TIMING IS ALMOST EVERYTHING

12 Steps to Executive Success in Software Management

Roland Racko

Copyright

Roland.Racko@timingisalmosteverything.com

First Edition

Cover design: Roland Racko
Interior design: Roland Racko

www.timingisalmosteverything.com

Contents

Acknowledgment

Special thanks to Ivar Jacobson, the driving force behind SEMAT Essence. The more casual conversations we've had over the last two decades inspired the mindset behind this book. The heated debates we've had have sharpened us both – who could ask for more than that?

Bushels of props to some people who have unwittingly contributed to this book by their sterling examples of what it means to be relentless in the pursuit of software excellence – Ed Yourdon, Tom DeMarco, Meilir Page-Jones, Tom Plum, Steve Weiss, P.J. Plauger. That includes an additional special fist bump to P.J. who counseled, "By the way, try not to write a book that fills a much-needed vacuum."

A very warm and heartfelt gratitude to hundreds and hundreds of public seminar students, magazine column readers, software managers and clients whose frustrations and laments have taught me more about what works and what doesn't than any studying I could have done on my own.

Finally, huge thanks go to Ivy Green whose support made timely publication of this book possible.

Introduction

ATTENTION: CEOs, start-up entrepreneurs, executives.

Is it a burning goal of yours to have a successful, world-class, software project?

Could you also be one of those senior executives who feel that they report to their computers, rather than their computers reporting to them?

Or are you relaxed because information technology projects are running so smoothly that you feel like nobody needs your guidance anymore? If you are in this relaxed group, then this book is likely not for you. You can stop reading, put it down, and gloat about your successes to the person sitting next to you on the airplane.

Since you are still reading, let's speculate that you feel a certain frustration regarding information technology and/or you are curious to know more. And, let us further speculate, that you would like to enhance the influence and control you have over the business value and success of software construction in your company.

GOAL OF THE BOOK

All the equipment you need to enhance that influence and control is in this book. There are 12 precise executive steps. These steps change the timing and influence of your power. This book includes tips, tricks and tools - some old, some new, about the "how" to exert that influence. The book details give you:

- A set of "management-by-query" style of non-geek questions that you can ask yourself and your software team. They are queries that will help guide the team to deliver better business value from your software

systems – queries which will engage, inspire and enliven those who report to you

- Critical tactical timing guidance that gives you the best leverage for communicating, adopting and transitioning to the ideas behind each management query

- Jargon-free monitoring tools that deliver precision feedback about your software system's progress that you likely have not yet experienced

- An objective way to diagnose trouble areas in your software development process and ways to improve performance in those areas

- A framework for understanding what has previously prevented your influence from producing the maximum business value from your software systems.

In short, these pages show the non-technical executive (someone having approval control over software development) a new way to corral the software development process in a manner that enhances the business value of the product and the project.

How?

By building on insights you already possess: life experience. The insights used here have always been part of your experience, even though that experience, has been mostly, if not entirely, non-technical. The book conceptually divides the assertion of executive influence in 12 steps, each of which is strategically-timed. Each step draws on those life insights.

You are not simply reading a handbook on how a largely non-technical executive can successfully manage software teams using strategic timing, but you are also about to learn how to repurpose your best personal insights to achieve maximum business value from your project.

The guiding pages of this book show how to time the following software team actions:

- implementing a “software architecture” as a framework and context to discuss proposed solutions,
- establishing "software floor plans" to describe and critique the software architecture of those proposed solutions,
- creating common terminology between users, stakeholders and developers,
- constructing software components in an ordered way derived from the software floor plans,
- optimizing the context in which to begin the full funding of the project.

These team actions are designed to bridge and reconcile disparate definitions of success that can exist between executive and team. Coalescence of success definitions enhances delivered business value of the project.

AUDIENCE OF THIS BOOK

The principal audience is senior executives, at any level, frustrated by company past experiences with software projects.

Entrepreneurs of start-ups, who suddenly find themselves unexpectedly pummeled by the need for elaborate software systems just to start the business, will also find this book exceptionally valuable. Start-ups often attract passionate, yet unseasoned software talent. The "old-hat" ideas get shortchanged in start-ups because the lack of prior real development experience has not yet solidified the usefulness of "old-hat" techniques. Reviewing these will avoid some horrendous start-up goofs that would otherwise delay entry into the window of opportunity

.

Is there anybody who should put this book down and leave themselves out? Yes.

If you are a theoretician, methodologist, agile fanatic or sensitive to politically incorrect software language, you should run screaming for the exits and remove the battery from your laptop or e-reader before it self-destructs.

INFLUENCE: TIMING AND MAGNITUDE

Influence has two components – timing and magnitude, i.e., *when* the influence is delivered, and the *strength* of that delivery. A secret of this book's ideas is to alter the precise point in time at which the executive exerts influence.

Typically, an executive expresses influence the first moment that executive presents a software team with a problem to be solved with software. The influence is initiatory in its timing and low key in its magnitude. Presuming the software team is a good one, the team first refines the problem statement and then precisely defines a set of computer system requirements which will act as an effective solution for the opportunity. The executive then exercises a confirmatory influence, saying yes, (or maybe no) to the proposed system.

Influence is generally not exerted with commanding strength by the executive until there is a significant problem to be solved. All too often this takes place after the software team has exerted tremendous effort to accomplish the goals of the executive as the software team understands those goals.

