Faster and Less Volatile Requirements Discovery

By Dr. Timothy Meehan and Norman Carr
www.guibot.com

The software development community has generally recognized that the waterfall method is fundamentally flawed as an approach to the software project lifecycle [1]. The most recognized weakness in the approach is that system requirements are rigidly defined too early in the development lifecycle, which produces incorrect or incomplete requirements. The poor requirements then create a situation of requirements volatility. Success or failure of a software project is largely dictated by the quality of the requirements [2-3], and the more volatility and inaccuracy in the requirements the more likely a project will go overtime, over budget or release the wrong functionality. Given that some requirements will not, or cannot, be known at project inception, and, furthermore, those that are 'known' are often later revealed to be erroneous or to be expensive to change leaves us with a conundrum on how to obtain accurate requirements in a practical timeframe. Simply put, the more quickly and more accurately requirements are obtained, the better.

Responses in the industry to requirements volatility advocate an evolutionary or iterative approach to requirements gathering and analysis coupled with incremental implementation. The iterative methods include the Rational Unified Process (RUP), Xtreme Programming (XP) and Agile methods. All have been championed with the assumption that complete requirements are initially unknown, nebulous or wrong and will therefore change and evolve [1, 4-6]. To overcome the requirements volatility, the iterative methods rely heavily on prototyping and feedback for the requirements discovery, arguing that exploratory programming is essential for obtaining the correct requirements to arrive at a solution. Figure 1 illustrates the general path for iterative approaches. Requirements, analysis and design, implementation, are testing are performed, then the iterative release is shown to the customer for feedback and agreement

