Patterns for Story Craft

David West[1], Jenny Quillien[2]

[1]New Mexico Highlands University (Department of Computer Technology), Las Vegas, New Mexico, USA

[2]New Mexico Highlands University (School of Business), Santa Fe, New Mexico, USA

profwest@fastmail.fm, jenny@jqsolutions.org

Abstract. *Stories are pervasive and central to all software development methods, but, curiously, the purpose of stories (also known as Scenarios and Use-cases) as well as the skills required to create them are seldom discussed. This paper offers a set of patterns that will help both developers and users hone the competencies necessary to generate productive and powerful stories. The underlying argument made is that perhaps THE essential skill required for software development is the ability to 'tell a good story.'*

1. Background

Story Telling is so ubiquitous, both in software and the rest of our lives, that we seldom give it much thought; the creative writing class seems far removed from our own activities. However, if you think about it for a moment, we do point to the story as an essential tool:

•Scenarios and Play Scripts have long been advocated for business strategic planning, innovation, and to devise responses to change.
•Scenarios—stories about objects working together to accomplish tasks—have always been an essential tool for 'requirements discovery and documentation.'
•Object-orientation uses stories to define and design objects as well as for capturing how objects interacted to complete tasks.
•Use-cases are formalized stories. UML provides formal diagrams for capturing and representing stories.
•Agile relies on the User Story to frame a unit of work and to limn the nature of the work to be done.

Storytelling, since time began, has been the mainstay of mankind when it comes to making collective sense of life, teaching, entertaining, and acculturating newcomers. Stories are the backbone of human cognition. Most of what any one of us has learned, retained, or communicated uses a story as medium. What we learn without the support of stories generally results from sheer perniciousness and rote memorization—care to remember your painful days with multiplication tables and irregular verbs in Latin? Stories, by definition, are memorable.

Intelligence is about planning. Stories allow us to coalesce a complex set of facts or intuitions into a coherent whole—scenario planning for multiple plausible futures, for example. Intelligence is about integration. Story based memory preserves the connectivity of events and permits us to extrapolate from what we have actually

experienced. For example, we may have never actually seen a panda but if we are told that pandas are mammals then we 'know' that pandas give live birth to their young.

Intelligence is also about thinking less whenever possible. Stories provide scripts and skeleton stories which means we do not have to re-invent the wheel over and over again. Scripts are stories with a kind of memory structure we can call on. For example, when entering a dentist's office, a restaurant, or blog site, we can, almost on 'automatic pilot,' play our part in a fairly standardized interaction. The more scripts we know the less effort we exert, but there is the caveat of the ever present danger of assuming more than is true and missing out on new insights.

Story skeletons allow us to quickly dress our experiences onto a culturally common skeleton. Consider, as does Roger Schank, the skeletons of divorce stories. *We grew apart. All men are shits. A younger bride bolstered his failing manhood. We only married for sex and then got bored. He left me for another woman,* or with Woody Allen's twist, *my wife left me for another woman*. With skeleton stories, the same danger of complacency exists—we all have acquaintances with politically immovable attitudes, exasperating us by constantly force fitting new stories onto their old tired skeletons without a second's thought.

Also, stories can sometimes be compressed to captions (pattern names for example). Memory rich stories versus stories with memory succinctness offer a trade-off between multiple labels, or ways of referring to a story, with general applicability, and a few labels with specific applicability.

And, intelligence is about wisdom, knowing which of many stories is the most relevant to use, pertinent to our audience, or, perhaps, paradigmatic for action.

We would like to add to these points raised by Schank that stories are essential to the programmer as a professional. Schools provide each generation of developers with *techne,* universal and 'thin' knowledge such as mathematics, but it is *metis* knowledge (from the French noun *métier*, meaning craft or trade), rising from experience, from doing slightly different but related tasks, that gives us the thick deep knowledge of accomplished developers. *Metis* knowledge, the 'nose' for problem seeking and problem solving, will be understood and transferred in story form.

In essence then, stories are effective, powerful, easily remembered, high bandwidth tools of problem solving, entertainment, and information transfer, whose force resides in their primarily evocative nature. Even the simplest story, operating like a vacation snapshot, will bring to mind a dense association of other stories, circumstances, and knowledge. That's why we like them, tell them, and remember them.

1.1 Caution and Caveat

Stories do have a dark side and we have alluded to some problems with stories already, but explicit mention of some downsides to stories is warranted before we proceed. First, stories always have a narrator (often implicit) and therefore a specific perspective or point of view. In *Moby Dick* the hunt for the great white whale is told by Ishmael and from his perspective. We do know something about the story from Ahab's point of

view as he is allowed some leeway to tell parts of the story. But we know nothing about what the White Whale thought, or how the story played out from his (or her?) point of view. Stories deliberately highlight some aspects at the cost of leaving other aspects in shadow. One novelist who played with this theme is Lawrence Durrell, his series *The Alexandria Quartet* tells the same story from the perspective of four different narrators.

Another problem arises from the fact that stories offer us shortcuts in our thinking coupled with the fact that a story can be reduced to an icon. When this happens we dissociate from a story in a way that is not usually done when we deal with the full narrative. For example, take the caduceus, the symbol associated with medical doctors (mostly in North America, but frequently in other parts of the world as well). The caduceus has a rod, two entwined snakes, and wings. It is the symbol of Mercury (Hermes), the god of commerce. From ancient times, medical practitioners identified themselves using the Rod of Asclepius, one snake, no wings. At some point the two symbols were conflated and the more 'dramatic' caduceus came to symbolize the story of medicine. For anyone who knows the real story, the caduceus implies that doctors are merchants.

One last caution related to the fact that stories have perspectives. All of us are confronted with a bewildering array of constantly shifting 'data' in the form of sensory inputs. Our minds 'make sense' of this data using stories. There is no 'test' that can confirm our stories as accurate or true. The single most compelling argument for the 'correctness' of a story is 'self-consistency.' Is there a consistency across all the stories we know, hear, and tell? If yes, then we believe we have 'got it right.' But sometimes the same data can be interpreted and made self consistent in different ways—one reason we have multiple religions and religious wars. In extreme cases, we see individuals, like those diagnosed with paranoid schizophrenia, who have constructed stories that explain all the data, consistently, to form a totally idiosyncratic story of all that is and why it is. Individuals with such completely idiosyncratic stories are deemed 'mentally ill,' but it is a common observation that artists, geniuses, and innovators all share a trait with the paranoid schizophrenic: an ability to take the same 'data' and tell a very different story that 'makes sense' of it.

2. Theory

Peter Naur's breakthrough insight was that software development is not about building an artifact to specification, i.e., a deterministic, mechanical production process compatible with the old Cartesian dream of a grand narrative into which we could fit the entire world. Rather, Naur argues, software development involves collective 'Theory Building' which accommodates a dynamic, always in flux, set of multiple and overlapping partial stories about what's going on. A theory, for Naur, is an understanding of, "*an affair of the world and how the software would handle and support it."* Both a comprehensive understanding of the domain and the software implementation are part of Naur's definition of theory.

In other words, a theory is really a collection of stories (*Figure 1*), short narratives that we tell to remind ourselves of what we know and what we believe about the world and the things in that world. We also have, through stories, an understanding of values, behaviors, and rules of interaction in myriad circumstances. This theory is mostly non-

conscious, but is recalled as needed, via the telling of a story or recognition of an icon (for example, the Christian cross or the pharmaceutical icon of chalice, snakes, and wings) or other signal that represents a story.

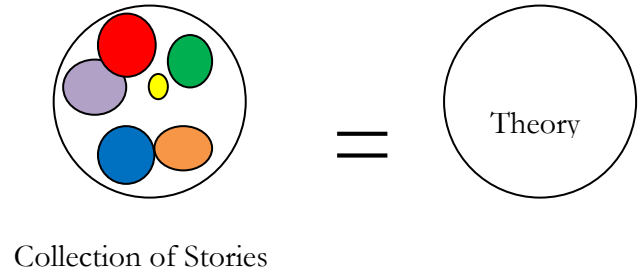


Figure 1. A Theory is a collection of Stories

Stories lead us to natural domain decompositions and to the simple, effective, and flexible design of actors (some of which are automated) in that domain—thereby realizing the long sought but elusive goal of IT/Enterprise Integration.

3. Ten Patterns

The set of ten patterns presented here elaborate and explain the concepts of theory. They are offered to provide a foundation that will help the 'Whole Team' become proficient in the ability to create and share the stories upon which every software development method is dependent. In the figures used throughout the essay, the patterns are represented by labeled rectangles; each pattern, name in ALGERIAN FONT, is described with an abbreviated version of the 'canonical form.' Although we facilitate the introduction of these patterns by presenting each one independently, the reader will soon realize that they complement and support each other in various ways. The summary offers a composite overview.