As an example, when the system is finally installed, the executive unleashes great, intense influence to correct an often fabulously disappointing situation as the executive discovers the horrendous gaps that exist between what was desired and what was delivered. Despite the considerable energy made available at this point in time by the executive, the system never quite hits the mark, because making major repairs to a fully constructed system is incredibly expensive and draining. Also, such repair can be politically arduous since the budget is typically exhausted.

In an even more futile waste of energy, the disappointment is typically followed by the executives trying to exhort developers to use the latest tools, techniques, research or methodologies to improve the next software system. Usually, the new exhortations fall largely on deaf ears.

What prevents those exhortations from being successful?

Often developers become numb. They become numb when they experienced that the last set of exhortations had insufficient support follow on from the same management team that exhorted them.

Hint: exhortation without thoughtful transition planning and support doesn't work. (We examine ways to correct that failure in step 12.)

This book re-orders the timing point to unleash strong executive influence. Further, it strengthens the confirmatory moment, the early moments of system solution proposal, rather than strengthening the end of the project, when it is nearly always futile and frustrating. The assertion of influence in those initial conversations, using a “management by query” style, triggers a cascade of dynamic positive effects.

The first of these effects aligns the meaning of success for both technical staff and executive. In that initial conversation, both technical staff and executive perceive and define the word "success" differently. They may not always acknowledge that difference. The executive probably thinks of success as meaning the computer system delivers value to the business. The technical team, having more direct experience with the extraordinary complexity inherent in today's systems, thinks differently. They may often only think of success in terms of simply getting the computer system to run at all, by a scheduled delivery date.

To improve the executive's influence over the business value delivered, and to reduce disappointment in the system, what is needed in this conversation is something that will allow

both definitions of success to operate and flourish concurrently. The resulting operative framework then has a sufficiently common language that is both one of the goals as well as the byproducts of the ideals to which this book is a simple yet useful guide.

As an example of the usefulness of a common language and architecture, consider building a resort home. From the first day of conception through the completion of the home, drawings, floor plans and blueprints are used to guide and coordinate the process. They are a thinking tool, a planning tool, a specification tool, an assembly tool and a communication tool. They are an inseparable part of building a home throughout all phases of construction. In order to function on such a broad scale, they are drawn according to conventions, using common verbal terminologies, which are worldwide construction industry parlance.

Similarly, all manufactured items have drawings for various stages of product development -- the larger Boeing airplanes have more than 100 kinds of floor plan-like drawings. The documents used to describe software have equivalent industry standards. Using the insights of this book, special technical training is not necessary for the executive to evaluate the business value of certain of those software documents and, if necessary, reshape that value.

For some of you, a few of the techniques mentioned in the text may appear to be "old hat" because you have heard about them before. You may be inclined to believe, out of habit, that either they are being used in your company or they were historically used but were experienced as ineffective. You might be tempted to simply dismiss those techniques as "old wine in new bottles." The difference is that this book optimizes the "when" of using those techniques. Additionally, the book shows you how to examine the validity of any historical belief about their effectiveness.

HOW THIS BOOK IS ORGANIZED

Part 1: The Executive Role

Chapter 1 explains what makes software intrinsically difficult. It illustrates that the book's method of questions, used in a novel way, at the right time, by the executive can circumvent those intrinsic difficulties. It shows how the tactic of using certain questions can also trigger a change in the software team's behavior in ways that additionally address those intrinsic difficulties.

Chapters 2 through 4 detail the "management by query" questions (and appropriate team answers) used in the dialogue between the team and the executive. The questions attack the intrinsic difficulties of software discovered in Chapter 1 and will frequently reset the courses of action the software team will take to deliver improved business value.

To provide the executive and team with a consistent vocabulary during their dialogue, these chapters also introduce the world of the "Essence" software standard from the Object Management Group (see http://www.omg.org). Developed by members from the SEMAT community (see http://www.semat.org), SEMAT developed the proposal for Essence and submitted it to OMG. OMG has adopted it and has declared ownership of it.

Rather than elaborate the full specification of this standard, this book highlights what is useful for the executive. It will strictly avoid many theoretical terms of that standard (undoubtedly evoking howls, boo's and hisses from theoreticians and purists). It also does not "sell" or proselytize Essence nor explain all its historical evolution. The book will, however, show the important way that executives, users, stakeholders and developers reading this book can use and benefit from the utility of the jargon-free aspect of Essence.

Part 2: Making Success Happen

The remainder of the book covers the last of the 12 steps to success in detail. The sum total ideas of this book may be radical for some organizations, or at least different, especially the ideas of Essence or the philosophies behind the "management-by-query" executive style. Without insertion and transition guidance, the value of this book can easily get lost in the chaos that often poses as software project progress.

These final chapters, therefore, provide a plan for adopting and inserting the book ideas into a company and making a smooth transition to them. Some executives, ones who perceive the utility of the executive and team dialogue mentioned earlier, will want to make certain that the dialogue becomes a routine company occurrence. These chapters enable such an objective, even in a worst case scenario – an executive in charge of a medium-sized company which has a software development group described as being in a high state of "anarchy."

According to the Software Engineering Institute, about 75 percent of all companies fit that description. Stated another way, 75 percent of all companies are building software with more effort and frustration than is really required.

Generally, such unfortunate companies have:

- no idea about the precise cost of software bugs or bug repair,
- no continuous monitoring of software quality,
- no one with specific responsibility to look for ways to get extended return on investment from original, first-time system building efforts; and

- no one responsible for running an explicit, continuous, formal, scheduled review of the exact manner by which people build the computer systems.

