Guide for Software Factory Teams

Author: Henri Karhatsu, participant in the first Software Factory project called Gnobles

Forewords

About Software Factory

This guide is meant for all the students participating in Software Factory projects in the Department of Computer Science, in University of Helsinki. Software Factory is quite a unique concept that has three main viewpoints:

1. Education - Software Factory is a great place for students to learn software engineering in a realistic but still safe environment.
2. Research - Software Factory provides researchers with an outstanding possibilities to examine various angles related to software engineering, team work and so on.
3. Entrepreneurship - Software Factory with real business projects encourages entrepreneurship and offers start-ups a place to implement a prototype of its software product.

Please read more about Software Factory at www.softwarefactory.cc.

Work-like but safe environment

For students Software Factory provides a realistic working environment. There is a real customer with real business needs that the team should be able to fill. You work constantly together, several hours a day, just like in real workplaces.

On the other hand Software Factory provides students with a safe environment to learn. Typically in work life tasks are allocated so that the people who are most experienced in certain area do the tasks in that area. In the Factory on the other hand you are welcome to try new things and learn. If you fail, nobody is firing you.

It is also quite probable that the project will contain people from several cultures. In the first Factory project there were situations when people were at the same time speaking Finnish, English and Hindi. So the Factory is a good place to learn cultural aspects and English, which both are valuable assets in the work life.

About this guide

This guide is written based on the experiences from the very first Software Factory (SF) project. It is very important to understand that in each project the team should choose and find those practices that best fit for that team. However, this guide may give some ideas that are good for your team and helps teams to achieve better results in the project.

The SF teams are not supposed to be traditional command-and-control teams where the manager says what the others do. Instead SF should encourage people towards self-organization and shared leadership. It means that this guide is meant for all project participants, not just for those who work for example in some lead or senior role.

You may notice that the guide contains lots of methods and tools that can be found in the agile software development literature. This is not a coincidence; agile software development is a modern way to create software products and especially useful in short projects. Imagine the waterfall-like situation in your seven-weeks project where you would spend the first four weeks with specification and the last three weeks with implementation. What would you be able to do in three weeks? Are you sure the original specifications would still be valid after the first week of implementation? Or imagine that your team would have strict roles where each individual would do her own work without too much "interrupting" the others. How efficient would the team be?

Agility is about using certain methods but obviously that is not enough. What matters is that how your team uses these methods. Most of the methods like daily meetings, pair programming or retrospectives are meant to encourage communication between team members. There is no way to guarantee that your project will be a success when you use all the things listed below. However, there is a guarantee that if your team does not communicate, then your project is going to fail.

Starting a project

Who are my team members? What are we going to do here? What technology will we use? What is Kanban? In the beginning you may have more questions than answers. Don't worry. Things will usually sort out. There is no single and correct way to tackle all the possible problems in the beginning but here is something that may help you.

Here are two common advice which will be useful especially in the beginning of the project but also later on when the projects gets going:

1. Ask yourself: What is the most relevant thing to do next?
2. Start working with small pieces. Just get things moving, little by little.

Well, what could be an answer to the first question in the beginning of the project? If you know nothing about the product that your team will be doing, the most important thing might be to get some domain knowledge. Try to get your customer close to you and squeeze the information out of her. Or if most of your team members don't know much about the technology that you will be using, then start learning it. Or maybe you need to setup the development environment before you even start learning the technology.

The second advice has two aspects. At first, it is trying to say that even though you have hundreds of questions or problems, don't try to solve all of them at once. Secondly, it is important to get things going on. When you just do things, you and your team will learn. You will find new viewpoints, and maybe even some of the problems may disappear without even noticing.

Experiences from the first project

When we started our project, the main problem was that the team had very limited experience with Ruby on Rails, our development technology. There were four people that had participated the Rails course a couple of weeks earlier but the others had basically no experience at all. So based on this the team decided to use the first week just on learning Rails and also the version control system (Git). There were no requirements that anyone should do any real tasks; it wasn't important to produce features to the final product, it was important to learn.

The team was split into three sub teams that each had one senior developer in it (a person who participated the Rails course). Each sub team was developing a so-called 1-day-app, which means "any application that you can build in one day". In practice this was done almost two days. Each sub team, or basically the senior in the team, could decide the learning methods to be used. We relied heavily on pair programming and even used coding dojos (each writes code for 5 or 10 minutes and then you change the writer). Afterwards thinking this probably wasn't the best way to learn. Some team members said that they didn't have time to think and they couldn't try things enough by themselves. It's said that pair programming works best when you have two juniors with pretty much same level of skills. But in our case we didn't even have juniors, we had beginners. So, what is the lesson learnt? Your team needs to find the learning methods that best fit for your team.