Let us start with these three (shown in *Figure 2*):

•*Stories provide themes* which, in turn, provide the underlying content of patterns, a sort of generalized PATTERN itself. This is certainly well known to most programmers so we won't dwell on this aspect.

•*All stories have structure and content*, FLESH AND BONES.

•*Stories contextualize and inform each other*, connected as it were in a large INDRA'S NET. For example the familiar Christmas stories of *The Immaculate Conception, Joseph and Mary Travel to Bethlehem, The Three Kings Follow the Star,* are stories which contextualize each other and have their place within the larger Christian narrative.

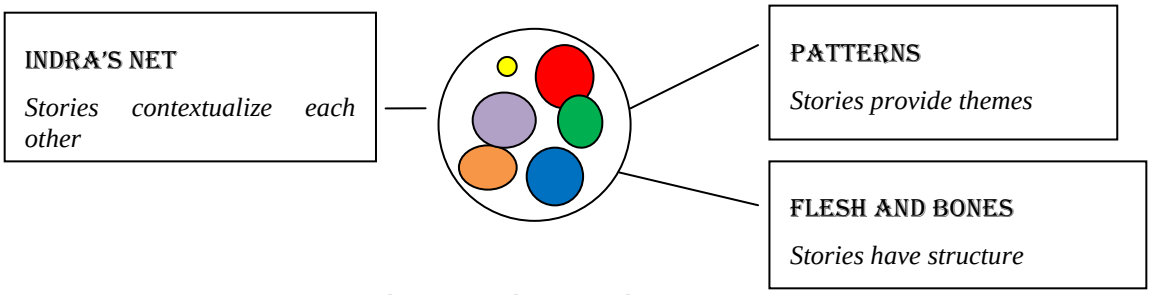


Figure 2. Three Basic Patterns

4. Pattern: *PATTERNS*

Context: Two different contexts are at play here. One context arises when you are seeking to better understand a domain. In this case you are looking for repeated examples of organization (structure) and dynamics (processes). You need to be sure your solution is congruent with these observations. A second involves the utility at the design and implementation (coding) level. Given the thousands of algorithms, millions of designs, and trillions of lines of code already developed, it is almost certain that somewhere there is a design or code solution that does 90% of what you want to do. One of your tasks is finding the most useful existing solutions.

Problem / Forces: There can be a lot of commonality among the various solutions found to different problems and the problems themselves. Also, there is a desire to avoid reinventing from 'the whole cloth' each instance of a solution. However, they are not clones even though solutions are related and have commonality; variations in the solution when applied in different contexts must be accommodated. There can be time pressure. Familiarity with the domain is a force at play.

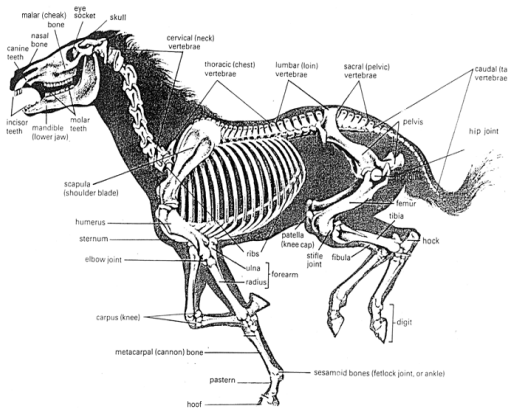
Solution: Patterns. A lot has been written about patterns, ranging from the general principles in Alexander's *The Nature of Order* to his more famous *A Pattern Language* and we all know that hundreds of patterns (design, analysis, organizational, and domain specific patterns) have been developed by the PLoP community. We will not go into this variation of story, except to note that patterns can be found at any level—from the principles governing the Universe to discrete algorithms in a block of computer code. We will also say that putting a pattern into more of a story structure instead of the 'canonical form' would probably increase their understandability as well as the ability of the community to remember a broad range of patterns and recall them to mind when useful.

Consequences: Positive consequences of patterns are shortcuts and time saving means of thinking and of implementation. Some negative consequences could be mistaking form for substance, and opaque pattern names (e.g. Visitor and Flyweight) that do not have the same evocative power as, say, Façade, or other patterns from the *Design Patterns* book.)

A very real potential problem is that as the number of patterns increases so does the search space for the most useful patterns, approaching the scope of the original search space, i.e., millions of lines of code. This problem could be alleviated (but not solved) if there were some way to clearly identify synonym patterns and categories of patterns, all within a single repository.

Example: You work in an office and handle a lot of paperwork. Some of that paperwork involves documents that are used to collect information. Each has a separate title – application form, permit form, qualification form. As you work with these you detect a lot of similarities among them – the fact that they are all called 'Forms, for example. Frequently you have to create a new document of this type and would like a 'template,'i.e., a starting point that captures all of the similarities among the documents. So you look closely at the similarities and try to identify a single abstraction that they have in common. After a bit of thinking and observation you can conclude that all Forms share a common pattern: they are an arrangement of text strings and entry fields within a bounded space. You can then use this pattern to create an electronic form generator with four classes of object: the form itself provides the bounded area, Text Strings, Entry Fields, and Validation Rules.

5. PATTERN*: FLESH AND BONES*



Context: As the opening of our essay asserts, stories are useful in any and every context—from the most general understanding of the world around us to the literate code that Knuth argued we should be writing. By extension, good story 'structure' is applicable in any context.

Problem / Forces: A story artifact must have some kind of structure or definition (bones) and must resolve to some degree the dynamic tension between the dominant forces of specificity and ambiguity (flesh). Specificity is required for us to be able to discern a path forward towards the implementation of some bit of software. Ambiguity is required to allow creative latitude with regards to exactly how we should move

forward. Specificity captures constraints but ambiguity provides innovative and adaptive potential. There is also a force towards oversimplification of a story, turning it into a mere specification or statement of a requirement. This force tends to trivialize storytelling.

Solution: A 'good' story, at minimum, means a full cast of characters, a plot, a description of interactions among the characters to advance the plot (script), cues, and outcomes. Consider how, in a personal letter, Robert Louis Stevenson couches his description of his illness in story form with characters, plot, and interactions which advance the plot:

Robert Louis Stevenson's Battlefield

> *"For fourteen years I have not had a day's real health; I have wakened sick and gone to bed weary; and I have done my work unflinchingly. I have written in bed and written out of it, written in hemorrhages, written in sickness, written torn by coughing, written when my head swam for weakness; and for so long, it seems to me I have won my wager and recovered my glove. . . I was made for a contest, and the Powers have so willed that my battlefield should be this dingy, inglorious one of the bed and the physic bottle."* (source) William Gass, *Habitations of the World*, Cornell University Press, 1985.

Figure 3 shows the structural relationship.

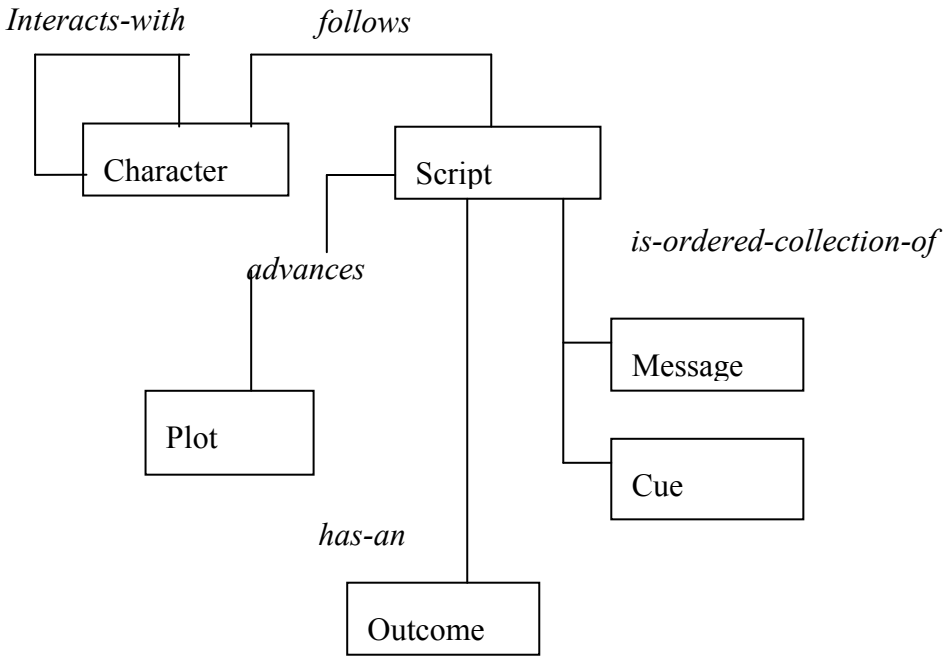


Figure 3. Graphical representation of story structure

All characters are objects. We use the term ‘object’ not in the sense of a programming construct, but in the sense of anyone or anything that provides a service to others. In the case of ‘things,’ we anthropomorphize them and talk about them as if they were animate and intelligent, i.e., as if they were human. This is an old technique—the behavioral driven object design technique.