This book shows ways of measuring your company's anarchy (see Chapter 5), so you can compare yourself to that scenario and adjust the transition and insertion process accordingly.

PREREQUISITES FOR THIS BOOK

The book presumes the executive has general management skills. The language of this book is largely non-technical. Every attempt has been made to avoid the use of multi-letter technical acronyms. No particular computer literacy is presumed, although it is helpful if you have enough understanding of your information technology department to be able to write a one-sentence description of each of the major computer systems in your company.

For those technical terms that are used, there is a Glossary for reference whenever we have been constrained to compress a complicated concept into such a one-word technical term. When you finish the book, it is worth studying the Glossary in full, as it will help you get your tongue around a vocabulary that will necessarily become part of your enhanced influence and power.

And now, let us begin to learn the details of management by query and the importance of timing.

PART 1 - THE EXECUTIVE ROLE

1

Does the Past Lie to You

Does this sound familiar? "...and I have this great plan from the software team. It will only cost us 50 million dollars," says your CIO/CTO. What goes through your head? "Damn, another 50 million dollars' worth of grief" or something like that?

Information technology should be invisible. It should be an enabling force that allows you to smoothly service existing business and aggressively absorb new business. You should be able to sit in the boardroom, say to the board of directors that you have just hit the "go" button on the project, and then say to everybody that all of them can sit back and watch the plan successfully unfold.

The plan the CIO proposes is undoubtedly great. Based on past company history however, you might feel that implementation of the plan using computers will likely fall far short. But history is misleading you. You are experiencing a symptom of something else going wrong, not the plan.

What is that something?

In this book you are going to learn the reasons things go wrong. And most importantly, you will learn the 12 steps to set them right. But before we look at all of that, it's useful to understand the crucial, top-level differences between software building and other things about which you are likely very knowledgeable such as sales, marketing, distribution,

finance, manufacturing. With that understanding, the logic of the 12 steps will be apparent.

However much like black magic that information technology might seem to you to be, the basic thing going wrong is that information technology is, in fact, currently a very crude technology in most companies. This is true, irrespective of the genius of modern hardware. It is also possible that the behavior of your CIO leads you to believe that software is mysterious rather than crude. Because of that mysteriousness, you pay the CIO good money and then tend to let the CIO alone. But what other part of your business do you let alone? What makes software exempt?

If you can decide to intervene, to raise the level of software construction to somewhere above crude, then software can be changed from an overhead filled with unpleasant surprises to something that is a true enabler of good, profitable business. In the following chapters, you will learn how to raise that level.

LEVERAGE FOR SATISFACTION – THE INCEPTION POWER POINT

Do you have the personal power to raise that level, to stop software from being a torment?

Absolutely yes.

Your point of power is that actual moment when the 50-million-dollar proposal is first being made. Let's call this conversational moment your "inception power point." Do something different in that inception power point and your world will change.

That something different is not about giving new kinds of executive orders, rules or exhortations. Currently, in that moment, you probably ask questions like:

- "What are the features?"
- "When will it be completed?"

- "What is the return on investment?"
- "How long until market availability?"

Those are important questions and you can continue to ask them. However, this book is about getting new results. To get new results, we ask new questions. To that initial conversation, we add some new high-payback questions to that list. The asking of those new questions will send a very loud, unambiguous message to all the people below you.

How will that happen?

In order to answer the new questions, those people will have to think differently and alter their behavior from its prior routine. That behavioral alteration will cause a pervasive, profound change in the way your software supports your business.

What are the questions?

Those questions are part of the 12-step process, and it all begins in Chapter 2. You may be eager to skip ahead to see what they are. If you are eager, go ahead and skip if you like. But come back later when you want to understand just exactly what it is about software that makes it possible for certain questions to cause such profound change. We'll start that understanding by next exploring what makes software so tricky.

THE DISTINCTION THAT MAKES SOFTWARE DIFFERENT

The people in your company are made of atoms. The things they work with day-to-day – smartphones, paper forms, memos, copier machines and so on – are also made of atoms. It's atoms working with atoms. In contrast, information technology is about computer bits, weightless electronic elements living inside wires, working with other computer bits.

People - as atoms - have a property that is extraordinarily different from computer bits. That property is the ability to

innovate, to handle exceptions to the rule, to ad-lib, to deal with surprise or emergency, to make up new rules. People do all of this in a matter-of-fact way all day, and generally do so with social grace, intelligence and adaptability. People have the ability to follow business policy and procedure, and yet make extemporaneous judgment calls about an adjustment in the moment if need arises. They have the ability to respond to changes in the weather or marketplace, temporary shortage of resources, and even “bad hair days.”

Computer bits have no such innovative abilities.

Computer bits are astonishingly dumb. Bits need direction and counseling. This fundamental difference makes software intrinsically difficult to get right. That difficulty presents an enormous unspoken challenge to a software team; it is a challenge which subverts the team energy in powerful ways and prevents full realization of some other business goals as we'll see in the short discussion below.

Slowly, over time, your information technology team identifies the major replicable and routine portions of the business procedures to be performed. Then they try to predict what adaptive behavior might be needed. The team then expresses all this in terms of bits working with bits. Software developers translate the business policy and exception handling skills of people into fixed rules that bits can do to other bits.

But in a way, it is an oversimplification to call this process a "translation" since fixed rules are not intrinsically innovative. The translation process is more like a simulation -- getting bits to at least partially simulate, at faster speed, what innovative people atoms would do if they had the appropriate resources for whatever was the task at hand.