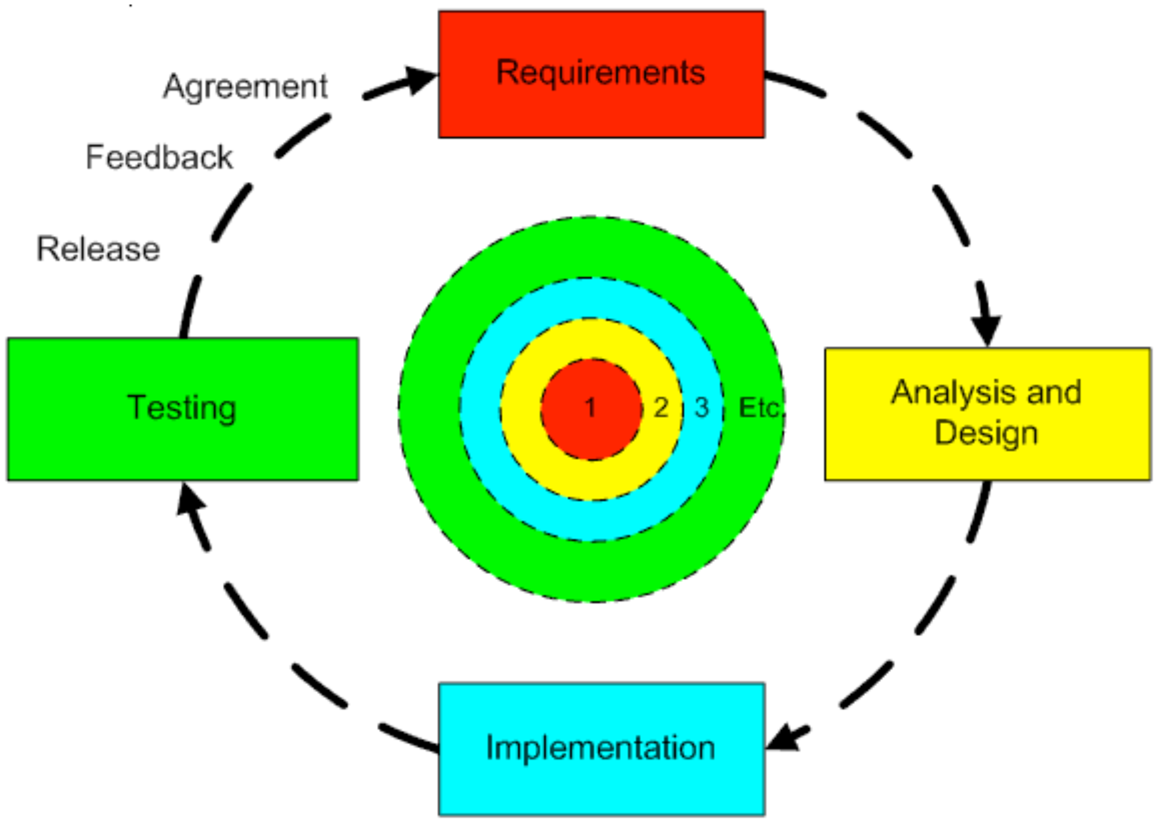

Figure 1. Iterative Development methods place greater emphasis on requirements and analysis in the early iterations, while placing greater emphasis on implementation and testing in the later iterations.

Our experience with iterative methods, most notably RUP and UML, has been that iterative approaches, although certainly more effective than the waterfall approach, retain an undesirable risk of failure in their present incarnation. A fundamental premise of the iterative methods is that feedback through early system construction will lead the system development in the right direction with less requirements volatility and greater requirements clarity. We have found this not always to be the case.

Kruchten et al render a practitioner's guide on what to do and not to in order to avoid the possible pitfalls in iterative development [7]. Even after such guidance, we find that "analysis paralysis" can happen or volatile divergent requirements ensue, with either one having the potential to gridlock the iterative approach. Furthermore, going to small system or subsystem builds to break the gridlock starts to become a prohibitively expensive exercise and time consuming practice for complex systems. With no guarantee that such an exercise will break the gridlock and move development in the correct direction, we find the risk untenable. Inevitably, the practical world comes bearing down on us [8]. To accelerate the iterative methods and guarantee an incremental result in the correct direction, we still need a means for quicker and less volatile requirements discovery.

Reasons for the Gridlock

Whether analysis paralysis or volatile requirements, we conclude that the fundamental cause of inefficiency is ambiguity – analysts and end-users have difficulty expressing their needs into words that can be interpreted by programmers to produce the correct system. Even after prototyping to better express the system needs, designers and programmers can lack the confidence or ability to proceed because the information provided to them *during the current iteration* is incomplete, inconsistent, or even contradictory.

we find that ambiguity is largely attributable to narrative use cases or narrative user stories. In the iterative methods, narrative user storyboarding is generally the preferred method of gathering the system purpose. In Coburn's book, "Writing Effective Use Cases" [9] he presents a very effective tabular form for gathering the substantive elements needed in a use case specification, but he also advocates narrative discussions with respect to the user scenarios. We don't. If used as the primary communication between stakeholder and developer, we find that no matter how well written, prose text leaves too much room for ambiguity, error or omission when attempting to thoroughly and accurately describe the behavior of a software system. The inability of the text prose to precisely and completely communicate requirements grows proportionally with the complexity of a scenario or use case. Ultimately, describing the system requirements as prose places enormous pressure on the analyst to write a narrative that can be interpreted in only one way by all stakeholders. In the English language, this ability is especially rare rather than common place.

In addition, some measure of completeness is essential during each iterative cycle. Text media provides neither inherent control upon the completeness of the requirements it describes, nor does it offer any explicit check of its inherent level of ambiguity. Hence there is no guard against the risk of developers making erroneous assumptions or misinterpretations. The familiar disconnect between customer and coder is, therefore, a result of the inability of prose to enable a "measure of understanding" or even a "measure of ignorance".

Documentation maintenance is equally problematic. Changing prose increases the risk that new interpretations and new ambiguities will be introduced. Inserting new prose does not offer any adequate gauge on the cost or effort necessary for those changes. The level of impact of those changes upon the existing or developing system is then also open to interpretation on the level of effort required for delivery. Again, further unacceptable risk is introduced, and we reach an untenable situation.

Constructing functional prototypes to overcome and clarify the density and ambiguity of prose documents, often introduces new problems of its own. The biggest being that building meaningful prototype functionality can require more work than can be delivered in an adequate time frame necessary obtain agreement on the requirements. For an internal project we might have more latitude on the time frame. However, for customer projects, the customer is more interested in getting it right the first time, so that the requirements interviews and review of prototypes does not unacceptably impede their immediate workaday world.