By plot we really mean purpose, why the objects are interacting in this story, what it is that they hope to accomplish, why they hope to do so, and relationships to the circumstances that make the purpose meaningful.

Interaction equals dialog or script. Stories take place in the real world, not inside of a computer and all of our actors (remember we have anthropomorphized all our inanimate entities) are dependent on message—dialog—to communicate and to respond to communications. A cue is the only kind of communication that is not a message. A cue might be a simple signal, like the phone ringing. (Or, if your story is at the level of programming, an interrupt.)

The structure is echoed in any kind of story, from a corporate mission statement to a block of code. The elements may be simpler or more complex, but they are there. If your code is written as a story, with this structure, it will be readable by other human beings—as literature.

A good story should also ‘leave something to the imagination.’ There should be some degree of ambiguity in a story, leaving open the possibility of different resolutions, different interpretations, in other words different ways of indexing, matching, and retrieving data. We have all had the experience of reading an engaging novel and then been disappointed by a film version that seemed ‘poorer’ with less evocative power than the book we had so enjoyed. When we use stories to support the development of software, this ambiguity opens the door to different implementation solutions—different conceptions of what the design should look like (as in the concept of ‘late binding’). Consider a shopping story: People come to your commercial website and need to quickly find out if you have one or more products that will satisfy their desires, be able to compare options if more than one exists, perhaps get some technical questions answered, and make a selection and purchase. Each phrase of this story clearly defines something that must be satisfied but opens a multitude of different ways to do it. The very first task, the search for potential products, opens ways to organize your site, ranging from alphabetical to topical layout to emulating the floor plan of your bricks and mortar store.

One way to preserve ambiguity is to use (as metaphors) natural language variable names and message/method names. For example, in a story you would use the phrase, "*where are you?*" as the label for a message sent by the control tower to an airplane rather than the eventual technical method label which might be the terse, *"location."*

Ambiguity also spotlights those elements of our solution most likely to change with changing context; i.e., it helps us find those places where we can say, “*today, it makes sense to do it this way, but the solution might change in different circumstances.”* Consider the shopping story again. Whatever search mechanism is used, it will need to be variable—in response to whether the customer is a repeat shopper (you can customize how they ‘see’ your store based on prior shopping history), has a specific

→

product in mind, has a vague notion of a need but is sure there is a product that will satisfy that need, or has extensive experience shopping in your physical stores, etc.

Consequences: Agile was originally supposed to be about *exploratory* iterative incremental development. Tim Cahill, in *Jaguars Ripped My Flesh* (a book on travel writing), notes "*The explorer is the person who is lost."* Ambiguity is form of being lost—and it is great fun and absolutely key to creativity.

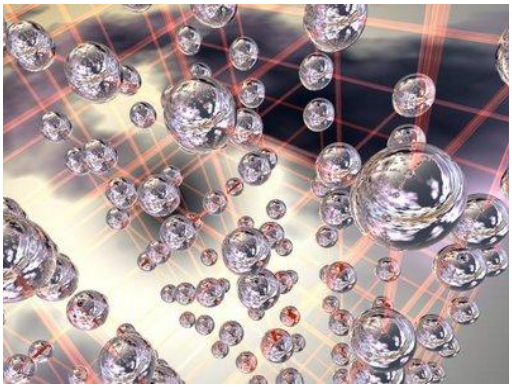
Tom Kelly, in *The Art of Innovation*, talks about the 'designer's brief,' the equivalent of the software developer's specification or Agile's User Story. He notes that the brief must strike a fine balance between enough information and sufficient constraints to guide the designer without having so much that there is no opportunity for interpretation or innovation. Agile stories are supposed to provide the same thing—enough information to proceed without tying our hands. [It is interesting to note that the polar opposite of Agile—Model Driven Architecture (MDA) is all about the 'perfect spec,' the model that is so complete and formally defined that nothing is required except some mathematical transformations to turn the model into code.]

Ambiguity also reflects an important principle of the Lean Product Development approach—defer the hard decisions (and the irrevocable decisions) until the last possible moment. Eventually you will be forced to tell stories where the ambiguity is resolved. In fact, it cannot be otherwise as the computer has so little imagination it cannot handle ambiguity in even the minutest detail.

Of course, ambiguity can be dangerous. A refusal to provide information that might be key to resolving ambiguity as in the infamous 'need to know' principle at the heart of governmental secrecy policies is a prime example of such danger. The resolution of ambiguity might also be an arduous task, and therefore never completed. For example, a hundred years ago, Chinese civil service exams would include questions about governance, citing a Confucian aphorism. The answer to such questions involved more than disambiguating Confucius, it also required meta-knowledge of obscure poets that lived several hundred years before Confucius. Your only clue would be an anomalous character or metaphorical allusion. The danger is that the meta-knowledge required might itself require meta-knowledge, with no resolution but an infinite regression à la INDRA'S NET.

Example: You are building a personal web site. Your first attempt focuses on the home page where you have just a few elements (characters) that give only a general hint of who you are. These elements are mostly links to other pages, as yet undeveloped, such as links to: my writing, what I read, my hobbies or personal life, my teaching, my consulting. When you add pages you are extending the story you are telling about yourself. The page devoted to 'my library' reveals how you classify your reading – serious, software professional, intellectual, anthropologist, and, of course the "guilty pleasures" of detective novels and science fiction. This is a form of character development with each new element of the page revealing another aspect of your character. Each page introduces new plot lines as well—invitations to the reader to interact with you, perhaps by making comments (as in a blog) or sharing their knowledge with other potential viewers of the site.

6. Pattern: *INDRA'S NET*



Far away in the heavenly abode of the great god Indra, there is a wonderful net which has been hung by some cunning artificer in such a manner that it stretches out indefinitely in all directions. In accordance with the extravagant tastes of deities, the artificer has hung a single glittering jewel at the net's every node, and since the net itself is infinite in dimension, the jewels are infinite in number. There hang the jewels, glittering like stars of the first magnitude, a wonderful sight to behold. If we now arbitrarily select one of these jewels for inspection and look closely at it, we will discover that in its polished surface there are reflected all the other jewels in the net, infinite in number. Not only that, but each of the jewels reflected in this one jewel is also reflecting all the other jewels, so that the process of reflection is infinite. (source) F. H. Cook: *Hua-Yen Buddhism: The jewel net of Indra. 1977*

Context: Anywhere you encounter stories that need disambiguation because the semantic meaning of their parts (words) requires other stories to 'set the stage.' Take the famous example: *"The man saw the woman in the park with the telescope."* This sentence can only be disambiguated by telling stories about, perhaps, the man visiting Griffith Park (the one in LA with the famous telescope), or the story about the lonely recluse whose only human contact came from observations of others in the park across the street from his penthouse.

Problem / Forces: Ambiguity is a necessary and powerful force for creativity and innovation. Our pleasure in joke telling is the discovery of multiple meanings: *If I said you had a beautiful body, would you hold it against me?*

In some cases, ambiguity is a symptom of a lack of clarity and meaning, resolvable through more precise syntax, more specificity in word selection, phrasing, and detail: *Comprehension of the problem was greatly improved by oversight.*

In most cases, however, the ambiguity can only be resolved by taking into account the context (contextualizing stories) in which the ambiguity is embedded. Context has much to do with comprehension and interpretation of information and the wisdom involved in choosing the most appropriate story.

Solution: The dependency on context to properly interpret any story leads to a massive combinatorial explosion of meaning/interpretation relationships, therefore create a

densely coupled web of stories, à la Indra's Net. Essentially you are constructing a single narrative that allows all of the specific stories to be seen in their shifting contexts. This is not as improbable as it might sound. Consider any Sherlock Holmes type mystery, or novels such as *The Historian*, by Elizabeth Kostova—a story of a girl's search for her missing mother. Each step of the story is explicitly developed by someone telling the heroine a story, about local history, about an event, about a communication. Also consider how our awareness of culture is via the telling of stories which provide the most natural domain decomposition possible.

Consequences: As a pattern, INDRA'S NET is a constant reminder to think holistically and to recognize that what is ambiguous now can be resolved by telling or discovering more stories which elaborate the context. This elaboration of stories as context can arise at any stage of a project. One of the more important insights of Agile is that storytelling must continue all the way down to code and the stories told to us by the code when it does what we expect and also when it doesn't. This is one of the ways we keep our eye on learning from failure and building intelligence based on our ability to predict.

