How I Estimate Software Development Projects

1) how to determine your application size
2) how i estimate software projects
3) cost estimating in an agile environment

Alvin J. Alexander

How I Estimate Software Development Projects

http://alvinalexander.com

First edition, published October, 2014

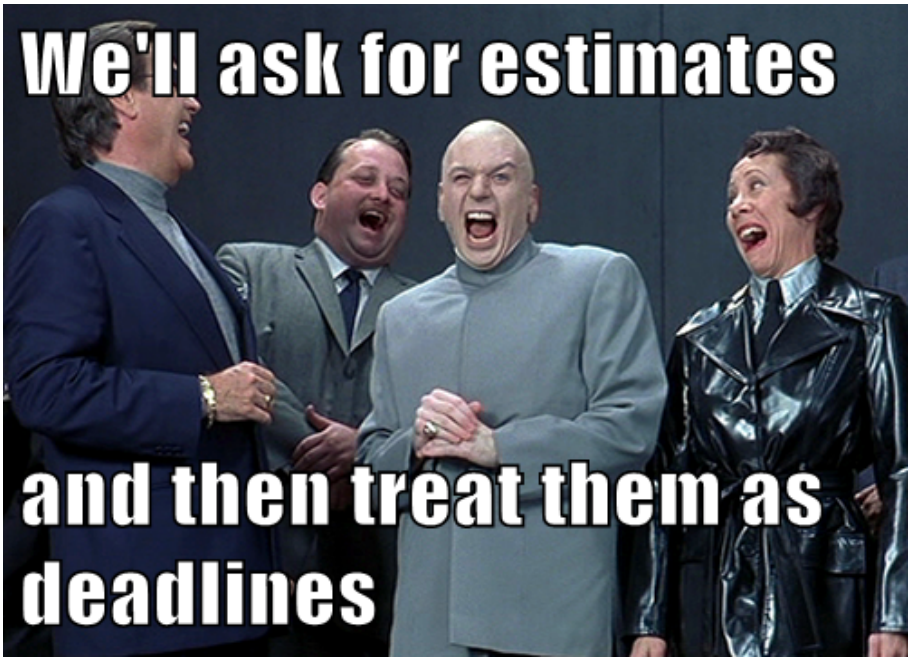
Other books by Alvin Alexander:

- Scala Cookbook (http://amzn.com/1449339611)
- A Survival Guide for New Consultants (http://amzn.com/1495348784)
- How I Sold My Business: A Personal Diary (http://amzn.com/1495427099)
- Play Framework Recipes (PDF eBook)

Table of Contents

Dedication

I dedicate this book to whoever created this image:



I don't know the original source of this image, but it summarizes one of the things that programmers fear most about cost estimating. And since I write about being forced to create "fixed price bids" several times in this book, this image somehow seems appropriate.

Preface

Dateline 2002. We're in the United States in the year following the September 11, 2001 terrorist attacks, and the economy is sluggish. In my case, I own a small software consulting firm in Louisville, Kentucky, the 16th largest city in the United States. Our business is profitable, but times are tough, and two of my business partners want to lay off some of our developers. Until this time I'd never had employees "on the bench," but now we have three programmers with no billable time, and the partners want to lay them off.

To date owning my own consulting firm has been pretty easy, but now we face a difficult decision. We're at a crossroad: if we're willing to bid projects on a fixed-price basis -- and we can do so successfully -- we can keep all fifteen of our employees and maybe even grow the company. If we're not willing to do that -- or can't do that -- we'll have to lay off several of our programmers, and there's no telling where that road leads.

Our programmers are all good and I like them all, so I don't see laying them off as an alternative. As much as I don't like it, we're going to start bidding on fixed-price projects.

To get our feet wet, we bid on a small, fixed-price "pilot project." Our developers estimate $24,000 for the project, and the client agrees to the price. We complete the project, but in the end it actually costs us $36,000, a difference of 50%. If we keep going like this, not only will we have to lay off several developers, we'll also have to shut down the business. We have to learn how to estimate better -- much better -- if we're going to survive.

The rest of the story unfolds quickly. I dig into the history of estimating the time and cost of software development projects, and after reading about several techniques I don't like very much, I learn a technology known as *Function Point Analysis*, or FPA. The technology is over 20 years old, but to my great surprise, it's still the best thing going.

With my business in jeopardy I buy several books on FPA and read them as fast as I can. There are a few points I can't understand from the books alone, so I find the next "Introduction to FPA" course I can find, which is in Virginia. I fly there, learn from an experienced practitioner, and even learn a few advanced "tricks of the trade." Once I understand FPA, I use it to bid fixed-price projects.

The technology is a success. Our development teams bring projects in on time and within budget, and we gain the confidence of clients. Having gained the trust of our clients, we're then able to go back to working on projects on a time-and-materials (T&M) basis, which in my mind are better for everyone involved. (In my opinion, with fixed-price projects there is always a "winner" and a "loser," and that's not good if you want healthy, long-term relationships.) But even in the T&M world clients want our "best estimate" before we begin each project, so as the years go on, I continue to use FPA to estimate the size and cost of our projects, and it continues to work well.

Overview

In retrospect I learned how to estimate the time and cost of software development projects in three lessons, so this book contains those lessons in the order I learned them. The three "lessons" are based on tutorials I've written and presentations I've given, and they correspond to how I estimate the time and cost of software development projects using *Function Point Analysis* (FPA) as a central tool. Within this document I'll refer to them as Lesson 1, Lesson 2, and Lesson 3, and they are titled:

1. How to Determine Your Application Size Using Function Point Analysis
2. How I Estimate Software Projects Using FPA
3. Cost Estimating in an Agile Development Environment

A very short summary of the three lessons goes something like this:

- FPA is a technology you can use to determine the *size* of software applications.
- In a manner similar to how the size of a house is measured in square feet, an application has a *functional size* that can be stated in Function Points (FPs).
- Once you can measure the size of your applications in Function Points, you can use all sorts of metrics to help you estimate the time and cost of future development work.
- The most important metric is probably your team speed (or productivity rate), which is measured in Hours/FP (such as "3.0 Hours/FP").
- Once you're comfortable with this process you can take the next steps and use advanced FPA techniques for other purposes, such as providing "back of the envelope" estimates very early in the software lifecycle when you're meeting with a client or colleague to discuss a new idea for an application.

While FPA works very well for most business applications, there are also areas where it doesn't work as well, and those limitations are also discussed.

Who this book is for

This book is written for anyone who can benefit from learning how to estimate the size and cost of software development projects more accurately. This includes:

- Business analysts
- Project managers and CIOs, who need to understand the size and the reasonable time/cost to build a software application

- Programmers who want to learn how to estimate better (or at least learn a different way to estimate)

- Owners of software consulting firms (like my own), who want to be able to give their clients great estimates all the way through the project lifecycle

The Benefits of Function Point Analysis

From my experience, I've found that with a small amount of experience, understanding the functional size of your applications leads to a goldmine of other information that will help you run a successful software development business, including:

1. The ability to accurately estimate:
    a. Project cost
    b. Project duration
    c. Project staffing size
2. An understanding of other important metrics, such as:
    a. Hours/FP (what I refer to as project speed or velocity)
    b. Cost per FP
    c. Project defect rate
    d. The productivity benefits of using new or different tools

As an example of what FPA can do for you, my software consulting firm was able to bid on projects on a fixed-price basis (with confidence), something we could never do before. This gave us a significant competitive advantage against our competition.

Introduction (FAQ)

Given the background of the previous sections, let's dig into some FPA definitions and details. I'll begin with a frequently-asked question (FAQ) format, because I think that's a great way to introduce FPA.

Q: What is Function Point Analysis?

Function Point Analysis, or FPA, is an ISO standard process for determining the *functional size* of a software application. Just like the size of a house is measured in square feet, a software application also has a "size," which can be measured in "Function Points".

Every software application -- whether it's Microsoft Word or Excel, web applications like Facebook or Twitter, or a mobile app -- has a functional size that can be measured in Function Points, or FPs. If someone trained in the practice of FPA takes the time and has the necessary information, they can determine the "functional size" of any software application.

Q: Okay, so what is a Function Point?

A *Function Point*, or FP, is a standard unit of measure that is used to state the size of a software application. A person trained and certified in the process of counting FPs is called a *Certified Function Point Specialist*, or CFPS, and they can measure the size of software applications in FP units.

When a CFPS measures a software application, the end result might have them stating, "This application has a size of 1,000 FPs." That's great, but what does it *mean*?

That question is the purpose of Lesson 1, but in short, Function Points have the following characteristics:

- A Function Point is a standard way to measure the functional size of an application
- A Function Point is measured from the perspective of a user of the application
- A Function Point is independent of the technology used to develop the application
- It doesn't cost much to count Function Points
- Counting Function Points is a repeatable process

Let's expand on those bullet points.

First, Function Points are a standard way of measuring the functional size of an application. In fact, FPA became an ISO standard in 2003. There is also a large user group named the *International Function Points User Group*, or IFPUG, that is formed around the process of determining the size of software applications. (See ifpug.org for more information.)

Function Points are measured from a user's perspective. FPs measured by (a) a person looking at all the windows/screens of an application, and (b) how the application stores its data, and "counting" the functionality that they see. (To be clear, FPs have nothing to do with source lines of code.)

Because of this approach, FPA is also independent of the technology used to develop the software application. The application can be written using languages like Java, PHP, Ruby, Scala, JavaScript, and databases like Oracle, MySQL, MongoDB, and Cassandra, etc. The technology used doesn't matter to FPA; the only thing that matters is what a user of the application can see and do with the application.

FPA is also a low cost technology. It typically takes less than 1% of a development budget to have a CFPS or other FPA specialist count the size of your applications. So if an application ends up costing about $250,000, it should cost only about $2,500 to have a CFPS estimate its size beforehand (or count its size after it has been developed).

Finally, FPA is repeatable. If two CFPS specialists count the size of a given application, their counts should be within 10% of each other, and are typically much closer than that. FPA has many rules that govern how an application should be counted, and those rules are stated in a thick book titled, *Counting Practices Manual*, or CPM. These rules help guarantee the consistency of Function Point counts.

Q: When is FPA useful?

FPA is useful any time you need to know the size of a software application. Or, more likely, what you really want to know is an accurate estimate of the time/cost to develop the application, and using FPA to determine the functional size of an application helps you get that estimate.

Personally, I have used FPA to:

- Bid on fixed-price and T&M projects.
- Provide "back of the envelope" cost estimates to clients and prospects at lunch and dinner meetings.
- Verify the accuracy and completeness of software requirements specifications. Advanced FPA techniques provide checks and balances to help you find missing functionality in requirements documents. (See Lesson 3 for more information.)
- They work well with Use Cases and User Stories. (In fact, while some people have trouble defining what a Use Case is, FPA provides a perfect definition of a Use Case.)

Another useful aspect of FPA is that once you understand how the process works, you can say things like, "Historically we write new software at the rate of 2.5 Hours/FP." This rate of "Hours/FP" can be thought of as your *development speed* or *productivity rate*.

Understanding your speed will help you understand the cost of future projects, and can also help you understand whether new programming languages and technologies help make your teams

more productive. It can also help when a new project is developed faster or slower than your historical speed. Imagine that your historical speed is always around 3.5 Hours/FP, but then a new project is developed at something like 2.5 Hours/FP. This enables you to say, "Wow, we developed that application fast," and then also ask, "Since we know that we developed that application fast, what was different that enabled us to be so much faster?" The key point here is that you *know* you were faster, it didn't just "feel" faster.

Q: Are FPs really analogous to "square feet"?

I'll say "yes" in that they are both units of measure relating to "size." The biggest benefit I get from knowing the size of an application is that I can apply my historical development speed to understand how long it should take me to build that application, and what the cost of building that application should be.

A good analogy is when I had a house built back in 1999. I worked with a straightforward homebuilder who said, "Al, you have two choices here. First, how many square feet do you want to build? Second, what quality of materials do you want to use?" Then he continued, "Let's say that you want to build a house that's 2,000 square feet. If you want to use cheap materials we can build it for $80 per square feet. That's $160,000. If you want to go with top of the line materials then you're looking at more like $120 per square foot, so that's $240,000. What would you like?"

Don't read into this example that building a software application is *exactly* like building a house. We all know that there are a lot of other variables, and it's not quite this simple. But Function Points do get you a lot closer to this situation.

For example, when I first learned about FPA one of my teams was working on a Java software development project that was building a Swing (GUI) application to run on both Microsoft Windows and Mac OS X computer systems. When we were building this application we were able to bid on it at a rate of about $250/FP. So, is it exactly like building a house? No. But were we able to bid on fixed-price projects? Yes. And we couldn't do that just a year earlier.

Q: What are the benefits of FPA?

I stated most of the benefits earlier, and this is also similar to the previous question, so I'll quickly restate the most important points:

- You can learn how fast your development teams are, i.e., their speed or productivity rate, measured in Hours/FP
- You can use this historical development speed to predict the time and cost of future projects
- You can use this speed to understand whether changes in technology help your teams
- You can use advanced FPA techniques to make very early time/cost estimates

Another interesting point is that eXtreme Programming advocates told us that we should try to estimate future projects using "Yesterday's Weather." The theory is that if you're a weather forecaster, you can go on television and say, "Today's weather will be just like yesterday's weather," and roughly 75% of the time you'll be correct.

Applied to software projects, this means that you say something like, "This new Project B feels a lot like Project A, and that project cost $100,000, so this one should cost $100,000 also." I never liked that approach, but at the time it was the only thing we knew.

However, when I learned about FPA, I realized that FPA gives us a much better version of Yesterday's Weather. With FPA we know (a) our historical development speed, (b) the size of our previous projects, and (c) the size of our new project, so we can estimate that new project knowing our speed and its size.

XP also told us that Project Owners (the customer) have a right to an estimate. As a business owner, I wholeheartedly agree with that. (Here's a link to customer, programmer, and manager rights.)

Q: Where is FPA effective?

As you'll learn when you dig into the technology, FPA is most effective when applied to business applications. These applications typically take input data from users, apply relatively simple or well-known algorithms to that data, store the data, let users edit the data, and display the data in a variety of output formats.

Although FPA is technology-independent and doesn't care whether you use Oracle, SQL Server, MySQL, MongoDB, Cassandra, or other database technologies, it generally assumes that your application will store, add, modify, and delete data.

If that sounds limiting, it may be helpful to know that I just described most business applications, including banking, finance, and manufacturing applications, as well as applications like Twitter, Facebook, Instagram, etc.

Q: Are the cases where FPA is not effective or should not be used?

Yes. I have found that FPA does *not* work well in the following cases:

- When you're first learning new technologies. For instance, I recently wrote an application where I had to work heavily in processing sound files. I had never done this before, so FPA could not help here. (You could say that there was no version of "Yesterday's Weather" I could refer to.)

- Bug fixes. As you'll see when we get into the details, FPA uses "averaging" techniques that work in the *macro*, and bug fixes are often in the *micro*.

- When you're trying to invent something new. Imagine asking Einstein in 1905, "Okay, you figured out the Theory of Special Relativity, now how long will it take you to invent

the Theory of General Relativity?" (As it turns out, he published the General Theory in 1916.)

- Creativity. By working with a diverse set of clients, I've learned that beauty is indeed in the eye of the beholder. Once you've worked with a client for a while you actually can estimate how long it will take to make something look nice *for them*, but until you have that experience with a client, you need to state "Creative Design" separately.

- When your application is more heavily based on algorithms and less heavily based on manipulating data.

What this means is that FPA works well for most typical business applications, but wouldn't be appropriate for something like estimating the cost of software that will control a spaceship or rocket, or other applications that are depend heavily on scientific algorithms, and are light on situations where end users process and manipulate data.

Q: Does the size of an application always correspond to the cost required to develop it?

No. It's very possible to have two software projects that both have a size of 1,000 FPs whose cost can be orders of magnitude different.

When you count Function Points you end up with a value like "1,000 FPs." This is called an *Unadjusted Function Point Count* (UFPC), and this UFPC is actually the end of the ISO Standard.

> Once estimators have the UFPC for their application, they tend to diverge a little bit in how they make their estimates. Or put more accurately, there are many tools and techniques that take the UFPC as input, and those tools/techniques apply a series of factors to the UFPC to determine time/cost estimates; cost estimators diverge in the tools and techniques they use at this point. Personally, I usually just take the UFPC and multiply it by a productivity rate (Hours/FP) to determine an estimate of the hours required, but there are many more ways to use the UFPC to get time/cost estimates.

To see how different an estimate can be for two applications that have the same size, imagine two projects with the same UFPC count, one named *Project A* and the other named *Project B*. Project A is developed for 10 users, and those users aren't too concerned about uptime. Of course they'd like the application to always be running, but if it's down an hour a week, they can live with that.

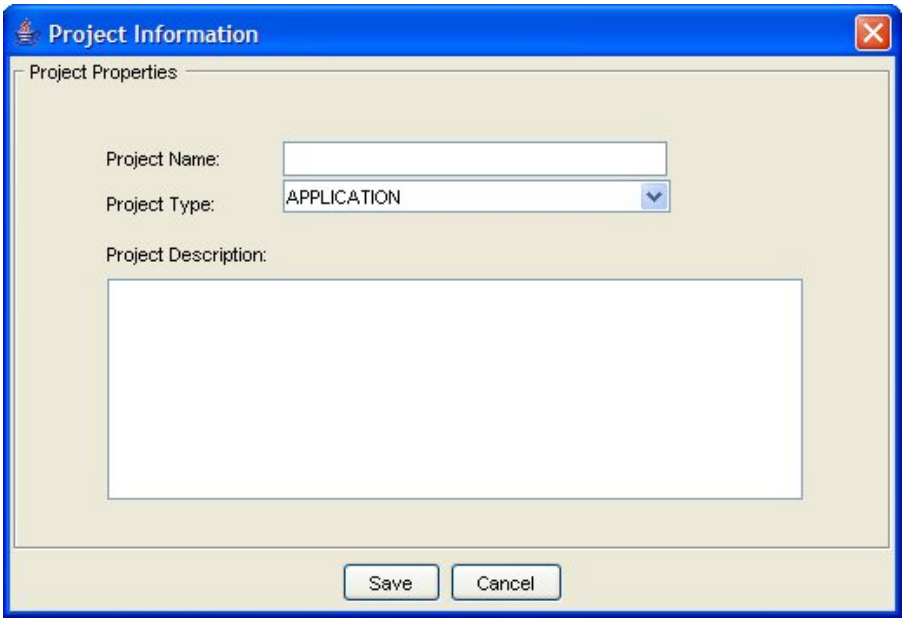
Now imagine that Project B is the size of Facebook or Twitter, with millions of users around the globe, so the application needs to scale immensely. Furthermore, any downtime at all is unacceptable. Both applications have a functional size of 1,000 Function Points, but their deployment and maintenance costs are incredibly different.

I'll discuss Unadjusted and Adjusted Function Point counts more in Lesson 1.

Q: Can I use FPA to determine whether a small new feature will take eight hours or 16 hours?

Typically, no. As you'll see in the sample count in Lesson 1, FPA uses "averaging" techniques to make counting Function Points very fast. These averaging techniques work well on a *macro* level, but aren't always accurate on a *micro* level.

For instance, in Lesson 1's sample count you'll see this screen:



When we're counting Function Points in the sample count we'll see this screen and think, "This screen has five DETs and 1 FTR, so it's a 'Low' EI, which counts as 3 FPs." An experienced FP counter can do this in their head in a matter of seconds, which is a good thing when you're counting very large applications.

Now, when you're estimating the time and cost of the complete application and you know that your developers work at an average speed of 2.5 Hours/FP, you'll state that has a functional size of 3 FPs will take 7.5 hours to develop. This is a *macro* level assumption that happens to work out very well.

However, on the *micro* level there can be any number of reasons that this particular screen will actually take more or less time to develop. For instance, suppose that it's the first screen developed in the application. In that case you may need to create the database table(s) behind it, or may need to create the initial Model/View/Controller framework behind your application. Or a senior-level developer may take this screen as an opportunity to train a junior-level developer. For any number of reasons this specific screen could happen to take more than 7.5 hours or less than 7.5 hours to develop.

So, in general, no, FPA is not used for small estimates like this.

That being said, once you learn how FPA works you can try to adapt its approach however you want to. You may come up with your own techniques that are based on FPA that work well on a micro level.

Q: Can I use these techniques on "agile" projects?

Absolutely. See Lesson 3 for a collection of ways that you can use FPA techniques on agile projects.

Q: Can I use these techniques to make very early "back of the envelope" estimates?

Absolutely. Again, see Lesson 3 for examples.

