INVEST in Good Stories: The Series

Bill Wake

http://xp123.com

Table of Contents

Introduction

"A user story is a chunk of functionality (some people use the word *feature*) that is of value to the customer."
— Kent Beck & Martin Fowler, *Planning Extreme Programming*

Extreme Programming introduced *user stories* as a way to help customers and developers agree on how to grow a system.

Stories provide a lightweight, scenario-oriented approach in which the whole team can participate.

INVEST

After running into too many stories that were too large or too technology-oriented, I devised the INVEST acronym to remind people of the key attributes of good stories.

This ebook is a selection of lightly-edited posts from the http://xp123.com blog. It consists of (part of) the article that introduced INVEST, and one chapter for each letter in the acronym.

I hope this book gives you insight into ways to improve your stories.

Feel free to reach out to me with questions or comments; I'd be delighted to hear from you.

Bill Wake
bill@xp123.com

About the Author

Bill Wake

Bill Wake is a programmer, trainer, and coach for Agile teams. He has been involved with Agile methods (starting with XP) since 1999.

Bill is best known for the INVEST acronym, the Arrange-Act-Assert test pattern, and his writing on TDD, refactoring, and user stories. His occasional articles appear at http://industriallogic.com and http://xp123.com.

Bill lives in Williamsburg, Virginia, where he enjoys lectures and concerts in the historic area. He loves music, and plays mountain dulcimer and other instruments with varying levels of skill.

INVEST in Good Stories

The **INVEST** acronym can remind teams of the good characteristics of user stories.

In XP, we think of requirements as coming in the form of user stories. It would be easy to mistake the story card for the "whole story," but Ron Jeffries points out that stories in XP have three components: Cards (their physical medium), Conversation (the discussion surrounding them), and Confirmation (tests that verify them).

A *pidgin language* is a simplified language, usually used for trade, that allows people who can't communicate in their native language to nonetheless work together. User stories act like this. We don't expect customers or users to view the system the same way that programmers do; stories act as a pidgin language where both sides can agree enough to work together effectively.

But what are characteristics of a good story? The acronym "INVEST" can remind you that good stories are:

- **I** - Independent
- **N** - Negotiable
- **V** - Valuable
- **E** - Estimable
- **S** - Small
- **T** - Testable

Independent

Stories are easiest to work with if they are *independent*. That is, we'd like them to not overlap in concept, and we'd like to be able to schedule and implement them in any order.

We can't always achieve this; once in a while we may say things like "3 points for the first report, then 1 point for each of the others."

Negotiable... and Negotiated

A good story is *negotiable*. It is not an explicit contract for features; rather,

details will be co-created by the customer and programmer during development. A good story captures the essence, not the details. Over time, the card may acquire notes, test ideas, and so on, but we don't need these to prioritize or schedule stories.

Valuable

A story needs to be *valuable*. We don't care about value to just anybody; it needs to be valuable to the customer. Developers may have (legitimate) concerns, but these framed in a way that makes the customer perceive them as important.

This is especially an issue when splitting stories. Think of a whole story as a multi-layer cake, e.g., a network layer, a persistence layer, a logic layer, and a presentation layer. When we split a story, we're serving up only part of that cake. We want to give the customer the essence of the whole cake, and the best way is to slice vertically through the layers. Developers often have an inclination to work on only one layer at a time (and get it "right"); but a full database layer (for example) has little value to the customer if there's no presentation layer.

Making each slice valuable to the customer supports XP's pay-as-you-go attitude toward infrastructure.

Estimable

A good story can be *estimated*. We don't need an exact estimate, but just enough to help the customer rank and schedule the story's implementation. Being estimable is partly a function of being negotiated, as it's hard to estimate a story we don't understand. It is also a function of size: bigger stories are harder to estimate. Finally, it's a function of the team: what's easy to estimate will vary depending on the team's experience. (Sometimes a team may have to split a story into a (time-boxed) "spike" that will give the team enough information to make a decent estimate, and the rest of the story that will actually implement the desired feature.)

Small

Good stories tend to be *small*. Stories typically represent at most a few person-weeks worth of work. (Some teams restrict them to a few person-days of work.) Above this size, and it gets to be too hard to know what's in the story's scope. Saying, "it would take me more than a month" often implicitly adds, "as I don't understand what-all it would entail." Smaller stories tend to get more accurate estimates.

Story descriptions can be small too (and putting them on an index card helps make that happen). Alistair Cockburn described the cards as tokens promising a future conversation. Remember, the details can be elaborated through conversations with the customer.

Testable

A good story is *testable*. Writing a story card carries an implicit promise: "I understand what I want well enough that I *could* write a test for it." Several teams have reported that by requiring customer tests before implementing a story, the team is more productive. "Testability" has always been a characteristic of good requirements; actually writing the tests early helps us know whether this goal is met.

If a customer doesn't know how to test something, this may indicate that the story isn't clear enough, or that it doesn't reflect something valuable to them, or that the customer just needs help in testing.

A team can treat non-functional requirements (such as performance and usability) as things that need to be tested. Figure out how to operationalize these tests will help the team learn the true needs.

