2

SCHMOOZING WITH THE IDEAS

2.1. OVERVIEW

The major ideas of this book and how they relate to one another are introduced in this chapter. Some of the words and phrases which will be frequently encountered are explained and their contexts clarified. This chapter may be viewed as an informal and readable glossary of the themes with which we will be mainly concerned going forward.

2.2. IDEAS AND THEIR RELATIONS

Every book is a book of ideas. We may use facts, figures, and opinions to contradict or corroborate, but ultimately we are trying to establish ideas. Ideas in isolation are exciting, but really exciting things begin to happen when ideas relate to one another. Each of us has our own "eureka" moments, when hitherto disparate notions click together in congress. Books enrich us by uncovering some of this clicking together and somehow convincing us that relating ideas is not such an esoteric game after all. Some reflection and discipline can get us there, which they surely can.

I sometimes think the very attitude toward books has changed with the popularity of the Web and the ubiquity of information it has brought with it. We rely less and less on books for our facts; the onus of books has shifted to

presenting new ideas or at least relating existing ideas in novel or insightful ways.

This chapter is kind of an *idea map* for this book. Here you will be introduced to some of the keynote ideas you will need to live with from now on. Certain words and phrases will recur in this book, and this chapter will ensure we are all on the same page about their meanings and insinuations. Usually, a glossary does something like that, but it comes at the end of a book. As I am too lazy to flip back and forth, I usually find myself assuming meanings of words and phrases and reading on, only to find later that the author meant very differently. The reverse is also true at times; two ideas may be very closely related, but I suffer by assuming they are gulfs apart. Once I was reading the requirement specifications for a large project, with a pressing need to make sense of them. I frequently came across "users" and "merchants" and got confused about their overlapping roles, only to realize later they were one and the same business entity. Reading the glossary might have helped, or might not have. Unfortunately, reading glossaries is no fun, just as reading dictionaries is not, no matter how rich their content is. Therefore, I decided to front-end the glossary idea and throw in some more verbiage, so that you have a fair idea what the book is about before taking the plunge. Hopefully, acquaintance with the themes up front will also help you absorb the narrative better. In the next few sections, quick summaries of the ideas in this book are given, followed by an effort to tie the threads together. Before that, however, let us savor two instances where ideas separated by eons and disciplines have come together in hair-raising harmony. These have always motivated me in ferreting around for relations amongst ideas.

In 1936, Alan Turing (1936) published the paper "On Computable Numbers, with an Application to the Entscheidungsproblem." Turing introduced the notion of what subsequently became known as the "Turing machine." Turing was exploring the ideas of computability, and his Turing machine was initially a thought construct to illustrate his arguments. Bewitchingly simple in conception, a Turing machine is ultimately a symbol-manipulating device that consists of an infinite-length tape, a head (which can move left or right) to read and write symbols at box-like locations or cells on the tape, and a table of instructions. It has since become manifest that a Turing machine is a complete model for the basic operations of a computer. But this relation of ideas, from a pure thought construct to the ubiquitous utility of computers as we know them today, does not end the story of Turing machines. DNA — the genetic character sequences regarded as the "blueprint" of life — has been married to the Turing machine idea in recent research into the feasibility of DNA computers (Parker 2003; Johnson 2000). This serves as just one instance of how ideas are not bound by provenance or politics.

Turning to another area, a Bose-Einstein condensate is a phase of matter formed by bosons (particles having integer spin, named after Bose) cooled to temperatures very near to absolute zero (0 Kelvin or –273.15 degrees Celsius). This was first predicted in 1925 through the work of Albert Einstein and Satyendranath Bose. It was produced seventy years later by Eric Cornell and Carl Wieman (1995) at the University of Colorado in Boulder. Tim Berners-Lee (1999), while working as a software engineer at CERN, the European Particle Physics Laboratory, in Geneva in the 1980s and early 1990s, conceived, built, and refined the infrastructure of what was to become the World Wide Web. The evolution and dynamics of the Web have been the subject of much scrutiny, leading to the startling question: Could the Web or the Internet represent a gigantic Bose condensate (Barabasi 2001)? This is another example of confluence of ideas from fields seemingly far apart.

