing. That kind of power is the one that can *drive industry changes*, demand the *re-construction* of an entire team or conclude that a team was successful even if it did not achieve all the yearly objectives. Therefore, it constitutes a *fundamental requirement* to be able to **respect**, **understand**, **use**, but **not be afraid** of that power. Every *coach* and his *players* should move into that direction **every single day**, anything else constitutes an *overhead*.

4. Evaluation is a continuous process

Every *highly competitive* environment requires **strict** evaluation processes in place. In particular, sports teams treat evaluation as a *very critical* aspect of their system. Players are judged based on their *performance* and their *overall behaviour*. Coaches are criticized for their *tactics* or *roster decisions* and team owners are expected to spend the *maximum budget* for the team. Every single game is like a new evaluation test for everybody to pass. Repeated failures to rise against the expectations, will eventually result to *substitution*. There is *no paperwork* and all the assessment happens **real-time**, based on the *deliverables*.

5. Distinct duties based on skills

Successful teams are build on top of *extremely diverse* players. These players were hired in order to fulfil the requirements for a specific position inside the team, and that's what they know to do best. Their role must be **strictly defined**, because their performance completely depends on that factor. It is unlikely to see a *defender* to be placed on an *attacker's* position or a *play-maker* to have to cover the position of the *center*. That would be at best *inefficient* and it should **never happen**. At least not in any *serious* professional team that has a *reasonable* coach. As a result, team's roster changes quite often according to needs and expectations for the new season. However, it is only under very extreme occasions that the entire *core-team* is replaced. That usually happens **only** if the team **failed terribly** to achieve the planned objectives. In that case, the whole team is anyway *under serious re-construction* and everything is *redefined*. Even then, defining the new roles among members of the team should be based strictly on the players *qualifications* and *experience*. You **do not try** to create players. You **know** what you need and you get the *experienced players* to perform for the team.

6. Wins come game by game

Whoever believes that the reason behind great teams **lies only** under *infinite budget*, *modern facilities* and an *enormous population of fans* has been probably misled to a wrong direction. These are significant factors, but constitute only the *"fuel for the engine"*. What makes sports teams a very interesting example is that they treat each game as a stand alone challenge, with ultimate goals: *to win*, *improve*, *learn* from their mistakes and *understand* their opponents. That is their **manifesto principles**.

They will *not rush* or *force* situations, they will *mature* **game by game** and they will *connect* to each other via practice. That allows *quality*, *discipline* and *communication* to come as natural result of hard work. The coach is responsible to drive the team into that path. If that doesn't happen, most likely the team will end up like a

boat without a captain, waiting to sink at the first bad weather. The efforts must be *synchronized*, *well-defined*, *locked-down*, targeting only *short-term commitments* within the team's capabilities.

4 The MAD model

The main purpose behind the **Manager, Architect, Developers (MAD)** proposal is not to create a *new* software development methodology or *replace* methodologies that already exist and are used for decades. The goal is to suggest a **very strict core-baseline** around *hierarchies*, *communication*, *responsibilities*, *team structuring* and *evaluation*. All these fundamental ingredients tend to be *neglected*, *over-complicated* or *simply disappear* when companies develop software under pressure. It is strongly believed that the lack of a *clear policy* that defines the *minimal procedures* around **any chosen methodology**, constitutes the *Achilles' heel* of these environments.

Inspired from systems such as the one of *professional sports clubs*, the paper brings some different ideas under consideration. It presents a certainly **more aggressive** model with respect to the *desired team structure* and it tries to leave out all the unnecessary *time consuming* luxuries. However, at the same time it argues how that *aggressiveness* could **benefit** the overall development process by *minimizing* the overheads, *simplifying* the decision making and *granting authority* and *control* to the roles required.

