JJ's Reference Architecture

Part 9

-

Author: Jan-Joost van Zon
Date: December 2014 – June 2017
Under Construction

Team Management

[Under construction. A personal brainstorm of possible ways to go.]

This documentation section describes techniques you can employ for the hands-on technical leading of a software development team.

Focus Points

It is still debateable, whether everything falls into these key areas. I am trying to structure this…

- Dialog
- Support
- Direction
- Planning
- Strategy

- Keep your cool
- Be open to other views

Dialog

- Is the one I might forget about, but if I put extra focus on it, it would help a lot.
- Exchange ideas between team lead and team member.
- Exchange ideas between team members.
- Get to know eachother to make optimal 'use' of skills but also to enhance morale en good feeling, mutual understanding and promote openness.
- Assess what the team member's problems are, regarding the way he feels about the work.
- Dialog should sometimes be limited too, because this job requires focus and quiet often, in order to do it well.

Periodic Individual Meetings

- Periodically talk to each developer individually about work done. Ask:

    - Currently working on
    - Done this period
    - Problems encountered
    - Large time-consumers
    - Positive notes
    - Other points of attention
    - Solutions to the problems
    - Next work to do
    - Good question: 'What way of working, would make you enthusiastic? Not looking at the way we work here.'
    - Good question: 'Can you tell me some interesting new things you found out?'
- The team lead also asks himself the same questions.

Reporting to Management

- Lead makes a report of issues discussed, preferably not grouped by employee, but grouped by development area. Employees can be mentioned with tasks.
- Lead discusses report with management.

Limiting Dialog

- Prevent people from getting distracted by too much dialog.
- Promote sympathy for people that have ear plugs with music on.
- Only bother people when useful.
- Many people need to focus to program.
- You should be considerate towards people that need undistracted focus, ask more carefully if they can be disturbed and make them know, that is OK to say no to that.
- There are people that seem to handle distractions better, although I doubt their productivity does not get harmed.
- But perhaps I am wrong and these are the people that you can bother more often to vent about am issue.
- Consider moving a conversation to another room, so others can focus.
- Not speaking too loudly also helps a lot.

Support

Support from the team lead is mostly inward directed toward the team members. But support is also has an outward direction from the team towards the rest of the organization or to the customer.

- A strategy of giving team members more freedom.
- Give people the responsibility, that their code works well and that others can still read it and adapt it.
- Support with solutions and techniques.
- Give examples
- Lead by example.
- Formulate suggestions with prefixes like "I would do it like this." or "In my experience this works well."
- So do not talk in definites, allowing the other person to form their own opinion and understand why or why not to take a certain route.

- You are going to have to accept that things will be programmed in a less than perfect way, or not precisely the way you would do it, and allow programmers to grow into making things better on their own.
- Look for a compliment to go along with it.
- When someone does not asks enough questions, as a lead you should inquire yourself.

Question Rotation

- Questions from outside the development department are handled on a rotating basis.
- Everyone takes turns in a day of being the one to ask questions.
- This will make sure most of the time developers can continue undisturbed with their programming work.

Support & Direction

- Support and Direction are very closely related.
- Support is more oriented towards the coworker: what can you do to make him do a better job and feel better doing it and advance in his job.
- Direction is more about how can I make the work be beneficial to the client or rest of the organization. It is more outward directed.
- Both are about how can we do the work optimally to serve the goals.

Developer Skills

You can test someone's skills in something by giving them a specific task, that requires one skill in particular.

Meta-skills:

- Research
- Documentation
- Bug fixing
- Process flow programming
- API programming
- UI design (not graphical, just the screen layout and program flow)
- Programming for reusability
- The worst-practices and almost worst practices lists from the architecture documentation.

Tools & techniques:

- Database development
- SQL

<TODO: Add more… Could be a pretty extensive list.>

Direction

Miscellaneous Pointers

- Guard the main outlines
- Accept that people work outside the borders.