A more insidious shortcoming of building functional prototypes is the difficulty of directly mapping prototype interfaces to a narrative use case specification. It is difficult to reliably trace use case prose fragments to prototype features, because they co-exist independently of one another.

Short Circuiting the Iterations

Recognition of the problems described above inspired us to devise a more reliable method for harvesting and recording user requirements, whose sole purpose is to arrest requirements volatility. Additionally, it has to work with and communicate to end-users, programmers, and executive management, regardless of their sophistication with software systems.

We found the most effective approach for the rapid development of accurate and unambiguous requirements is yielded by drafting the user workflow in activity diagrams and collaboratively prototyped GUIs simultaneously. By combining activity diagrams and user interfaces, we define the user scenarios graphically. With this method, we have essentially dumped narrative use cases, except where the narrative provides value, such as the statement of purpose or goal, glossaries, etc.

The resultant GUIs and activity diagrams help both the customer and developer identify ambiguity and incompleteness, while reducing the risk of misinterpretation. When performed within the context of a use-case driven approach for identifying user goals, we achieve very fast stakeholder buy-in. Essentially, we have added a clarity loop to the iterative path as illustrated in Figure 2.

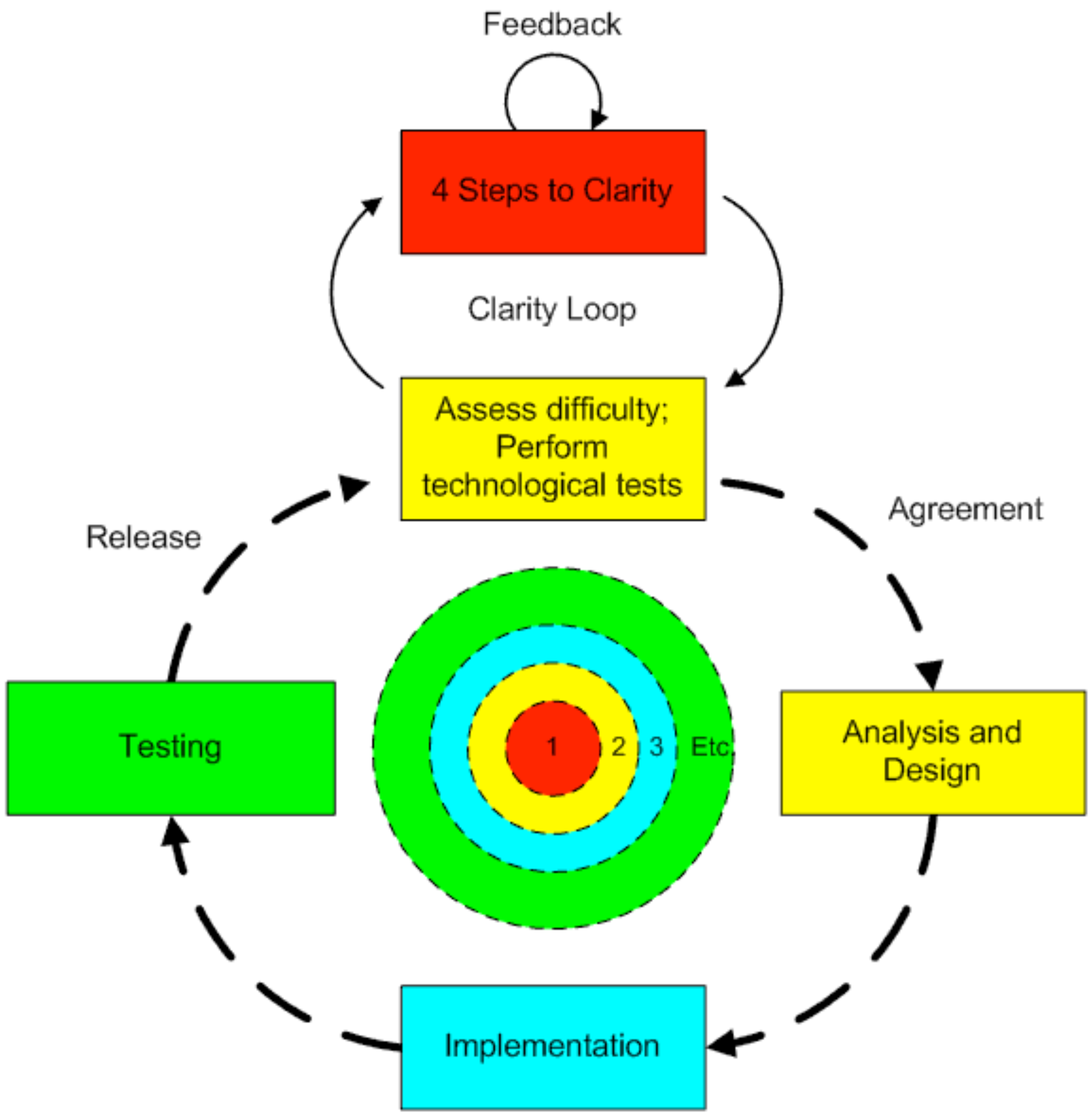

Figure 2: The Clarity Loop added to the iterative process with feedback and agreement engaged earlier in the iterative cycles.

For the mini-waterfalls in iterative steps, combining activity diagrams and GUIs places the feedback and agreement earlier in the cycle than the iterative cycles illustrated in Figure 1. Thus, the analysis and design, implementation and testing are more meaningful rather than exploratory.

By representing the business processes, rules and behaviors in the activity diagrams and their corresponding GUI prototypes, the desired representation of the business process as a software system is quickly established. The activity diagrams map the business process while the GUI mirrors the business process for the desired software system. Their marriage comprehensively describes the implementation of that system. Furthermore, the

GUI's interaction workflow will be user friendly by design, since it is crafted to be representative of the user's workflow for the business task at hand.

We can liken this approach to building construction. Many architectural drawings are produced iteratively and incrementally until agreement on the requirements of the building, its site, features, facilities and cost is reached. The specification for the innards such as design of structure, electrical and plumbing may require much iteration, but the customer is now less involved at this stage as should be true for a software system design. Once construction begins, any change requests can be judged in terms of impact to schedule and cost. The customer is then fully aware of those costs, and more informed decisions can be made. The same is true for this approach to software development.

