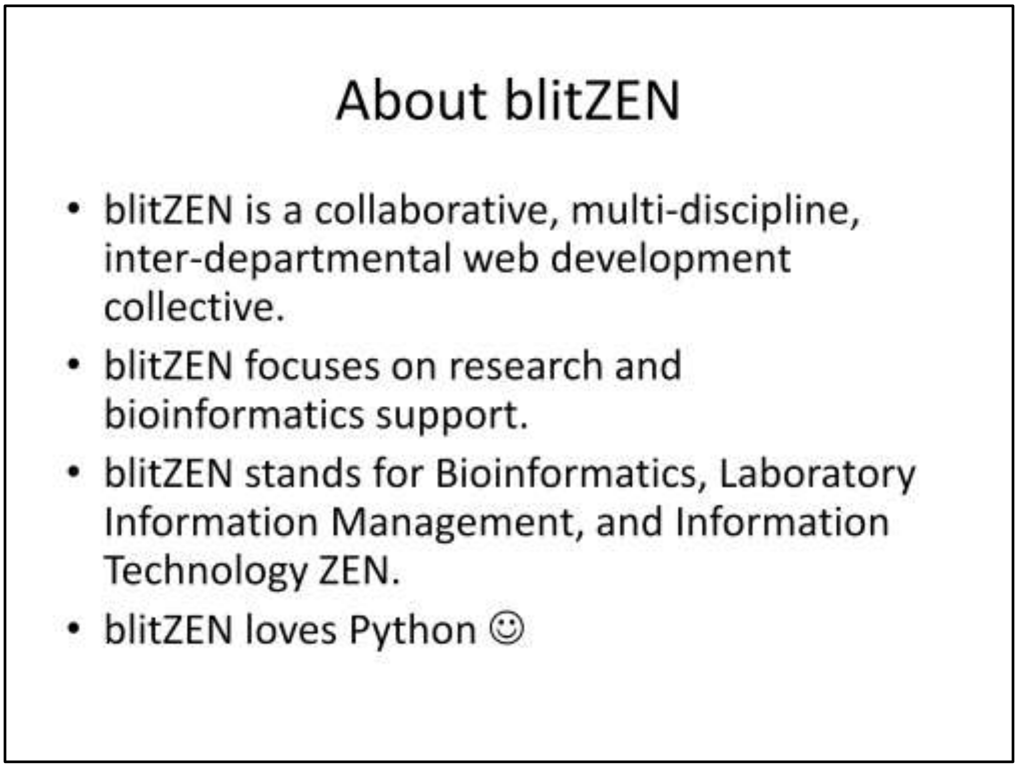
The blitZEN Method

An awesome approach to developing software.



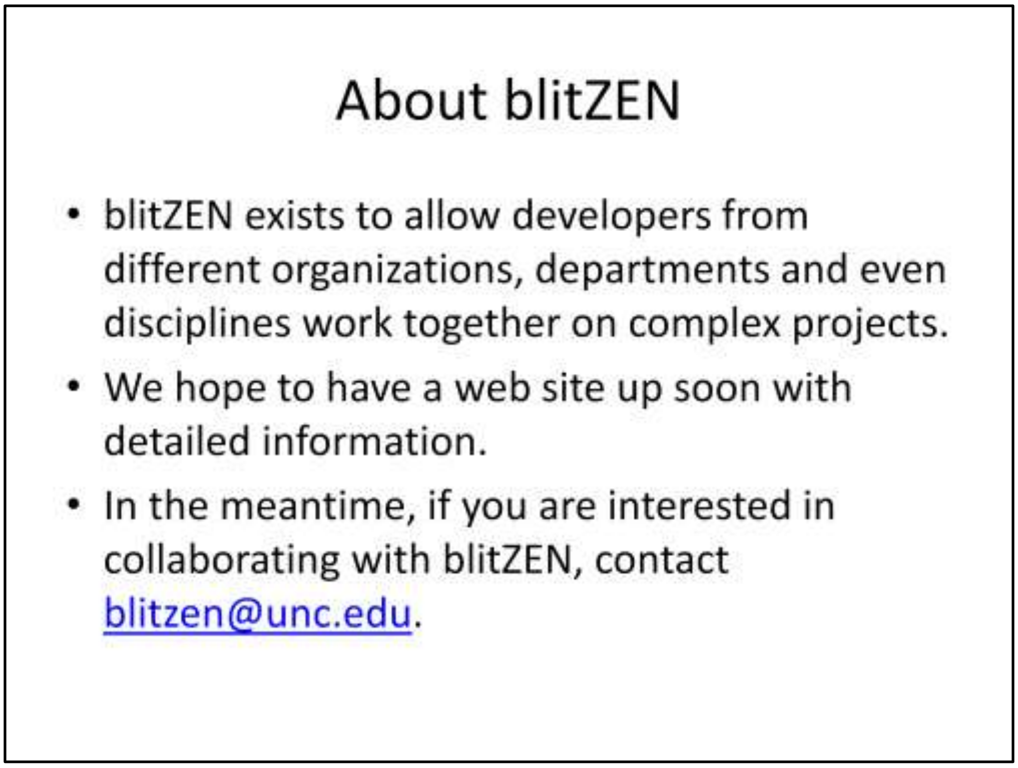
About blitZEN

- blitZEN is a collaborative, multi-discipline, inter-departmental web development collective.
- blitZEN focuses on research and bioinformatics support.
- blitZEN stands for Bioinformatics, Laboratory Information Management, and Information Technology ZEN.
- blitZEN loves Python ☺

Point 2: this is the realm where we operate currently, but this is really only because of our departmental affiliations. This methodology and our collaborative spirit has the potential to be successfully utilized in any development project.

Point 3: coined by Rob Lineberger in 2008.

Point 4: there's a lot about Python that makes it suited to the sorts of work we do, and I would argue, general web development tasks as well. The methodology is not strictly tied to any platform or programming language. We try to look at each problem with a fresh approach and remain objective as we search for the best tools to meet our requirements. That said, in spite of our objectivity, it's proven difficult to find a situation where Python doesn't exceed our needs. ☺



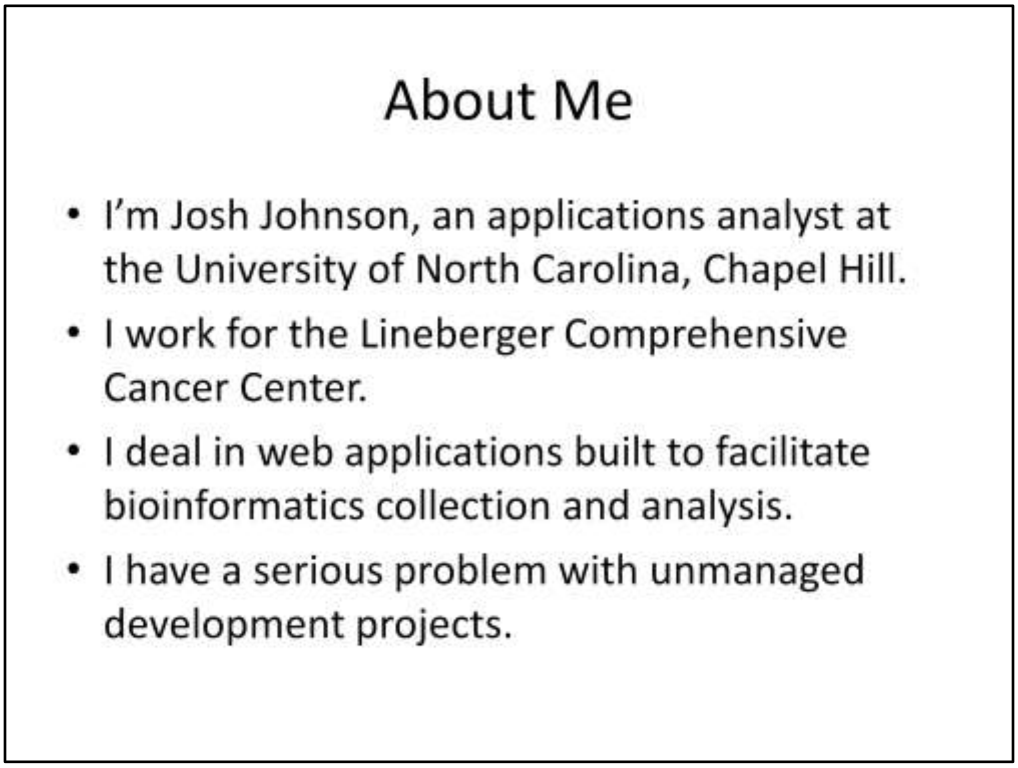
About blitZEN

- blitZEN exists to allow developers from different organizations, departments and even disciplines work together on complex projects.
- We hope to have a web site up soon with detailed information.
- In the meantime, if you are interested in collaborating with blitZEN, contact blitzen@unc.edu.

Point 1: Previously, I've worked for another center on campus, and was very frustrated with the roadblocks that stood in the way of really leveraging the resources we had at hand. Things got worse when I got into Python and Plone and started reaching out to the community. It became apparent that in our area, and especially on campus, we have a huge wealth of experience, talent, and ambition, and we rarely interfaced in any meaningful way. One of blitZEN's core goals is to try to address that problem.

Point 2: keep an eye on http://blitzen.unc.edu. A new site will be up soon.

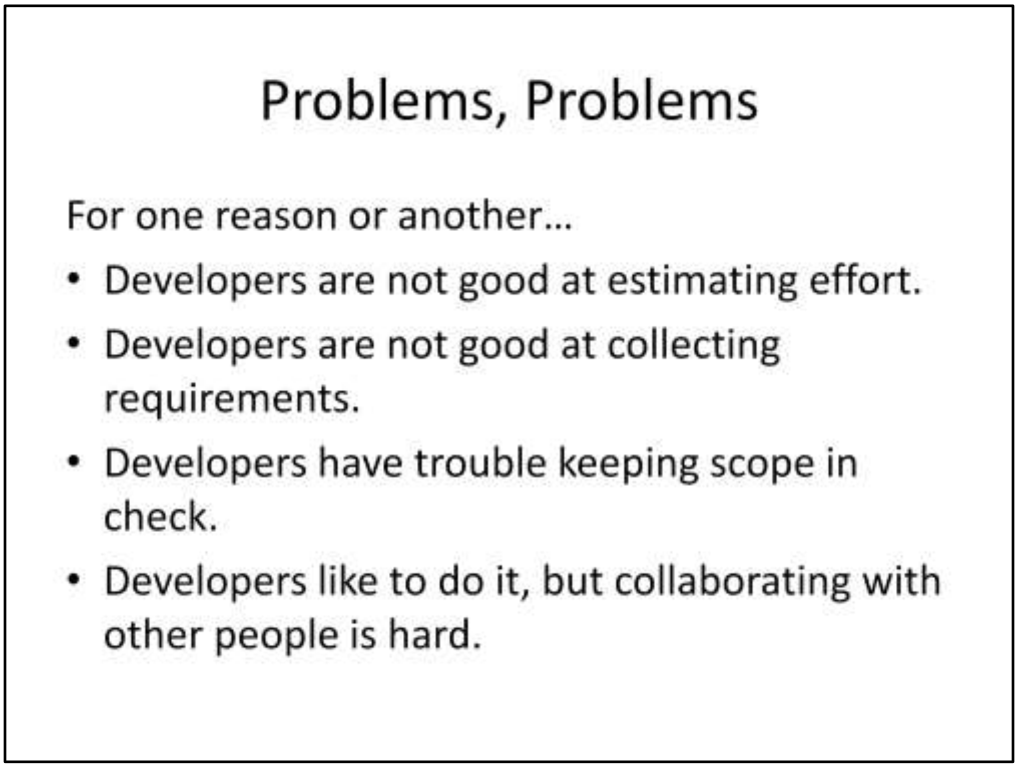
Point 3: We're really interested in hearing from people who want to work with us. We've got a lot to offer, and can utilize other people's resources pretty effectively without stepping on any departmental toes. We're also open to outside collaborators: developers/visionaries at large in our general area, at other universities, and within the open source communities we're part of.



About Me

- I'm Josh Johnson, an applications analyst at the University of North Carolina, Chapel Hill.
- I work for the Lineberger Comprehensive Cancer Center.
- I deal in web applications built to facilitate bioinformatics collection and analysis.
- I have a serious problem with unmanaged development projects.

More Resume Stuff: I've been a professional web developer since 1999. I've been programming since I was about 15 (if you don't count futzing with Q-BASIC programs in MS-DOS 6 some years prior). I did PHP development at a high level, almost exclusively, for the first half or so of my career. Lately I've been working with Python and doing a lot with Plone, and WSGI. I also am pretty good with Perl, I'm Cold Fusion MX certified, and have real-wordl experience with a strange beast called 'OmniMark'.

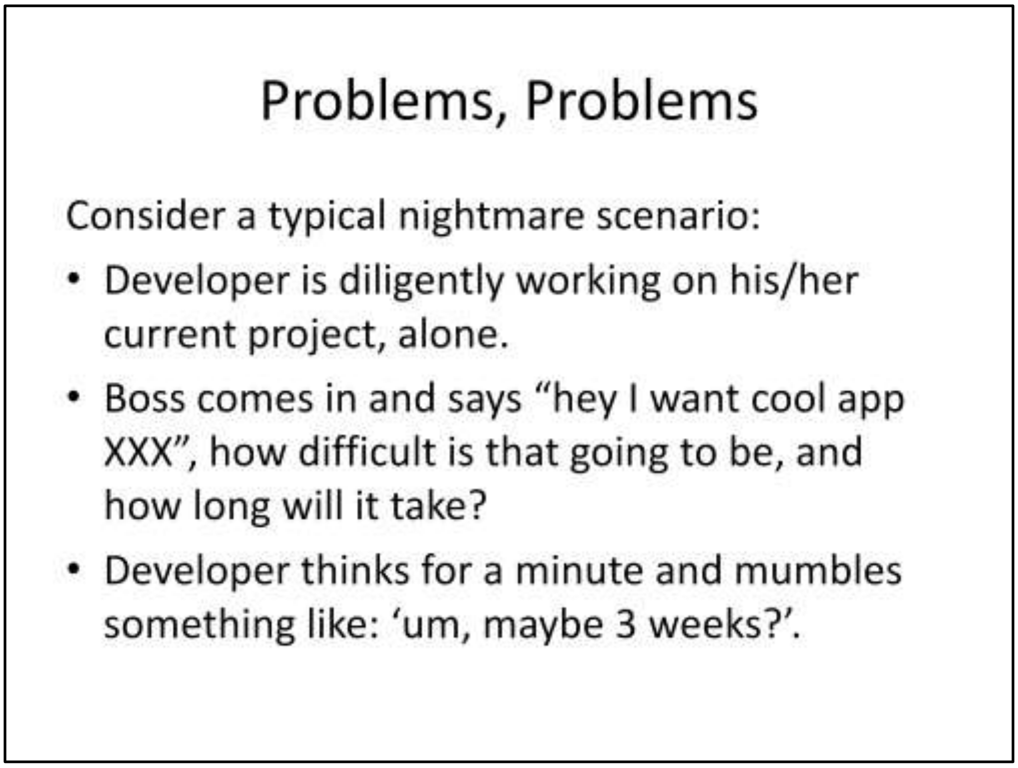


Problems, Problems

For one reason or another…

- Developers are not good at estimating effort.
- Developers are not good at collecting requirements.
- Developers have trouble keeping scope in check.
- Developers like to do it, but collaborating with other people is hard.

I don’t like to generalize, so lets not assume that I mean *every* developer is bad at *all* of these bullet points. But I think its reasonable to assume that *any* developer is probably had difficulty with at least *one* of these problems during their career.

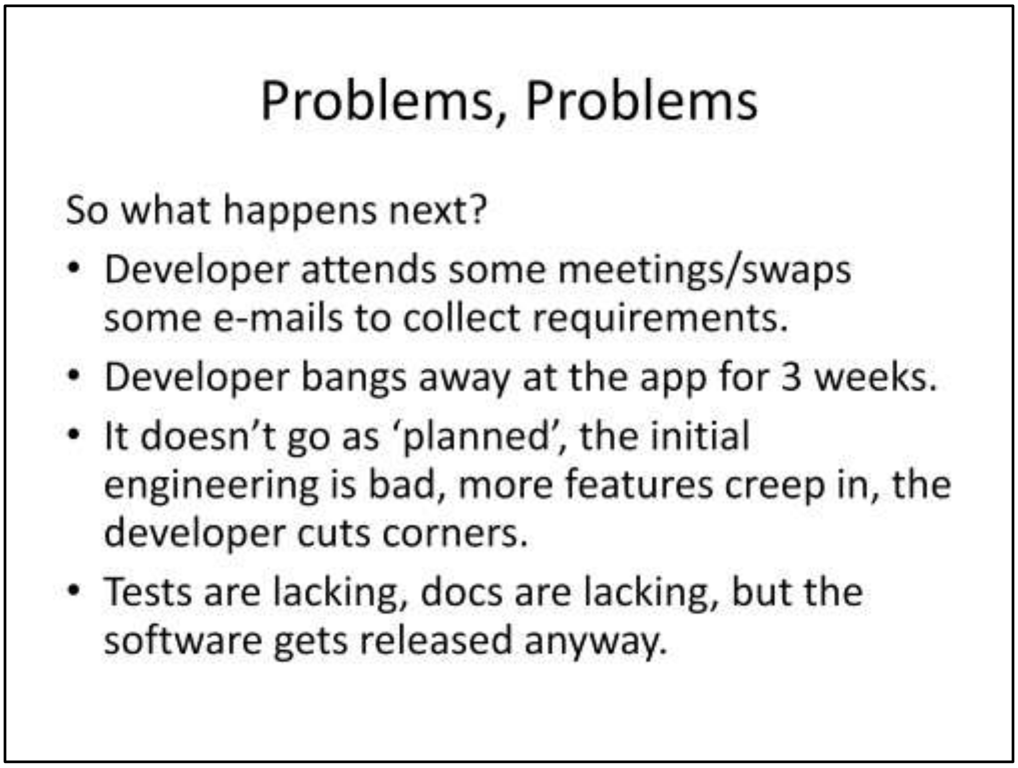


Problems, Problems

Consider a typical nightmare scenario:

- Developer is diligently working on his/her current project, alone.
- Boss comes in and says “hey I want cool app XXX”, how difficult is that going to be, and how long will it take?
- Developer thinks for a minute and mumbles something like: ‘um, maybe 3 weeks?’.

This is the sort of things I have nightmare’s about anyway.



Problems, Problems

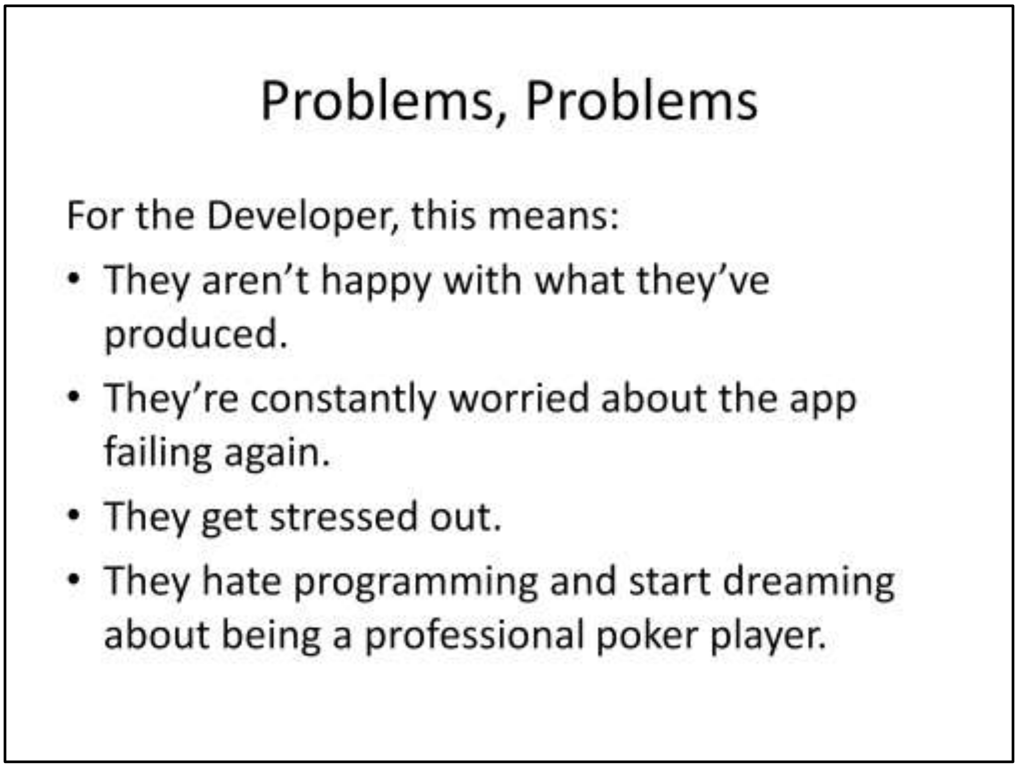
So what happens next?

- Developer attends some meetings/swaps some e-mails to collect requirements.
- Developer bangs away at the app for 3 weeks.
- It doesn't go as 'planned', the initial engineering is bad, more features creep in, the developer cuts corners.
- Tests are lacking, docs are lacking, but the software gets released anyway.

Problems, Problems

And then…

- Developer spends long nights and many hours, while trying to keep up with other work, fixing bugs and adding more ad-hoc features.
- Eventually the app is stable and the Developer is sleeping again.
- But now the 3 week project has taken 3 months.


Problems, Problems

For the Developer, this means:

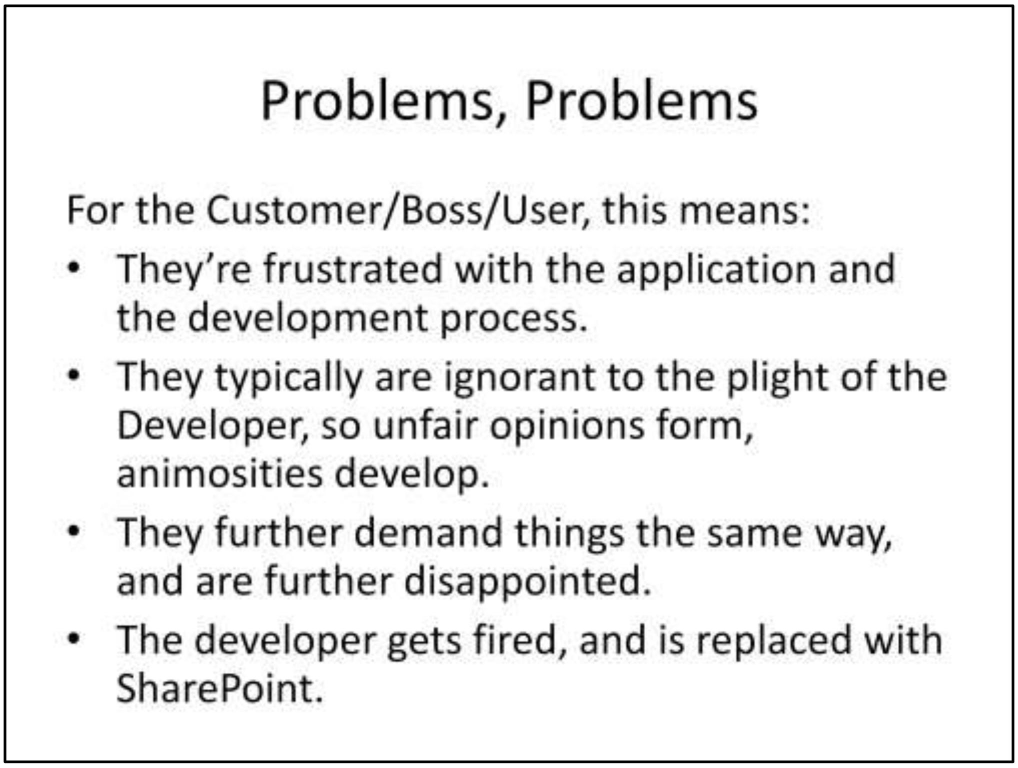
- They aren't happy with what they've produced.
- They're constantly worried about the app failing again.
- They get stressed out.
- They hate programming and start dreaming about being a professional poker player.