We no longer have story specialists in our 'modern' times. As a consequence, all of us need to be aware of the entire canon of stories that define our culture. Anytime a small group becomes 'keepers' of a story or set of stories, they will be the only ones that know and understand those stories. If they disappear, so to do the stories. This is a direct analogy with Theory as described by Naur. As he pointed out, Theory lives almost entirely in the heads of those involved in its development. Break up the group and the theory goes away; it even dissipates in the minds of those who originally had it.

If the community possessing the theory were enlarged, say, if everyone in the IT department possessed the theory of all the software generated by that department, then the theory would persist from discrete project to discrete project. It would also be integrated—every bit of software would reflect the same general theory instead of being the idiosyncratic product of the particular team involved in that narrow effort. A widespread sharing would mean that even with natural flow of people into and out of the department, a critical mass would remain capable of maintaining the theory and conveying it to newcomers in fairly short order.

And if the Theory were expanded still further, to encompass everyone in the organization – business and IT – you would have 'One Theory to guide them all.' In fact, the ability to create a single theory is the only way that the long sought goal of IT/Business integration will ever come to be.

All that said, take care not to fall into the centipede's dilemma (thinking about which foot came next) when thinking about your net of stories. Holistic thinking is different than trying to 'trace all the links.'

Example: We have developed a lot of debugging techniques over the decades, all based on a kind of linear trace of program execution: *The program is here, and my variables have this value. Then the program does this, which moves it here and changes the value of that variable to this.* All of this is fine, until you start working with a language like Smalltalk – where the work is distributed across multiple objects, includes all kinds of objects that have nothing to do, at least directly, with your program. Trying to trace the actual thread of execution – the way you did in a procedural program – is like trying to

trace a single thread of conversation in a group discussion. The conversation does not take a linear direction, going from this speaker to that speaker, in order. Instead, many people are talking, often in parallel, and taking parts of the conversation in unexpected directions. In Smalltalk the flow of execution can involve actions by all the objects in the image, including tons of objects that you did not create and of which you were perhaps totally unaware. It is amusing – but not funny – to watch novice Smalltalk programmers become totally lost when trying to apply their linear, procedurally programming, debugging techniques.

7. Different Kinds of Stories

Let's now look (*Figure 4*) at how *stories come in different kinds*, so to speak. We'll address some of the main kinds of stories moving from larger scale general stories to more concrete and specific ones.

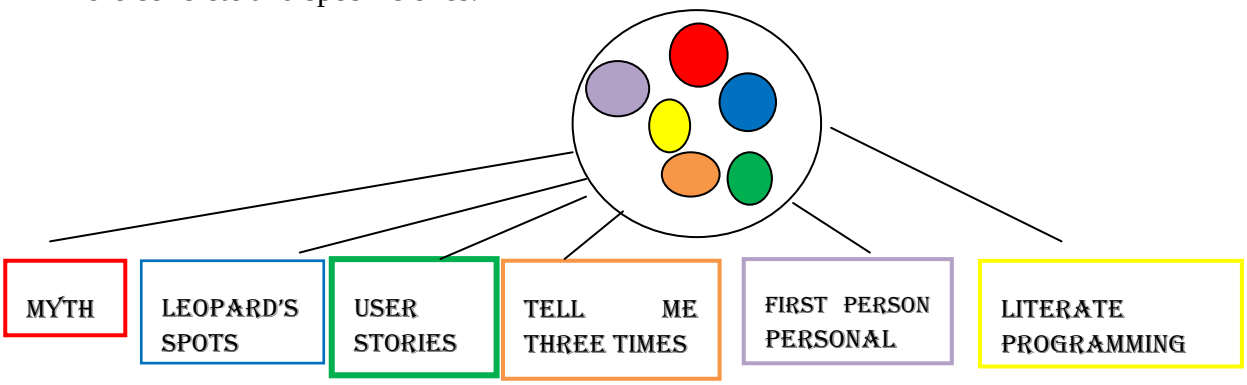
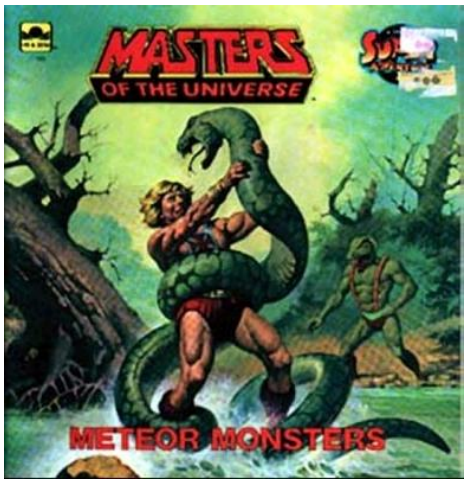


Figure 4: Different Kinds of Stories

8. Pattern: MYTH



Context: Whenever there is a process of reifying and anthropomorphizing a community there is a need for myths. Myths are not (totally) fiction; rather, they are very general stories that illustrate proper behavior or action and/or overarching values and principles. Myths employ epic characters (organizational founders or other exemplar figures) and have plot outcomes that exemplify a value or principle.

Problem / Forces: Everyone on a team, or in an organization, requires guidelines for what to do, how to act, how to be a proper member. They need to know The Rules. The Rules cannot be captured in any precise form, nor can the entire set of rules be delineated. There are always exceptions, which apply meta-rules for when to make such exceptions, and meta-meta-rules for when the meta-rule does and does not apply. This is an infinite regress—hence the bold claim for impossibility. Rules change as a function of the situations in which they are applied. More importantly, rules change when the people subject to those rules decide, by their actions, that a rule should be modified, discarded, or made more specific. There is, in fact, a reciprocal relationship between rules and human behavior: rules shape behavior; behavior shapes the rules. People need the latitude to 'interpret' rules but they need enough of a principle or guideline, and explicit examples of how others (preferably gods and heroes) have interpreted the rules in different circumstances.

Solution: Myths must be told. They contribute to our 'wisdom' when choosing a course of action, to our interpretation of events, and to our re-examination and revamping of our old stories.

Start with telling a Mission Statement myth, a succinct statement of organizational values and intent, that everyone knows is 'fictional' but is nevertheless a key guideline for developing policy and strategy.

Make liberal use of very terse myths or adages, like the physician's, 'first, do no harm' principle. [Note that even an adage reflects the structure of a story. Characters: physician, patient. Interactions: treatments. Outcome: no harm.]

Use Hero myths or founder stories, myths that are frequently used to illustrate values and practices. Seymour Cray, it is told, built a boat every winter. He would sail it all summer and learn as much as possible about how it performed, then burn it to the ground in the fall. He would then use what he had learned to build a better boat the next year. Whether the story is true or not, it is a myth in principle because the point of retelling it is to offer guidelines for the behavior of Cray employees: do your work but constantly reflect on and improve what you do without getting attached to particular processes or practices; tools or outcomes. Develop a love of learning.

Consequences: Myth equals the set of stories that limn an organization's world view, values, principles, and expected behaviors. Myths are far more powerful than direct instruction. Think about the difference between a simple dictum, "take risks and learn from your mistakes," and the story told about Thomas Watson, CEO of IBM: During an interview, Watson was asked if he intended to fire a vice-president who had recently made a mistake that cost the company millions of dollars. Watson's reply: "*Hell No! I just paid for his education."*

Myths can also do harm. One pernicious corporate myth is that of the Titan of Industry, or the Wall Street 'Master of the Universe.' Not only does this kind of myth justify obscene salaries and total disdain for the public, it also is responsible for the expectation that Information Systems must be monolithic and integrated. The mythic image of a Master Control Panel (as front-end of that kind of monolithic system) at the fingertips of the omniscient CEO (who will use it to steer the company to dizzying heights) is a very real myth in U.S. Corporate culture.

Example: Myths, as the 'Titan of Industry,' shape all of the other stories we tell in an organization and guide decisions made for good or for bad. Most companies share the following common myth about themselves. This myth depicts the company as a 'machine' that takes raw materials of some form and turns them into finished products or services that are sold for more than they cost and therefore generate profits. These "machines" are 'controlled" and "managed" by individuals, with at least one such individual being the top dog, the master controller. When we first made the transition from automating basic business operations (e.g. payroll) to creating "Management Information Systems" all of our efforts were shaped by this myth. We assumed that MIS involved collecting certain bits of data, packaging them in a form that made them information, and feeding them to a manager who would digest, deliberate, and direct – issue orders to the machine to do X instead of Y, or to increase the rate of A. This myth was powerful enough that it actually defined the architecture for individual programs. The Program Structure Chart (PSC) with its "master control module" at the top, with afferent modules to the left and efferent modules to the right, and transform modules in the middle – all creating their own mini-pyramids of command and control. Eventually this myth led to the creation of monolithic 'integrated applications," to the detriment of the business, all notions of simplicity, and the ability to match the adaptability demands of the organization with the adaptive capabilities of the information system.