These examples wake us to the gregariousness of ideas. Even if we cannot see every thread in its full light (in all honesty, DNA and that condensate stuff are far beyond me), we can appreciate the beauty and, more often than not, the utility of the braid. The trope of a braid in describing the intermingling of ideas has been powerfully exploited in the Pulitzer Prize–winning book *Gödel, Escher, Bach: An Eternal Golden Braid* (Hofstadter 1979).

We will now look at the ideas in this book and their connectedness. After this chapter, whenever reference is made to a word or phrase to which a subsection is devoted here (like *enterprise software* or *metrics*), we will implicitly take it to have the meaning and context described below.

2.3. THEMES IN THIS BOOK

2.3.1. Enterprise Software Systems

Enterprise software systems are in a way the very *raison d'être* of the software engineering profession. The phrase will be encountered many times as we go deeper. How are enterprise software systems characterized?

Enterprise software systems usually support business processes; they need to respond to changing user needs, they are bound by business and technological constraints, and their soundness of design and implementation is of *material* interest to different groups of stakeholders. To the above must be added another feature which is becoming increasingly common to these systems: their scope of operation ranges across a diverse spectrum of geography, nationality, and culture. Fowler (2003), in his book *Patterns of Enterprise Application Architecture,* says enterprise applications are characterized by persistent data, concurrent data access, lots of "user interface screens," needs to integrate with other enterprise applications, needs to bridge conceptual dissonance between diverse

business processes, and complex business "illogic." This is indeed a very insightful list; the last two points in particular deserve some elaboration.

Conceptual dissonance is an apt expression for the great (and grave) diversity of interests every enterprise system has to reasonably resolve. An enterprise application is usually a confluence of many business processes, none of which is obligated to align with others. These nonaligned, often contradictory interests are represented by different stakeholders, all of which stand to gain or lose materially from the success or failure of the system. The gains and losses, however, are very different in nature and extent. Building and maintaining enterprise software systems is about balancing very many tugs and pulls in very many directions to minimize losses and maximize gains for every stakeholder.

Fowler also hits the nail on the head when he talks about the illogic of business logic. Business logic is about business; it is not about logic. Logic is an overloaded and overworked word, with such exalted trappings as reason and syllogism. "Business rules" would be a better name for the instincts and credos that drive any business. They have no reason to be logical or at least logical in the sense software engineers (let alone logicians) understand the term. A major challenge of enterprise software development is to capture the capriciousness of business rules in the formal structure of design and ultimately code.

Enterprise software systems usually do not involve complex mathematical operations on data. It would be highly unlikely to have such a system solve differential equations. Enterprise systems do, however, deal with very large amounts of data and their storage, display, and simple manipulations. Another aspect of enterprise software is that the users of such systems are different than the developers, and these two groups are different from those who commission the building of the software. This is of much consequence, as we have at least three different groups associated with the systems, which speak three different languages and yet must talk to one another.

Thus, in essence, enterprise software systems are software applications which are commissioned to support business processes that involve diverse groups of stakeholders. These systems may have a scope of development and operation that ranges across a variety of technological, geographical, and cultural domains.

We will next be more precise about the stakeholders.

2.3.2. Stakeholders

The *Merriam-Webster Online Dictionary* (2006) defines *stake* as "an interest or share in an undertaking or enterprise." (We will ignore other less germane

definitions, one of which is the device for burning dissonant individuals in less tolerant times.) And we have just been talking about *enterprise* software systems! Simply put, stakeholders are people whose material interests stand to be affected, for better or for worse, by an undertaking with which they associate. Material interest is different from such spiritual passions as the joy of discovery, the love of humanity, or the bliss of doing something well. Very often, material interest means immediate financial gain. Sometimes the connection may not be that quick, but stakeholders are always in the game for some worldly gain. There is a time in everyone's life when worldly gains seem so gross. Then we are able to take a more practical view of the world. This view is very important for those in the business of developing enterprise software.

