Where Egos dare

Recently and I have had reason to look through some old software engineering textbooks, the kind of thing I used to read as an undergraduate and junior programmer. This is reminded me of a few concepts I haven't thought about in several years. One of these is egoless programming. As I recall my university lecturers where very keen on this concept, it seemed to be *a good thing*. Although, to be honest, I've always had my doubts...

What exactly is egoless programming?

I guess it all depends on what you mean by ego, so I turned to my favourite text on management and psychological (and all that kind of stuff) where I found the following definition:

> "**ego** has to make sense of the internal conflict in our mind between the id and superego and the external world. The ego is the decision-making part of personality and is engaged in rational and logical thinking. It is governed by the reality principle." (Mullins, 2002)

Now this looks a bit confusing. Does egoless programming mean programming without rational and logical thinking? Well, I've seen my fair share programs and I must say a lot of them do seem a lack rational and logical thinking, but I really don't think anyone wants to advocate this is a design and programming technique. What would an irrational program look like? What would an illogical program look like?

I'll admit, the first half the sentence "conflict in our minds" makes sense, I often experience conflict when I'm programming: do I use ++i or i++, or even i:=i+1? And I'm often torn between doing something the "modern" way (say with a template function specialisation) or the "old fashioned way" (with lots of verbose code). So maybe conflict-less programming would be a good idea.

But then, if there was no conflict why do we need people? It is the human judgement, honed over years of programming that makes developers so valuable. If there is no conflict, if there is an obvious solution every time we can automate it, bring on the case tools. Or even just enhance the compiler. The fact is, resolving conflicts and balancing competing forces is a fundamental part of what we as software developers do.

So clearly this is not the right definition for ego.

Maybe what the books mean is superego-less programming, if egoless is good, then surely, super-egoless must be better?

> "**Super ego** is the conscience of the self, the part of our personality which is influenced by significant others in our life. " (Mullins, 2002)

Well programming without consciousness, that doesn't sound good. Once or twice I've programmed into the wee-small hours of the morning and come pretty close to coding in my sleep but I wouldn't recommend it.

I'm sure I've met managers who would prefer it if our significant others didn't influence our lives. A manager once asked me to cancel a holiday, however the thought of how my significant other would react prevented me from agreeing.

So, I don't think superego-less programming is a good thing either. That leaves as with id-less programming.

> **"id** consists of the instinctive, hedonistic part self.." (Mullins, 2002)

This sounds more like it. Id-less programming - programming without fun. This is work, this is serious, fun has no place in code. (Erh, why did I get into this business?)

While I've know many managers who don't see fun as an essential part of the job, I can't say any have really objected to a bit of fun. After, a bit of fun, a few smiles in the office, makes the day much more, erh, fun.

Try again

So I'm not a lot closer to understanding what egoless programming is meant to give me. Maybe I'm reading too much of the definition maybe what we want is a simpler definition of ego. So this time I turned my dictionary, this definition looks more hopeful

> "ego n. pl. egos 1. The part of a person self that is able to recognise that person as being distinct from other people in things. 2. A person's opinion of his or hers own worth: men with fragile egos." (Collins, 2001)

So maybe egoless means we can't tell ourselves from other people, we lose ourselves in some kind of great group. Let's forget that 20th century literature and popular philosophy emphasise the individual, we want to hire programmers who can't tell themselves for anybody else. That might get really confusing at times, and where is the difference of opinions that can lead to so many useful insights?

Or maybe you want people who think they are worthless, we want programmers who don't have very high opinion themselves. I can see this be a great position for manager to be in when it comes to the annual pay reviews, or for contacting renewal.

> **Manager**: Well then Bob, I'd like to renew your contract for another 3 months
>
> **Bob**: No, no, you don't want me, remember that bug in my code? John had to fix it last week?
>
> **Manager**: Yes, I see, well, I can't throw you out on the streets, so what say I keep you for another three months with a 25% reduction?
>
> **Bob**: I am not worthy

It seems to me that ego is an essential part of people, and even of software people. On the whole its more fun to work with people who are confident - I'm sure most people would say no if invited to join a team of people racked by self-doubt. Actually, we want programmers with ego's. We want programmers who care, we want people who say "I'm proud to have worked on this project."