*9. PATTERN: **LEOPARD'S SPOTS***

The leopard and the Ethiopian were great hunters on the veldt where their tawny solid color blended into the background. The game, zebras and giraffes and other succulent creatures so feared the hunters that they moved to the forest – a place filled with splotches of light among patches of dark. The prey animals with their stripes and patches of brown could not be seen by the leopard or the Ethiopian who immediately recognized that they needed to change their appearance to blend into the forest. The Ethiopian went first, changing his skin to a lovely black and dark brown with subtle hints of purple. The leopard then asked the Ethiopian for help in changing and the Ethiopian replied that he had some black left over so he used his five finger tips to the leopard's skin leaving the distinctive five dark spot pattern that the leopard carries to this day. (source) Rudyard Kipling's, '*How the Leopard Got His Spots."*

Context: As children we learned about our culture through "*Just So*" stories: specific stories that explained why 'things came to be the way they are.' A fine example is Rudyard Kipling's, '*How the Leopard Got His Spots."* Leopard's Spot stories in an organization or in the world of software are less whimsical than Kipling's but have the same purpose—to expose our understanding of why things are the way they are.

Problem / Forces: Why something is done, or why a particular decision is made can be highly circumstantial and the circumstances rapidly fade from memory. People, and organizations, are creatures of habit—once they start doing something in a particular way they continue, often long after that way of doing things ceases to be useful. At some later time a behavior can be seen as incompatible or erroneous and in need of change, but we dare not change it because we do not remember why it is the way it is.

Solution: A Leopard's Spot story allows any decision, practice, design, or implementation to become essentially 'self-documenting' by encapsulating the meta-data about an entity with the entity itself. As stories they have far more power than simple assertions, "*that's just the way it is,"* or *"because Mommy said so."* Leopard's Spot stories are told when you want to understand why something is being done the way it is and what decisions and circumstances led to a given practice. The object is not merely to explain why, but to do so in a manner that invites the asking of 'why' and 'what-if' questions, i.e., critical 're-indexing' and labeling of information. You may want to find a different way to do things but do not want to run afoul of invisible cultural barriers to the new ideas.

Consequences: When we need to reconsider an existing element of our system, having a Leopard's Spot story allows us to understand what is and the circumstances that led to it. Simultaneously, it affords us the opportunity to compare the specifics of the circumstances then with the circumstances now and isolate the forces that are prompting the need for change. The comparison then provides the parameters and constraints that help us decide how to make modifications.

Watch out that you do not take Leopard Spot stories at face value and always remember that they reflect a 'reality' of a particular place, time, and storyteller. This type of story is of value only when examined – that is their purpose – and the unexamined story, like the unexamined life, is 'not worth living.'

Example: Once upon a time, Microsoft created an operating system for desktop machines which were relatively expensive and the exclusive property of employees. Employees, being very jealous people, did not want to share their valuable new toys and demanded a mechanism that would prevent others from using the computer. But they also wanted the computer to be easy to use, so it should remember things about the rightful owner and not require constant updating of messy things like passwords required in order to use specific applications. And that is why, when you use Internet Explore to access a secure application, it prompts you at quitting time to close the browser – to erase its memory of you – in order to prevent an evil-doer from taking advantage of the poor browser and its knowledge to gain unauthorized access.

10. Pattern: *USER STORIES*

Context: You have been asked to develop some software that will help a customer accomplish some task but are just as unsure of exactly what is needed, how to satisfy that (those) need(s) and are unsure of exactly what form the solution will take. In some cases you have an idea (somewhat imprecise) that 'the computer' will play some important role in the solution, but are not sure of what might be satisfactory computer behavior and performance.

Problem / Forces: Your customers know that they want 'something.' They may be able to describe part or all of what they want, but anything they say is subject to change. In many cases the customer has only the vaguest idea of what it is that they need; they may only be able to express the 'problem.' In almost every case, the customer has only the most nebulous ideas about what is possible—what technological solutions are available or the impact that a given technology will have on the way they do their work.

No amount of bullying, coercion, or formal requirements will make the customer able to specify completely and unambiguously what they want before development begins!

Multiple transformations will be needed as you journey from concept to executing code that yields computer behavior acceptable to the user. At each transformation step (idea, analysis, design, test, code, integrate, deploy) you will have to make decisions about what you are doing and why; and the customer must understand and find these decisions appropriate and acceptable.

Solution: Since there is no way to unambiguously and accurately state all requirements upfront and since there are no formal transforms that would take you from specification (even if possible) directly to executing software [The MDA people are wrong about that.], the way out is to 'explore and discover' a solution by disambiguating the need *and* the means for satisfying that need. A User Story initiates a three-way conversation between the customer, developers, and the computer (software).

USER STORIES, properly understood, also allow us to get away from the paradigm of the 'once and for all' set of 'requirements and delivery.' The IT/Business relationship requires an on-going conversation about unfolding collaboration in constantly changing circumstances. The simple graphical representation omits this critical aspect—that this conversation is constant— not a one-time event! The conversation is iterative, incremental, and somewhat fractal as all parties attempt to convey, disambiguate, and achieve a consensus understanding of the problem/need and how it is to be addressed.

Figure 5 shows a simplified model of what typically happens as a User Story develops (which hampers optimal usage of the technique.)

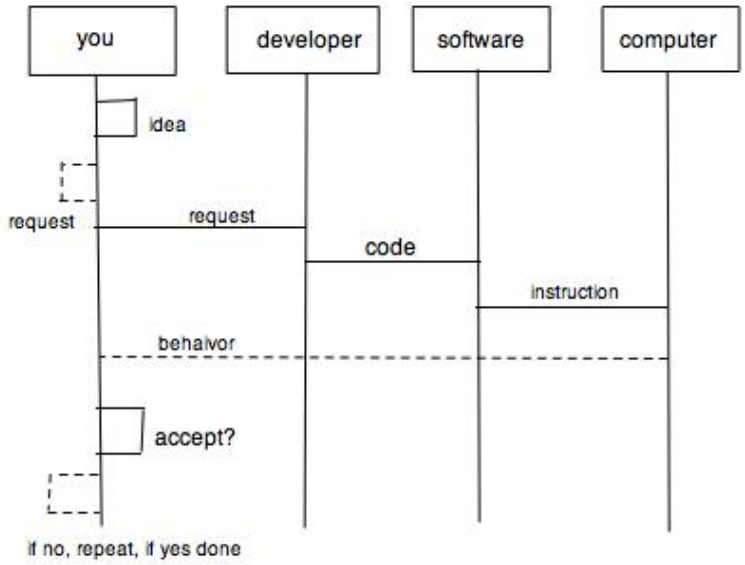


Figure 5. Erroneous but common representation of the IT/Business Conversation

Consequences: This pattern is fundamental to Agile, particularly XP. All of the practices, including the one named User Story, are intended to support this specific pattern as a dynamic, constant, cycle of *think* (about the problem) – *articulate* (a proposed solution) – *express* (the solution) – *analyze* (computer behavior / feedback) – *revise* / amend (your thinking) – *repeat* until satisfied. And with the constant reiteration, we begin to recognize a meta-pattern: a new pattern will create a new situation, possibly new problems and a need for a further round of patterns.

Example:

Client: *"You know what would be cool,"* he said while looking at the class yearbook. *"What if there were a way to get this pretty girl's phone number and address just by touching her picture?"*

Software Developer*: "I know how to do that. We will build this website and use a mouse click even to retrieve the information from a database that we can buy from the school – after all, demographic information about students must be public."*

Client: *"Neat. We will call the site Yearbook."*

Software Developer*: "Here it is, what do you think?"*

Client: *"Awesome, but you know what would be even better? We should allow each of these people to share other things about themselves, like the new photo they took after they lost that fifty pounds, or maybe a picture of the trophy that they won for fencing last year."*

Software Developer: *"We can do that."*

Time passes. . .

Software Developer: *"Here it is. All the person has to do is enter this code, request that the photo be converted from pqr format to dfv format, then crop it. And then enter these coordinates to have it appear on the page at the left margin."*

and. . .

Comment from the Module in FIRST PERSON PERSONAL: *"That seems pretty hard, you are thinking like a programmer. Why can't we hide all that and just have the user tell the picture to go here, and the picture figures out how to do any conversion and placement? You know, the drag-and-drop metaphor?*

11. Pattern: *FIRST PERSON PERSONAL*

Context: This pattern is used when you are engaged in design and the implementation of those designs—in other words, when you are concerned with modularization, decomposition, distribution of behaviors, and the code-based definition of your modules.

