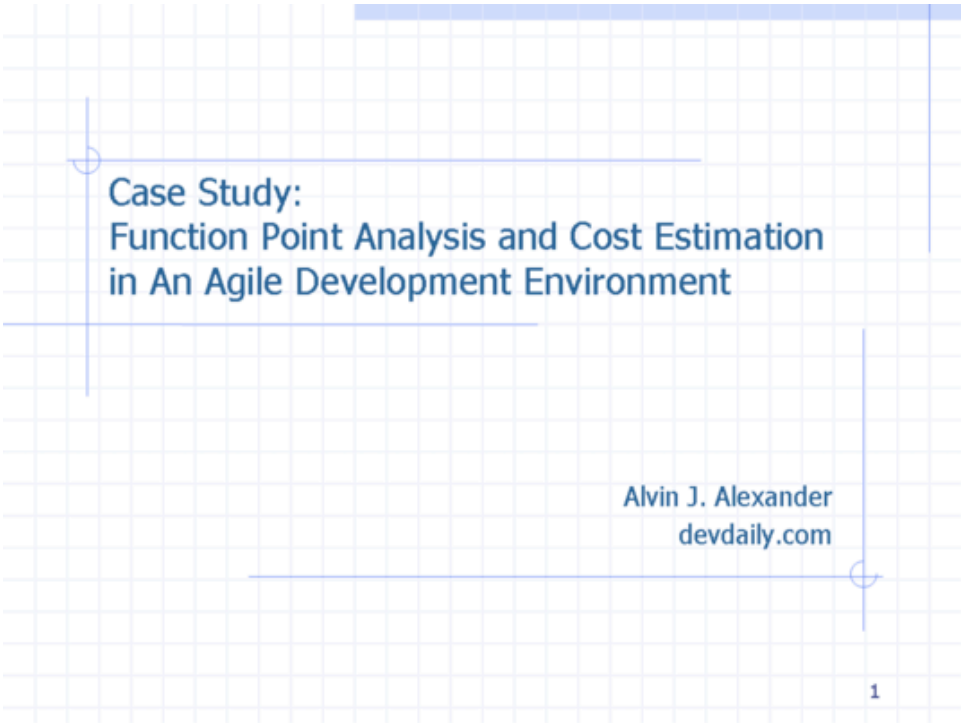
Book 3

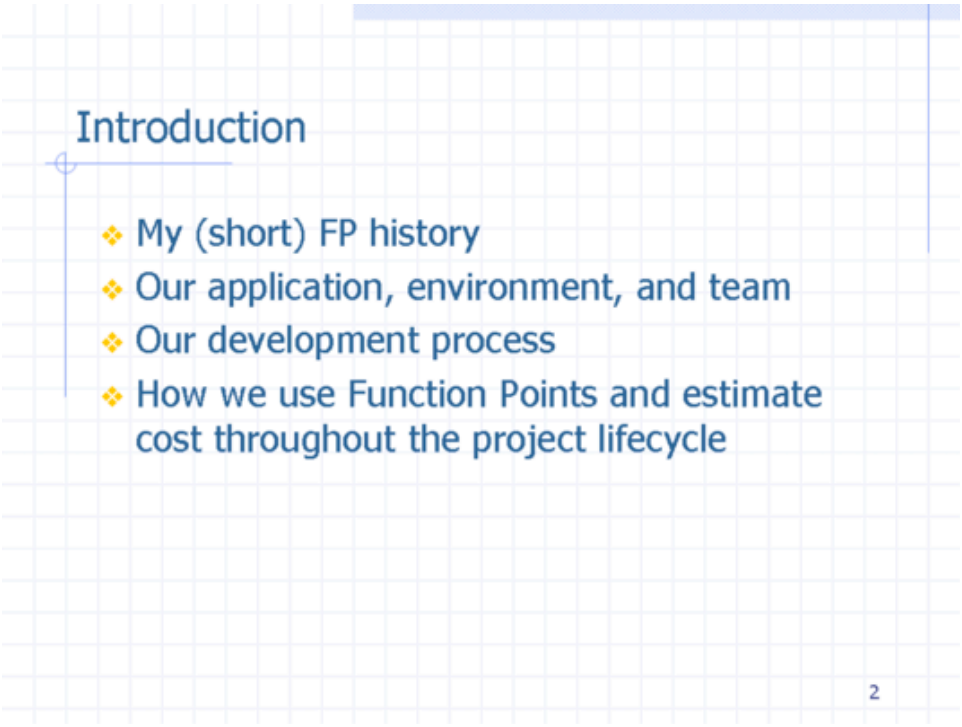
Cost Estimating in an Agile Development Environment

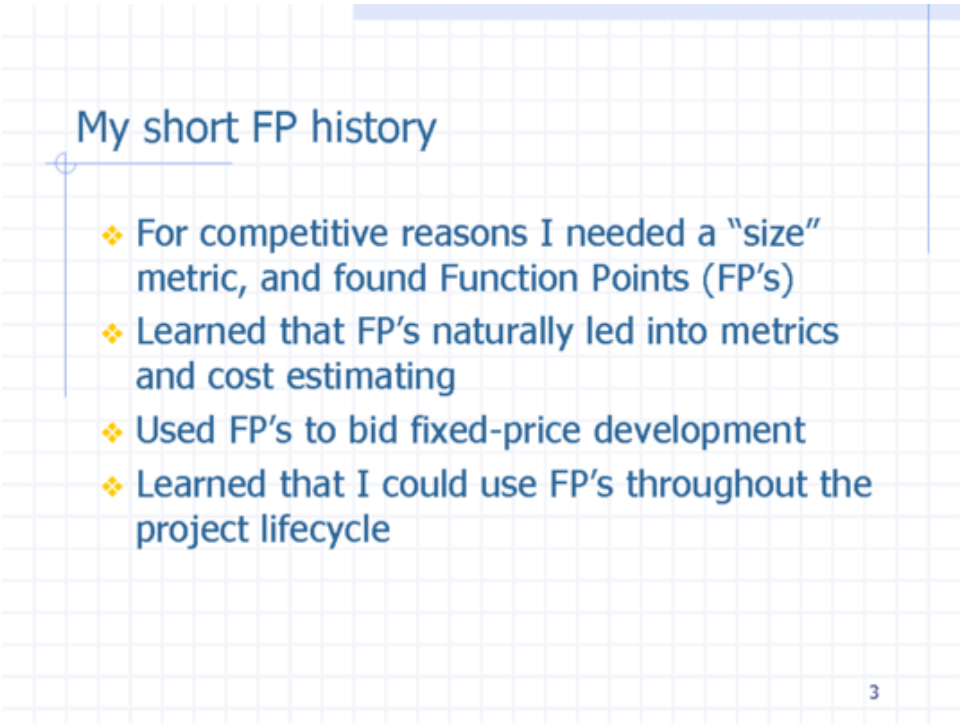
(early release)

In this third book I'll use the slides I gave at a speech several years ago to describe how you can use Function Point Analysis (FPA) techniques in an "agile" development environment. This talk was based on a multi-million dollar software project that I worked on for five years. We had as many as 12 developers on the project at one time (and as few as two initially).