Q: You mentioned that FPA provides a great definition of a Use Case; what is that definition?

In FPA there is a term named *Elementary Process* that provides a terrific definition of a Use Case. If you adjust the definition of an Elementary Process to define a Use Case, it would look like this:

> "A UML Use Case is the smallest unit of activity that is meaningful to the user. A Use Case must be self-contained, and leave the business of the application in a consistent state."

If you think of the content that would be in Use Cases like, "Add a User," "Delete a Tweet," or "Post a Photo," I think you'll agree that this is a good definition of a Use Case. I explain this in much more detail in this article on my website.

Summary

I hope this Introduction has been a good start on what FPA is, and also what it is not. Given everything presented so far, it's time to move on to Lesson 1.

Lesson 1

How to Determine Your Application Size Using Function Point Analysis

Background

I don't know what his original concept looked like, but Function Points (FPs) were introduced by Alan Albrecht of IBM in 1979. Despite being introduced so long ago, they remain largely unknown by most developers today. For instance, back in 2002 when I first learned about Function Point Analysis (FPA), I personally knew over 120 software developers at 20 different companies, and none of them had ever heard of Function Points.

The closest I ever came was earlier that year when a friend started telling me that he really wished there was a good metric for determining the size of software applications, because he sure was tired of making up estimates all the time. (Actually, he referred to this process as "pulling estimates out of thin air.") He was looking at something called *Use Case Points*, which are a very informal, non-repeatable, non-standard approach I had previously dismissed. He was trying to get this to work, and I suggested that he take a look at FPs instead.

As I learned, being aware of FPs and what they add to your overall understanding of the software development process is amazing. At the very least, understanding FPA helps take much of the fear out of tasks like estimating time and cost. Additionally, it can also serve as an aid in measuring and improving product quality as you begin to understand critical metrics such as project velocity and defect ratio.

Objectives of Function Point Analysis

Most practitioners will probably agree that there are three main objectives within the process of FPA:

1. Measure software size by quantifying the functionality requested by and provided to the customer

2. Measure software development and maintenance independently of technology used for implementation

3. Measure software development and maintenance consistently across all projects and organizations

In working towards objectives 2 and 3 above, several organizations have created large repositories of FP counts that cross projects, technologies, and organizations. These repositories can be an invaluable tool for your first estimation efforts, because they let you compare your project to similar projects that have been developed by other organizations around the world.

Other Useful Information

Before we get into the practice of FP counting, it will help to know a few other background points:

There is a large FPA user group

A large user group known as the *International Function Points User Group*, or IFPUG (ifpug.org), is responsible for carrying the FP torch. IFPUG is a non-profit, member-governed organization, consisting of over 1,200 members in 30 countries around the world. At the time I learned FPA, version 4.2 of the IFPUG specifications for counting FPs (referred to as the *Counting Practices Manual*, or CPM) had just been released.

FPA is an ISO Standard

The *Unadjusted Function Point Count* of IFPUG v4.1 is an ISO Standard. In the "Sample Count" of this lesson you'll learn the basics of performing an Unadjusted FP Count.

FPA is a de-facto standard

In addition to being an ISO standard, FPs are used as the de facto standard for cost estimating applications like Cocomo II, Construx Estimate, and other software cost estimating packages.

You can become certified in FPA

A *Certified Function Point Specialist*, or CFPS, is a person who has passed the official IFPUG certification test. The CFPS designation must be renewed every three years.

There is an official manual

The *Counting Practices Manual*, or CPM, is the official manual created and distributed by IFPUG. It details the official counting rules used by CFPS practitioners. These rules help to keep counts consistent from one CFPS to another. Version 4.1 of this manual was over 300 pages in length.

There are FP data repositories

Because many organizations have been using FPA for quite some time, there are several large repositories of project data, where companies have provided FP counts along with other project information, such as the technologies used, man hours, and overall project cost. With accurate FP counts and other project data, you don't have to feel alone when making project estimates.

Overview – A five step counting process

In this section I'll provide a brief overview of the FP counting process, and then we'll dig more into the nitty-gritty details of the process.

To start at a high level, there are five steps in the process of counting FPs. They are:

1. Determine the type of count
2. Identify the scope and boundary of the count
3. Determine the unadjusted FP count
4. Determine the Value Adjustment Factor
5. Calculate the Adjusted FP Count

I'll introduce steps 1, 2, 4, and 5 during our sample count, because they are most easily introduced by using an example. Right now I'd like to dive into Step 3 of the process, because this is where the actual FP counting takes place.

In Step 3, FP practitioners look at a software application in terms of five standard functions, which we'll examine next.

Five Standard "Functions"

When counting Function Points in Step 3, there are five standard functional areas that you count. These areas are called "functions," and the first two are called *Data Functions*, and last three are called *Transaction Functions*. The names of these functions are listed below with very brief descriptions:

1. Data Functions:
    a. *Internal Logical Files* (ILF) - typically data that is maintained in your application
    b. *External Interface Files* (EIF) - typically read-only data that is maintained outside of your application (like a read-only web service)
2. Transactional Functions:
    a. *External Inputs* (EI) – screens where you add or modify data in your application
    b. *External Inquiries* (EQ) - typically read-only reports that users of your application see, where the reports have no significant processing logic
    c. *External Outputs* (EO) - something of a combination of an EI and an EO, where generated reports can contain processing logic, and ILFs can be maintained

I'll provide much more detailed definitions and examples of these terms in the pages that follow.

Using this terminology, when a person that counts FPs looks at a software system, they see something like this:

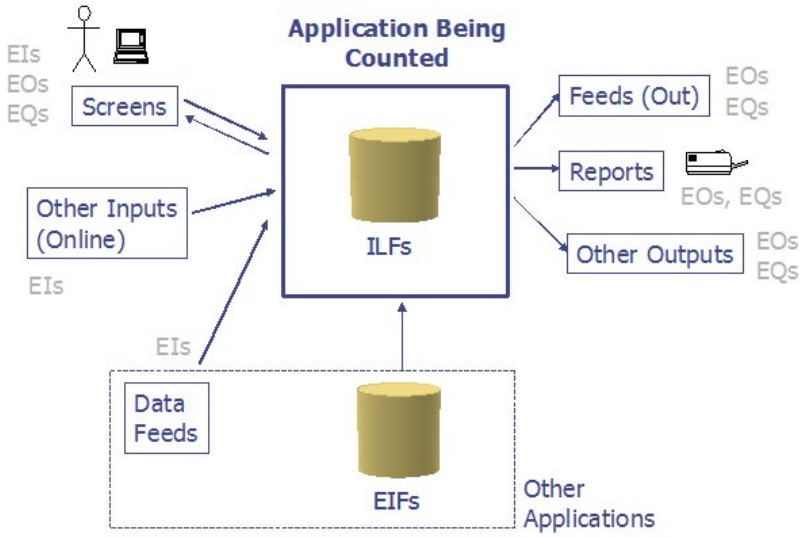


Figure 1: The view of a software application from the eyes of a Function Point practitioner.

Details on the Five Data and Transactional Functions

This section provides more detailed information and definitions on the five Data Functions and Transactional Functions, but before getting into the details of the five functions there are several terms that you need to understand, because they'll be used in each of the subsequent definitions. These terms are important to know, and the definitions are taken directly from the CPM.

User identifiable

This term refers to defined requirements for processes and/or groups of data that are agreed upon, and understood by, both the users and software developers.

Control information

Control information is data that influences an elementary process of the application being counted. It specifies what, when, or how data is to be processed.

Elementary process

An elementary process is the smallest unit of activity that is meaningful to the user. An elementary process must be self-contained and leave the business of the application being counted in a consistent state.

Data Element Type (DET)

A data element type is a unique, user recognizable, non-repeated field. This definition applies to both analyses of data functions and transactional functions.

Record Element Type (RET)

A record element type is a user recognizable subgroup of data elements within an Internal Logical File or External Interface File.

I know that those terms may not make much sense yet, but I hope they'll become more clear as you dig into the five functions and then work through a sample count.

1) Data Functions - Internal Logical Files (ILFs)

ILF stands for *Internal Logical File*. In my words, ILFs represent data that is stored and maintained within the boundary of the application you are counting. As you count the Function Points that are due to ILFs, you are basically putting a size on the data that your application is being built to maintain. (If you want to think of it using my house analogy, it's like walking into a room in a house, pulling out a tape measure, measuring it, finding that it's 15 feet wide by 20 feet in length, and declaring that the room is 300 square feet in size.)

The more precise IFPUG definition of an ILF is:

> "An ILF is a user-identifiable group of logically related data or control information maintained within the boundary of the application. The primary intent of an ILF is to hold data maintained through one or more elementary processes of the application being counted."

Furthermore, for data or control information to be counted as an ILF, both of the following IFPUG counting rules must also apply:

1. The group of data or control information is logical and user identifiable
2. The group of data is maintained through an elementary process within the application boundary being counted

Examples of ILFs

Samples of things that *can* be ILFs include:

- Tables in a relational database or documents in a NoSQL database
- Flat files
- Application control information, perhaps things like user preferences that are stored by the application
- LDAP data stores

This isn't to say that all these things *are* ILFs, just that they can be. For example, a database table named "songs" is probably an ILF if it's in an application like iTunes. That's because the songs in the application will be *maintained* (added, edited, deleted) through iTunes, and also because a "song" is a concept that an iTunes user can easily identify.

Function point counts resulting from ILFs

An important concept to know is that FPA uses "averaging" techniques that help make the counting process very fast. In regards to an ILF, this means that you'll look at how many Data Element Types (DETs) and Record Element Types (RETs) are in the ILF and then assign a

"complexity" rating of Low, Average, or High to that ILF. A *Low* complexity ILF is worth 7 FPs, an *Average* ILF is worth 10 FPs, and a *High* ILF is worth 15 FPs. This is shown in the following two tables.

Table 2 is a standard table that FPA practitioners know very well. You use it to determine whether the ILF you're looking at has a complexity level of Low, Average, or High. You do this by determining the number of RETs and DETs in the ILF, and then looking at this table to see whether the ILF's complexity is considered to be Low (L), Average (A), or High (H).

RETS	**Data Element Types (DETs)**		
	1-19	20-50	51+
1	L	L	A
2 to 5	L	A	H
6 or more	A	H	H

Table 2: *The ILF complexity matrix*

As an example, suppose that I'm looking at an ILF and I count that it has 5 DETs and 1 RET. According to this table, that's a *Low* complexity ILF. Then imagine that I have another ILF with 22 DETs and 2 RETs; that's an *Average* complexity ILF.

Now that you know whether the ILF under consideration has a complexity of Low, Average, or High, you come to Table 3 to determine the number of FPs that should be counted for the ILF. As Table 3 shows, a Low complexity ILF is worth 7 points, an Average ILF is worth 10 points, and a High complexity ILF is worth 15 FPs.

Complexity	**Points**
Low	7
Average	10
High	15

Table 3: *Function points that are assigned for the ILF complexity levels*

This is what I mean when I say that FPA uses "averaging" techniques. Instead of always worrying about the nitty-gritty details, such as the fact that an ILF has 22 DETs and 2 RETs, you just say that it's an Average ILF, so it's worth 10 FPs.

How this works when counting an application

Now imagine that you're counting the ILFs in a "personal information manager" application, and after the first three ILFs you have a grid of FP data that looks like this:

ILF Names	# DETs	# RETs	Complexity	# FPs
Notes	5	1	L	7
Contacts	22	2	A	10
Reminders	4	1	L	7
			Total:	**24**

As you can see from this simple example, it's easy to track FP information in a spreadsheet or application.

2) Data Functions - External Interface Files (EIFs)

EIF stands for "External Interface File." In my words, EIFs represent the data that your application will use in a read-only manner, and is not maintained (added-to, edited, or deleted) by your application.

The official IFPUG definition of an EIF is:

> "An external interface file (EIF) is a user identifiable group of logically related data or control information referenced by the application, but maintained within the boundary of another application. The primary intent of an EIF is to hold data referenced through one or more elementary processes within the boundary of the application counted. This means an EIF counted for an application must be an ILF in another application."

An EIF example

For an example of an EIF, first imagine that you work for a large corporation named MegaCorp. MegaCorp has an "employees" database table, and they insist that any application that is developed must read employee information from this table. The employees table is maintained by some other people and some other application you know nothing about; you just know that in your application, when you need employee information, you're supposed to read it from this table. Because your application (a) uses that data but (b) does not maintain that data, it is considered an EIF for your application.

Note that in modern applications you may need to access that data through a web service instead of accessing the database table directly. In that case you will count the data differently, but it will still be counted. (More on this later.)

Function point counts resulting from EIFs

Assigning an FP value to an EIF is the same as assigning one to an ILF. First, you determine the number of DETs and RETs in the EIF, and then you do a lookup in Table 4 to determine whether the EIF has a complexity of Low, Average, or High.

	Data Element Types (DETs)		
RETS	**1-19**	**20-50**	**51+**
1	L	L	A
2 to 5	L	A	H
6 or more	A	H	H

Table 4: *The EIF complexity matrix*

Then, once you know the EIF complexity, you look in Table 5 to determine the number of FPs to count for your EIF.

Value	# Function Points
Low	5
Average	7
High	10

Table 5: *Function Points assigned for each EIF complexity*

You'll notice that the lookup tables for ILFs and EIFs are almost identical, but more FPs are assigned to ILFs than EIFs. (It may be a little too early to ask this question, but can you guess why?)

At this point we are finished looking at how data is *stored* in an application. As shown, data is considered to be stored in an ILF or EIF. Next, we'll look at data transactions. That is, we'll look at how users add, edit, delete, and view data in applications, and then assign Function Points to those transactions.

3) Transaction Functions - External Inputs (EIs)

EI stands for "External Input." Here's the official IFPUG definition of an EI:

> "An external input (EI) is an elementary process that processes data or control information that comes from outside the application boundary. The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system."

General examples of EIs include:

- Data entry by users
- Data or file feeds by external applications

A few specific examples of EIs include:

- Adding a post to Facebook
- Creating a new tweet on Twitter
- Deleting a contact in an address book application

Function point counts resulting from EIs

Allocating FPs to EIs is similar to the process we used for ILFs and EIFs. However, in this case, instead of performing a lookup based on DETs and RETs to determine a Low/Average/High complexity, you perform the lookup using DET and FTR values. As you'll recall from the earlier definition, *FTR* stands for "file type referenced," so a FTR can be either an ILF or an EIF.

As an example, suppose that you have found an EI that has 5 DETs, and during its processing it references an EIF named Users and an ILF named Process (two FTRs). You would then go into Table 6 looking for the complexity of an EI that has 5 DETs and 2 FTRs. Table 6 shows that an EI with these attributes is considered an Average complexity EI.

FTRs	Data Element Types (DETs)		
	1-4	5-15	16+
0-1	L	L	A
2	L	A	H
3 or more	A	H	H

Table 6: *EI complexity matrix*

Just as we did with ILFs and EIFs, you then look at Table 7 to find the weight for an Average EI. Table 7 shows that an Average complexity EI is worth 4 FPs.

Complexity	# FPs
Low	3
Average	4
High	6

Table 7: *The number of Function Points assigned for each EI complexity*

If this isn't clear yet, fear not: you'll see examples of EIs in the "Sample Count" of this lesson.

4) Transaction Functions - External Outputs (EOs)

An External Output is referred to as an *EO*. IFPUG defines an EO as follows:

> "An external output (EO) is an elementary process that sends data or control information outside the application boundary. The primary intent of an external output is to present information to a user through processing logic other than, or in addition to, the retrieval of data or control information. The *processing logic* must contain at least one mathematical formula or calculation, create derived data, maintain one or more ILFs, or alter the behavior of the system."

Examples of an EO include:

- Reports created by the application being counted, where the reports include derived information

Function point counts resulting from EOs

Allocating Function Points to EOs is just like the process for EIs. Again, you perform your lookup using DETs and FTRs, and end up with a resulting Low/Average/High complexity.

As an example, suppose that you have a process that you've identified as an EO, and it has 10 DETs and references two FTRs. As before, you take this information and look at Table 8 to determine that this EO has Average complexity.

FTR	Data Element Types (DET)		
	1-5	6-19	20+
0-1	L	L	A
2-3	L	A	H
4 or more	A	H	H

Table 8: *EO complexity matrix*

Then you look at Table 9 to determine the number of FPs for an Average complexity EO, and you find that it has a value of 5 FPs.

Complexity	# FPs
Low	4
Average	5
High	7

Table 9: *The number of Function Points assigned to each EO complexity*

5) Transaction Functions - External Inquiries (EQs)

The last transactional function is referred to as an *EQ*, or External Inquiry. The IFPUG definition of an EQ is as follows:

> "An external inquiry (EQ) is an elementary process that sends data or control information outside the application boundary. The primary intent of an external inquiry is to present information to a user through the retrieval of data or control information from an ILF of EIF. *The processing logic contains no mathematical formulas or calculations, and creates no derived data*. No ILF is maintained during the processing, nor is the behavior of the system altered."

Examples of EQs include:

- Reports created by the application being counted, where the report does not include any derived data.

- Other things, including "implied inquiries." (This is a little beyond the scope of this tutorial, but implied inquiries are typically used to create dropdown widgets and other list and tree widgets in the UI, where those widgets are dynamically populated with data from ILFs and EIFs.)

Function point counts resulting from EQs

Allocating an FP count to EQs is very similar to the process for EIs and EOs. Again, you perform your lookup using DETs and FTRs, and end up with a Low/Average/High complexity.

As an example, suppose that you have a process that you've identified as being an EQ, and it has 20 DETs and references 4 FTRs. You then look at Table 10 and see that an EQ with these values is a High complexity EQ.

FTRs	Data Element Types (DETs)		
	1-5	6-19	20+
0-1	L	L	A
2-3	L	A	H
4 or more	A	H	H

Table 10: *EQ complexity matrix weights*

Just as with the previous transaction types, you then look into Table 11 to find that a High complexity EQ is worth 6 FPs.

Complexity	# FPs
Low	3
Average	4
High	6

Table 11: *The number of Function Points assigned to each EQ complexity*

Summary

Taken together, the two data functions (ILFs and EIFs) and three transactional functions (EIs, EOs, EQs) represent the five functions that are counted in a FP count. When you add up the FPs from these five functions, you'll have the total "Unadjusted Function Point Count" for the application you're analyzing. There are good reasons for the other steps in the FPA counting process, but Step 3 -- determining the unadjusted function point count -- is really the heart of the process.

It's worth mentioning at this point that Tables 2 through 13 are standard FPA tables. When I counted Function Points on a regular basis I used to carry a "cheat sheet" with me that had these tables printed on them (along with some of the more obscure FPA rules that I couldn't remember). If you count FPs on a regular basis you'll quickly commit these values to memory.

A Sample Count

In the next section of this lesson we'll perform a Function Point count on a sample application. After a brief introduction to our application, the sample count will proceed in the following order:

Step 1: Identify the type of count

Step 2: Identify the scope and boundary of the count

Step 3: Determine the unadjusted function point count

Step 3a: Determine the count resulting from ILFs

Step 3b: Determine the count resulting from EIFs

Step 3c: Determine the count resulting from EIs

Step 3d: Determine the count resulting from EOs

Step 3e: Determine the count resulting from EQs

Step 4: Determine the Value Adjustment Factor (VAF)

Step 5: Calculate the Adjusted Function Point Count

Every Function Point count you ever perform will follow Steps 1 through 3e. Those are all part of the ISO Standard.

What happens after Step 3e varies significantly. The result of Step 3e is an "Unadjusted Function Point Count" (UFPC), and some people take this number and feed it into other software, where they apply a variety of "application complexity factors" that help them obtain estimates of time and cost. I discuss the most typical adjustment factors in Step 4, but they are things like scalability requirements, uptime requirements, and so on.

Personally, when I finish with Step 3e I usually just multiply the UFPC by a productivity rate. For instance, if it's a Java Swing GUI app, I might multiply the UFPC by 4.0 Hours/FP to determine the total cost. If it's a web application I might multiply the UFPC by 3.0 Hours/FP to get the cost. To be clear, I use these values for "typical business applications," and the values I use are based on historical data. If there is something atypical about the current project, I have to find other ways to factor in whatever it is that's atypical about the current project.

An introduction to our example

So far I've given you a lot of background information and counting rules. The best thing to do now is to jump into the process of counting a sample application.

To demonstrate how the FPA process works, I'm going to count a "thick client" application that I've created. The application is named *FPTracker*, and as its name implies, it's a tool that I use when I perform FP counts on other applications. For instance, if I came to your company to perform FP counts on your applications, I would record the data from the functions in your applications using this tool. Its output provides some convenient reports and an analysis of the data to help keep me from making errors in my counts.