For all these attributes, the feedback cycle of proposing, estimating, and implementing stories will help teach the team what it needs to know.

Conclusion

As you discuss stories, write cards, and split stories, the INVEST acronym can help remind you of characteristics of good stories.

Postscript - Added 2-16-2011

I've been asked a few times where the INVEST acronym came from. I consciously developed it: I sat down and wrote every attribute I could think of applying to good stories: independent, small, right-sized, communicative, important, isolated, etc. This gave me a page full of words. Unfortunately, I haven't kept that list.

Then I grouped the words into clusters:

- Isolated, independent, separate, distinct, non-overlapping, ...
- Important, valuable, useful, ...
- Discrete, triggered, explicit, ...
- ...

The categories were a little fuzzy; I had about ten.

I identified "centers"; words that captured the essence of their category. Some clusters had two or three plausible centers. That was ok, as I just wanted their first letter for the acronym.

Then it was scramble time: take the three or four clusters that had to be there, plus some of the less important ones, and scramble the initials to find a word that fit. I wanted a word that had 4-6 letters, with no repeats.

I tried a lot of combinations. I remember at one point that if I had a G I could make VESTIGE or VESTING. Having VESTIN_, I realized I could turn it around to get INVEST, which sounded much better than VESTIGE:) So INVEST won, and it's been popular enough that I'm sure I made the right choice.

Independent

The INVEST model is a reminder of the important characteristics of user stories, and it starts with **I** for **Independent**.

Independent stories each describe different aspects of a system's capabilities. They are easier to work with because each one can be (mostly) understood, tracked, implemented, tested, etc. on its own.

Agile software approaches are flexible, better able to pursue whatever *is* most valuable today, not constrained to follow a 6-month old guess about what *would be* most valuable today. Independent stories help make that true: rather than a "take it or leave it" lump, they let us focus on particular aspects of a system.

We would like a system's description to be consistent and complete. Independent stories help with that: by avoiding overlap, they reduce places where descriptions contradict each other, and they make it easier to consider whether we've described everything we need.

Three common types of dependency:
overlap, order, and containment

What makes stories dependent rather than independent? There are three common types of dependency: *overlap* (undesirable), *order* (mostly can be worked around), and *containment* (sometimes helpful).

Overlap Dependency

Overlap is the most painful form of dependency. Imagine a set of underlying capabilities:
A, B, C, D, E, F

with stories that cover various subsets:
A, B
A, B, F
B, C, D
B, C, F
B, E
E, F

Quick: what's the smallest set of stories that ensure that capabilities A, B, C, D, E, F are present? What about A, B, C, E? Can we get those and nothing else?

When stories overlap, it's hard to ensure that everything is covered at least once, and we risk confusion when things are covered more than once.

Overlapping stories create confusion.

For example, consider an email system with the stories "User sends and receives messages" and "User sends and replies to messages." (Just seeing the word "and" in a story title is suspicious, but you really have to consider multiple stories to know if there's overlap.) Both stories mention sending a message. We can partition the stories differently to reduce overlap:

User sends message
User receives message
User replies to message

(Note that we're not concerned about "technical" overlap at this level: sending and replying to messages would presumably share a lot of technical tasks. How we design the system or schedule the work is not our primary concern when we're trying to understand the system's behavior.)

Order Dependency

A second common dependency is order dependency: "this story must be implemented before that one."

Order dependencies complicate a plan,

but we can usually eliminate them.

While there's no approach that guarantees it, order dependency tends to be something that is mostly harmless and can be worked around. There are several reasons for that:

1. Some order dependencies flow from the nature of the problem. For example, a story "User re-sends a message" naturally follows "User sends message." Even if there is an order dependency we can't eliminate, it doesn't matter since the business will tend to schedule these stories in a way that reflects it.
2. Even when a dependency exists, there's only a 50/50 chance we'll want to schedule things in the "wrong" order.
3. We can find clever ways to remove most of the order dependencies.

For example, a user might need an account before they can send email. That might make us think we need to implement the account management stories first (stories like "Admin creates account"). Instead, we could build in ("hard-code") the initial accounts. (You might look at this as "hard-coding" or you might think of it as "the skinniest possible version of account management"; either way, it's a lot less work.)

Why take that approach? Because we want to explore certain areas first. We consider both value and risk. On the value side, we may focus on the parts paying customers will value most (to attract them with an early delivery). On the risk side, we may find it important to address risks, thinking of them as negative value. In our example, we may be concerned about poor usability as a risk. A few hard-coded accounts would be enough to let us explore the usability concerns.

Containment Dependency

Containment dependency comes in when we organize stories hierarchically: "this story contains these others." Teams use different terms for this idea:

you might hear talk of "themes, epics, and stories," "features and stories," "stories and sub-stories," etc. A hierarchy is an organizational tool; it can be used formally or informally.

A good organization for *describing* a system
is rarely the best organization for *scheduling* its implementation.