Getting bits to simulate human ad-libbing is fantastically complicated and difficult. The level of difficulty involved parallels the difficulty of getting a cricket to perform (or simulate) singing "Happy Birthday" instead of chirping and then have the cricket go on to do jazz improvisation on the theme.

Software developers consider themselves wildly successful if they get bits to perform just the more routine, non-innovative aspects of what people atoms do without error. They make guesses about which probable innovative behaviors will occur often enough to warrant trying to codify them and putting them in the system. Those guesses are rarely complete because, by definition, they are about behavior that is intrinsically innovative and spontaneous and occurring within an environment which is itself also changing. In other words, it is often quite enough to just get a computer system to run at all.

Another characteristic of software that makes it inherently difficult is the fact that almost every piece of software is a custom piece of software. The development team is making something that it has never built before. That means the team does not have the conscious competence and experiential nuances that come from having already built the same thing several times. The team, despite elaborate preparations, is essentially exploring new territory to a greater or lesser degree with the inherent inefficiencies of such an exploration.

Given the above points, developers historically tested computer systems to the point of that first success, stopped, cheered wildly and breathed a sigh of relief. Because of the difficulty of getting bits to simulate the business behavior of even a single group of people, programmers rarely looked outside that single group's targeted programmed computer system. Computer System A was developed for Group A without reference to Computer System B being developed later for Group B. It wasn't even thinkable that there could be either routine or adaptive behaviors that were identical across different groups. There rarely was an attempt in the first effort to explore if there was any kind of additional return on the initial translation investment that could be used in the second effort. Systems were developed in isolation from one other, with the above limited definition of success being adequate. This bias is largely still true in contemporary software teams.

LOST - SHARED BUSINESS PRACTICES - IF FOUND, PLEASE RETURN TO OWNER

In simulating the behavior of people atoms into behavior of computer bits, what tended to get lost were behaviors in different company groups of people atoms that were identical across groups. Put another way, because software developers in the past built computer systems one at a time and looked at isolated groups of people in business departments one at a time, there was little attempt to identify business behavior common to the many different departments, procedures or groups of people. There was the unspoken presupposition that every new system was always new territory. The result was that in every new system with its new group of people, there was a certain amount of re-translating of what was, in fact, a similar business behavior. Almost without exception, it was a new translation that occurred for every new system. Thus, we see that not only does constructing software take great effort because it's simulating innovative behavior, but it takes even more effort because teams inadvertently redo a lot of the effort in subsequent projects.

These similar business behaviors, this common life blood, can be an important corporate asset when a company can extract it and save its software expression. It is one of the ways the company can buffer itself from the exuberant creativity and adaptability of humans – define and isolate those activities that are the same. There is tremendous business value there because it represents simulations which do not have to be reinvented. Training those bits once, getting them to simulate the behavior correctly, transmitting that simulation expertise forward to the next team and then exploiting that expertise in every future system makes software take less effort in the long run.

In this book, the forward transmission of expertise from an originating software effort is called "extended return." Extended return is additional payback on the original effort. It shows up as shorter time to market for future systems which utilize that effort. And it imparts greater reliability in every future system. Additionally, organizing all new systems

around software containing those shared business practices helps ensure that new systems always deliver at least the minimum value inherent in those practices. When these benefits are realized, software moves closer to the role of a true enabling resource and moves out of the intrusive position it now occupies.

This common blood, which represents shared business practices, was rarely assembled in the past. That failure occurred because it was nobody's explicit job responsibility to collect, archive, catalog and distribute those computer bits. Additionally, it was nobody's job to explicitly design computer systems so that they had an "architecture," a software floor plan which would support both utilization and preservation of that shared business behavior in software.

ASK NEW QUESTIONS; GET NEW BUSINESS VALUE

To ameliorate the intrinsic difficulties of constructing software we have identified, the questions you, as a high-level executive, need to add at that inception power point, are simply ones that probe discerningly for the existence of an architecture and construction process which:

- supports innovation, adaptability and flexibility in response to user or environmental change both during project construction and after completion
- preserves and utilizes shared business activities
- accounts for the custom nature of software

You ask questions that demand, as an implicit part of the answer, that the life blood is present in those systems and that it is directed toward the other goals you want. You use questions which probe, in a new way, for high business value, and payback greater than just the routine user functions expected by the stakeholders of the software.

Those questions will be new, and possibly even disarming, to your team the first time they are heard. They are expecting

you to ask about features, not value. However, repeated asking of certain kinds of questions about architecture and value, accompanied by an insistence on lucid answers, will forever change your company.

In the next three chapters we will look at 11 steps of this 12-step guide. Each step is a question you ask your team about the software system's value. As well, we will highlight what useful answers sound like.

POINTS TO PONDER

In your company, how much collected shareable software expertise is transmitted forward from one project to the next? How much is lost? If you don't know that, do you know what stops your company from measuring that?

Are your non-software departments like manufacturing or sales better at collecting and sharing common business or technical expertise? If they are, what has stopped the software groups from following that lead? What keeps the software groups exempted from behaving like the other departments?

If there is a forward transmission of shareable expertise in your software teams, in what presentation form is it transmitted? By word of mouth? By standardized documents? By programmers' or developers' code? Do you know if people can easily access and read that presentation? Does anybody pay attention to that which is transmitted forward? What is your measurement for that attention?

Do you expect to get extended return on investment from new business procedures that are created for your current sales or manufacturing or shipping groups? Do you expect to be able to redeploy those procedures if you open up a division in another country? Who is responsible for ensuring

that redeployment is possible? What stops your software group from operating in a like manner?