- Be involved at the beginning of a task at the end of a task and during the execution of a task too.
- If your involvement starts in the middle, do not try to to stear it too much into another direction.
- You could find the biggest problem area and then suggest that as a point of improvement.
- Meetings: (borders the topic of Dialog)
    - Try keep focus in meetings, so things do not go off topic too much.
    - On one hand I dislike it, when a meeting about a topic that could have been 30 minutes turns into 5 hours of a whole range of topics with only loose ends and inconclusives at the end.
    - On the other hand I dislike it, when the response to a proposal is: that is not what this meeting is about.
    - The intermediate version would be brief attention to off-topics and main focus on the topic at hand.
    - It also depends on whether it is a general meeting or a conversation about a specific topic.
- API choice: Medior developers and juniors are *not* free to choose any API without collaborating with seniors, leads and architects. It just has too high an impact. Not that we should not be open to new API's.
- Framework development: Same point as API choice: juniors and mediors only adapt framework in colaboration with seniors, leads and architects.

Vary a Developer's Tasks

What I often see, is that each developer is assigned to his application and never changes focus, because everyone feels, that this person is the 'go-to guy' for this topic. The motivation behind this is that it would be more economical to let the guy that knows most about it, do the work. That makes sense intuitively, but what's intuitive, isn't always what's right. It is an over-simplified view on the issue that will cost you money. You lose something when you do not vary a programmer's work.

- There is the risk that comes a developer becomes ill or falling away for another reason.
- It excludes other developer's ideas from the subject.
- It will limit the developer's personal development, because he never gets to do something else.
- It will encourage developers to not think about: "What if someone else had to work with my code?"
- The lack of critical peer-review could make the code quirky and hard to maintain.
- There is no opportunity for developers to learn from eachother.
- There is less incentive to review eachother's code, which can translate to bugs and increased development time.
- It will be more difficult to scale up the amount of developers on a job.
- It will be more difficult to move a developer from one project to another, for instance when there is not much work in one area anymore, or higher priority work in another area.
- It does not promote a joint way of working, that makes it easier for others to pick up eachother's work.

I cannot stress enough that it is a bad idea to let a only one developer work on a specific thing all the time. Did I mention that the the software developer's personal development is slowed

down? I have never been in a situation where this costs more and I have seen productivity go up doing it.

Here are suggestions of how to spread and vary the work over multiple team members. Variations can be mixed and mingled at will. That is the point. Be flexible.

- Variation 1:
    - Junior front-end
    - Medior back-end
    - Senior framework
- Variation 2:
    - Senior coaches medior
    - Medior coaches junior
- Variation 3:
    - Senior reviews code of mediors
    - Medior reviews code of juniors
- Variation 4:
    - 1 developer does rework of previous iteration
    - Another developer does new features.
    - Every now and then they switch
    - That way rework is not in the way of new developments.
- Variation 5:
    - One developer answers questions coming from outside the department and each day a different developer does it.
    - If needed other tedious task go to that developer.
    - That way the other days you know you can keep working on actual software development uninteruptedly and you at most only lose that one day that it is your turn.
- Variation 6:
    - Each developer or a set of developers works on a different part of the application or major feature.
    - Perhaps a junion/medior pair can work on a feature together.
- Variation:
    - A developer gets a similar task as before because he is now proficient at it.
- Variation 7:
    - A developer gets intentionally different tasks than before, to spread knowledge over the team and enriching the developer's knowledge.
- Variation 8:
    - Some developers work on high-risk experiments.
    - Other developers work on low-risk productive feature development.
    - That way you create room for doing experiments that you do not know the outcome of yet or do not even know how long it is going to take or if it gives any useable results, without it actually stalling productivity completely.
- Variation 9: (very specific)
    - Seniors do:
        - Task management
        - Code review
        - Architecture
        - Framework development
        - Data model design for the most part

        - Software design
        - Rsearch / orientation of possible solutions
    - Mediors program:
        - Business logic
        - Data model implementation
        - *Some* framework and data model design but in tight collaboration with seniors.
    - Juniors do:
        - Front-end development, but under supervision so that patterns are followed.
        - Repetitive tasks, but only ones the junior can handle.
        - This may sound a bit deprecating, but sometimes a junior developer, when asked to do something in a very specific way, apply it the same all over the place, do it totally differently when you turn your back. That says something about the need for guidance. You have to monitor junior developers.