4.1 The Hierarchy

Each team is formed by **only one** *Manager* and **only one** *Architect*. They have the **same level** of authority and they are the only ones that can make the **final calls** in the part that they own. Both of them are reporting to the same *Director*, who acts as the *product owner* and overall *supervisor* for the entire progress. In particular, the Manager owns the *non-technical*, *business oriented* side of the product, together with the *career development* for all the involved *Developers*. The Architect *owns the solution* and the developed *features* and he has **full control** of the entire development team. At the core of the team is a mix of *Senior* and *Junior Developers* that their combined skills are able to ensure *on-time* deliverables (see Figure 1).

4.1.1 The Director

He interacts directly with the *Manager* and the *Architect* of the team and he is responsible for their career paths. He has the ability to **overwrite** decisions that were taken at lower levels, but at his **own risk**. In practice, he is continuously aware of the *overall* development status and has visibility **only** to very important obstacles that arise. Everything else remains quite **transparent** at his level. He can be seen as the equivalent of the *sports team* owner.

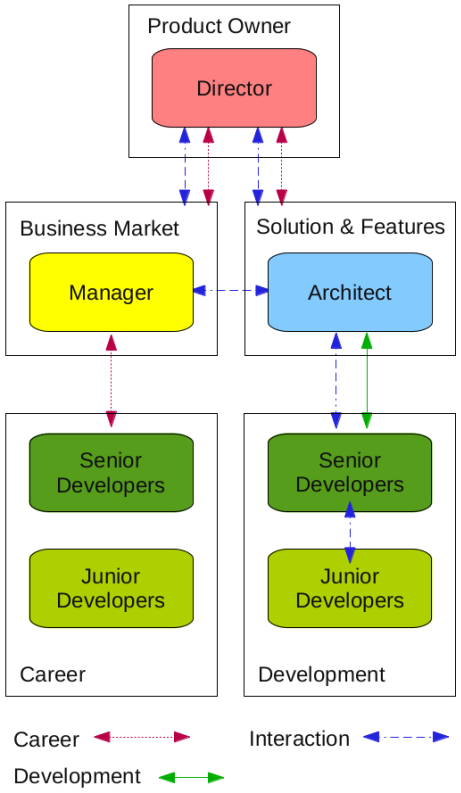

Figure 1: The hierarchy

4.1.2 The Manager

He is accountable for ensuring *revenue growth*, establishing valuable *customer* and *third-party* collaborations, *driving* developers career path, *marketing*, *branding* and *competitors analysis*. In practice, the manager is the person that leads the business aspects of the product. He interacts directly with the Architect for synchronizing the *commitments* of the team, *release dates* and other *mandates* and he reports back to the Director.

Nevertheless, he **never interferes** directly with the *development process*, any *technical decisions* or even the chosen *software development methodology*. These decisions belong to the Architect and the

Developers. In that way, all the responsibility for the technical side of the product is delegated into the overall development team, which now has **all the freedom** for increasing *productivity*, *efficiency* and *quality*, any way it believes that works best, **with no restrictions**!

4.1.3 The Architect

He is the equivalent of the *coach* in a sports team. He is the owner of all *technical decisions* related to the product and has the *authority* to **refuse any changes** that could *downgrade the quality* of the product or *change* the initial planned schedule for deliverables. Indisputably, the Architect constitutes *the mastermind* of the product, the *key figure* for decision making and he is *challenged to prove* that he can achieve all the desirable objectives. He has an *extremely technical* background with *many years* of software development in his career and he is closely related to the product area that he works.

The *MAD model* considers that the *one to one* mapping with the role of a coach is incredible. Therefore, the Architect is granted **full-control** over his team. He has the ability to *choose*, *replace* or *move around* Developers under his **own terms**. In return, he is **fully accountable** for the *performance of the team* and the *product features* delivered. The Architect communicates directly with Director and the Manager for all the *higher level* decisions. At the same time, he interacts daily with all the Developers of the team, while he *takes seriously* and *discusses* all the technical advices that are coming from his Developers.

4.1.4 The Developers

Highly skilled and talented individuals, that can communicate well and tackle together all kind of technical challenges that arise during software development. The majority of the team must be formed by *senior engineers* with a very small pool of *junior engineers*. All of them are willing to *work hard*, *perform* and *learn* from each other. It is very crucial that each member of the team **is motivated** to be part of the targeted project and **enjoys** working with all the used technologies. Otherwise, the team creates *weak links*, which are hard to balance.