During the first week we also listed the core features in the product, thought how our Kanban board would look like (see more below), decided the definition-of-done (see more below), setup the development environments and implemented the initial database and some other basic features of the product. These things were mainly done by senior developers while the juniors were learning. However, it is important to understand that all of these things were discussed with the whole team before they were accepted.

Kanban and work process

Kanban is a process model that comes from lean software development, which is tightly related to agile software development. You may have heard about Scrum, which is another - probably better known - agile process model. When implementing Kanban in your team, you may notice that you are actually using some pieces of Scrum. The reason is that Kanban only defines three limitations and the rest is decided by you. So even thought Kanban doesn't say that you should have for example 15 minutes daily stand-up meeting every morning, you may well have it. Because it makes sense and is good for your team.

Kanban is based on three rules:

1. Visualize the work flow (by using Kanban board).
2. Limit the work-in-progress (WIP) for each working phase.
3. Measure the lead (cycle) time.

The first rule means two things: "Split the work into pieces, write each item on a card and put on the wall. Use named columns to illustrate where each item is in the work flow. [1]" So in practice everything that you do moves through the Kanban board as a card (or note or ticket). So everyone can see all the time what is happening in the project and what are the people doing at the moment. You can also see for example if there are plenty of tasks waiting for testing, which indicates that the someone in the team need to focus on that part a bit more.

The second rule tries to prevent a situation where tasks get stucked and don't get finished. In practice if the WIP is 2 and you already have two tasks, you cannot take a third one until you finish the existing ones. Another implication is that if some phase in the work process meets the WIP, then no new tickets can move to this phase until the team makes the existing tasks moving. So, all in all the goal is to get tickets moving as fast as possible through the Kanban board.

The third rule is there so that you can "optimize the process to make lead time as small and predictable as possible" [1]. Kanban is part of lean thinking and in lean thinking it is important to continuously improve the process.

Read more about Henrik Kniberg's thoughts about differences between Kanban and Scrum: http://www.crisp.se/henrik.kniberg/Kanban-vs-Scrum.pdf. [1]

A new Kanban/Scrum book published by Henrik Kniberg and Mattias Skarin: http://www.infoq.com/minibooks/kanban-scrum-minibook.

Experiences from the first project

The team found Kanban very useful and good process model to use. Like I said above, in the first week we tried to create such a Kanban board that is good for our team. We decided to have the following columns:

- Assigned: The task that is assigned to someone but this person (or pair or sub team) hasn't started doing it yet.
- Blocked: A special column between Assigned and Working. If the team is not able to proceed with some task due to external reasons, then a task can be moved into this column. An example of this was that we couldn't continue setting up the test server before the IT department did some actions on their side. Note! There is big risk that tasks get stuck into this column for too long time. You could decide that the blocked column should be cleared every day at least temporarily. For example every day you go and ask if the test server is ready. If it still isn't, then the task becomes blocked. Nevertheless, if this column starts to collect a lot of tasks, then there is probably something wrong with your process.
- Working: Working means designing, prototyping, writing tests, coding - anything that the developer does for getting the task finished.
- Code review: When the developer felt that the task is ready, she had to ask for code review. In the beginning only senior developers could do it but later on everyone who was willing to. The purpose of code review was to make sure that whatever we build, we build it with good way using object-oriented methods and Ruby on Rails style. This was also very important feedback for the developers about their work, and it also was a tool for adding communication between developers.
- In quality assurance (QA): Code review was our white-box testing tool, QA was more like black-box testing. Someone just had to make sure that the property really works like it should from the user's point of view. You may wonder that doesn't the term QA also cover code review? Yes, it does but we had these columns since we wanted to integrate the Kanban board with our ticketing system and the ticketing system happened to have the term "in QA" (read more about ticketing system below).
- Verified: When the tasks passed QA, it was moved to the Verified column.

Typically tickets moved from left to right but in case code review or QA didn't pass, then the ticket moved back to the Working phase.

All the working phases had a WIP limitation of two / person except the Assigned column. The reason for having two was practical. For example there may be tasks that are closely related to each others and it makes sense that the same developer does both of them. So WIP one would have been too few. On the other hand we never had a need to have three tasks running at the same time. In the Assigned phase there was however a limitation of only one. The reasoning for this was that you shouldn't reserve too many tasks for yourself before you actually have time to do them.