Some stakeholders are easy to identify. In an enterprise software project, the customer (represented by a small group from the customer organization) is a stakeholder; the customer is pumping money into the project in the hope it will help bring back more money. The development team (a group of individuals assigned to the project by the organization contracted by the customer) is a stakeholder; if the project goes well, the team wins accolades, which may eventually translate to enhanced responsibility and a higher pay packet for its members. The users (for an online application, the multitude of individuals who access it over the Web, to send, fetch, or review information) are stakeholders; they are spending time and energy, and maybe money, on the application in the hope of being adequately recompensed. These are just the overt stakeholders. There may be other covert ones too. What about competing companies or groups within the same company whose interests are not best met if the project succeeds? What about other business entities curious to see how the system impacts the users, so their strategies can be accordingly tuned? Thus, stakeholders exist at many levels of visibility and invisibility. The development team, a stakeholder itself, has the job of reconciling the stakes of all other stakeholders. The overt ones are paramount, but it pays to keep an antenna open to the covert ones too.

Let us now examine a special kind of stakeholder: the practitioners.

2.3.3. Practitioners

Practitioners may or may not preach, but they are professionally obligated to practice. Engineering to a large extent is a practitioner's profession. Things must be made to actually work, even if they should work in theory. When we talk about practitioners in this book, we will be referring to individuals or groups who are expected to carry out the techniques described. Practitioners certainly have a significant stake in the success of what they practice, so they

are stakeholders too. However, it was imperative to give them another, more specific name, as this book is mainly practice oriented. Whatever idea is put forward, if it is worth its salt, should help practitioners somewhere do their jobs better.

So far we have dwelled upon the system and its players. Let us now limn another key word in the title of this book: metrics.

2.3.4. Measure-Measurement-Metrics

Even within the software engineering context, there is hardly any consensus as to what measure, measurement, or metrics truly means. Discord is good, as it usually signifies independent thinking. Pressman (2000) says "a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process" and "measurement is the act of determining a measure." The IEEE (1990) Software Engineering Standards define metric as "…a quantitative measure of the degree to which a system, component, or process possesses a given attribute." In this book, whenever we talk about metrics, we take it to cover the entire gamut of measure, measurements, and metrics. We will concern ourselves more with the utility of a metrics-driven approach to enterprise software development and less with its pedagogical aspects. The chapters in Part 1 are devoted to a general review of metrics in software engineering, as well as some key themes in software measurements. We now take a passing glance at how metrics relate to our lives.

Nothing is truer than Mah and Putnum's words: "We manage things 'by the numbers' in many aspects of our lives.…These numbers give us insight and help steer our actions" (Pressman 2000). Inflation rate indicates the state of an economy, blood cholesterol level signifies the state of one's heart, and money is a measure of one's station in life. All three of these work on certain assumptions. Inflation takes into account only a "basket" of goods and services, cholesterol level is just one among many factors that make or break a heart, and money cannot buy happiness. Yet policies are based on the inflation rate, care goes into keeping cholesterol in check, and the pursuit of money is deemed a worthy endeavor. Thus the assumptions behind these metrics, though certainly not sweepingly true, are apt enough under certain situations. These are just some of the metrics which guide decisions on national, societal, and personal levels. Metrics give us some numbers upon which we base the branching logic of our decision trees. While it is important to remain mindful of the limitations the assumptions of every metric place upon it, it is equally important to understand that metrics can and do help traverse the maze of complex decision making in our lives.

Decision making is an intriguing process, tied closely as it is to the very functioning of the human mind. Decisions come out of a chemistry of observation and perception, facts and hunches, objectivity and subjectivity. Use of metrics streamlines the decision-making process to a large extent, clearly delineating the subjective and the objective aspects and their interfaces. In this book, it will be illustrated how metrics help us make expedient choices in the software development process. I always find it easy and useful to view metrics as some kind of *heuristic.* This brings us to the next important idea of this book.

2.3.5. Heuristics

Maier and Rechtin (2000), in the second edition of their seminal work *The Art of Systems Architecting,* treat *heuristics* as "abstractions of experience." The word "heuristics" certainly has a Greek origin, but different authors give different shades to its original meaning. Maier and Rechtin (2000) say the Greek word *heuriskein* means to "'find a way' or 'to guide' in the sense of piloting a boat through treacherous shoals." According to Luger (2004), the original word means "to discover." Luger offers a down-to-earth take on a heuristic as "a useful, but potentially fallible problem-solving strategy…" and adds that "much of what we commonly call intelligence seems to reside in the heuristics used by humans to solve problems." According to Polya (1945), in his classic *How to Solve It: A New Aspect of Mathematical Method*: "Heuristic, as an adjective, means 'serving to discover'…The aim of heuristic is to study the methods and rules of discovery and invention."