Developers represent the equivalent of the *players* in a sports team. They are *in control* as **they** build the software and each of them has *expertises* around a specific area. Their knowledge **is mandatory** to match *as close as possible* the tasks that they are assigned. In the *MAD model*, Senior Developers are **required** to mentor continuously *at least one* junior member of the team. They become responsible for his *performance* and his *deliverables*, as part of their *senior duties*. Developers follow a hierarchy based on seniority and **only** via that hierarchy they are able to *propagate requests* to the Architect, Manager or Director, *when* and *if* needed.

4.2 Communication levels

Usually, communication levels are a **mix of confusion** and they tend to be *mistreated* inside a software development process. The majority of the companies are victims of an *irrational phenomenon*, on which different teams **struggling** to *synchronize actions* and *dependencies* for achieving common goals, but with *no real effect*. The problem appears so often that is almost impossible to fix, unless the communication hierarchies are *redefined* from scratch.

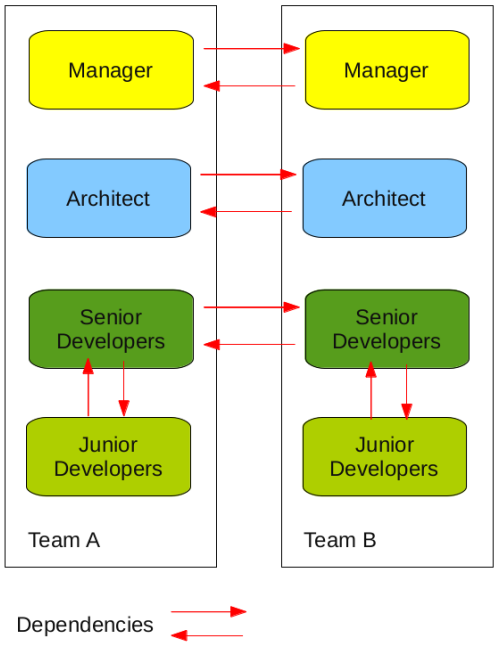

Figure 2: Communication levels

The delegation of responsibility to people with **no real authority**, while there is the expectation from them to define a common direction among teams is at best **pointless**. If everyone can submit a requests against *Team B* and anyone can receive a request from *Team A* at **all levels**, without *any restrictions*, then a *chaotic model* defines communication. Although chaotic models have there *own amazing order*, and that's why

actually companies sometimes succeed to deliver a product, that order is **not optimal** for software development. In these situations, there is *no co-ordination*, *no accountability*, changes are *hard to track*, architecture remains a *virtual concept* and dependencies across teams become a *daily nightmare*.

Therefore, by having as *pre-requisite* the MAD hierarchy, the MAD model suggests a system for communication that is based on *formal ranking*, similar to the military system. The roles that people have inside the company they now start having **real power** and **meaning**. They actually define to which people are *able to talk* directly from another team. The communication happens progressively in a *bi-directional trend* from *lower* to *upper* layers and vice versa. As long as there is no agreement at any level, the information is passed to the different rankings, until it reaches the *highest* ranking (e.g. Director), who decides the final actions. In combination with the proposed *MAD hierarchy*, the communication levels are now structured and clear (see figure 2).

4.3 Building teams

MAD contradicts *the perception* that having a scrum team is always the equivalent of having an *efficient* or *productive* team. The reason is that scrum teams are usually lacking of the **absolute authority** for *decision making* and although their size supposed to be small, in practice that is only a *virtual concept*. When a team **is not autonomous**, *shares* responsibilities and complex dependencies with other teams and it **does not have** a dedicated *Architect* and *Manager*, then the *team* considers to be abstract. *Accountability* is lost in the process, *synchronization* becomes difficult to manage and *cross-scrum communication* introduces **bottlenecks**.

