

A Persona-Based Approach for Exploring Architecturally Significant Requirements in Agile Projects

Jane Cleland-Huang, Adam Czauderna, and Ed Keenan

DePaul University, Chicago, IL 60422, USA

jhuang@cs.depaul.edu, aczauderna@gmail.com, ekeen2@cdm.depaul.edu

Abstract. [Context and motivation] Architecturally significant requirements (ASRs) drive and constrain many aspects of the architecture. It is therefore beneficial to elicit and analyze these requirements in early phases of a project so that they can be taken into consideration during the architectural design of the system. Unfortunately failure to invest upfront effort in exploring stakeholders quality concerns, can lead to the need for significant refactoring efforts to accommodate emerging requirements. This problem is particularly evident in agile projects which are inherently incremental. [Question/Problem] Existing techniques for early discovery of ASRs, such as Win-Win and i*, are typically rejected by agile development teams as being somewhat heavy-weight. A light-weight approach is therefore needed to help developers identify and explore critical architectural concerns early in the project. [Principal ideas/results] In this paper we present the use of Architecturally-Savvy Personas (ASP-Lite). The personas are used to emerge and analyze stakeholders' quality concerns and to drive and validate the architectural design. ASP-Lite emerged from our experiences working with the requirements and architectural design of the TraceLab project. The approach proved effective for discovering, analyzing, and managing architecturally significant requirements, and then for designing a high-level architectural solution which was designed to satisfy requirements despite significant interdependencies and tradeoffs. [Contributions] This paper presents the ASP-Lite approach and describes its support for architectural design in the US\$2 Million TraceLab project.

Keywords: personas, architecture, requirements, architecturally significant requirements, tradeoffs.

1 Introduction

The overall quality of a software intensive system is measured according to whether the system meets its functional requirements and addresses the underlying quality concerns of its stakeholders. For example, a safety-critical avionics system must guarantee levels of safety through performance and dependability requirements, while a mobile phone service must provide reliable hand-over as

a subscriber moves across various towers, deliver high quality voice and data service, and also provide fast response times for placing calls and sending text messages [18].

The quality requirements for a system represent a special subset of Architecturally Significant Requirements (ASRs) that describe non-behavioral constraints on the system. Yu et al., identified over 100 different types of ASRs [9] including qualities such as reliability, maintainability, safety, usability, portability, and security [4, 12]. ASRs are quite varied in their impact on the system and in the way in which they must be specified. For example, a performance requirement might describe the response time or throughput requirements of a system, while an availability requirement might specify the need for a system to be available 24/7 or to have less than 1 hour of scheduled downtime a week.

ASRs play a strategic role in driving the architectural design of a software intensive system [5, 15] and are often used as a selection criteria for deciding between alternate architectural options [16]. Architects must therefore understand the stakeholders' quality concerns and then utilize technical knowledge to design an architectural solution which balances the potentially complex interdependencies and tradeoffs of the requirements. Techniques such as Architecture Driven Design (ADD) [5] assume a starting point of clearly specified quality concerns documented in the form of Quality Attribute Scenarios, while the Volere approach presents proactive techniques for eliciting and documenting quality concerns [24]. Cohen also describes an agile approach for specifying ASRs (i.e. constraints) in the form of user stories [10]. Despite these techniques many projects fail to adequately explore quality concerns. For example, Franch et al. conducted a survey of ASR elicitation techniques in 13 different software projects [3] and found that in many cases projects did not specify any such requirements. In one documented case, a customer assumed that a web page would take no more than two seconds to load but did not specify this as a requirement. Following deployment he complained that the page loaded too slowly. Identifying ASRs in advance can help to mitigate this type of problem.

1.1 Quality Concerns in Agile Development

The continuous move towards agile development practices highlights the importance of developing light-weight approaches for handling ASRs. Abrahamsson et al. discuss the role of architecture in the agile process [1]. They conclude that "a healthy focus on architecture is not antithetic" to agility, and advocate for finding the architectural sweet spot for a given project so that the emphasis on architecture is customized to the needs of the particular project. Similarly Beck asserted that architecture is just as important in XP (eXtreme Programming) projects as it is in any other project [6]. In an interview conducted for IEEE Software's special edition on the Twin Peaks of Requirements and Architecture [8], Jan Bosch outlines the growing acceptance of designing and constructing a system incrementally, and allowing both functional and non-functional requirements to emerge as the project proceeds. This practice assumes that refactoring the architecture to accommodate newly discovered NFRs is an acceptable cost

of doing business in an agile project, and contrasts with more traditional practices such as ADD [5] and WinWin [7]. In these approaches ASRs are rigorously elicited and analyzed in early phases of the project and then used to drive and evaluate candidate architectural designs. Scott Ambler proposes some kind of middle ground, in which architectures are *sketched* in early phases of the project [2]. Denne and Cleland-Huang describe the incremental funding method (IFM) in which architectures are planned upfront, but then delivered incrementally as needed to support the user-required functionality [13]. This approach has been shown to increase the financial return on investment of the project. Unfortunately the agile mantra of ‘no big upfront design’ is often used to justify a less than effective exploration of the quality requirements during early phases of the software development lifecycle, thereby increasing the likelihood of later costly refactoring efforts.