A New Beginning

Our approach for requirements discovery has become much more simple, direct and meaningful. We achieve clarity earlier, because the business process is mapped in the business language while the GUI prototypes depict the business user interface. Feedback is immediate when drafting the user interfaces simultaneously with specifying the business process. Functional prototyping development now has a better and therefore less costly starting point for subsystem builds. In addition to greater clarity and less ambiguity, this approach results in less disconnect between end-users, analysts and programmers, easier understanding of requirements by programmers, fewer go-arounds between programmers and analysts and end-users, faster buy-in and ultimately fewer iterations. Executive management also receives a clearer snapshot of what they are purchasing, and therefore are more inclined to support the project.

We are not purporting elimination of iterations. Early interviews almost inevitably provoke nebulous content. We do find, however, that sources of misunderstandings and ambiguities emerge much earlier than they otherwise might – the "measure of understanding" and the "measure of ignorance" are improved. Consequently, it is easier to determine where best to direct further analysis. Earlier identification of either well or poorly understood features of the system results in better cost estimation and improves knowledge of the effort required to complete the project.

This process also affords a number of additional advantages over narrative use case scenarios. As the user scenarios are mapped in the activity diagrams, we are simultaneously drafting the test scenarios in the same go. The paths in an activity diagrams provide the test scenarios for the system. Provided that the system behaves in the manner illustrated by the activity diagram alone, then a test has succeeded. Behavior outside the path determines failure.

Much of current IT practice involves integration services, which can be a gotcha point in any software project. Depending upon the initial level of ignorance, such as learning and understanding an API, a web service, legacy or other system can balloon the cost of a project. By using our approach, the activity diagrams and necessary user interfaces for the integration provide a better measure of ignorance allowing for better targeted analysis on points of integration and how they will be implemented.

Activity diagrams also provide an excellent tool for gauging the impact of change. Changes in an activity diagram can be minor, such as “raise an error message here so that the user understands the problem”, or more significant, such as an addition of new swimlane that represents an exterior system to be integrated. Many synchronous paths or decision points in the activity diagram implies that the complexity of the system will increase. Therefore, the level of effort is determined more immediately with activity diagrams, and significant additional levels of effort can be identified as a major change request that impact schedule, costs or scope delivery.