The FPTracker application consists of the following primary process areas:

- Project management, including creating, editing, and deleting projects
- Entity management, including creating, editing, and deleting ILFs and EIFs
- Process management, including creating, editing, and deleting EIs, EOs, and EQs
- Process group management, which is a mechanism for grouping processes
- Reporting, which includes several useful FP reports

For the purposes of this tutorial, I'm only going to cover a subset of the application. In a few moments you'll learn more about the FPTracker application by seeing the data it stores, the screens it offers, and two output reports it provides.

Let's begin the count now by following Steps 1 through 5.

Step 1: The type of count

The first step in the FPA process is to determine the *type* of count for the application at hand. The three different types of Function Point counts are shown in Table 12.

Name	Definition
Development Project FP Count	Measures the functions provided to the users with the first installation of the software being delivered (i.e., a new application that is being developed).
Enhancement Project FP Count	Measures the modifications to an existing application.
Initial Application FP Count	Measures the functionality provided to users in an existing application.

Table 12: *Types of FPA counts*

Because my FPTracker application already exists and is in production, the count type is an "Initial Application FP Count."

That's all we have to do for Step 1. Note that assigning this type does not affect the UFPC. The "type" is often used by software tools that estimators use after Step 3 of the FPA process, and those tools use different equations for the three different types.

As a quick example of this, a cheat sheet I have shows these different equations for the three different types of application counts:

```
# development project
Adjusted FP Count = (UFPC + CFP) * VAF

# enhancement project
Adjusted FP Count = ((UFPC + CHGA + CFP) * VAFA) + (DEL * VAFB)

# initial application
Adjusted FP Count = UFPC * VAF
```

The variables in those equations don't matter at this point; the only thing I wanted to show is how one particular system uses different equations for the different types of counts.

Step 2: Identify the scope and boundary of the count

The second step of the FPA process is to identify the scope and boundary of the count. While this sounds trivial, I've been surprised by how many times I've talked to a client and found that they were unable to completely and accurately define the accurate scope and boundary of their application at our first meeting. I've found that just asking clients to identify the scope and boundary of their application helps to clarify their thinking -- something I wish I knew many years ago.

For this example, the *scope* of the count will be defined by the data, screens, and reports that I'm about to show you; you should not make any assumptions about any behavior that may appear to be missing.

Regarding the *boundary* of this application, for our purposes FPTracker should be thought of as a simple, standalone software application with its own self-contained database. Unlike a suite of applications like Microsoft Office, or a combination of applications that can be found on an Intranet or Extranet site, FPTracker is not tied to any other applications in any way. Therefore, if I were to draw a boundary line to set off FPTracker from other applications, it would be similar to the boundary line shown in the example back in Figure 1.

Step 3: Determine the unadjusted function point count

Although the first two steps are important to set the groundwork for your count, it's the third step of the process that takes the majority of your time, and is the heart of the count.

In Step 3, you'll count the data functions and transactional functions that yield the Unadjusted FP Count (UFPC). As you've seen, to this point we haven't been dealing with any numbers, but we're now ready to dig in and start adding them up.

Step 3a: Determine the count resulting from ILF's

In the FPTracker application the data is stored in a series of relational database tables, so we'll use those as the basis of our ILF analysis. The following is a list of the database table names used in the FPTracker application:

1. Project
2. Entity
3. Process Group
4. Process
5. ProcessDETs
6. ProcessEntities

For the purposes of the current discussion I'm going to assume that each of these database tables corresponds to an ILF. This is actually not a good assumption, as you'll see shortly, but it makes the current discussion easier. (We'll clean up this wrong assumption very soon.)

The fields stored in each database table are shown in the tables that follow. These tables show each database table field name, a description of the field, an indicator as to whether the field should be counted as a DET, and other notes/comments about each field. The last row of each table provides a count of the total number of DETs in the database table.

Table 13 shows the fields and DETs for the Project database table.

Field	Description	Count as a DET?	Notes
project_id	Sequential id, system-assigned.	No	This is a technical artifact. It is not user-recognizable, and therefore not counted.
project_name	The name a user assigns to a given project.	Yes	
project_type	The type of project count.	Yes	
description	A description of the project.	Yes	
	Total DETs:	**3**	

Table 13: *Fields in the "Project" database table*

Table 14 shows the fields and DETs for the Entity database table.

Field	Description	Count as a DET?	Notes
entity_id	Sequential id, system-assigned.	No	System-generated sequence number. Not user-recognizable.
project_id	Foreign key.	Yes	Do count a DET for pieces of data that are required by the user to establish a relationship with another ILF or EIF. Foreign keys usually fit this definition.
name	Name of the entity.	Yes	
type	Type of entity (ILF or EIF).	Yes	
num_rets	Number of RETs in the entity.	Yes	
num_dets	Number of DETs in the entity.	Yes	
complexity	Calculated complexity (Low, Average, or High).	Yes	
	Total DETs:	**6**	

Table 14: *Fields in the "Entity" database table*

Table 15 shows the fields and DETs for the Process database table.

Field	Description	Count as a DET?	Notes
process_id	Sequential id, system-assigned.	No	System-generated sequence number. Not user-recognizable.
pg_id	Foreign key.	Yes	
name	Name of the process.	Yes	
type	Type of process (EI, EO, or EQ).	Yes	
sort_order	Order of appearance when displayed.	Yes	
num_ftrs	Number of FTRs.	Yes	
num_dets	Number of DETs.	Yes	
complexity	Low, Average, or High.	Yes	
description	Description of the process.	Yes	
	Total DETs:	**8**	

Table 15: *Fields in the "Process" database table*

Table 16 shows the fields and DETs for the ProcessGroup database table.

Field	Description	Count as a DET?	Notes
process group id	Sequential id, system-assigned.	No	System-generated sequence number. Not user-recognizable.
project id	Foreign key.	Yes	
name	Name of the process group.	Yes	
	Total DETs:	**2**	

Table 16: *Fields in the "ProcessGroup" database table*

Table 17 shows the fields and DETs for the ProcessDETs database table.

Field	Description	Count as a DET?	Notes
id	Sequential id, system-assigned.	No	System-generated sequence number. Not user-recognizable.
process_id	Foreign key.	Yes	
sequence	Sort order, for display purposes.	Yes	
description	User-supplied description.	Yes	
	Total DETs:	**4**	

Table 17: *Fields in the "ProcessDETs" database table*

Table 18 shows the fields and DETs for the ProcessEntities database table.

Field	Description	Count as a DET?	Notes
id	Sequential id, system-assigned.	No	System-generated sequence number. Not user-recognizable.
process_id	Foreign key.	Yes	
entity_id	Foreign key.	Yes	
sequence	Sort order, for display purposes.	Yes	
	Total DETs:	**3**	

Table 18: *Fields in the "ProcessEntities" database table*

As I mentioned, these database tables don't correspond directly to ILFs. Specifically there is not a one-to-one relationship between the database tables and the way a *user of the application* logically views the data being stored. As an FP counter -- and also the primary user of this application -- I think of the data *logically* being stored like this:

- Project
- Entity
- Process Group
- Process
    - ProcessDETs
    - ProcessEntities

What I'm saying is that, as the user of this application, I think of the Process DETs and the Process Entities as being logical subgroups of the overall Process data. "Okay," you say, "but how does that matter to me as an FP counter?" To understand the implication of this, let's look at how our counts would differ if we (a) first assume there is a one-to-one relationship between database tables and ILFs, and then (b) treat the data as it is seen logically.

If you first think of the six tables as separate ILFs, because of the number of DETs they contain each table would be considered to be a Low complexity table. Because each Low ILF is worth 7 FPs, the FP count for these six ILFs would be 42 FPs (6x7=42).

However, if you look at the tables from a "logical" perspective instead of the physical RDBMS layout, what you really have are four logical groups. Because of their DET and RET counts, all four of the groups are considered Low in complexity. So, in this case, the total count is 28 FPs (4x7=28).

Therefore, had I counted the *physical implementation* of the tables as opposed to a *logical grouping*, I would have over-counted this portion of the application by 14 FPs, or 50%. This is a good example of where a physical implementation differs from a correct, logical grouping for an FP count.

Total count due to ILFs

To tally up the FP count resulting from ILFs, just create a list of your ILFs as shown in Table 19. As you can see from that table, I believe that I have four ILFs, and because of the RET and DET counts, each table has a Low complexity. Each ILF is therefore worth 7 FPs, and the total count resulting from the four ILFs is 28 FPs.

ILF	# RETs	# DETs	Complexity	Function Points
Project	1	3	Low	7
Entity	1	6	Low	7
Process Group	1	2	Low	7
Process	3	13	Low	7
			Total:	**28**

Table 19: *The FP count due to ILFs in our sample application*

Step 3b: Determine the count resulting from EIFs

In the FPTracker application there are no EIFs, so EIFs contribute zero FPs to the overall FP count.

If I was just counting FPs for this application I would quickly move on to Step 3C, but because this is a tutorial, I'll add a few comments here.

As I mentioned earlier in the EIF introduction, an Employees database table is a good example of an EIF. In a large corporation an Employees table is typically a read-only data store, and if it was used by FPTracker it would most likely be an EIF, because it would only be used for a purpose like logging into the application, which is a read-only process where you verify the user's username and password.

The defining rule here is whether or not the fields in the data store are maintained by the current application, or not. If the data store is maintained (modified) by our application it can be considered an ILF, but if it is not modified by our application and is only accessed in a read-only manner, it would be considered an EIF.

At the end of this step we have finished looking at data stores (ILFs and EIFs), and we're now ready to start counting Function Points for our application's transaction functions (EIs, EOs, and EQs).

Step 3c: Determine the count resulting from EIs

To begin counting the transactional functions we need to look at the application's user interface (UI). To do this in this tutorial, I'll show all of the application screens first, and then provide a description of each process. At that point we'll determine which of the screens are EIs, EOs, and EQs.

Note: Because FPA is technology-independent I use the term "screen" in this document as a generic way of referring to a frame, window, dialog, panel, or other type of user interface component/container that can be used on desktop, tablet, phone, watch, and other devices. If you happen to know the Java programming language, you'll notice that the "screens" in the following figures are a combination of JFrame, JWindow, and JDialog components/containers.

Application screenshots

The following figures show screenshots of the FPTracker application.

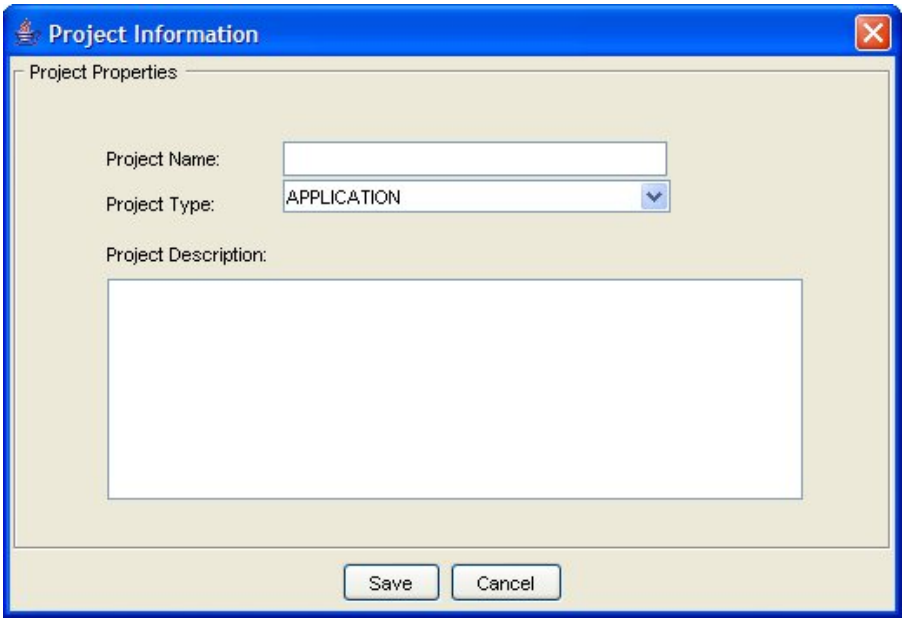


Figure 2: The "Project Information" window lets users create a new Project.

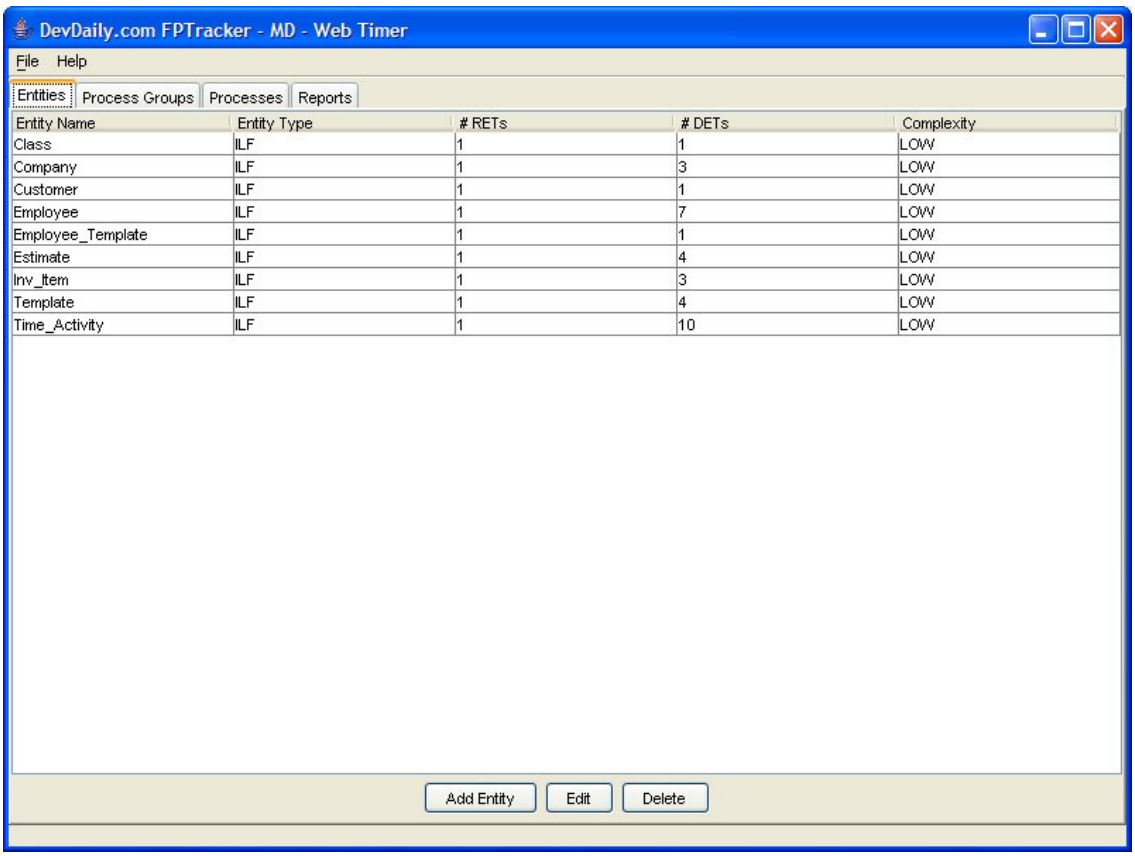


Figure 3: This screen shows the "Entities" tab on the main window of the application. From this screen users can view, add, edit, and delete entities.

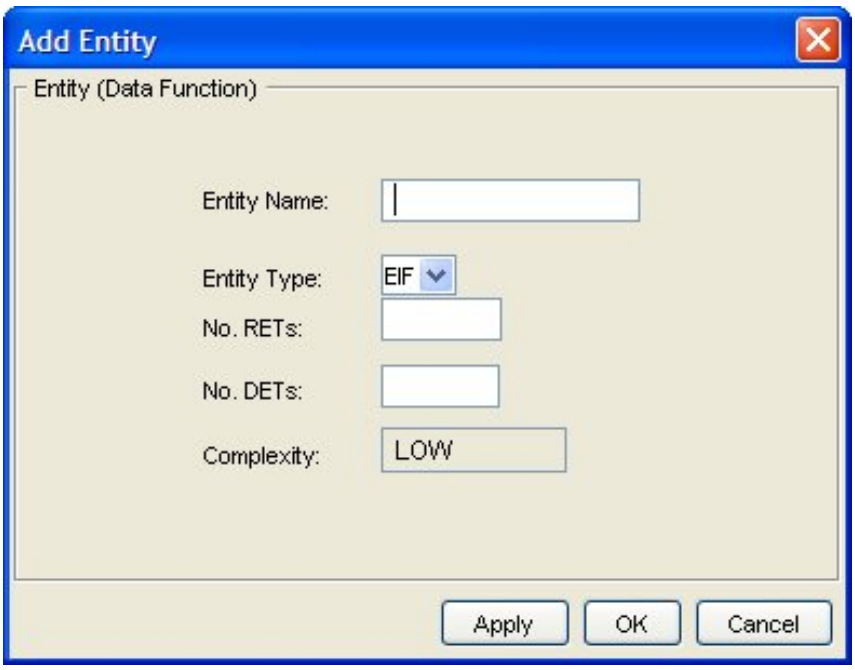


Figure 4: The "Add Entity" screen lets the user add new entities to the project.

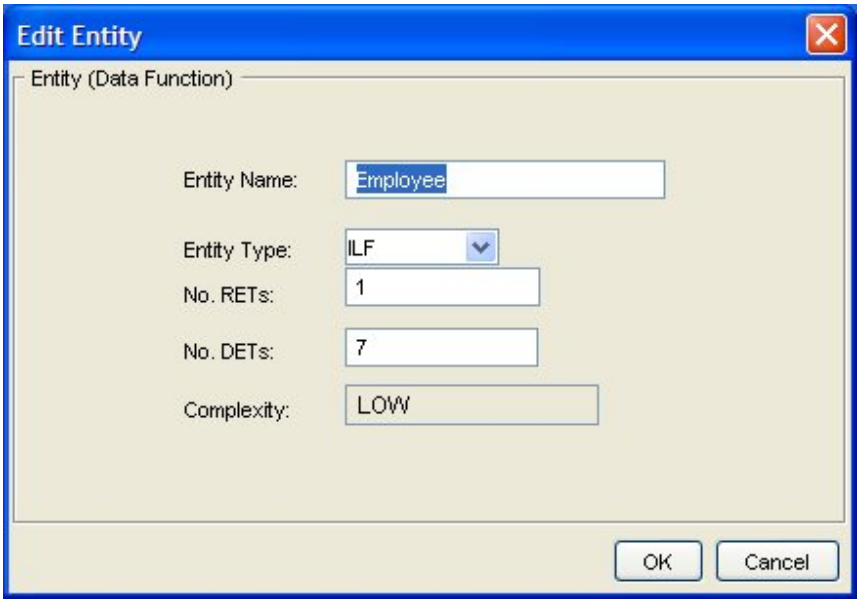


Figure 5: The "Edit Entity" screen lets the user edit an existing entity in the project.

Figure 6: The "Delete Entity" screen asks the user to confirm their attempt to delete an entity in the application.