In this paper we propose a novel approach for capturing and evaluating architecturally significant requirements in agile projects, and then using them to drive and evaluate the architectural design. Our approach, which we call ASP-Lite (Architecturally Savvy Personas - Lite), utilizes HCI personas to express quality concerns in the form of user stories, written from the perspective of specific user groups. While personas have traditionally been used to explore the way individual user groups will interact with a system, ASP-Lite focuses primarily on quality concerns and constraints of the system, and is used to design and evaluate the architectural design. Our approach supports incremental delivery of the architecture, reducing the risk of designing an architecture which requires excessive refactoring.

1.2 Our Proposed Approach

ASP-Lite emerged from our own experiences in the TraceLab project [17], a US\$2 Million endeavor funded by the US National Science Foundation and developed by researchers at DePaul university, the College of William and Mary, Kent State University, and the University of Kentucky. The core part of the project involved developing an experimental environment in which researchers can design experiments using a library of pre-existing and user-defined components, execute their experiments, and then comparatively evaluate results against existing benchmarks.

Early in the project it became apparent that there were some challenging and conflicting quality goals that would impact both the time-to-market and the long-term adoption of the system. To fully explore and understand the impact of early architectural decisions, we developed a set of personas that represented distinct sets of users’ needs, especially those needs which impacted major architectural decisions. The personas were initially developed through a series of brainstorming activities by the core project team. They were presented to collaborators from all participating universities during an initial project launch meeting and refined until all meeting participants were satisfied that the personas provided a realistic and relatively complete representation of TraceLab users’ quality concerns. The

personas were then used throughout the remainder of the project to guide and critically evaluate architectural design decisions.

1.3 Paper Structure

Section 2 of this paper introduces the concept of Architecturally-Savvy personas. Section 3 describes the overall process model integrated into the SCRUM framework. Section 4 describes the personas that we created for the TraceLab project, and then section 5 describes the way in which the personas support the architectural design process. Finally, Section 6 describes the benefits of our approach and reports on our initial findings.

2 Architecturally-Savvy Personas

HCI personas were first introduced by Cooper as a means for integrating user goals and perspectives into the design process [11]. A persona provides a realistic and engaging representation of a specific user group, and is typically depicted through a picture and personal description that portrays something about the psyche, background, emotions and attitudes, and personal traits of the fictitious person [21, 22]. The task of creating a persona usually involves surveying and interviewing users, identifying optimal ways for slicing users into categories, collecting data to demonstrate that the proposed slices create distinguishable user groups, discovering patterns within the user groups, constructing personas for each group, and then creating scenarios describing how the persona might interact with the system under development. A project will typically have from about 5-8 personas.

While personas are typically used for purposes of User-Interaction design, there are a few examples in which they have been used as part of the requirements elicitation process. Dotan et al. evaluated the use of personas to communicate users' goals and preferences to project members as part of a two-day design workshop for the APOSDLE project [14]. Similarly, Robertson et al. also discussed the use of personas for gathering requirements when actual stakeholders are not available [24]. In both cases, the focus was on eliciting a general set of requirements and/or goals.

In contrast, ASP-Lite focuses on architecturally significant requirements. This emphasis impacts the way we slice (or categorize) groups of users in order to create a set of personas whose needs represent distinct sets of quality concerns. In the TraceLab project, many of these competing concerns emerged as a result of an initial Joint Application Design (JAD) session and a series of subsequent brainstorming meetings. As a result, researchers, developers, and architects worked collaboratively to create a small and distinct set of personas and to write a series of architecturally significant user stories for each of them. These user stories focused on qualities such as performance (i.e. how fast?), reliability (i.e. how reliable?), and portability etc.