Problem / Forces: It is impossible to build a monolithic solution to even small-scale problems. It is always necessary to decompose solutions into a collection of interacting modules that are loosely coupled and highly cohesive. It is necessary to decide exactly what distinct behaviors and what information should be assigned to each module and also know how that module will be designed and implemented in a programming language.

You have two perspectives from which to make your assignment and design decisions: (a) from outside the module, or, (b) from the point of view of the module. The former will always lead to overly complicated, overly centralized, and overly controlled solutions that are difficult to evolve or adapt.

Solution: A mini-autobiography written in the First Person Personal. This means you pretend to be (liberally employing metaphors here) a sentient entity and want to know what it is you do; what resources you require to do it and how to get them; and, what others might want to know about you—both your characteristics and your state. You want to do a few things very, very, well, you do not like to boss others around, but you are rather lazy and would like to delegate really hard work to others. You want others to know exactly what they can expect of you, but do not want them privy to your innermost thoughts and actions. Use an Object Cube to succinctly and explicitly capture all that needs to be made public about yourself and, if appropriate, all that is required to implement you as software.

Figure 6. Object Cube shows your six facets.

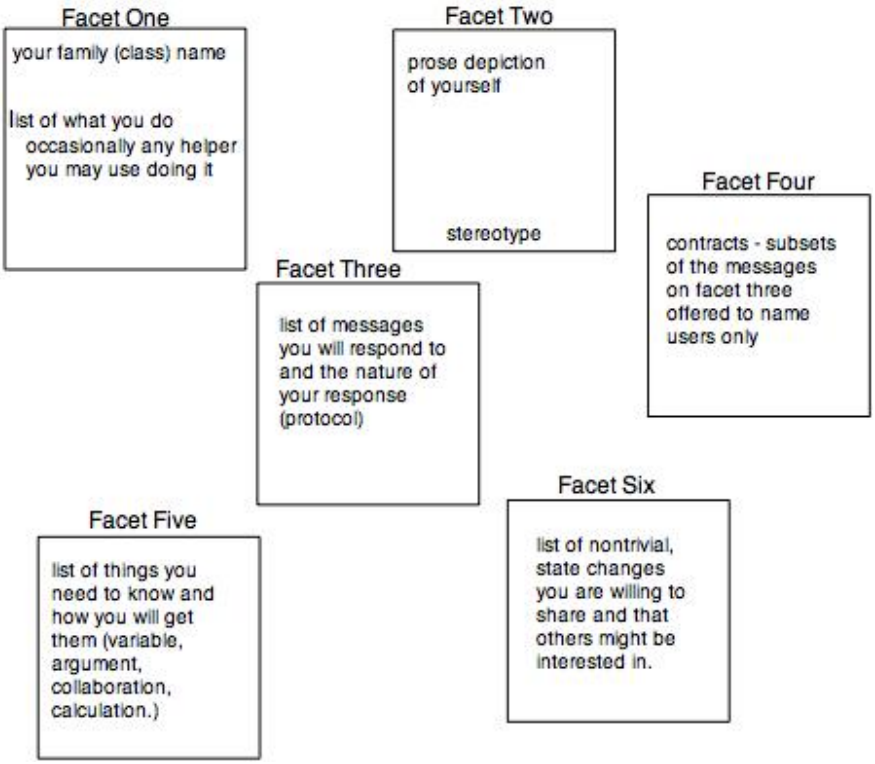


Figure 6. Object Cube

Story development begins (Facet One) with you (as the object) asking yourself what is it that you do, or should reasonably be expected to do. You can use the input of others—their expectations of you—as part of your thought process but ultimately you need to decide what you are willing to accept responsibility for doing. You capture these decisions in the form of a short list (after all, you do not want to be all things to all people, but do some few things very, very well). You can also think about each task you assume and consider how hard it is and whether or not you need an assistant, or collaborator as they are called, to help you with some part of that task.

A brief description of yourself in simple prose is always handy and can be on Facet Two. You should also consider if you are like other objects in important ways, and if there is an umbrella label for you and your fellow travelers. This is called a stereotype (Rebecca Wirfs-Brock's idea) and an example might be: you are a stock portfolio and you notice that much of what you do (add, delete, return subset of stocks) is identical to what collections do. So you, or at least part of you, could be realized by using an existing object called a collection. Collection would be your stereotype.

Specifying exactly what messages you will respond to, along with the exact syntactic form that those messages should take, is important (Facet Three). Occasionally you will decide that you will respond to some messages only from specific clients and you need to establish a contract to that effect (Facet Four).

Perhaps the most important part of your object story is looking at what you are willing to do (your responsibilities) and then deciding exactly what information you need to do that *and* how you will obtain that information (Facet Five). You have four choices: (a) save the information inside of yourself in a variable; (b) obtain the information as an argument to a message requesting a particular service; (c) obtain it via collaboration with another object—while in the process of responding to your triggering message; (d) or calculating that information on the fly.

There is a special kind of information that you will almost certainly be expected to provide, which is information about yourself, your description. A description is a list of attributes and their current values that tell others about your nature. A description should be isolated in a collaborating object—a kind of ordered collection probably. This is the single biggest distinction separating you from a mere data entity.

The final part of your story is deciding what changes you might experience that might be interesting to others and that you are willing to share with them (Facet Six). No one is interested in trivial changes, like a mere change in the value of a variable (with some exceptions); they are interested in changes that alter your nature or your abilities. For example, no one cares that the gallons of jet fuel, in you as an airplane, just went from 9000 to 8999 even though that is technically a change of state. They would be interested if the value went from 1 to 0, however, because that change affects your ability to fly. Others might also be interested to learn that you were just hijacked, or that you had a health emergency on board, because how others treat you will vary dramatically with that kind of change.

Consequences: If all objects are allowed to develop and tell their stories, you end up with an equitable distribution of workload across everyone involved (both automated objects and human objects or users). You also ensure that the right task is assigned to the right object/person and do not ask a human object to perform complex cubic calculations that would be better off delegated to a computer implemented object. You also realize the goal of simple and elegant code. Your objects have fewer methods and each method is implemented with fewer lines of code.

Perhaps the most important consequence of using this kind of story is achieving a level of verisimilitude with the domain that would otherwise be impossible to attain.

Another major beneficial consequence is the resulting simplicity of design and implementation for all objects. Instead of building large monolithic systems, you build

replaceable and easily updatable components. This allows you to easily change the overall system of the business and all the components that comprise that system.

FIRST PERSON PERSONAL is always used in conjunction with USER STORIES! USER STORIES help identify the objects and lay out the communications among objects involved in completing a given task. FIRST PERSON PERSONAL then allows you to define the objects themselves: extracting a communication from the user story and making responding to it a responsibility of an object—the first pass of object design is this kind of responsibility assignment. The USER STORY also provides an opportunity to ask if you are communicating with the 'right object'—maybe that one over there can do the job better and more easily?

A USER STORY can then be used by the object to make decisions about itself. Should I accept that responsibility (respond to that message)? If I do, will I need help (a collaborator)? If I need a collaborator should they actually be doing all of the work, or is it true that I am only going to delegate a subtask or two. If I do this what do I need to know? How can I get it: do I have to memorize it or can I ask for from somewhere else? These questions are always asked and answered in the context of several User Stories—all the User Stories in which a named object is present.

The vocabulary of the customer – of the domain – is always used when telling User Stories and completing object cubes. The mapping to more formal names for Classes, instance variables, the class of objects residing in instance variables (the Types, if you are an insecure novice programmer), message protocol, event names, etc., is done only after you are comfortable with the set of User Stories and object-cubes you have created. In this assignment you follow conventions and idioms of the programming language and development shop.

Of all the patterns suggested here, this one is the least likely to generate negative consequences—at least directly—and it is germane to the manipulation, indexing and retrieval of information. The most common, indirect, negative consequence is a feeling of 'being silly' by talking about objects as if they were people. Professionals often resist this lack of 'dignity' in object analysis efforts. When Dave taught object analysis, he semi-seriously suggested that the team, "meet offsite, begin the day with a pitcher of Tequila Sunrises, and engage in a few spirited rounds of singing *I'm a Little Teapot* before picking up index cards and doing analysis."

Example: Consider the previous dialogue given as an example of USER STORY as also an example of FIRST PERSON PERSONAL.