It seems "egoless programming" doesn't really stand up to analysis. We want people who are rational, logical, proud of their work and bring a positive attitude to work.

Origins and setting

The term egoless programming originated with Gerald M. Weinberg's *The Psychology of Programming* in 1971. In the silver aniversay edition. (Weinberg, 1998) and in IEEE Software (Weinberg, 1999) Weinberg has reprised claimed his original ideas had been misunderstood and misinterpreted. To be fair, Wienberg was making an argument for teams, it just happened people remembered the sound bit, *egoless programming* and forgot a lot of his other ideas.

For Weinberg *egoless programming* is about code reviews and letting others comment on your work. Although the benefits of code reviews are well known they are not always conducted routinely. There are a variety of problems, not least because it often takes longer to review code than it takes to write it in the first place.

In Weinberg original essay he suggested programmer's egos lead them to hide their code, and protect it, they don't want other people passing comment on it. Does this really happen? There is no shortage of programmers posting their code on SourceForge for all to view.

I think the problem is more the social setting of the review. In reviewing your code, and giving feedback, there is a great capacity to hurt someone's feelings. Receiving feedback can be hard, and it can hurt. Simply being told "think of egoless programming" is like being told to keep "the British stiff upper lip."

Giving criticism so it doesn't hurt, and receiving criticism without feeling personally attacked are skills themselves. One organisation I know did code reviews by e-mail, a day or two after your check-in you would receive an e-mail from your reviewer listing your mistakes. This may be efficient but it is also brutal. Some developers would actually hold off check-in until the last minute then make a lot in a short space of time, often this usually meant the reviewers could get the reviews done before the release so negating the whole point of a review.

Neglected teams

Programming teams are hardly knew concepts, they have been knocking around software development books for many years. Ever since programming projects got beyond the abilities of one person we have had teams.

The problem is that our traditional text books devote hundreds of pages to technical issues and almost nothing to team work. A statement to the effect that "much software is developed in teams, good teams need egoless programming" is about as far as many go.

For example, if Pressman (1994) even mentions teams in the text it doesn't make it into the index. His fourth edition (Pressman, 1997) edition is slightly better but with over 800 pages you could easily miss the few pages on teamwork. Other texts can be better, for example Somerville (2001), devotes a whole chapter (20 pages) to managing people, and six pages alone to team working, pretty good going until you notice it is an 800 page book!

So, while we can all agree that teamwork is important few of us actually devote time to thinking about how to make it work. There is an assumption somewhere that teams just work, once told we are a team then all is sweetness and light.

Personally I don't see this myself. Teamworking is a skill just as much as C++ or SQL is, and we need to learn it. In fact, each team needs to relearn the skill itself. This may be especially true of programmers, I know quite a few like myself who preferred the warm dry computer room at school to the wet, muddy football pitch.

The point is teamwork doesn't just happen, we need to be encouraged to work in teams. It can be scary talking to new people, it can be terrifying trusting somebody else to do work, and it can be demoralising to see somebody else do apiece of work that you really want to do, and I haven't even mentioned fixing of the people's bugs!

Simply extolling the virtues of teamwork, asking people to practice "egoless programming" doesn't make it happen. If a company want these objectives they have to work for them.

The irony

Perhaps the greatest irony of all is that people are social animals, we actually like interacting with other people, working and playing with other people. In fact work can be so much more enjoyable when you work with great team, people we trust, people we value. When we trust people work becomes so much easier, we don't need to keep an eye on them, we don't need to secretly double check their code, and we feel happy for them to the see our code.

There are times when competing is the right solution. Put two teams in football field, two companies in the same market, any company and Microsoft! Competition is a bedrock of capitalism, competition drives us, we all want to win. But competition is not always the right answer, sometimes we get more by co-operating then competition.

This is why companies exist, because as a group of co-operating individuals we can achieve something that the same individuals can't achieve by competition. And that is why programming teams exist, because some programmes and is too big for one person to write.

It isn't easy

One of the most difficult things for a team to do is to overcome Conway's Law (Kelly, 2003, Conway, 1968). This is usually stated as:

> "if n developers work on a compiler, it will be an n pass compiler"

It is so easy on any project to count the number developers and divide the project into that number of pieces, even if the project can divided into more, or less pieces. This way we can keep everyone busy, everyone can have their own space and get on with a piece of work.