Finally but crucially, the close relationship between activity diagrams and GUIs resulting from this techniques defines total end-user scope and acceptance criteria at the same time as the requirements are collected. Use cases bound the global scope, activity diagrams describe the scope for each use case, and the GUI elements associated with each activity diagram path represent the complete scope of the user interface. If the system follows the behavioral paths of each activity diagram and the user interface elements exist to allow the completion of those paths, then acceptance is met.

Drafting the use case specification, acceptance criteria and test cases is performed simultaneously with the construction of the use cases, activity diagrams and GUI’s. Publishing the specification becomes an easy task by simply combining all of these elements. The stakeholders of the system and the developers of the system remain on the same page with respect to business goals (use cases), business tasks (activity diagrams) and user interfaces.

4 Steps to Clarity

Our requirements analysis approach is now very straightforward. Goals are defined with Cockburn use cases, but instead of writing narrative user scenarios, we describe the tasks to achieve the goals with activity diagrams. Their corresponding GUI prototypes then represent the system visually. The customer sees their business processes and the user interfaces they will be using to fulfill those business processes.

We call this requirements process “The 4 Steps to Clarity”.

- Define the goals through use cases
- Determine the tasks to achieve those goals through activity diagrams
- Draft the corresponding GUI’s to visualize the system
- Publish the specification

The process more easily maintains the communication and understanding from stakeholder to programmer. End-users and executive management understand use case and activity diagrams, because the diagrams are written in their business language.

Programmers understand the required system behavior from the activity diagrams. Everyone understands GUI's because they use them every day.

A Simple Example

To illustrate the process we use the example of withdrawing money from an ATM in Figure 3. The bank client wishes to withdraw money, which we state as the goal, or use case. To withdraw the money the bank has included an additional goal of identifying the customer.

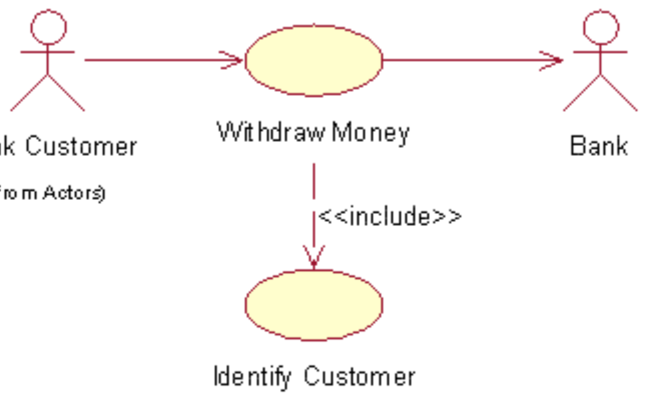

Figure 3: ATM use case example.

The general steps are the same regardless as to whether the bank client deals with a teller or uses an ATM. They identify who they are, choose an account, and then specify an amount to withdraw. The bank authorizes the transaction, and the amount withdrawn is decremented from the total amount in the account. At the bank counter, the business flow is represented in Figure 4 as