It may be interesting to note that typically WIP limitations are not defined per person. So instead of having e.g. WIP two for each developer in the Working column, we could have had for example WIP ten for the whole team. This would have made the tasks more like the property of the whole team instead of the property of individuals. Probably it would have also given us better possibilities to improve our process than the setup we used. So in the coming projects you should perhaps consider the team approach instead of individual approach. That is what you usually see in the Kanban board pictures.

The left-most column in our Kanban board contained the individuals or teams doing the tasks. In the beginning we tried several approaches but pretty soon ended up to the situation where each one's name is written in a post-it note and this note was located in the first column. If someone started to work with someone else, then the note was easily relocated so that everyone could see that these two people are now working as a pair. This was very flexible way to change pairs and show that in the board.

In our project we didn't measure our lead time. There was no time when we felt that we should. Instead we tried to keep the tasks so small and flexible that is easy to stop working with them if they take too long time. However, this is quite a risky approach since in software development you should be aware of your velocity. If you don't measure your lead time, you cannot know what the velocity is.

Ticketing system and Kanban

Kanban is good for seeing the overall picture but it is not good for details. In SF you will have a ticketing system that you can use for those details. It is probably enough to write only a title and id of the task to the Kanban card but all the details can be in the ticketing system.

In the ticketing system you can also more easily prioritize the tasks. The developers can then pick the most important tasks and then move them to the Kanban board. Also if your product backlog will at some stage contain over one hundred tickets, for practical reasons the Kanban board may not be a good place for all of them.

The ticketing system also contains some phases for each ticket. You should consider how these are used together with Kanban or maybe not at all. Well, at least when a task has gone through the whole board, it probably should be closed in the ticketing system as well.

Experiences from the first project

In the beginning the Kanban board also contained the product backlog but at some stage the amount of tasks increased heavily. This lead to a practical problem that the post-it notes just didn't stick in the board and they were difficult to arrange. That's why the team decided to keep the product backlog in the ticketing system. When a developer started a task, she created a new note to the board (either to Assigned or Working phase).

The original ticketing system contained too few process phases. We wanted to have basically the same ones as in the Kanban board. Luckily (or actually unluckily) we were able to modify the system so that we got there more phases. The problem was that they were useless. Even thought it was nice to have the Kanban steps also in the ticketing system, there was no actually benefit of that. And in practise the team members forgot to either use the steps in the ticketing system or in the board. So, it would have been completely fine if the ticketing system would have just contained the phases New, Assigned to someone and Closed (or similar). Basically for developers it is just enough to see whether someone is already working some task or is it still free. And the people who want to see the big picture of the current situation can see that - surprise - from the Kanban board.

Definition of Done and Kanban

Your team needs some kind of rules that tell when a ticket can move to the next stage in the Kanban board. This is called as Definition-of-Done (DoD). The rules help to prevent doing the same mistakes all over again and also make the process run smoothly. Again, it is the team that has to decide what rules are good for the team. It is although important that when you agree about the rules, everyone agrees. Also remember the important agile principle: change is a good thing. If you notice that your DoD needs to be changed, then change it. As long as you make sure that everyone in the team knows about the change and still agrees about it.

Experiences from the first project

We defined DoD for each main step in Kanban process. E.g. before you can move a task from Working to Code review, all the tests should pass, the UI code should be localized and the models (as model in model-view-controller pattern) should have decent validations. In Code review the DoD said that the senior makes sure that the code is written in object-oriented and rails way. And the DoD for QA was that someone in the team tests the task.

The DoD we defined in the beginning was changed only once. We added to the Working phase that the Rails seeds file (database population file) needs to have relevant data so that the other developers are able to use ready-made data in their own tasks. In the beginning the team forgot to look at DoD but the problem was fixed after the team discussed about it. Pretty fast everyone learned to use DoD in their work.

Daily meetings

Communication is a crucial thing if you want to have a successful project. One way to increase communication is to have a daily meeting where each person tells what she did yesterday, what she is going to do today and does she have any obstacles or blockers. In Scrum these meetings are called daily Scrums. Sometimes you may feel that others tell something that is not relevant for you but there certainly will be situations where it is very important to hear what others are doing. You may e.g. notice that for some reason you are doing the same thing as someone else, so maybe you could work together. Or if you tell about your problem, there may be someone who had a similar problem and solved it.

Good communication typically is such that it becomes an obvious thing when it works and you perhaps don't even recognize the benefits. You may for example feel that daily meetings are not needed. But if you stop having those meetings, eventually you will find out different kinds of problems that occur due to the lack of communication. On the other hand it is important to keep those meetings short. If you spend every morning one hour for meetings like this, then the drawback (lost development time) is bigger than the value. If the meeting seems to get stucked, you can always decide that the particular issue is discussed after the meeting by those who are concerned about it.