But is this always sensible? If a project naturally has a front and a back-end why divided it into three pieces just because we have three programmers? The bigger question is: what are we trying to optimise here? Are we just trying to make four programmers look busy?

A common variation on Conway's Law states:

> "if n developers work on a compiler, it will be an n-1 pass compiler, somebody has to manage "

The assumes that the roll of managers is as police. They are there to command and control the workers (programmers) and ensure they get the work done.

Teamwork

If we want our teams to really work well and develop good software we need to move away from simple exhalations to "egoless programming." We need to break our projects into the optimal development and work as teams, rather than slavishly divide by the number of programmers and work as a group of individuals.

This means we have to think seriously about making our teams work well together. Fortunately some of the more recent writings on software development have started to put a greater emphasis on teamwork (e.g. Eckstein, 2003, Cockburn, 2002) and the (in)famous pair programming (Beck, 2000, Coplien, 2003) is a good example of this. But learning to work as a team doesn't start and stop with pair programming.

It helps if a team actually knows one another. Do they lunch together? Do they socialise together? Some companies try to get teams to socialise by arranging a Friday afternoon "beer bash", these can have an artificial feel to them (especially if everyone is driving home) but is a start.

Other high profile "teambuilding activities" like white-water rafting, or paint balling can also be the subject of jokes and mockery. Its important to match your team building efforts to the attitude of the team, maybe a trip to the pub, or the cinema is more in keeping with your team. Even ad hoc communal gathering areas such as kitchens can be far more effective than one off events.

This requires ongoing expense and commitment from the company, after all that kitchen space is an continuing cost while a paintball day is a one off expense. But this long-term commitment is what is required to build a good team, it doesn't happen over night.

Companies need to look at their own mechanisms: do they reward people for teamwork? Or do they award annual bonus based on individual heroic coding efforts? And are teams broken up once a project finishes, or can they move together onto the next project? Are people allowed to sit together? Or are people squeezed into what ever space can be found when they arrive on day one?

There is a lot individual managers can do here too. If they see their role as commanding and controlling the developers they aren't going to get the best from them. They need to learn to develop their teams, encourage people to work together and learn together.

Keep your ego, make teams work

The fact is we want developers to have egos, we want them to be proud of their work, we want them to think logically and rationally. But we want to harness these ego's within a team. We want the team to succeed. This can only happen if we are socially aware and build towards this goal.

We can't expect any of this to happen just because we say it should happen. We need to work hard to make these teams work, people need to learn how to work in teams, how to work with their colleagues, how to give constructive feedback and how to accept feedback.

Unfortunately, neither Microsoft nor Rational sells a tool to do this, its something you have to create yourselves. You can bring in outside help but this is a long process, the rewards are great but it won't happen over night.

Bibliography

Beck, K. (2000) *Extreme programming explained,* Addison-Wesley.

Cockburn, A. (2002) *Agile Software Development,* Addison-Wesley.

Collins (2001) *Collins Paperback English Dictionary,* Harper Collins, Glasgow.

Conway, M. E. (1968) How do committees invent?, *Datamation*.

Coplien, J., and Harrison, N. 2003 *Organizational Process Patterns*, http://www.easycomp.org/cgi-bin/OrgPatterns, Wiki web site for forthcoming book

Eckstein, J. (2003) *Scaling Agile Processes (forthcoming),* Dorset House, New York.

Kelly, A. (2003) The original Conways Law, *Overload.*

Mullins, L. J. (2002) *Management and organisational behaviour,* Prentice Hall.

Pressman, R. S. (1994) *Software Engineering: A practictioner's approach - European Adaptation,* McGraw-Hill Book Company.

Pressman, R. S. (1997) *Software Engineering: a practioner's approach (European adaptation),* McGraw-Hill.

Somerville, I. (2001) *Software Engineering,* Pearson Education, Harlow.

Weinberg, G. M. (1998) *The psychology of computer programming,* Dorset House Publishing.

Weinberg, G. M. (1999) Egoless Programming, *IEEE Software*.