Upward Management

- Lead does ask critical questions to management about strategy and decisions.
- Lead advises management about (future) employees to hire.

Budgeting

- Lead investigates software and hardware requirements and opportunities to use newer software, offer better quality, quantity, service or reduce costs.

<TODO: Write a little more about it.>

Planning

Borders the topic of 'Direction'.

Tasks

- Split up tasks into multiple distinct pieces of work.

    If the task is to broad, some people that think they can add another 200% of work to it, that they made up themselves. It will be unclear where a task starts and ends.

- As the lead, always give an expected estimated time it will take.

    This will make someone aware of expectations, and of how deep to go.

- Lead must control task subdivision
- Lead must be asked for new task and discuss how to solve
- Lead must be informed of completed task
- Lead must be asked in case of unclarities
- Lead communicate the following rules

    - Begin Task -> Go to lead
    - End Task -> Go to lead
    - No Task -> Go to lead
    - Questions/unclarities -> Go to lead.

    Fill in the leads name and basically hang up the wall.

- Data data model extension task should mention:

  '+ Add the basics around this field to all the architectural layers.' Discuss verbally which layers. Use the Layering Checklist from the Software Architecture document to make a selection.

- A task should involve all the required work, but what is not asked should be left out

  But this comes with problems.
  Ideally a developer knows which things to involve in a task, and which things to not involve.
  But in practice this is often not the case.
  One developer may not know what a task implies, another developer will involve too many side-issues.
  If you need to mention everything in a task, this is too much of a burdon on the one administrating the tasks.
  > TODO: Brainstorm and reformulate later.

Strategy

- Strategy is broad. It takes anything you could do and molds it to fit your current goal.

<TODO: Write something about it.>

Keep Your Cool, Be Open to Other Views

<TODO: Write some more about it.>
<TODO: Use this phrase? All this heated discussion about subjective things. Can't we just keep calm and discuss pros and cons and try to solve the problem together? Nothing is black and white in the choice of tools and techniques. It is always a trade off.>
<TODO: Use this phrase? Be less sarcastic.>
<TODO: Use this phrase? Someone with a way of working that does not promote maintainability, might not just be stupid.>

Software Lifecycle

Branching, Versioning & Release Management

<TODO: Describe a versioning methodology. >
<TODO: Assembly versions.>

Server Architecture

[Under construction]

The server subdivision is subject to the needs of the organization, so this overview is just a suggestion. The main concerns are safeguarding and keeping things optimal. Economics might force you to look for alternatives, but from a technical point of view the full set of servers with

recommended configuration is advised. Cutting corners might make your IT run less efficiently, which would translate to cost overhead too.

<TODO: Mention the split-up into a C: and D: drive.>
<TODO: Reconsider the sizes of the development workstation drives after some cleaning up and counting disk space and considering extra dev tool requirements.>
<TODO: Consider the machine configuration needs in more detail.>

Stage	Name	Remarks	Configuration Focus Points
Development	Database server	Stores a development copy of all the databases we use.	SQL Server, decent performance, particular focus on having enough RAM.
Development	App server	Where development can use a shared FTP server if needed, run long processes to aleviate the development workstations. Can also host shared web services, be it third party, be it internally developed ones, even though for that last thing it is usually better to run it on the development workstations.	IIS, preferrably many-core. SQL Server installation is advised, for delegating number crunching from the main development database server to another server. RAM is also important, since heavy number crunching processes may use a lot of memory.
Development	Source control server	For storing the source control database, running the source control services, running builds, unit tests and code analysis upon each check-in.	TFS. Must be decent configuration for each checking requires a heavy process to run, and the development team has to be able to work efficiently.
Development	Workstations	Each software developer's own machine.	Two 21" monitors. No laptops, those run 2x slower. Core i5 for junior and medior developmers. Core i7 for senior developers and sofware architects, since they will more commonly work with larger solutions. At least 8 GB RAM, so you can run large-cache applications and services.