Point 1: As developers, especially of a certain skill level and a certain mindset, we care deeply about the quality of our work. When we're doing emergency fixes, we never have time to refactor, or fix tests, or update documentation.

Point 2: As the quality suffers, the stability goes with it. And every time something goes wrong with the production code, they have to stop what they're doing, triage the issue, and then attempt a fix. This often means taking risks with production data. It's not only disruptive, it's a lot to have hanging over your head.

Point 3: I think as developers, we tend to be a bit more high-strung than most folks. We're always thinking, prototyping, planning, researching. Firing on all cylinders, 'full bore', as they say. So when we get stressed out, it's really bad. I've known so many developers with serious health problems, and I'm almost sure it's the stressful nature of our work.

Point 4: *sigh* If I hadn't seen this one first hand it would be really funny.

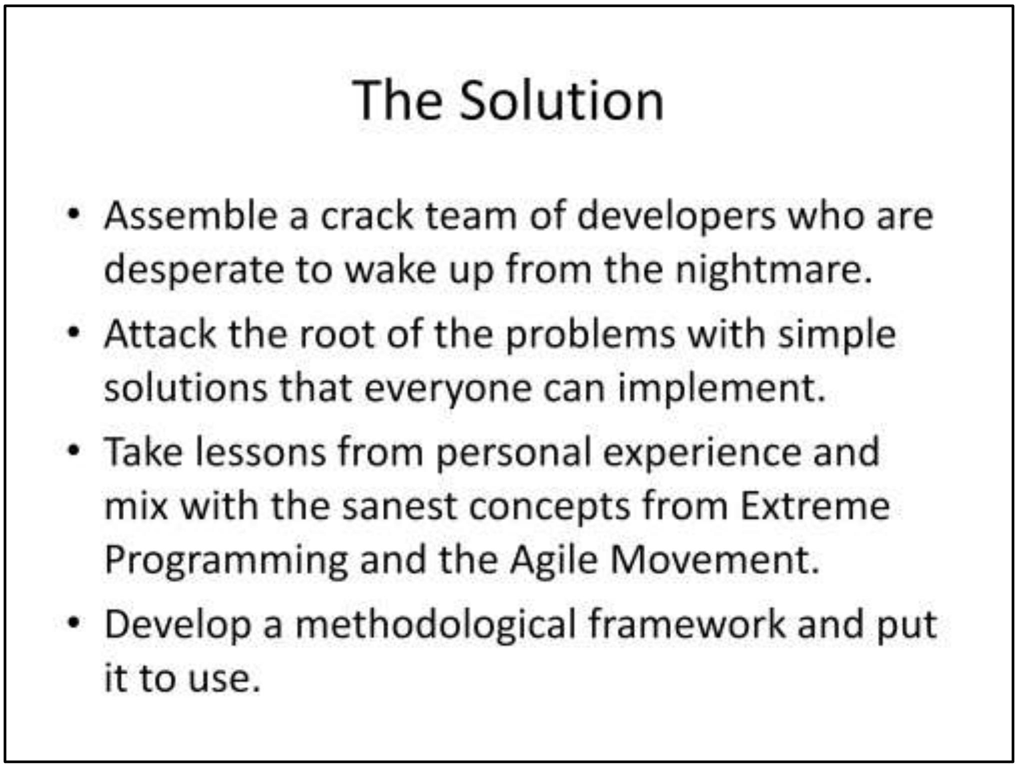

Problems, Problems

For the Customer/Boss/User, this means:

- They're frustrated with the application and the development process.
- They typically are ignorant to the plight of the Developer, so unfair opinions form, animosities develop.
- They further demand things the same way, and are further disappointed.
- The developer gets fired, and is replaced with SharePoint.

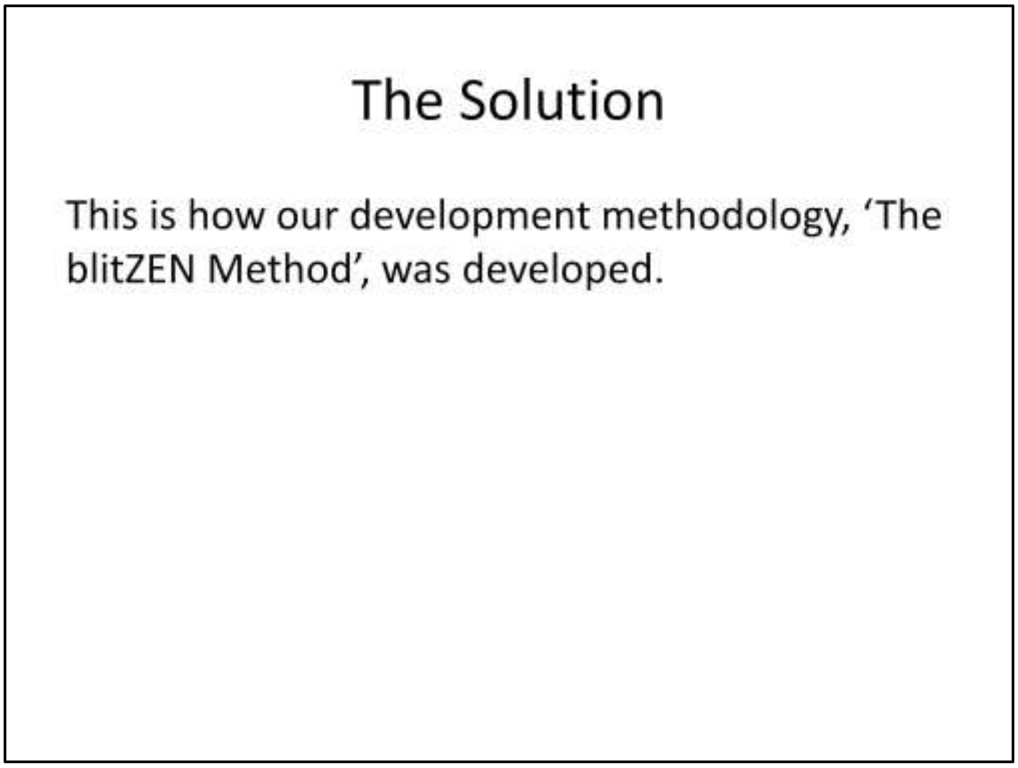
Ha, Ha, Ha.

So what can we do?



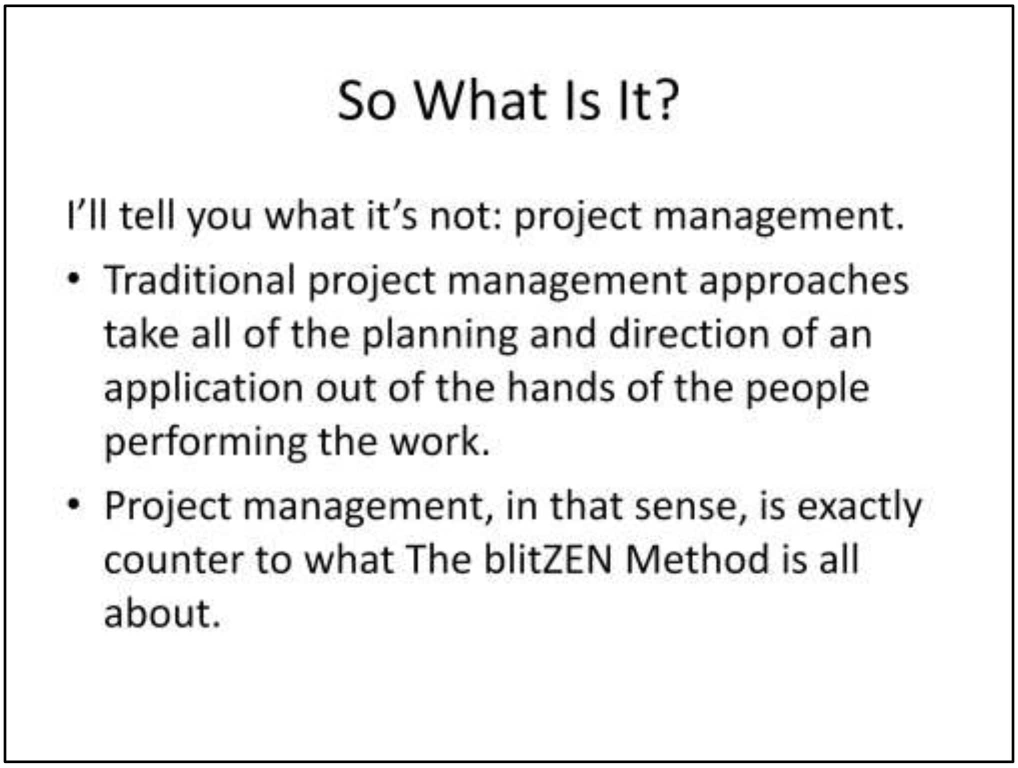
The Solution

- Assemble a crack team of developers who are desperate to wake up from the nightmare.
- Attack the root of the problems with simple solutions that everyone can implement.
- Take lessons from personal experience and mix with the sanest concepts from Extreme Programming and the Agile Movement.
- Develop a methodological framework and put it to use.



The Solution

This is how our development methodology, ‘The blitZEN Method’, was developed.

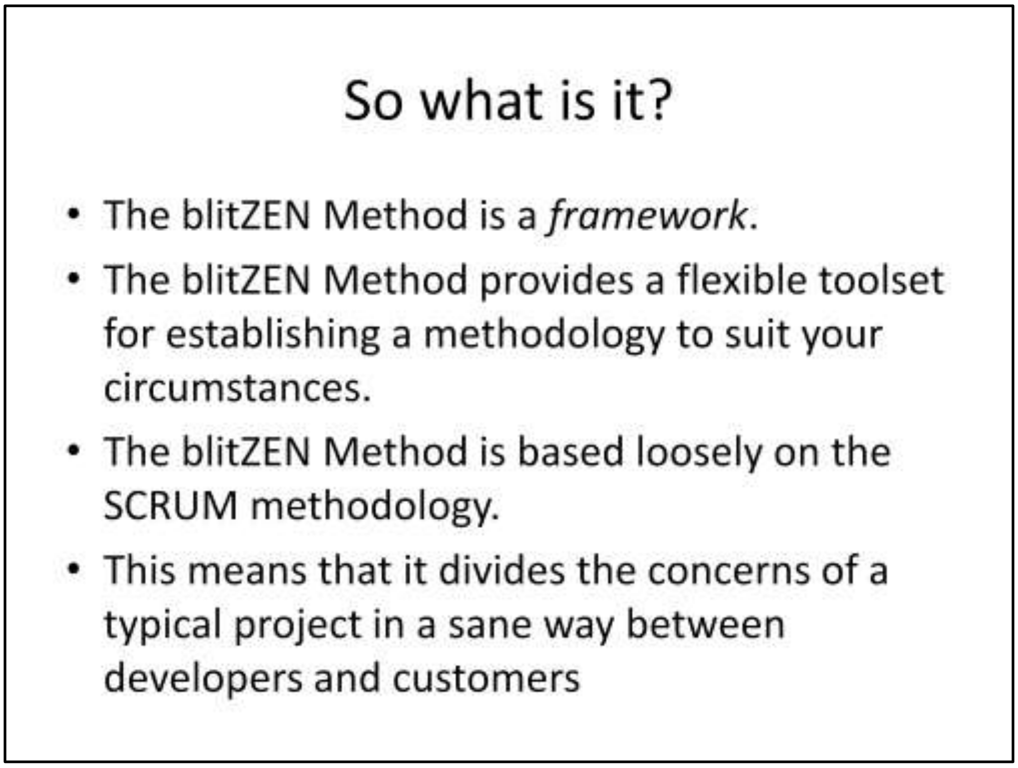


So What Is It?

I'll tell you what it's not: project management.

- Traditional project management approaches take all of the planning and direction of an application out of the hands of the people performing the work.
- Project management, in that sense, is exactly counter to what The blitZEN Method is all about.

The blitZEN Method is more about guiding a development process from beginning to end.



So what is it?

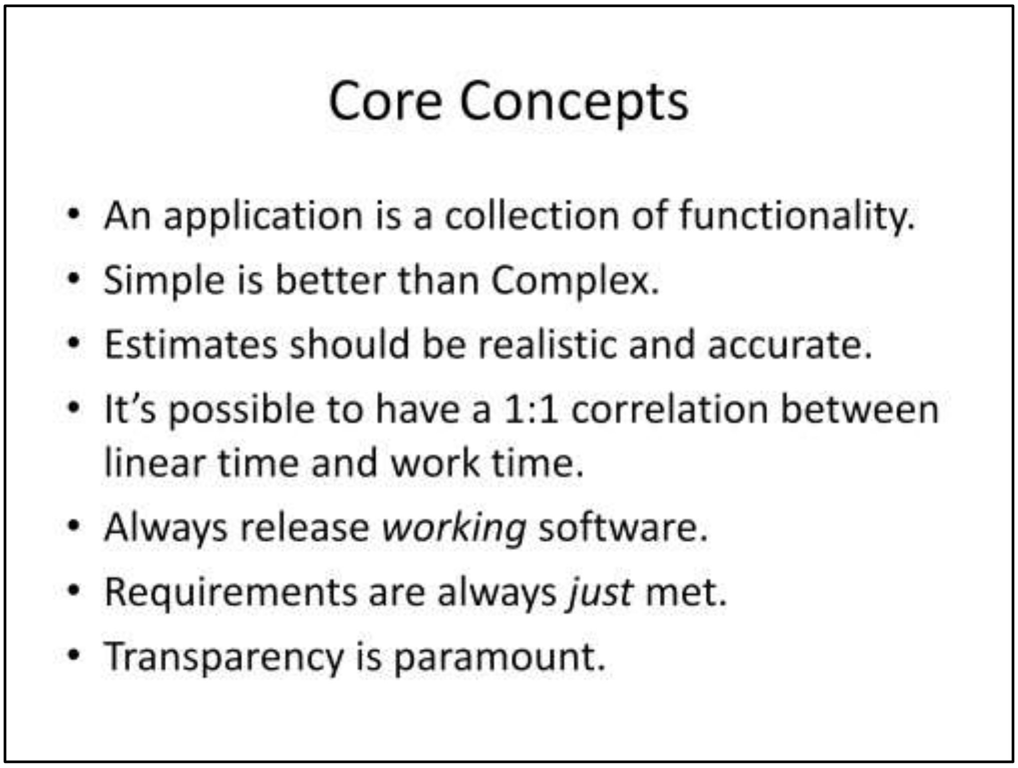
- The blitZEN Method is a *framework*.
- The blitZEN Method provides a flexible toolset for establishing a methodology to suit your circumstances.
- The blitZEN Method is based loosely on the SCRUM methodology.
- This means that it divides the concerns of a typical project in a sane way between developers and customers

Point 1: It’s flexible, it’s there for you to cherry pick the parts you want to use, and expand on.

Point 2: This expands on the framework concept. Your needs may require a more agile, less regimented approach. You can do that. You may be building software for the DOD and have to spec out everything down to the finest detail. It will support that too. You might have customers who are very interested in guiding the process. Great, that’s a big win. But what if you don’t? The process can handle that too.

Point 3: SCRUM was developed as a more general-purpose methodological framework, but it too is very flexible. Check out the SCRUM Guide, a short (21 page) ‘body of knowledge’: http://www.scrum.org/storage/scrumguides/Scrum%20Guide.pdf

Point 4: This is a core SCRUM concept and something we benefit from heavily in The blitZEN Method. In traditional project management approaches, there’s a strict separation between the developers and the customers, often with another completely segmented project manager (or project management team) in the middle. Typically, requirements are just ‘tossed over the fence’ and the development team is left to it’s own devices as far as interpreting the requirements and planning the actual development. SCRUM removes that barrier and integrates the customer into the process. It eliminates the need for a dedicated project manager, and instead one of the developers is donned ‘the ScrumMaster’. The ScrumMaster engages the customers to participate in the process, and picks up the slack when they are unable (or unwilling). This has the effect of shifting some of the burden of requirements gathering and design work off of the developers or a detached project manager and on to the customers. Essentially, it encourages the customer to really take ownership and responsibility for the project’s success. The end result is a better product, a customer more in tune with what’s gone into their product, and developers that aren’t impeded by a lack of access to the customer (or worse, impeded by the customer having access to *them*)



Core Concepts

- An application is a collection of functionality.
- Simple is better than Complex.
- Estimates should be realistic and accurate.
- It's possible to have a 1:1 correlation between linear time and work time.
- Always release *working* software.
- Requirements are always *just* met.
- Transparency is paramount.

Point 1: this is important. Instead of looking at an application as a data store or a user interface, we look at it as a collection of *things people do*.

Point 2: this is a python axiom, but it's always easier to solve a big complex problem (like, say a large web application) by breaking it down into smaller pieces. This also creates the potential to work on different features in parallel. Another benefit: if we break down a complex problem into smaller pieces and document those pieces, we can more easily ensure that we've captured all of the requirements, easily set priorities, etc.

Point 3: there are few things worse than throwing an off-the-cuff estimate to your boss and having to come back and change it after you have more time to think.

Point 4: Possible doesn't mean it ever happens, but it's one of our goals. It's nice to be able to say "we'll have the first version of the software in your hands on April 20th" with confidence, and know that's the amount of work we've actually put into it. Even if a project took 12 months from start to finish due to delays or other happenstance, I can document that we actually only worked on the project for, say 3 months. People (non technical people especially) don't think in terms of effort. If I say "it'll take a week", what I mean is it will take me approximately 40 hours of constant, uninterrupted, hands-on-keyboard time, not that I'll start it on Monday and you'll get it Friday. And it's hard to articulate that. We want to eliminate that frustration and enhance the communications between customers/bosses and developers.

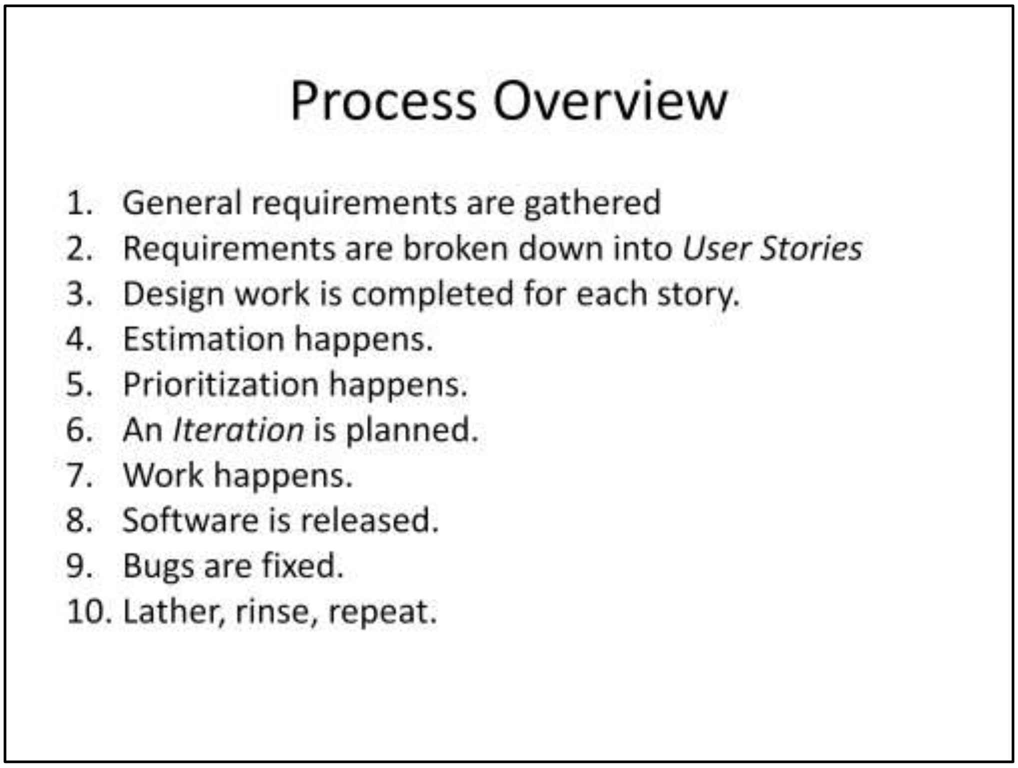
Point 5: This is a big influence on the way we plan and design our applications. We want to ensure that when a development cycle is complete, the user could use it for their day to day duties. It might not be *easy*, but it will be complete. This helps us prove we're making progress, engage our users in giving us interface feedback and finding bugs.

Point 6: this goes hand in hand with point 4. By repeating the mantra of 'does it meet the requirement' and never going beyond, we avoid building features nobody uses, over engineering a feature unnecessarily, and other pitfalls of cowboy programming. This is an iterative process, as long as it meets the requirements, we'll have a shot at making it 'awsome' down the line.

Point 7: this is HUGE. I've run into some serious animosity towards developers in the past. Part of it is just a lack of understanding what we do. Less technical people look at us like evil wizards. The magic we do seems to breed distrust, which puts us under undue scrutiny.

Suddenly the 2 hour office conversation about how the GIL works (or how awesome Rush is), makes us lazy and unproductive, in spite of the many, many extra hours and late nights we routinely put in.

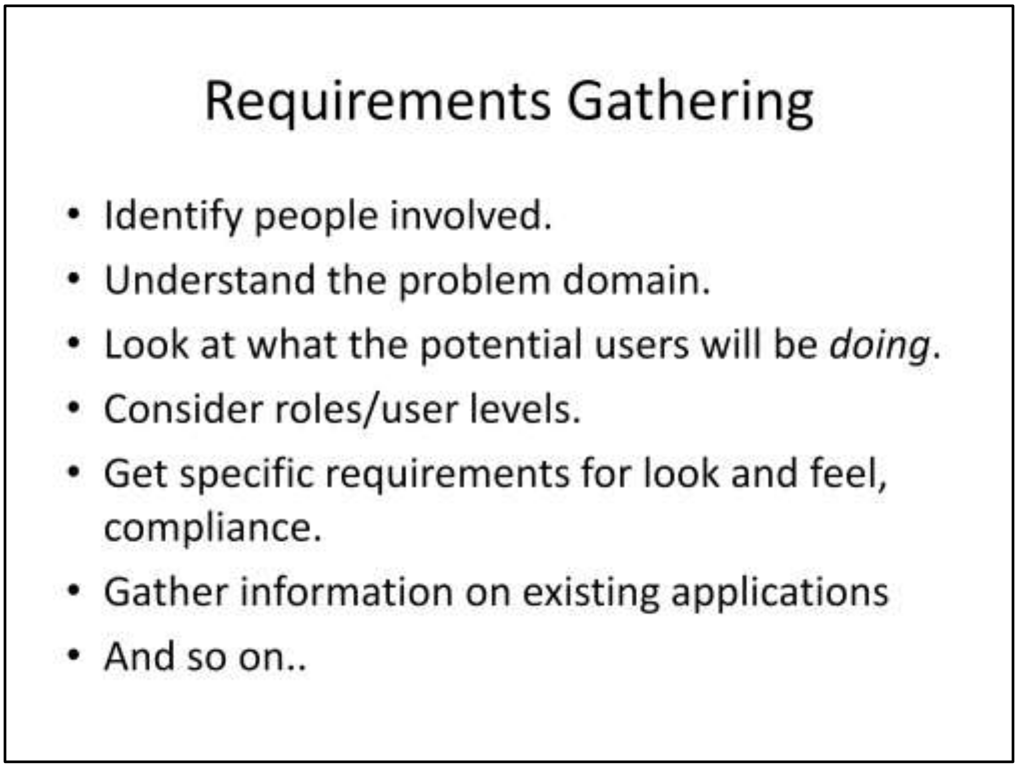
Transparency fixes this. We document everything possible, and put it in places where people who may be concerned about our productivity can see it. We get people involved in the process, so they're expectations are properly managed. We want to replace that distrust with awe and appreciation. ☺



Process Overview

1. General requirements are gathered
2. Requirements are broken down into *User Stories*
3. Design work is completed for each story.
4. Estimation happens.
5. Prioritization happens.
6. An *Iteration* is planned.
7. Work happens.
8. Software is released.
9. Bugs are fixed.
10. Lather, rinse, repeat.

This is a very general breakdown, based on how we do things within the blitZEN group.



Requirements Gathering

- Identify people involved.
- Understand the problem domain.
- Look at what the potential users will be *doing*.
- Consider roles/user levels.
- Get specific requirements for look and feel, compliance.
- Gather information on existing applications
- And so on..

Point 1: this includes both actual people (customers/principals/eventual users) and roles.

Point 2: During this phase we really dig into what our customer's are doing. I've tended to try to not be an expert on the problem domain so I can be more objective, and save time (if I'm building a particle accelerator management app, do I really need to understand the finer points of the Heisenberg Uncertainty Principal?) but I've found it extremely helpful to try to get a really good handle on the industry or scientific discipline as much as I can. This also helps bridge the communications gap and leads to better requirements capture.

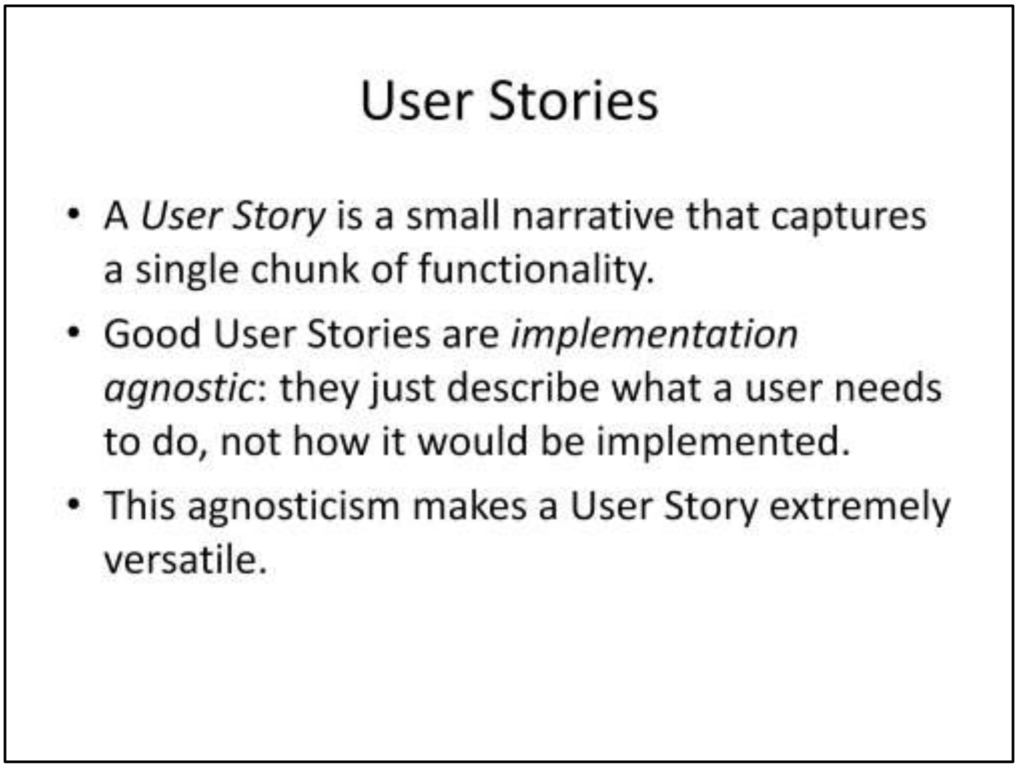
Point 3: This is a big point. This methodology doesn't take many philosophical stances. This is probably the only one. Forget about data models, forget about engineering, forget about the user experience, and focus on pure functionality. There will be a time when you can refactor, enhance, or otherwise make the application 'awesomer'.

Point 4: this is a continuation of the first bullet point, but we get into more specifics about what sorts of things the users will be doing and when or if it should be restricted.

Point 5: gather industry standards for the problem domain, pin down if there are any regulatory statutes in play (Section 508) or privacy concerns (HIPPA). Will there need to be mobile access? Is there an existing website theme/layout that the application should fit in with? Those requirements will influence every aspect of the application, from how forms are built to the layout of a page. They also influence the security model and platform decisions.

Point 6: this is a great time to research the competition. This will give you talking points for justification of your project, design decisions, etc. It also can help you understand how other developers are looking at the same problems, and help you understand the nomenclature and key concepts of the problem domain in an often more accessible way.

Point 7: there are a million ways to gather requirements, and a million data points that you might need to consider. This is by far **not** a comprehensive list!

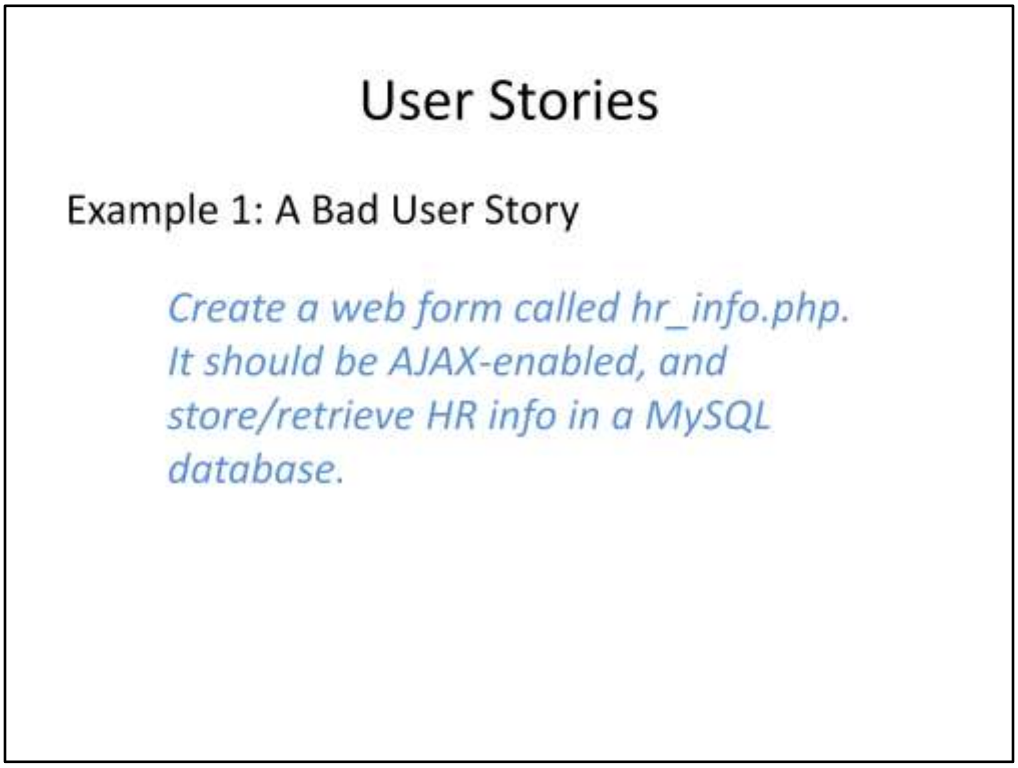

User Stories

- A *User Story* is a small narrative that captures a single chunk of functionality.
- Good User Stories are *implementation agnostic*: they just describe what a user needs to do, not how it would be implemented.
- This agnosticism makes a User Story extremely versatile.

Point 1: Wikipedia has a pretty good article that sums up the industry definition: http://en.wikipedia.org/wiki/User_story

Point 2: This is essential. Remember that User Stories are meant to be expressed in language that your *user* or *principal* will understand, not the developers.

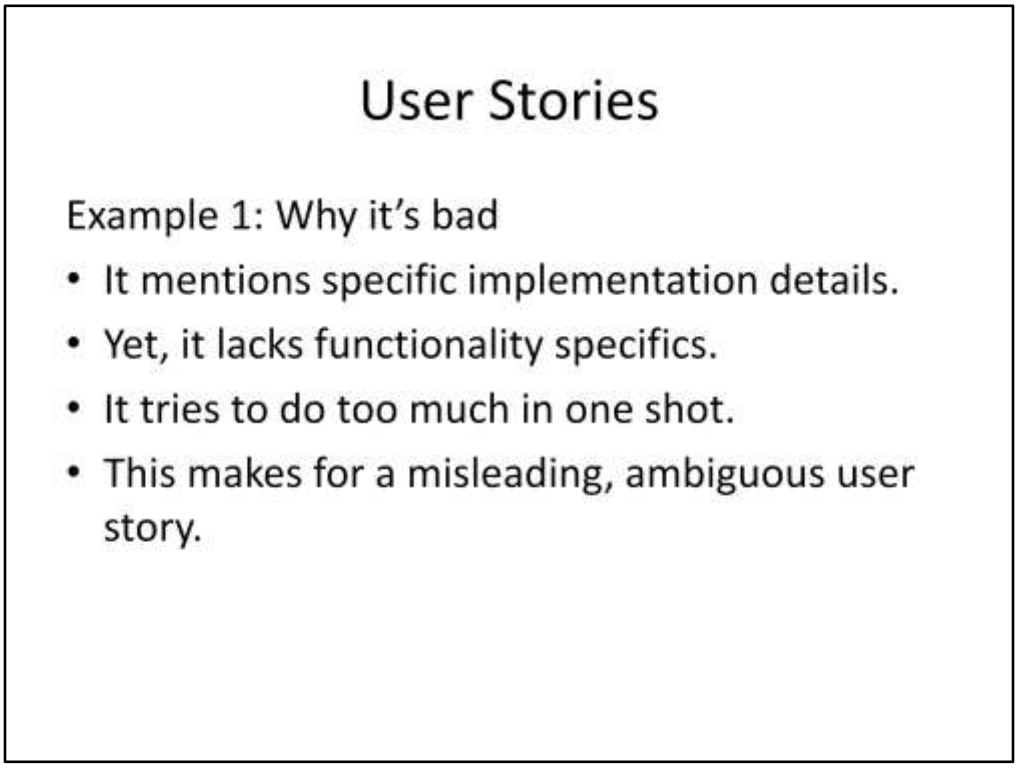
Point 3: the versatility equates to time/effort savings in the long run. A set of user stories can easily become the basis for end-user documentation, a developer guide, a blueprint for integration and systems tests, even a memorandum of understanding or customer invoice. It also means rewriting the software for a new platform becomes less complicated. Instead of trying to assess what the software is doing yourself, you have a very good narrative explaining each piece of functionality in neutral terms.



User Stories

Example 1: A Bad User Story

Create a web form called hr_info.php. It should be AJAX-enabled, and store/retrieve HR info in a MySQL database.


User Stories

Example 1: Why it’s bad

- It mentions specific implementation details.
- Yet, it lacks functionality specifics.
- It tries to do too much in one shot.
- This makes for a misleading, ambiguous user story.

Point 1: it mentions the data store, AJAX, the development platform, and even the url that it will be accessed at.

Point 2: Who uses this? What does ‘HR Info’ mean? Why does it need to be AJAX enabled?

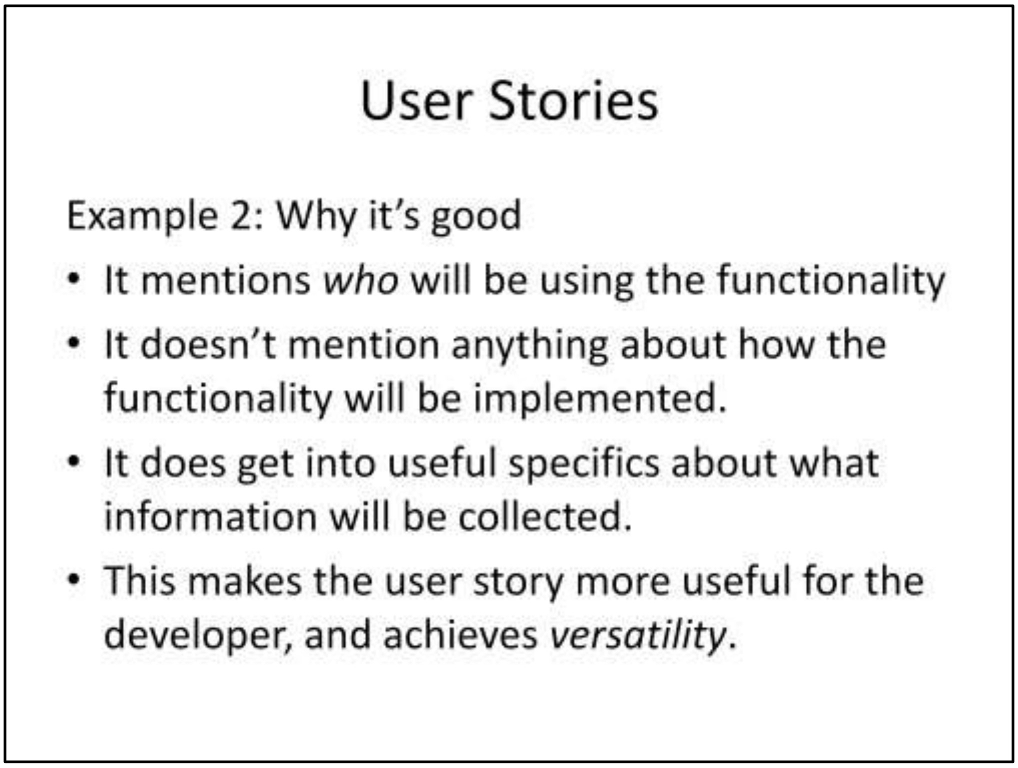
Point 3: It’s talking about a web form, CRUD, and remote access via AJAX. In my mind, the AJAX bit is another piece of functionality. It will undoubtedly have specific requirements.

Point 4: that ambiguity can hide necessary requirements, affect the accuracy of our estimates, and worse, lock us into a certain way of approaching the problem. The beauty of the User Story is that it frees the developer from external constraints, allowing them to craft a solution that best meets the requirements.

User Stories

Example 2: A Better User Story

Our HR manager needs a way to store and retrieve human resources information. Information includes SSN#, typical contact info, salary history, and position information.


User Stories

Example 2: Why it's good

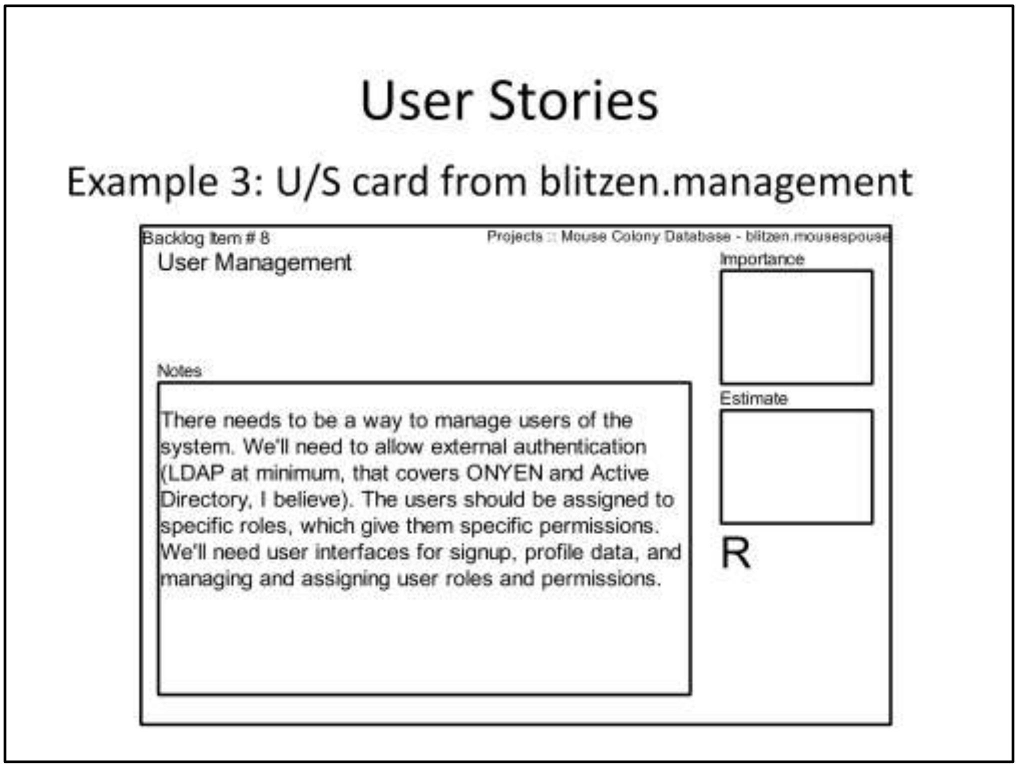
- It mentions *who* will be using the functionality
- It doesn't mention anything about how the functionality will be implemented.
- It does get into useful specifics about what information will be collected.
- This makes the user story more useful for the developer, and achieves *versatility*.

Point 1: This helps define access controls, helps set the scope of the story, and points the developer to the right people to ask for more details during design.

Point 2: This does a bunch of things for us. First, it allows us to decide the best way to implement the requirements without restriction. Second, it takes the user story's text out of the technical realm, making it more accessible to our customers and end users.

Point 3: this is a big deal. I like to look at a user story as a rough blueprint, and blueprints need to be pretty specific to be useful. We focus on capturing specifics that will help direct the design and R&D processes.

Point 4: Another important point. A user story that gives enough information to spark brainstorming on novel and comprehensive (but not necessarily heavily engineered) solutions does so much to facilitate the whole process. Other benefits include the customer/boss/new developer accessibility mentioned before and the user story as a springboard for end-user and developer documentation.


User Stories

Example 3: U/S card from blitzen.management

Backlog Item # 8

Projects :: Mouse Colony Database - blitzen.mousespouse

User Management

Importance

Notes

There needs to be a way to manage users of the system. We'll need to allow external authentication (LDAP at minimum, that covers ONYEN and Active Directory, I believe). The users should be assigned to specific roles, which give them specific permissions. We'll need user interfaces for signup, profile data, and managing and assigning user roles and permissions.

Estimate

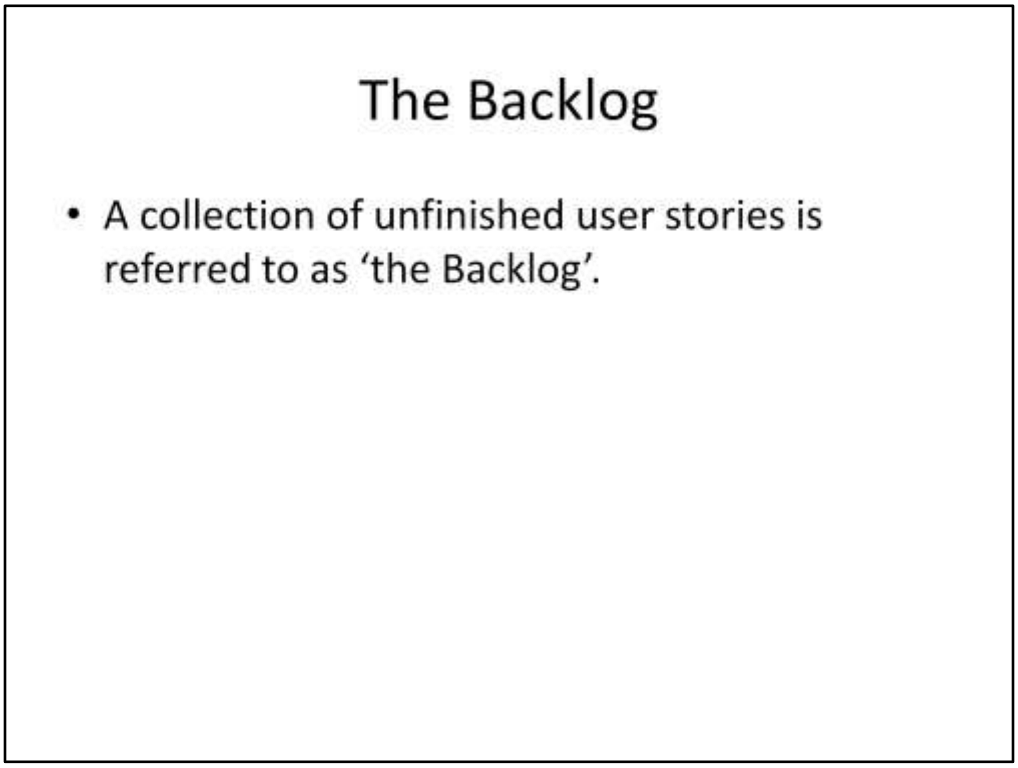
R

We started with an excel-based SCRUM management 'system' that could generate cards that looked similar to this.

We can see the major parts of the card:

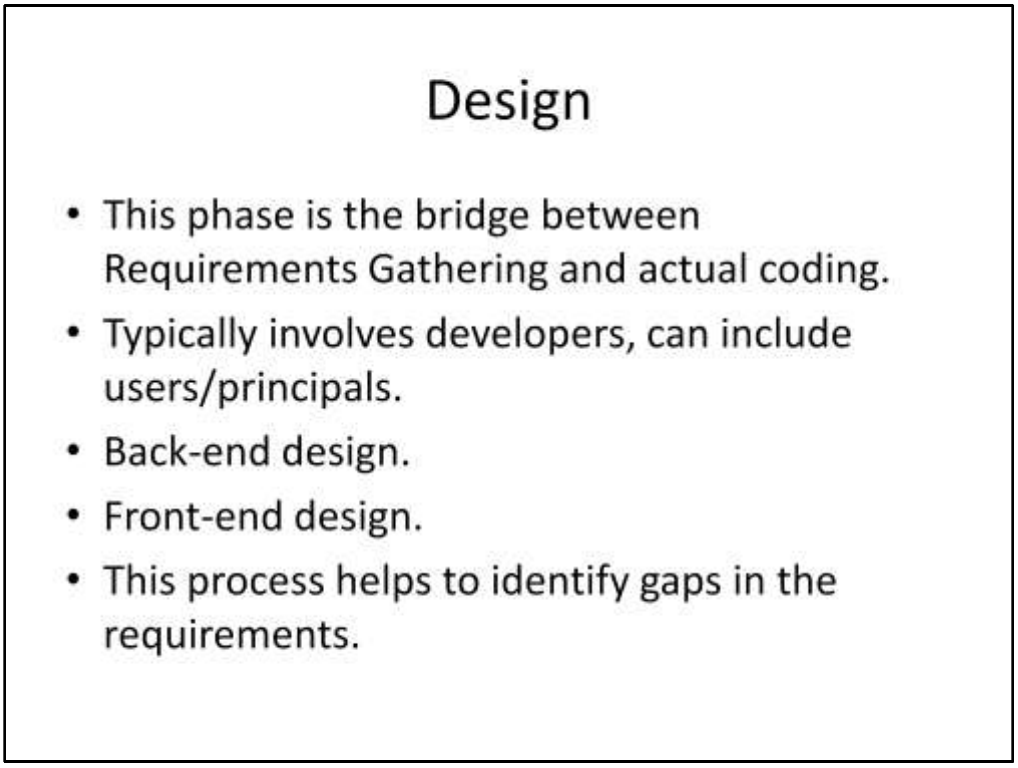
- The title (User Management)
- The body of the user story (labeled Notes here for reasons I can't remember)
- An area to write in the importance and estimate
- The 'backlog item #', used to help identify the user story uniquely amoungst its nieghbours.

Because it's a web-based card, there are also some links in the form of a crumbtrail at the top. The capital 'R' is a symbol indicating that there are *resources* attached to the user story, typically implementation documents, mockups, links to spec documents that the user story has to comply to, etc. We added that so that if we were looking at a printout of the card, we'd know to check the website to look at the resources.


The Backlog

- A collection of unfinished user stories is referred to as ‘the Backlog’.

I put this slide in here to define the term. As User Stories are created, they’re ‘added to the backlog’.


Design

- This phase is the bridge between Requirements Gathering and actual coding.
- Typically involves developers, can include users/principals.
- Back-end design.
- Front-end design.
- This process helps to identify gaps in the requirements.

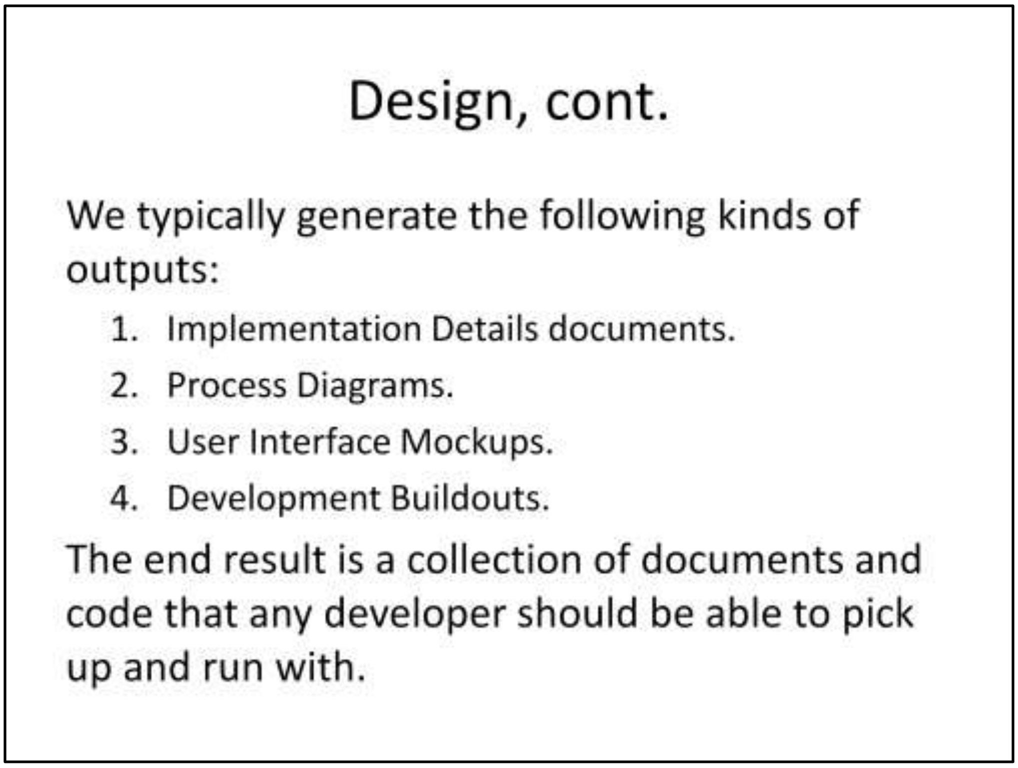
Point 1: This is where we get technical.

Point 2: the level of involvement is variable. If you're generating good design documents, you can scale it back to just one or two core developers. Ideally, the principal(s) and any developers who will be working on the project will be actively involved in the design process, but in reality that is sometimes difficult to do due to time constraints. I've found it works well to have the developers do some design individually, and then get together to discuss and evaluate each others work. We then take the designs to the customer.

Point 3: this is the point where the development team solidifies any platform decisions, does R&D on those platforms, and figures out how to develop and/or deploy them. This would also include data modeling, UML modeling, etc.

Point 4: this means user interface mockups, process flow charts, etc.

Point 5: We've found that this phase of the process can save us a lot of time and headache; it's a great time to interface with users and customers to clarify any ambiguities, discuss user interface concepts, and double-check that the platform chosen will adequately meet the requirements.



Design, cont.

We typically generate the following kinds of outputs:

1. Implementation Details documents.
2. Process Diagrams.
3. User Interface Mockups.
4. Development Buildouts.

The end result is a collection of documents and code that any developer should be able to pick up and run with.

1. We usually create one of these per User Story. The document gets into finer details about how the functionality will be implemented. To use an architectural analogy, If the User Story were the elevation drawing, the Implementation Details would be the structural blueprint. The goal is to be able to hand a reasonably skilled developer the User Story and Implementation details and they would be able to figure out how to build the functionality without much direction.

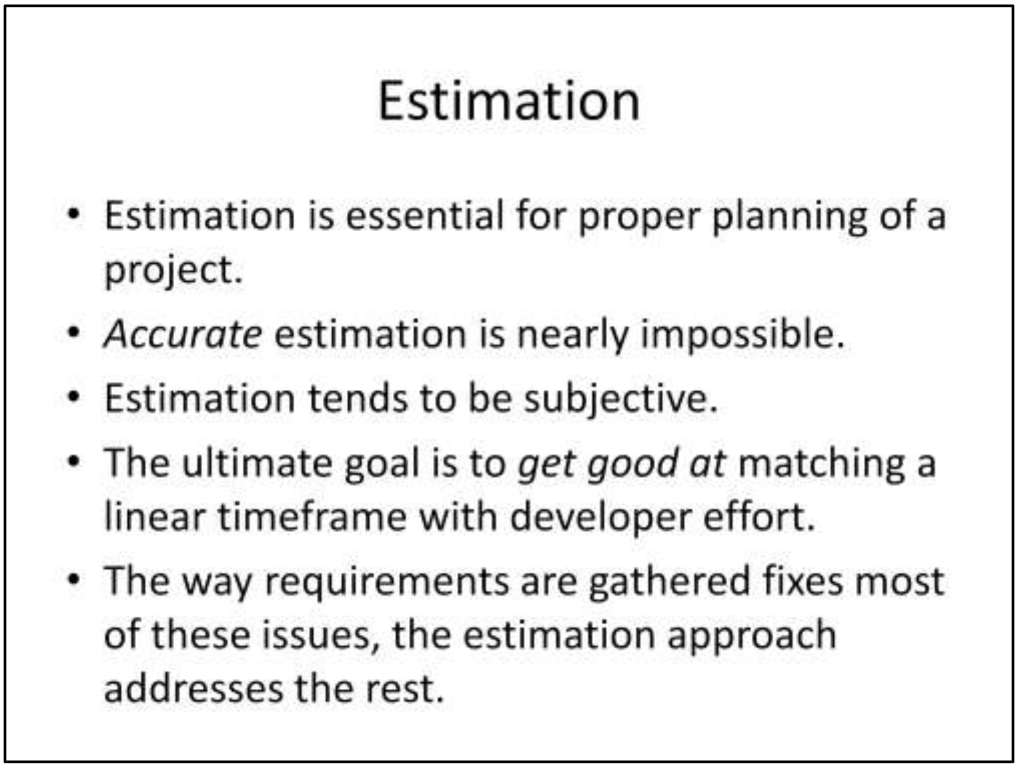
2. I find it helpful to express the processes implied by the user story in simple flow charts. It's a nice way to show the user how the system will work without getting wrapped up in field labels and font colors. We will also typically do larger diagrams showing how the processes in the stories flow together. There are also Use Case diagrams which can be useful for showing how the types of users (Actors) relate to the different user stories.

3. Also useful for conveying to the user what we're planning to do, and helps identify any usability requirements. We try to use generic, pencil-and-paper style mockups to avoid nit-picking layout discussions The way the application looks and the way the user interacts with it is important, but not at this stage. Also, we tend to work with frameworks, and the UI presented to the user will vary depending on what we use, so this also helps keep the document platform agnostic.

4. As part of the R&D process, I like to put together some sort of a build/development/deployment environment. In the case of Plone apps, this typically means a zc.buildout environment, in other situations it's an egg meant to be installed in

a virtualenv. Aside from the obvious advantages, this helps collect libraries/products that are supporting the application, gives us a vector for documenting the development/deployment process, and creates a nice portable ‘download and go’ package any developer could use. Figuring out how to write tests and settling on an approach/file layout happens for us at this stage to (if it’s a new platform).

The last point is a big part of this process. The user story, along with the diagrams, UI, and a development buildout is just about all a new developer needs to complete the functionality. This makes the lead time required for a new developer minimal, and increases our ability to collaborate with other developers and contractors.


Estimation

- Estimation is essential for proper planning of a project.
- *Accurate* estimation is nearly impossible.
- Estimation tends to be subjective.
- The ultimate goal is to *get good at* matching a linear timeframe with developer effort.
- The way requirements are gathered fixes most of these issues, the estimation approach addresses the rest.

1. This is obvious, but it’s important to never loose sight of this fact.

2. You’re never 100% accurate. Aside from unknowns (sickness, hardware failure) there are also intangibles. Things like unresponsive users, mystery requirements that pop up out of nowhere, refactoring, poorly made assumptions about the platform chosen, etc.

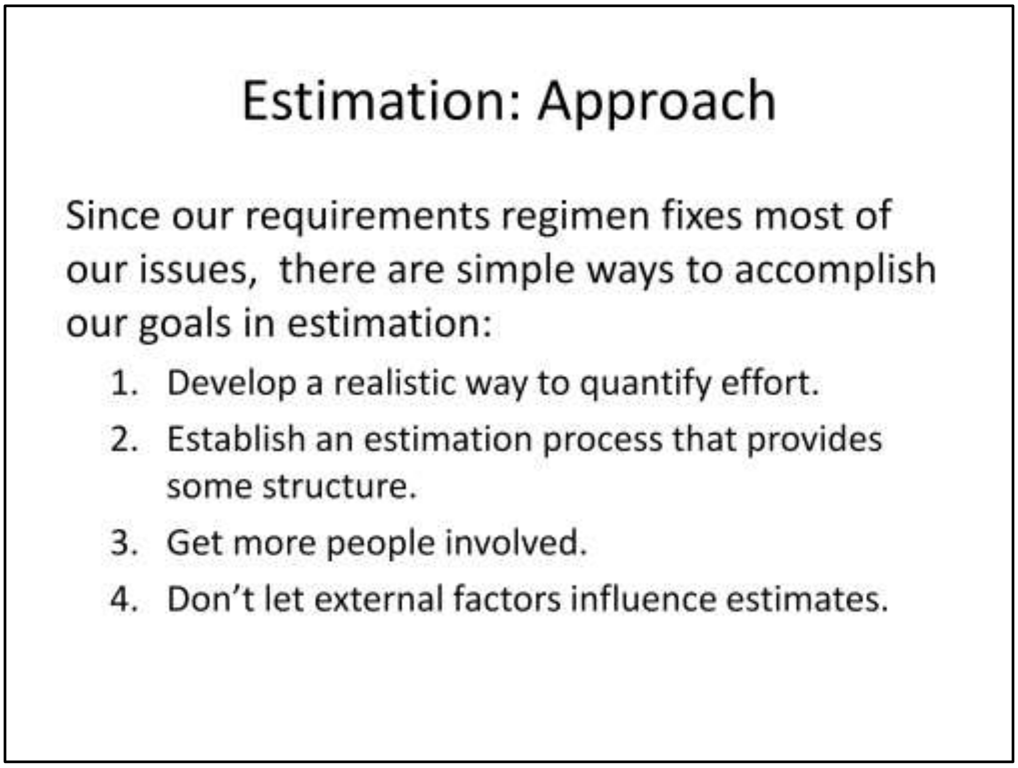
3. When faced with a problem to solve, most people will differ greatly in how they envision it coming together. Software development is especially bad in this regard. There aren’t any good metrics, even with a well-known platform and problem domain with which to easy say “solving problem x will take y number of days”.

4. This is a key point. It’s nearly impossible to hit target dates or estimate project costs if you can’t quantify the real amount of effort a project will take. In our work in a university setting, this means struggling with inadequate funding and resources, since grants are typically written *before* requirements are gathered. Our ability to innovate suffers. In a commercial setting it means wasting money and affecting your bottom line.

   But the key words here are ‘get good at’. The real goal really is estimating with confidence.

5. I really think the main issues with estimates really stem from other problems, like scope creep, bad R&D, poor design, lack of forethought, unclear requirements, and poor communication between project planners and principals. Hammering out the

functionality into manageable chunks, involving the principals as much as possible and putting an adequate amount of effort into design solve the bulk of the problems. Turning those manageable chunks into units of effort and time that almost always hit their targets makes life easier for everyone.


Estimation: Approach

Since our requirements regimen fixes most of our issues, there are simple ways to accomplish our goals in estimation:

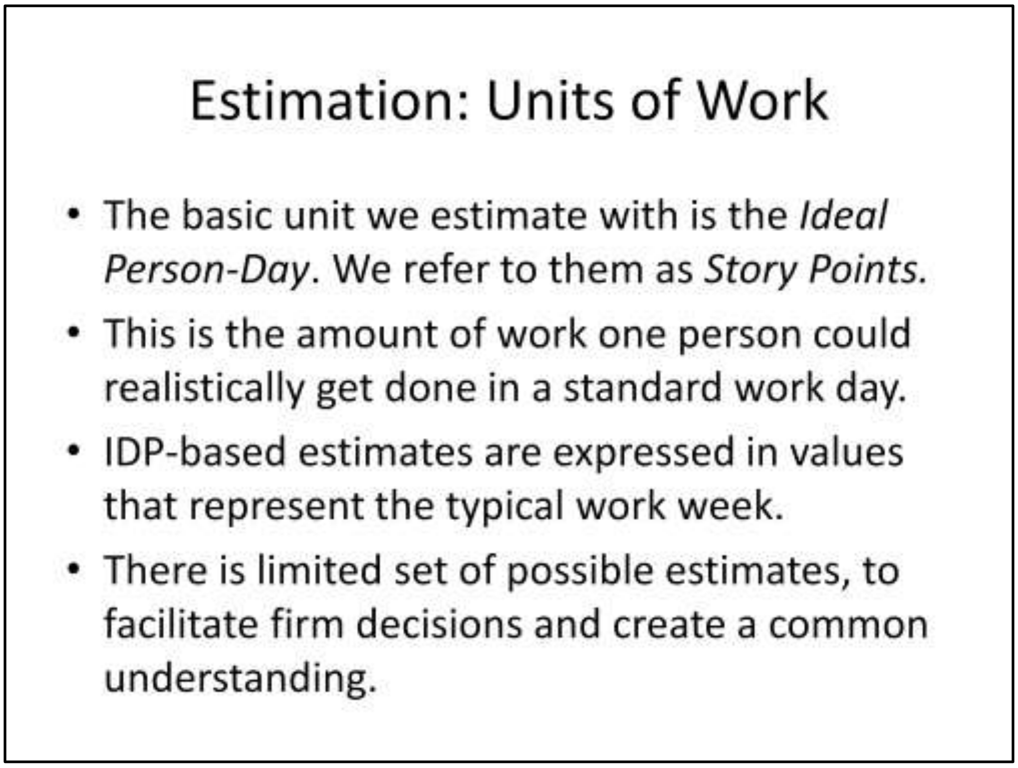
1. Develop a realistic way to quantify effort.
2. Establish an estimation process that provides some structure.
3. Get more people involved.
4. Don't let external factors influence estimates.

1. It's important to be able to express effort in mutually agreed upon terms. When your boss asks you 'how long will that take?' and you say 'about a week', what does that really mean?

2. A big part of the problem with estimates, especially outside of any regimented methodology is that they tend to be off the cuff, ad-hoc, and most often *on the spot*.

3. The more skilled people giving their opinion on how long something will take that you have, the more you can tease out false assumptions, identify best practices, and better utilize the available brain power.

4. This is a really big problem. Your boss comes into your office and says "how long will this wizz-bang thing take", and you say "um…. 2 weeks?", and your boss' expression changes. So either he starts pulling out features because he wants it sooner than it's reasonable, or worse, you change your estimate to less time to accommodate, and you struggle to finish and your quality suffers.



Estimation: Units of Work

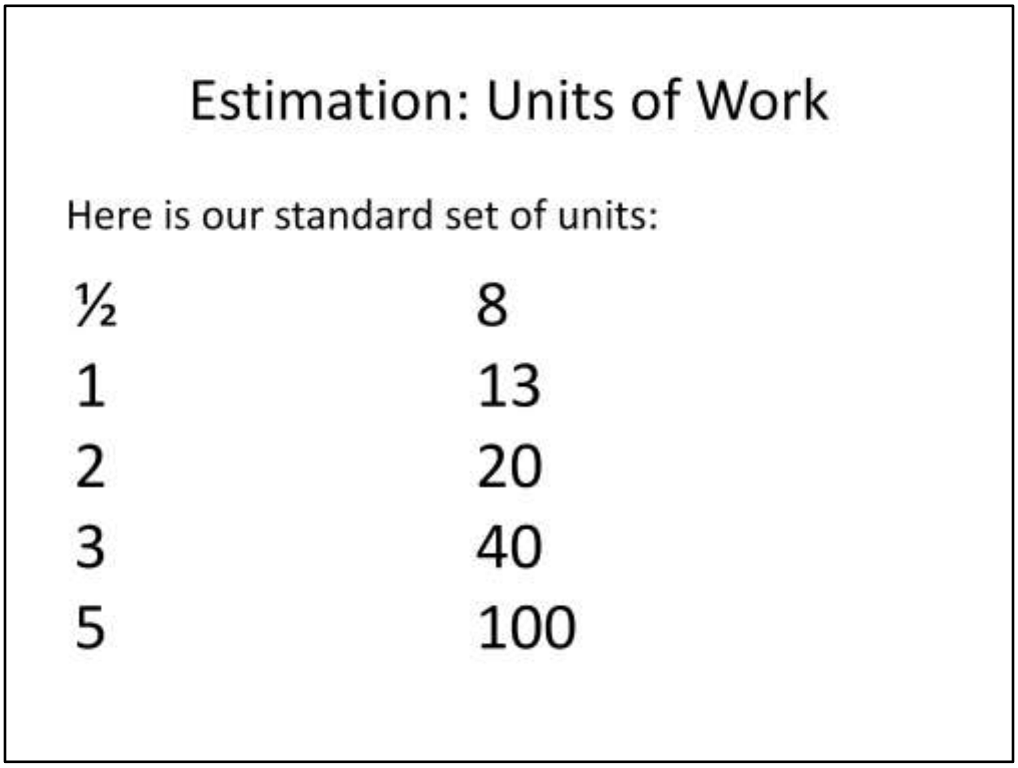
- The basic unit we estimate with is the *Ideal Person-Day*. We refer to them as *Story Points.*
- This is the amount of work one person could realistically get done in a standard work day.
- IDP-based estimates are expressed in values that represent the typical work week.
- There is limited set of possible estimates, to facilitate firm decisions and create a common understanding.

1. I’m not sure where the ‘Ideal Person-Day’ term comes from ☺. ‘Story Point’ is common in the industry. We tend to use Story Point, mostly because it further shields non-technical users from assumptions about linear time, or relative difficulty.

2. It’s important to note that we’re not talking about 8 hours. We’re talking about coming in in the morning at your typical time, taking your lunch break, having a smoke in the afternoon, then going home at the usual time. This helps to put people in the right frame of mind. It also helps the customer (and the project planner) correlate between linear time and effort. It also helps cover certain issues that tend to mess up estimates. Typically if you say something will take 10 hours, what do you really mean? 10 hours in one day? 2 days? 1 hour a day for 2 weeks?

3. You won’t see estimates like ‘26 hours’, or 1.75 IPD.

4. This avoids debate over fractional amounts of effort. The definitions are very explicit; everyone (including your boss) knows exactly what ‘1 IDP’ means.



Estimation: Units of Work

Here is our standard set of units:

½	8
1	13
2	20
3	40
5	100

Remember that we estimate assuming that we're going to come in on a Monday, pick a user story to work on, and work on it. On Friday, we try to not have any major work hanging in an unfinished state.

½: what you could get done between the time you get in in the morning and lunch, or finish in an afternoon. Could you get two of these done in one day.
1: a full day's work.
2: 2 days.
3: 3 days of work. You'd still have enough effort left in the week to do a 2 day project.
5: a full work week.

We've yet to estimate a user story longer than 5 (if even that).

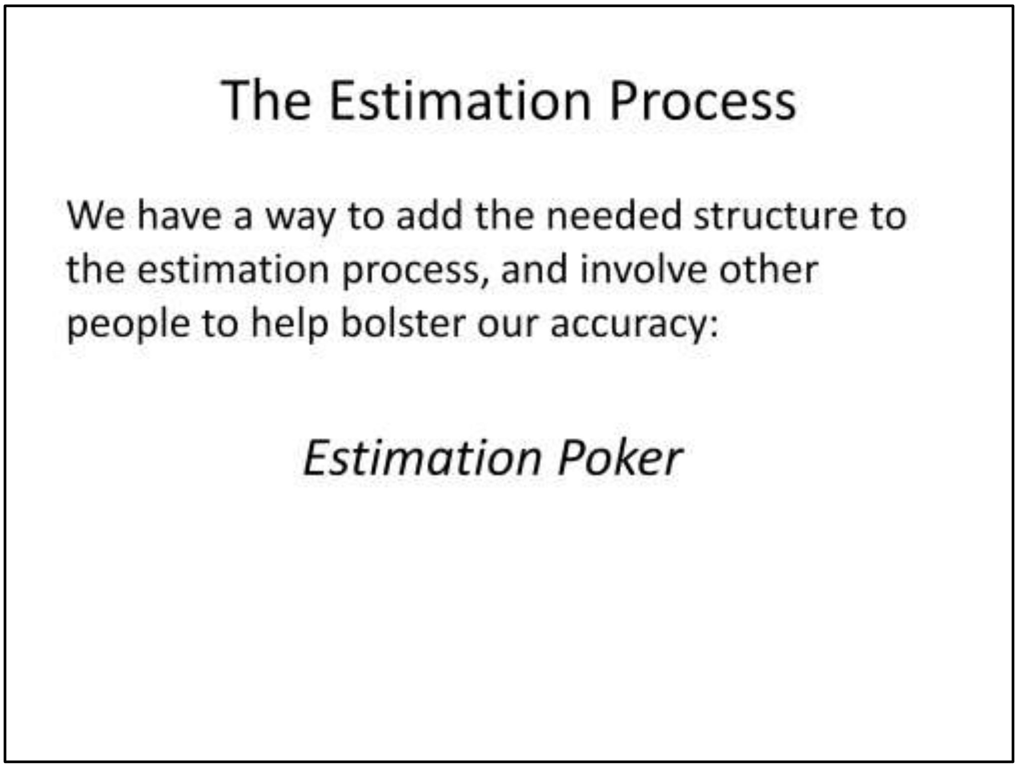
The gaps get bigger here because the actual time taken to complete a task grows as it spans multiple weeks. Realistically, if a project is complex enough to require this much time, it could probably be broken down into smaller user stories.

8: a week and 3 days.
13: 2 weeks, 3 days.
20: 4 weeks
40: 8 weeks
100: 20 weeks

Estimation: Units of Work

Wait, what happened to 4?

- The logic is that if it takes more than 3 days, it will probably take the whole work week.
- Experience has shown that projects spanning multiple weeks will tend to take longer to finish.



The Estimation Process

We have a way to add the needed structure to the estimation process, and involve other people to help bolster our accuracy:

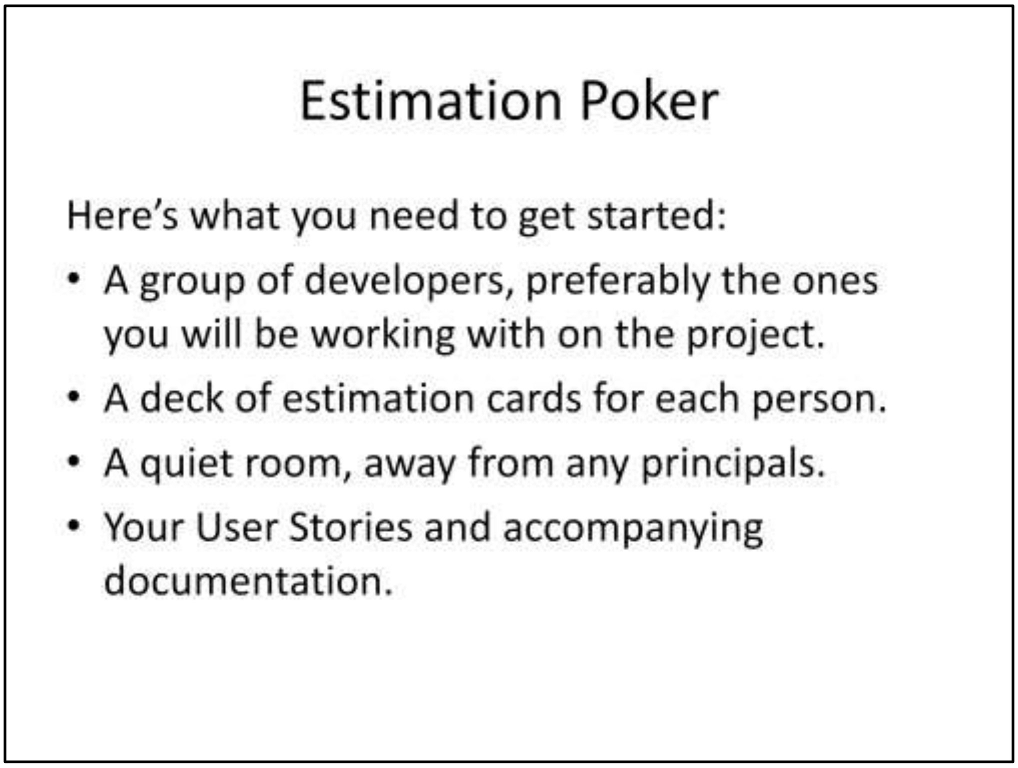
Estimation Poker

Estimation Poker is also known as Planning Poker.

Estimation Poker

Estimation Poker involves a group of people going through each user story in a project and *covertly* choosing an estimate.

The estimates are then revealed, and discrepancies are discussed until a consensus is reached.



Estimation Poker

Here's what you need to get started:

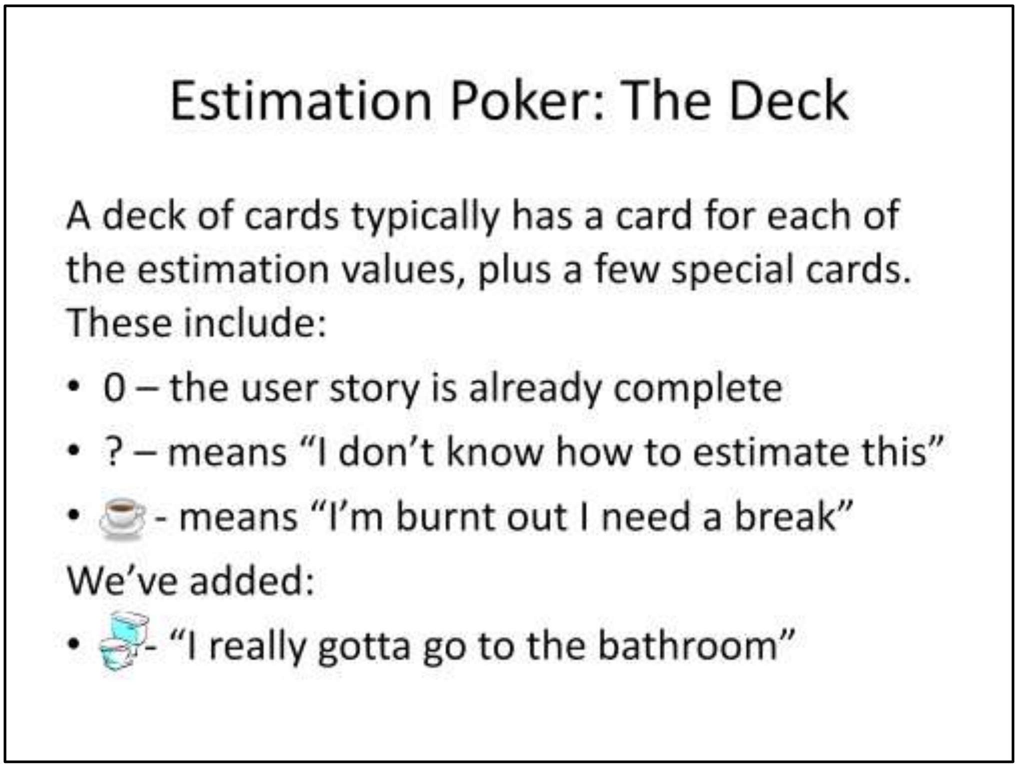
- A group of developers, preferably the ones you will be working with on the project.
- A deck of estimation cards for each person.
- A quiet room, away from any principals.
- Your User Stories and accompanying documentation.

Point 1: you typically estimate with people who you're going to be working with, but as long as they understand what you're doing and the implementation details, any skilled developers can provide the sort of dialogue you need to hammer out good estimates. The more the merrier, but we've successfully done this with only 2 people.

Point 2: you can make these or buy them. We've even done both, and even have done estimation over IRC.

Point 3: this is the one part of the process where having your boss/customer in the room can be a real hindrance. Bosses don't like to hear realistic estimates sometimes, and you really want to make sure that nothing is influencing your estimates. The next (or concurrent) phase of the process, Prioritization, will decide if a feature is going to be put off or implemented. More on that in a bit.

Point 4: you'll need to collect your User Stories and any other resources that will help clarify any ambiguities in the User Stories. In the past we've printed out our stories, but since we developed blitzen.management, we do estimation poker over IRC, or in a room with a projector, so we can refer to the user stories and resources on-line.



Estimation Poker: The Deck

A deck of cards typically has a card for each of the estimation values, plus a few special cards. These include:

- 0 – the user story is already complete
- ? – means "I don't know how to estimate this"
- ☕ - means "I'm burnt out I need a break"

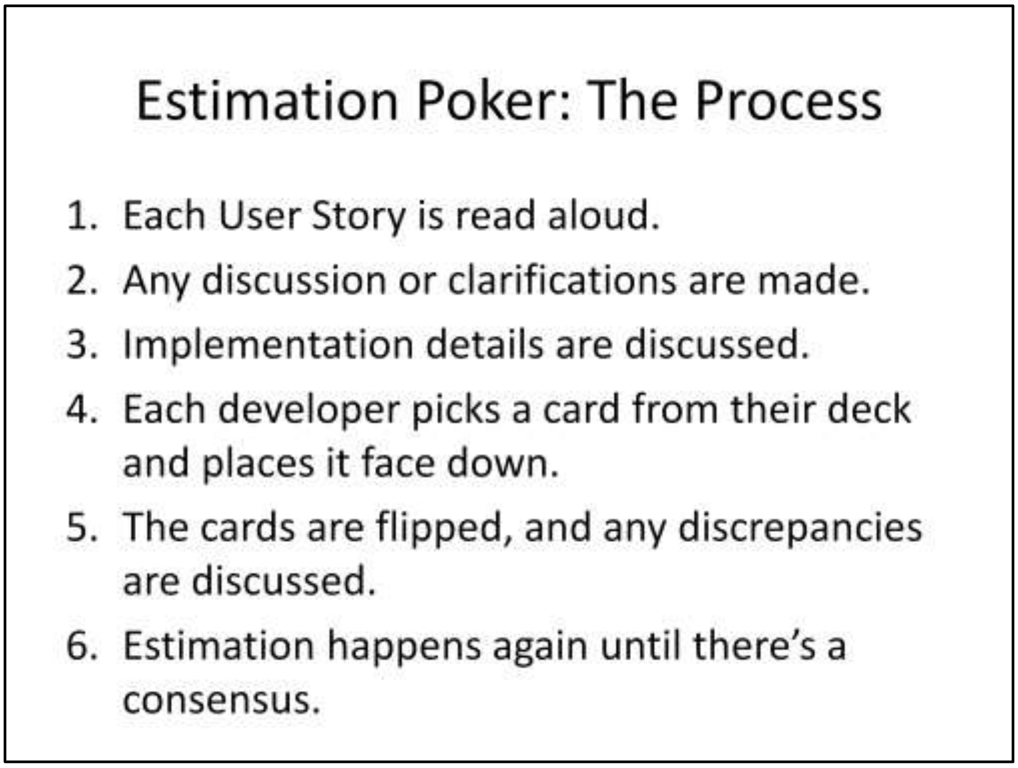
We've added:

- 🚽- "I really gotta go to the bathroom"

0: we also use this to indicate items that will literally take only a few minutes.

?: this comes in handy when you're dealing with developers with different skill levels. Where some folks will tend to pad their estimates to account for their lack of knowledge, it's really best to just say you don't know how to estimate it, and then discuss that with the group.

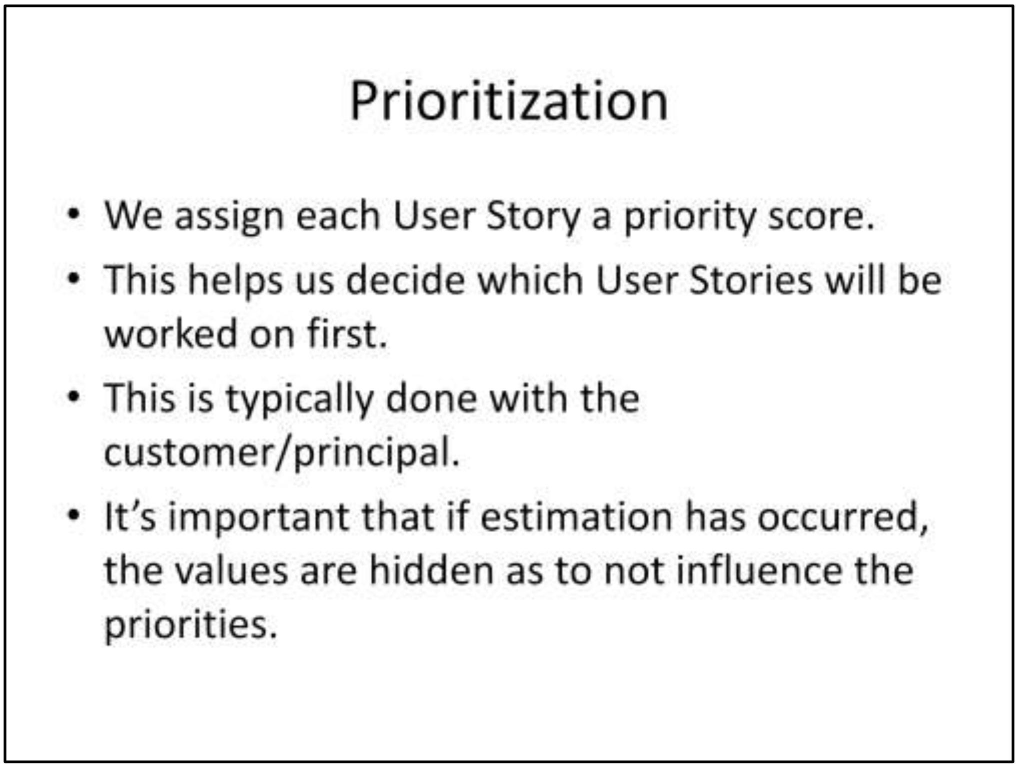
One of the decks we've used (good quality, $5):
http://store.mountaingoatsoftware.com/products/planning-poker-cards-burndown-design


Estimation Poker: The Process

1. Each User Story is read aloud.
2. Any discussion or clarifications are made.
3. Implementation details are discussed.
4. Each developer picks a card from their deck and places it face down.
5. The cards are flipped, and any discrepancies are discussed.
6. Estimation happens again until there's a consensus.

We've never had to, but there is often some sort of an egg timer or other time-limit on the card-picking, discussion, and debating phases.

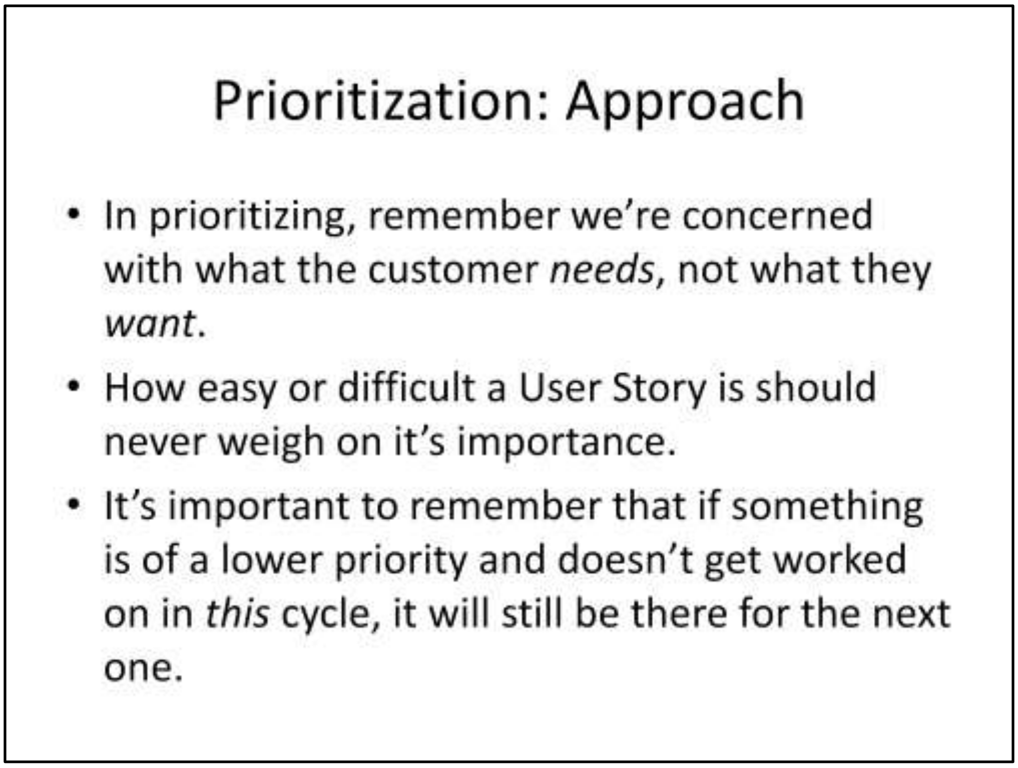
Typically if a consensus can't be reached, someone who's been donned 'the project owner' will make the call.



Prioritization

- We assign each User Story a priority score.
- This helps us decide which User Stories will be worked on first.
- This is typically done with the customer/principal.
- It’s important that if estimation has occurred, the values are hidden as to not influence the priorities.

Point 4: this is a key point, and an issue I’ve run into personally.



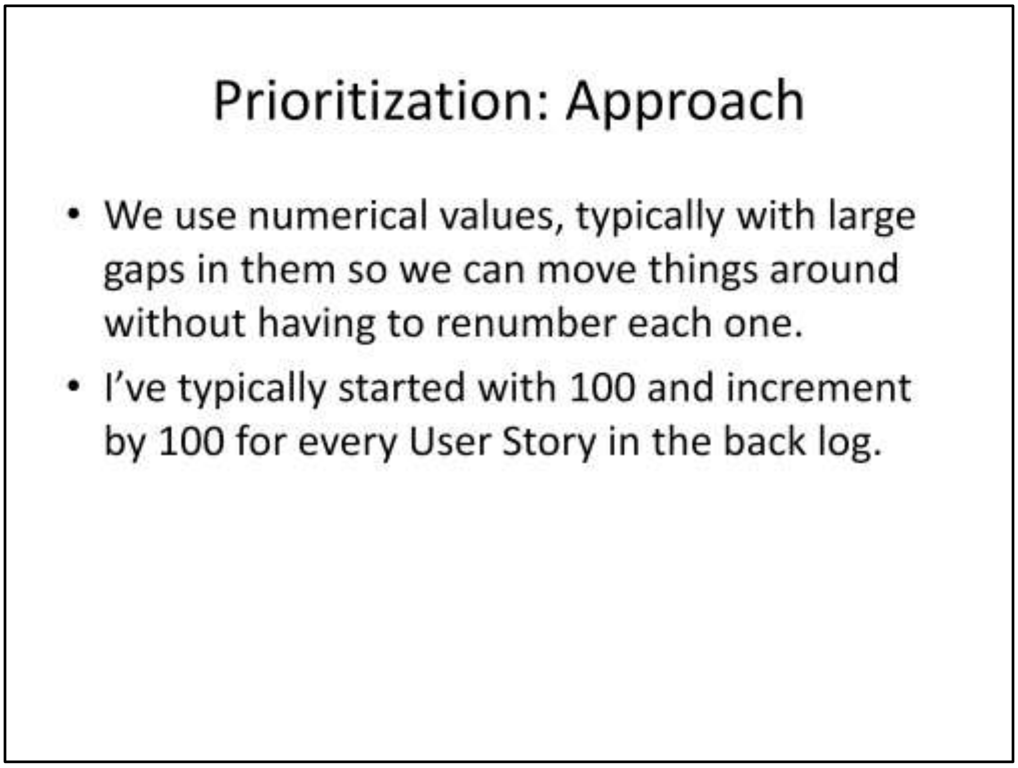
Prioritization: Approach

- In prioritizing, remember we're concerned with what the customer *needs*, not what they *want*.
- How easy or difficult a User Story is should never weigh on it's importance.
- It's important to remember that if something is of a lower priority and doesn't get worked on in *this* cycle, it will still be there for the next one.

Point 1: We've already captured what they're asking us for. Now we need to focus on what's most important to them. This allows us to cull some of the more 'bell and whistle' features outright, or put them off till another development cycle. The up side here is that we never loose any requirements, we just put them in the backlog. This also forces the customer to think about the scope of what they're asking you to do.

Point 2: this is a big one. If it's important it's important. If the customer wants it, it will be done. As developers we don't (or shouldn't) care how hard something is to do. Remember at this point we've captured the requirements, we've figured out how we're going to implement the user story, so if it's made it this far, then it *must* be of some importance.

Point 3: this is easy to overlook. The backlog is not a black hole, it's a staging area.

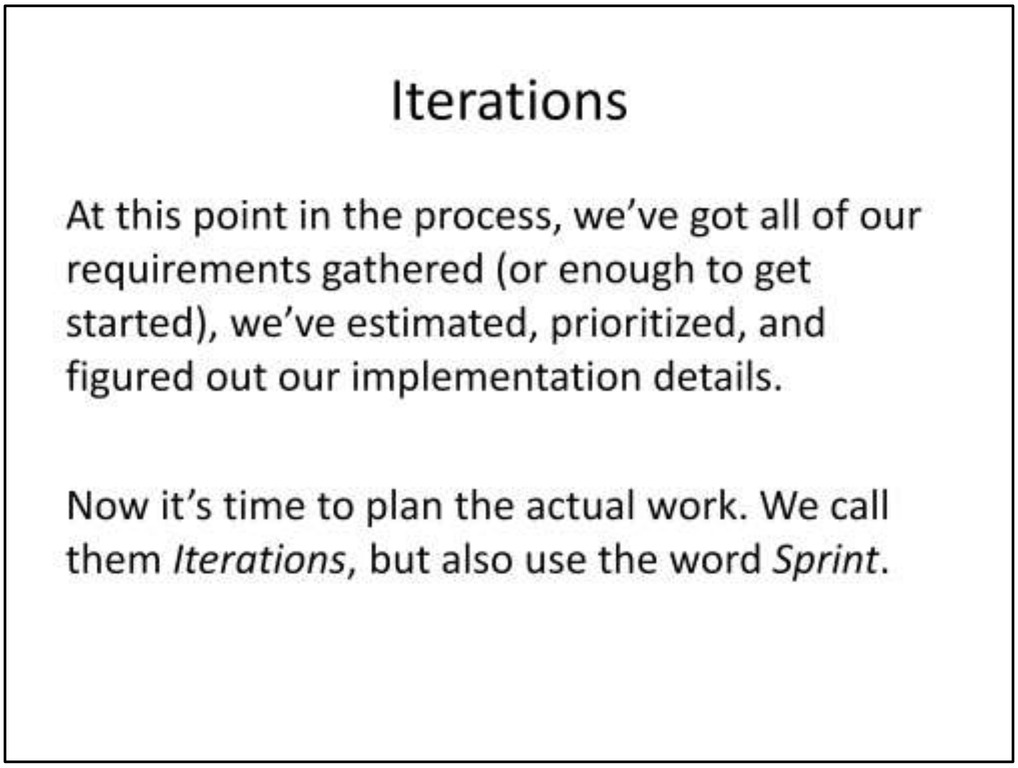


Prioritization: Approach

- We use numerical values, typically with large gaps in them so we can move things around without having to renumber each one.
- I've typically started with 100 and increment by 100 for every User Story in the back log.

Point 1: we'll often think of new user stories or break up existing ones during estimation or prioritization. It all depends on how involved your principal has been in the process.

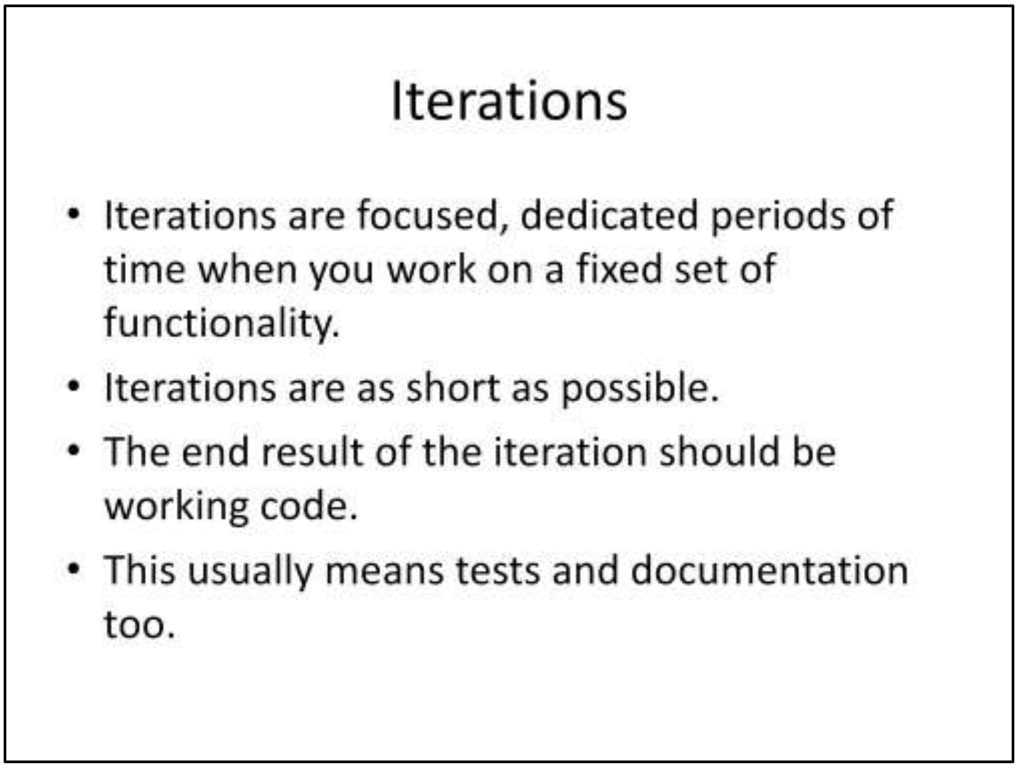
Point 2: like BASIC, but no GOTOs.



Iterations

At this point in the process, we've got all of our requirements gathered (or enough to get started), we've estimated, prioritized, and figured out our implementation details.

Now it's time to plan the actual work. We call them *Iterations*, but also use the word *Sprint*.



Iterations

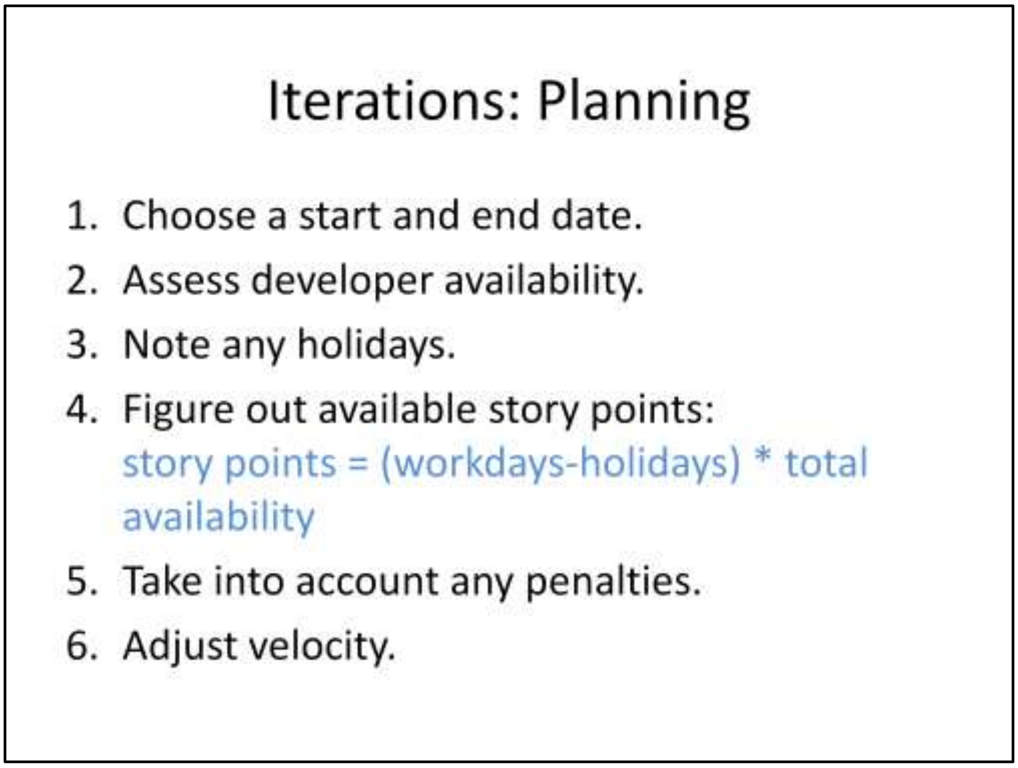
- Iterations are focused, dedicated periods of time when you work on a fixed set of functionality.
- Iterations are as short as possible.
- The end result of the iteration should be working code.
- This usually means tests and documentation too.

Point 1: this is important. It’s possible to do other things while you’re in a sprint, it’s really best if you can focus all of your attention on the project during the sprint timeframe. It’s even more important that the User Stories or other requirements DO NOT CHANGE, under any circumstances, during the sprint.

Point 2: As short as 1 week, but typically 2 weeks. No more than 4. The shorter the better, especially if the project requirements are in flux.

Point 3: Another core concept of this process. You should be able to deploy the code to a staging server for QA and solicit feedback from your principals.

Point 4: SCRUM prescribes that **every** user story should have an automated test to prove it’s complete.


Iterations: Planning

1. Choose a start and end date.
2. Assess developer availability.
3. Note any holidays.
4. Figure out available story points:
   story points = (workdays-holidays) * total availability
5. Take into account any penalties.
6. Adjust velocity.

1. It should always begin on a Monday, and end on a Friday. The duration depends on the number of user stories you'll need to complete to get working software out the door.

2. Figure out who's going to be around and working on the project during that time. If anyone will not be able to devote their full attention to the iteration, pro-rate their availability.

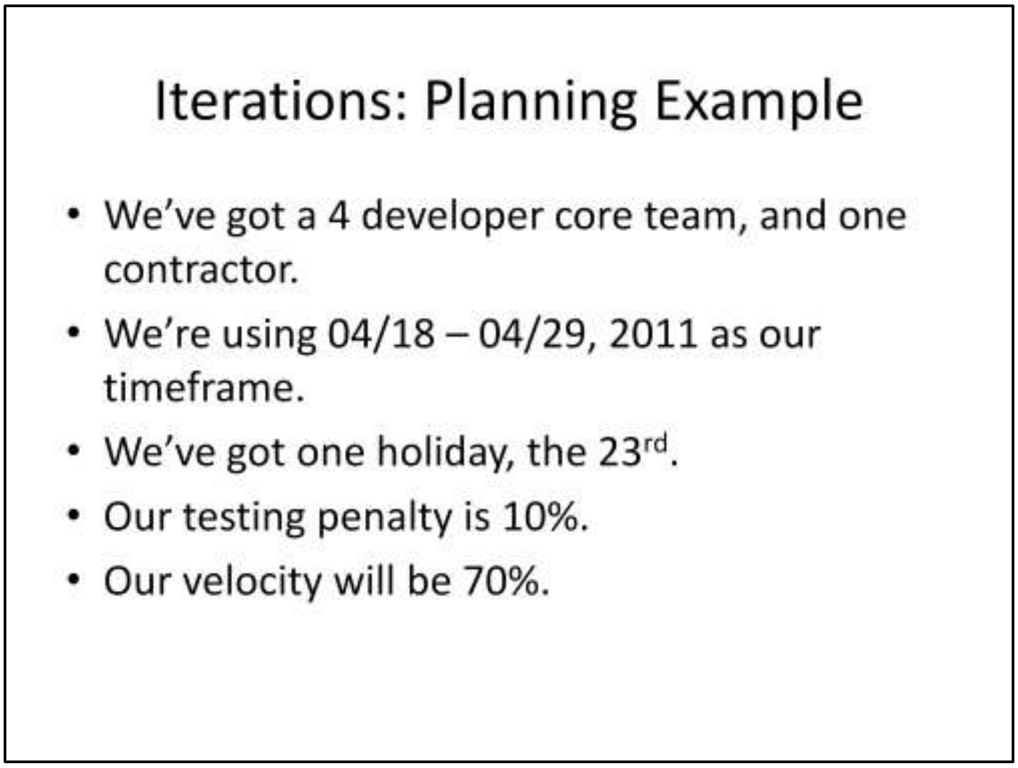
3. Don't work when you don't have to :P

4. An example follows.

5. We'll often add a penalty for testing or documentation, since we tend to not think about that when we're thinking about estimates. This reduces the number of available story points to make sure there's room. This is totally optional.

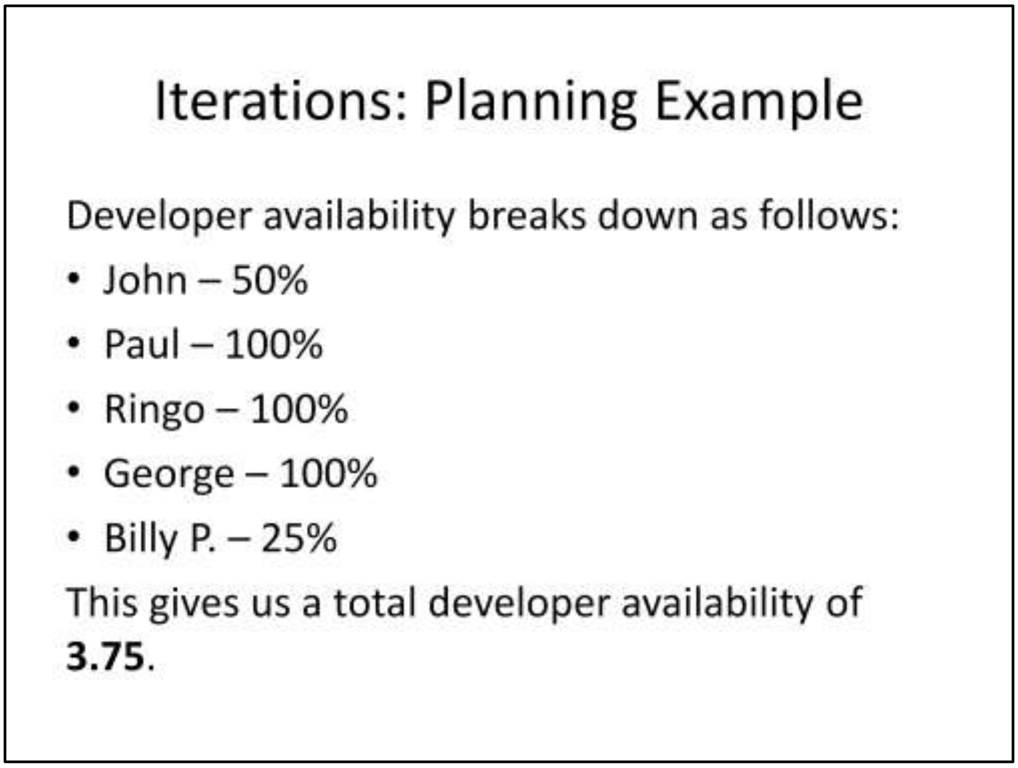
6. Velocity is the rate at which you can complete work. You could think of it as a measure of efficiency, but in this context it's more like a 'realism enhancer'. It's a way to further make the estimates more accurate. 100% velocity would be one story point of work completed, per developer, per day, and that's considered an unattainable goal.

   The amount you use is variable, but most well-run SCRUM shops shoot for 80%. The blitZEN team is currently using 70% for most sprints. When first starting out with the methodology, or when using a new development platform, it can be beneficial to decrease your velocity to something like 50%, and work up to the 80% high water mark over time.



Iterations: Planning Example

- We’ve got a 4 developer core team, and one contractor.
- We’re using 04/18 – 04/29, 2011 as our timeframe.
- We’ve got one holiday, the 23rd.
- Our testing penalty is 10%.
- Our velocity will be 70%.



Iterations: Planning Example

Developer availability breaks down as follows:

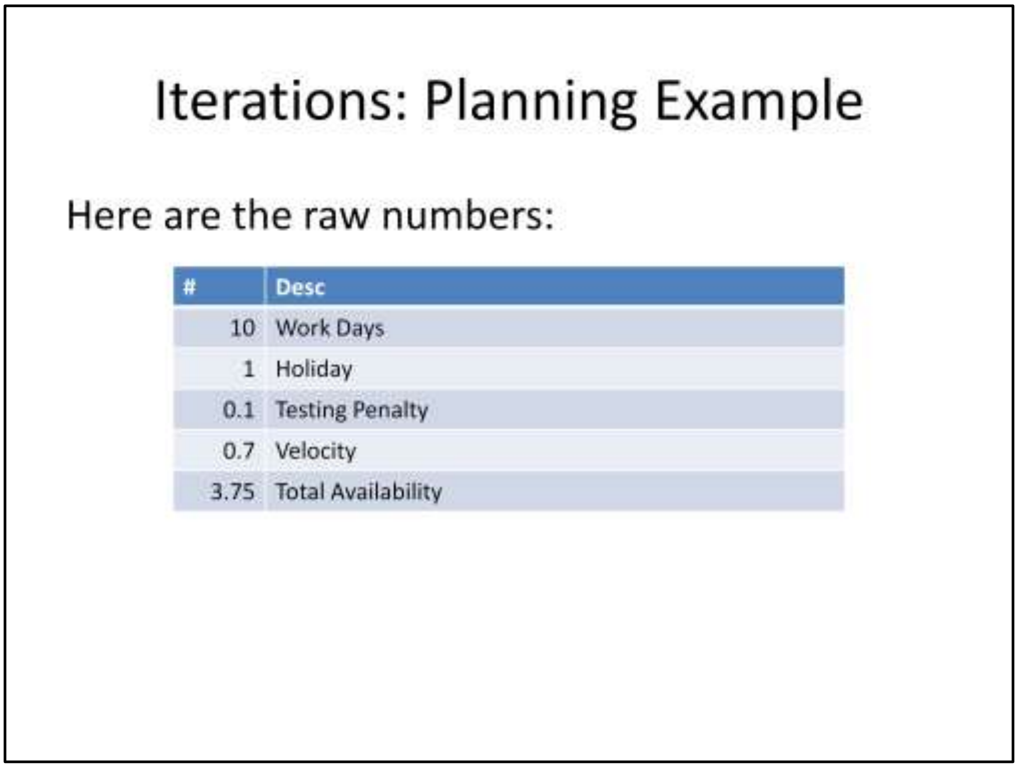
- John – 50%
- Paul – 100%
- Ringo – 100%
- George – 100%
- Billy P. – 25%

This gives us a total developer availability of **3.75**.

We express the availability in terms of whole developers: 0.5 + 1.0 + 1.0 + 1.0 + 0.25 = 3.75

John has recently gotten married to a well known conceptual artist, and isn’t as involved as he used to be.

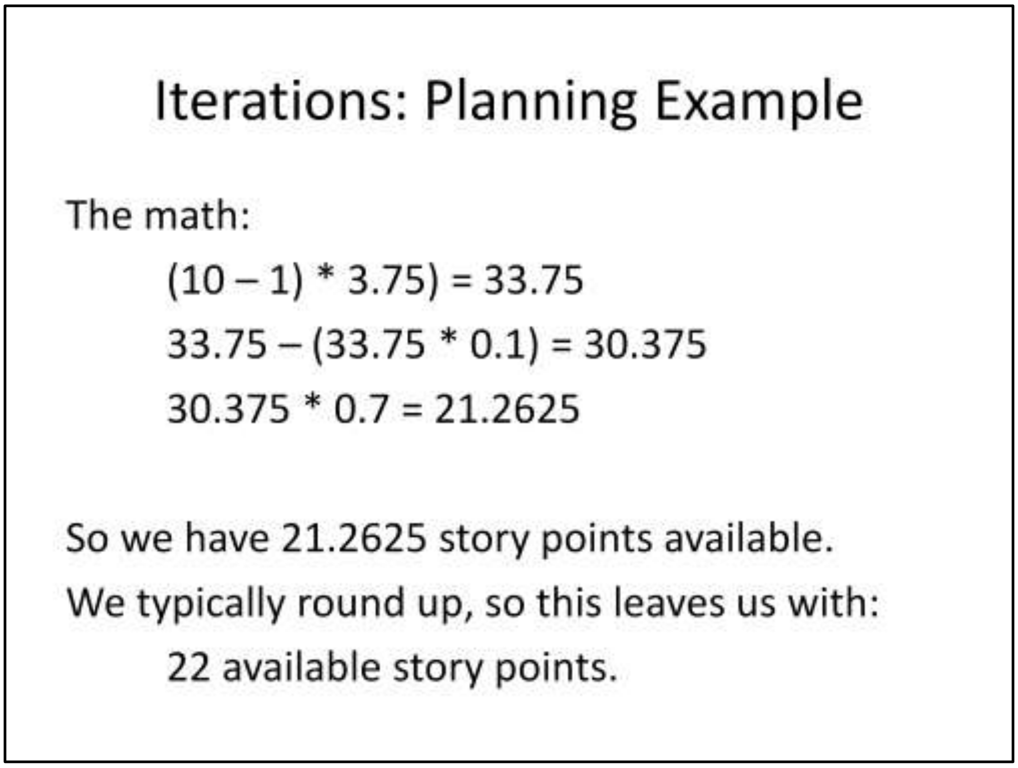
Billy P. is a contractor from California that the group has worked with off and on for years.



Iterations: Planning Example

Here are the raw numbers:

#	Desc
10	Work Days
1	Holiday
0.1	Testing Penalty
0.7	Velocity
3.75	Total Availability



Iterations: Planning Example

The math:

(10 – 1) * 3.75) = 33.75

33.75 – (33.75 * 0.1) = 30.375

30.375 * 0.7 = 21.2625

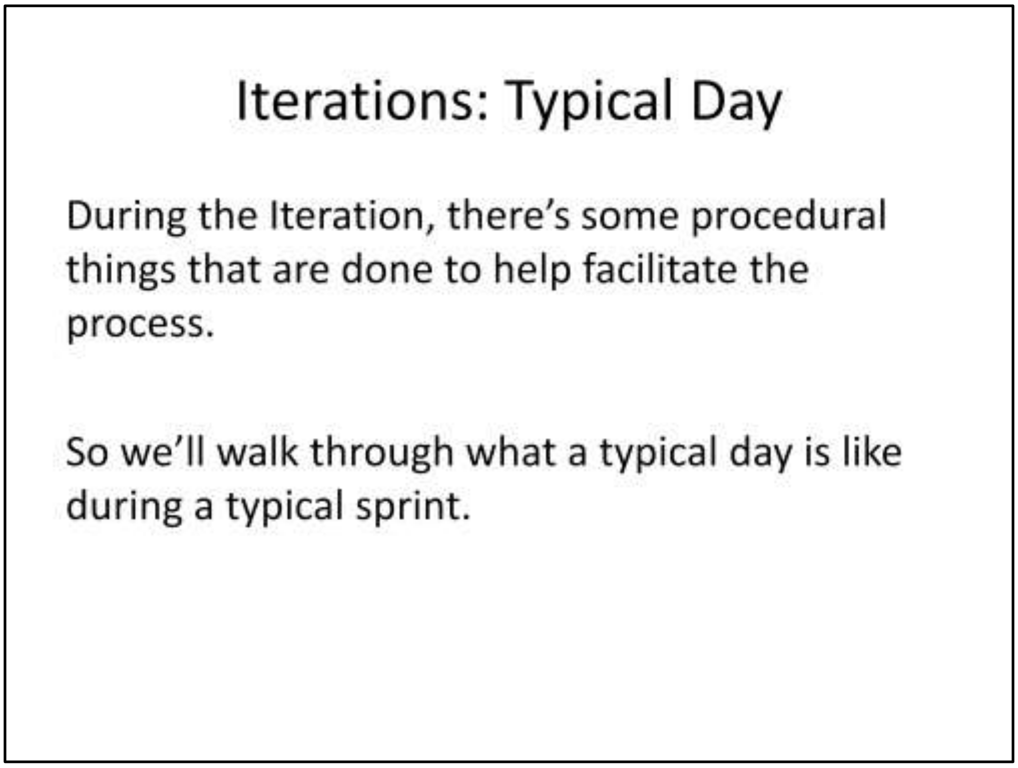
So we have 21.2625 story points available.

We typically round up, so this leaves us with:

22 available story points.

Iterations: Planning Meeting

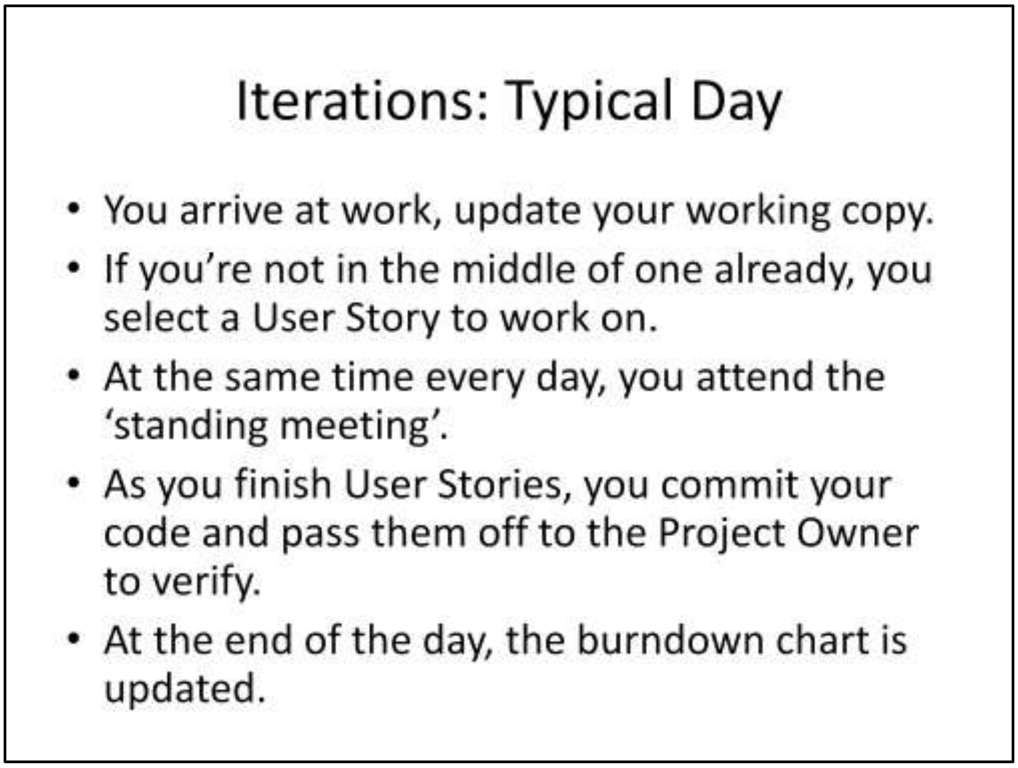
- SCRUM prescribes a planning meeting before each iteration.
- Everyone attends, even folks that weren't involved in the other phases of the process.
- Discuss the timeline.
- Go over User Stories that will be worked on during the sprint.



Iterations: Typical Day

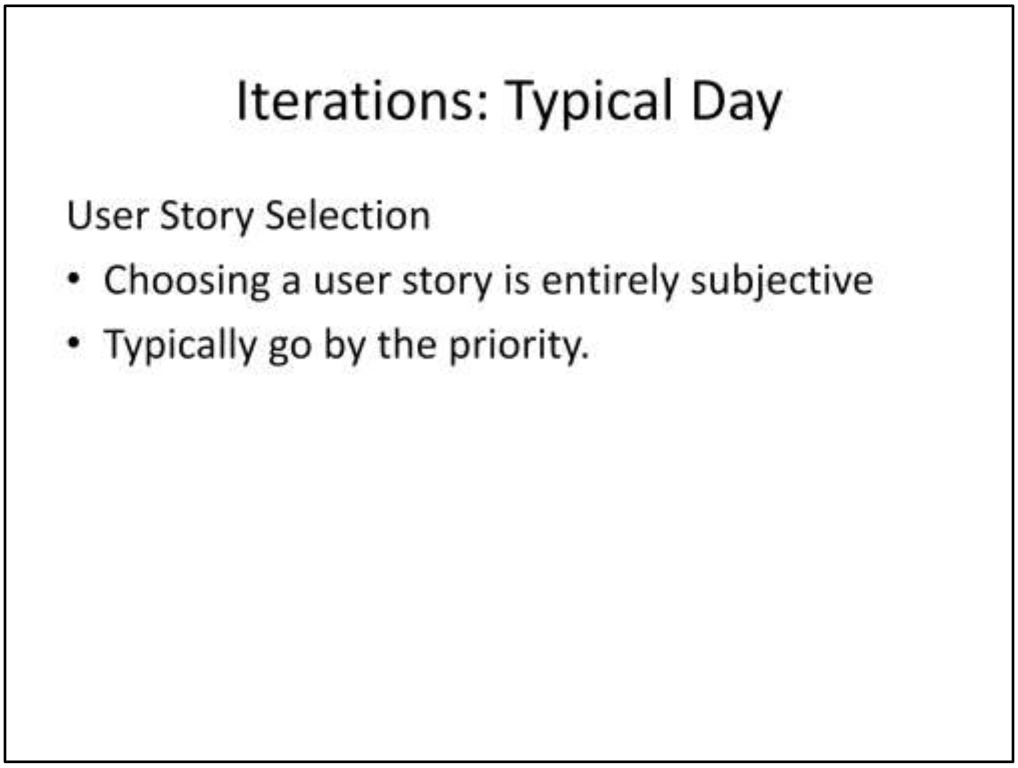
During the Iteration, there's some procedural things that are done to help facilitate the process.

So we'll walk through what a typical day is like during a typical sprint.


Iterations: Typical Day

- You arrive at work, update your working copy.
- If you're not in the middle of one already, you select a User Story to work on.
- At the same time every day, you attend the 'standing meeting'.
- As you finish User Stories, you commit your code and pass them off to the Project Owner to verify.
- At the end of the day, the burndown chart is updated.

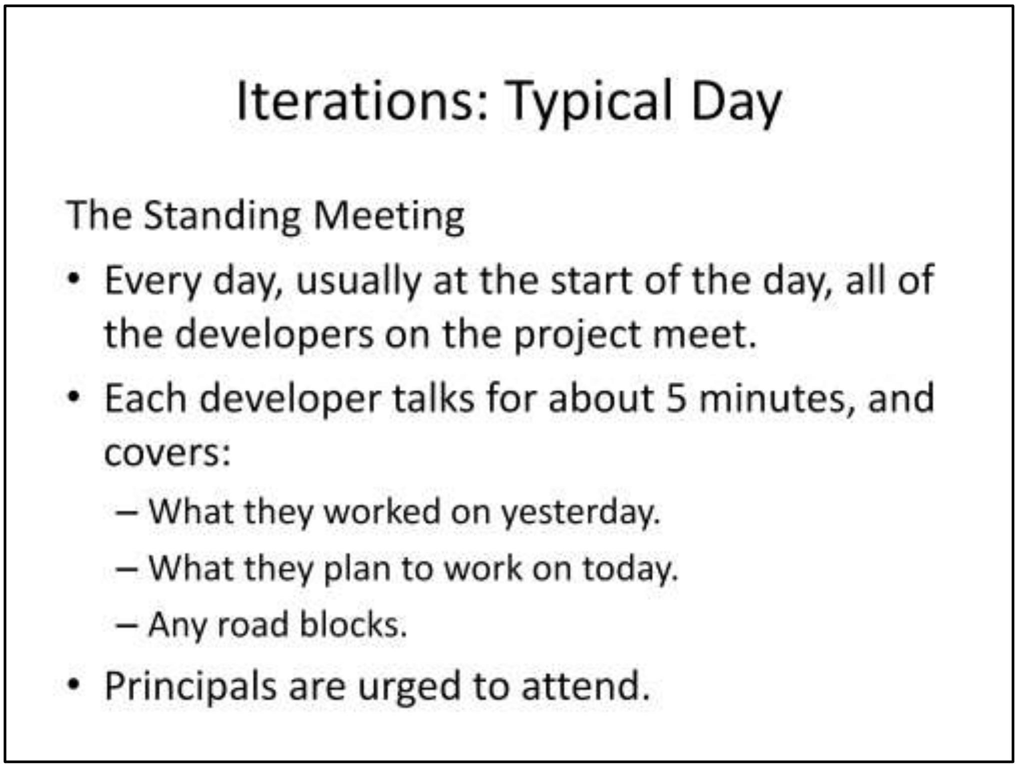
These points will each be expanded upon in the next few slides.



Iterations: Typical Day

User Story Selection

- Choosing a user story is entirely subjective
- Typically go by the priority.



Iterations: Typical Day

The Standing Meeting

- Every day, usually at the start of the day, all of the developers on the project meet.
- Each developer talks for about 5 minutes, and covers:
  - What they worked on yesterday.
  - What they plan to work on today.
  - Any road blocks.
- Principals are urged to attend.

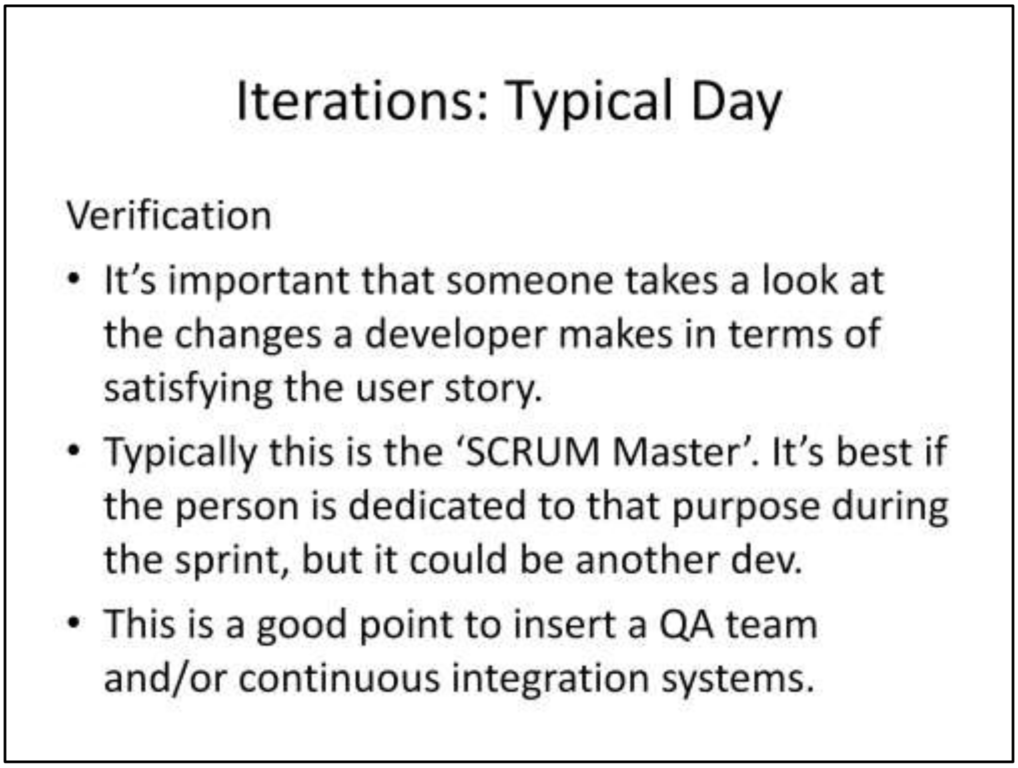
It’s called a ‘standing meeting’ because you’re not supposed to sit down. It’s supposed to be very brief, and not distract the development team from their work.

We’ve done standing meetings over IRC, with really good results.

Point 1: we’re sort of all over the place when it comes to typical start times, so we typically meet at 10 AM. It gives the early birds a chance to do something constructive before the meeting.

Point 2: It’s really important to limit how much time you waste here. With a larger team (5-7), this could mean taking up a lot of time. Egg timers can be helpful here. The three points everyone should cover are useful to help keep the conversation on track. Just bear in mind it’s an oral report of what’s going on, tailored to the potential non-technical audience. Road blocks should be noted and addressed after the meeting.

Point 3: this is a huge deal, very beneficial to the process. The talk tends to be somewhat technical, and there’s always the fear of the principal creeping the scope, but it’s really helpful for both the developers and the principals if they all know exactly what’s going on and who’s doing what. Remember *transparency*!



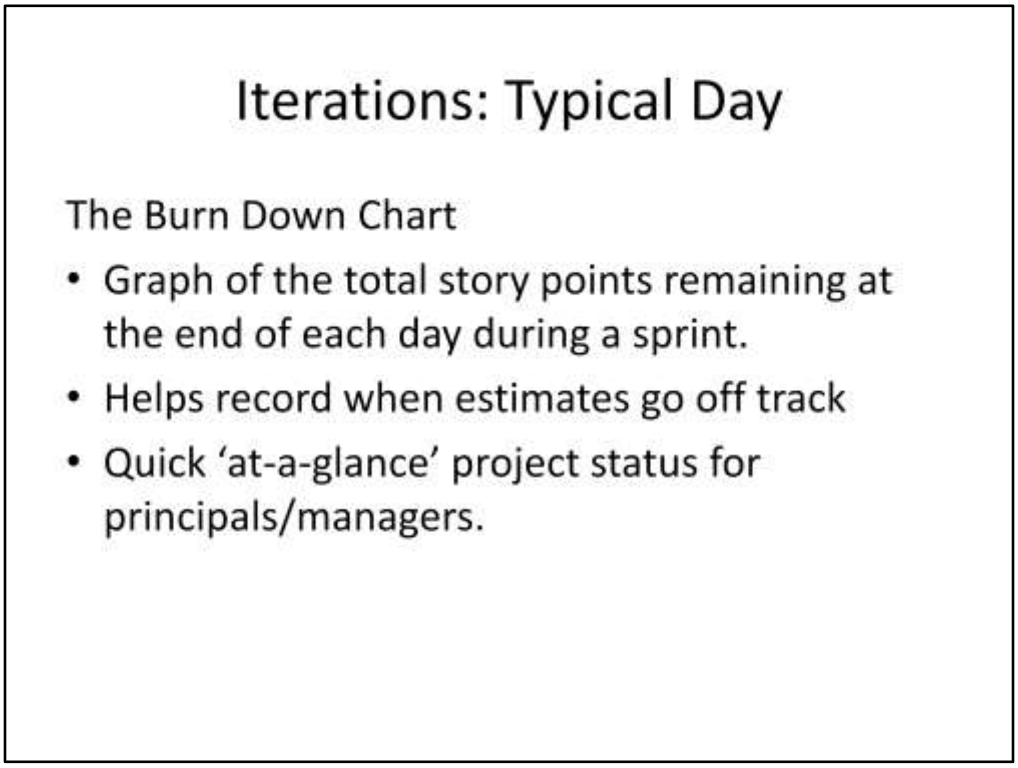
Iterations: Typical Day

Verification

- It's important that someone takes a look at the changes a developer makes in terms of satisfying the user story.
- Typically this is the 'SCRUM Master'. It's best if the person is dedicated to that purpose during the sprint, but it could be another dev.
- This is a good point to insert a QA team and/or continuous integration systems.

Point 1: functional tests can help here.

Point 2: this could also be the Project Owner, or done in a two-phase fashion, depending on that persons' level of involvement.



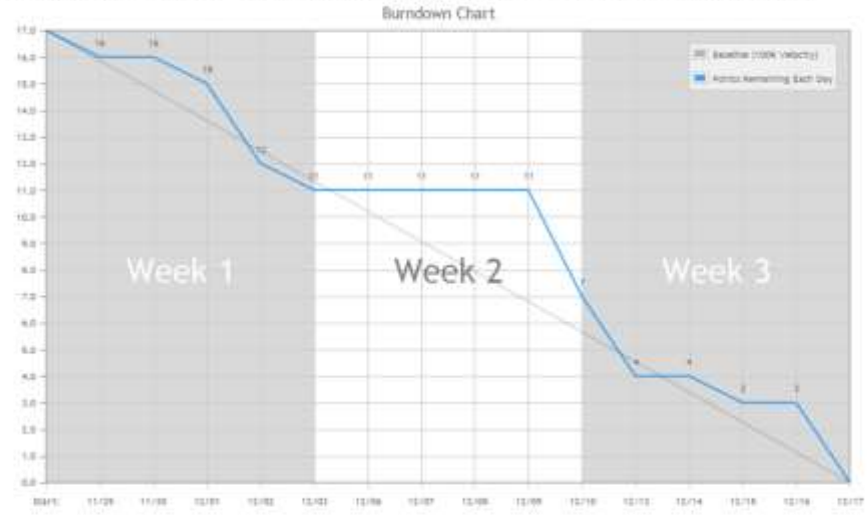
Iterations: Typical Day

The Burn Down Chart

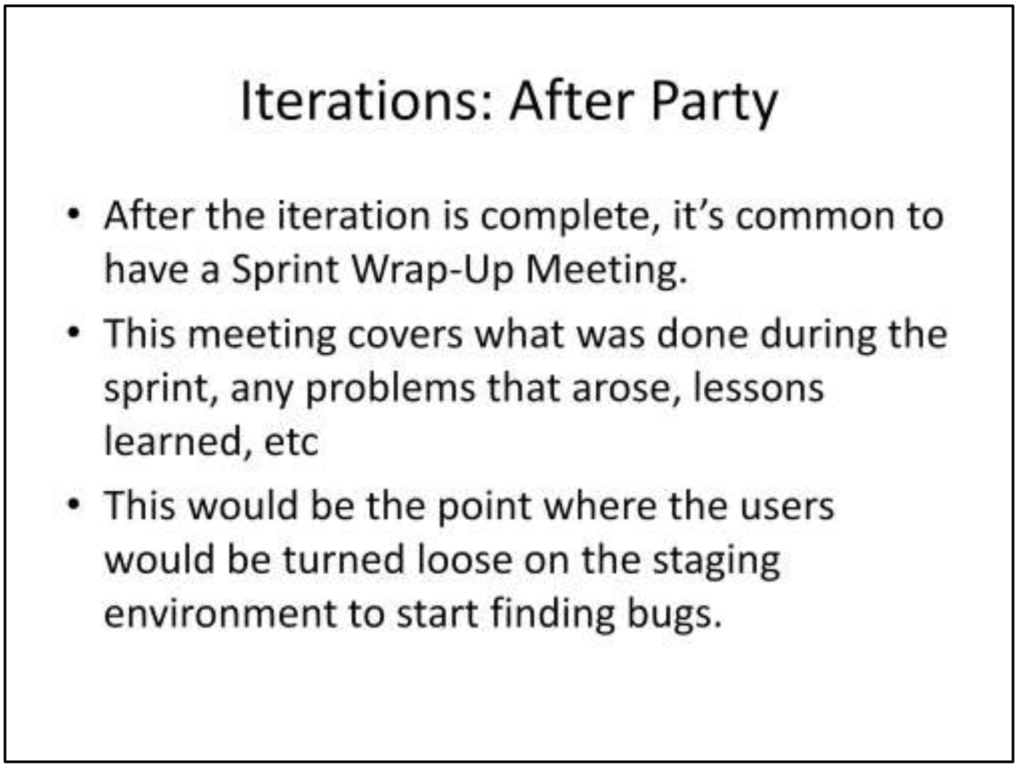
- Graph of the total story points remaining at the end of each day during a sprint.
- Helps record when estimates go off track
- Quick ‘at-a-glance’ project status for principals/managers.

Iterations: Typical Day

Example B/D chart from blitzen.management

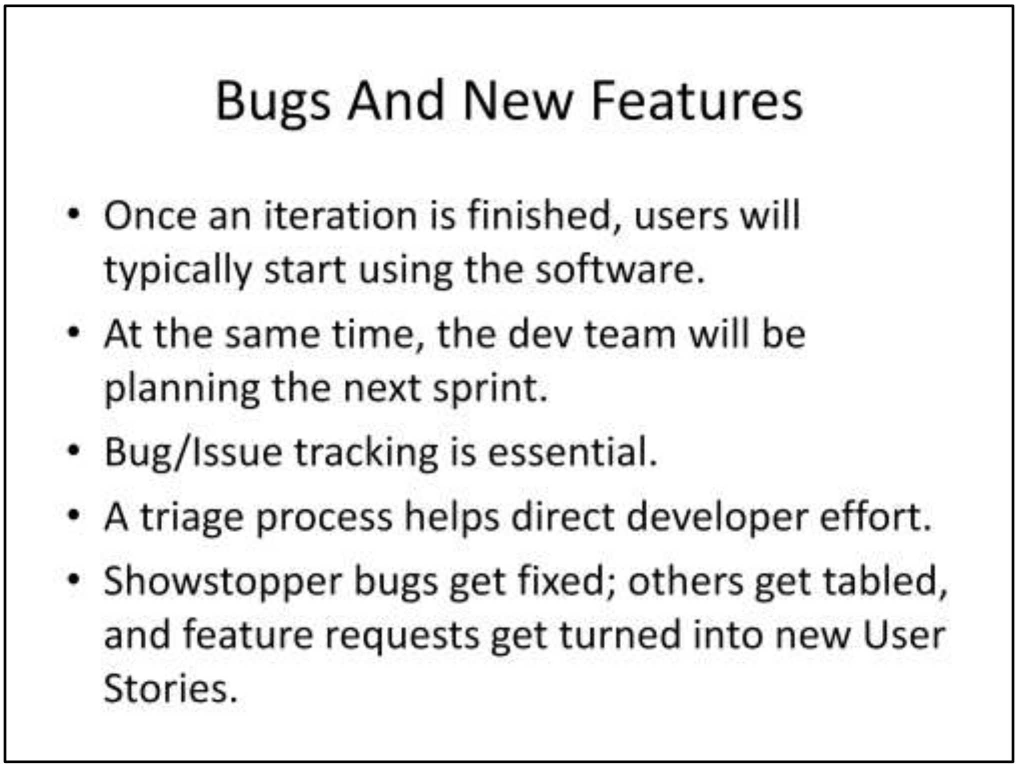


This was a 3-week sprint. We had 17 story points available, and although we got hung up in the second week, we got all of the user stories completed by the end of the sprint.


Iterations: After Party

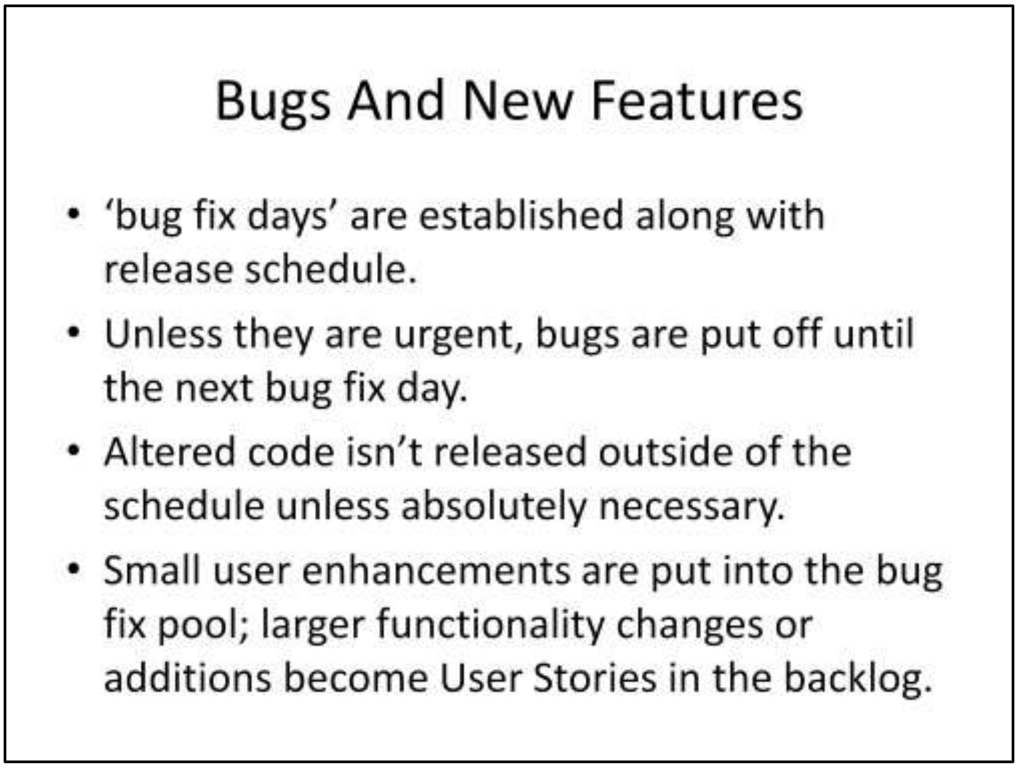
- After the iteration is complete, it's common to have a Sprint Wrap-Up Meeting.
- This meeting covers what was done during the sprint, any problems that arose, lessons learned, etc
- This would be the point where the users would be turned loose on the staging environment to start finding bugs.

SCRUM prescribes that this meeting feature a demo to the principal(s).



Bugs And New Features

- Once an iteration is finished, users will typically start using the software.
- At the same time, the dev team will be planning the next sprint.
- Bug/Issue tracking is essential.
- A triage process helps direct developer effort.
- Showstopper bugs get fixed; others get tabled, and feature requests get turned into new User Stories.

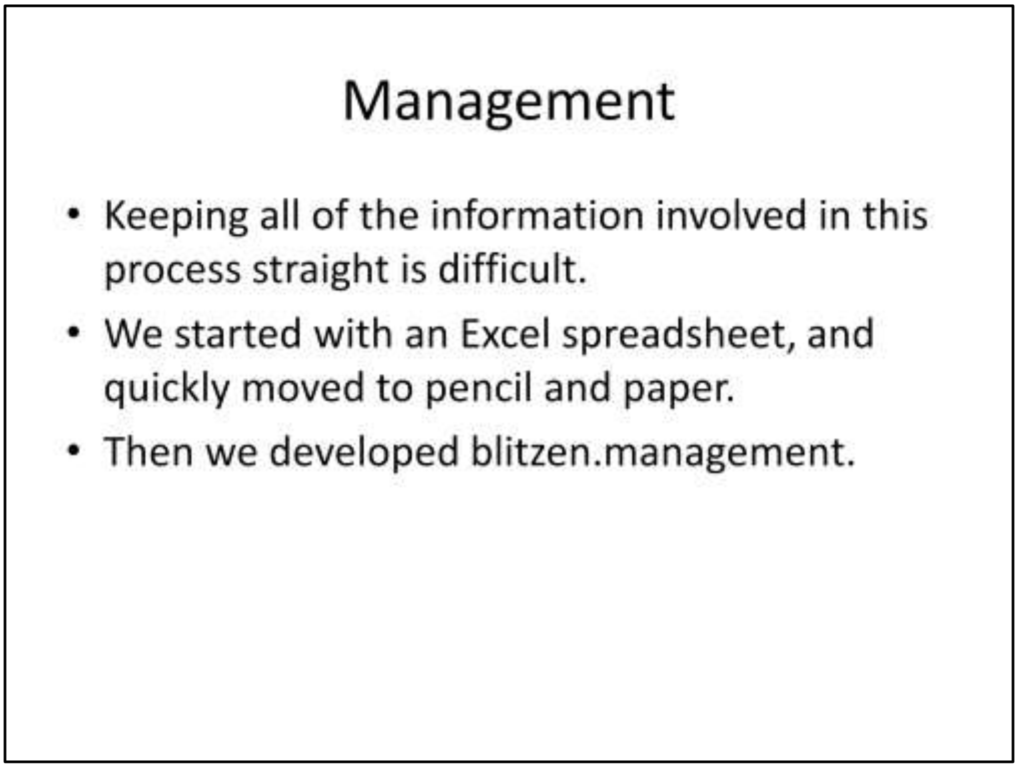


Bugs And New Features

- ‘bug fix days’ are established along with release schedule.
- Unless they are urgent, bugs are put off until the next bug fix day.
- Altered code isn’t released outside of the schedule unless absolutely necessary.
- Small user enhancements are put into the bug fix pool; larger functionality changes or additions become User Stories in the backlog.

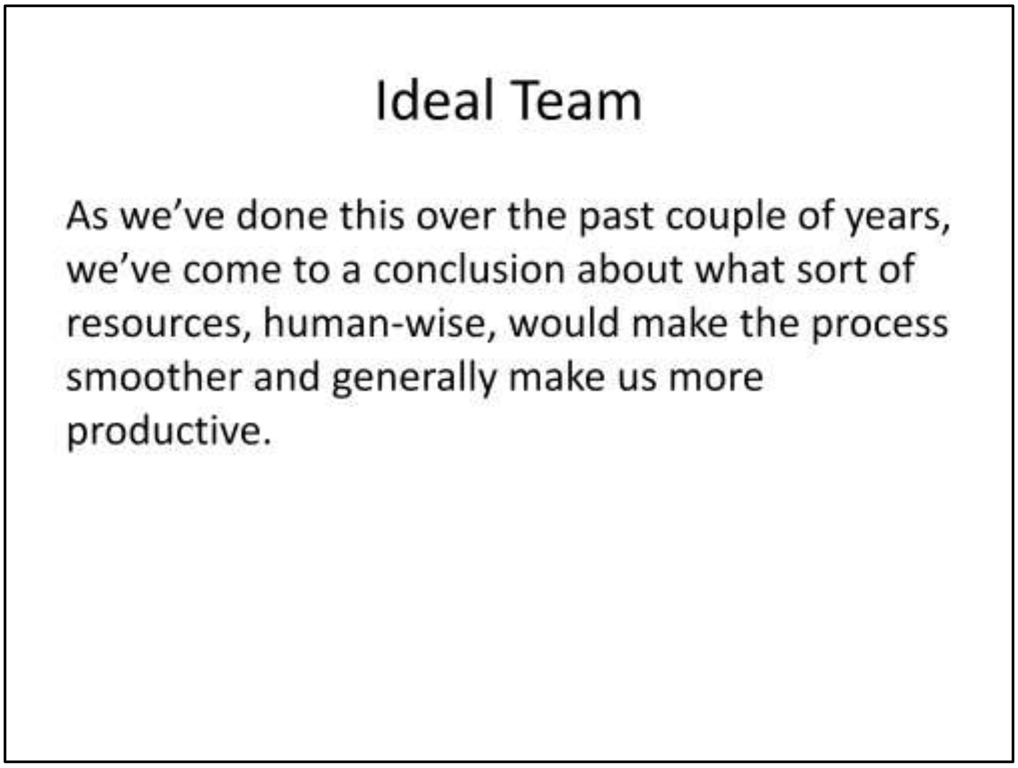
Bugs And New Features

- As the backlog is consumed and bugs are fixed, sprint planning happens again to estimate and prioritize the new User Stories.



Management

- Keeping all of the information involved in this process straight is difficult.
- We started with an Excel spreadsheet, and quickly moved to pencil and paper.
- Then we developed blitzen.management.



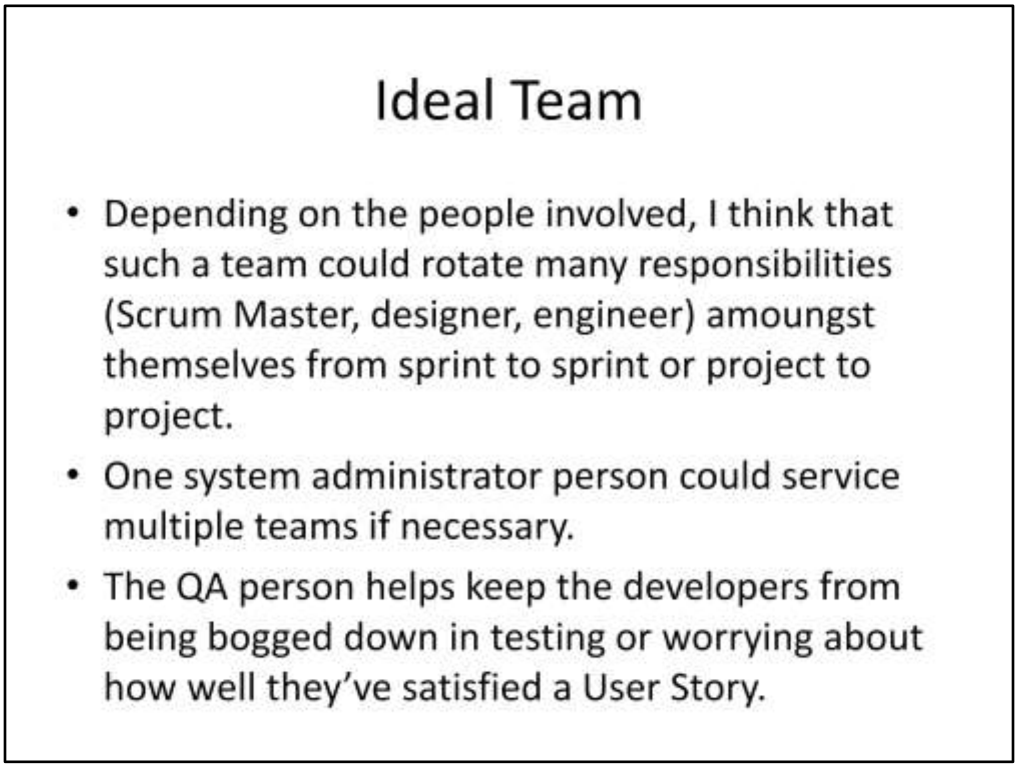
Ideal Team

As we've done this over the past couple of years, we've come to a conclusion about what sort of resources, human-wise, would make the process smoother and generally make us more productive.

Ideal Team

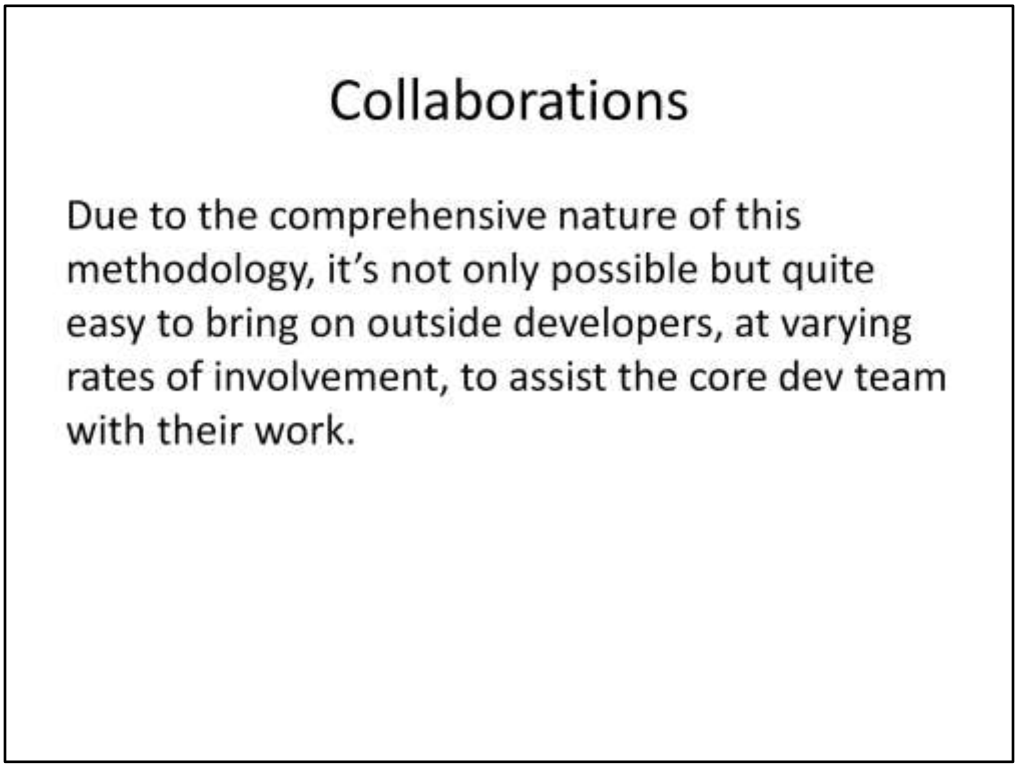
I think a 5 person team would be ideal, broken down as such:

- 1 systems administrator
  – handles buildouts, CI systems, VCS
- 3 developers
  – At least one senior, lead developer.
- 1 QA person
  – Responsible for testing, triaging tickets, verifying work as it's completed.



Ideal Team

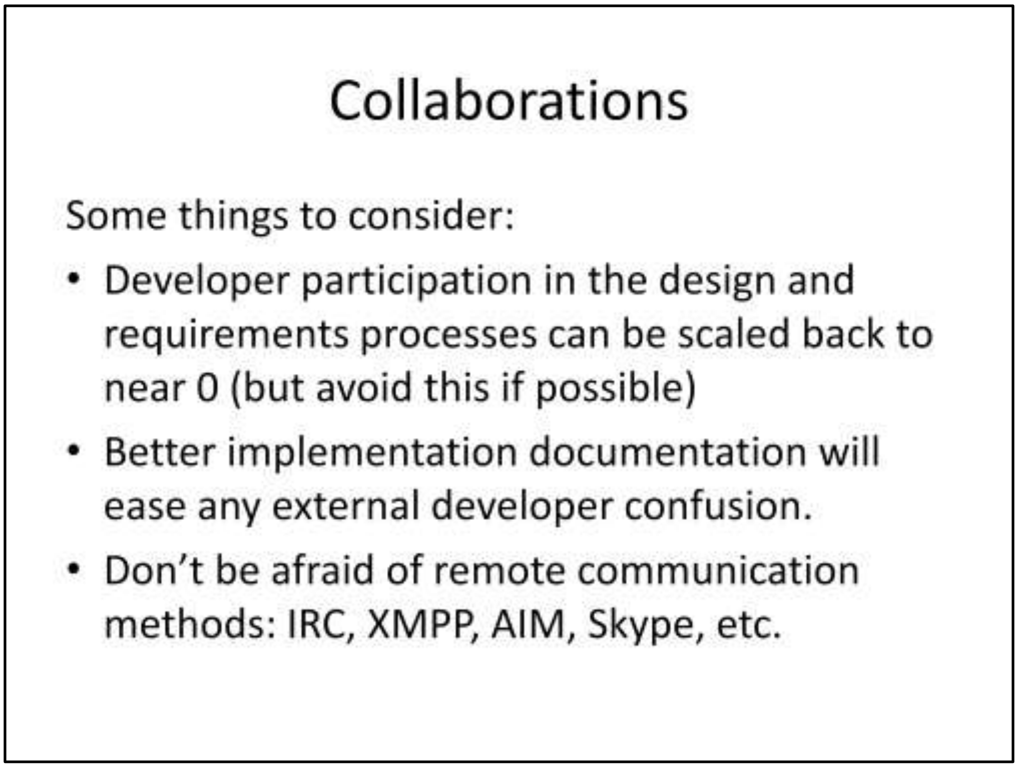
- Depending on the people involved, I think that such a team could rotate many responsibilities (Scrum Master, designer, engineer) amoungst themselves from sprint to sprint or project to project.
- One system administrator person could service multiple teams if necessary.
- The QA person helps keep the developers from being bogged down in testing or worrying about how well they've satisfied a User Story.



Collaborations

Due to the comprehensive nature of this methodology, it's not only possible but quite easy to bring on outside developers, at varying rates of involvement, to assist the core dev team with their work.

We've actually done this with great success.



Collaborations

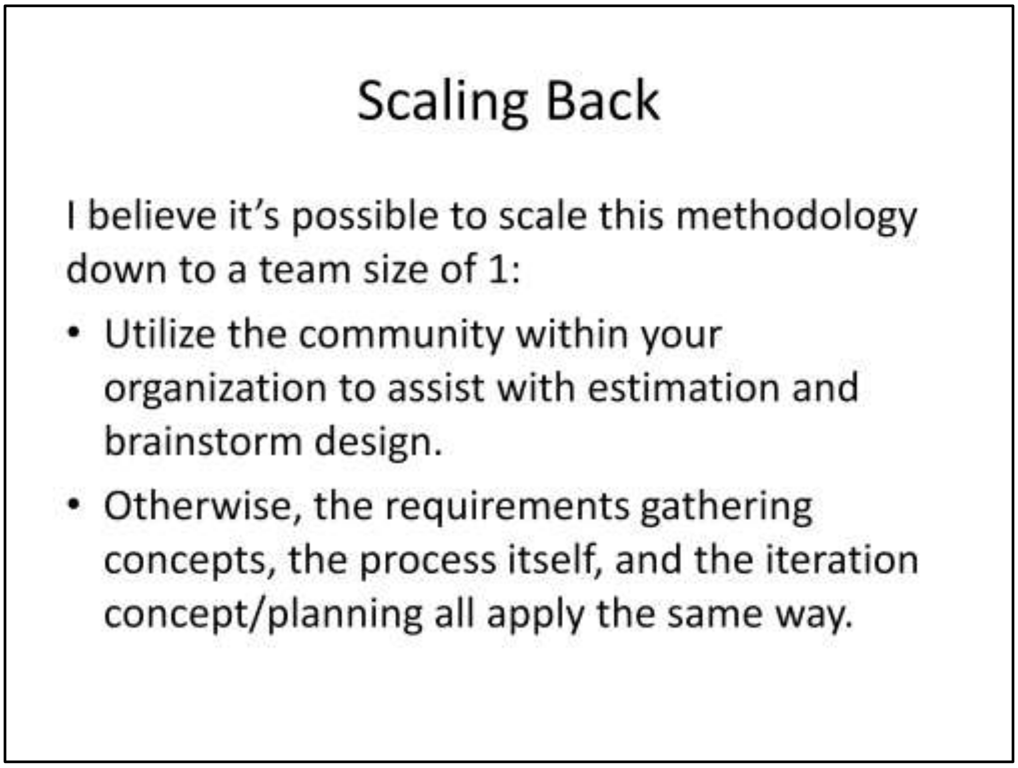
Some things to consider:

- Developer participation in the design and requirements processes can be scaled back to near 0 (but avoid this if possible)
- Better implementation documentation will ease any external developer confusion.
- Don't be afraid of remote communication methods: IRC, XMPP, AIM, Skype, etc.

Point 1: it's really best to have everyone involved in at least the design process, but it's OK if not.

Point 2: This helps alleviate problems that can arise from developers not being involved in the planning process.

Point 3: in some ways, this can make the sprint or the planning go smoother. People can tend to chat while they're doing other things, verses attempts to find a good time for disparate groups to get together in one place. When we did standing meetings over IRC they took less time, and were generally less of a disruption. The ability to log the conversation easily is also a nice advantage.

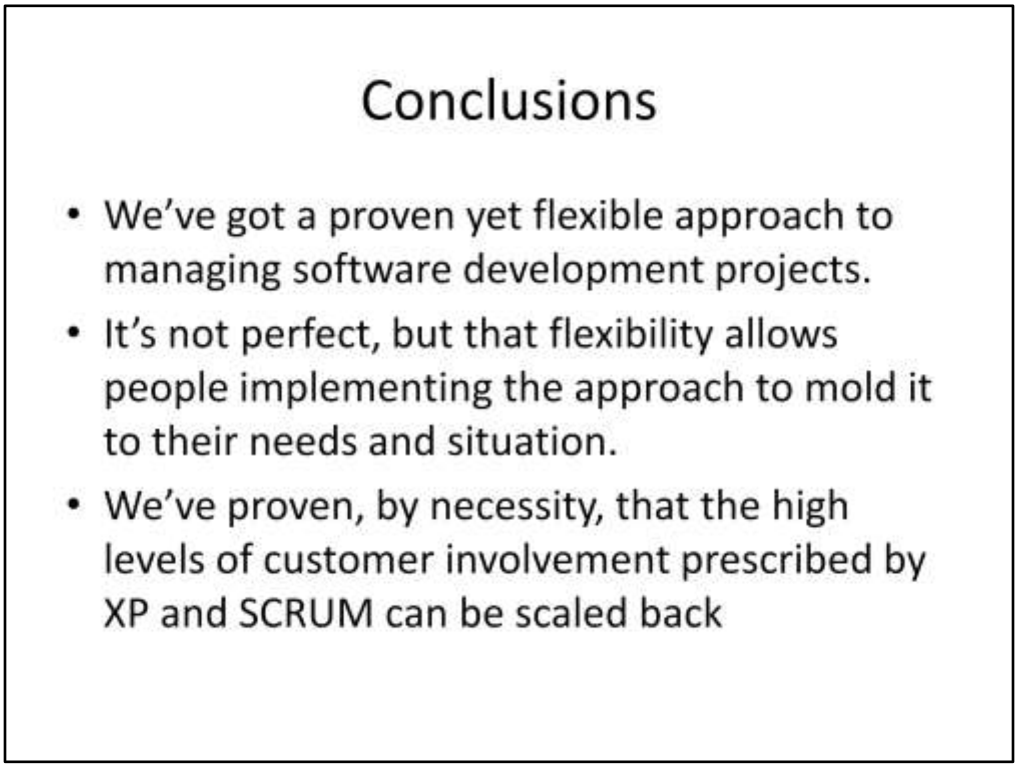


Scaling Back

I believe it's possible to scale this methodology down to a team size of 1:

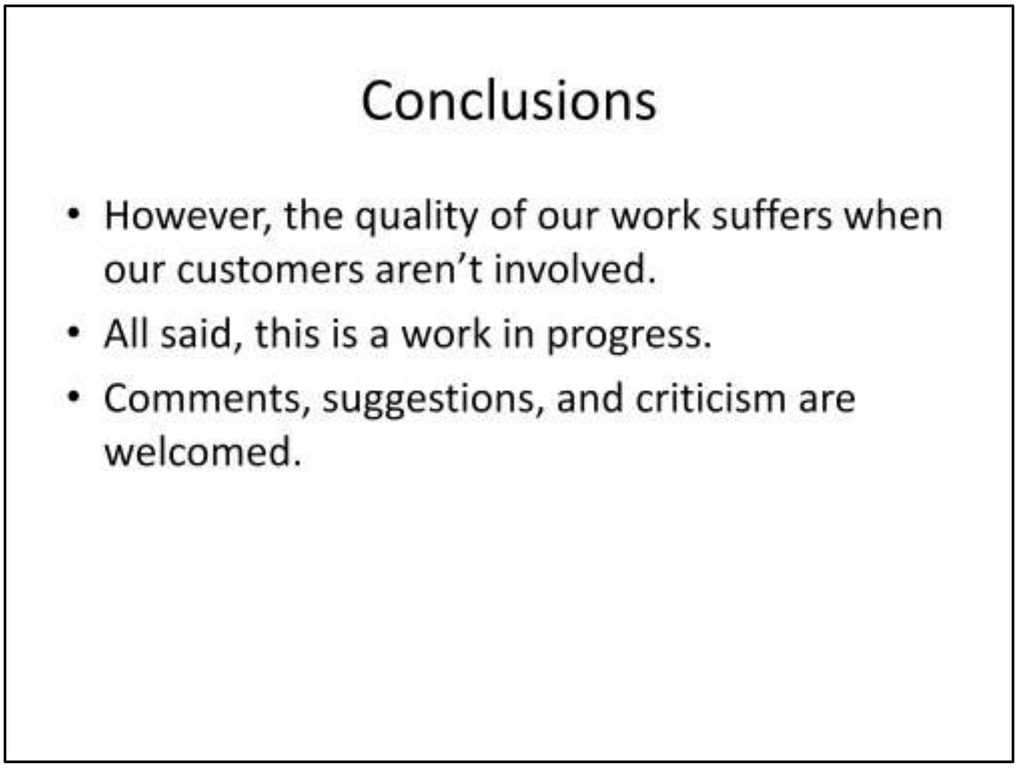
- Utilize the community within your organization to assist with estimation and brainstorm design.
- Otherwise, the requirements gathering concepts, the process itself, and the iteration concept/planning all apply the same way.

I haven't tried this yet.



Conclusions

- We've got a proven yet flexible approach to managing software development projects.
- It's not perfect, but that flexibility allows people implementing the approach to mold it to their needs and situation.
- We've proven, by necessity, that the high levels of customer involvement prescribed by XP and SCRUM can be scaled back



Conclusions

- However, the quality of our work suffers when our customers aren’t involved.
- All said, this is a work in progress.
- Comments, suggestions, and criticism are welcomed.