Cutting through these thickets of definitions, the notion of a heuristic is familiar to many of us. To cite some of the instances mentioned by Luger (2004), checking to see if an appliance is plugged in before complaining that it does not work is a heuristic, as is the strategy of "castling" in chess to fortify your king. Doctors and car mechanics use heuristics all the time. Nausea and stomach pain? Likely diagnosis: food poisoning. Too much smoke and low gas mileage? The car may be due for an engine tune-up. Every professional has his or her own bag of heuristics, which are refined and expanded with experience. We are also taught many heuristics in school. The sum of the angles of a triangle is 180 degrees is one of the earliest and most widely used heuristics in geometry. It helps in the deduction of many clever things about a triangle, given specification of some of its sides and angles. Many of the heuristics we use in getting through an average day at home, work, or school are so ingrained in our education, culture, or just general awareness that we hardly notice the ways they make us smarter, helping us avoid past mistakes and make better judgments. In fact, in our everyday lives, much of what is feted as sterling "common sense"

is actually a subconscious collection of heuristics interlinked and annotated for quick recall and application.

But heuristics are not eternal truths. Luger's comment on their potential fallibility is significant. Many a time a doctor's or car mechanic's diagnosis goes wrong (in spite of their most sincere efforts): nausea and stomach pain may not just be food poisoning, malodorous smoke and low gas mileage may be due to the demise of one or more engine cylinder, and the angles of a triangle will not add up to 180 degrees if the triangle is described on the surface of a sphere. Thus heuristics come with certain assumptions about scope and context, and one needs to remember these when using heuristics.

In spite of this fallibility, the state of heuristics indicates a discipline's maturity. Maier and Rechtin (2000) give a list of "heuristics for systems-level architecting" in their book, classifying each heuristic as *prescriptive* or *descriptive*. Culled from the literature and the authors' own research, almost all are very pithy and some quaintly aphoristic: "Success is defined by the beholder, not by the architect"; "one person's architecture is another person's detail"; "if you can't analyze it, don't build it"; and so on. Polya (1945) devotes more than three-quarters of his book to what he calls the "Short Dictionary of Heuristic," which contains sections such as "Can You Derive the Result Differently?," "Decomposing and Recombining," and "Have You Seen It Before?" Both works, set apart by half a century and different disciplines, share the commonality of purpose in compiling a set of useful heuristics for practitioners.

Heuristics closely relate to something of a more homely name: rules of thumb. In my area of undergraduate study, electrical engineering, we had Fleming's left and right hand rules, where the thumb *really* entered into the rules. In general, rules of thumb are quick and easy judgment aids (although the thinking that went into them might have been neither quick nor easy) that can be widely applied. In his *The Timeless Way of Building,* Alexander (1979) explains how design involves calling forth rules of thumb gathered through experience and practice. He goes on to add that "…each one of us, no matter how humble, or how elevated, has a vast fabric of rules of thumb, in our minds, which tell us what to do when it comes time to act."

Sets of time-tested heuristics or rules of thumb are still evolving for software engineering. Metrics can go a long way in enriching this body of common knowledge. The formulation of a metric encapsulates much reflection, awareness, and experience. Applying it in a given scenario lets us leverage the background wisdom without going through the motions again. A good metric goes much beyond being just a number; it becomes a heuristic for guiding software development through the "treacherous shoals" of changing user requirements, technological and business constraints, and ever-gnawing competition.

In this book, we will build some metrics and show how they can be applied. The thrust of the discussion will be toward distilling the scope and aptitude of the metrics into heuristics. The heuristics should be useful even without the scaffolding of metrics derivation.

But for any metric, heuristic, or rule of thumb to work for the better, one needs a closed-loop system, that is, a system with feedback.