I've already written about my FPA background in the first two books, so as I go through the next few slides I'll skip over those details and only discuss new content. The two slides shown here cover that history, so I'll skip them …

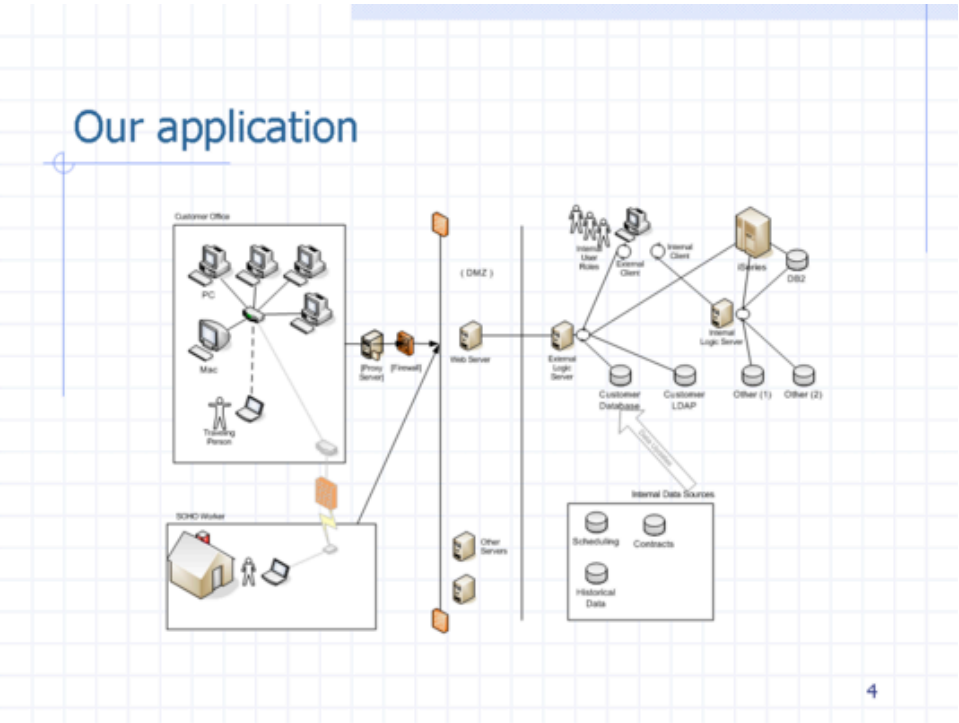




Back in 2002 my company was hired to build a fairly large corporate software application. The application was for one of the largest magazine printing companies in the United States. The application itself was very complex, but it was made even more complicated by the desire to give customers access to almost the exact same software that employees would be using.

The goal at the time was to be a little like Dell, which let you build and order a completely custom computer online. They wanted to more or less do the same thing for their clients, i.e., to let them build their magazine issue, place the ads and content, figure out how to package and mail the issue to their customers, and determine the cost of everything on the fly.

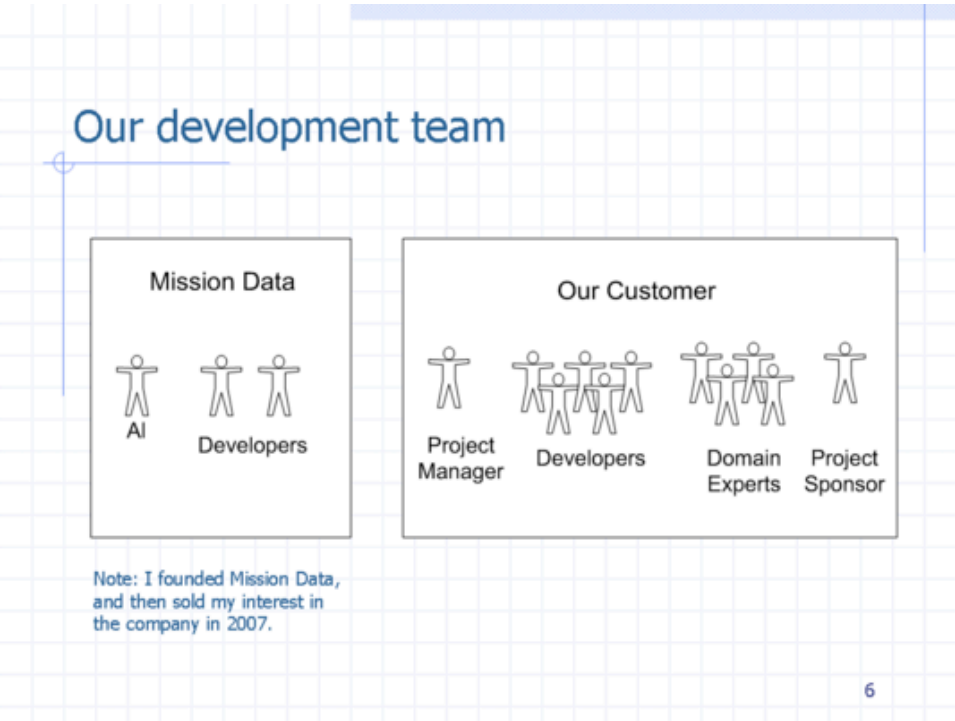
This doesn't sound too bad, just build a web application, right? Unfortunately, no. Remember that this was started in the year 2002, so technology isn't anywhere near where it is today. We also had to do create some very complicated graphical software, basically letting users create and lay out their magazine(s) from scratch in something of a WYSIWYG environment. We decided to write the application as a Java Swing (GUI) application, with a custom client/server communication layer.