In an attempt to minimize the obvious overheads, the *MAD model* proposes the **autonomous teams**. The fundamental difference is that in order to form a *new* autonomous team, a number of basic requirements must be first satisfied. These requirements will help to *improve* accountability, *control* over dependencies and *communication* across teams.

4.3.1 Basic requirements

Each created team must have a very small size, **no more** than *7 people*, including the *Architect* and the *Manager*. The deliverables of the team are *distinct* and as *independent* as possible from the ones of any other team. Even if that independence is not always possible, the overall architecture of the product must have as a goal to design *self-contained* and *re-usable* components, with minimum *hard links*. Only then, these components could be distributed across different teams for being implemented, without any strong bindings. Potentially, these teams could be seen as some kind of *feature teams*, **but** with *complete ownership* of the feature, *exclusive authority* over the feature and the absolute *responsibility*.

4.3.2 Recruiting process

The **most important** process for building a team is the *recruiting process*. The criteria on which you choose the members of a team is one of the *hardest*, but also the most *crucial* tasks. The selection must be done **wisely**.

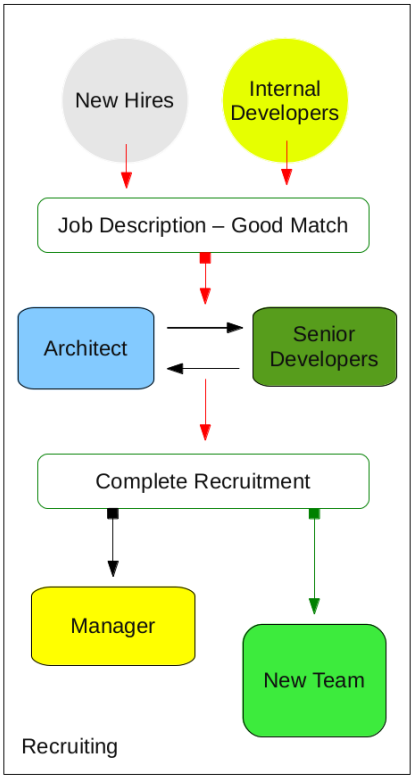

Figure 3: Building a new team

In the MAD model the responsibility for recruiting is very clear (see figure 3). The Architect is the one that will have to *choose* the right members for the team and build the team the way he *desires*. In this effort, he could probably get the advice of Senior Developers, but at the end he has always the final call. The Manager will get recruiting requests from the Architect. If it is a new hire, the Manager will have to *approve* the budget.

Otherwise, if it is an internal moving the Manager will have to arrange to move the Developer into his new team, when that is possible.

The main difference against the usual process of recruiting compared to the one described here, is the fact that *authorities* and *responsibilities* now have a *dedicated owner*. The **impact** for an Architect if the team members are not sufficiently qualified for the position will be *enormous*. It becomes obvious that it is of the Architect's *main interest* to find the right people for the new team, as he will be *fully accountable* for the end results. The same way that a *coach* is for any sports team. Therefore, by delegating the **full control** of the recruitment process into the Architect, it is strongly believed that the *recruiting standards* will be **raised** accordingly.

5 Conclusions

This paper discussed the *agile manifesto* and how companies usually misinterpret the core ideas behind it. It argued that the agile practices should not be considered as the one and only way to develop software and it highlighted some of the reasons. It described how other systems that do not develop software function and it proposed a *core-baseline* around that concept, called the *Manager Architect Developers (MAD)* model.

Although, the suggested MAD model **does not** constitute a new methodology, someone could easily *follow*, *adopt* and *build* on top of that minimal model. The *simplistic structure* of hierarchies and the *delegation of responsibilities* constitute a strong foundation for *efficiency* and *productivity*. In conclusion, the intention of this paper was to *initiate* a different way of thinking around the software development process and *present* an alternative, inspired mainly from other environments.

References

[1] Manifesto for Agile Software Development
`http://agilemanifesto.org/`

[2] Principles behind the Agile Manifesto
`http://agilemanifesto.org/principles.html`