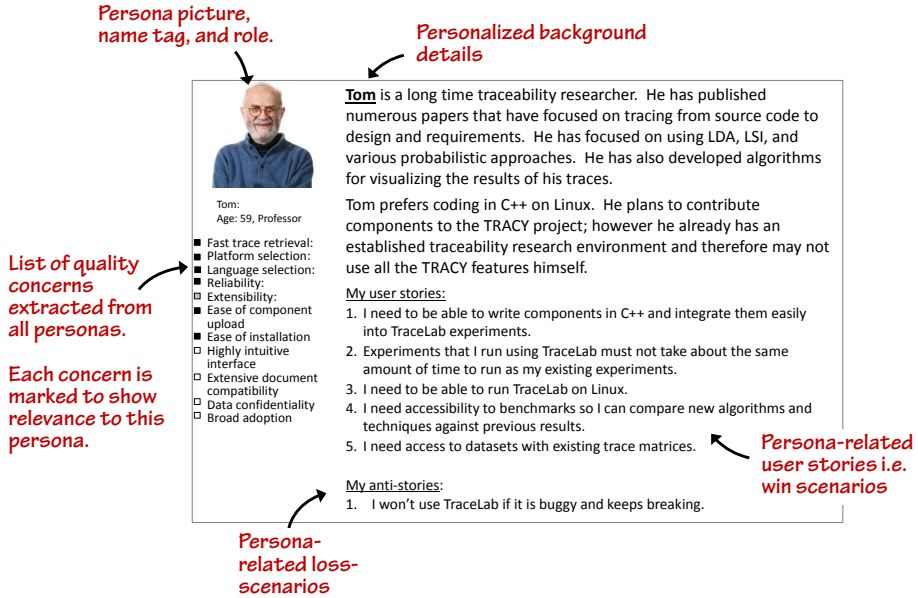


Fig. 1. Light-weight personas used as part of the agile development process to highlight quality concerns (i.e. Non-functional requirements). Personas are used to drive architectural design and to evaluate and validate candidate design solutions.

Once ASR-related user stories were identified for each persona in the project, they were compiled into a project-wide list containing quality concerns from all the personas, and then summarized in a succinct format as shown in the left hand side of Figure 1. A simple classification process was then used to mark each quality concern as *high* (black), *medium* (gray), or *low* (white) importance to each of the personas.

For example, Fig. 1 depicts Tom as a seasoned traceability researcher who has already established an effective research environment on the Linux/C++ platform. His particular concerns include (1) the ability to create components in C++ as this is the language of choice in his research group, (2) the need to run TraceLab on Linux, (3) the need to be able to easily compare results from existing experiments against benchmarks, (4) the ability to retrieve and rerun previously conducted experiments, and finally (5) the need for publicly available data sets. A deeper analysis of his functional requirements leads to the identification of several quality concerns related to language selection, platform selection, and ease of sharing experiments and components across research group boundaries.

3 Architecturally Savvy Personas and SCRUM

ASP-Lite can be used in any development environment; however we describe it within the context of the SCRUM process framework which was the primary

project management process adopted in our project [23]. SCRUM is anchored around the concept of a *sprint*, which typically represents about 2-4 weeks of work, and also daily scrum meetings at which team members meet to assess progress, identify roadblocks, and plan next steps. At the start of each sprint, a set of features are selected from the prioritized list of features in the product backlog and placed into a sprint backlog. Each sprint produces potentially shippable code. In prior work, Madison [19] augmented the SCRUM process by injecting architectural concerns into the backlog so that architectural issues could be addressed incrementally throughout the development process.