The biggest caveat about a hierarchical decomposition is that while it's a helpful strategy for organizing and understanding a large set of stories, it doesn't make a good scheduling strategy. It can encourage you to do a

"depth-first" schedule: address this area, and when it's done, go the next area. But really, it's unlikely that the most valuable stories will all be in a single area. Rather, we benefit from first creating a minimal version of the whole system, then a fancier version (with the next most important feature), and so on.

Bottom Line

Independent stories help both the business and technical sides of a project. From a business perspective, the project gets a simple model focused on the business goals, not over-constrained by technical dependencies. From a technical perspective, independent stories encourage a minimal implementation, and support design approaches that minimize and mange implementation dependencies.

Related Material

- *Composing User Stories* - eLearning from Industrial Logic
- *User Stories Applied*, by Mike Cohn

Negotiable

In the INVEST model for user stories, **N** is for **Negotiable** (and **Negotiated**). Negotiable hints at several important things about stories:

- The importance of collaboration
- Evolutionary design
- Response to change

Collaboration

Why do firms exist? Why isn't everything done by individuals interacting in a marketplace? Nobel-prize winner Ronald Coase gave this answer: firms reduce the friction of working together.

Working with individuals has costs: you have to find someone to work with, negotiate a contract, monitor performance carefully—and all these have a higher overhead compared to working with someone in the same firm. In effect, a company creates a zone where people can act in a higher-trust way (which often yields better results at a lower cost).

The same dynamic, of course, plays out in software teams; teams that can act from trust and goodwill expect better results. Negotiable features take advantage of that trust: people can work together, share ideas, and jointly own the result.

Evolutionary Design

High-level stories, written from the perspective of the actors that use the system, define capabilities of the system without over-constraining the implementation approach. This reflects a classic goal for requirements:

specify what, not how. (Admittedly, the motto is better-loved than the practice.)

Consider an example: an online bookstore. (This is a company that sells stories and information printed onto pieces of paper, in a package known as a "book.") This system may have a requirement "Fulfillment sends book and receipt." At this level, we've specified our need but haven't committed

to a particular approach. Several implementations are possible:

- A fulfillment clerk gets a note telling which items to send, picks them off the shelf, writes a receipt by hand, packages everything, and takes the accumulated packages to the delivery store every day.
- The system generates a list of items to package, sorted by (warehouse) aisle and customer. A clerk takes this "pick list" and pushes a cart through the warehouse, picking up the items called for. A different clerk prints labels and receipts, packages the items, and leaves them where a shipper will pick them up.
- Items are pre-packaged and stored on smart shelves (related to the routing systems used for baggage at large airports). The shelves send the item to a labeler machine, which sends them to a sorter that crates them by zip code, for the shipper to pick up.

Each of these approaches fulfills the requirement. (They vary in their non-functional characteristics, cost, etc.)

By keeping the story at a higher level, we leave room to negotiate: to work out a solution that takes everything into account as best we can. We can create a path that lets us evolve our solution, from basic to advanced form.

Feedback

Waterfall development is sometimes described as "throw it over the wall": create a "perfect" description of a solution, feed it to one team for design, another for implementation, another for testing, and so on, with no real feedback between teams. But this approach assumes that you can not only correctly identify problems and solutions, but also communicate these in exactly the right way to trigger the right behavior in others.

Some projects can work with this approach, or at least come close enough. But others are addressing "wicked problems" where any solution affects the perceived requirements in unpredictable ways. Our only hope in these situations is to intervene in some way, get feedback, and go from there.

Some teams can (or try to) create a big static backlog at the start of a project, then measure burndown until those items are all done. But this

doesn't work well when feedback is needed.

Negotiable stories help even in ambiguous situations; we can work with high-level descriptions early, and build details as we go. By starting with stories at a high level, expanding details as necessary, and leaving room to adjust as we learn more, we can more easily evolve to a solution that balances all our needs.

Related Material

- Composing User Stories - eLearning from Industrial Logic

Valuable

Of all the attributes of the INVEST model, "**V**aluable" is the easiest one to, well, value. Who is against value?

We'll look at these key aspects:

- What is value?
- The importance of external impact
- Value for whom?

What is Value?

Value depends on what we're trying to achieve. One old formula is IRACIS. (Gane & Sarson mentioned it in 1977 in *Structured Systems Analysis*, and didn't claim it was original with them.) IRACIS means:

- Increase Revenue
- Avoid Costs
- Improve Service.

Increase Revenue: Add new features (or improve old ones) because somebody will pay more when they're present.

Avoid Costs: Much software is written to help someone avoid spending money. For example, suppose you're writing software to support a call center: every second you save on a typical transaction means fewer total agents are needed, saving the company money.

Improve Service: Some work is intended to improve existing capabilities. Consider Skype, the voice network: improving call quality is not a new feature, but it has value. (For example, more customers might stay with the

service when call quality is higher.)

IRACIS covers several types of value, but there are others:

Meet Regulations: The government may demand that we support certain capabilities (whether we want to or not). For example, credit card companies are required to support a "Do Not Call" list for customers who don't want new offers. If the company didn't provide the capability by a certain date, the government would shut down the company.