2.3.6. Feedback

Feedback is one of the most fundamental techniques of engineering. Like all fundamental techniques, it goes beyond a discipline and spreads across life. Feedback is one of those tenets that seems to work because it is so intuitive and seems so intuitive because it works. In the simplest of terms, feedback is a mechanism for controlling an activity by regulating the input based on the output.

Feedback is nearly everywhere. We use it all the time, often without realizing it. Pressing on the accelerator increases the speed of a car: the visual perception of the car's speed is processed back to the foot, to modify the pressure and control the speed. If you are hungry after a hard day, you ingest food rapidly and in large servings, but nearing the level of satiety, the quantity and celerity of intake go down until you stop altogether. This happens as signals from the stomach go back to the brain, which controls the hand that feeds the mouth. Autopilot systems on aircraft monitor altitude and other parameters and feed them back to the system to generate required levels of thrust to maintain the plane on an even keel. As mentioned earlier, often the phrase "closed-loop system" is used to denote a system which has a feedback path from the output to the input. A comparator mechanism gauges the actual output vis-à-vis the desired output and adjusts the input accordingly. Success of a feedback mechanism hinges on a few key factors. We must be able to measure the input and output and have a clear notion of what the output needs to be. Feedback techniques are not yet sufficiently mature in enterprise software development. Metrics can play a crucial role in harnessing this classic engineering stratagem in software building.

But do software development processes allow for feedback loops? Not all do, but the iterative and incremental development model does.

2.3.7. Iterative and Incremental Development

Larman and Basili (2003), in their paper "Iterative and Incremental Development: A Brief History," highlight how the roots of iterative and incremental

development (or IID, as the authors abbreviate it, true to software's acronymic culture) go far deeper and are older than the recent interest in "agile" methods. Among the most interesting insights, we learn how Royce's (1987) article titled "Managing the Development of Large Software Systems," widely considered to be the waterfall manifesto, in fact contains germs of iterative and incremental development. "If the computer program in question is being developed for the first time, arrange matters so that the version finally delivered to the customer is actually the *second* version insofar as critical design/operations areas are concerned" (italics added). How close in spirit this advice is to Brooks's (1995) credo — "Plan to throw one away; you will, anyhow" — which Raymond (2001), in his classic essay *The Cathedral and the Bazaar,* explains as "…you don't really understand the problem until after the first time you implement a solution.…So if you want to get it right, be ready to start over *at least* once. Brooks (2000), in his keynote speech at the 1995 International Conference on Software Engineering, declared: "The waterfall model is wrong!" In his 2000 Turing lecture "The Design of Design," Brooks (2000) goes a step further and titles one of his slides "The Waterfall Model Is Dead Wrong"!

Instead of being too judgmental, I have found that it helps to regard both the waterfall and the iterative and incremental approaches to software development as being complementary in a subtle and useful manner. What goes on inside an iteration is often not very different from the waterfall tenets. The acts of analyzing, designing, building, and testing a software system *must* have an element of sequential linearity; one cannot analyze after testing (although testing before building, or at least thinking and talking about testing before building, is recommended in some circles). The most marked departure of the iterative and incremental worldview from the waterfall is the recognition of the inherently evolutionary nature of software. Gilb (1977), in his book *Software Metrics* (arguably the first book on software metrics and undoubtedly the first juxtaposition of "software" with "metrics," thus coining the phrase), was basically promoting iterative and incremental development when he talked about how complex systems have a better chance of success when implemented in "small steps," each step having a "clear measure of successful achievement as well as a 'retreat' possibility to a previous successful step upon failure." He also underlined the scope of receiving "feedback from the real world" and how this helps better allocation of resources and correction of errors. Retreat and feedback are the key concepts in iterative and incremental development.

As Campbell (1984) observed in his *Grammatical Man* — a book which brilliantly matches depth, width, and concision — evolution in the natural world, contrary to popular belief, is far from a unidirectional ascent from lower to higher forms of life. The path is replete with false starts, cul-de-sacs, peaks, and plateaus. An ineluctable feedback mechanism is continuously at play, mod-

erating the journey through countless repeats and revisits toward deeper levels of perfection. The iterative and incremental model takes its lesson from these evolutionary trends.