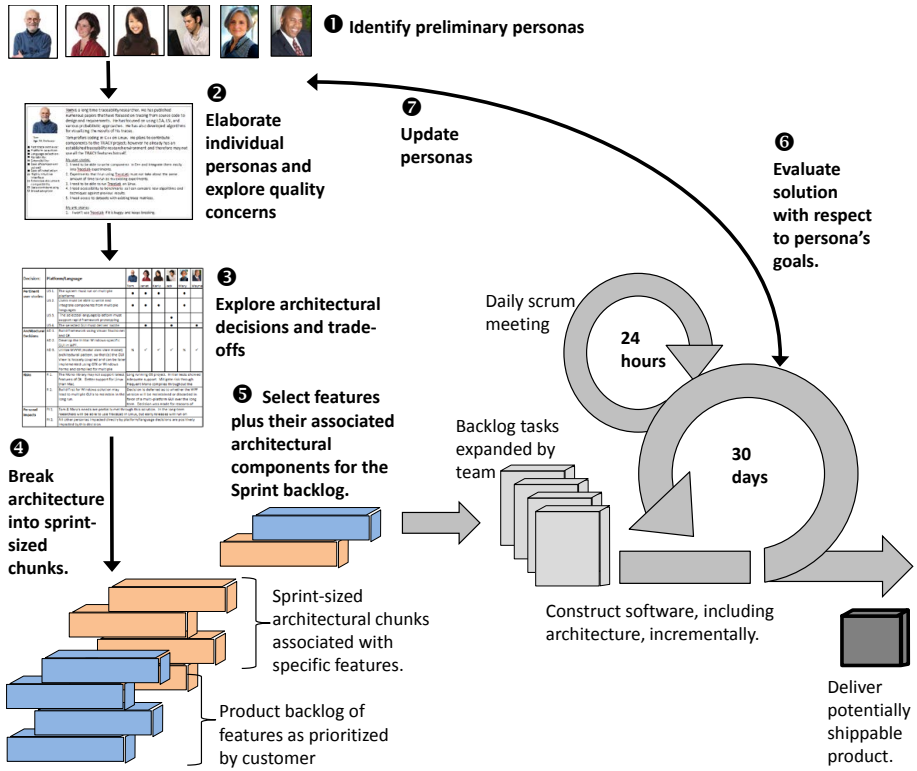


Fig. 2. Personas capturing role-specific quality concerns are used to augment the basic SCRUM life-cycle model in agile projects

ASP-Lite also augments the Scrum process in several important ways, depicted in the steps of Figure 2. First, ① a set of personas are identified and ② fleshed out. They are then used to ③ drive the architectural design and analysis process and then ④ the produced architecture is broken into sprint-sized chunks [20]. The product backlog is then populated with both functional features and architectural elements. ⑤ At the beginning of each sprint, the customer chooses features to implement, and the developers identify the architectural elements

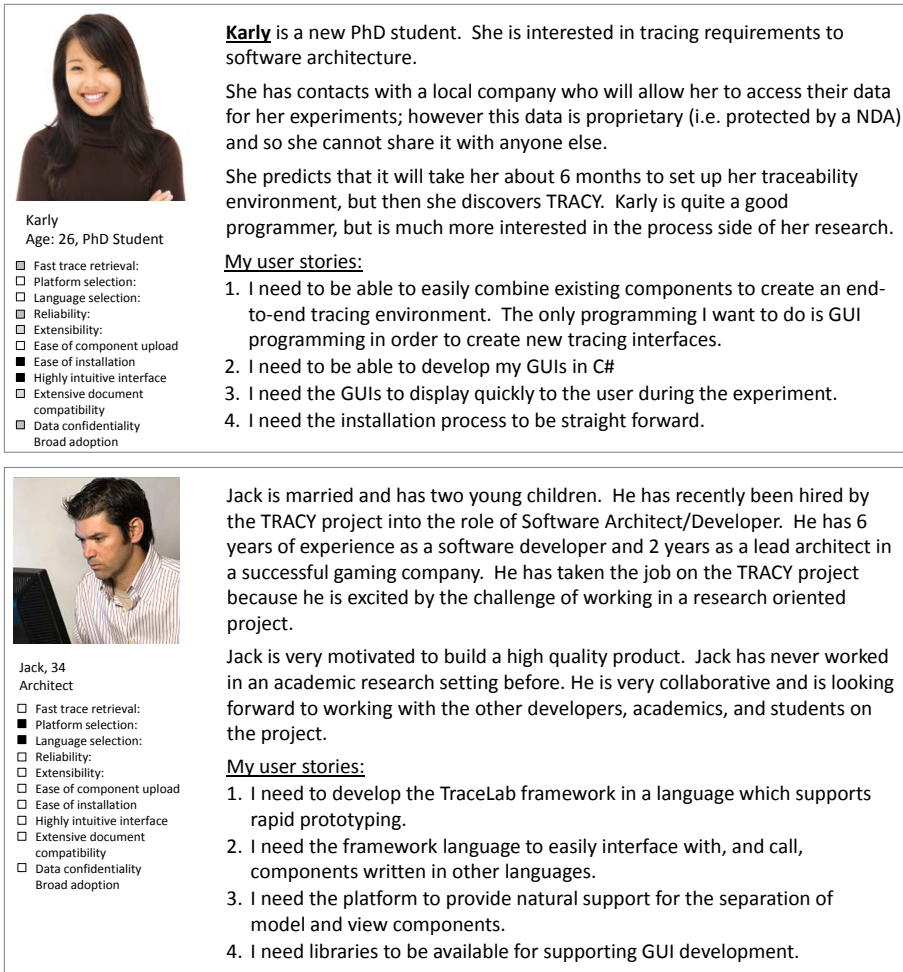


Fig. 3. Two additional personas identified for the TraceLab project

that are needed to implement them. ⑥ Throughout the sprint, the system under development is evaluated with respect to the identified personas and when necessary, i.e. if new functionality is introduced, and further clarification is needed, ⑦ the personas are re-evaluated and modified accordingly.