Build Reputation: Some things are done to increase our visibility in the marketplace. An example might be producing a free demo version of packaged software, to improve its marketing. In effect, these are an indirect way to increase revenue.

Create Options: Some things give us more flexibility in the future. For example, we may invest in database independence today, to give us the ability to quickly change databases in the future. The future is uncertain; options are insurance.

Generate Information: Sometimes we need better information to help us make a good decision. For example, we might do an A-B test to tell us which color button sells more. XP-style spikes may fit this category as well.

Build Team: Sometimes a feature is chosen because it will help the team successfully bond, or learn important to the future.

Several of these values may apply at the same time. (There's nothing that makes this an exhaustive list, either.) Because multiple types of values are involved, making decisions is not easy: we have to trade across multiple dimensions.

Valuing External Impact

Software is designed to accomplish something in the real world.

We'll lean on a classic analysis idea: describe the system's behavior as if the system is implemented with a perfect technology. Focus on the effects of the system in the world.

This helps clarify what are "real" stories: they start from outside the system and go in, or start inside and go outside.

This also helps us avoid two problems:

- "stories" that are about the solution we're using (the technology)
- "stories" that are about the creators of the system, or what they want

If we frame stories so their impact is clear, product owners and users can

understand what the stories bring, and make good choices about them.

Value for Whom?

Who gets the benefit of the software we create? (One person can fill several of these roles, and this is not an exhaustive list.)

Users: The word "User" isn't the best, but we really are talking about the people who *use* the software. Sometimes the user may be indirect: with a call center, the agent is the direct user, and the customer talking to them is indirect.

Purchasers: Purchasers are responsible for choosing and paying for the software. (Sometimes even these are separate roles.) Purchasers' needs often do not fully align with those of users. For example, the agents using call center software may not want to be monitored, but the purchaser of the system may require that capability.

Development Organizations: In some cases, the development organization has needs that are reflected in things like compliance to standards, use of default languages and architectures, and so on.

Sponsors: Sponsors are the people paying for the software being developed. They want some return on their investment.

There can be other kinds of people who get value from software we develop. Part of the job of a development team is balancing the needs of various stakeholders.

Summary

We looked at what values is: IRACIS (Increase Revenue, Avoid Costs, Improve Service), as well as other things including Meeting Regulations, Generating Information, and Creating Options.

We briefly explored the idea that good stories usually talk about what happens on the edge of the system: the effects of the software in the world.

Finally, we considered how various stakeholders benefit: users, purchasers,

development organizations, and sponsors.

Value is important. It's surprisingly easy to get disconnected from it, so returning to the understanding of "What is value for this project?" is critical.

Estimable

Estimable stories can be estimated: some judgment made about their size, cost, or time to deliver. (We might wish for the term *estimatable*, but it's not in my dictionary, and I'm not fond enough of estimating to coin it.)

To be estimable, stories have to be understood well enough, and be stable enough, that we can put useful bounds on our guesses.

A note of caution: Estimability is the most-abused aspect of INVEST (that is, the most energy spent for the least value). If I could re-pick, we'd have "E = External"; see the discussion under "V for Valuable".

Why We Like Estimates

Why do we want estimates? Usually it's for planning, to give us an idea of the cost and/or time to do some work. Estimates help us make decisions about cost versus value.

When my car gets worked on, I want to know if it's going to cost me $15 or $10K, because I'll act differently depending on the cost. I might use these guidelines:

- <$50: just do it
- $50-$300: get the work done but whine to my friends later
- $300-$5000: get another opinion; explore options; defer if possible
- $5000+: go car shopping

Life often demands some level of estimation, but don't ignore delivery or value to focus too much on cost alone.

We'll go through facts and factors affecting estimates; at the end I'll argue

for as light an estimation approach as possible.

Face Reality: An Estimate is a Guess

If a story were already completed, the cost, time taken, etc. would be (could be?) known quantities.

We'd really like to know those values in advance, to help us in planning, staffing, etc.

Since we can't *know*, we mix analysis and intuition to create a guess, which could be a single number, a range, or a probability distribution. (It doesn't matter whether it's points or days, Fibonacci or t-shirt sizes, etc.)

When we decide how accurate our estimates must be, we're making an economic tradeoff since it costs more to create estimates with tighter error bounds.

How Are Estimates Made?

There are several approaches, often used in combination:

- **Expert Opinion AKA Gut Feel AKA SWAG**: Ask someone to make a judgment, taking into account everything they know and believe. Every estimation method boils down to this at some point.
- **Analogy**: Estimate based on something with similar characteristics. ("Last time, a new report took 2 days; this one has similar complexity, so let's say 2 days.")
- **Decomposition AKA Divide and Conquer AKA Disaggregation**: Break the item into smaller parts, and estimate the cost of each part — plus the oft-forgotten cost of re-combining the parts.
- **Formula**: Apply a formula to some attributes of the problem, solution, or situation. (Examples: Function Points, COCOMO.)
  Challenges:
  - formulas' parameters require tuning based on historical data (which may not exist)
  - formulas require judgment about which formulas apply

- formulas tend to presume the problem or solution is well-enough understood to assess the concrete parts

- **Work Sample**: Implement a subset of the system, and base estimates on that experience. Iterative and incremental approaches provide this ongoing opportunity.
- **Buffer AKA Fudge Factor**: Multiply (and/or add to) an estimate to account for unknowns, excessive optimism, forgotten work, overheads, or intangible factors. For example: "Add 20%", "Multiply by 3", or "Add 2 extra months at the end".