Very interestingly, the co-evolution model (Brooks 2000) seeks to establish how the evolution of the problem space is influenced by the evolution of the solution space. Among the key benefits of the iterative and incremental model is that it allows for the inevitable changes in the problem domain and helps tune the solution domain to the latest realities of the problem.

Chapter 6, the first chapter in Part 2, is devoted to discussion of the dynamics of the iterative and incremental model in detail. The techniques presented in this book are all geared toward facilitating feedback and hence applicable when software is built iteratively and incrementally. Although iterative and incremental development is hard to understand and harder to apply (it takes more than one iteration to get used to, believe me!), it remains the most expedient software development philosophy. We will have many occasions to explore its innards in this book.

Iterative and incremental development as a process works so well for software even as it is absurd in other conventional engineering disciplines. The reasons lie in the peculiarities of software as an industrial artifact, some of which are described in Chapter 3. Information technology, or IT, is *ITerative,* and the industry has more or less awakened to this truth. Yet much confusion still lurks regarding the place of processes in the software sun. Why and where does software development need processes?

2.3.8. Process

A process is a set of predefined and coordinated activities prescribed to practitioners — in teams or individually — with the intention of fulfilling an objective. The second edition of *The Unified Modeling Language User Reference* defines a software development process simply "as the steps and guidelines by which to develop a system" (Rumbaugh et al. 2005). More specifically, a software development process has been called "the set of activities needed to transform a user's requirements into a software system" (Jacobson et al. 1999). Processes ensure consistent levels of quality and repeatability in any industrial production. Awaking to the need for processes in enterprise software production marks a milestone. It comes with the crucial recognition of software as a product of engineering, bound by the usual demands of reliability, consistency, safety, and usability that set apart good engineering from bad. It is good that "process" in software engineering has come to have overtones related to discipline, quality, and such salutary traits. References to "process-driven approaches" nowadays are often taken to mean ways of doing things that are not just ad hoc or

instinctive but include time-tested techniques and the wisdom from past successes and failures. When process is mentioned in this book, we are talking about some standardized way of doing some software development activity.

Processes are great things. They allow for teams of practitioners with very different preparation and perception (or even political views!) to work together and produce consistent results. They allow for smart integration of technology and human skills toward the generation of superior software. But processes are not everything. There are certain areas — and in many of these lies much that is enchanting in software development — where processes cannot help and may also hinder. In the 2000 Turing Award Lecture, Brooks (2000) makes the compelling point that great designs come not from processes but from great designers. And where do great designers come from? Brooks says great designers need to be grown deliberately and managed imaginatively. He adds another sly and sapient slant: "We have to protect them [great designers] *fiercely.*" The next bullet reads: "From managers." And the next: "From managing." The message comes through loud and clear. The most reflective and thoughtful activities of software development need undistracted efforts of the mind. Processes cannot give us great ideas, but they can guide us as to how best to use a great idea or make the best of a not-so-great idea. Processes can help turn ideas into ubiquitous utility — so ubiquitous and so utilitarian that we no longer wonder about them, like the lightbulb, air travel, or even the Web. We make best use of processes when we are sensitive to both their strengths and limitations.

Practitioners in the enterprise software development business often love and hate processes at the same time. It is charming to see processes align a diverse set of individuals and technology toward a common goal. It is equally maddening to share the frustration and angst of going over the facile motions of process for the sake of process. I am increasingly led to believe that, just like antipatterns, there may be antiprocesses, sets of activities which never should have been *processized* to begin with. In general, metrics play an important role in identifying the facility or fatuity of processes.

Software engineering processes are sometimes prone to what I call the "unicorn effect"; it is easy to get very excited about great processes without ever getting to see one. This is a dangerous circumstance for enterprise software development. To ensure this does not happen, artifacts come in handy.