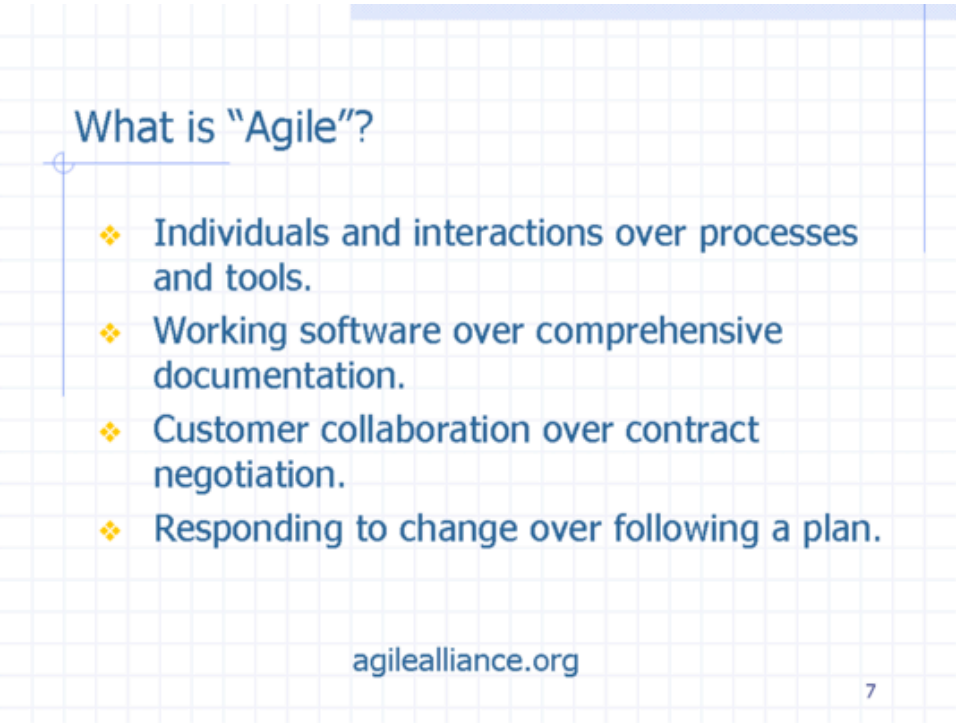
To continue to raise the degree of difficulty on this project, we agreed to train and mentor our client's programmers as we created the application. Our hope was that my company would write 100% of Phase 1 of the project, 75% of Phase 2, 50% of Phase 3, and 25% of Phase 4 as their programmers were brought up to speed.

In part because we had never worked with this company before, and in part because they had a distrust of consulting firms, they also required us to "bid" on each phase of the project in a fixed-price manner. (Actually, that isn't entirely true. What really happened was that Phase 1 involved a lot of learning by both parties about how to write software together. After Phase 1 our "Project Owner" asked us to bid on the projects in a fixed-price manner until we all felt like we were getting a fair deal again (and I agreed with his assessment).)

Our development team consisted of me meeting constantly with our client to write "requirements specification" documents to try to create a backlog of work for our programmers. Once we began working on each project in a fixed-price manner I began writing very detailed requirements specifications, and then later when we went back to Time & Materials (T&M) projects, I began writing much lighter documents, and over time I started having some of my developers meet with the client to write the documentation themselves.

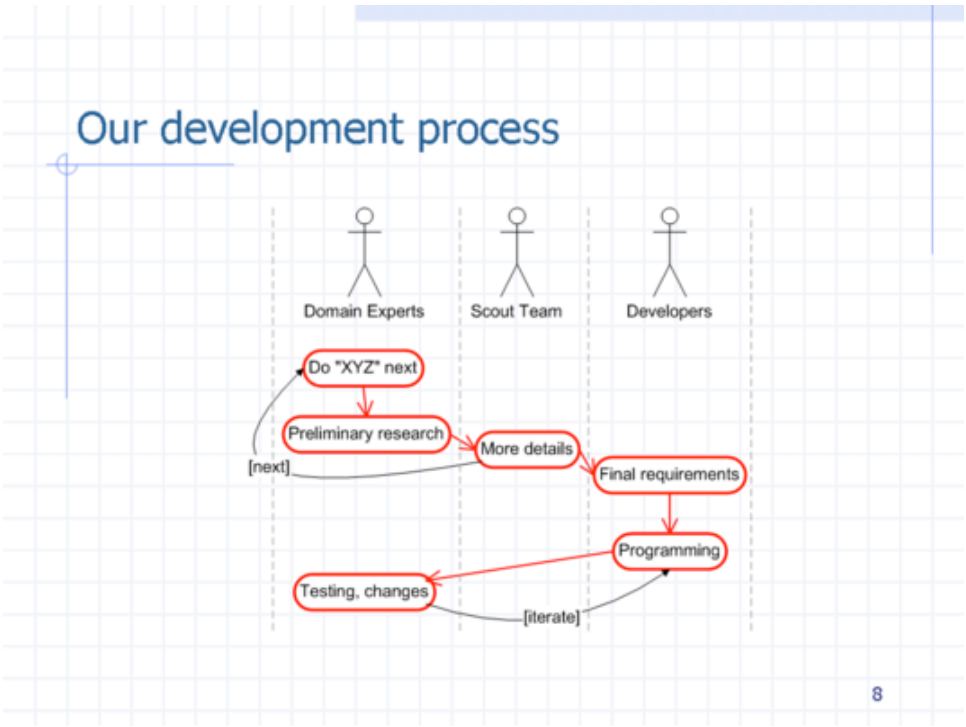


By now most people have their own ideas of what “agile” means, so I’ll just share these bullet points. I’ll also say that we were not very “agile” during the fixed-price projects, but became much more agile as the project went along.



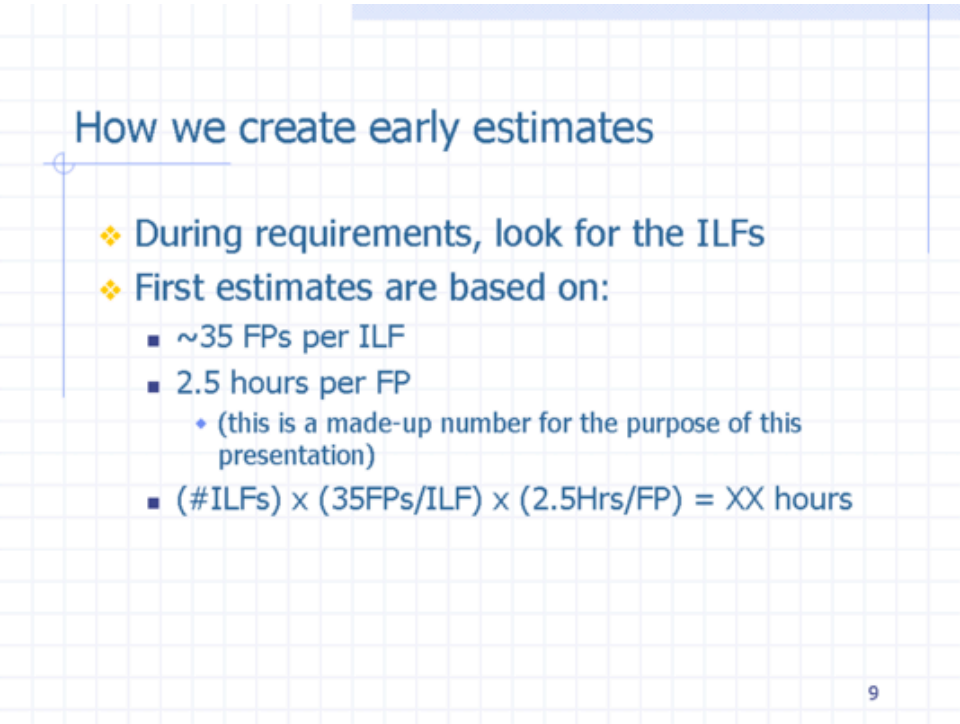
Once we switched back to T&M projects and attempted to become as agile as possible, our development process looked like this slide. I managed the project along with my client's project manager, and together we were essentially the "Scout Team" shown in this figure. We would figure out which parts of the project should be worked on next, in conjunction with my client's domain experts.