Why Is It Hard to Estimate?

Stories are difficult to estimate because of the unknowns. After all, the whole process is an attempt to derive a "known" (cost, time, ...) from something unknowable ("exactly what will the future bring?").

Software development has so *many* unknowns:

- **The Domain**: When we don't know the domain, it's easier to have misunderstandings with our customer, and it can be harder to have deep insights into better solutions.
- **Level of Innovation**: We may be operating in a domain where we need to do things we have never done before; perhaps nobody has.
- **The Details of a Story**: We often want to estimate a story before it is fully understood; we may have to predict the effects of complicated business rules and constraints that aren't yet articulated or even anticipated.
- **The Relationship to Other Stories**: Some stories can be easier or harder depending on the other stories that will be implemented.
- **The Team**: Even if we have the same people as the last project, and the team stays stable throughout the project, people change over time. It's even harder with a new team.
- **Technology**: We may know some of the technology we'll use in a large project, but it's rare to know it all up-front. Thus our estimates have to account for learning time.
- **The Approach to the Solution**: We may not yet know how we intend to

- solve the problem.
- **The Relationship to Existing Code**: We may not know whether we'll be working in a habitable section of the existing solution.
- **The Rate of Change**: We may need to estimate not just "What is the story now?" but also "What will it be by the end?"
- **Dysfunctional Games**: In some environments, estimates are valued mostly as a tool for political power-plays; objective estimates may have little use. (There's plenty to say about estimates vs. commitments, schedule chicken, and many other abuses but I'll save that for another time.)

- **Overhead**: External factors affect estimates. If we multi-task or get pulled off to side projects, things will take longer.

Sitting in a planning meeting for a day or a week and ginning up a feeling of commitment won't overcome these challenges.

Flaws In Estimating

We tend to speak as if estimates are concrete and passive: "Given this story, what is *the* estimate?"

But it's not that simple:

- "N for Negotiable" suggests that flexibility in stories is beneficial: flexible stories help us find bargains with the most value for their cost. But the more variation you allow, the harder it is to estimate.
- "I for Independent" suggests that we create stories that can be independently estimated and implemented. While this is mostly true, it is a simplification of reality: sometimes the cost of a story depends on the order of implementation or on what else is implemented. It may be hard to capture that in estimates.
- Factors that make it hard to estimate are **not stable over time**. So even if you're able to take all those factors into account, you also have to account for their instability.

Is estimating hopeless? If you think estimation is a simple process that will yield an exact (and correct!) number, then you are on a futile quest. If you

just need enough information from estimates to guide decisions, you can usually get that.

Some projects need detailed estimates, and are willing to spend what it takes to get them. In general, though, Tom DeMarco has it right: "Strict control is something that matters a lot on relatively useless projects and much less on useful projects."

Where does that leave things? The best way is to use as light an estimation process as you can tolerate.

We'll explore three approaches: counting stories, historical estimates, and

rough order of magnitude estimates.

Simple Estimates: Count the Stories

More than ten years ago, Don Wells proposed a very simple approach: "Just count the stories."

Here's a thought experiment:

- Take a bunch of numbers representing the true sizes of stories
- Take a random sample
- The average of the sample is an approximation of the average of the original set, so use that average as the estimate of the size of every story ("Call it a 1")
- The estimate for the total is the number of stories times the sample average

What could make this not work?

- If stories are highly inter-dependent, and the order they're done in makes a dramatic difference to their size, the first step is void since there's no such thing as the "true" size.
- If you cherry-pick easy or hard stories rather than a random set, you will bias the estimate.
- If your ability to make progress shifts over time, the estimates will diverge. (Agile teams try to reduce that risk with refactoring, testing, and

- simple design.)

I've seen several teams use a simple approach: they figure out a line between "small enough to understand and implement" and "too big", then require that stories accepted for implementation be in the former range.

Historical Estimates (ala Kanban)

For many teams, the size of stories is not the driving factor in how long a story takes to deliver. Rather, work-in-progress (WIP) is the challenge: a new story has to wait in line behind a lot of existing work.

A good measure is *total lead time* (also known as cycle time or various other names): how long from order to delivery. Kanban approaches often use this measure, but other methods can too.

If we track history, we can measure the cycle times and look for patterns. If we see that the average story takes 10 days to deliver and that 95% of the stories take 22 or fewer days to deliver, we get a fairly good picture of the time to deliver the next story.

This moves the estimation question from "How big is this?" to "How soon can I get it?"

When WIP is high, it is the dominant factor in delivery performance; as WIP approaches 0, the size of the individual item becomes significant.