2.3.9. Artifacts

Kruchten (2004), in his book *The Rational Unified Process: An Introduction,* calls an artifact "a piece of information that is produced, modified or used by

a process." An artifact usually manifests as a textual or pictorial record of the output of one or a set of related tasks mandated by a process. The word has an artistic ring to it; in fact, it derives from the Latin *arte* by skill (ablative of *art-*, *ars* skill) + *factum,* neuter of *factus,* past participle of *facere* to do (Merriam-Webster 2006). Artifacts are produced when the skill of extraneous (as human) agency is applied to environmental ingredients. The brilliant Altamira cave paintings (most memorably the charging bison) created by predecessors to modern man, whose drawing skills used the ingredients around them (the cave walls, the color pigments, maybe some flesh-and-blood inspiration bison), are certainly artifacts on one level. (The artifact product has far outlived the process which was used to arrive at it, and herein lies its triumph; processes are only remembered when something went wrong with them.) On another level, a piece of software design — boxes and lines scribbled on the back of an envelope or a sequence diagram with all annotations — is also an artifact. It creates a new way of thinking, gleaning new information from existing information such as user needs and technological and business constraints. Artifacts are easily confused with documents. All artifacts need to be documented if they are to be preserved for review and reuse, but artifacts are ultimately the thinking that goes behind the mere recording.

There are conflicting views on whether code qualifies as an artifact. In my opinion, it does. Code is essentially new information created from the synergy of all the factors affecting software development. In fact, code is by far the most flexible — in terms of both structure and function — artifact that the software development life cycle produces. The techniques described in this book will lead to the recording of new information such as metrics data or classification of entities; these are all artifacts.

2.4. THREADS AND BRAIDS

In the last section, we reflected on the major ideas of this book and saw how one leads to another. Some aspects of interplay of these ideas must already be clear. How this book tries to bring together the threads in a braid is discussed next.

As mentioned earlier, the harmony of different, even seemingly contrasting ideas has been illustrated to great effect in Hofstadter's book *Gödel, Escher, Bach: An Eternal Golden Braid.* Hofstadter (1979) says: "I realized that to me, Gödel and Escher and Bach were only shadows cast in different directions by some central solid essence. I tried to reconstruct the central object, and came up with this book." The book is a significant work and deals with issues deep

and resonant. I borrow the construct of a braid to underline the intermingling of the ideas in this book.

Enterprise software systems are the field of our interest. We are not focusing on software that is built for instructional purposes (such as classroom projects), for highly specific scientific computing, or for mere pleasure. Some of our discussions may be pertinent to such types of software families, but they certainly are not our main concern. Enterprise software systems have *stakeholders* associated with them, usually several different groups. *Practitioners* are a special class of stakeholders; the practice of their profession involves building enterprise software systems and resolving the (often conflicting) stakeholders' demands from the system. We seek to establish how *metrics* can help practitioners do their job better; we take "metrics" to cover the whole gamut of measure, measurements, and metrics. The application and interpretation of metrics lead to more general and intuitive *heuristics* — rules of thumb which allow practitioners to make informed judgments as they go about enterprise software development. Metrics and heuristics facilitate *feedback* in the development process, a crucial factor in ensuring the solution stays aligned to the ever-evolving problem. The *iterative and incremental development* model has built-in feedback mechanisms which use inputs from the real world (user responses) to better tune the objective and course of development. Iterative and incremental development is a particular kind of *process,* a set of guidelines for the construction, delivery, and maintenance of software systems. *Artifacts* are outputs from processes; they embody the information that is produced, modified, or consumed by process activities. Some of the methods in this book call for a special way of arranging and interpreting information from the development activities; we also call such products artifacts.

This interplay of these ideas has strongly influenced the organization of the book (Figure 2.1) and the suggested reading plan in Chapter 1.

2.5. SUMMARY

This chapter reflected on the significant ideas permeating this book and their interactions. As we go deeper, we will discover more related concepts and other interesting ways how all of these play with one another. Some of these ideas are actually collections of ideas; they are grouped together so that they can be referred to easily later, to explain, refine, or dissect further.

We are now ready to get into the meat of our matter. Let us proceed to Part 1; the next chapter gives an overview of the place of metrics in software engineering.