			SSD for of 256 GB. Split up into a C: drive for windows <TODO: specify size> and a D: drive for data, main source code, but also aother frequenty accessed things. An extra 'spinning disk' drive with at least 256 GB storage for amont other things the ability to hold large database backup files.
Development	Laptop	One laptop for a whole team, just to connect to your workstation when you are in a meeting or being on the road to a customer	Relatively low specs. Core i3, a moderate amount of RAM.
Production	Database server for number cruching	For doing the heavy processing, like datawarehouse imports and processes, heavy reporting, heavy pre-calculation processes, to aleviate high-traffic production servers.	Many cores
Production	App server for number crunching	Same as above.	Many cores
Production	Database server for high traffic	For all the databases involved in high-traffic, storing data of user applications and services.	
Production	App server for high traffic	For all the production web sites and services.	Note that RAM is very relevant, to meet in-memory caching needs.
Test	Database server		
Test	App server		

DTAP

<TODO: Write something about this. Include:

- Explain DEV, TEST and PROD. And ACC. And then also the INTERNAL and EXTERNAL production environments and their benefits, mostly with regards to publishing. Use the prefix ACC, instead of ACCEPTANCE, because the prefix goes all over the place e.g. ACC_BuildlingBlocksDB instead of ACCEPTANCE_BuildlingBlocksDB. But then again I like this better: acceptance.electroluxservice.JJ-bv.nl >

Folders

If your servers and all your development workstations have a D: drive for data, then put the folders on the D: drive, otherwise put the the folders on the C: drive. Make a folder in the root of the drive with your company name:

D:**JJ**

In this folder, each environment gets a sub-folder written in all capitals.

D:\JJ**TEST**
D:\JJ**PROD**
D:\JJ**DEV**

Even machines with only one environment on it, should get a sub folder with the environment. That makes it better visible what environment you are working on, you can easily emulate the situation on a development workstation without additional configuration and it allows you to move environments from one server to the other.

Also add the following folders:

D:\JJ**Install**
D:\JJ**Backup**

<TODO: Add descriptions to each folder above.>

Those are not put in the environment folder.

The environment folder can contain the following sub folders:

D:\JJ\PROD**Images**
D:\JJ\PROD**IO Files**
D:\JJ\PROD**Log**
D:\JJ\PROD**Utilities**
D:\JJ\PROD**Web**

<TODO: Add descriptions to each folder above.>

The Images folder contains images e.g. uploaded from an application, not images that are content of the application, so not icons or basic content of the web site, but user-uploaded images or images uploaded by content management systems. The Images folder can contain sub-folders to keep images apart from eachother, that belong to a different application or set of applications.

D:\JJ\PROD\Images**QuestionAndAnswer**
D:\JJ\PROD\Images**Synthesizer**

An application image sub-folder can contain again sub-folders, for resized images with particular dimensions:

D:\JJ\PROD\Images\QuestionAndAnswer	Contains original images.
D:\JJ\PROD\Images\QuestionAndAnswer**100x100**	Images resized to 100x100 pixels
D:\JJ\PROD\Images\QuestionAndAnswer**320x280**	Images resized to 320x280 pixels

The resolutions above are simply examples. You can have different sizes per application.

<TODO: Describe the Utilities folder, that you use fully qualified application names and version sub folders. Same for the Web folder, and add to that that it contains both web services as well as web applications.>

Development Workstation

Development workstations should have the same kind of folder subdivision, also put on the same drives as on the servers. You might not put web sites in these folders, but Imags, IO Files and Logs should be present in the same folder structure as the servers.