Rough Order of Magnitude

A rough order of magnitude estimate just tries to guess the time unit: hours, days, weeks, months, years.

You might use such estimates like this:

- Explore risk, value, and options
- Make rough order of magnitude estimates
- Focus first on what it takes to create a minimal but useful version of the

- most important stories
- From there, decide how and how far to carry forward by negotiating to balance competing interests
- Be open to learning along the way

Conclusion

Stories are estimable when we can make a good-enough prediction of time, cost, or other attributes we care about.

We looked at approaches to estimation and key factors that influence

estimates.

Estimation does not have to be a heavy-weight and painful process. Try the

lighter ways to work with estimates: counting stories, historical estimates, and/or rough order of magnitude estimates.

Whatever approach you take, spend as little as you can to get good-enough estimates.

Related Material

- #noestimates - Exploration of alternatives to estimation.

Postscript: My thinking on this has definitely evolved over the years, but I've always felt that Small and Testable stories are the most Estimable:)

Small → Scalable

S is for *Small* in the INVEST acronym. I now use another **S** that I think captures the idea even better: *Scalable*.

Scalable means "able to be changed in size or scale," and that's handy in a story: We want stories sized to what we're trying to do.

It's helpful to think about stories at three levels: high, medium, and low.

High-Level Stories: Themes and Activities

The highest level view helps us learn the shape and extent of the system we need.

These items are so big that most people don't even call them stories. You may know them as "themes" (Kent Beck), "activities" (Jeff Patton), "epics" (SaFE), or "kite level" (Alistair Cockburn). (We'll ignore the subtle differences.)

For example, suppose we're creating a new car rental system. We might talk about:

- Reserving cars
- Renting cars
- Returning cars
- Analytics

As we look at this list, we may realize we haven't thought about "Loss and Damage": what happens if a car is stolen or damaged? Perhaps the concept should be "Fleet Management," and include prepping cars.

I'm not a car rental expert, so surely there are other major areas of the system. But it'll be closer to a dozen than a hundred at this level.

These aren't stories somebody can just go implement. We use these to answer, "Have we forgotten anything big?"

Middle Level: The Headline

The middle level is where stories' headlines are. This is the level of story

maps and release plans.

For a headline to make sense, we have to talk about a particular kind of story: External and Valuable. Such stories start from outside a system, and describe a user or system action that accomplishes something a stakeholder cares about.

Headlines don't need to follow a template, but I default to the one Industrial Logic suggests: **Role-Action-Context**:

- **Role** - the user or system triggering the story
- **Action** - what happens
- **Context** - when, where, and/or how

Example: Customer purchases item
Example: Customer purchases item with debit card

Some teams create headlines and stories that are super-detailed, but only cover part of a system action. Don't do that; make your stories describe a full (though possibly small) and valuable interaction. Keep the details a level down.

Ironically, people sometimes create and excuse these partial "technical stories" (ugh) as their way to keep stories Small. They may be small, but they're not *user* stories. A Product Owner can't work with them effectively.

"Technical stories" (ugh) may be small,
but they're not *user* stories.

Stories that users value and that can scale up or down provide a lot of power in an Agile delivery model. If we can assemble the most critical stories and deliver them in a minimalistic way, we can quickly get a working version of the system. (See Alistair Cockburn's idea of a Walking Skeleton.)

Once the system is working, even if in a clunky way, we can start using it, while we simultaneously enhance it over time as usage scales. This may get us early payment (yay!) but will certainly give us early feedback.

We need feedback because... like it or not, we're ignorant. We have *theories*

about what's valuable, but we've never put them to the test. We build a house of cards out of our assumptions. (Read about Minimum Viable Products too.)

I'm sure there are domains where Product Owners exactly understand the value, but I don't run into them. Where I go, people tend to have limited data about usage, sales, etc. Even when they have data about what users and customers do, they tend to guess about the future and about what would appeal to non-users and non-customers.

Low Level: The Details

At the bottom, we care about how we do things.

For example: ***Customer purchases item***

This story can be delivered in many ways.

For example, I used to buy soda water cartridges from a website where you just emailed them a purchase request, including your phone number in the email. Then Joe called you to get your payment info and shipping details.

Other ways this story might work:

- In a physical store, a customer might ask the clerk for an item. The clerk pulls it from the shelf behind them, hand-writes a receipt with a carbon copy, gives you the item and the original receipt, and sticks the carbon copy on a spike.

- In a supermarket, a customer might walk through the store, put the items they want in a cart, then check out with clerks at registers in the front of the store. Some grocery stores have a self-checkout, where you scan your own items.
- When you buy something at Amazon.com, you have your credit card on file, and Amazon attempts to cross-sell and up-sell while you check out. (Your purchase also feeds a lot of reports and analytics, perhaps described in other stories.)

A single headline allows for many variations.
***That* is scalability in stories.**

We're used to different stores doing things differently, but we sometimes forget that our own implementations can evolve as well. A single headline allows for many variations. *That* is scalability in stories.

You may recognize this approach as being *iterative*. It's iterative not just in the software implementation, but also in the details of the story.