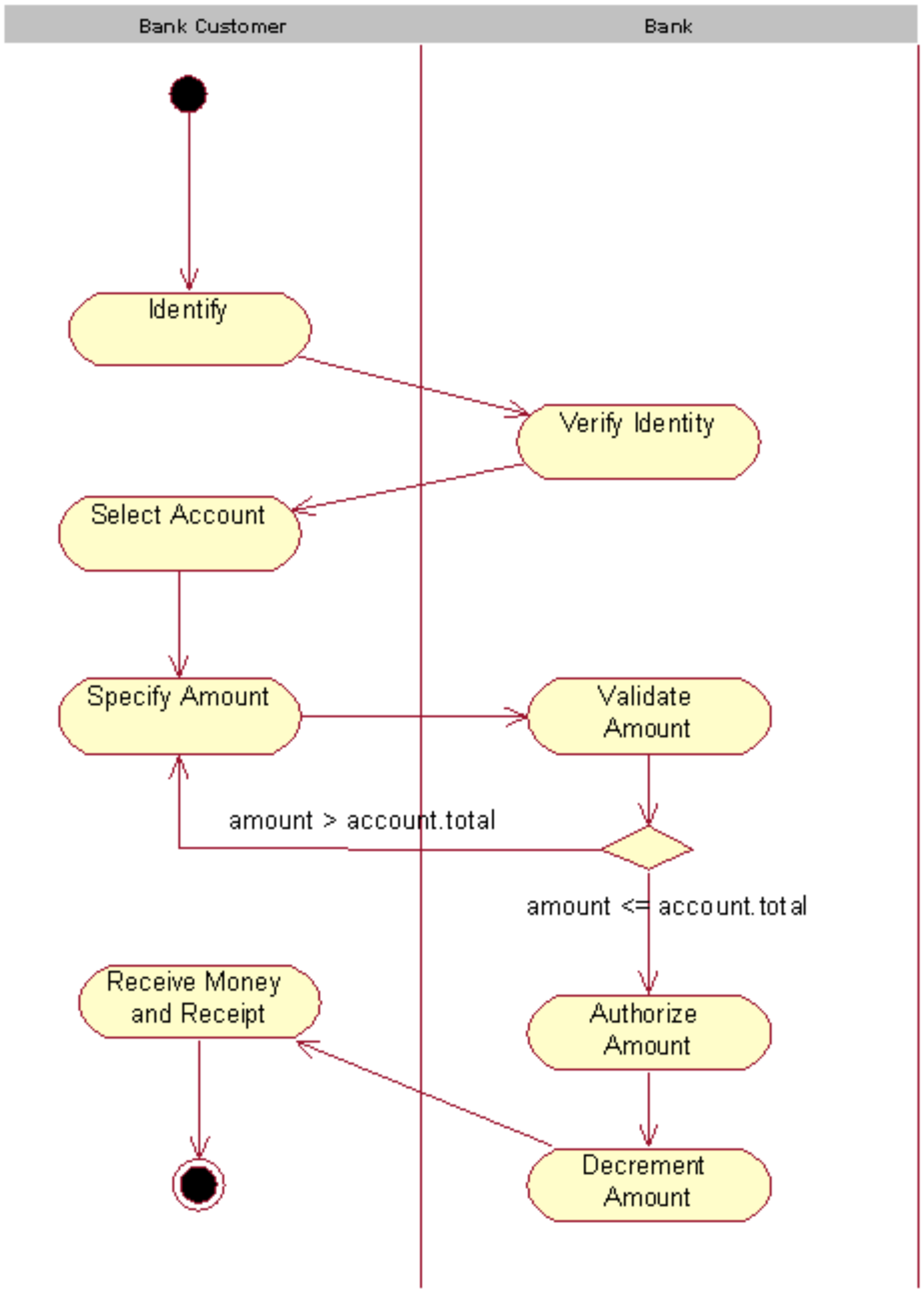

Figure 4: Withdrawal business flow

With an ATM, the identity is usually verified via a bankcard and a pin number. Now the activity diagram becomes in Figure 5