Put the source code folders in the same spot as all your coworkers. Then things keep cooperating with eachother. Most of the times relative paths work, but sometimes they don't so it is a good plan to all have out copies of the source code in the same location. In case of TFS it should be D:\TFS that is mapped to the outermost root of the source control system. Not to a branch, not to a Collection, but to the server name or IP address of the source control server.

<TODO: Use this phrasing? - TODO: If servers store data files on C, then development worktations had better store it on C too, because it is incredibly confusing to have it in a different spot in development, for instance when you try to emulate a TEST or PROD environment on your development workstation and the letter C or D can be overlooked so easily in configuration files that you keep on making mistakes.>

Backups

<TODO: Write something.>

Appendices

Appendix A: Layering Checklist

This checklist might be used if you want to bulk-program the architecture for an application by going through all the layers one by one:

- Data: Database structure
- Data: Data migration
- Data: Entity classes
- Data: Repository interfaces
- Data: Default repositories
- Data: NHibernate mappings
- Data: SQL queries
- Data: NHibernate repositories (optional)
- Data: Other repositories (optional)

- Data: Other mappings (optional)
- Business: LinkTo
- Business: Unlink
- Business: Cascading: DeleteRelatedEntitiesExtensions
- Business: Cascading: UnlinkRelatedEntitiesExtensions
- Business: Enums
- Business: String Resources
- Business: EnumExtensions
- Business: Validators
    - Delete Validators too
    - Warning Validators (optional)
- Business: SideEffects
    - SetDefaults SideEffects too
- Business: Managers
- Business: Extensions
- Business: RepositoryWrappers
- Business: Dtos (optional)
- Business: Calculations
- Business: Visitors
- Business: Factories (optional)
- Business: Api (optional)
- Business: EntityWrappers (optional)
- Business: Other helpers (optional)
- Presentation: ViewModels
    - Item ViewModels
    - List item ViewModels (some may only need IDNameDto, no ListItem view model)
    - List ViewModels
    - Detail ViewModels
    - DocumentViewModel (optional)
- Presentation: ToViewModel
    - Singular forms
    - WithRelatedEntities forms
    - ToListItemViewModel
    - ToScreenViewModel
    - CreateEmptyViewModel (not every view model needs one)
- Presentation: ToEntity
    - Singular forms
    - WithRelatedEntities forms
    - From screen view model
- Presentation: Presenters
    - List Presenters
    - Detail Presenters
    - (Edit Presenters)
    - Save methods in Detail (or Edit) Presenters.
- Presentation: Views (Mvc / UserControls…)
    - List Views
    - Detail Views
    - Main View (optional)

Appendix B: Knopteksten en berichtteksten in applicaties (resource strings) (Dutch)

Er is een bepaalde structuur waar binnen we werken voor knopteksten en meldingen in onze applicaties. De hele bedoeling is maximale herbruikbaarheid, minimaal vertaal werk en correcte teksten. Dat doen we door hele algemene teksten op plaats X te zetten, zo veel mogelijk domein termen op plaats Y, en alleen wat er dan over is, komt in specifieke projecten te staan. Dit kan het verschil betekenen tussen 100'en of 10000 teksten.

Resources worden op dit moment overal neergezet waar ze niet thuis horen, met verkeerd hoofdlettergebruik en verkeerde interpunctie. En op andere plekken worden resources gewoonweg niet gebruikt en staat alles hard op 1 taal.

Hier moet secuurder mee om worden gegaan. Van ontwikkelaars wordt verwacht zowel de Nederlandse taal als de Engelse taal in de resource files te zetten. Bij twijfel over Engels, vraag het een collega.

Hoofdletters, interpunctie, spelling

Hoofdlettergebruik etc. is conform de taalregels van de betreffende taal.