Scalability and Splitting

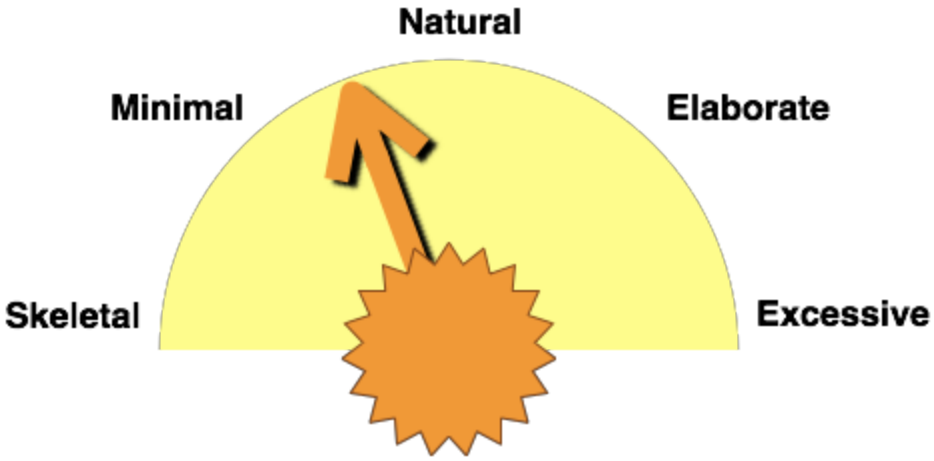
One headline covers many potential stories.

You may already be used to the idea of splitting stories. Given a story, the team comes back and says, "This story is too big—we need a smaller bite."

But splitting is painful: you build up a full description of a story, with all the capabilities and ideas you imagine for it. Then the team takes a scalpel, knife, or sledgehammer to it, tearing it apart.

The process can drive a product owner to tears.

This is where Scalability comes in. Think of the intensity of a story as a dial: skeletal, minimal, natural, elaborate, excessive. (These are fuzzy, not definitive levels.)



Move from *splitting* stories to ***intensifying*** stories.

Move from *splitting* stories to *intensifying* stories.

As a product owner, work from the scaling-up perspective. Start small — smaller than you might expect — and grow your stories over time as you deliver.

How Big?

Big stories are tricky.

- It's hard to know we've got exactly the right features (as we're adding things without a lot of experience).
- Big stories often mix both more and less valuable parts, so it's hard to separate out their contribution.
- Big stories strain the delivery team to digest and deliver a big chunk of work.

Big stories are tricky.

You may have learned to make stories that are MMFs: Minimal Marketable Features. But even that's too big in a continuous-delivery world. Instead, think of Minimum Useful Features - small things that still makes somebody's life a little better (and can grow into big things).

For example, consider an improvement in a search engine: the user interface stays the same, but search results are a little better. This is worth releasing even if it doesn't merit bullet points in an ad.

I've often encouraged people to make stories smaller - hours to days of effort, not weeks to months.

Can stories get too small? Definitely. I like Jeff Patton's analogy of overly-atomized stories as a bag of leaves, stripped of structure and burying you in details.

How can you deal with too-small stories? Merge them. (This is one reason I like the "major" headline view - you can have stories that cover different

details, then generalize and merge them to talk about the headline and the current state of implementation.)

Unfortunately, many "agile" planning tools take a hierarchical, atomized view, and don't help much when thinking in terms of scalable stories. It takes work to maintain the big picture.

Conclusion

John Gall said, "A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system." It's a good reminder for stories.

Remember the three levels from earlier:

1. **Theme - Top-level**: Are we covering the right things?
2. **Headline - Mid-level**: Is the user getting a capability they care about?
3. **Details - Lowlevel**: Do we need a new capability, or can we improve an old one?

Rather than splitting too-big stories, train yourself to start with smaller versions, then extend and intensify them. If you have to split, take a user's perspective, not a technology perspective.

You will find that scalable stories will:

- Help make progress visible
- Provide early utility
- Give you feedback that helps you steer to a more valuable result
- Help reduce market and technical risk

Testable

T is for Testable in the INVEST model.

A *testable* story is one for which, given any inputs, we can agree on the expected system behavior and/or outputs.

Testability doesn't require that all tests be defined up front before the first line of code is written. We just need to be appropriately confident that we *could* define and agree on them.

Testability doesn't require that tests be automated. There are advantages to automation, but tests can be useful without it.

Testability doesn't require that we treat tests as a full specification, where any missing piece brings down the whole edifice. Instead, we focus on interesting tests that clarify confusing or controversial decisions.

Testing, even using tests as a collaborative tool, is not a new thing. There are tools to help specify tests - xUnit, Cucumber, fit, or even plain English. But tools aren't enough; there are still plenty of mistakes you can make.

We'll look at challenges you face regardless of tool or technique:

- Magic
- Intentional Fuzziness
- Computational Infeasibility
- Non-Determinism
- Subjectivity
- Research Projects

Magic

Some tests rely on magic or intuition. Unfortunately, (withholding judgment about humans) computers don't have either one.