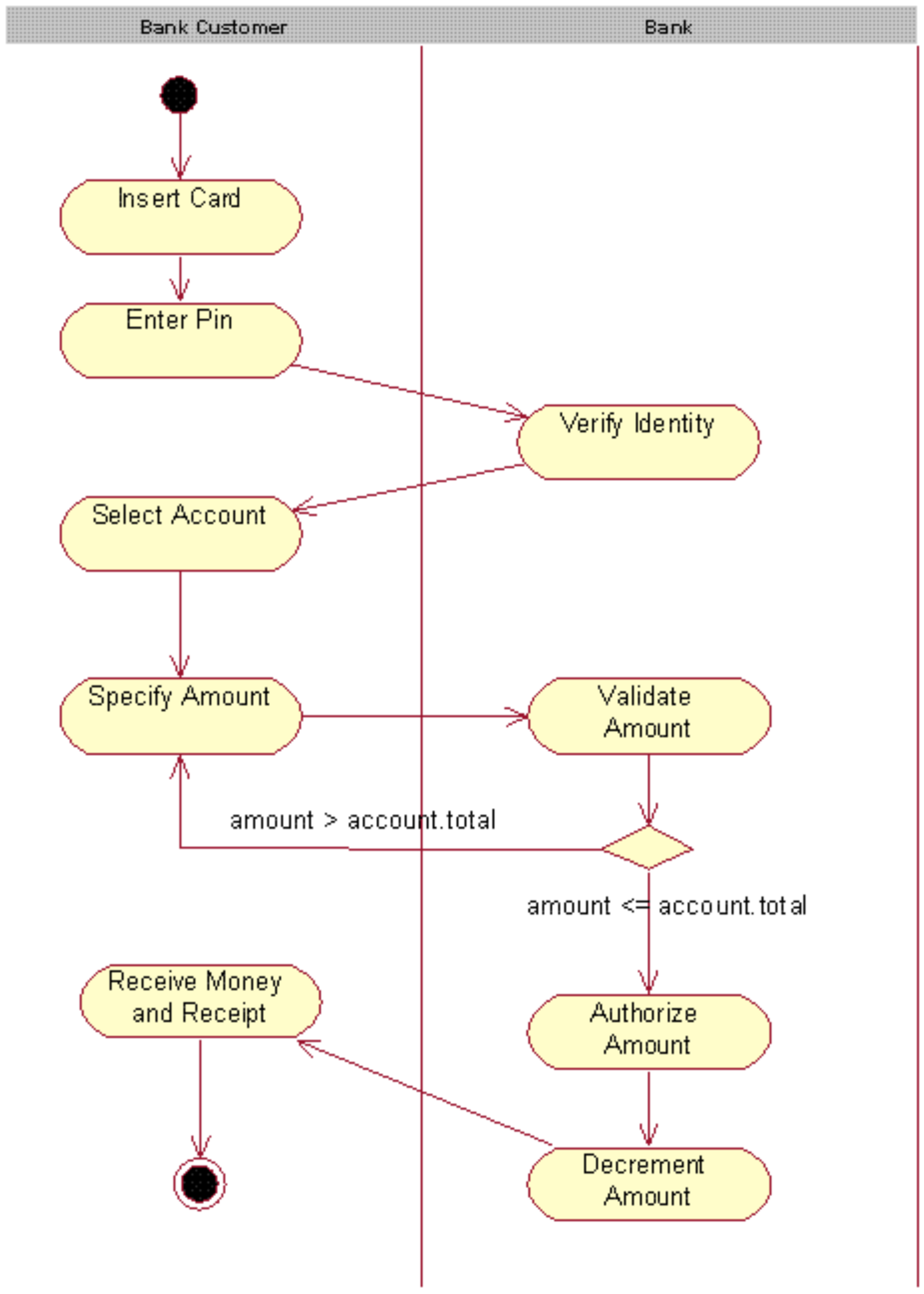

Figure 5: ATM Withdrawal workflow.

The GUI prototype for the action steps of the activity diagram becomes in Figure 6