Experiences from the first project

In the very first Factory day our team decided that we will have a daily meeting every morning at 10 am. Each one should participate but if someone cannot make it, she should tell about it beforehand.

The meetings were held in Scrum style. We aimed to keep them in less than 15 minutes and everyone was standing during the meeting. We probably never exceeded the fifteen minutes, typically ten was enough. Standing may sound stupid but it assures two things. At first, everyone concentrates on the meeting better and secondly it makes sure that the meeting doesn't last too long.

In each meeting every team member told what she had done yesterday (or previous time when she was in the Factory), what she is going to do today and if there are any blockers. Typical blockers were that the team member had no next task (which was easy to solve) or she had some problem with her task. Another example of blocker was cases where someone was about to start doing a certain thing but someone else mentioned that another task would be more useful for the team.

Customer demos

It is a good idea to have a weekly customer demo where you represent all the new features for the customer. Remember that there are different kinds of customers. Some people are able to participate a lot, some may be there only every two weeks. So your team and the customer has to find your own way to work. On the other hand the more you are able to talk with the customer, the better it is. On the other hand the team also needs its own space and freedom to work.

Even though your customer would be very active and visit the Factory every day, the weekly customer demos may still be useful for several reasons. At first all developers may not necessarily have a clear picture about the whole. In the demo they see what the other developers have done. At second the demo is a good place to stop - also for the customer - and see what we actually have achieved and what is the most important thing to do next. If you just work day-by-day without any checkpoints, you may end up fine-tuning irrelevant things. And one positive thing in demos is that they may have a motivating factor also. In your daily work you may sometimes feel that you haven't actually done much. But when you see all those ready tickets that the whole team has done, then you can see that the team together has achieved a lot.

It is up to you how you want to have the demo. How open the discussion will be? Are the features commented right away or only after the presentation? Is only one person talking or maybe two or three? Who is using the computer? There are many ways and you can try different approaches to find the best way for you and your customer. But always make sure that there is someone making notes, maybe even two people. Otherwise you will forget all the important details.

Experiences from the first project

In the first week the team had a workshop where it discussed what features in the system are so called core features and what are nice to have. At the end of the first week the customer asked when those core features are ready. The team lead responded that by the next Wednesday, which became the date of the first customer demo.

In the first demo we presented all the tasks that we had completed, except those that were not so relevant from the customer's point of view (like setting up the test server). The customer gave us comments and proposed changes. At the end of the demo we asked the customer to define two or three main things that he would like to see in the next demo. This was extremely valuable question since in a demo you hear a lot of requests from the customer but all of them are not equally important. The question also forces the customer to think what really is important right now. For our team the answer to this question was our major guideline for the coming week. This was completed with all the notes that we made.

At the end of the first demo we agreed the date for the next one. We had about one week to work and then there was the second demo. The second demo was organized so that the team lead presented the features and the discussion was only after that. And again at the end we asked about the most important things to do. By accidentally the team found a weekly rhythm: customer demo once a week and a retrospective after the demo.

Retrospectives

After a customer demo it may be a good time to sit down for a while and have a retrospective, like in Scrum. Retrospective has a bit similar effect as the demo, it raises you up from the daily working and makes you look things from a higher perspective. Good thing is that you can hear how other team members feel and what kind of worries they may have. Together as a team you should discuss about these issues and find ways to improve your team work. Of course, retrospectives can be a good break from daily work. However, the main purpose is to improve the team work.

There are several ways how to have retros. Just use your imagination. The basic idea is to look back and discuss how the team and individuals have worked, and then look forwards in order to find possibilities to improve.

Experiences from the first project

In the first retro we had three questions: "what went well?", "what went not so well?" and "it would be good if...". Every team member was supposed to find at least five answers for each question. The answers were written to the post-it notes and placed to three tables (one for each question). Then the team started to go the notes through one by one and everyone was able to comment anytime. We found a lot of positive things and lot of things that we could do better or in a different way. One of the main things was that even thought we had started from zero, the team had learned a lot and was in the right track.

The first retro wasn't however the biggest success since it lasted over two hours. Or maybe it was good to have so long retro in the beginning but afterwards we tried to keep the retros a bit shorter. In some retros we used those same three categories but in some the thinking was more open. Also almost every time we used some different method for collecting ideas or discussing about them. All in all, the retros were useful for the team but it is also important to make sure that they don't become boring happenings where you actually wouldn't like to participate.