What does your software team do to make software systems adaptable to unexpected user behavior, internal company political change, marketplace change, regulatory change, and hardware change either during project construction or after delivery of the finished project?

2

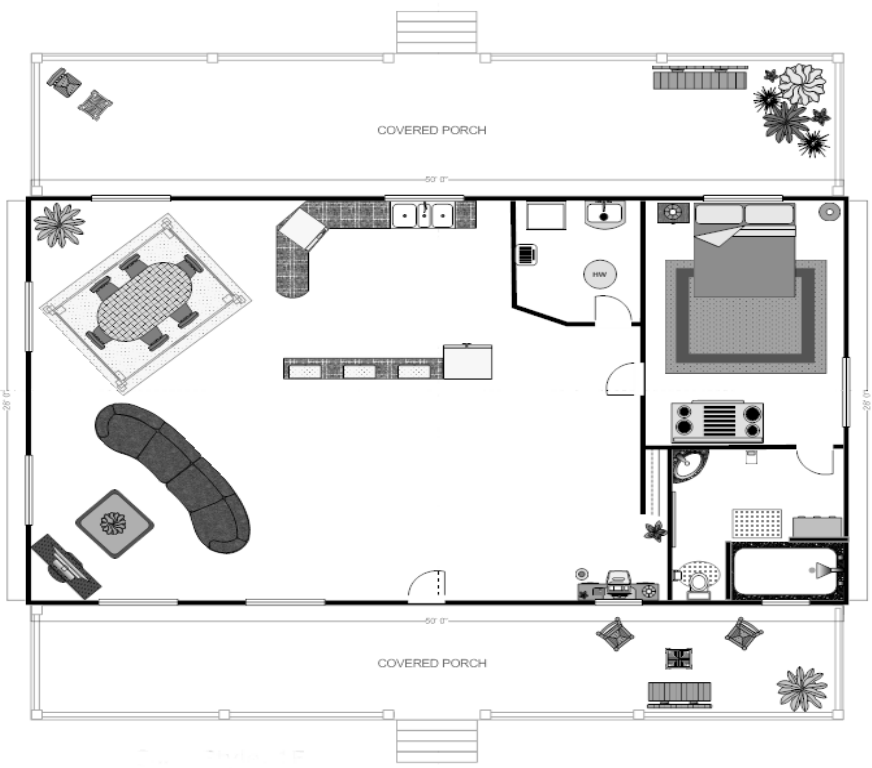
Exercising Your Power With a Velvet Glove

In this chapter, we cover the first 8 steps of this 12 step guide. Each step is a high payback question about value, which in some way addresses the intrinsic difficulties of building software we highlighted in Chapter 1. You could be wondering if you have to take a degree in Computer Science in order deal with these high payback questions. A college degree is not required. You already ask similar questions in other areas of life or business. Now, you're just going to ask them to your CIO, CTO or other technical staff. They will be a kind of velvet glove covering a dominant hand which coaxes and teases your team into new behavior. First, let's look at an area where you use similar questions routinely to determine value about a prospective purchase. Then we'll demystify software by drawing the parallels to that innate understanding you already have.

SIMPLE QUESTIONS FOR A FAMILIAR SCENE

Suppose you were in the market for a resort cottage in the hills near Tuscany. Nothing elaborate, just a simple place where you can get an occasional but vital recharge. Your resort agent lists off these features: one bedroom, bath, living room, two covered porches, kitchen and special 4 burner stove. Let's say this feature list is acceptable. Then the agent shows you this diagram.

Figure 2-1: Floor Plan - Resort Cottage



Like it? The diagram instantly portrays a horrible botch-up regarding design of the cottage. The features are all there, just like you wanted, but the value delivered by this cottage is sharply reduced by the way the features were put together. It doesn't matter what the cost of the cottage is, or whether it will be finished on schedule. It doesn't matter how state-of-the-art the 4 burner stove with grill is. Its location in the bedroom might be right for a hyper-chef who wakes up in the middle of the night with ideas, but a bad match for you. It doesn't matter how glowingly the agent portrays your future happiness owning this cottage.

Looking at the diagram you can ask these questions: Does the path between the kitchen work area and stove seem reasonable? Will food items find their way into difficult-to-clean spaces? Can you add another bedroom without

compromising privacy issues regarding the bath? Can you add anything without mandating another bath as well?

All of this can be seen and inspected because a readable diagram was available that showed how the features were assembled. And, if every time the agent shows you some new cottage, you make similar questions about the floor plan, it won't take long before the agent learns what you want besides obvious features. The agent gets subtly, almost covertly, trained to understand the kind of value you're after.