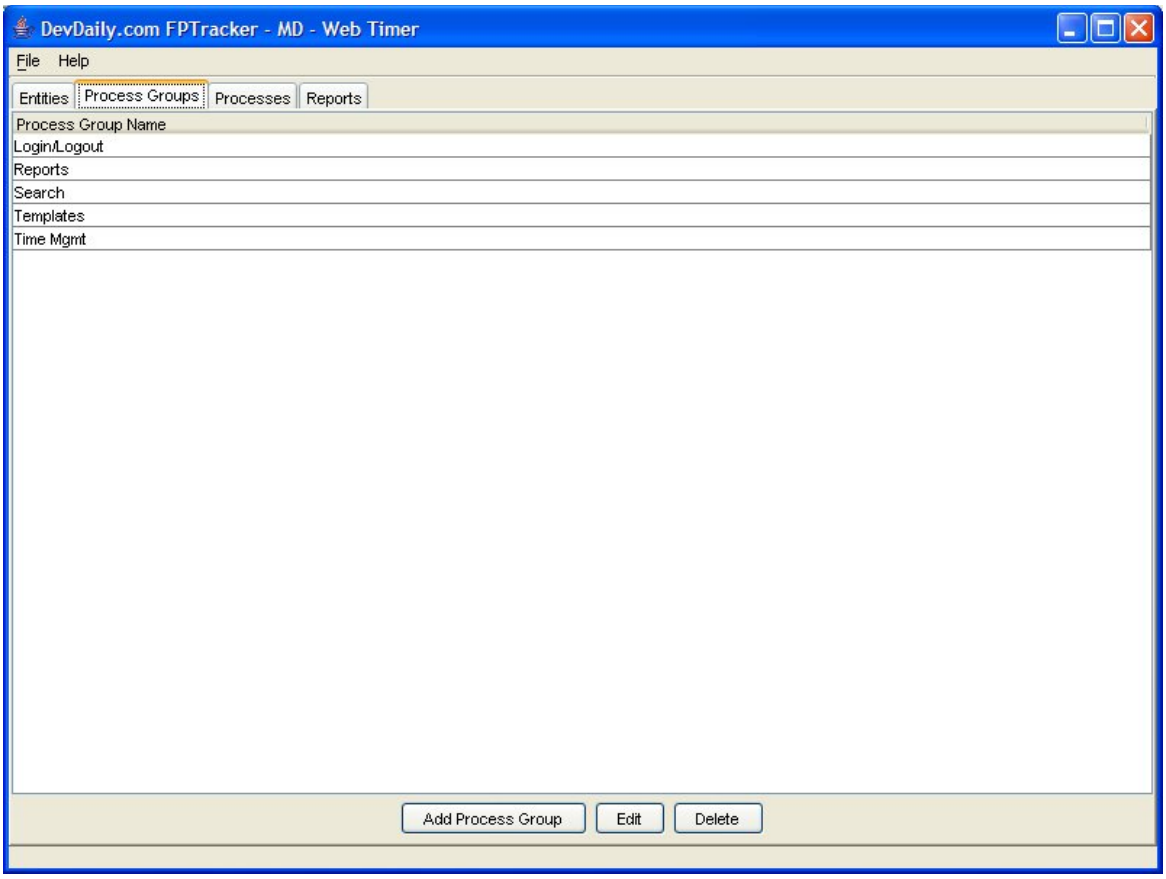


Figure 7: From the "Process Groups" tab in the main application window, users can add, edit, and delete Process Groups.

Figure 8: The "Add Process Group" screen lets the user add new Process Groups to the application.

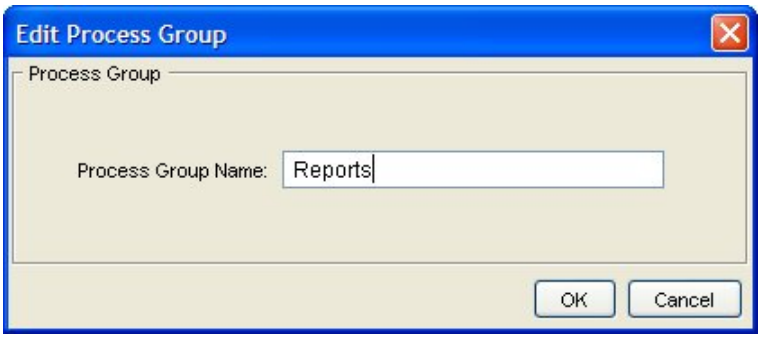


Figure 9: The "Edit Process Group" screen lets users assign a new name to a Process Group.

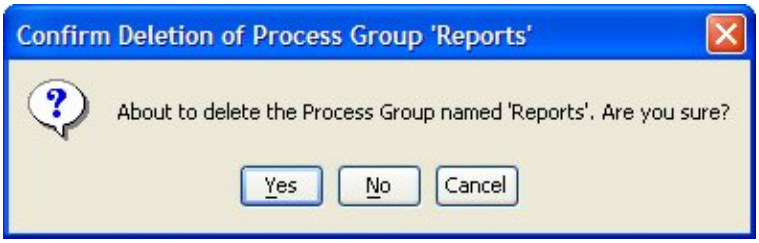


Figure 10: This window lets the user delete a selected Process Group.

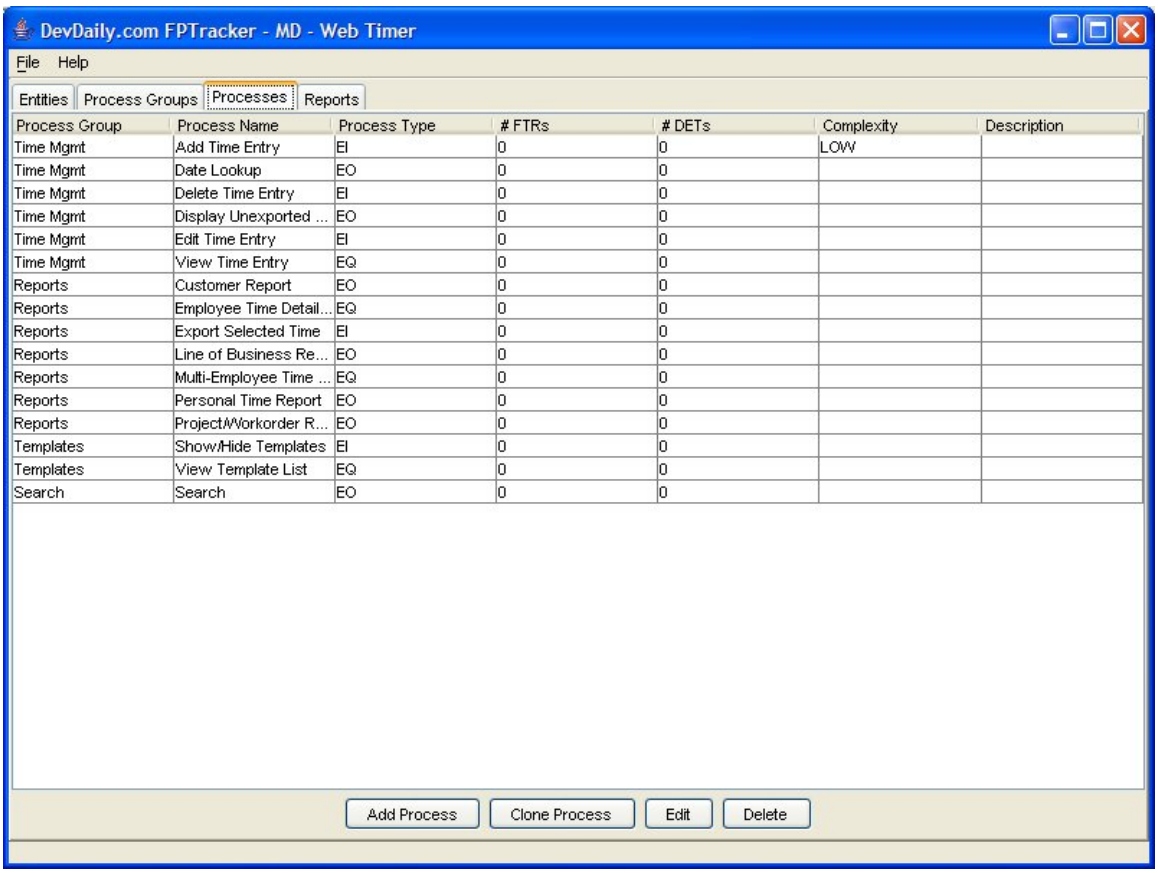


Figure 11: This figure shows the Processes tab on the main application window. From this location, users can add, edit, clone, and delete processes.

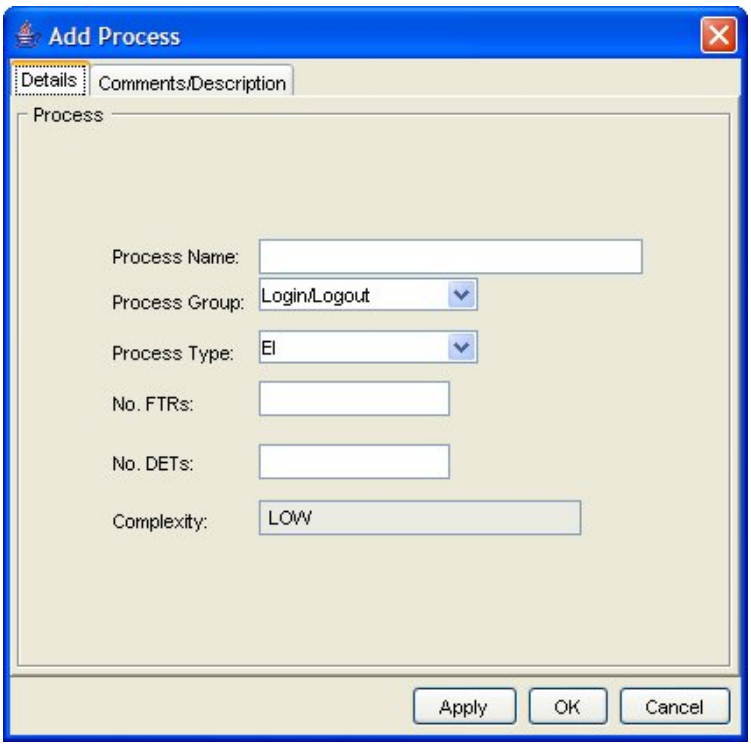


Figure 12: The "Add Process" window lets the user define a new process.

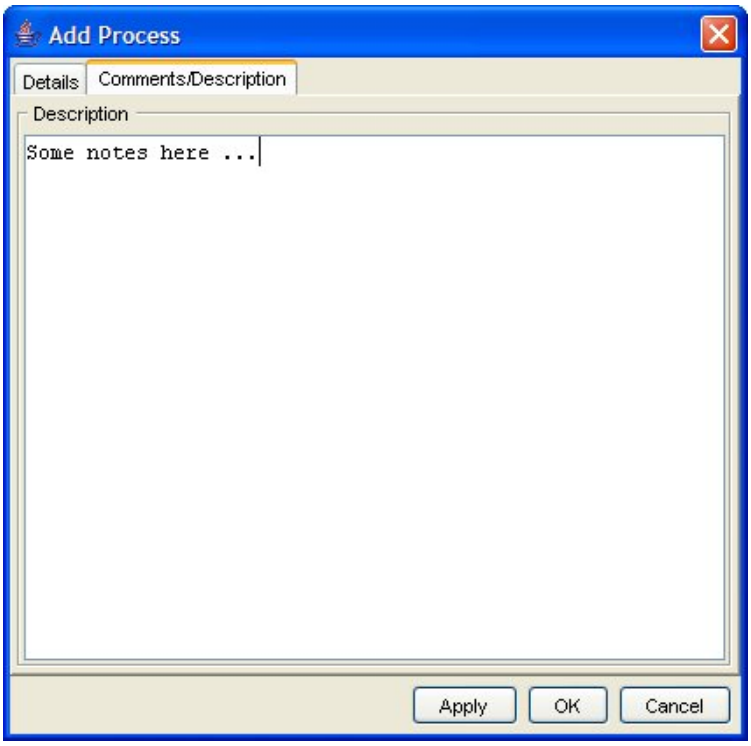


Figure 13: This image shows the details of the Comments/Description tab on the "Add Process" window. Users can optionally enter information here when creating a new process.

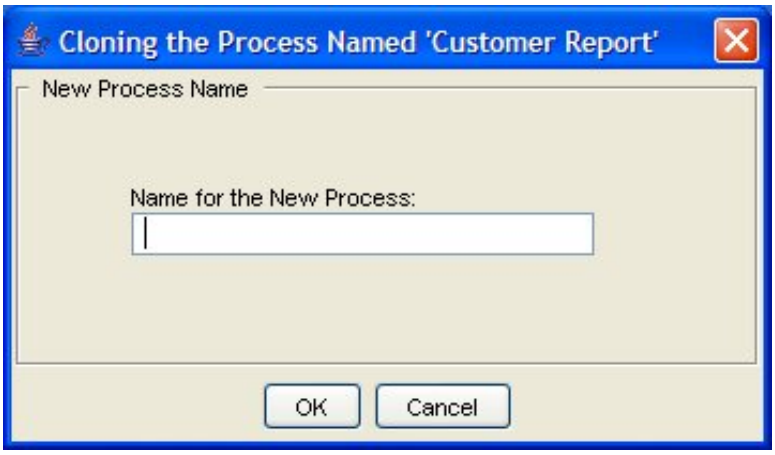


Figure 14: Users see this window when cloning a Process in the application.

Unadjusted Function Point Count

Function Type	Funtional Complexity		Complexity Totals	Function Type Totals
ILF	9 Low	x 7 =	63	
	0 Average	x 10 =	0	
	0 High	x 15 =	0	63
EIF	0 Low	x 5 =	0	
	0 Average	x 7 =	0	
	0 High	x 10 =	0	0
EI	1 Low	x 3 =	3	
	0 Average	x 4 =	0	
	0 High	x 6 =	0	3
EQ	0 Low	x 3 =	0	
	0 Average	x 4 =	0	
	0 High	x 6 =	0	0
EO	0 Low	x 4 =	0	
	0 Average	x 5 =	0	
	0 High	x 7 =	0	0
			Unadjusted Function Point Count:	**66**

Figure 15: This image shows an "Unadjusted Function Count Report" that is produced by the application.

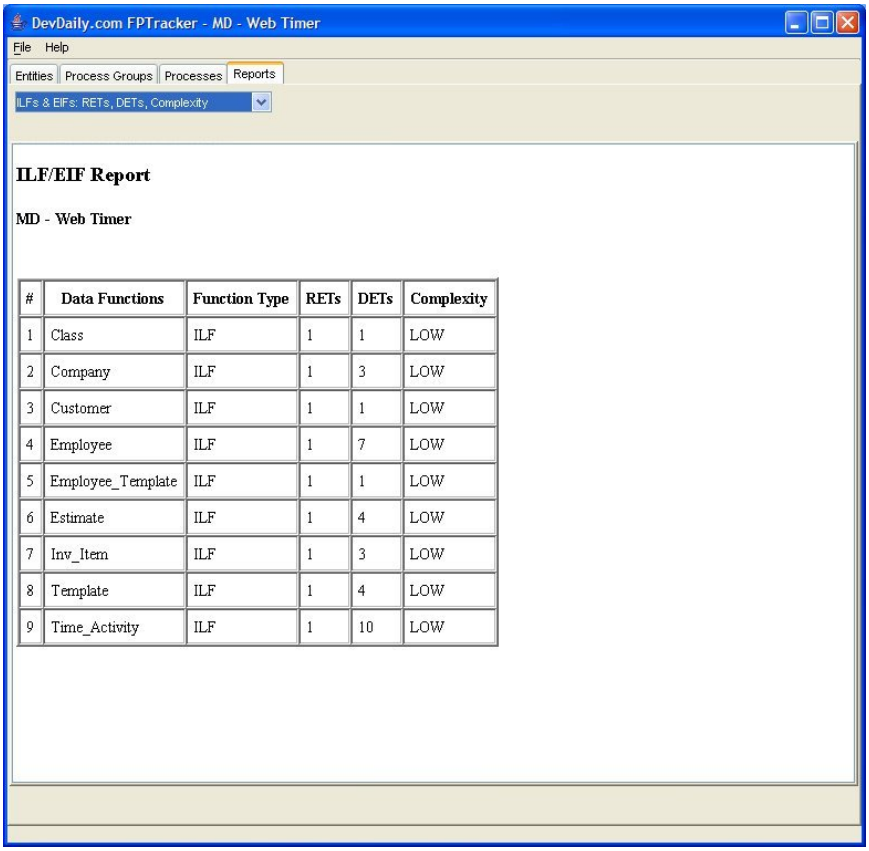


Figure 16: This image shows an "ILF/EIF" report produced by the application. The report lists all of the ILFs and EIFs in the current project.

Now that you've seen the screens in the application, the next step is to determine which of these correspond to EIs, which are EOs, and which are EQs. You'll do that right after I supply you with a short explanation of the elementary processes in the application.

Brief process descriptions

Because you haven't seen this application before, it may be hard to understand the processes in this application. Therefore, Table 20 provides a brief name and description of each process.

Process Name	Process Description
Create a Project	This screen is displayed in Figure 2. This lets the user create a new project. Note that for the purposes of this tutorial I'm skipping other project-related processes, including the ability to list, edit, delete, and clone projects.
Display Entity List	The list is displayed in Figure 3. This is a list of every Entity that the user has identified for the application they are counting.
Add Entity	Shown in Figure 4, this dialog lets the user add a new Entity.
Edit Entity	Shown in Figure 5, this dialog lets the user edit an existing Entity. This dialog appears when the user highlights a row in the Entity List, then selects the Edit button. Alternatively, they can double-click on the desired row.
Delete Entity	Shown in Figure 6, this confirmation dialog is displayed when a user selects a row, then presses the Delete button.
Display Process Group List	Shown in Figure 7, this is a list of every Process Group the user has identified.
Add Process Group	Shown in Figure 8, this dialog lets the user define a new Process Group.
Edit Process Group	Shown in Figure 9, this dialog lets the user edit an existing Process Group.
Delete Process Group	Shown in Figure 10, this confirmation dialog is displayed when a user selects a row in the Process Group table, then presses the Delete button.
Display Process List	Displayed in Figure 11, this is a list of every Process that the user has identified for the application they are counting.
Add Process	Shown in Figures 12 and 13, this dialog lets the user define a new Process.
Edit Process	Although not shown, this dialog is identical to screens shown in Figures 12 and 13, other than the title of the dialog. These screens let the user edit an existing Process.

Delete Process	Although not shown, this confirmation dialog is displayed when a user selects a row, then presses the Delete button. For the sake of our count, assume that it is a simple confirmation dialog, similar to the dialog for deleting a Process Group.
Clone Process	Shown in Figure 14, this dialog lets the user make a duplicate copy of a Process. This makes it easy to create new processes which are similar to an existing process.
Display UFPC Report	Shown in Figure 15, this is a typical report that totals up the number of FPs in the application you are recording. The report is displayed after the user selects the report type from a drop-down list of reports in the Reports tab. The list of data shown in the drop-down list is hard-coded into the application.
Display ILF/EIF Report	Shown in Figure 16, this report shows the ILFs and EIFs (i.e., all the FTRs) in the application, along with their associated number of DETs and RETs, as well as their complexity level. It is selected from the same drop-down list as the UFPC Report.

Table 20: *Elementary processes in the FPTracker application*

There is a little more to this application than what is shown in these figures, but for the purposes of this paper, these screens and descriptions define the scope of the application under consideration.

For our counting purposes, I'll also assume that each data-entry screen can result in one or more error screens. For example, when a user is creating a new project, the system will display an error dialog if they do not provide a project name, because this is a required field.

In the next section I'll classify each of these processes as an EI, EO, or EQ, and determine the number of FPs associated with each process.

Function points resulting from EIs

Table 21 lists the External Inputs in the FPTracker application, along with the DETs and FTRs for each EI, the complexity that results from the number of DETs and FTRs, the number of FPs for each EI, and a total count of FPs resulting from EIs.

Process	# DETs	FTR Names	# FTRs	Resulting Complexity	# FPs
Create Project	5	Project	1	Low	3
Add Entity	7	Project, Entity	2	Average	4
Edit Entity	7	Project, Entity	2	Average	4
Delete Entity	4	Project, Entity	2	Low	3
Add Process Group	3	Project, ProcessGroup	2	Low	3
Edit Process Group	3	Project, ProcessGroup	2	Low	3
Delete Process Group	4	Project, ProcessGroup	2	Low	3
Add Process	9	Project, Process, ProcessGroup	3	High	6
Edit Process	9	Project, Process, ProcessGroup	3	High	6
Delete Process	5	Project, Process, ProcessGroup	3	High	6
Clone Process	3	Project, Process, ProcessGroup	3	Average	4
				Total:	**45**

Table 21: *Function Points due to External Inputs (EIs) in the FPTracker application*

Step 3d: Determine the count resulting from EOs

According to the FPA rules, we have only one EO, and that is the UFPC Report. This report is considered an EO, and specifically not an EQ, because it contains derived data. The "Complexity total" column and the "Total UFPC Count" at the bottom of the table are derived fields, and EQs cannot contain derived fields like this.

This report is actually a difficult report for me to count right, and that's why I've included it here. It's hard for me because of the rules surrounding duplicated fields in reports. Rather than get into the complexity of the rules, I'm just going to state that there are at least seven DETs in this report, and they are:

1. Title
2. Function type
3. Functional complexity
4. The complexity totals
5. The function type totals
6. The Unadjusted FP Count
7. A DET that is counted for the menu option to choose this report

Process	# DETs	# FTRs	Resulting Complexity	# FPs
UFPC Report	7	3	Average	4
			Total:	**4**

Table 22: *Function Points due to External Outputs (EOs) in the FPTracker application*

As mentioned, because of the repeating fields this is a difficult report for me to count. Fortunately, when I double-checked this report with a colleague, we both came up with an Average score, even though we differed on the field count slightly. That's one of the nice things about the "averaging" process that FPA uses: very rarely does a disagreement about the detailed rules actually affect a count.