1) Resources kunnen hele zinnen zijn. Die moeten correct geschreven zijn: In het Nederlands begint dat met een hoofdletter en eindigt het met een punt, tenzij het een vraag is, dan met een vraagteken. Blijkbaar is het nodig om dit aan te geven, want het gebeurt vaak niet goed.
2) Voor Engels gebruiken we de Amerikaanse spelling.
3) Namen van properties, classes en andere titels zoals knopteksten zijn in het Nederlandse zijn als volgt: "Links in artikel", dus alleen beginnen met een hoofdletter. En dus geen punt erachter.
4) Namen van properties, classes en andere titels zoals knopteksten in Engelse titels doen we als volgt: "Table of Contents", dus alle woorden beginnen met een hoofdletter, alleen onbelangrijke woorden zoals 'in', 'and', etc. in kleine letters.
5) Er is dus een verschil in hoofdlettergebruik tussen volzinnen en losse titels.

Assemblies

Resources worden met de assemblies meegecompileerd*.

Termen worden zo veel mogelijk hergebruikt. Daarom zijn er plekken bedacht waar de termen thuis horen. Je moet in deze volgorde op zoek naar een resource die misschien al bestaat: (Update: De hoeveelheid verschillende plekken waar resources staan is een zwakte van dit ordeningssysteem, omdat het verwarrend kan zijn. In toekomstige oplossingen is het wellicht een idee om resource teksten mmer op één centrale plek te zetten. Dubbelzinnigheid van termen in meerdere domeinmodellen is daarbij wellicht meer een uitzondering dan een regel, waar omheen gewerkt kan worden.)

1) 'Save', 'Close', 'Edit', etc. staan in Framework.Resources, toegankelijk via de *CommonResourceFormatter* class.
2) Validatiemeldingen uit Framework.Validation, toegankelijk via de *ValidationResourceFormatter* class.

3) CanonicalModel: een tussenmodel voor uitwisseling van gegevens tussen verschillende systemen, toegankelijk via de *CanonicalResourceFormatter* class.
4) Business layers bevatten alleen vertalingen voor de overige teksten die niet in het canonical model staan.
5) Ook teksten die niet direct domeintermen zijn, maar wel in applicaties worden gebruikt op plekken waar het gaat over een bepaald business domain, mogen in de business layer, zijn resources gezet worden.
6) Presentation layer bevat over het algemeen geen teksten. Die zetten we in de business layer: we hebben al genoeg plekken waar we resources neerzetten.

Tips

1) Gebruik van placeholders zoals {0} is toegestaan, maar dan moet je wel een class erbij maken, die de placeholders vervangt. Zie Framework.Resources voor een voorbeeld. Het is dan verstandig om de resources zelf internal te maken en alleen de class die de placeholders vervangt public te maken. Kijk echter uit dat je het daarbij geschikt houdt voor meerdere talen, want een creatief met placeholder opgebouwde resource string werkt al gauw niet voor een andere taal.
2) Negeer dat de beschreven werkwijze kan resulteren in berichtteksten met hoofdlettergebruik zoals: 'Het **Ordernummer** is niet ingevuld bij de **Bestelling**.' Als we dit aanpakken, doen we dat met een algoritme, niet met nog meer resources.
3) Het is verstandig om teksten in de applicatie algemeen te houden. Dus bijv. 'Artikelen', i.p.v. 'Artikelen in dit boek'. Dit scheelt vertaalwerk. Ook dit is verstandig: 'Artikel 1: Naam is verplicht.' Daarbij zijn de teksten 'Artikel', 'Naam' en '{0} is verplicht.' waarschijnlijk allang vertaald. Door de 'dubbele punt' notatie aan te houden ('Artikel 1:'), voorkom je het verwerken van de term in een zin, wat voor iedere taal een compleet andere grammatica kan zijn, wat voorkomt dat er complete volzinnen vertaald moeten worden.

** Resources staan niet in een database, omdat het applicaties trager maakt en kun je de code niet draaien op omgevingen waarbij je geen toegang hebt tot de database. In sommige situaties kun je niet eens compileren zonder toegang te hebben tot een specifieke database. Bovendien geeft mee compileren van resources in specifieke projecten ons de mogelijkheid dubbelzinnige termen anders te vertalen per business domein.*