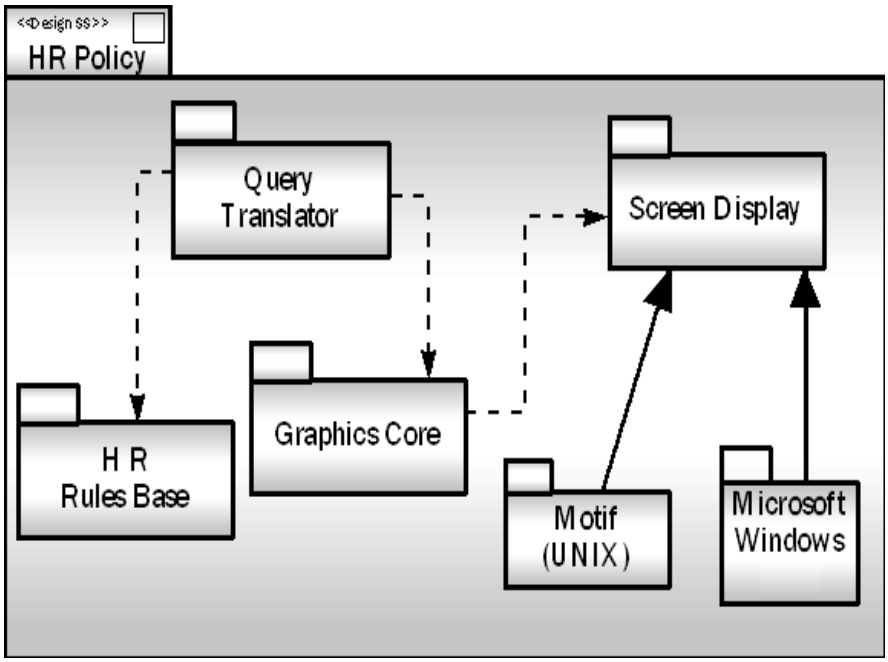
At the risk of being obvious, there is an important rule of thumb that can be taken from the resort cottage example before going on. It is this - a superior way to ensure that value gets delivered is to focus on the architecture of a solution, focus on how things work together; focusing exclusively on features of a solution is the least effective way.

Evaluating a resort cottage requires examining both features and value returned. In this same common sense way, examination of features and value drives the kinds of questions you will ask about a software system.

SIMPLE QUESTIONS FOR THE SOFTWARE SCENE

Suppose that Figure 2.2 below is the diagram of the 50-million-dollar system your CIO proposed. It is a Human Resource Policy application designed to allow employees around the world to query a central site for questions about vacations, sick leave, overtime and so on. In this diagram, the bigger square boxes with little square ears are major components of the Human Resource Policy application. The small little boxes attached to the upper right corner of the bigger boxes, the little “ears,” are places where something like a part number or other identifying data would be placed if this diagram were produced by your team. (For our discussion purposes in this chapter, the "ears" are left empty.)

Figure 2-2: Software Floor Plan Diagram - Human Resources Query System



For the purposes of this illustration, the dotted arrows show movement of data or communication between components of the system. For example, the graphics core sends some kind of data to the Screen Display, perhaps special graphic symbols (in the form of bits) that the Human Resources people like. The solid arrows indicate that computing services (bits doing something to bits) are being made available for use by the box that touches the arrowhead. For example, the Screen Display, which constructs the Human Resource screens for query responses, uses the services of Microsoft Windows Operating System to do that construction. The boxes are the core of a component. The arrows are the communication connections or fittings (technicians say “interface”) between the components.

That’s the basics of navigating this kind of diagram. Your team may use some other diagramming scheme, but it will have similar notions and graphics. They won't call it a “floor

plan", since that isn't technical enough for geek-types; they will have another name. Go with it. Summarizing this diagram: there are components and the components provide either services or data to one another. With this insight into the "furniture" of a Software Floor Plan Diagram, we can do the next 8 steps, and that is, form the high payback questions.

QUESTION #1 - What Goes On in Each Box?

The name of each box should give an obvious clue about what the component does in the context of the solution. Fuzzy names, like "network stuff" to use an extreme example, should arouse your suspicion that more homework needs to be done by the team. At a minimum, your team should be able to give you a simple one-sentence description that makes sense to you. Keep asking until you get that sentence. Also, see the glossary definition for "cohesion".

The remainder of the questions in this chapter depends on your having satisfied yourself that you understand the basic function of each component. It is possible that some small number components do not have direct business function but are rather a kind of glue required by the technology at hand to support business functions. That's OK, but you should be clear about which components are of one kind or the other.

Here are some sample answers for question #1:

"The Query Translator converts the user's typed question into a computer form which can be understood by HR Rules Base and Graphics Core."

"The HR Rules Base contains all the human resource policy rules for the company."

"The Graphics Core contains graphs, charts, pictures, company logo, signatures and so on needed for making the responses visually appealing and presentable."

“The Screen Display makes the pretty pictures on the user’s desktop computer or web browser.”

“Microsoft Windows is a service provided by Microsoft's software which helps the Screen Display make the pretty pictures on a user’s desktop monitor.”

“Motif is a service provided by the UNIX software which helps the Screen Display make the pretty pictures for those users connected to UNIX. The system architecture can use either Motif or Microsoft Windows according to the user’s input device equipment.

Question #1 is a first probe for good architecture. If simple sentences cannot convey useful meaning about the components, the team doesn't have its arms around this application yet. This question lets the team know that you want to poke around inside this system, see how it is put together and that you will not be satisfied with a simple feature description like “it helps our employees understand our company benefits.”

One of our clients was so strongly committed to getting clear responses to this question that he stopped hiring computer science graduates for programmer and developer positions. Instead, he hired English majors and philosophers and then trained them in programming techniques. He did this because that way, he always got systems whose components’ descriptions were clear to understand. And more importantly, he got systems that future programmers could understand when they came back later to add modifications to the system.

QUESTION #2 - How much dependency do we have on “X”?

Dependencies on services that are not part of your team's effort can cause havoc if improperly handled. The dishwasher in your resort cottage needs electric services. No electricity, no clean dishes. With this question, you are evaluating the risk the new system will present to your business due to dependencies, whether they are internal or external. “X” in

the question above is anything on the diagram that provides services or data to another box. So for this diagram, a question would be “how much dependency do we have on Microsoft's software?” A variation on this question is “what are all the dependencies?”