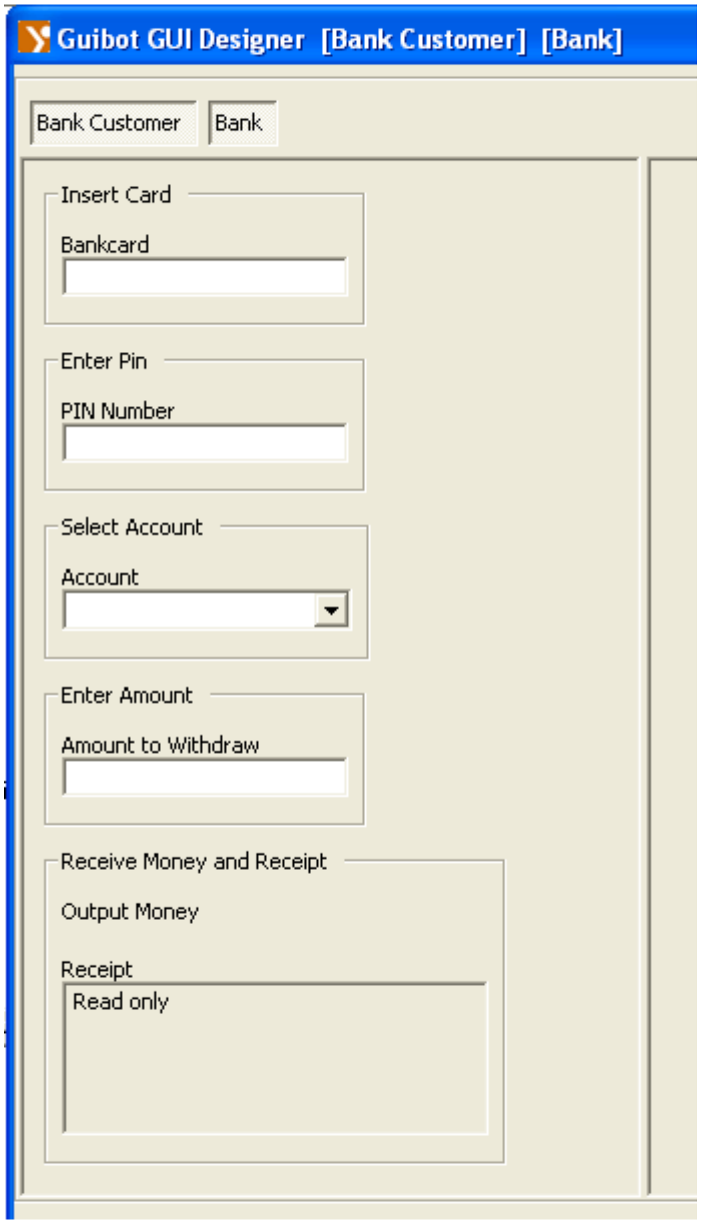

Figure 6: ATM prototype user interface.

Therefore, each step in the business process that requires a user interface is mapped to a user interface component. Since the bank card slot, money slot and receipt slot will be hardware interfaces, specifying the requirements of the system against the business process with this method produces an implementation agnostic specification, where the business process is the focus. If technology changes the business process, this change can easily be interjected into the activity diagram. New activities with a technology prototype mapping to the new action steps in the diagram as illustrated between the activity diagrams and user interfaces above.

Conclusion

Independent of the iterative method, we find this process results in quicker and more accurate requirements discovery. Primarily, the faster results are due to the visualization of the system in the context of business goals and workflow. Therefore, we tend to have fewer and faster iterations, better prioritization, better estimates due to a better gauge of the level of effort, and faster buy-in from the end-user.

Even were fewer total iterations not realized by this process, it most certainly kick starts any software project in a productive direction. The easy bits and understood bits materialize very quickly, while the nebulous parts show themselves more quickly and analysis can be better applied to those parts. Hashing out activity diagrams and GUI's together with the customer is a far more effective and less costly means of describing the nebulous parts as compared to partial system building only to find what is built is not correct.

1. P. Kruchten, *The Rational Unified Process - an Introduction*, Addison-Wesley-Longman, Reading, MA, 1999.

2. The Standish Group, Chaos Report, 1994.

3. D. Leffingwell, IBM Developer Works, www-128.ibm.com/developerworks/rational/library/347.html.

4. Agile Manifesto, agilemanifesto.org; Agile Alliance, www.agilealliance.org.

5. XP Programming.com, An Agile Software Development Resource, www.xprogramming.com; Extreme Programming: A Gentle Introduction, www.extremeprogramming.org.

6. I. Spence, K. Bittner, *What is iterative development – Series*, IBM Developer Works, www-128.ibm.com/developerworks/rational/library/mar05/bittner/index.html.

7. P. Kroll and P. Kruchten, *RUP Made Easy - A Practitioner's Guide*, Addison-Wesley-Longman, Reading, MA, 2003; C. Larman, P. Kruchten, K. Bittner, *How to Fail with the Rational Unified Process: Seven Steps to Pain and Suffering,* www.agilealliance.org/articles/articles/How_to_Fail_with_the_RUP_-_Kruchten_and_Larman.pdf.

8. TheophileEscargot, *Politics-Oriented Software Development*, www.kuro5hin.org/story/2005/1/28/32622/4244

9. A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, Boston, 2000.