12. Pattern: *TELL ME THREE TIMES*

Context: The context for this pattern is known as the Sheherezade Principle, after a line in one of the Arabian Night tales, "T*ell me three times and I know that it is true*." [Most people are more likely to recognize, "*What I tell you three times is true."* as a line from Lewis Carrol's *'Hunting of the Snark.*'] Systems engineers at NASA coined this term in connection with the triple redundancy computing built into manned spacecraft. This is an implementation pattern that you use when you are deep into the testing-coding-build cycle within an iteration and have a keen interest in the intelligence skills of prediction and interpretation of failure.

Problem / Forces: Just as there is ambiguity in what you want, there is ambiguity in knowing whether or not you have obtained it. You have 'correct' feedback in the form of a computer behavior, but what if it was a fluke, just a one-time idiosyncratic result? Will it work next time? Will it work in a variety of similar situations, using similar information? There are a lot of unknowns, including interaction with other executing software, which could potentially be a cause of error conditions. Your shop has correctly adopted test-driven development as a development method.

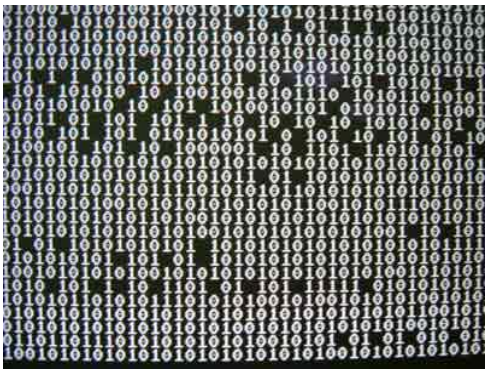
Solution: Devise a set of test stories describing a situation, expected stimuli, and expected responses. Tests are stories. Just as any other story, they involve characters receiving messages, within specific settings, and responding in observable ways. In some cases, a story might involve repetitive sending of the same stimulus / message to make sure that the response is invariable across time. Similarly, it might involve sending the same stimulus / message to a set of peers to see if all of them agree.

If you are a good TDD'er you write your test before you write your code. You then run your unit tests after writing your code (that's once), after refactoring (that's twice), and after integration or daily build (three times)—and, then, you can be fairly certain that the code is correct, 'true.' There are other triads, involving different kinds of tests that are also run three times: for example, acceptance tests are run at Story Card sign-off, during demonstration / retrospective day, and after deployment.

Because of its special circumstances, the world of testing, this is a rather minimalist pattern. It is minimalist also in the sense that a true tester could find numerous other patterns and probably longer sequences of tests than the simple triad 'floor' presented here.

Consequences: Thinking of testing in terms of stories and the existence of some sort of minimal level of testing accomplishes two objectives: (a) it keeps the Whole Team in the loop instead of delegating QA to some outside specialists, and (b) does so in a format with which everyone is familiar—the story. It also pushes the development team in the direction of defining a comprehensive set of tests (test stories), increasing the odds that you will devise the correct tests, sufficient test coverage, and tests that are understandable and interpretable.

13. Pattern: *LITERATE PROGRAMMING*



Context: Programming. Technically, each programming language is a separate context because different languages allow different kinds of expressiveness.

Problem/Forces: Computer instructions must ultimately take the form of binary numbers and primitive, hardwired instruction codes—a language that humans are not proficient with. 'High level' languages provide programmers with a more 'natural' language in which to express their desires to the computer. There is still uncertainty about exactly how the computer (and the compiler) will interpret the programming code—a non-trivial distinction between what was said in code and the meaning behind that expression.

The computer is not the only entity that must read, understand, and interpret the code. Other programmers, at different points in time or from other pairs or even other teams, will also need to understand programs.

Programming languages put severe constraints on your ability to compose—it would be difficult if not impossible, for example, to write a poem or a romance novel in FORTRAN (but easier in languages like Lisp or Smalltalk perhaps).

Solution: As Knuth has argued, write your code in the form of a story to be read by other humans. This means that your code will be in the form of a (somewhat stilted because of syntax restrictions) narrative story with a stated goal, messages sent to identified actors, and stated expectations of responses. Code stories should be intertwined with test stories—the way that some novels switch from different character perspectives and plot lines, only to come together in a final resolution.

Consequences: Program code is aligned, metaphorically and structurally, with everything else we know about our domain, problems in that domain, and software

solutions to those problems. [The most problematic aspect of old waterfall ideas about development was the lack of uniform communication and storytelling as development work progressed from one team to another. Each team, necessarily, reinterpreted the work products of the preceding team, leading to a divergence of intent from user to programmer.]

If your code and your tests are true stories, they are accessible to everyone; you are able to use a consistent format across the entire development cycle(s), and across the Whole Team, including users and managers.

Literate programming will require rethinking programming: what was simple idiom will have to be elevated to the level of standards and conventions. It will also force programmers to avoid using cryptic and terse names in their programs, whether message, variable, or method names.

Example: A humorous example of literate programming borrowed from Richard Gabriel and Kevin Sullivan:

better watchout
better !cry !pout
lpr why
sant claus <north pole> town

cat /etc/passwd > list
ncheck list
ncheck list
cat list | grep naughty > nogiftlist
cat list | grep nice > giftlist
santa claus <north pole> town

who | grep sleeping
who | grep awake
who | egrep ‘bad|good’
for (goodness sake) {be good}

14. Theory Building and Recall

We also want to consider how Theory Building revolves around individual and collective memory and powers of recall (*Figure 7*). A significant pattern, the final pattern in our essay, involves tools and crutches for supporting memory and recall.

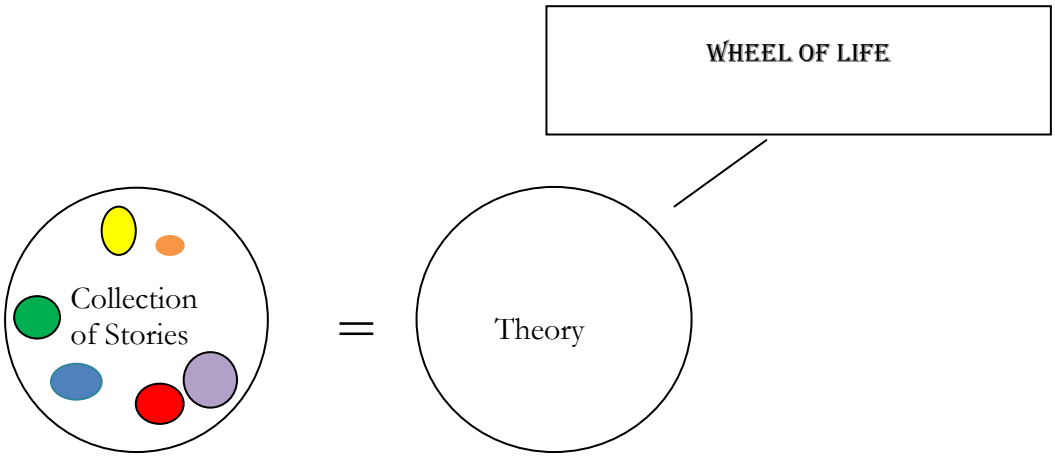


Figure 7. Remembering Stories

15. Pattern: *WHEEL OF LIFE*

Context: As Schank noted, if you do not have access to information, then, for all practical purposes, the information doesn't exist. Often there is often a need to provide 'instant recall' of large volumes of information for members of a community. Conventional cataloging and indexing schemes are too indirect and subject to categorization error. However, ancient visualization tools, such as the Mandala, common to almost every culture, have developed within this general context, allowing for the simultaneous depiction of huge amounts of evocative information.

Mandalas can be highly abstract so, for our purposes, the more representational form of the Wheel of Life is more useful. The Wheel of Life identifies the primal forces/values that drive everything—symbolically represented at the core of the figure. It also depicts, in an outer ring, dynamics, such as the stages of existence of a human life, or the structure of an organization, then, around the periphery, important but external forces and a multitude of icon, each recalling to mind a specific story along with its context.

Problem / Forces: A Theory à la Naur will consist of a large number of individual stories. Since each story is contextualized by all the others stories (INDRA'S NET), it is necessary to have all the stories at hand all the time. This does not mean that all the stories need to be in your conscious mind (something that is probably impossible) but you need to be able to recall any story at any time, often unpredictably. You simultaneously need a gestalt and a detailed view of the available stories and how they relate.

Simple indexing and standard search and retrieval methods are too fallible and too subject to human error. Typical collection indexing schemes—key words, set categories, whole-text—create as many problems as they solve. For example, is there any place or technique (e.g. Google search) that will allow you to find a set of patterns that might prove useful to you as you consider a specific domain, problem, or solution possibility?