Every system has dependencies. From your perspective, it is important to evaluate the business risk associated with each of the system dependencies and any limitations it imposes on flexibility and adaptability. If your system is highly dependent on something which is subject to change, frequent upgrade, or is marginally reliable, or composed of new untested technology, then the system presents higher business risk. So this question is an explicit probe about business risk.

Good answers are something like “we have isolated our strongest dependency to 3 Microsoft connecting points which are among the most stable ones in Microsoft's marketing history.” The answer shows that the team has been giving thought to minimizing dependency as opposed to wantonly using every possible connection, bell and whistle that might be alluring or glitzy. A primary art of system architecture design is minimizing dependencies.

Dependencies aren’t just about services or data provided by components. Quality of service, often called the “service level agreement”, is also a kind of dependency. For example, many systems that include a network will have a concern about “bandwidth,” the number of bits that can be carried on the wires or fiber optics of the networks. Your questions about dependencies should probe for quality of service levels and fallback plans when service levels are thrown awry because of a wobbling satellite or severed fiber optic cable. In the Human Resources example, any arrow might potentially be implemented as part of a network. There are no universal answers here, except the one that, historically, it seems a company can never have too much bandwidth or too much network fallback.

QUESTION #3 - Which dependencies have we chosen to insulate the system from, what factors led to those choices instead of others and how have we done that?

In the resort cottage, an architect can choose to build the cottage for summer use only or for both winter and summer use. One of the obvious ways to build for both seasons is to make the outside walls thick enough, so that a barrier material can be placed inside the wall to protect against winter cold. The architect could also protect against loss of electric power by including a portable power generator in the cottage appliance wiring circuit, especially if the architect has trumpeted that this cottage is an "all-electric" design. Each choice has an additional cost, but reduces certain kinds of risk or dependency.

Designers of computer systems have the option to insulate the system components shown in the Software Floor Plan Diagram in similar ways. If you don't ask them about such insulation, their habitual choice will probably be to provide minimal insulation in order to keep overall operating and system construction costs down. The company culture may also subtly encourage developers to never try to insulate, since it takes extra development time to evaluate the trade-offs. They will be most tempted to follow the "never-try" habit if the perceived unspoken management priorities have historically seemed to favor meeting the delivery schedule above all goals. So this question gives your team permission to undertake a thoughtful evaluation of ways to minimize the impact of change that could occur in critical dependencies. We will discuss a formal document, the VoxDoc, which your team can build to capture change impact for one and all to see in Chapter 8.

A sample thoughtful answer to this question is something like "we know that the HR Rules Base is in great flux. Policy is revised sometimes as much as twice a year to keep up with competitive employment offerings and our own desire to offer maximum value to our employees. So we have built a little stabilizing filter that sits between the query translator

and the rules base. It presents an unchanging face to the query translator and messages rules so that they always look the same to the translator even if the rules are changing underfoot. We chose to insulate only the dependency with the greatest amount of historical change history and are crossing our fingers that the rate of change of everything else stays as small as it has been historically. The stabilizing filter should handle even changes to federally mandated medical policy."

QUESTION #4 - What is the performance cost of the insulation?

Insulation in computer systems has at least two kinds of cost. There is a design and build cost that is a one-time affair which occurs during system construction. The second kind of cost is the effect on system performance (usually speed) that the insulation exacts as the system runs in operation. The insulation will require some computer resources to do its insulating, thus degrading to some degree, overall system responsiveness. This question probes that impact.

Here is the structure of a satisfactory answer for the earlier stabilizing filter: "There is about a 2 percent performance penalty for that stabilizing filter for query loads as high as 1000 queries per minute. It translates into an additional .5 second delay in web browser response." There are no magic numbers for performance degradation values here, but it should be rationally related to the underlying main business requirements and user experience expectations. Often, that means a smallish performance penalty is tolerable. If it is not relatively small compared to resources needed by other components, then there is risk that the system will collapse under unexpected heavy loads. A special performance engineering group may be required, if you do not already have one, to optimally balance insulation resources requirements against other business requirements.

QUESTION #5 - Where do we insulate against changes in government regulations, competitive trends, marketplace trends etc.?

The architect who designed your resort cottage took into account local, regional, and possibly national building regulations and codes. If he was any good at all, the architect figured into his thinking, trends that he saw happening in those regulations or zoning practice trends which might affect the future serviceability of the cottage.

In a similar way, this question probes for dependencies or assumptions which could be conceivably permeating all components, dependencies which are not related to the arrows in the Software Floor Plan Diagram. To prepare for this question, you, the non-technical executive, should speculate on all the things in the business context you are aware of that might give your business a rough ride over the next several years. It is generally not cost effective to design computer systems which are insulated against every conceivable business contingency that could happen. But the team will take their design priorities from those areas which you scrutinize.

Satisfactory answers would point to specific components on the Software Floor Plan Diagram that performed the insulation against the external business risk trends that are important to you. There may be system speed degradation for that insulation as was mentioned in question #4. Such degradation potential is also worth examining and is another thing called out in the VoxDoc alluded to earlier.

QUESTION #6 - What happens if we change hardware or network components?

Systems can be built in one of two ways: either critically dependent on one-of-a-kind hardware features or, alternatively, built with standardized fittings into which arbitrary hardware can be plugged. “Fitting” here, means the mechanical plugs and electrical signal specifications which need to be common between two components in order for them to work well together. Although there is a spectrum of

choices between these two poles, the speed of technological development and the voraciousness of computer user appetites argue heavily towards emphasizing standardized fittings for hardware.