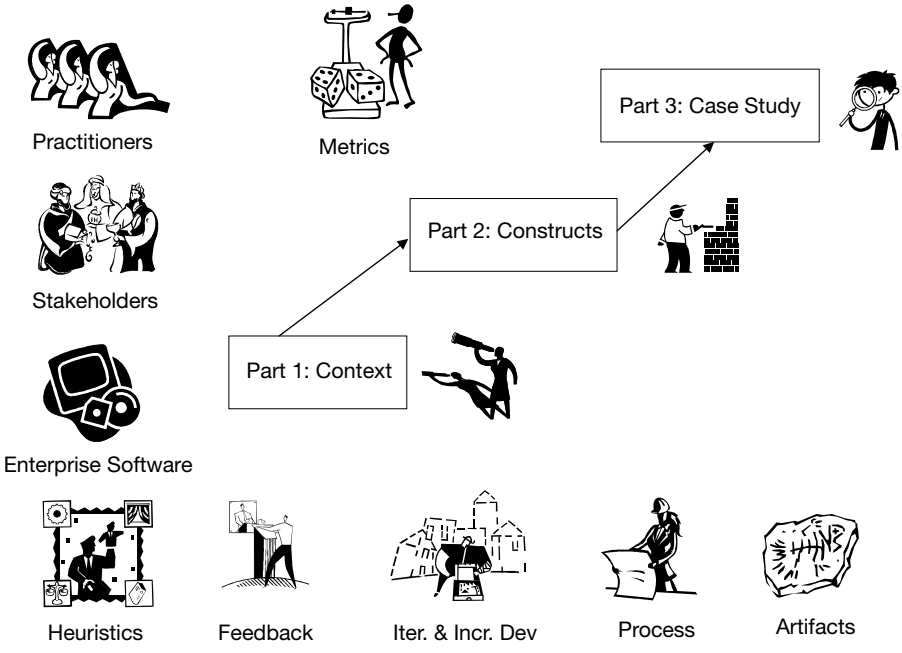

Figure 2.1. Interplay of ideas and organization of this book

REFERENCES

Alexander, C. (1979). *The Timeless Way of Building.* Oxford University Press.

Barabasi, A.-L. (2001). The Physics of the Web. http://physicsweb.org/articles/world/14/7/9.

Berners-Lee, T. (1999). *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor.* Harper San Francisco.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley.

Brooks, F. P. (2000). The Design of Design, A. M. Turing Award Lecture. http://www.siggraph.org/s2000/conference/turing/index.html.

Campbell, J. (1984). *Grammatical Man: Information, Entropy, Language and Life; The Story of the Modern Revolution in Human Thought.* Penguin Books.

Cornell, E. and Wieman, C. (1995). BEC home page. http://www.colorado.edu/physics/2000/bec/.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Addison-Wesley.

Gilb, T. (1977). *Software Metrics.* Winthrop Publishers.

Hofstadter, D. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid.* Harper San Francisco.

IEEE (1990). IEEE Software Engineering Standards, Standard 610.12-1990, pp. 47–48. standards.ieee.org/software/.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process.* Addison-Wesley.

Johnson, R. C. (2000). Time to Engineer DNA Computers. http://www.nature.com/embor/journal/v4/n1/full/embor719.html.

Kruchten, P. (2004). *The Rational Unified Process: An Introduction,* 3rd ed. Addison-Wesley.

Larman, C. and Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer,* 36(6):47–56.

Luger, G. F. (2004). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 5th ed. Addison-Wesley.

Maier, M. W. and Rechtin, E. (2000). *The Art of Systems Architecting,* 2nd ed. CRC Press.

Merriam-Webster (2006). *Merriam-Webster Online Dictionary.* http://www.m-w.com.

Parker, J. (2003). Computing with DNA. http://www.nature.com/embor/journal/v4/n1/full/embor719.html.

Polya, G. (1945). *How to Solve It: A New Aspect of Mathematical Method.* Princeton University Press.

Pressman, R. S. (2000). *Software Engineering: A Practitioner's Approach.* McGraw-Hill.

Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly.

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. In *ICSE '87: Proceedings of the 9th International Conference on Software Engineering,* pp. 328–338. IEEE Computer Society Press.

Rumbaugh, J., Jacobson, I., and Booch, G. (2005). *The Unified Modeling Language User Reference,* 2nd ed. Addison-Wesley.

Turing, A. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf.