The Agile Customer's Toolkit

Tom Poppendieck

Introduction

In the past, the customer was asked to define requirements, and then told to go away for months while the technical people developed a solution. The focus of customer activity then needed to be on getting everything right up front because there would be no second chance.

This led to lots of over specification and waste.

Numerous studies have attributed high failure rates of software projects to lack of involvement by the business people commissioning it. Agile approaches recognize that the customer needs to be a business developer and expect the customer side to make all decisions impacting business value just as software developers are accountable for technical decisions. Agile methodologies bring the customer and the software developer into close, ongoing contact so that they can learn together

- ✓ What is the simplest way to address the customers business problem
- ✓ How best to deliver what is needed.
- ✓ How to deal with change over time.

This takes effective collaboration through rich bidirectional communication and feedback. This communication happens mostly at the level of conversation but there is need for a carefully selected set of practices to accommodate the myriad of details which arise.

Toolkit context

Most agile methodologies were articulated by software developers and they have proven very effective for rapidly delivering high quality code that is flexible and does what the customer needs as the customer understands their needs at the **end** of a project. High ceremony, sequential methodologies, on the other hand tend to be embraced by those who do not understand code but feel a need to see progress at things they think they do understand, mostly documents. Agile methods try to deliver frequent increments of working software to these folks, something they can understand by actually using it, rather than investing in detailed, comprehensive documents that deliver little value. Agile methods have proven capable of very rapidly delivery of very high quality flexible code. However, they have not devoted much energy to helping the customer side of the process work out what they need.

The thinking tools I am recommending for the agile customer toolkit are not new. They have been known and advocated by leading thinkers for years. Most of the tools are agnostic about process context, fitting as well in a sequential approach as in an agile approach.

XP introduced one new tool, the idea of chunking requirements and planning into user stories. The agile customer's toolkit contains customer practices to find and manage user stories covering features, behavior, and interfaces. I assume the project is using Scrum or XP planning practices and assuming XP style development practices.

Tool types

The thrust of this talk is to show how to apply these ideas in an agile context as defined by the principles of lean thinking and exemplified by extreme programming.

This paper discusses ten tools for agile customers in four categories.

- ✓ Decision tools are the foundation principles that guide application of all the other tools. They state presumptions about the nature of complex undertakings.
  1. Lean software development principles[1]
  2. Concurrent development
- ✓ Role tools describe how to organize and motivate the work of the project team and to staff it with the necessary people
  3. Team role structuring
  4. Chartering[2,3]

- ✓ **Interface tools** explain how the various roles on the team organize their work and communicate with each other
  5. XP customer and joint practices
  6. User story based interaction with developers [4,5]
- ✓ **Story and customer test tools**[6] are for managing with the content of the project itself and producing the software itself.
  7. Ubiquitous domain language[7]
  8. Role model – context, capabilities, conceptual models
  9. Essential task model – intent / responsibility[8]
  10. Interface content / navigation /behavior model

Decision Tools

Agile is a mindset, not a set of practices, rules, or tools. Agile focuses on people, value, and flow to deliver customer value and minimize waste. Agile principles guide team members in deciding what tools to pick for each task they face and in deciding how to apply each tool. This paper is a short catalog of tools that belong in an agile customer's toolkit and includes a brief discussion of how each might be important in an agile software development project.

Values, principles and practices

The practices people choose to employ on a project, either individually or collaboratively, reflect both their skill set and the principles and values that guide their application. By principles, I mean guiding ideas or rules for deciding. Practices are specific things people do. The overall effectiveness of an organization depends at least as much on its values and principles as it does on its skills at any particular practices. What we need to understand is which values and principles are most effective to the customer to ensure that a software project delivers business value.

Tool 1: Lean principles

Lean principles were first developed in manufacturing, and then applied to product development, logistics, and even to construction. There they frequently generate factor of two improvements in both productivity and quality. In June of 2003, Mary Poppendieck and I published, *Lean Software Development, an Agile Toolkit* which presents lean software development and management practices as 22 thinking tools organized around the 7 principles of lean thinking.

1. Eliminate waste – only add value, not inventory.

2. <u>Amplify learning</u> – iterate.
3. <u>Decide as late as possible</u> – defer commitment.
4. <u>Deliver as fast as possible</u> – Pull value and eliminate delay.
5. <u>Empower the team</u> – train, trust, and lead.
6. <u>Build integrity in</u> – both customer perceived and conceptual.
7. <u>See the whole</u> – avoid sub-optimizing.

These principles are proven to foster effective collaboration among many disciplines in complex situations. All published agile methodologies are consistent with these principles. These principles are simple rules that permit every project participant who understands the mission of the project to make good decisions about what to do next without explicit, detailed direction. They provide a theoretical context to enable effective tailoring to adapt published methodologies to the needs of a particular project context. Mindfulness of these principles makes the customer practices described in this toolkit agile, not the practices themselves.

Tool 2: Concurrent development. (Iterative and Incremental)

It is critical to apply lean ideas at the level of principles rather than practices. If one looks to manufacturing practices for guidance on software practices, one makes SERIOUS MISTAKES!

When lean principles are applied to product development, they generate an approach (Figure 1) known as concurrent development. Concurrent development of the mission, the requirements, and the implementation permits the team to *amplify learning*, discovering from frequent feedback what the impact of each decision will be and to adjust. Concurrent development permits the team to *decide late* when it has the best understanding it is going to get to achieve the best fitness for use it can within the constraints of the project. The fundamental responsibility of management and project leadership is to staff the team and shape the work to make inevitable ongoing rich, two-way upstream / downstream communication and involvement. At a minimum, this means reliable, daily contact, at best full collocation so discussion and negotiation of directions can occur at any time.

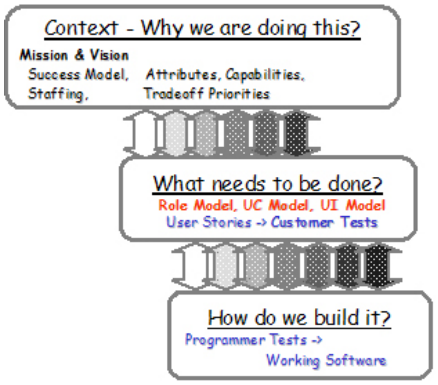

Context - Why we are doing this?
Mission & Vision
Success Model, Attributes, Capabilities,
Staffing, Tradeoff Priorities
What needs to be done?
Role Model, UC Model, UI Model
User Stories -> Customer Tests
How do we build it?
Programmer Tests ->
Working Software

Figure 1 – Concurrent Development #

Role Tools

Developing software is too complex to be undertaken using a static recipe. Most projects need people with diverse skills, experience and knowledge able to think and adapt their practices to achieve their project goals. The interplay among capability, responsibility and collaboration is what makes it possible for complex projects to succeed.

Boundaries and emergence.

The study of complex adaptive systems teaches that adaptive behavior will emerge only if the system has effective boundaries. Project success depends on proper attention to four critical boundaries:

1. The project's charter defines at a high level the boundaries of what is in the project intent and what is the context the project must accommodate.
2. The charter also sets boundaries on the resources committed[3] to achieve the project mission including staff time, funding, and completion timeframe.
3. The staff allocation boundary should include members with all the skills, knowledge, and decision authority required to execute the project.
4. The communication boundary must extend to the entire project community[9]. The project community needs to include all the people who influence or are affected by the project in addition to specifically allocated team members.

Tool 3: A three legged stool.

The whole team is responsible for achieving the project mission specified in the charter. While everyone with something to contribute can participate in discussion of any topic, align authority to decide an issue in the end on competence. Three categories of roles (Figure 2) can be distinguished within the project team.

1. Some team members need to be able to establish and maintain the project mission and context, to provide resources and leadership, to remove barriers, and to communicate with parts of the project community outside the team. These are manager roles.
2. Some team members need to determine specifically what needs to be done to deliver the business value the project is chartered to deliver. These are customer roles. Team members in cus-

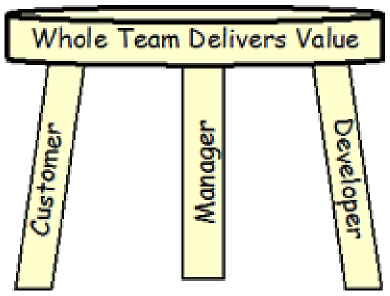


Figure 2 - Three Legged Stool

tomer roles must have the business expertise and authority to decide feature details and priorities.

3. Some members of the team need to actually delivery the software that does what is determined to provide value. These are **developer** roles. People playing developer roles have the competence and responsibility to make all technical decisions about how to best implement the features.

All three roles are necessary to deliver project value. The number of people will vary with the skills an experience of each team member and the nature of the project. (Depending on the size of the team, their domain experience, and the nature of the project, team members may fill roles in more than one category.)

Responsibilities.

Each leg of our stool is vital to project success. Agile methodologies mostly focus their advice on practices for the developer leg and on the seat that is composed of joint practices describing how the whole team works together. The responsibilities of the customer side are clear and numerous but supporting practices are not provided and this has caused more than one agile project to fail to accomplish its mission. This articulation of responsibilities is in reaction to the inadequate or ineffective participation of business/customer side which has repeatedly been identified as a primary cause of software project failures.

An agile manager will not direct the work but will remove all impediments to the teams doing the work and will either lead the team or support a leader who is on the team. A key to managing an agile project is to set it up so the whole team can really work together effectively. Lean principles require eliminating structural waste and maximizing opportunity for feedback.

- ✓ The whole team sits together for the duration of the project so they can plan, communicate and adapt as they learn what works for them for this project. – *amplify learning* and *eliminate the waste of waiting*
- ✓ The whole team succeeds or fails as a team, not as individuals. Measuring or rewarding individuals will destroy collaboration. – *See the whole, don't sub-optimize.* The members of the team will apply their respective technical or business skills and practices from their disciplines to achieve the purpose of the team and learn from each other.

Tool 4: Chartering

A software development project is the work of a community contributing a variety of skills and having interests in the outcome. The community includes everyone on the team as well as others who can affect or are affected by the project.[9] The interests of all members need to be aligned and kept aligned through a shared understanding of the project purpose.

Every member of the team must continually assess the state of the project and how they each can most effectively contribute to its progress if they are to collaborate effectively together. Each contributor's focus on the shared purpose (Figure 3) will be different depending on their individual capabilities and goals. A contributor's passionate pursuit of the projects purpose depends on their clear understanding of what's in it for them

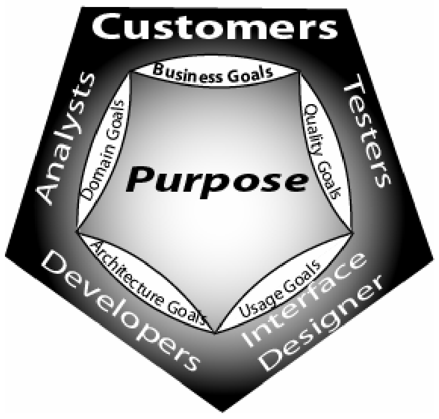


Figure 3 - Team Perspectives

Tradeoff alignment.

A critical boundary dimension is what qualities the final result must display. Typical qualities include, features, completion date, usability, defect level, performance, availability, and cost, either monetary or in terms of dedication of people to this effort rather than some other activity. Some project will have other quality dimensions.

A key decision that will shape the project is which ONE dimension is primary, which is secondary, and which will the authorizing sponsors accept tradeoffs in order to achieve the priority 1 or 2 goals.

All project stakeholders must come to agreement on which project attributes are number 1 and 2 and which have more flexibility. As business context changes, this can be revisited but it must be communicated immediately to the entire team as it will affect how they spend their time. (See Table 1)

Product data sheet.

The tradeoff matrix is only one part of a guiding charter for the team. The team leadership generates a Product Data Sheet (PDS)[2] at the beginning of a project to

- Negotiate boundaries among the stakeholders.

- Identify the people and other resources that must be allocated the team.

The PDS is shared with all members of the team and defines the boundaries within which the team makes tradeoff decisions in the course of the project. To ensure consistent decisions immediately communicate any changes to the PDS to the entire project community.

A PDS must be kept to ONE PAGE. A longer document will not be used or useful. If you think you need more space, you do not understand what you need well enough to proceed. Keep working and thinking until you can describe what success means for this project concisely in a way that all stakeholders agree to.

Table 1 - Product Data Sheet.

Project / Product Leaders:

Authorizing[3] Sponsors:

Project Mission Statement

Clients / Customers

Prioritized Features

Project Context Diagram[3]

Mission Tradeoff Matrix

Quality Dimension	Top Priority	Next Priority	Accept Tradeoffs	Dimension's Target Value
Scope	x			(Features, Perf)
Schedule			x	
Low Defects		x		
Resources			x	

Client Benefits

Committed Resources[3]

Key Issues / Risks

Key Events[3]

Interface Tools

The principles and product data sheet guide project participants in how to decide what to do but they do not provide a template for interaction in pursuit of the project mission. This next batch of tools offers strategies for working together as effectively as possible.

The most fundamental interfacing truth is that face-to-face conversation is by far the most effective way to communicate and collaborate within and among roles because of the instant multimodal feedback available. Face-to-face collaborators may choose to write things down but the meaning of their documents will rest on memory or their conversation.

Tool 5: XP Practices

The developer side of an agile team will adopt some version of the extreme programming development practices. (Figure 5) These practices are very effective together and very satisfying to the developers. They enable the developer side to rapidly produce very high quality flexible code.

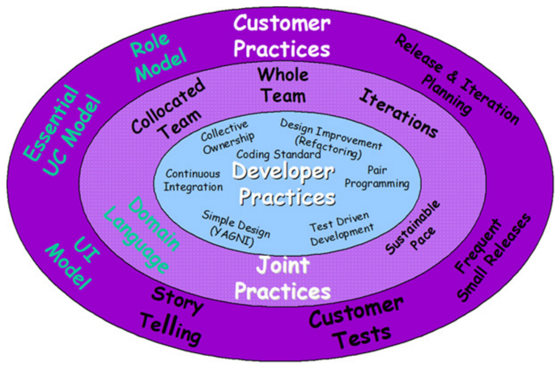


Figure 5 - Extreme Programming Practices

These practices are the engine that determines what is possible for the team to accomplish. They manifest most of the principles of lean thinking in the programming context.

This development engine determines how fast the project is capable of going. The developer side of an XP expects to *deliver as fast as possible* consistent with it's commitment to *build integrity in* and expects customers and managers to do everything possible to avoid slowing them down. This means never making the developers wait for decisions or information. Waiting is a form of waste to be eliminated.

XP specifies 4 responsibilities of the customer side of an agile project team.

1. **Tell stories** – describe what the software must do to provide value. This is primarily conversation.
2. **Define tests** that demonstrate what each story means precisely. These should be automated.
3. **Decide which stories to do next.** The customer defines what is valuable either to directly provide benefit or to *amplify learning* about what will work best.
4. **Work iteratively** and learn as the project progresses.

Entire books have been written about many of the developer side practices - pair programming, test driven development, refactoring, continuous integration & unit testing, planning game, and a book about user stories is in press; no book has more than 11 pages about customer practices.

THAT is what customer side practices are about.

- Work iteratively, breadth first with gradual refinement of story details to provide a gradually narrowing context for decisions.

- Work iteratively DEPTH first to rapidly deliver working software to get rapid feedback from the most important remaining work.
- Develop a common team vocabulary to discuss the project with the developers based on the language of the DOMAIN.
- Develop a task centered organization for the stories you pass to the developers based on user-goal centered use cases
- Develop a UI that will be highly usable for those who using the application.

Tool 6: User Stories

Stories are the primary tool the customer side and the developer side use to organize their communication with each other. They are the final "output" of the customer practices. All the rest of the work the customers do is focused on generating stories and their confirming tests. The use of stories rather than use cases or "system shall…" type requirements for communicating what is needed is a big deal. It permits the developer side to reliably deliver requested features in a predictable time which builds trust and *empowers the team* to control its course.

Customer side team members are responsible for choosing stories to cover everything they care about. During release and iteration planning they need to explain the details well enough to enable the developers to do useful estimates so the project can proceed predictably. During the iteration they in which ask the developers to implement the story they need to explain or work out concepts, business rules, variations, interfaces and exact results expected for each alternative path through each story. All these detailed decisions get documented twice, once in the code written by the developers and once in detailed tests the customer defines to record the decision.

Story anatomy.

User stories have three parts:

1. The Card which is a token naming the topic and a few key facts about it.
2. A Conversation between developers working on estimating or implementing the story and one or more customers with the expertise, judgment, and decision authority to specify or decide the details. This conversation will be fairly general when planning and quite detailed when the story is being coded.
3. Confirmation in the form of **customer tests** that specify exactly what results are expected. These will preferably be automated executable tests which may be prepared by the customer side using a framework such as

FIT or ordinary code written by the developer side of the conversation. In either case, the customer specifies the test cases that prove the story is correctly implemented from the business perspective. Early in the conversation, a few key tests may be outlined on the back of the card, later, detailed thorough tests are needed for everything the customer cares about. These tests amount to executable specification the application is guaranteed to satisfy.

User stories are the primary unit of commitment between the customer side and the developer side. The stories are planning units for developer side work. They are used for planning the order in which things the customers side values will be done.

The story card.

The appropriate amount of story detail fits on a 3x5 inch or 4x6 inch ruled index card. If you run out of room on a card, use a SMALLER card. The idea is to be brief. The card is not the place to capture details. The card is a signal to discuss, organize, or to work on a feature and not a means of recording all the details. If you need more space, you either need to break down the work into smaller pieces or create some sort of supplementary document. A card will have a title to facilitate discussion; one or a few sentences for a brief reminder what is wanted; and an estimate by the developers of the relative effort to implement it. The back of the card is sometimes used to note a few sample customer tests.

Teams have experimented with using various computerized representations of story cards but the consensus among XP teams is that handwritten cards are best, especially during planning activity. They are easy to tear up and rewrite. They are easy to sort into priority order and arrange into stacks by iteration or any other criteria. They are easy for developers to pick up when they begin work on one and post on a board when they are done.

Story size.

XP strives to build trust between the developer side and the customer side by reliably and regularly delivering valuable software. Size matters here.

The value is ensured because the customer steers by choosing which stories to do each iteration. Each story must be small enough that the developers have confidence in their estimates of how long it will take to do. If they cannot estimate, they ask the customer to explain more about what the story covers and possibly break it down into smaller, individually valuable stories that they can estimate and which the customer may prioritize separately.

This combination of trust, predictability, and business value is difficult to achieve. Developer practices enable the developers to do their part. Customer practices and joint practices enable the customers to do theirs.

Story integrity.

A place to start the effort of creating a system perceived to have integrity is to understanding the needs of the people who will be using the system and of the business process the system will support. The business process determines the value and the user characteristics determine the usability issues. The business process usually depends on the system to supply, capture, update, or manipulate certain sets of information subject to applicable business rules and policies. A person able to use the system effectively must know or be able to easily recognize the actions to take to capture or access or modify information as required for the transaction the system is supporting for them.

Usage-centered domain-driven integrity.

The agile strategy to ensure that a project delivers real business value is concurrent development. There are two related aspects to product integrity, conceptual and perceived (Figure 6).

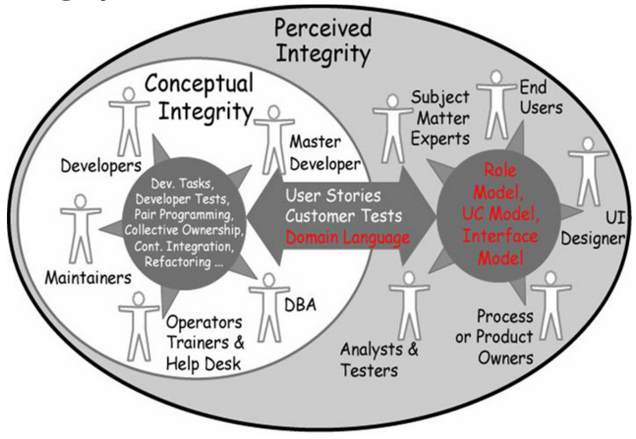

Perceived Integrity
Conceptual Integrity
Developers
Master Developer
Maintainers
Operators Trainers & Help Desk
DBA
Dev. Tasks, Developer Tests, Pair Programming, Collective Ownership, Cont. Integration, Refactoring ...
User Stories
Customer Tests
Domain Language
Subject Matter Experts
End Users
Role Model, UC Model, Interface Model
UI Designer
Analysts & Testers
Process or Product Owners

Figure 6 - Conceptual and Perceived Integrity

Conceptual integrity refers to the internal structural quality, understandable architecture, and code maintainability that the development team builds in to the product by applying XP engineering practices.

Perceived integrity is the responsibility of the whole team and includes:

- Suitability for effective and efficient use by the end users.
- To achieve the customers and product or process owners business goals.
- In a manner that correctly implements the definitions, business rules, and constraints articulated by the subject matter experts.

Perceived integrity arises from rich continuous communication between those who need the system to carry out their responsibilities and those who build the system. This communication can only happen if the two sides share a common language to speak about the system in. This language needs to be based on the language of the user domain but shaped to adequately describe the parts of the system visible to the users. In summary,

- The vocabulary of communication is the domain language.
- The unit of work and planning commitment is the user story.
- The unit of business process value is the use case. Many user stories will be derived from and organized by use cases.
- The unit of specification is the customer test. Customer tests may cover a single story or several stories in the same use case.
- Other stories are derived from the UI content and navigation models which are the means of ensuring usability.

Story Telling Tools

Our first six tools have addressed how the team makes decisions and how they can organize their work to build mutual trust via effective communication. What remains is to discuss practices that help the customer side team members to decide what stories to tell and how to prioritize them. Through discussion, informal modeling, and coding, the whole team develops a domain language to use to tell stories about what will deliver the business value the team is chartered to enable. They use models of the characters, their goals, and the tools and information they will employ to reach those goals to identify their stories. In short, they do usage centered design to find the stories. (Figure 7)

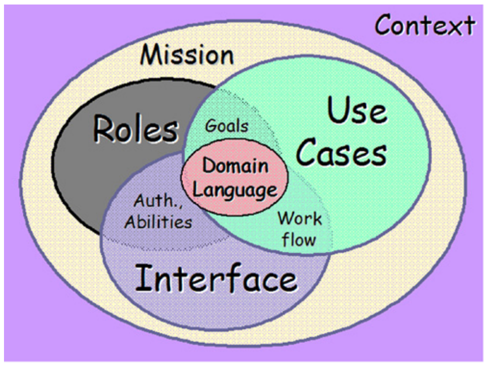


Figure 7 - Realizing the Mission

Story Perspectives.

Achieving integrity is a complex undertaking because it results from the interaction of at least three perspectives (Figure 8):

- People and their roles, context and capabilities,
- Business process goals, policies, and workflow, and
- Domain entities, business rules, policies, and constraints.

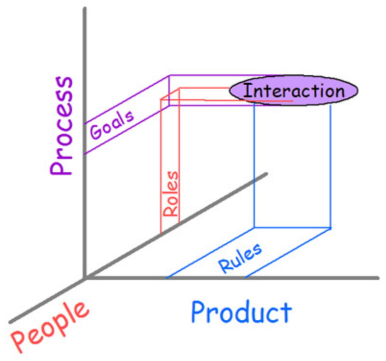


Figure 8 - Interaction Perspectives

User stories need to cover all these perspectives. Start with whichever is most important; usually this is the people. Then iteratively address all three concurrently to refine and detail each perspective as you learn.

Model the roles people play in relation to the system, noting their capabilities, the demands on them, their knowledge level, their likely level of familiarity with the application itself. Use one index card for each role. (This is not a big model!)

Then, for each role, identify the tasks each will use the system for. Write down the goal of each task, one goal per index card with a note as to which role owns the goal. This is the start of a use case model[8] and the origin of many story cards.

Then, for each goal, identify the important concepts that the user is going to expect the system to know about, remember information about, and enforce rules and relationships about and write each concept on an index card. Most concepts will be used by several goals. This is the start of a domain model[7].

Tool 7: Domain Language

A deep shared vocabulary used by the entire the team to discuss its work is the foundation of effective story telling. The language of the domain is the best metaphor for thinking about the application in, especially when discussing it with the customers and other stakeholders. After all, the customer already knows the domain terminology and domain concepts need to be represented by the application for the user to manipulate. Policy and business rules will be described in terms of domain terms.

Every team needs to create a common shared domain language that everyone on the team uses with the same meaning and expectations as an agreed upon way to structure their shared knowledge. This language simultaneously expresses the developers deepening knowledge of the domain and the customers growing knowledge of the solution they will use to create business value. The language will evolve and deepen as the application grows, new topics are discussed and more powerful ways of understanding are discovered[7].

The customers write stories and tests covering the domain entities the system has to present to the users, the relationships among them, the business rules and policies constraining their behavior and more and these have to be understood by the developers in the same way they are by the customer side.

Selective abstractions.

While we advocate the language of the domain, it turns out that the language of the subject matter experts (SME's) is not ultimately satisfactory because it is both too nuanced with context and not precise enough. Developers and analysts cannot be expected to understand this language the same way an SME would. In any case, the software will never capture the full meaning of all the domain terms it deals with.

The key to a successful project ubiquitous domain language[10] is a selective, mutually agreed to abstraction. The team needs to choose a set of meanings that are consistent with the customer use of the terms but have a precise constrained meaning in the application. These entity concepts should be the actual business objects contained in the code so that their attribute and behaviors can match customer expectations exactly without need for complex translation.

This language emerges through refactoring of the code and refactoring of the use cases and interface design as both sides collaborate to define and create the application. It reflects both the SME's deep domain insights and the developer and analyst implementation insights. It may well contain invented application concepts that are not common domain usage but make the application intelligible and learnable.

Domain language model.

The domain language is not an analysis model because it is not implementation free. Neither is it a design model because it does not include the full architecture needed to deliver the application but only touches the ideas that the customers directly need to know about and validate.

The whole team evolves this language which is directly implemented as the business object layer of the application. The model is implemented in code; its behavior is tested in customer tests. It is generally not recorded in formal documents outside the code and tests because it is continuously refactored to improve its clarity and effectiveness for addressing the goals of the application.

The team will anchor conversations with informal UML diagrams whiteboard style including as needed class diagrams, interaction diagrams, and state diagrams. These may be temporarily noted on index cards or snapshots of whiteboards. Details are recorded in application code and customer test code.

It is common for a team to start a project by implementing the basic business domain objects and their basic behaviors before investing in the user interface. This gives the customer side sometime to work on designing the interface. To design the interface, you need to understand the users' goals in some detail.

Tool 8: Role Model

The developer side needs stories but the customer side usually has to start with understanding the people who will be using the system. The people have characteristics and work in contexts that constrain the user interface and workflow design options. The people have goals and mental models of their domain, responsibilities and business goals. So we start with people tools.

Role Modeling from Usage Centered Design supplies the tools necessary to understand the people who will be using the system to create business value. Lean principles and concurrent development suggest that the model does not need to be complete up front but can be detailed when and to the extent it is needed.

The starting point is a simple list or stack of cards with one role for each category of expected users.

Know the actual users.

The customer side members of the project team are normally developing a new or modified business workflow in addition to a new software application to support part of that workflow. The business processes that the new workflow implements creates the value the project delivers. Usability problems, which can dramatically reduce that potential value can happen if the customer side does not understand the experience the users will have using the new application in their new work context.

Role modeling starts by identifying who will be using the system. A person able to use the system effectively must know or be able to easily recognize the actions to take to capture or access or modify information as required for the transaction the system is supporting for them.

Role inventory.

If you are familiar with use case modeling, you know the concept of actor. A role is an actor who happens to be human. Actors which are external systems need a completely different style of analysis for their interfaces. A role is not a job description but rather a type of use of the system. E.g., writer, editor, reviewer of a word processing application.

Record each role on an index card so you can sort, organize and relate them by importance, by similarity, or other characteristic. Do not hesitate to tear up role cards and rework them as you learn more.

Role description.

The standard practices of usage centered design support a focused analysis of the factors that determine the characteristics of a role which guide creation of a usable interface. This analysis probably does not need to be done up front and the extent to which it needs to be done at all will vary from project to project. The interfaces are probably not the first components to be built. The team can defer investing in detailed role analysis until the iterations in which interfaces for those roles are defined.

Role priority.

Not all user roles will contribute equally to the net return on the investment made in the new application. Satisfying the needs of the high priority roles is usually the most valuable place to begin implementing the new system. Understand and invest energy and thought in making the system highly suitable for use by the high priority roles. You can afford to pay less attention initially to the lower priority role needs.

The concept is breadth first inventory of roles and depth first concentration on those with the largest impact on the success of the project.

Role cards may become or may generate user story cards. For example, A story might describe authorizations each role gets to use various system abilities.

Tool 9: Essential Use Case Model

People use the system to achieve some business purpose. Once we understand who will be using the system we can explore just what each important role needs the system to be capable of doing. While role modeling concentrates on making the new application fit the people executing the tasks to enable them to use it quickly and with low error rates, use cases concentrate on the tasks and transactions the people are interacting with the system to achieve.

Goals of each role.

Create a goal card for each goal an actor/role expects the system to help them achieve. After the user playing a role communicates an intent, the system is responsible to know how to respond, to remember what was input, to apply domain rules, to create what was requested, to communicate with some party or whatever the actor expects. These are all things the customer will need to write stories, hold conversations, and write tests about.

A user's goal may or not become a user story. If a team has lots of experience in a domain, the goal may be all they need for a story. Some goals will need to be detailed as use cases, and a broken into a collection of user stories and tests. The customers may do quite a bit of organizing, redefining, and detailing these goals before the software is done. The idea at the beginning is simply to identify a set of goals to enable some initial planning and prioritizing and to identify a place to start.

Goals use domain language.

Each user goal is an intension the user expects to interact with the system to accomplish. Each goal should be an active verb phrase. A verb phrase includes minimally a verb and a direct object that the verb acts on. ("Take food order", "Schedule order pickup", "Make service appointment"). The object of or target of the action verb will usually be a domain language concept - (food order, order pickup, service appointment.) Communication within the customer side and with the developers depends on everyone on the team having the same understanding of the domain concepts the system is responsible for dealing with.

These concepts are the fundamental vocabulary of the users and need to be also the fundamental business vocabulary of the application. To ensure this, create a model of the domain concurrently with your creation of the goal model. Record each concept on an concept card. Include a brief description of what the team agrees to mean when that concept is used. As you refine your stack of goals, maintain your stack of domain concepts.

So now we have role cards, goal cards, and domain concept cards. Focus on identifying the really important cards that will make or break the application not getting a complete set of cards or on details.

Use cases as a customer tool.

Use cases can lead to analysis paralysis if the customer team does not know what it is doing. The key issues are getting the goals and steps at the right level of detail and leaving out all non-behavioral detail. *Decide* details *as late as possible* to minimize the cost of change so the team can try many alternate solutions to *amplify learning.* This permits the team to *see the whole* and understand relative priority and value.

A use case is not a story. A story is about something the developers are asked to build and a use case is about something the users intend to accomplish by using the system to manipulate domain objects. Nonetheless, use cases can be very helpful to the customer in generating stories, discussing the details with developers, and defining a good set of tests that cover all the variations the customers care about.

Goal levels.

The hardest part about identifying the goals of a role is to get the granularity right. There is a continuum of possible levels of detail. Goals live at three levels.: summary, user-goal, and sub-goal. The user-goal level is the one the majority of your use cases should be at. If a goal is at the user-goal level than:

- How often an actor does it is a measure of how much business value the actor has produced…Logging in is clearly not at this level.
- The goal is accomplished (or abandoned) in a single session. The user would start up a session with the system to achieve the goal and could shut it down when it is complete

Of course, goals form a hierarchy and later you may well break down goals into steps either

- To explain what it means to accomplish the goal,
- To factor out common or complex details, or
- To break down a use case into stories developers can estimate reliably.

Goal hierarchy.
The goal is to keep most of your use case goals at user-goal level. Sometimes you will need to factor out common or details steps into sub-goal level goals. A few use cases will be useful at summary levels which tie together a collection of user goals into an overall business process that transcends a single user interaction with the system.

Of course there is actually a continuum of levels of goals. For each use case with a goal at some level the steps of the use case will be at a lower level. The goal is why the actor is interacting with the system and the steps explain how the interaction takes place.

Prioritize goals.
With less than a week of work, the team will have dozens of goal cards and supporting role cards and domain concept cards. The developer side needs the first batch of stories soon. Which ones to detail first? -- It's best to start with the most important ones which are those that deliver the most value. A key reason for keeping the goals on cards is to support planning and estimating. An effective strategy is to sort the tasks based on a variety of criteria including:

- By how often they will be done
- By importance to project success
- By developer estimate of technical risk
- By developer estimate of size

Given all this information, the can order the cards into <u>M</u>o<u>SC</u>o<u>W</u> piles

- <u>M</u>ust Have – Do first, top priority, the system is not worth having if these goals are not supported.
- <u>S</u>hould have – Important, System usability or performance would be impacted if these goals are not supported.

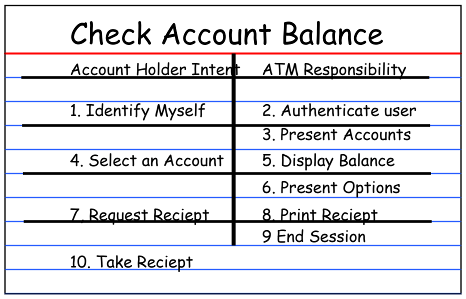
- Could have -- Convenient, nice to have features that would benefit some roles or some situations but work arounds exist.
- Won't Have -- not needed for this project. Discard pile.

If the goals are too large for the developers to estimate risk of effort confidently, the customers will have to break them down into stories that are smaller. They may do this be expanding the goals into more detailed use cases.

Refine and refactor tasks.

Next the customer selects the goals/stories that will fit into the objectives allocated for the first release and to start investing in some details for how the user goals will be realized.

A story card is supposed to have a few sentences describing what the story is about. For stories about user goals, start with essential use cases[6]. Essential use cases will usually fit easily on the front of an index card as they identify a sequence of user intents and consequent system responsibilities. It is common to use a two column format. The essential style leaves out most of the detail common in move verbose styles including UI details, domain and policy details, validation details, etc. It focuses only on actor intent (what result does the actor intend to achieve?) and system responsibility (what must the system do to respond to the actors communicated intent?) and the conditions that these create.



Check Account Balance

Account Holder Intent	ATM Responsibility
1. Identify Myself	2. Authenticate user
	3. Present Accounts
4. Select an Account	5. Display Balance
	6. Present Options
7. Request Reciept	8. Print Reciept
	9 End Session
10. Take Reciept	

Figure 9 -Essential Use Case

Actor intent.

The actor's intent is what the actor expects to accomplish from an interaction step. Each intent is a lower level goal contributing to the use case user-goal. Express the intent as a verb phrase without conditionals. The system implicitly will be responsible for offering some kind of interface that will permit the user to express the intent. These system responsibilities will normally be allocated to some user interface objects invoked by the current active workflow controller.

System responsibility.

The second column of the essential use case records the responsibilities the system must fulfill in response to the users expressed intent. The format is again a verb phrase without conditionals.

Responsibilities of the system are often passed down the objects in the business domain layer though some of them may be handled at the controller level if they simply invoke another window. Typical system responsibilities include providing some output to the user, changing the internal state of some business domain object, or setting up the context for the next user interaction.

Refine goals into use cases.

Use cases need to be refined in two situations. First, if a use case is too large to be a suitable story, the individual scenarios of a use case are often a good way to break it down. The main scenario becomes one story, one or more of the alternate courses become additional stories, perhaps with different priorities. The second reason to refine use cases is to discover a full set of customer test cases. The customer is responsible for testing every situation they care about. Every step of the use case is a potential case where something can go wrong and either the user or system must take some action other than the normal courses. If the customers want these exceptions covered, they must define tests specifying what should happen.

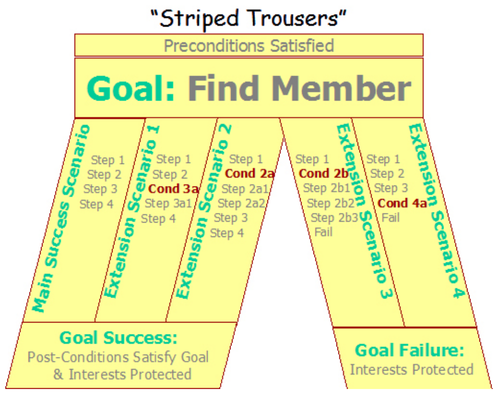


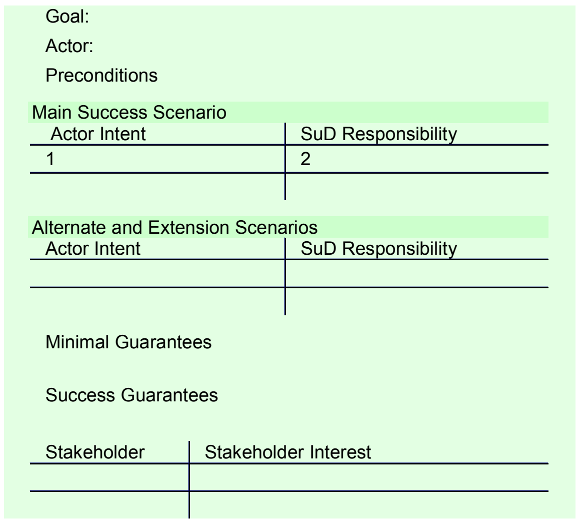
Figure 10 - Main & Alternate Scenarios#

This "striped trousers" approach (Figure 10) is detailed in Alistair Cockburn's *Writing Effective Use Cases* book[8]. The two keys to use this tool in an agile process are:

1. Keep the steps in essential form, deferring details to the conversation.
2. Recognize that the preconditions are test setup, the scenarios are test scripts, and the post conditions are the expected test results.

Thus "Full Dress" use cases identifying alternate success and failure scenarios, pre-conditions, guarantees and stakeholder interests are actually a test discovery technique that helps the customer side be confident that all cases they care about are covered. Of course you only do full dress at the last minute as preparation for or as an outline for your conversations with the developers and as a check to see that you have written all the tests you care about. The use case is a working document, NOT a deliverable!

Essential use case template.



Goal:

Actor:

Preconditions

Main Success Scenario

Actor Intent	SuD Responsibility
1	2

Alternate and Extension Scenarios

Actor Intent	SuD Responsibility

Minimal Guarantees

Success Guarantees

Stakeholder	Stakeholder Interest

Figure 11 - Essential Use Case Template

I offer this use case template as a reminder of the some things to focus on to ensure your conversation and tests cover all variations implied in the striped trousers (Figure 11). The preconditions and branch conditions are often the test setup steps. The scenario steps are the steps in the test. The post conditions and stakeholder interests are the test results the system is responsible for producing. The number of tests is at least the number of alternate scenarios in the use case.

If you write out use cases, keep most of them down to one page. If you need more space, consider breaking out some steps or scenarios into another use case. The use case is not a deliverable and should not be polished. Focus on the completeness and quality of the tests you derive from it instead.

A use case is sometimes the same as a story if it is short enough. Sometimes each scenario is a story. Other times a scenario must be broken down into several stories. When you break up a use case into several stories, you might want to note which use case a story came from to help remember the context and testing considerations for each story. In any case, remember that stories and use cases serve different purposes. And not all stories will come from use cases. Some will describe business rules and policies, other will request user documentation or specify response times or integration with a particular existing database for example.

Tool 10: Interface Content Model

Usage-centered design, of course, is most focused on creating effective interfaces. None of the agile methods have anything meaningful to say about the topic. They all trust that a user interface can emerge from repeated refactoring, a rather dubious expectation in cases in which tasks are complex and the people who will be using the application are significantly different from the people building it.

While the engineering practices of XP will enable user interfaces to be modified significantly, even quite late in the development process, it must be admitted that refactoring users is not viable. Once an initial release is put into production in the hands of a significant number of end users, they will rapidly develop proficiency via muscle memory that will not be easily relearned without a significant period of disruption.

The idea is to get the look and feel conventions and overall navigation of the application defined by the end of the first release so the visible interface, at least will not need to be updated. Deferred detailed design of individual screens and use cases until the iteration in which they are implemented.

User interface.

The interface tools are listed last because they tie together the domain language concepts and the use cases. They need to be done concurrently with the other models.

User stories are not the proper unit for addressing the user interface. Users interact with the system to achieve user-goal level use case goals and the user needs to figure out how to accomplish each goal and each alternate approach to getting there as the situation demands. A good UI design requires understanding of the user's context, workflow, and overall goals which are not covered by any one particular story.

Organize user tasks.

User interface design is a customer responsibility which means that the UI designer is a member of the customer side of the team. Usage-centered design teaches how to proceed from the domain concept model and use case model to an interface model. Having use cases written at the user-goal level is the first step in effective task clustering. Clustering use cases by role and workflow will identify tasks that need to be near each other in the navigation map and easy to find and to operate similarly.

Concurrent development means that the interface model, the use case model, and the domain language are being developed at the same time and that each influences the development of the other. There is no one best way to partition the user's goals and sub-goals into use cases and there is no unique best collection of domain language to use for the project. Rather, understanding goals drives the interface partitioning and interface opportunities influence how intents and responsibilities are allocated among use cases. Domain language constructs need to be invented to express and integrate both the interface and the use cases while capturing domain business rules and policies. This is an activity where effective object thinking will pay rich dividend in simplicity and usability. Conceptually, use cases, interface screens, and domain language concepts are all objects and can be split, merged, have responsibilities and attributes moved from one to another as needed to obtain a solution as fit for use as the teams skill permits.

Essential UI cards.

Make rough paper prototypes, index cards and post it notes work well and are easy to modify cheaply and quickly (Figure 12).

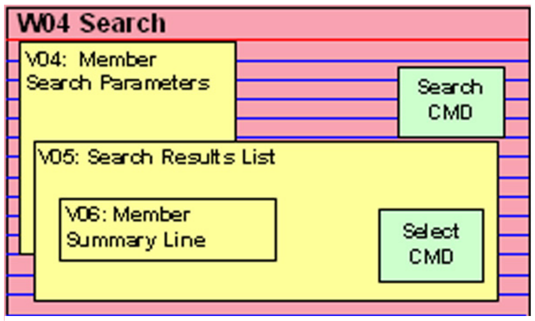


Figure 12 - Essential UI Card

Start with the highest priority use cases used by the most important roles. Cluster the tasks likely to be done as part of the same workflow and use the same business domain objects. Each cluster is a candidate interaction context. Each user intent is a potential command. Each interaction context (window) will present views of one or more domain language concepts. The views present or accept some or all of the values associated with the domain concept. Commands permit the user to generate events that the system is responsible to respond to in some way. The command may be implemented as a button, a double click, a selection, or any other appropriate idiom. If a cluster requires more than one window context, split views and responsibilities and create a segment of a navigation map with events that transition from one context to another.

Walk through the use case scenarios to see how easily each task and variation is accomplished for your initial design. Make changes until you achieve a good balance. Then proceed to add on the next most important use cases and continue the process.

You will create a collection of index cards and a navigation map that shows which other interfaces can be navigated to from each.

The customer side should complete the navigation map and settle on the look and feel issues for most of the Must and Should priority use cases before the first release of the application.

Each interface card either becomes a user story card or attaches to the story card it supports depending on developers estimate of the size of the implementation effort. The navigation map and style conventions become part of the context for all team estimates, just like crosscutting stories about security, logging, internationalization, etc. The team will probably implement the interface story in the same iteration or soon after the stories associated with the corresponding use cases.

Interface behavior tests.

Every story needs both detailed discussion of the details of how it works and of tests to prove that it does. At this point all we have for the user interface is an index card and post it not prototype that does not even have specific fields or button names or labels. The discussion is the time to work these detailed appearance issues out. The developers can probably generate the precise GUI by dropping widgets onto a canvas as they discuss the window. They will get attributes from the domain model… Or will refine the domain model by allocating the attribute needed on screen to the appropriate domain class.

Behavior is another issue entirely. The approach I use is to start by identifying the **states** of each window. These may be empty, populated, no search results, etc. Then for each state describe what the window is like in that state. For example, what is enabled, disabled, blank or populated, defaulted…Finally, when the window is in each state, what events will it respond to? E.g., commands, system events, keyboard input, etc. (Figure 13)

Then, for each state construct a state chart with three columns. The first column lists the events. The second describes the response of the window and system to that event and the third lists the resulting next window and state. (UML state diagrams get far too messy to be useful for this!)

And there you have the tests for the interface stories. Each event / response / next window state is both a specification for the code and a precise specification of the test. Since it is written at the event level automated tests can be driven just below the actual interface view level.

State Name	State Description

When this window is in this state

Event {or Result}	Produces System Response	Next Window.State

Figure 13 - Interface State Behavior Chart

The Agile Customer Toolkit.

Lack of effective participation by the business side is on of the most regretted shortcomings of IT projects in the eyes of CEOs today. Two important reasons why Agile approaches work better than sequential approaches are:

1. They all insist on ongoing close participation of the business side in the day to day progress of the project.
2. They embed principles of lean thinking to eliminate waste and enable effective collaboration within the entire team to identify and deliver business value.

However, agile methodologies are uniformly weak in helping the customer side deliver on their assigned responsibilities and numerous would be agile projects have failed as a result. In addition, none of the agile books or discussion groups says anything about user interface design or usability beyond how to do automated unit and functional tests of an interface.

The practices of Usage Centered Design and Domain Driven Design, applied in a manner consistent with lean thinking fill this gap and the decision tools and roles tools guide the business side in acting as a full partner in development projects..

---

[1] Poppendieck, Mary and Tom, *Lean Software Development - An Agile Toolkit*, Addison Wesley (2003), See also www.leanprogramming.com

[2] Highsmith, James A. *Adaptive Software Development – A Collaborative Approach to Managing Complex Systems.* Dorset House (2000)

[3] III, Immunizing Against Predictable Project Failure – Charters and Chartering as a baseline for Change, *STQE* January/February 2001

[4] Cohn, Mike, *User Stories Applied for Agile Software Development* Addison Wesley (in press) available at www.userstories.com

[5] http://www.xprogramming.com/xpmag/expCardConversationConfirmation.htm

[6] Constantine, Larry & Lockwood, Lucy, *Software for Use,* Addison Wesley (1999), See also www.foruse.com for More Agile approaches to Usage-Centered Design

[7] Evans, Eric, *Domain Driven Design - Tackling Complexity in the Heart of Software*, Addison Wesley (2003)

[8] The structure of the use cases is from Cockburn, Alistair, *Writing Effective Use Cases,* Addison Wesley (2001), the essential style follows Constantine.

[9] Schmaltz, Davit *The blind man and the elephan*t, Berett-Koehler (2003)

[10] Evans (2003) has written extensively about this ubiquitous domain language and how to achieve it in a agile project.