Once we agreed on the elements of the next phase(s), we would start doing research on what the domain experts wanted/needed. We'd take a first stab at gathering those requirements and putting a design together. We'd work on several tasks like this, and then when one of my developers freed up from their programming work, they would take over the process of writing very lightweight requirements for that effort -- much more "user stories" than "use cases." Once everyone agreed that they were all talking about the same thing, we would "bid" on this effort, get approval, and begin programming and testing.

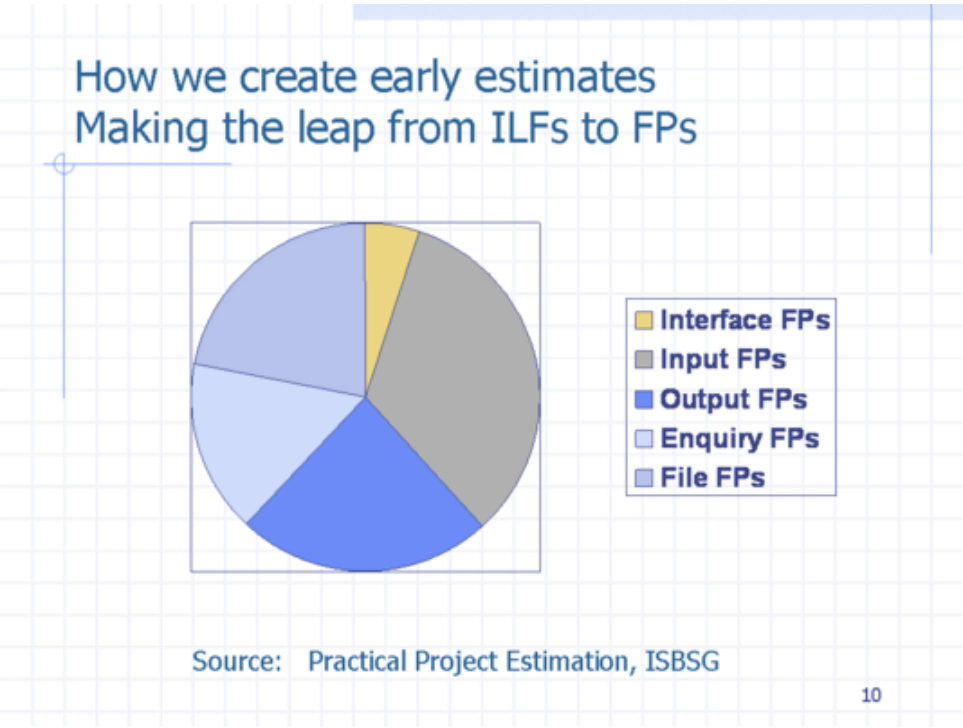


Each task we tackled like this usually consisted of somewhere between 300 to 1,000 man-hours of development time. I continued to have my developers create their own estimates, which included time for both (a) programming and (b) user acceptance testing. Actually, what we really did was use WBS to estimate the programming time, and then tack on an additional 1/3 of that amount for user acceptance testing. This was usually the norm, though I did notice that it varied a bit depending on who was doing the programming. (I often worked as the first line of acceptance testing, and would report the initial bugs. Once I couldn't find anything obviously wrong with the new code I turned it over to my customer's testers.)

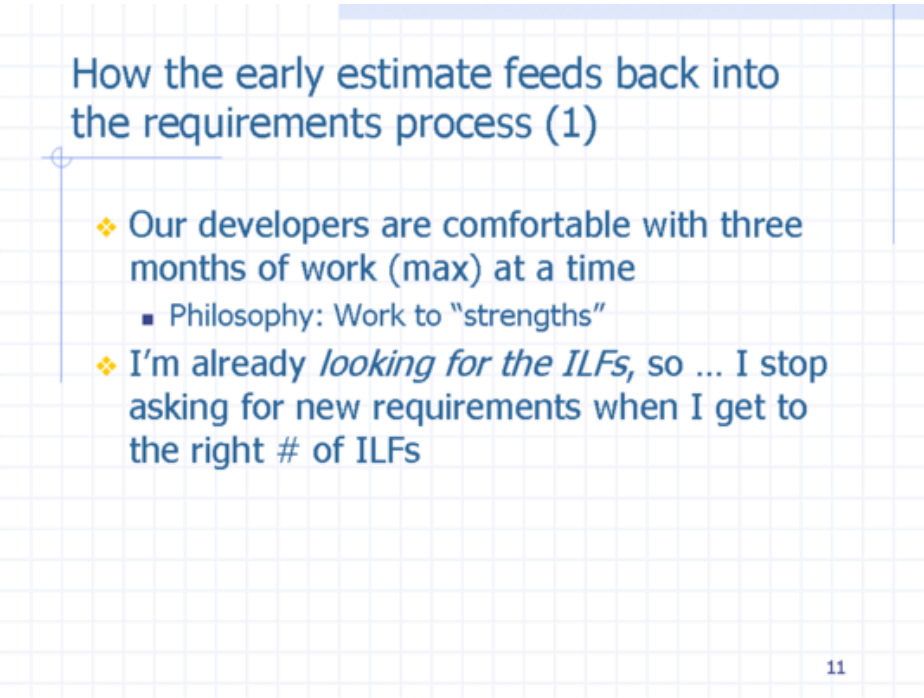
While I asked my developers for their estimates using WBS, very early on in the process I would start to create my own "back of the envelope" estimates using FPA. The way I do this is shown on this slide: I'd find the ILFs that we would be working with during this task; multiply them by 35 to approximate the number of FPs in the task; then multiply that by 2.5 Hours/FP to get the man-hours.



This magic "35" number comes from ISBSG historical data. It turns out that if you count Function points on a *lot* of projects and then look at how the data breaks down when the projects are completed, on average in most "normal" projects there are 35 FPs/ILF. Of course this isn't always true, but for most average/normal business applications, it is a very close approximation.

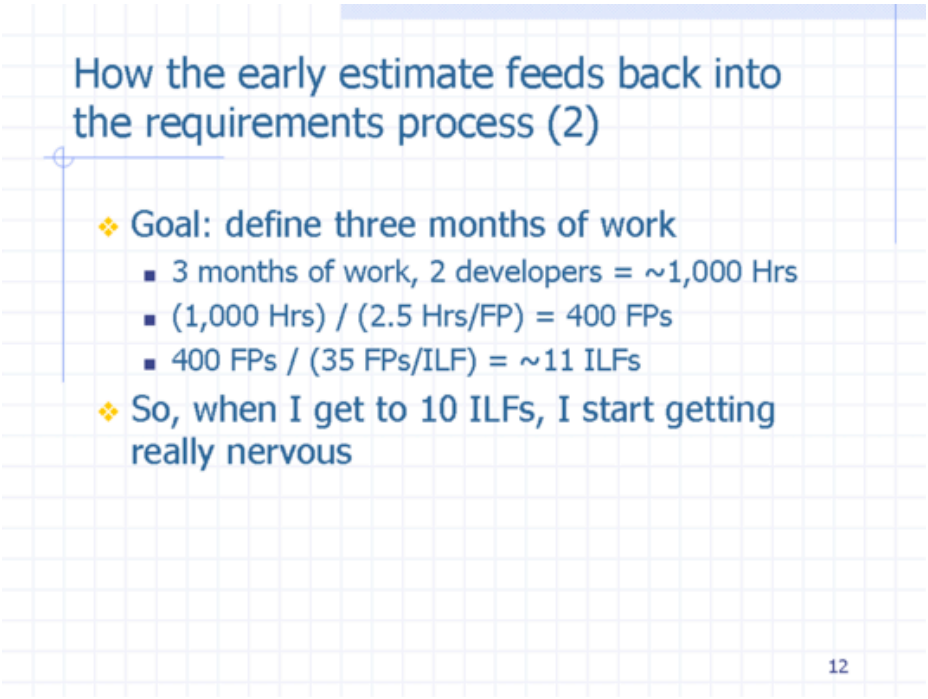


This approach created a feedback loop for the requirements process of our project. After working with our development team for a while, we learned that most of them were comfortable thinking about approximately three months of work at a time. In an effort to work to this comfort zone, we tried to create about this much work in each new task. The way I knew we were getting close to three months of work was by keeping an eye on the ILFs …



The math works out as shown in the following slide. Since our goal was to define three man-months of work at a time, and since most tasks had two developers, we aimed for about 1,000 man-hours of work per task.

1,000 hours of work divided by 2.5 Hours/FP meant that I was looking for about 400 FPs per task. Because on average there are about 35 FPs/ILF, that meant that I should try to manage each task when it got to around 10 ILFs. Of course each task didn't break up neatly like this, so when one complete task became too large, we tried to find ways of breaking that large task down into more manageable sub-tasks.

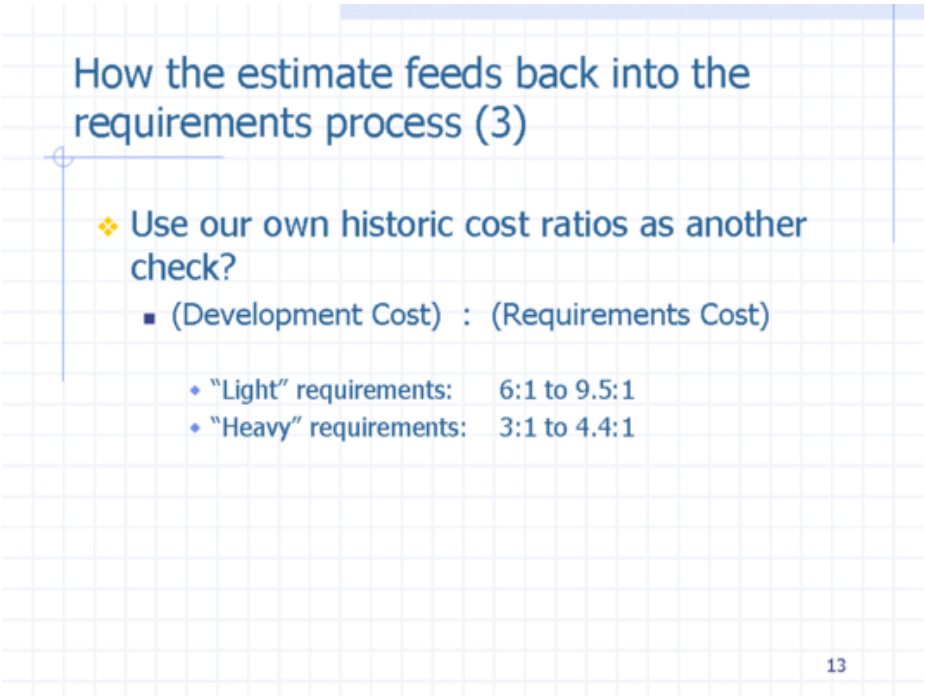


Another thing we learned as time went on is that there were relationships between (a) how long it took us to gather requirements for each task and (b) how long it took to do the programming work for each phase.

In the beginning of the project, when our requirements documents were very long and detailed, the programming work took as little as 3x the time that was required to create the requirements, and as much as 4.4x (though the 4.4x ratio is really a bit of an outlier, and the average was much closer to 3x).

Later on -- when our client trusted us and in fact new all of our developers by name -- we wrote much lighter requirements documents (much like "User Stories"), and the ratio of programming work to requirements work ranged from 6:1 to as much as 9.5:1.

These ratios gave us another way to check our programming time/cost estimates. If the requirements phase took 100 man-hours, and our programming estimate was something like 100 hours (very low) or 2,000 hours (very high), this gave us another check to ask, "Is something wrong here?"

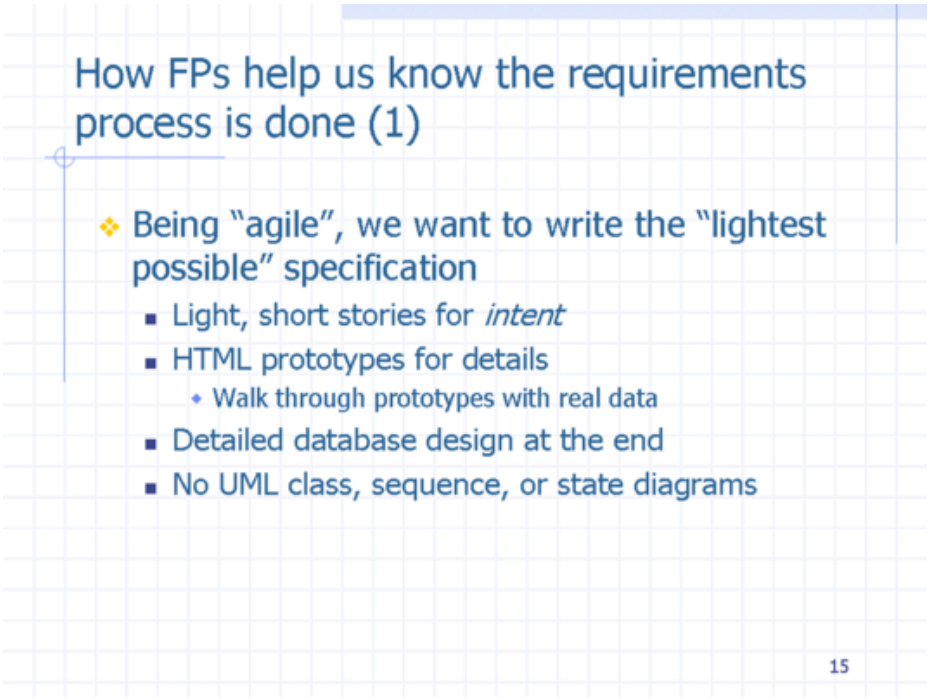


FPA also gave me a way to know when the requirements gathering process for each task was complete. I'll get into that in just a moment, but for now it's important to mention what we thought "agile" meant. As the next slide shows, we wrote short "User Stories" rather than Use Cases, and within those documents we focused on *intent* over actual implementation.

Most prototypes that were created were hand-drawn or created with HTML, and we used those prototypes to perform data walk-throughs as necessary.

I know I thought about database design constantly during our meetings (and my developers probably also did), but as a rule we didn't create formal schemas until the very end of the requirements phase. Each table and field had to be approved by our client, so as much as possible, this had to happen in the requirements phase. (Of course there might always be small changes later.)

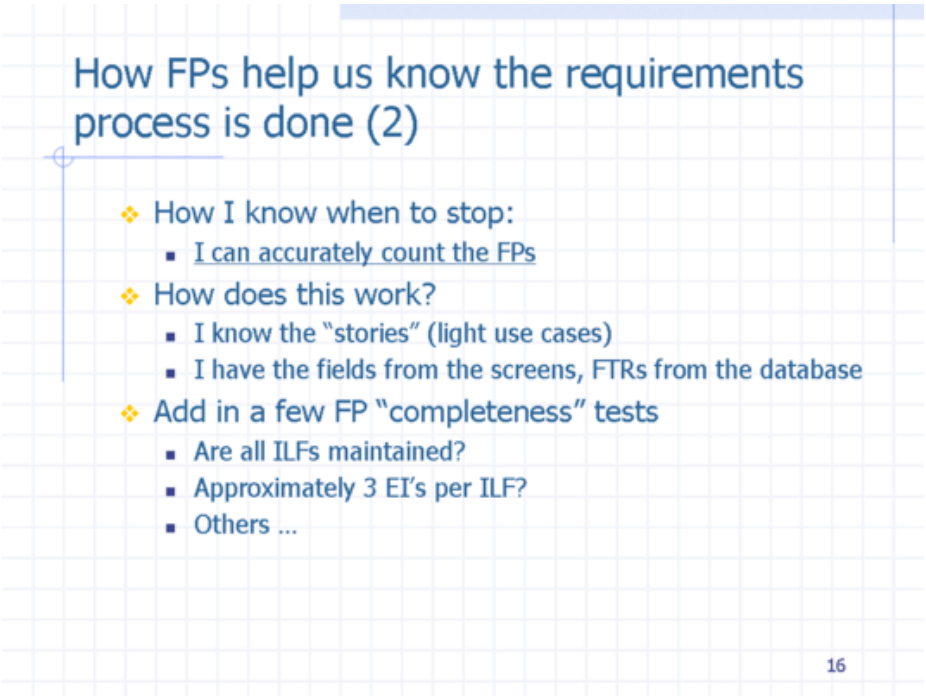
Finally, as you might guess, there were no UML diagrams anywhere in the process. Actually, I take that back, slightly: I am famous for drawing stick-figure diagrams showing "actors" and "use cases," so I'm sure there were a few of those on whiteboards from time to time.



A nice benefit of FPA in the process of writing our lightweight requirements documents is that I knew when the requirements were complete when I could accurately count the FPs. Because I knew the User Stories, knew what the screen prototypes implied, and further knew the database details, I could either (a) accurately count the FPs for each task, or (b) I knew something was missing.

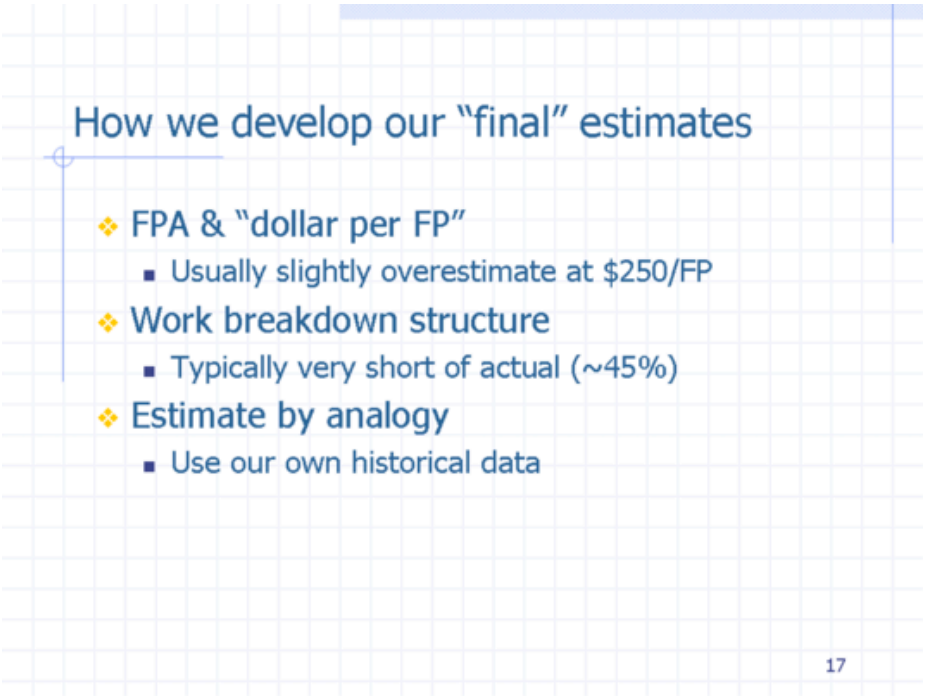
Another great FPA test you can perform is to make sure all the ILFs -- and each new field in the ILFs -- are "maintained", i.e., that there are add, edit, and delete processes for each field. As you know from the first book, an *ILF* is an Internal Logical File, and by definition, this means that this "file" (typically a database table) is maintained by *this* application. Therefore, if we had a new database table or fields within tables that weren't defined as being maintained within our requirements documents, I can tell that something is wrong.

Furthermore, in an "average" software application there are typically three EIs per ILF, so that gives me another check. (I once amazed a developer on another project by briefly looking at his requirements specification for the first time, and within minutes I asked, "Where are the XYZ reports?" He looked at me somewhat stunned, and then told me that the client didn't want them. I used the 3 EIs/ILF ratio to quickly see that functionality was missing in his spec.)



Our "final" estimates/bids were based on a combination of the techniques I just discussed, including (a) FPA, (b) WBS, and (c) the average ratio of requirements hours to programming hours, but in general I trusted my FPA techniques and 2.5 Hours/FP over the other techniques.

As shown in the slide, our WBS estimates were typically very far short of the actual required time. Once you know this, there's nothing wrong with it; I could just ask my developers for their estimates, and then multiple them by about 2.2. In fact, I learned that if their estimate wasn't significantly lower than my FPA estimate, this was another indicator that we probably needed to talk about something.



Of course as time goes on, things change, and those changes will affect your estimates. Anything like a new Project Sponsor, Project Manager (PM), new Domain Experts, new programmers, new technologies, etc., can all affect your project speed. I was amazed one time when a project switched from one PM to another and the project suddenly took on an entirely different speed, much slower at first, and then a little faster as time went on.

It's also important to know that there are many things you can't estimate with FPA techniques, and I tried to highlight many of those items in the first book. Those items include bug fixes, conversions, creativity, and all of the VAFs/GSCs discussed in the first book.

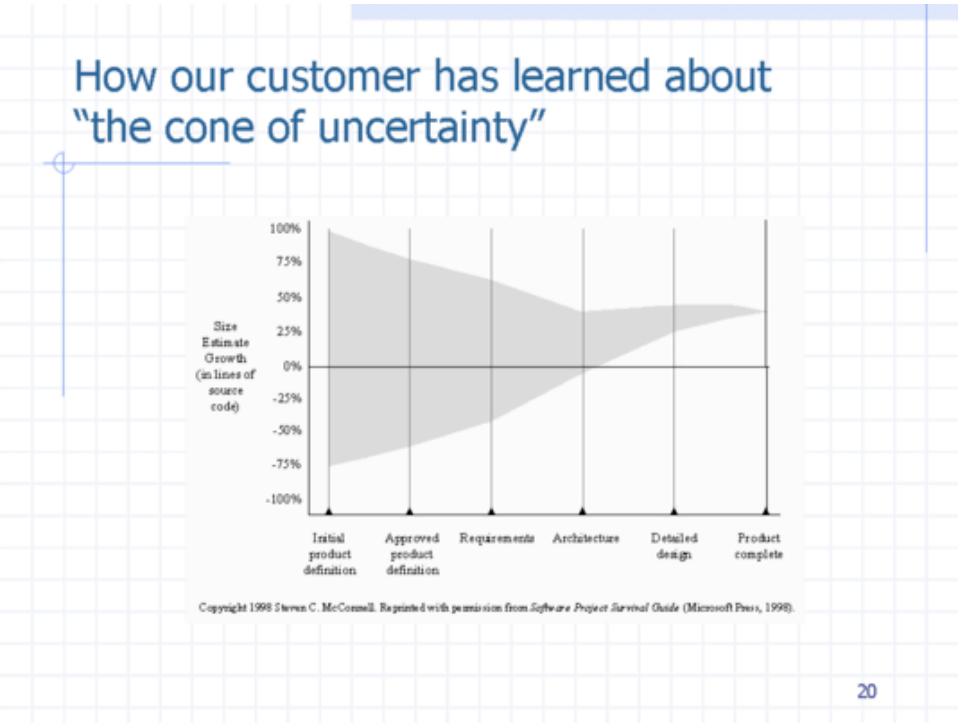
As a few final points about our agile process, we used the Twiki wiki for a lot of our developer documentation, and our developers liked to use the XPTracker plugin with it. (I don't know if this plugin is available any longer.)

I also found that our developers used to say, "It's ready for testing," at about 50-60% of the project budget. Because our client was having problems keeping up with testing (due to a lack of manpower/man-hours), I typically did the first round of testing, and once I worked through all of the obvious bugs I turned the software over to the client for testing. (FWIW, I later learned that they appreciated me turning over a more solid product to them, as that showed a respect for their time.)

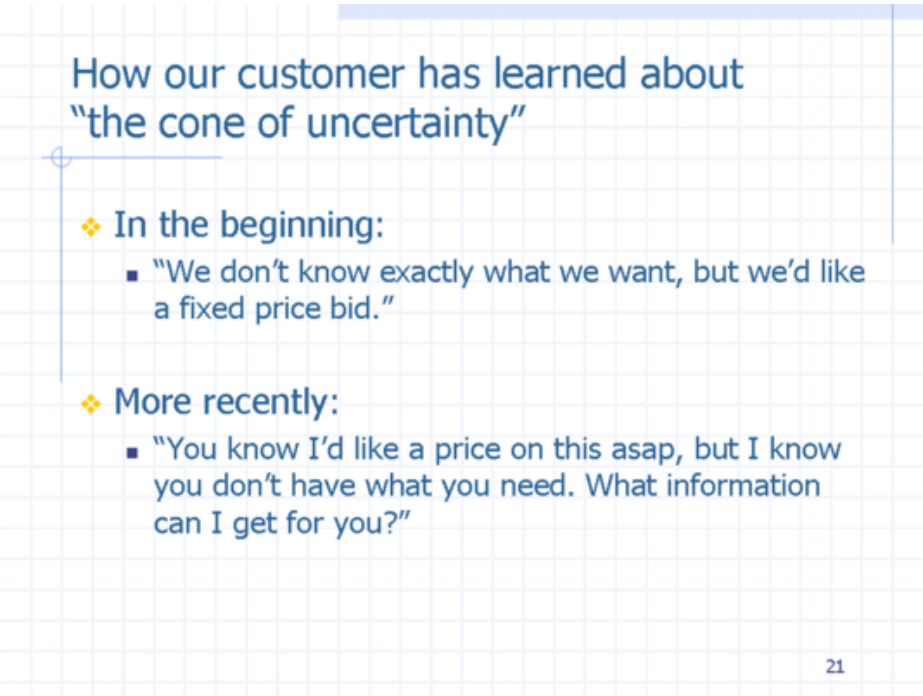
As time went on, we also found that there was typically a 3:1 ratio between development time and user acceptance testing time; not in man-hours, but in calendar time. For instance, if a project task took nine weeks to develop, it typically took three weeks for acceptance testing. This didn't seem to matter if there was one developer on the task or four developers; the ratio seemed constant.

An interesting side effect of working on a project with one client for many years is that after some period of time they learn about Steve McConnell's "Cone of Uncertainty." The *Cone of Uncertainty* simply states that until a software project is completed, you'll never know the exact project cost, and the farther you are away from having the software completed, the more uncertain you are about the final cost.

It also states that almost every software project ever defined has always grown from (a) what the customer thought they wanted into (b) what they really wanted/needed.

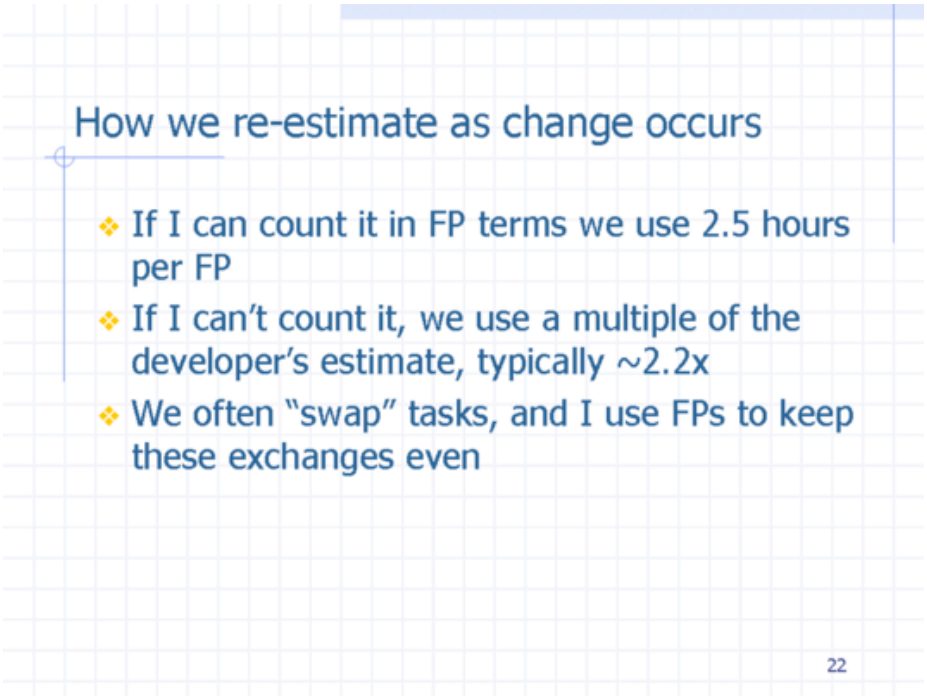


In our case we were fortunate that our client grew with us in this process. As the next slide shows, they initially thought that they wanted fixed-price projects, and couldn't understand why they were so hard to bid, but as time went on they learned that we needed more information to make any soft of responsible bid/estimate.



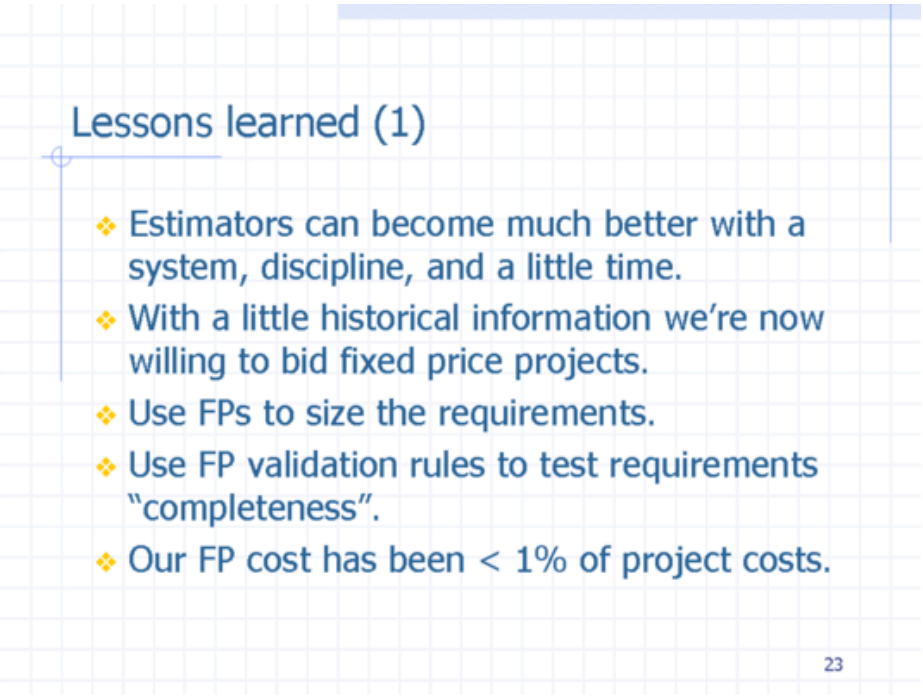
In software projects there is always change, and when there's change, there's also a need to re-estimate the cost/effort. When this happens on our project, I use the same techniques to estimate the change. If I can count the FPs in the change, I estimate 2.5 Hours/FP. If for some reason I can't count the FPs, or FPA doesn't make sense for the task, we'd use WBS and then I'd multiply the developer estimate by 2.2.

Finally, from time to time we would end up "trading" one task for another, and whenever we did this I'd use an FP count to estimate the difference in each trade. Over time we'd try to balance out the tasks we traded.

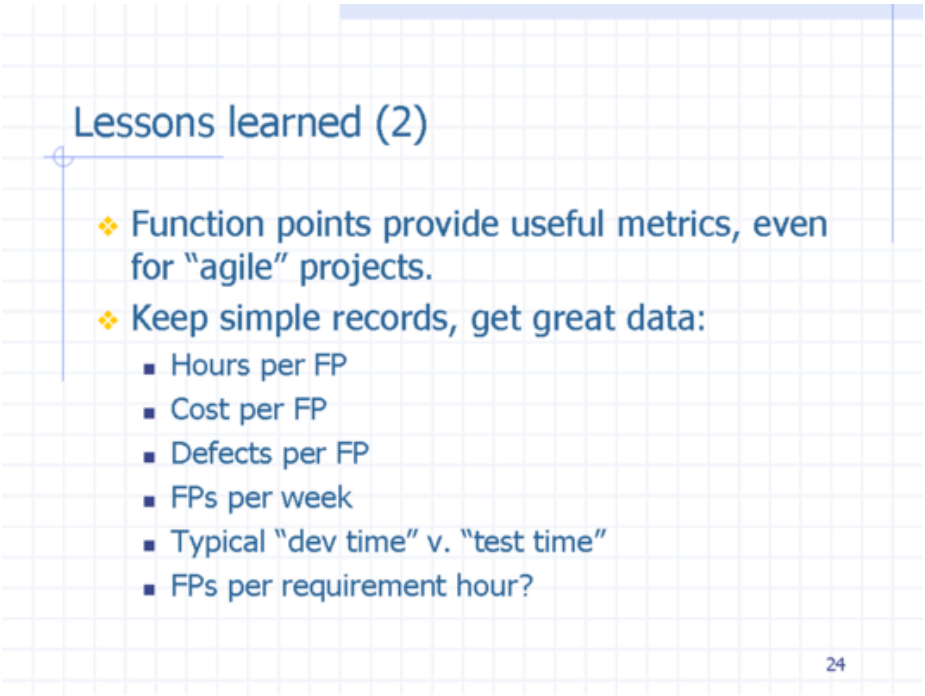


In conclusion, the following three slides show our "Lessons Learned" from using FPA in an agile development environment.

I've written about each bullet point on this slide, except for the last bullet point, which I covered in Book 1; the cost to use FPA is very small, typically less than 1% of your overall project cost. Once you get good at counting FPs, you can do so extremely fast.



I haven't discussed it much, but FPA techniques give you all sorts of good metrics that can be helpful on a project. The metrics shown under the second bullet point are all extremely valuable at different times on a project. ("Defects (bugs) per FP" was a real surprise to me, being very constant per development team.)



Finally, I've learned that it's helpful to talk to new clients about the "Cone of Uncertainty" early on in the software process. If they've never worked on a software project before they might not believe you on Day 1, but if you have a chance to work together for a while they'll soon learn about and understand the meaning of that diagram. If you use FPA, they'll also understand that when you can't count the FPs, you can't give them a responsible time and cost estimate.