Left to their own priorities, teams frequently choose making systems tightly dependent on proprietary hardware features, using unique fittings rather than standardized ones, in order to wring out maximum performance from the hardware. That behavior is an easy first choice if the special hardware appears to have lower capital costs.

This first easy choice is a kind of specialization of the system which adapts it to the specific hardware in ways that are typically difficult to reverse. When new demands require a later version or faster hardware, that reversal has to be done after all. The undoing of the specialization is costly and time-consuming above the cost of the later version hardware. To outside stakeholders, this effort is frustrating because it apparently adds no new functional value. A key shorthand word which technicians have when discussing this issue is "portability" - high portability means easy adaptability to new hardware, operating systems, browsers, smartphone devices and so on. So another way to ask the question is something like "how portable is the system software?"

Your team should be able to point to those components, if any, which have been specialized or adapted to particular hardware, devices, network electronics and so on. They should have a well-developed rationale for making those choices rather than more portable choices. And they should also have developed costs associated with being more independent of those specializations so a rational discussion of the near-term versus long-term risks can be evaluated.

A thoughtful answer to question #6 is - "the components feeding the Screen Display, Motif and Microsoft Windows, are the only hardware dependent components. Because we have already organized the Screen Display to accept either of

those two, we are in fact, ready for even a third or any possibility. Geez, Boss, we could connect to a pop-up toaster if we had to."

QUESTION #7 - What happens if we add a new line of business such as Y?

Your team should be able to point to components on the diagram which might be affected by adding a new business line. In all fairness, if the new line of business, "Y", is not something of which the team might have been previously aware, they may need some time to answer this question. In any event, beware of system architectures where the majority of components are affected by such a change. The likelihood is high that such architecture has not yet been refined to the point where the real core of your company's life blood has been found. For the HR Policy System, a good answer would be - "we only need to add some new rules to the H R Rules Base component."

All the previous questions probed to determine if the features of the system would continue to deliver value in spite of perturbations in the environment. They probed how the system was put together in an effort to see if rework of the system, for whatever reason during development or after, was as easy as possible.

QUESTION #8 - What happens if we want to share this system with another division (or company)?

This question probes for extended return on investment. It asks the team to identify components that can be shared by non-local systems. This question probes for company life blood, the bits that have utility across company boundaries, bits representing accumulated expertise transmissible to the next generation of system developers.

If you feel in a particularly puckish and provocative mood, point to the Software Floor Plan Diagram and phrase the question this way, "which of those components can we put in software inventory?" No CEO, no high-level executive has ever put this question this way before. Your team will look at

you dazed, perhaps as if you had spoken Sanskrit or twelfth century Gaelic. Just continue, “after all, John Deere Tractor Company has tractor seats in inventory, what stops us from having software components in inventory?”

Here is an exemplary answer to the question - “We've identified that there are patterns in the way different company divisions use graphic symbols. There are four different patterns. By plugging the appropriate pattern into the Graphics Core component, we can get extended return from at least five future systems that are planned in the other divisions. That makes the Graphics Core component an inventory item.”

Of course, it is possible that the answer you get is a response to the “what stops us” part of your question. In other words, your people actually delineate the present roadblocks to getting extended return on software. We will talk more about getting extended return in later parts of this book.

The questions of this chapter can provoke answers like “we can’t do that” or “it can’t be done.” While that may be true sometimes in the course of a company’s software process, it rarely is as definitive a statement as the strength of the voice tone of the answerer might imply. Architects and team leaders generally are very high integrity people and take great pride in their art, but can be conservative because of past projects that went bad. So they are reluctant to go out on a limb and support an idea which they feel will have chancy success. Often, an architect will say it can’t be done when what he really means is that he personally has not done it himself three times before. Or he may say that it’s impossible because he feels the cost will be alarming.

Don’t stop probing when you hear the phrase “it can’t be done.” But never ask “why” it can’t be done. Using that word “why” will typically get you rationalization, feelings, opinions or justification which then leads to debate rather than progress. Instead, like the software inventory question above, your most useful response is “well, what stops us from doing it?” or “what stops it from being possible?” or even “when did

you know it can't be done?" These alternatives to "why" will often open the way to a more objective discussion which will enable the impossibility to be resolved in a surprisingly delightful manner. (We rarely use "why" in that way in this book.)

In the next chapter we will look at guidance steps 9 and 10. They will tell you the way in which the activities of building the system are adding to or reducing the risk of increased software development costs and effort. In particular, we will look at whether it is useful for your team to think heroically.

POINTS TO PONDER

Has your team ever produced a drawing that was supposed to depict architecture? What purpose did they intend the drawing to be used for? Did its purpose include evaluating dependencies? If so, were the dependencies obvious in the diagram, or were only the features obvious?

How much of the computer system development cost has to be expended now before you know what the dependencies are? (Chapter 4 will discuss this question.) As a percentage of total development cost, does that cost parallel the percentage that needs to be spent in other company divisions to identify critical dependencies, divisions such as manufacturing, sales, engineering, or finance?

Do you have in place an inventory system for developer software components that is as sophisticated, clean and as well-financed as the inventory control procedures in other parts of your company? If not, what would stop you from appointing someone with sufficient authority and budget to make that happen?

For additional Appendix resources of the book, please visit:
http://www.timingisalmosteverything.com

========== END OF BOOK SAMPLE ==========