A test sets up a context, has inputs and/or steps, and produces output and/or side effects.

Just because we can write inputs and outputs doesn't mean we have a

good test. The outputs must also be a function of the inputs; they can't depend on other unstated inputs.

> *For example, we'd like to have a program to predict tomorrow's stock price, given its previous prices.*
>
> *We have lots of test cases - easily a century's worth of examples for hundreds of stocks.*
>
> *But, at its heart, this is a desire for magic, unless we believe something very unlikely: that the stock's price is determined only by the history of its prices. Suppose 50% of the world's oil refineries go offline for 6 months. Do you really believe that news wouldn't affect oil, airline, and automobile stocks?*

Intentional Fuzziness

In some cases, algorithms exist but choices have to be made. Some fuzziness is accidental but some is intentional: Sales wants red, and Marketing wants green. Rather than make a choice that will annoy one side or the other, the test hides behind fuzziness like "the appropriate color".

Computational Infeasibility

Some things are computable, perhaps even easy to specify, but aren't

feasible above a certain size. For example, finding the exact shortest path in a large map is very expensive.

You may be able to loosen your criteria and settle for an approximation. In the map, getting within a factor of 2 of the shortest path is substantially cheaper.

Non-Determinism

Some non-determinism is because of randomness or pseudo-randomness.

Other non-determinism happens when there is a range or set of acceptable

answers.

> *For example, I want a puzzle-making program that, given a list of words, produces a Boggle™ board containing them: a grid of letters, where you form words by bouncing between letters one at a time in any direction.*
>
> *If it can be done at all, it can certainly be done multiple ways, since given one solution you can generate others by mirroring or rotation. (You might have multiple solutions that differ by more than that).*
>
> *I don't care which solution is picked; any valid solution is ok.*

Finally, floating-point arithmetic is only an approximation of real-number arithmetic. Because of variations in computation, you might have answers that vary by some epsilon.

The first challenge with non-determinism is recognizing when it happens.

When it does, you may have to specify when two answers are equivalent or equally acceptable.

Subjectivity

Some tests are well-defined but appeal to subjective standards.

> *For example, "7 of 10 people agree it's fun." Refine what you mean by "people" all you want - but "fun" is subjective.*

Usability and other "non-functional" characteristics often have the same sort of challenge: "9 of 10 users can complete the standard task in <5 minutes with 30 minutes of training."

Subjective tests come with no guarantee of their feasibility. Furthermore, they can be extremely expensive to test. It is not wrong to have them, but you must recognize there's no guarantee about how easily subjective standards can be met.

Research Project

It is possible to specify or define a test that is valid, but for which you don't actually know how to solve it. (Testable? Yes! Risky? Very!)

That is: you can write tests for something you don't know how to implement, and any estimate you make is suspect because you're estimating a research project.

> *For example, consider the challenge of identifying pedestrians in a scene. Given an image, we could poll 1000 people to identify pedestrians with a high level of confidence. But we're not clear on how people make that call - we suspect it can be done algorithmically, but proving that has been a multi-year research project. (Progress in self-driving cars suggests it is possible).*

There's nothing wrong with having a research project if that's what you need and you accept the risk. Just don't fool yourself into thinking it's a SMOP - Simple Matter of Programmming - when it's not.

Testable Trigger Words

Given all the testing challenges mentioned above, how do we make sure we're writing good tests?

I've found there are trigger words that can warn you about potential

problems in the tests or in the discussion around them:

- "**Just**" - such a little word, but it's often used as a power move, to minimize the importance of concerns without actually addressing them. ("Just check each of these against all the others.")
- "**Appropriate**," "**right**," "**suitable**" - of course you want the appropriate thing done - but that doesn't define what "appropriate" actually means.
- "**Best**," "**worst**" - saying "best" doesn't *define* best.
- "**Most**," "**least**," "**shortest**," "**longest**" - if you're clear about what you want the most or least of, that helps, but these words can also hide computationally expensive choices.
- "**All combinations**," "**all permutations**" - these words may signal things that are well-defined but computationally infeasible.
- "**Any of**," "**don't care**" - these phrases often occur with non-deterministic examples. They're legitimate, but it can be tricky to specify test cases with such examples. (Will you list all possible solutions?)
- "**Fun**," "**easy to use**," "**people**," "**like**", ... - these are common in non-functional attributes, or may hide a research project.

- "**I'll know it when I see it**" - there's no chance you'll get this right in one round.

Summary

Testable stories help ensure that teams agree on what's wanted. Tests support collaboration: they help clear up muddy thinking, and help people build a shared understanding.

We've looked at several challenges with tests:

- *Troubles to fix*: Magic, Intentional Fuzziness
- *Challenges to be aware of*: Computational Infeasibility, Non-Determinism
- *Risks to not accept accidentally*: Subjectivity, Research Projects

We closed with some trigger words that can indicate when you're facing testing challenges.

This is the close of the series taking a more detailed look at the INVEST model.

What Next?

Continue exploring at xp123.com!

Visit xp123.com