Utilizing the Software Factory facilities

Software Factory provides you with the latest facilities. The room is small but it contains basically almost everything that a software development team needs. However, all the fancy tools are useless unless your team finds a way to use them.

The smart board is an excellent tool for prototyping. The best thing is that you can save your drawing and open it afterwards to continue your work. Well, you can do the same with paper protos but with paper it is difficult to cut and paste pieces and erasing is slow. And while paper protos tend to become quite small, the smart board is good for showing others what you have done. Maybe you can for example use the smart board in some daily meeting and present some essential work for the whole team? Well, find your way to use it.

Another nice thing are the wall monitors that can be used for showing relevant live information for everyone. Probably the most important usage is to show the status of build in one screen. Green is good, red is bad. But again, find your own way to use them.

And of course there are the white boards that can be used for Kanban and ad hoc planning, whatever suits you best. There is also a projector, relevant especially in the customer demos.

Experiences from the first project

In the beginning our team didn't know how to utilize the smart board. But probably in the first retro it was mentioned that we could use it more. Then one member of the team just started to prototyping and little by little the others saw how good tool it is for prototyping.

The team had pretty clear idea that the wall monitors will be used to show continuous integration (CI) information. Unfortunately since we were the first team, we had to wait the IT department to make some installations before we were able to use the monitors and CI. But after we got the screen showing either green or red whether the build is broken or not, our build was less often broken than it was before.

In the beginning we had only one white board but we asked for another one. When we received it we were able to extend the Kanban board so that the other white board was dedicated only for Kanban. The other was used occasionally, for example writing notes in retros.

Other agile methods to be used

Agile software development of course has some other tools and methods to use, like test-driven development (TDD) and pair programming. Again it is up to team to decide what methods to use.

Experiences from the first project

Our team did quite heavily TDD. It was sort of strategic decision, although not very clear in the beginning. If we had not concentrated on learning TDD and writing tests, we would have been able to write new features faster in the beginning. However, we would have had worse code quality and certainly more problems at the end of the project. Our test coverage was above 80% and it was amazing to notice how few bugs we had in the code. Even though during the last weeks we did a lot of changes to the system, we never encountered the typical situation that when you fix some bug, something else gets broken. There was no need to stabilize the system since the good code quality had stabilized it already. Thanks to TDD and code reviews.

In the beginning our team used pair programming when the focus was on learning Rails. But then the team members felt that they need to work more individually in order to learn better. After that we did pair programming only occasionally but maybe the reason was that we used it in a wrong way in the beginning. What probably is important however is that we had the code reviews. One benefit in pair programming is that it shares knowledge about the code between developers. But since we had code reviews, we were able to do that without pair programming. We found our own way of working.

Research in the Factory

One aspect of Software Factory is research. While you work, there are video cameras and microphones recording what you do. And probably you have to fill in some questionnaires. All this may sound annoying, even scary but don't worry. You will soon start ignoring them. Also there are strict policies who can access the material and everything is done in ethical way.

There is also one aspect that you probably haven't even thought about. The Factory provides students with a very interesting possibility for making their own research: writing seminar papers, bachelor's or master's thesis. It is very difficult to find a similar laboratory environment that at the same time is so close to real life.

Experiences from the first project

For most of us in the team it was probably strange to step in to a room where everything was recorded from the very beginning. But quite soon happened the same what the participants in Big Brother program have said: you just forget all the cameras and microphones. (Yes, there are similarities between BB and SF but SF is missing the Brother.) We had to also fill in a short questionnaire every morning and evening but it was quick to answer to five questions. It just became part of our normal life.

In our project one of the participants collected material for his master's thesis by keeping a diary and interviewing some of the participants afterwards. Another student wrote his seminar paper based on findings in the Factory.

Conclusion

If you read the whole guide, you probably noticed the phrases "what fits to your team best" or "find your own way" several times. The guide has hopefully given you a lot of ideas about what might work and what not but at the end it is the team that has the responsibility and freedom about its own work. Each team has different kind of members, with different background, with different communication and IT skills. And probably even from completely different cultures. This guide has not even discussed about global Software Factory that will contain several factories all around the world. The reason is that while writing this guide, there was only the one factory in Helsinki. So when the factories will work together, there will certainly be a whole bunch of new things to be solved.

But don't worry too much in front. Just remember the two basic rules:

1. What is the most relevant thing to do next?
2. Start working with small pieces. Just get things moving, little by little.

At the end your project has succeeded or failed at some level, but only thing that actually matters is that you have learned a lot.