Solution: Use a visualization tool, like the Wheel of Life Mandala so that you have an easily modifiable and maintainable visual depiction of the One Theory that can be developed in a highly distributable and sharable form. This pattern is useful across the lifecycle and satisfies the needs of the Whole Team to quickly communicate complex information and knowledge amongst themselves. This visual metaphor has been used with great success in the teaching of software architecture and in helping several different kinds of organizations bring all of their development teams to a common understanding of the business and IT organizations.

Consequences: Individuals can look at the Mandala and recall everything they know about software and how it supports the overall organization. They can also see, immediately, what it is that they do not know, simply by recognizing, "*I don't know what that icon represents, I have not heard that story yet."* Both individual stories and contextualizing relationships among those stories are simultaneously depicted.

This kind of visual metaphor is a supplement for the information radiators and white boards with ephemeral content that are deliberately and consciously produced in an Agile work environment. The purpose is the same: to evoke anything and everything that the Agile team knows about the work they are conducting.

Example: You have been asked to develop a comprehensive information system for a new insurance company. This is a large job and will involve writing hundreds of programs and making sure they all work together to produce the desired result. If you use a requirements document, you know it will be thousands of pages of extreme detail and take forever to complete and validate. Instead, you create a simple visual story using the Wheel of Life format. At the center put the forces that drive the entire company: a symbol for money, a person or object icon, and some sort of icon representing disaster or loss, a tornado perhaps. Then you create a larger circle and divide it into segments to represent the different parts of the company: sales, underwriting, claims, etc. In each segment you put an icon representing a single user story, like "Filing a Claim." Then create an outer ring and use a series of icons in that ring to depict the primary business flow – identify a prospect, complete an application, assess risk, build and deliver a policy, collect premiums, take a claim, pay claim, cancel policy because customer was irresponsible enough to use it, and so forth. Put this simple diagram, on a white board so it can be modified and extended at will; providing a powerful metaphor of the entire system in one place. As each new icon is added, it

serves as a reminder to everyone of what that icon is about and what is currently known. This Mandala then severs as your primary design and communication documentation.

16. Conclusion

We asserted a need to establish a comprehensive single, unified, general theory of the enterprise and of the software developed in support of that enterprise. We suggested that the stories provided a powerful, commonly understood, and universally employed means to develop such a theory. We then offered a small set of briefly described patterns to facilitate the development of story crafting skills by business and computer professionals. To facilitate the introduction of these patterns we presented them separately and 'in single file,' but clearly they work together, overlap, and intertwine when we actually put them to use. MYTH, LEOPARD'S SPOTS and USER STORIES all explore themes. PATTERNS, FLESH AND BONES and INDRA'S NET provide structure and context. FIRST PERSON PERSONAL and USER STORIES work hand in hand. INDRA'S NET and FLESH AND BONES deal with different but overlapping aspects of ambiguity. The WHEEL OF LIFE helps us visualize INDRA'S NET which is composed of larger (MYTH) to smaller (TELL ME THREE TIMES) patterns. FIRST PERSON PERSONAL and LITERATE PROGRAMMING help us reveal more clearly the PATTERNS at play.

Of course, stories, like any powerful tool are subject to misuse, but with increasing competence in storytelling we escape our overly mechanistic and 'Cartesian' mindset and become explorers of 'the affairs of the world' we need to support.

To quote ethnographer Clifford Geertz, our trailblazer is such techniques, "*. . . as a simple matter of empirical fact, our knowledge of . . .a culture. . .grows: in spurts. Rather than following a rising curve of cumulative finds, cultural analysis breaks up into a disconnected yet coherent sequence of bolder and bolder sorties. Studies do build on other studies, not in the sense that they take up where the others leave off, but in the sense that, better informed and better conceptualized, they plunge more deeply into the same thing. Every serious cultural analysis starts from a sheer beginning and ends where it manages to get before exhausting its intellectual impulse. Previously discovered facts are mobilized, previously developed concepts used, previously formulated hypotheses tried out; but the movement is not from already proven theorems to newly proven ones, it is from an awkward fumbling for the most elementary understanding to a supported claim that this has been achieved and surpassed."* (source): Clifford Geertz. *Toward an Interpretative Theory of Culture*, New York, N.Y: Basic Books, 1973

Figure 8 consolidates our discussion into one overview.

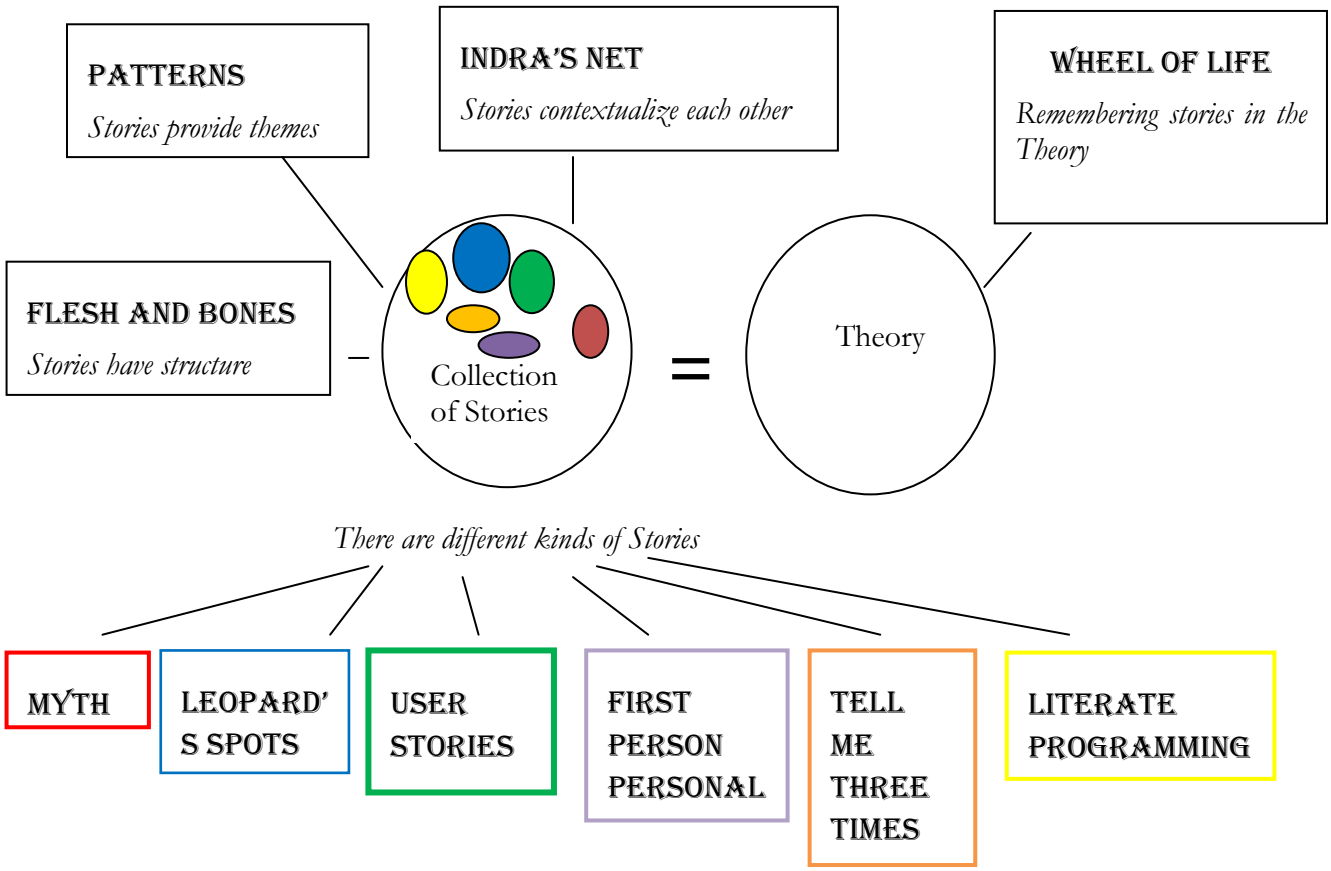


Figure 8. Overview

17. References

Alexander, Christopher, Sarah Ishikawa, Murray Silverstein, with M. Jacobson, I. Fiksdahl-King, S. Angel. (1977) *A Pattern Language*, New York: OUP.

Cahill, Tim. (2003) *Jaguars Ripped My Flesh*, Black Swan.

Kelly, Tom. (2001) *The Art of Innovation: Lessons from IDEO, America's Leading Design Firm*, Broadway Business.

Kuth, Donald. (1998)*The Art of Computer Programming*, Addison Wesley.

Kostova, Elizabeth. (2009) *The Historian*, Black Body Books.

Naur, Peter. (1985) "Programming as Theory Building," *Microprocessing and Microprogramming*, 15.

Shank, Robert. (1991) *Tell Me A Story: A New Look at Real and Artificial Memory*, Chicago: Athenaeum Press.