Step 3e: Determine the count resulting from EQs

Table 23 lists the External Inquiries (EQs) in the application. It also lists the number of DETs and FTRs for each process, and the complexity that results from those values.

I'll begin with the ILF/EIF Report. Fortunately, it's easier that the UFPC Report. It contains six DETs, including the title, data function names, function type, number of RETs, number of DETs, and complexity. Looking at the report, you can also see that it retrieves information from the Project and Entity ILFs, so there are two ILFs.

Implied Inquiries

The rest of the EQs in the table below are probably quite a surprise. I mentioned "implied inquiries" earlier in this document, and now you'll get to see what they are. They're very interesting because they're the *almost hidden processes* in an application that are easy to overlook.

For instance, if you look back at Figures 3, 7, and 11, you'll see that the UI has built-in lists of Entities, Process Groups, and Processes. These are the tables/grids in the Entity, Process Group, and Process tabs of the main application window. As you can imagine, some type of query had to be performed to generate these lists, and the queries that result in this "list" operation often qualify as implied inquiries. As the name suggests this data comes from queries, and the implication is that a query had to be made in the application to obtain this data.

The Process Group drop-down list on the Add/Edit Process dialog is another example of an implied inquiry. Just like the lists, it took some type of query to dynamically generate that list.

While I'm covering this topic quickly, it's important to note that other drop-down lists and tables may not qualify as implied inquiries. This is especially true when the data they contain is hard-coded in the application. No query is needed in this case; the application is just displaying static data, which is generally much easier than performing queries. This is an introductory tutorial, so that's all I'll say about this topic for now, other than to say that the CPM contains rules for all of these cases.

Process	# DETs	# FTRs	Resulting Complexity	# FPs
ILF/EIF Report	6	2	Average	5
Display List of Entities	5	2	Low	4
Display List of Process Groups	2	2	Low	4
Display List of Processes	7	3	Average	5
Implied Inquiry - Process Group ComboBox on the Add/Edit Process Dialog	1	2	Low	4
			Total:	**22**

Table 23: *External Inquiries (EQs) in the FPTracker application*

Total FP Count

Now that we've finished counting the ILFs, EIFs, EIs, EOs, and EQs for the FPTracker application, all you have to do is add up the individual transaction counts to get a total Unadjusted Function Point Count for the application. This result is shown in Table 24.

Function Type	Complexity	Multiplier	Line Item Sub-Total	Section Total
ILF	4 Low	x 7 =	28	
	0 Average	x 10 =	0	
	0 High	x 15 =	0	28
EIF	0 Low	x 5 =	0	
	0 Average	x 7 =	0	
	0 High	x 10 =	0	0
EI	5 Low	x 3 =	15	
	3 Average	x 4 =	12	
	3 High	x 6 =	18	45
EQ	0 Low	x 3 =	0	
	1 Average	x 4 =	4	
	0 High	x 6 =	0	4
EO	3 Low	x 4 =	12	
	2 Average	x 5 =	10	
	0 High	x 7 =	0	22
			Unadjusted Function Point Count:	**99**

Table 24: *The Unadjusted Function Point Count for the FPTracker application*

As you can see from Table 24, the total UFPC for this application is 99 function points.

Next steps

At this point the effort of counting Function Points in the FPTracker application is complete. As I mentioned earlier, the end of Step 3 is the end of the FPA ISO Standard. Once you're comfortable with how to estimate using UFPC information, where you go from this point is up to you.

That being said, even though Steps 4 and 5 aren't part of any standard, I recommend that you read them, as they still add a lot of value to your learning experience, especially if you're not familiar with all of the different factors that can make a software application more (or less) complicated.

Step 4: Determine the Value Adjustment Factor (VAF)

In this step we'll look at something named the *Value Adjustment Factor*, or VAF. But first, a warning about the VAF …

A warning about the VAF

Before I tell you what the VAF is, let me first give you a warning about it: To the best of my knowledge, most people don't use it. (Yes, that's a big warning.)

I can't tell you much about the history of the VAF, but what I can tell from the conversations I've had with other FPA practitioners and estimators is that they don't use the VAF, at least not directly. I think this stems from at least two reasons:

1) Most users count Function Points to derive a number that they can plug into another piece of software to determine a time and cost estimate. Those other software applications usually have their own equivalent of the VAF, and in fact, they instruct you to supply the "raw FP count," which is our Unadjusted Function Point Count (UFPC). So, in this case, the VAF competes against these vendor tools.

2) Some people -- myself included -- don't feel that the *General System Characteristics* (GSCs) in the VAF are flexible enough. You can see this in the math that follows, where you'll see that for two applications under consideration, if both start with the same UFPC count -- let's say 1,000 FPs -- after adjustments the hardest application in the world would be rated at 1,350 FPs, and the easiest possible application would be rated at 650 FPs after the VAF is applied. Let's say that the hardest application in the world has to scale to the size of Facebook or Twitter, and the easiest application would be written in Microsoft Access, run on Windows, and be used by only one user. Do you really think the first application is only about twice as hard to deliver as the second? I certainly don't, and that's why I don't use the VAF.

But with that being said, I do want to give you a quick introduction to the VAF because it introduces 14 system characteristics that are known to add to the complexity of software applications. If nothing else, knowing this list of characteristics can help in whatever technique you use to estimate the time and cost needed to deliver new software applications.

VAF Introduction

First up, here are a few facts and definitions to get the VAF ball rolling:

- The Value Adjustment Factor (VAF) consists of 14 "General System Characteristics," or GSCs.

- The GSCs represent characteristics of software applications. Each GSC is weighted on a scale from 0 (low) to 5 (high).

- When you sum up the values of the 14 GSCs you get a value named "Total Degree of Influence," or TDI. Because there are 14 GSCs that can vary from 0 to 50, the TDI can vary from a low of 0 (when all GSCs are low) to a high of 70 (when all GSCs are high).

The General System Characteristics (GSCs)

Next up, the here's a list of the 14 GSCs:

1. Data Communication
2. Distributed data processing
3. Performance
4. Heavily used configuration
5. Transaction rate
6. Online data entry
7. End user efficiency
8. Online update
9. Complex processing
10. Reusability
11. Installation ease
12. Operational ease
13. Multiple sites
14. Facilitate change

I didn't provide a definition of each GSC, but hopefully you can infer what each characteristic means from its name.

Given these GSCs, the VAF formula looks like this:

```
VAF = (TDI*0.01) + 0.65
```

With this formula, the VAF can vary in range from 0.65 (when all GSCs are low) to 1.35 (when all GSCs are high). In the next section you'll see that the VAF is applied directly to the FP count to determine the "Adjusted Function Point Count." As I mentioned earlier, I don't think this range is anywhere near as wide as it needs to be, which is why I don't use it.

Step 5: Calculate the Adjusted Function Point Count

The last step in the FPA five-step process is to determine the Adjusted Function Point Count. Per the traditional FPA approach that uses the VAF, for Initial Application counts such as ours, this is determined with the following equation:

```
Adjusted FP Count = Unadjusted FP Count * VAF
```

As you saw in the previous section, the VAF can vary from 0.65 to 1.35, so the VAF exerts an influence of +/- 35% on the final Adjusted FP Count.

Since I'm not going to bother working through the 14 GSCs for this tutorial, I also won't derive an Adjusted FP Count for the FPTracker application, but at least you have an idea of how this works.

The Value of Counting Function Points

In the Sample Count I showed you the "how" of FP counting, but now I'd like to return to the "why" of FP counting, because I think motivation is an important driver here. At the very least you should ask yourself, "Why should I add as much as 1% to my overall software development effort?" to count Function Points.

My experience with FPA has shown all of the benefits I mentioned earlier in this document. Once you have a history of developing applications and you also have FP counts for all those applications, you'll have these capabilities in your software development arsenal:

- The ability to accurately estimate:
    - New project duration
    - New project cost
    - Optimum project staffing size
- Your historical database will also yield other important metrics, such as:
    - Hours/FP (known as "project speed" or "productivity rate")
    - Cost/FP
    - Project defect rate
    - The productivity benefits of using new or different tools
    - The ability to estimate project time and cost very early in the software development lifecycle (as discussed in Lesson 3)

So I think a good question is, "What are these capabilities worth to you?"

As I mentioned in the Preface, a huge benefit for my company was that FPA-based estimating allowed us to bid on fairly large software projects in a fixed-price manner, something we had to do at that time to survive. Once I went back and counted the size of previous projects and then looked at the man-hours on those projects, I was able to apply our old productivity rates to new projects.

Another big, lasting benefit is that I can meet a new prospect, listen to them describe an application they want, and be able to give them a decent, quick, "back of the envelope" time/cost estimate. (See Lesson 3 for more details on these techniques.)

An example of how my approach works

As an example of how my approach works, I'm going to walk through a process that I followed with a team recently. This example involves two small phases of development on one project. On this project, the developers, project managers, users, technology, and application all remained constant. The team also followed a relatively formal process, starting with a requirements phase, then development, testing, and final delivery of a portion of an application.

The first phase

For the first project phase I made an estimate of 400 FPs. That estimate was based on me reading the requirements documents, and asking questions as needed. I was able to make an FP count because the requirements were well written, the database was designed, and there were enough UI prototypes with the user stories that I could estimate the fields/widgets on each screen.

Once the application was delivered, I counted the number of FPs that were actually delivered to the users, and that total was 440 FPs. This was a growth from the requirements phase of 11%. I wasn't involved in the process between the requirements phase and final delivery, but after reviewing the delivered application I was able to identify the changes that had been made to the application during the development process.

At this point I also had some statistics from the development process. For instance, I learned that two developers worked on the project and had a total of 900 man-hours, which comes out to be 2.05 Hours/FP. Unfortunately no records were kept relating to testing time, so I couldn't determine other useful metrics like, "Number of Testing Hours Per FP."

During the development phase the developers did not take any vacation time, and they also worked a little overtime to account for some company overhead that took them away from the project. As a result, the 900 programmer-hours spent on the project happened over a period of 12 weeks. Because 440 FPs of functionality were delivered, this resulted in a delivery speed of 36.7 FPs per week.

There are other metrics that you can determine, such as "Cost per FP," but this was not pursued on this project.

The second phase

Because this was an ongoing project, we repeated the same steps on the next phase of the project. Here is a summary of the process on the second phase:

1. The requirements were developed as before.

2. I counted the FPs based on the requirements documentation.

3. I supplied an estimate of the time required to develop the application, including the assumption of another 11% "scope creep" during this phase.

4. The code was developed, tested, and delivered. The estimate was accurate to within 5%.

5. I counted the amount of functionality that was delivered, and found it had grown by 6% rather than 11%.

6. At the end of this second phase of development we were again able to determine important project metrics, including:

    a. Number of developer hours per FP

    b. Number of testing hours per FP

    c. The ratio of time spent developing to the time spent testing

    d. Elapsed calendar time and FPs/week

    e. Overall cost per FP (including other time for management, documentation, etc.)

    f. The ratio of Requirements time to Development time, and other similar ratios

After this second phase, we were confident enough in our approach to continue using it on more projects, specifically that we could estimate new projects by (a) using an FP count based on the requirements documents, and (b) multiplying that FP count by a historical productivity rate, given by Developer Hours per FP.

I think you'll find that metrics like DeveloperHours/FP, TestingHours/FP, Cost/FP, FPs/week, and ratios of Requirements time to Development time to Testing time are all very useful when it comes time to estimate future projects.

A Note on Regression-Based Formulas

I think it's important to note that different companies and organizations in the cost estimating industry have created formulas to attempt to estimate time, cost, and optimum project staffing size. The formulas they use are regression formulas based on publicly-available FP counts. The data they use includes a UFPC count, project duration, project man-hours, etc.

While I generally agree with this approach in theory, their equations haven't worked very well for me. I think this is because organizations vary pretty dramatically in how they develop software. For instance, I generally work with experienced software teams, and if you try to use regression equations created for less-experienced teams, the equations probably aren't going to work very well. Also, in general, larger companies tend to have more overhead than smaller companies. This could be another complicating factor.

Despite my concerns, if you're really interested in cost estimating using equations like this, I encourage you to investigate them. The International Software Benchmarking Standards Group (ISBSG) is one organization that publishes these type of equations. Because I only have older copies of their data, it's possible that they have newer equations based on newer data. I found their equations in one of their publications titled, "Practical Project Estimation." You can find more information at isbsg.com.

I believe Construx *Estimate* also uses similar equations, though I don't know if they publish the equations that they use. See construx.com/Resources/Construx_Estimate for more information.

Summary

While Function Points were originally invented by Alan Albrecht in 1979, they remain a mystery to most developers today. That's unfortunate, because once you know the functional size of an application and have some historical data for your development teams, you can make accurate time/cost estimates.

A recap of Function Point Analysis process shows that Function Points:

- Are measured from the user's perspective (not a developer's perspective)
- Are independent of the technology used to develop the application
- Are low cost, adding less than a 1% overhead to your process
- Are repeatable, as studies have shown that certified function point counters can come within 10% of each other
- Are "use case friendly," because counting function points typically corresponds to processes defined in use cases (as well as user stories)

Using Function Point Analysis helps you more accurately estimate:

- Project cost
- Project duration
- Optimum project staffing size

An accurate counting of function points leads to a wealth of valuable statistics that can be used to improve the development process, including:

- Number of developer hours per FP
- Number of total man-hours per FP
- Cost per FP
- Number of FPs per month/week/day
- Number of bugs/defects per FP
- Number of bug/defect hours per FP
- Productivity increases (or decreases) due to technology changes

These metrics, and others like them, can be used as part of the feedback loop to improve your software development lifecycle.

Lesson 2

How I Estimate Software Projects Using FPA

Introduction

Lesson 1 provided an overview of how you can use Function Point Analysis (FPA) to determine the functional size of a software application. As I mentioned when writing about the Value Adjustment Factor (VAF) in that lesson, there is a big difference between the *size* of an application and its *cost*. Two applications with the same size can have very different costs.

In this tutorial I'm going to try to address how I estimate the cost of software development projects. There are several ways to estimate the time and cost of software development projects, including:

1. Using a Function Point count with a known/historical team project speed (Hours/FP)
2. Using a Function Point count with regression equations
3. Using a Function Point count with various estimating tools
4. Work Breakdown Structure (WBS)
5. "Yesterday's Weather"

In this tutorial I'll spend the majority of my time talking about the first approach because it's what I believe in most. I'll also discuss the WBS and Yesterday's Weather approaches, because I'm most familiar with them.

Before I begin, it's important to know that one major assumption of this tutorial is that you know what FPA is. If you don't already know about FPA, please read Lesson 1 so that the following discussion will make sense.

Background

For the purposes of this discussion I'm going to assume that the software application I'll be estimating and the software development team I'm going to be estimating have the following characteristics:

1. A formal requirements specification has been written by an experienced business analyst that I trust. The specification isn't "heavy"; it's a fairly light document with a style that might be called "light use cases" or "slightly heavy user stories." The spec also includes a database design, and screen prototypes for at least the most difficult UI elements.
2. The spec was written well enough so that with a little additional conversation with the Business Analyst who wrote the spec I was able to count the Function Points in the application with a high degree of confidence. I found the functional size of the application to be 750 FPs.
3. The application is a fairly typical business web application.

4. It is a business-to-business application (not business-to-consumer).
5. It's being written to support only one speaking language (English).
6. The application will need to support less than 1,000 simultaneous users.
7. Availability concerns are average/normal for this company.
8. The application is being written in Java.
9. Except for one person, the developers on the team all have five or more years of experience, and they have all worked together before. There is one new person on the team with two years of Java programming experience, but he used thick-client technologies like Swing and JavaFX, and doesn't have much experience on web applications.
10. The development team is going to use two technologies that are fairly new to them. That being said, some of the team members have explored the technologies on other projects, so they're not starting completely from scratch, and the technologies are well known and established.
11. The testing team members are also familiar, and we have worked together on previous projects.
12. The project sponsor and other "customers" for this application have worked on software development projects previously.
13. We have a repository of past projects that this team has worked on, so we know their average productivity rate (Hours/FP) on different projects.

As you can see from that list, I think all of these variables are important when making an estimate. For example, project specifications vary widely (wildly?) from one business analyst to another. No two development teams are the same. Customers that haven't worked on software development projects before need to be "trained" (for lack of a better term) in how software projects proceed, and what is expected of them at various points. New technologies affect productivity.

Beyond those factors, size of a project is also very important. Small projects are easy and longer projects much harder -- and I don't mean linearly harder. At some point trying to tackle too much all at once makes complexity increase in more of an exponential manner than a linear manner. (If you are familiar with the many delays of projects like Windows Vista, you'll know what I mean. Even Steve Ballmer said at that time that Microsoft will never try to change so many things all at one time ever again.)

Getting back to the requirements specification, if you don't have a spec, you can't really estimate the size, time, and cost of the project, can you? Until you have a spec that has been vetted by all the interested stakeholders, you can't really know if everyone is on the same page, can you? And if you don't know what everyone wants, you can't estimate what you don't know, right?

(Despite that last paragraph, I will show you in Lesson 3 how I estimate projects very early in the software lifecycle. And by early, I mean when I'm meeting with a CIO/CEO/CFO, Project Manager, or Project Owner for the first time. If they can explain to me what the "core" parts of the application are, I can use certain FPA techniques to get the into the right size/time/cost ballpark. But that's as close as I can imagine to estimating a given project without a proper requirements document.)

Estimating using FPA and Hours/FP

Because I've worked with this development team on estimates before, and because we know their productivity rates from previous projects, and because we don't see the two new technologies as being a major risk, I feel comfortable using their productivity rates from previous projects to estimate this project.

For instance, when this team has worked with this form of requirements specification before, their productivity rate on other web development projects has averaged 4.0 hours/FP. On some projects their productivity rate has been as fast as 3.7 Hours/FP, and on others it has been as slow as 4.5 Hours/FP, but 4.0 Hours/FP has been the average. (When you use "Hours/FP," the smaller number means that your team is more productive. Some people prefer to use "FPs/Hour," so that a larger number indicates the team is moving faster.)

Looking at their data, any time new technologies have been added to the mix their overall productivity rate has actually decreased, not increased. When they've used those new technologies on subsequent projects their rate has been closer to the average speed (or even better than average), but there have been times when new technologies didn't work for them, and as a result the projects took longer than average. In summary, on projects where significant new tools have been added the team has generally performed at a slower than average rate.

There's also the matter of a new, relatively inexperienced developer being added to the team. While some people will argue that this will make the team faster, I disagree, especially initially. It has been my experience that adding an inexperienced developer to a team with a certain development speed slows down the team, if only slightly. If the developer succeeds with the team, the team's speed tends to come back to the average, but this takes time, and there's also a chance that the developer won't succeed in his new role.

Choosing a productivity rate

Looking through the team's historical data, knowing that the two new technologies are being used, and knowing that a new, relatively inexperienced team member is being added to the equation, I believe the team will proceed more slowly than their 4.0 hours/FP average and tend more toward their 4.5 hours/FP high. To keep the math simple in this document, I'll assume that they will have an average productivity rate of 4.2 Hours/FP.

Multiplying that rate of 4.2 Hours/FP by the application size of 750 FPs yields a total of 3,150 hours. With four developers on the team (not including the new person), and assuming all four can work on the project simultaneously (a pretty big assumption), I divide the 3,150 hours by four, which leads to 787.5 hours per developer.

> Note: You have to be really careful assuming that you can split a project evenly between a set of developers. Someone coined the phrase, "No matter how hard they try, nine women cannot have a baby in one month." Or in this case, 3,150 developers cannot combine to finish this project in one hour.

Next, on longer projects I never assume a 40-hour work week. When projects are longer you have to account for things like personal time, vacation time, and company overhead. I typically use an availability rate of 35 Hours/Week, so in this case, 787.5 Hours/Person divided by 35 Hours/Week leads to an estimate of 22.5 weeks, which I'll round up to 23 weeks. (Some people like to use availability rates even lower than 35 Hours/Week. Again, this usually depends on what I call, "company overhead," which refers to how much time the developers need to attend company meetings, attend to other projects, etc.)

Factoring in acceptance testing

The productivity rate I used is based on previous projects, and that rate is known to include time for user acceptance testing and delivery of the initial application to its end users. As a result, the estimate of 3,150 hours should be the estimate for the entire project, including user acceptance testing and final product delivery.

For scheduling purposes, a Project Manager (PM) typically wants to know when the product can go into final user acceptance testing and when it can be delivered to the customers. The way I try to answer these questions is to again look at historical data for this team. Every team and every company is different here, that's why I want to use historical data as much as possible.

Looking at the historical data for this team, final acceptance testing on previous projects has taken an average of 25% of the development time. For example, on one previous project the entire development time took 20 weeks. The product was delivered for final acceptance testing at the 15-week mark, or at the 75% mark of the project development time.

Using that same 75% mark for this new project, 75% of 23 weeks is 17.25. Rather than rounding that down, I'll round it up to say that the product should be available for final acceptance testing on the 18th week.

Discussion

I'll discuss "Yesterday's Weather" more in a little while, but I hope you can see that by using FPA and keeping a database of metrics about previous projects, you can use a combination of FPA techniques and Yesterday's Weather to create the estimates I just showed.

If a company doesn't have a database of previous projects like this, all you can do is guess as to their productivity rate and when you'll get to final user acceptance testing. You can use databases of other company's productivity rates, but I've found that those databases usually vary too much to be useful.

Exercise: A more conservative estimate

For some customers I create an estimate range. For instance, if I've never worked with a company before I might say, "Well, I've never seen a team go faster than 2.2 Hours/FP, so you can certainly use that as the fastest estimate." And with a company like this that I've worked with before who has a repository of previous project information, we can look at that data and see that they've never gone slower than 4.5 Hours/FP, so you can use that for a worse-case estimate.

> Note: It's important when I say things like that, that I qualify them a little bit. When I make an estimate for a company, I have to assume that this new Project X is like their previous projects A, B, and C. In fact, I always ask the PM and developers, "Is Project X similar to your other projects in what it requires? Are there many more calculations in this project? Is the architecture different? How about number of users and scalability issues?" If Project X is not like previous projects, you have to be very careful in applying historical productivity rates to the new project.
>
> I know from my own experience that when my programming switched from traditional client/server work to more of a web service architecture, the first project took a bit longer than normal. The architecture took longer to deploy, and it took a while to adjust from making database calls to making web service calls, among other initial issues.

Estimating using Work Breakdown Structure (WBS)

When people don't know about FPA but they're still serious about making estimates, I'll guess that at least 90% of the time they use a Work Breakdown Structure (WBS) process. WBS consists of one or more people sitting down and saying, "Okay, how are we going to get this done?"

With that question in mind they come up with a laundry list of tasks that need to be accomplished to deliver the software application. They create their list in Excel or some other spreadsheet program, with a list of tasks and hours/task that looks like this:

- Setting up Development, Test, and Production environments - 8 hours
- Creating the Login screen - 4 hours
- Creating the Landing Page - 2 days
- Developing the XYZ Page - 12 hours
- etc., etc., etc.

The estimators continue this process, creating a list that is as detailed and thorough as possible. When the list is finished, they then sum up the total number of hours, and that's their estimate.

That sounds perfectly reasonable, right?

WBS accuracy

I have seen studies that show that WBS estimates drastically under-estimate the time and cost required to deliver complete software application. In my own experience, I've seen WBS under-estimate the total time by a factor of 1.25 to 2.50. That is, if someone tells me that they did a WBS estimate and came up with 1,000 hours, the actual answer is between 1,250 and 2,500 hours.

This is consistent with other published data I have seen. For instance, I have a booklet titled, "Practical Project Estimation," and they state:

> "For the projects with effort under-estimated, on average the actual effort was about 60% larger than the estimate."

Another thing that they state which I've found to be true is this:

> "Projects using both techniques (FPA and WBS) get it right or over-estimate slightly."

Personally, as a business owner, I'd rather have someone slightly over-estimate the work than drastically under-estimate it. Of course there are times when you absolutely need an application regardless of cost, but in the other cases where you're trying to decide if you want to go forward with a project, and you don't know if it will cost $250,000 or $500,000, well, that's a big range that can dramatically alter your ROI.

Consistency

One good thing about using WBS is that developers are often very consistent in how they estimate. Some under-estimate by 100%, some by 50%, some over-estimate; but they're usually consistent in this manner. When estimating projects, you can use this to your advantage.

For example, I once worked with a small development team that consistently under-estimated their effort by about 33%. Once I learned this, I could just factor that into their estimates. When doing this, their estimates were very close to my FPA-based estimates.

(Note: I have to confess, I'm a horrible WBS estimator. I usually under-estimate by 50-100%, which I have learned to take into account.)

Use feedback to improve WBS estimates

A good thing about WBS is that if you keep at it, and keep good records of your past estimates and actual results, you will get better. You'll learn what tasks you completely forgot about, and you'll also learn which tasks you under-estimated. There are tools you can use to help developers track these values, or you can just keep them in a spreadsheet.

By continuing to ask for estimates and then tracking them, you'll eventually be able to rely on "Yesterday's Weather" in a manner just like we did with FPA. In FPA we go back and look at productivity rates, but with WBS you can look at a new task and say, "Hmm, this task seems a lot like that task in Project A, and that took 64 hours, so I'll estimate 64 hours for this task."

The WBS estimate

Getting back to the WBS estimate, let's imagine that we worked through this process and we came up with an estimate of 1,600 hours. Given our team of four developers, this equates to 400 hours per developer. At 35 hours per week this is about 11.4 weeks of development effort.

Don't forget acceptance testing

When I first work with new clients I often find that they don't take enough time to account for final user acceptance testing. This even happens in "agile" environments where new code is released to testers constantly. There's always that last big push on "integration testing" that needs to be accounted for. For some development teams that final testing phase may be 33% of the time/cost budget, and for others it may only be 5%. The important thing is that you remember to factor it into your WBS estimate.

For the purposes of this example, as I mentioned in the FPA analysis, final acceptance testing for this team typically takes 25% of the overall time, so if development takes 11.4 weeks, acceptance testing should take an additional 3.8 weeks, and the initial 1,600 hour estimate needs to be increased to 2,133 hours.

Therefore, WBS predicts a total development time of 2,133 hours, or 15.2 weeks.

Yesterday's Weather

When we first learned about eXtreme Programming, the creators of that approach told us we should estimate using "Yesterday's Weather." What this means is that if I'm about to work on a task, I should say, "This new task X feels a lot like that old task C, so I'll estimate whatever time it took for C to be developed."

On a small scale this can work fairly well. On a large scale it can be disastrous. Let me give you an example.

Way back in the late 1990s, my company worked with a company I'll call "Company A." Company A had their own development team, and they wrote software for a vertical industry. However, they were all As/400 RPG programmers, and they needed to become a "web" company in a hurry, so they hired us to (a) help them convert their applications to web applications, and (b) mentor their programmers. All of that seemed to go well, and after a few years the final cost for our consulting services was about $1.2M. (I don't remember the exact figure, but that's in the ballpark.)

Then in 2001 we stumbled onto a similar situation. We met Company B, who had their own developers, and needed to convert their old (but large) application from an old technology to a new technology. When we bid on the project we tried to use Yesterday's Weather, so we said, "This project for the new Customer B feels a lot like the project we did for Customer A, so we'll bid $1.2M."

I started to submit my bid like that, but then got nervous. I thought, "No, there are some aspects of this new project that are going to be harder than the previous project." As time went on I increased my estimate, then increased some more, and then some more, finally arriving at $1.8M.

Our company won the project, but sadly even this estimate fell short of the actual value. What I didn't fully understand at the time was that we were going to be trying to take the best of two different applications and try to merge them into one. We also didn't know that the Domain Experts from this company would be *much* more particular about what the UI looked like. For a long time we also didn't get the programming assistance from the client that we thought we'd be getting.

So one moral of the story is that I think you can use Yesterday's Weather to estimate small tasks, but you can't use it to estimate larger tasks.

Another moral of the story really applies to all three estimating approaches: You really have to understand your client before you can give a responsible estimate. Some clients don't care too much about what the UI looks like, and others can be like Steve Jobs, and they care about the UI down to the pixel level. Both clients are fine, but the second one is much more expensive, and you need to factor that in when making your estimate.

Yesterday's Weather estimate

Because I don't believe you can responsibly use Yesterday's Weather to make large estimates, I decline to make an estimate here. :)

Other estimating techniques

Before moving on to the summary, I want to mention that there are estimating tools and databases available to you. All of these assume that you have an accurate Function Point count of the application to be estimated.

Construx Estimate

When I first dug into the estimating world, I quickly came across Steve McConnell. Long story short, he wrote a famous book named *Code Complete*, created a company named Construx, and his company created an application named *Estimate*. This tool takes Function Points or Lines of Code as input, and along with other VAF-like inputs, helps you to create time/cost estimates.

If you'll look into Estimate, you'll see that it uses a variety of techniques and algorithms to help you estimate your projects. One of these algorithms is named Cocomo II. I mention this because when you start digging into estimating software, you'll often see this name. I encourage you to read more about Cocomo II at this link on Wikipedia.

You can learn more about Estimate at this URL: construx.com/Resources/Construx_Estimate/

ISBSG

An organization named, "International Software Benchmarking Standards Group, Ltd.," or ISBSG, does what their name implies, creating and sharing benchmarks for people who create software.

I haven't used their data since I sold my consulting firm in 2007, but at that time they had a repository of applications that had been created using a variety of different technologies. The applications that were created were written by all sorts of different firms, and for different industries. At that time they used to sell that database/repository, and they also had an excellent publication titled, "Practical Project Estimation," which was based on that repository and other research. If estimating is important to your life, I highly recommend these products.

You can find our more information at this URL: http://www.isbsg.com/

(If you have a hard time remembering their acronym, my FPA instructor used to call their acronym, "Ice bags," which amazingly I have never forgotten.)

Estimating summary

At this time the summary of our estimates looks like this:

Technique	Hours Estimated
FPA and Hours/FP	3,150
WBS	2,133
Yesterday's Weather	N/A

The obvious question is, "Which estimate is right?"

The answer is that in the best case, you won't know until the project is complete. In the worst case, the project may change so much before it's delivered that you'll never know which answer was "more correct." (Many organizations track changes, and hopefully the effect of those changes, so hopefully you will have some idea of which estimate was more correct.)

That being said, let me share my final thoughts on this subject.

Estimating using FPA and Hours/FP

In practice, if I know the development teams and have historical data to work with, I've found that the FPA estimates based on Hours/FP are the most accurate. As I mentioned in the Preface to this book, when my business had to bid projects on a fixed-price basis, this was the only way we could proceed.

The hardest parts about estimating using an FPA-based approach are:

- Trying to estimate for new teams you know nothing about. If you don't have some information on how this particular team has worked in the past, all you can do are use industry averages, or your experiences with other teams.

- Trying to handle the effects of multiple variables changing, such as new technologies, new developers, a new project manager, etc.

When situations like this occur, all you can do is use your best judgment, and then keep an eye on the project to see if the project deliverables are in line with your initial estimate. But then again, this problem isn't any scarier than trying to provide the same estimate using WBS, is it?

Estimating using WBS

As mentioned earlier, I've found that using the WBS approach almost always underestimates the time/cost estimate, so you need to be aware of this. Hopefully you're familiar with the people who are making the estimate, so you can multiply their estimate by an appropriate multiplier to make the estimate correct.

Estimating using both FPA and WBS

My best advice is that you should estimate using both FPA and WBS. For example, I worked with one team where the FPA estimates were typically a little too high, maybe 5-10% too high, and the WBS estimates from the developers were almost always too low by about 40%.

On the first project we did together, we saw that my FPA estimate was higher, so we went with it, and indeed it was a little too high, but it was much closer than the WBS estimate.

On the second project we did together, we saw that my FPA estimate was higher than their estimate by about 35%, so we again went with the FPA estimate. As it turns out, the FPA estimate was extremely accurate this time. Their WBS estimate was closer to the FPA estimate because they realized some things they under-estimated in the first project, but they were still a little too low.

I'll skip all of the other projects we did together except to point out one more. On this particular project their WBS estimate was *higher* than my FPA estimate, which was very unusual. It turns out that there were some extenuating factors on this project that I didn't know about, so I had no way to account for them. But because we were using both estimating procedures we were able to see something unusual on this effort. For this bid we took their estimate and increased it by 35%, and in the end that estimate was very accurate.

Limitations

Because estimating is a hard process -- and an important process -- I feel the need to express some of my own limitations regarding estimating software development projects.

First, I've been working with computer programming in one form or another since 1987, but really got serious about the craft in 1993. I generally worked by myself or with 1-2 other people until about 1999. In my own programming work I have used languages like Fortran, C, C++, Java, Ruby, Scala, and a few others.

Second, from 1999 through 2010 I typically worked with experienced developers in a small consulting environment. I have worked with a few inexperienced developers straight out of school, but those were the exception and not the rule.

Third, I have worked with FPA and estimating the time/cost of software projects based on FPA since 2004. Every estimate I did until that time was based on WBS or Yesterday's Weather. I was a Certified Function Point Specialist (CFPS) from 2004 to 2007, when I let that certification expire. (CFPSs are required to pass a certification exam every three years, and once I knew what I was doing, it wasn't important to me to be certified.)

The largest development team I've worked with has been a team that varied from 8-12 developers, one project sponsor ("gold owner"), two project managers, and four domain experts ("goal donors"). We worked on one project for about five years, with a little bit of turnover in staff along the way. The code on this project was written in Java, and the last time I counted it,

the project had over 400,000 lines of code (in addition to a *lot* of JAR files from various open source projects).

Finally, I've found that the techniques I've shown in this tutorial work well on database-driven business applications, but I personally don't feel qualified to estimate systems-development projects or real-time applications. As an example of this, my degree is in Aerospace Engineering, and my first programming jobs were in that industry. Most of those programming efforts were scientific/engineering in nature, and FPA does not work in that setting.

References

The following references were used in the creation of this tutorial:

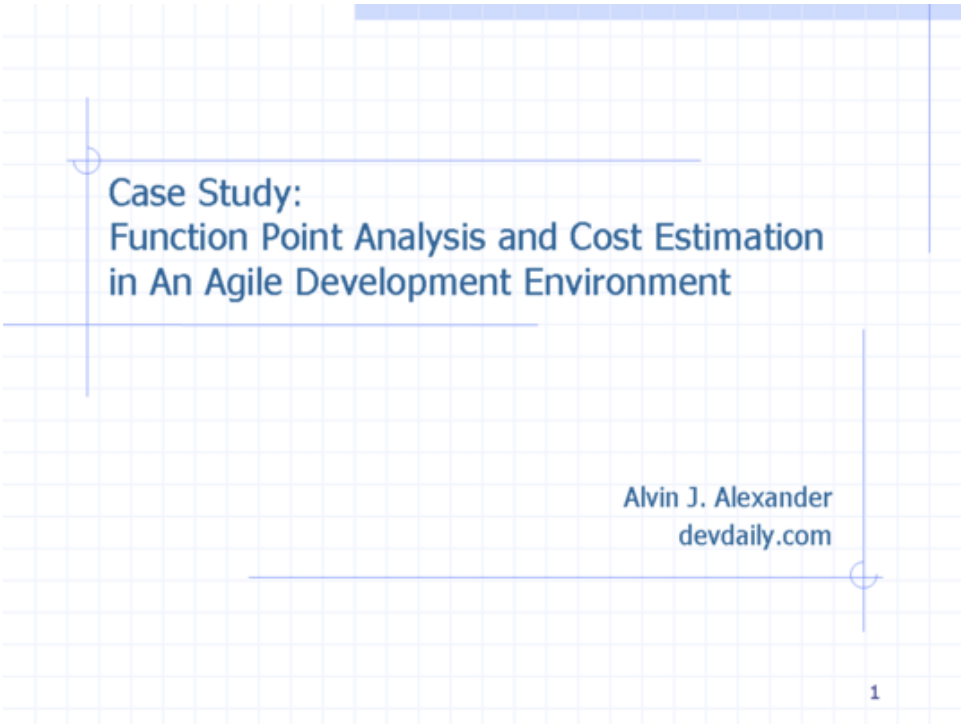
- Training material from the International Function Point User's Group (IFPUG)
- Data from the International Software Benchmarking Standards Group (ISBSG)

Lesson 3

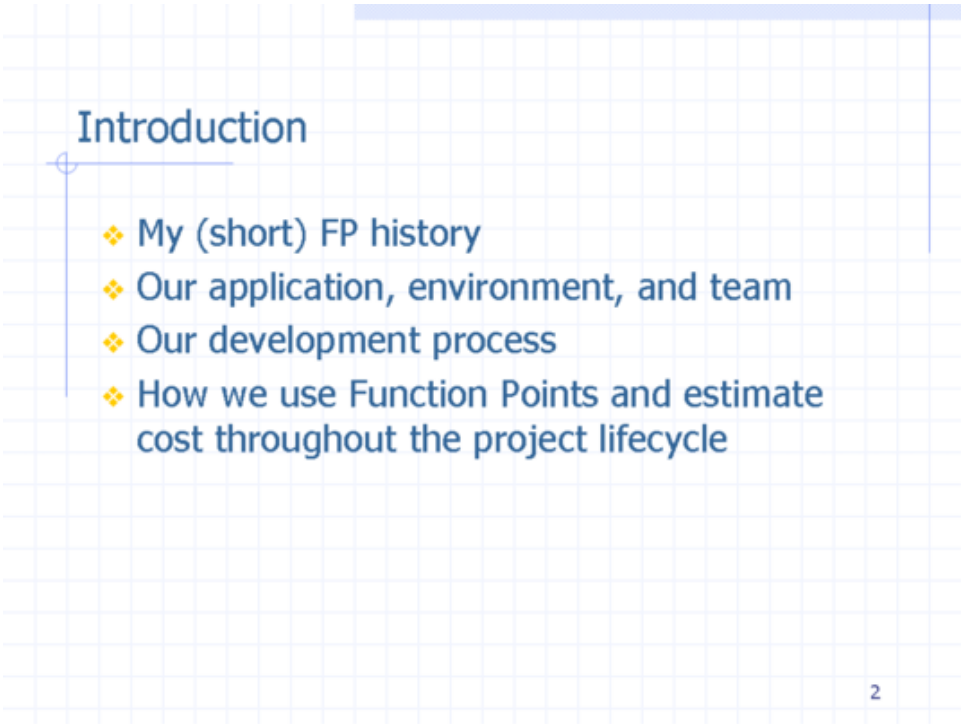
Cost Estimating in an Agile Development Environment

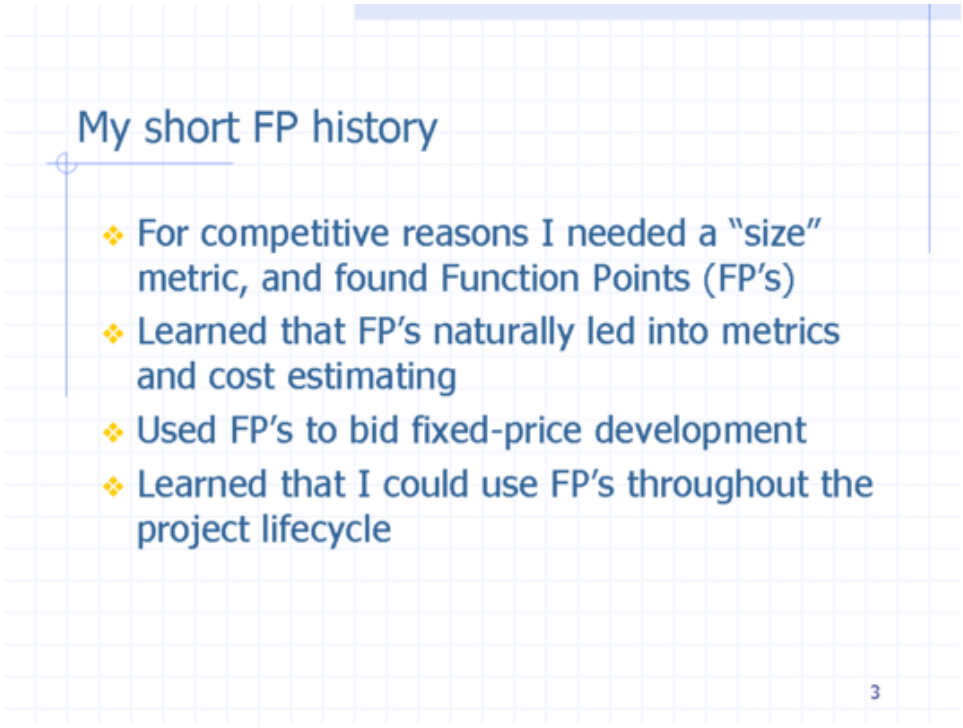
In this discussion I'll use the slides I gave at a speech several years ago to describe how you can use Function Point Analysis (FPA) techniques in an "agile" development environment.

This talk was primarily based on a multi-million dollar software project that I worked on for five years. As a little bit of background information, at some points we had as many as 12 developers working on the project, and as few as two initially. We had two project managers (myself, and a project manager who worked for my client), one project owner who had never experienced a large software development project, and four "domain experts" who contributed to various parts of the project.



I've already written about my FPA background in the first two lessons of this book, 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 quickly skip over them …

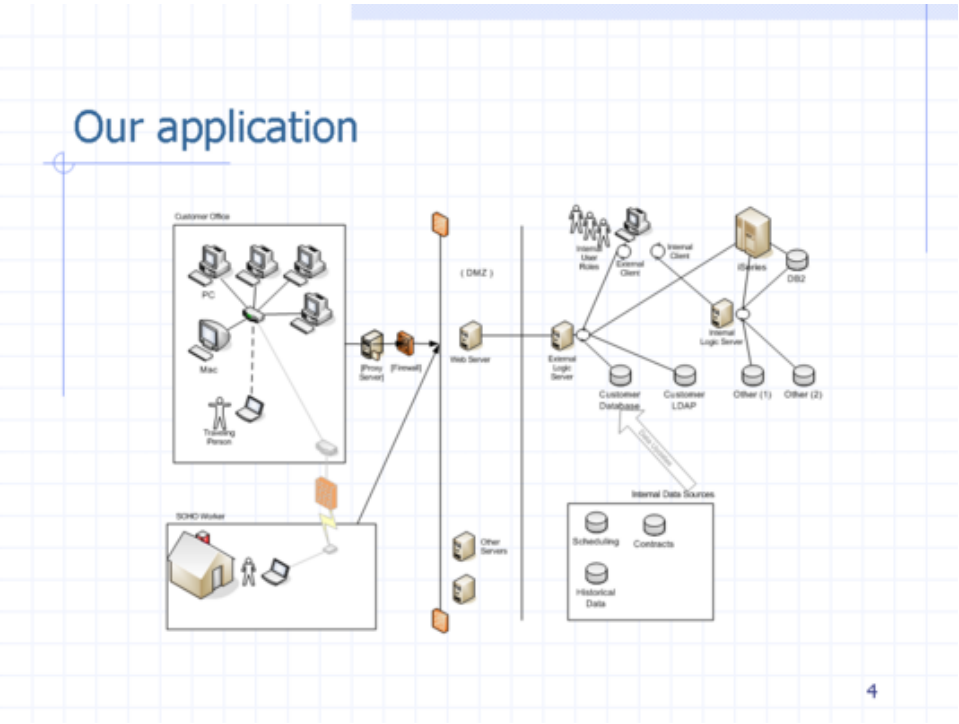




Back in 2002 my company was hired to build a fairly large software application. The application was for one of the largest magazine printing companies in the United States. The application itself was very complex, but it was made even more complicated by the desire to give customers access to almost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

The goal at the time was to be a little like Dell Computer, 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.

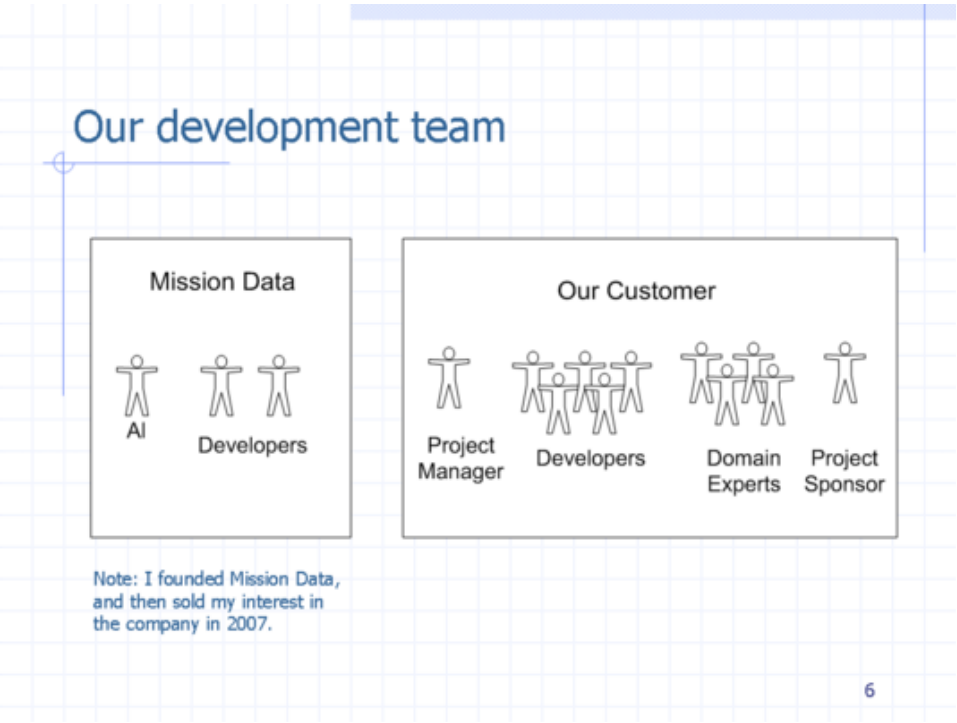
This doesn't sound too bad, just build a web application, right? Unfortunately, no. Remember that this was started in the year 2002, so technology wasn't anywhere near where it is today. We also had to do create some very complicated graphical software, basically letting users create and lay out their 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.



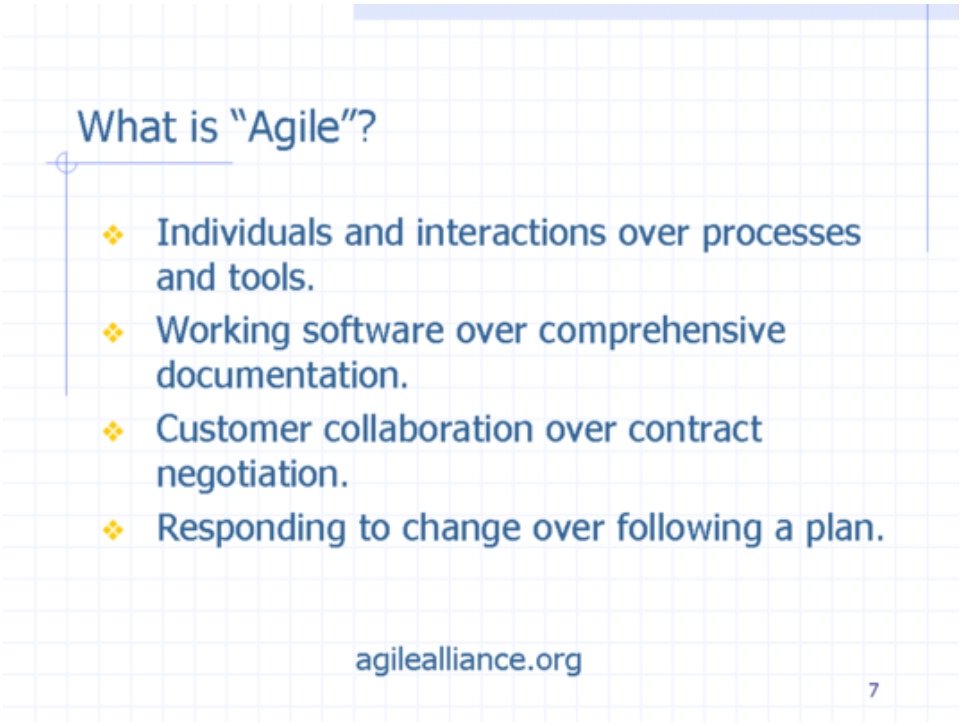
To continue to raise the degree of difficulty on this project, we agreed to train and mentor our client's programmers as we created the application. Our hope was that my company would write 100% of Phase 1 of the project, 75% of Phase 2, 50% of Phase 3, and 25% of Phase 4 as their programmers were brought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

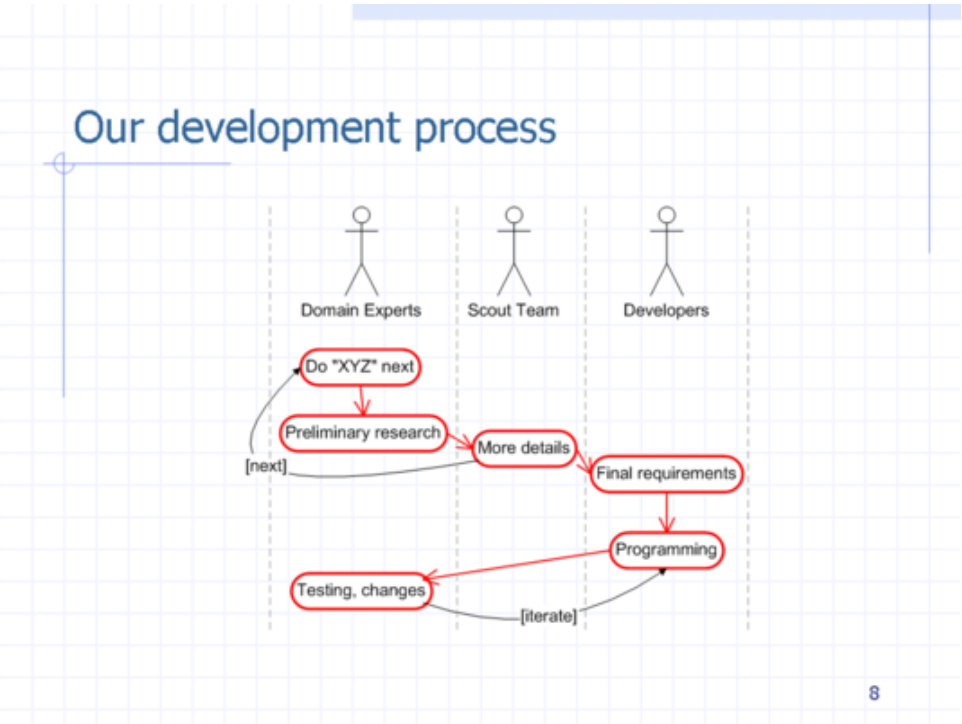


By now most people have their own ideas of what “agile” means, so I’ll just share these bullet points. I’ll also say that we were not very “agile” during the fixed-price projects, but became much more agile as the project went along.



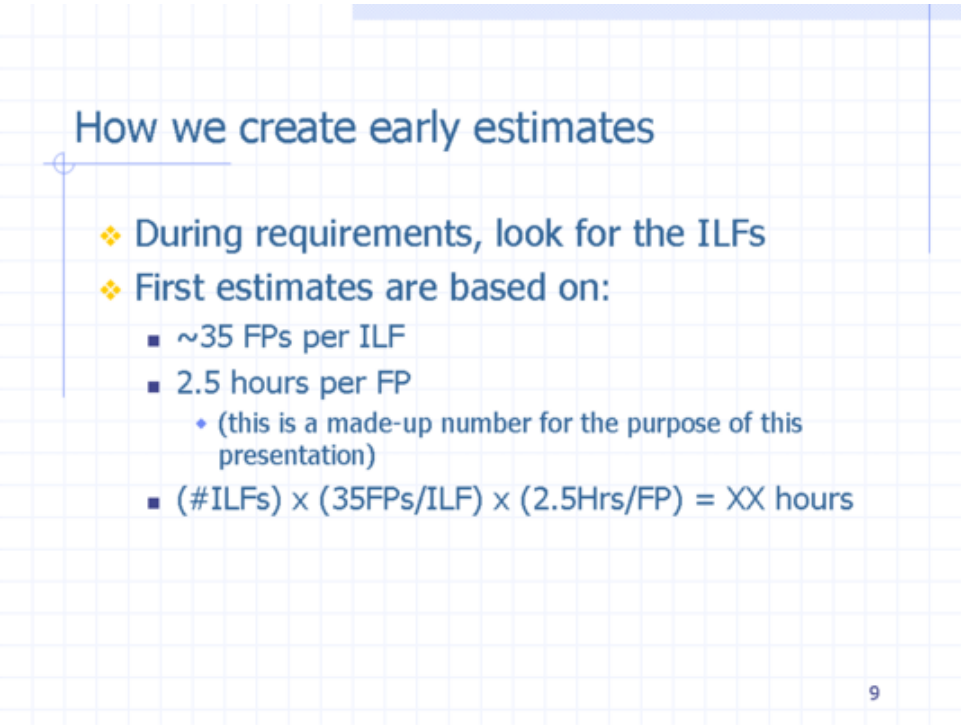
Once we switched back to T&M projects and attempted to become as agile as possible, our development process looked like this slide. I managed the project along with my client's project manager, and together we were essentially the "Scout Team" shown in this figure. We would figure out which parts of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



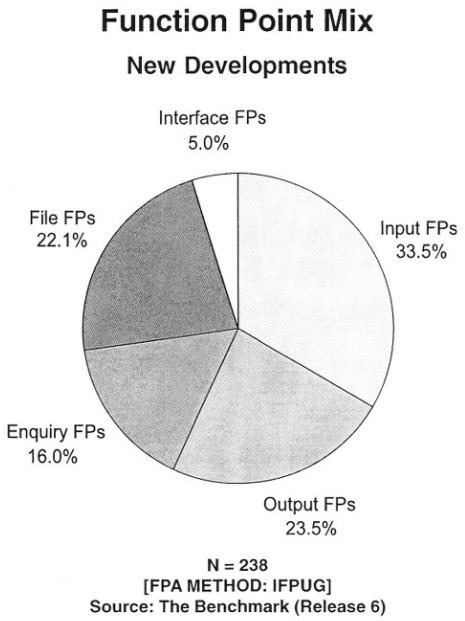
Each task we tackled like this usually consisted of somewhere between 300 to 1,000 man-hours of development time. I continued to have my developers create their own estimates, which included time for both (a) programming and (b) user acceptance testing. Actually, what we really did was use WBS to estimate 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.



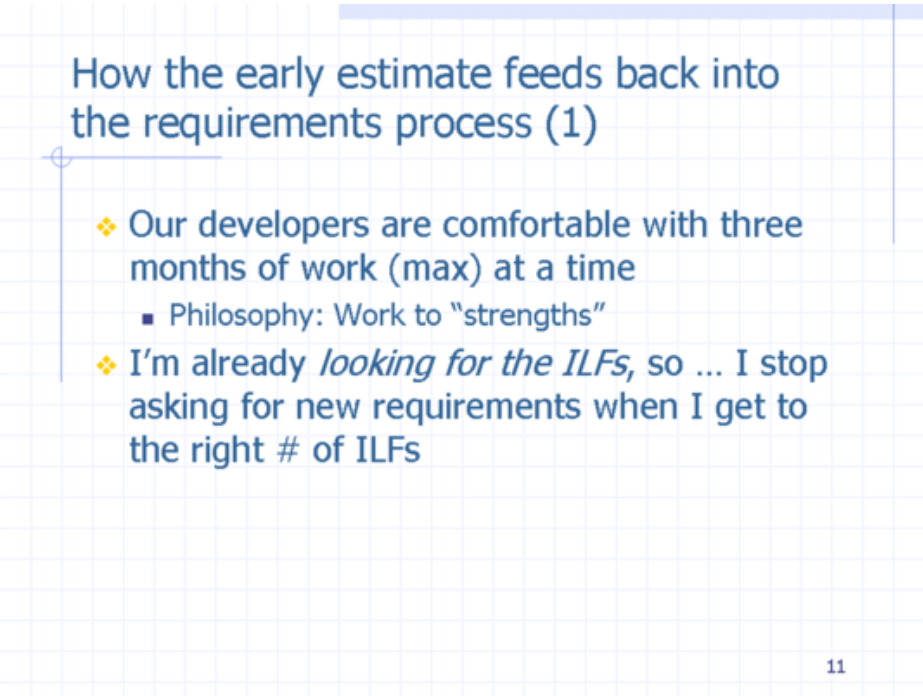
This magic "35" number comes from ISBSG historical data. It turns out that if you count Function points on a *lot* of projects and then look at how the data breaks down when the projects are completed, on average in most "normal" projects there are 35 FPs/ILF. Of course this isn't always true, but for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

You can get to the 35 FPs/ILF value from the following chart, and by also knowing that most ILFs are low to medium in complexity, with a mean score of 7.4 FPs.



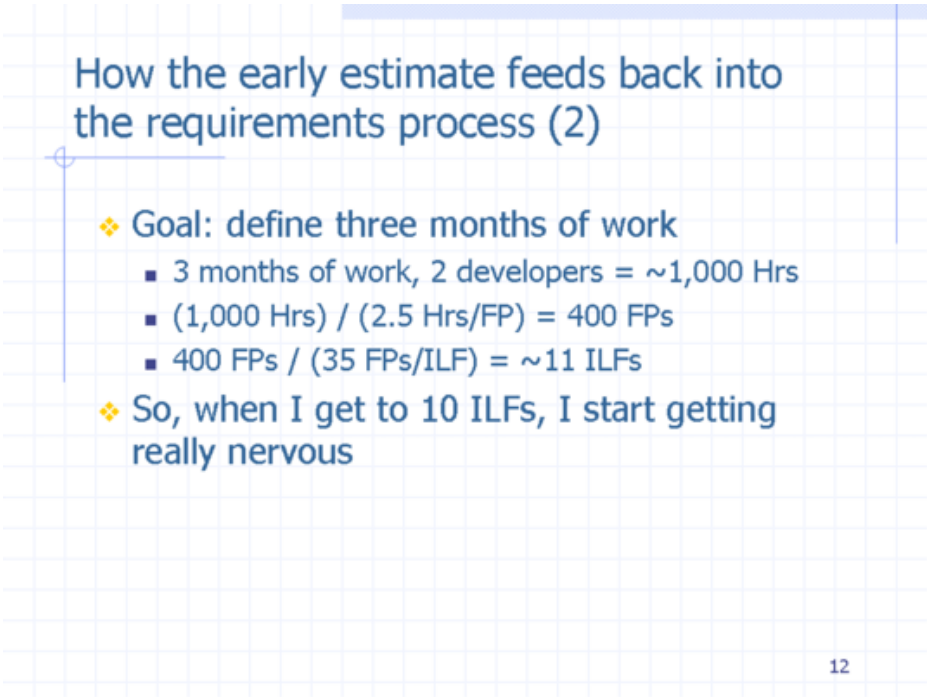
(This image comes from an ISBSG booklet titled, "Practical Project Estimation.")

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.

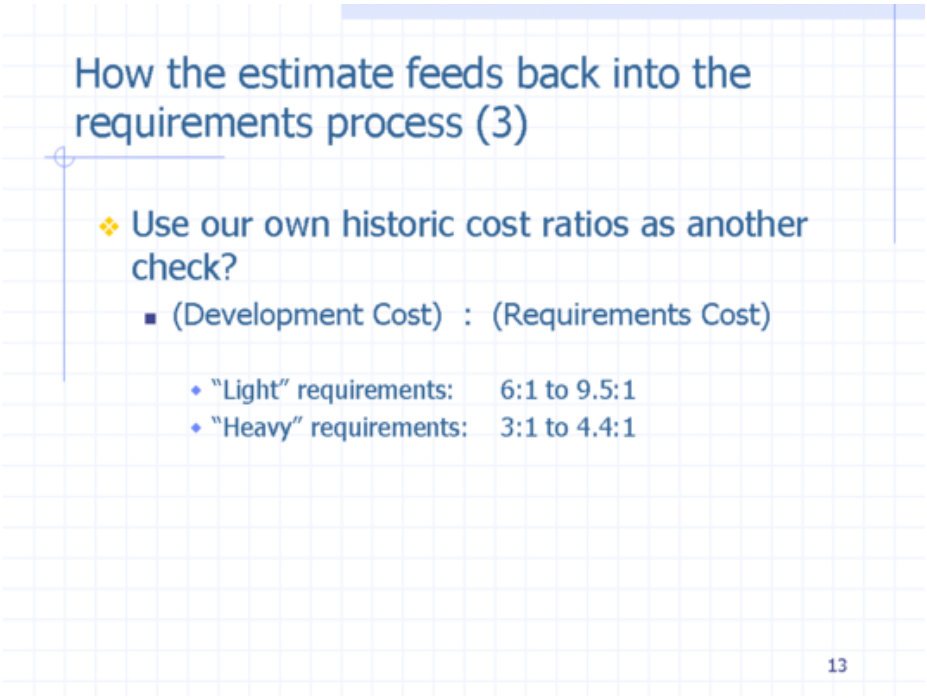


Another thing we learned as time went on is that there were relationships between (a) how long it took us to gather requirements for each task and (b) how long it took to do the programming work for each phase.

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?"

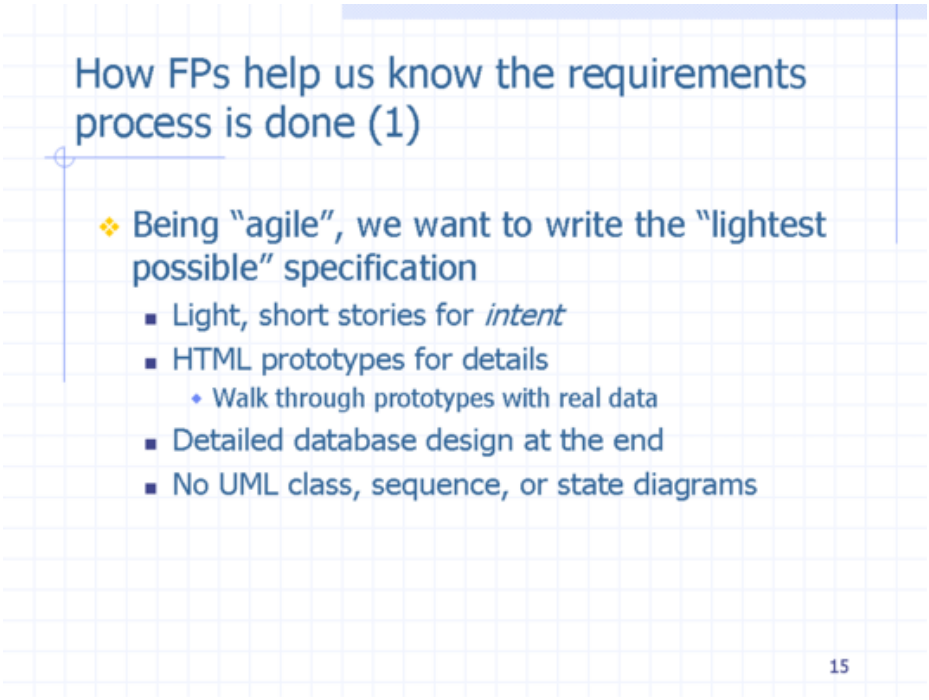


FPA also gave me a way to know when the requirements gathering process for each task was complete. I'll get into that in just a moment, but for now it's important to mention what we thought "agile" meant. As the next slide shows, we wrote short "User Stories" rather than Use Cases, and within those documents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

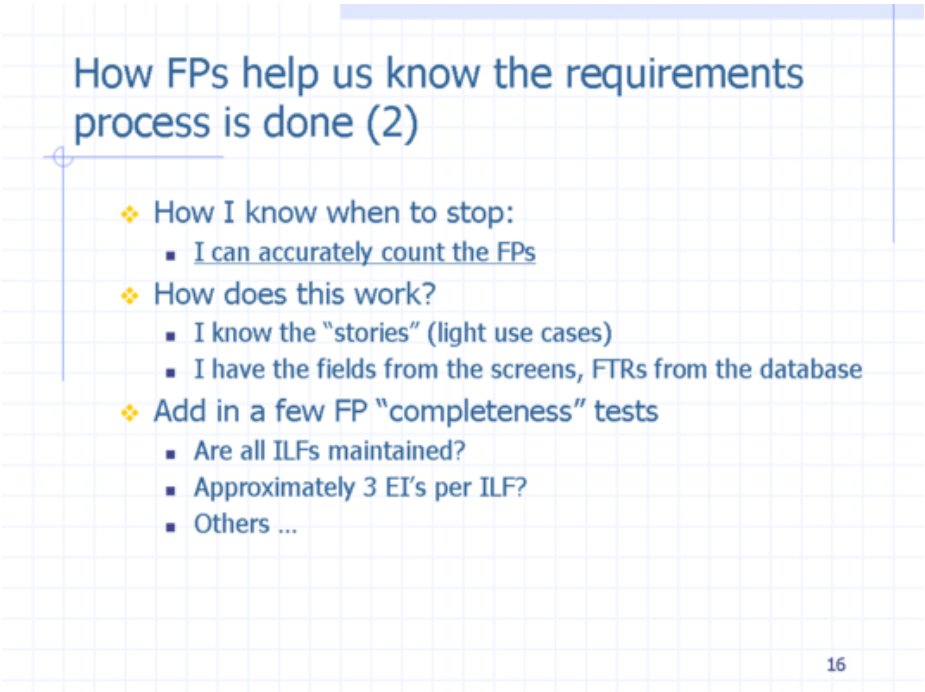
Finally, as you might guess, there were no UML diagrams anywhere in the process. Actually, I take that back, slightly: I am famous for drawing stick-figure diagrams showing "actors" and "use cases," so I'm sure there were a few of those on whiteboards from time to time.



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.

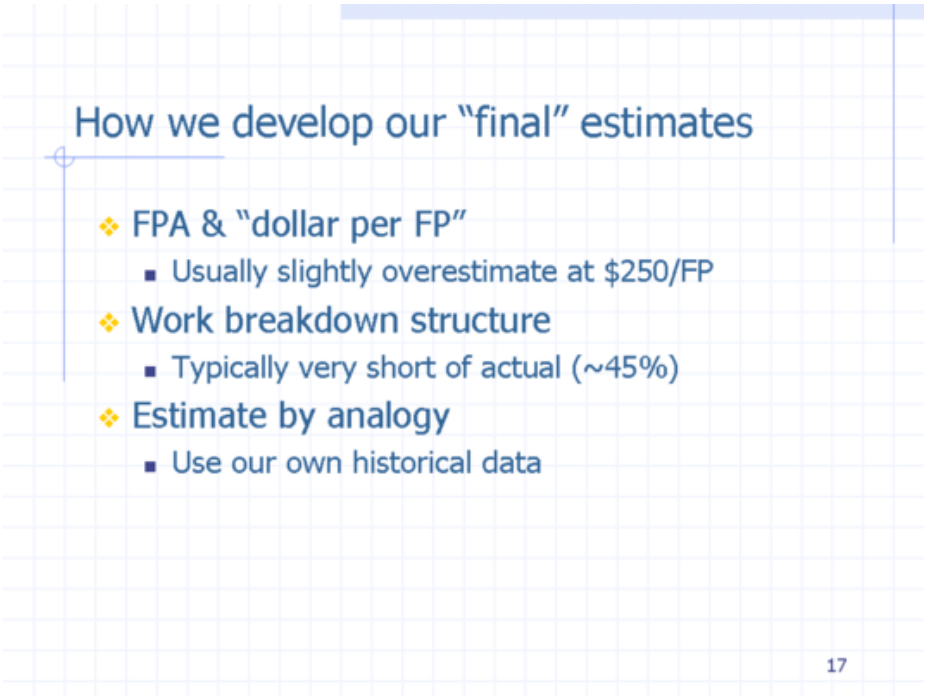
Another great FPA test you can perform is to make sure all the ILFs -- and each new field in the ILFs -- are "maintained", i.e., that there are add, edit, and delete processes for each field. As you know from Lesson 1, an *ILF* is an Internal Logical File, and by definition, this means that this "file" (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 lesson. Those items include bug fixes, conversions, creativity, and all of the VAFs/GSCs discussed in the first lesson.

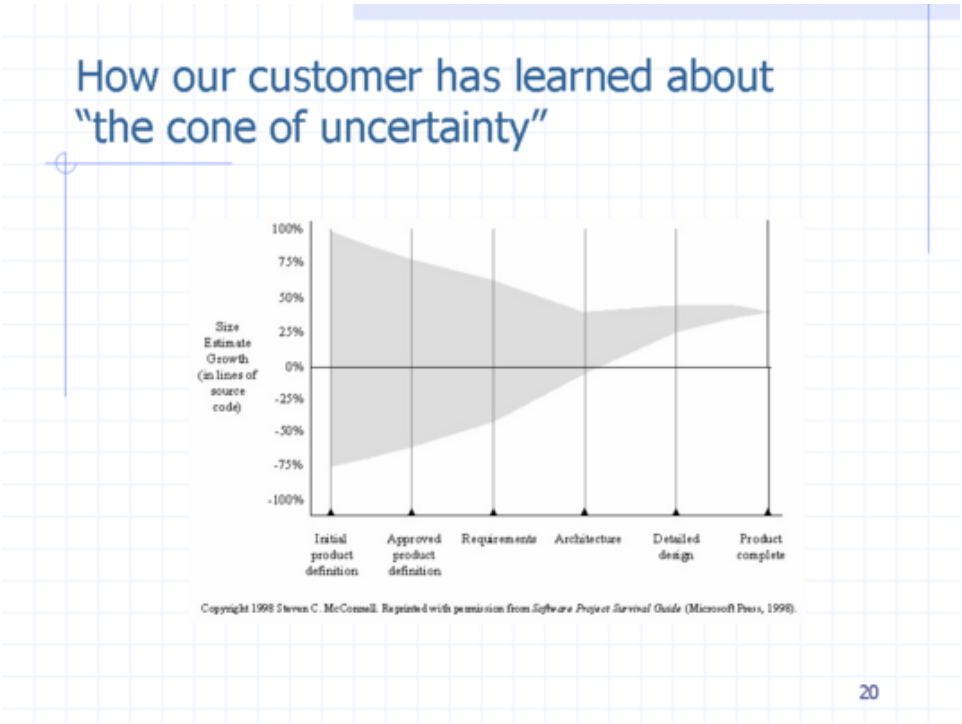
As a few final points about our agile process, we used the Twiki wiki for a lot of our developer documentation, and our developers liked to use the XPTracker plugin with it. (I don't know if this plugin is available any longer.)

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.)

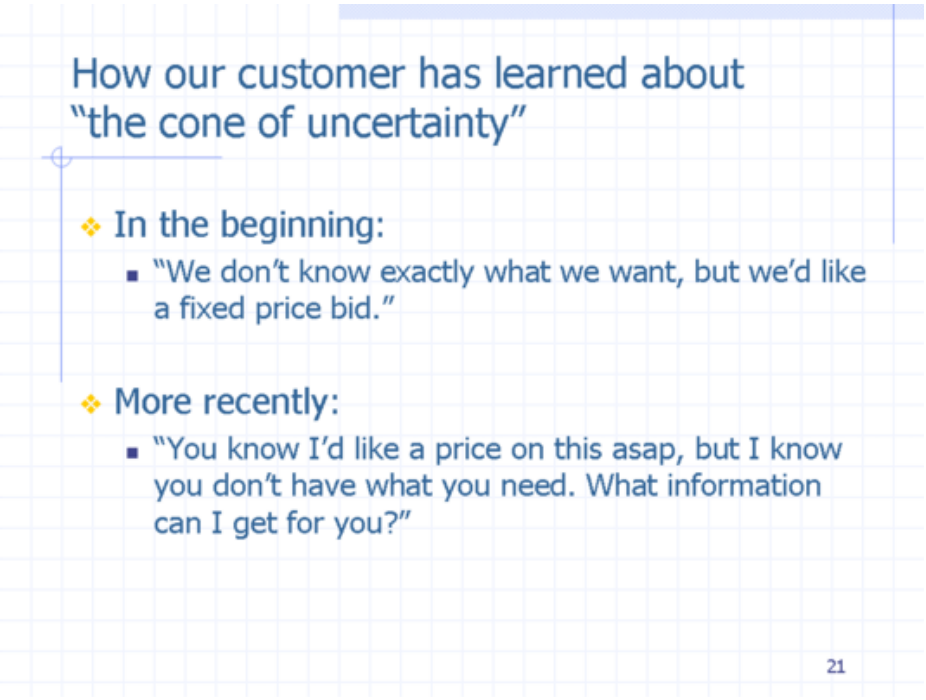
As time went on, we also found that there was typically a 3:1 ratio between development time and user acceptance testing time; not in man-hours, but in calendar time. For instance, if a project task took nine weeks to develop, it typically took three weeks for acceptance testing. This didn't seem to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

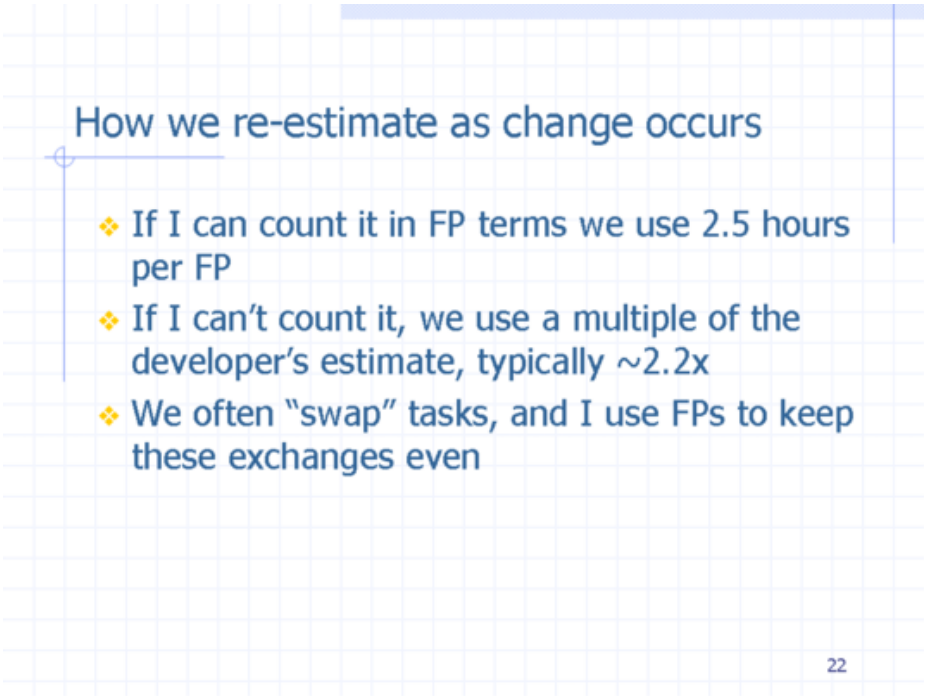


In our case we were fortunate that our client grew with us in this process. As the next slide shows, they initially thought that they wanted fixed-price projects, and couldn't understand why they were so hard to bid, but as time went on they learned that we needed more information to make any soft of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



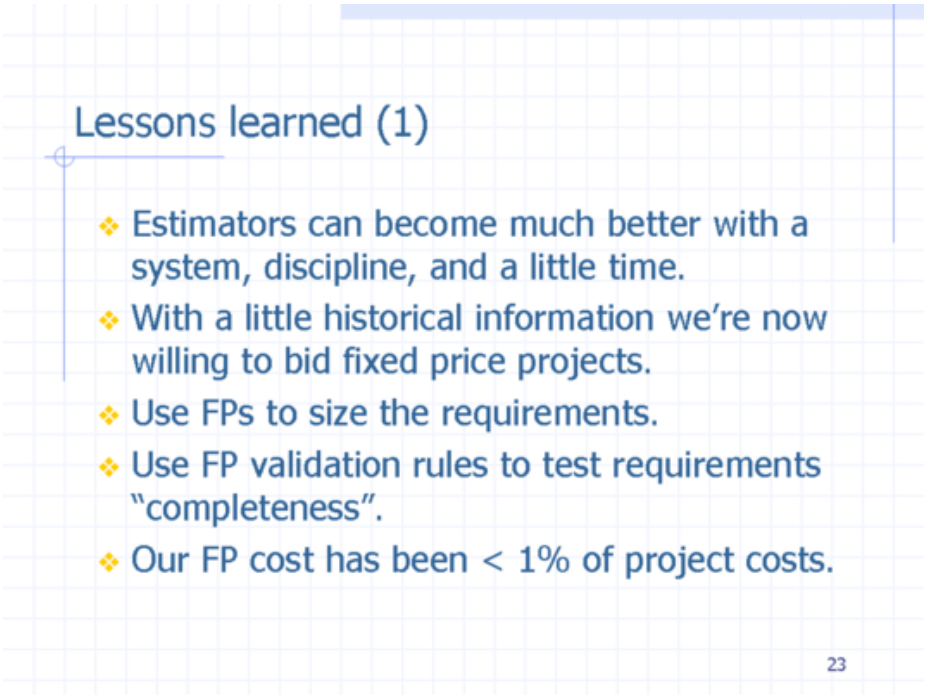
In software projects there is always change, and when there's change, there's also a need to re-estimate the cost/effort. When this happens on our project, I use the same techniques to estimate the change. If I can count the FPs in the change, I estimate 2.5 Hours/FP. If for some reason I can't count 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.

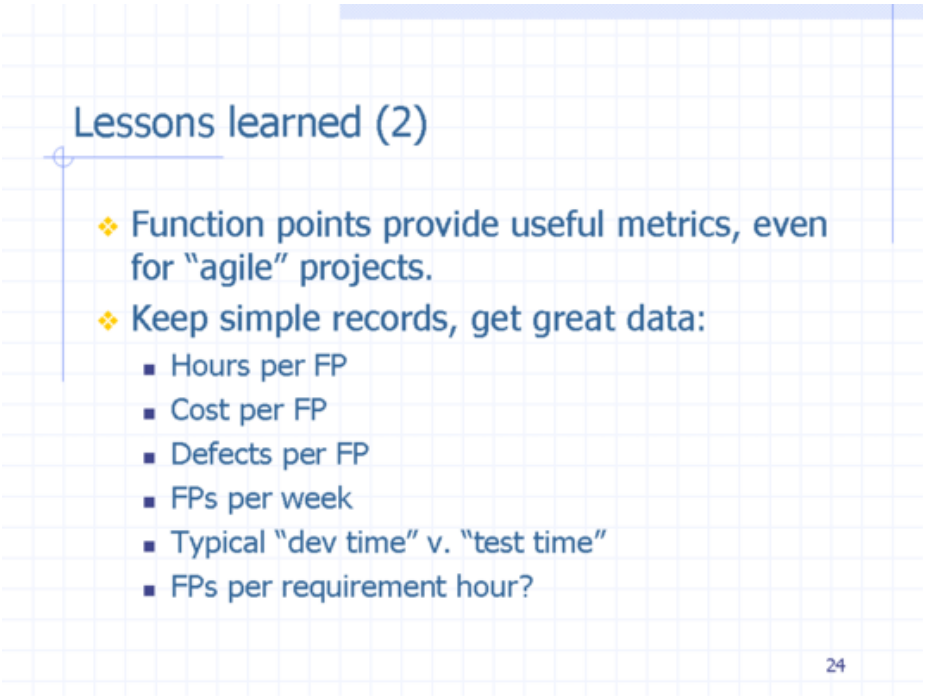


In conclusion, the following three slides show our “Lessons Learned” from using FPA in an agile development environment.

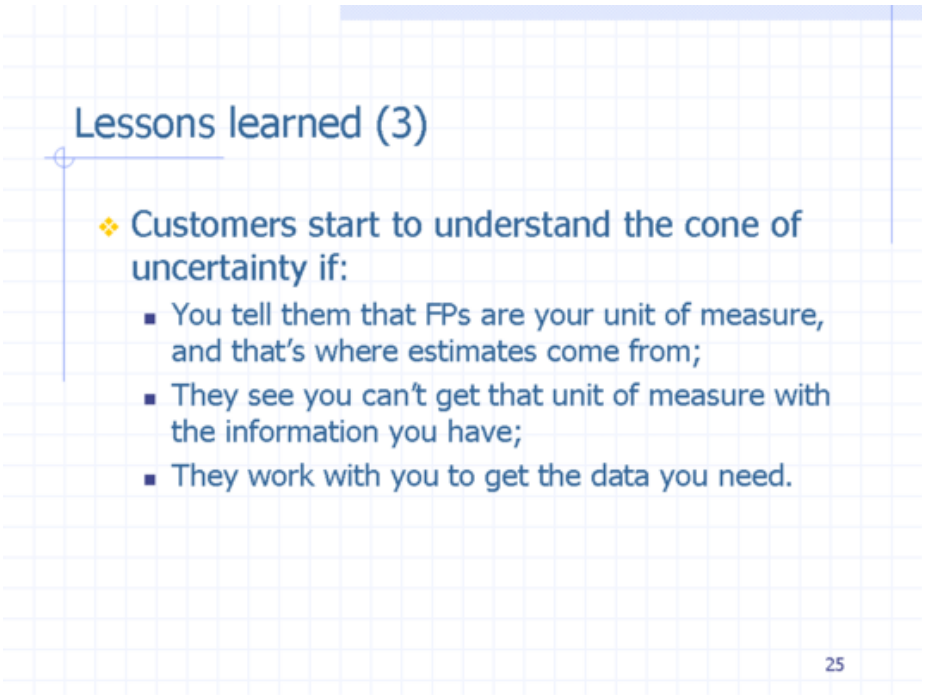
I’ve written about each bullet point on this slide, except for the last bullet point, which I covered in Lesson 1; the cost to use FPA is very small, typically less than 1% of your overall project cost. Once you get good at counting FPs, you can do so extremely fast.



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.



(the end)

Alvin Alexander
alvinalexander.com