4 Persona Creation and Use in the TraceLab Project

We created six personas for the TraceLab project. These included “Tom” previously presented in Fig. 1, and five additional personas discussed below:

● **Janet** has a PhD in Human Computer Interaction. She is interested in studying the ways users interact with traceability tools. Her research group develops GUI prototypes in C#. As Janet does not consider herself a programming whizz,

she needs to be able to easily replace or modify one or more GUI screens in an existing tracing workflow so that she can capture user feedback on traces. Furthermore she is likely to abandon TraceLab if she is not able to download, install, and use it with minimal effort.

- **Karly** is a new PhD student. She is interested in tracing requirements to software architecture. Karly has contacts with a local company who will allow her to access their data for her experiments; however this data is proprietary (i.e. protected by a non-disclosure agreement) and so she cannot share it with anyone else. Maintaining full control and confidentiality over her data is essential, and so she is only able to use TraceLab if she can keep the data on her own desktop. Before she discovered TraceLab she had predicted that it would take at least 6 months to set up her traceability environment. Karly is quite a good programmer, but is much more interested in the process side of her research. Karly is depicted in Figure 3.

- **Jack** has recently been hired by the TraceLab project into the role of Software Architect/Developer. He has 6 years of experience as a software developer and 2 years as a lead architect in a successful gaming company. He has taken the job on the project because he is excited by the challenge of working in a research oriented project. Jack is very motivated to build a high quality product. It is critical to Jack that the selected platform and language support rapid prototyping and that libraries are available for developing GUI elements of the design. Jack is also depicted in Figure 3.

- **Mary** is a program director of the funding agency supporting TraceLab. She wants to see a return-on-investment through broad buy-in of TraceLab from the traceability community, reduced investment costs for new traceability research, enabling productivity much earlier in the research cycle, and evidence of broad-ranging support for the most critical areas of Traceability research.

- **Wayne** is the technical manager for a large industrial systems engineering project. He could be described as an early adopter, as he prides himself in keeping an eye out for good ideas that could help his organization. Wayne is concerned that current traceability practices in his organization are costly and inefficient. He has heard about the TraceLab project, and is interested in trying it out, but is concerned about investing time and effort in what appears to be an academic project. He has decided to use TraceLab in a small pilot study to see if it can meet his needs. In particular he needs TraceLab to be installable behind his firewall, configurable to work with his data, almost entirely bug-free, and to provide a professional standard GUI that his workers can use intuitively.

While there is no guarantee that these six personas are complete, they represent a solid starting point for reasoning about the quality concerns of TraceLab's end users.

5 From Personas to Architectural Design

An analysis of the personas' user stories revealed a set of architecturally significant requirements (ASRs) and also some potential conflicts. Specific issues







Decision:	Platform/Language		 Tom	 Janet	 Karly	 Jack	 Mary	 Wayne
Pertinent user stories:	US 1.	The system must run on multiple platforms	●	●	●		●	
	US 2.	Users must be able to write and integrate components from multiple languages	●	●	●		●	
	US 3.	The source language of each component must be invisible at runtime				●		
	US 4.	The selected language/platform must support rapid framework prototyping				●		
	US 5.	The selected GUI must deliver 'razzle dazzle'		●		●		●
Architectural Decisions	AD 1.	Build framework using Visual Studio.net and C#.	½	✓	✓	✓	½	✓
	AD 2.	Develop the initial Windows-specific GUI in WPF.						
	AD 3.	Utilize MVVM (model view view model) architectural pattern, so that (a) the GUI View is loosely coupled and can be later implemented using GTK or Windows Forms and compiled for multiple platforms, and (b) the TraceLab engine can be compiled using Mono for porting to Linux and Mac environments.						
Risks	R 1.	The Mono library may not support latest features of C#. Better support for Linux than Mac.	Long running OS project. Initial tests showed adequate support. Mitigate risk through frequent Mono compiles throughout the project.					
	R 2.	Build first for Windows solution may lead to multiple GUIs to maintain in the long run.	Decision is deferred as to whether the WPF version will be maintained or discarded in favor of a multi-platform GUI over the long term.					
Personal Impacts	PI 1.	Tom & Mary's needs are partially met through this solution. In the long-term researchers will be able to use TraceLab in Linux, but early releases will run on Windows only.						
	PI 2.	All other personas impacted directly by platform/language decisions are positively impacted by this decision.						

Fig. 4. Architecturally significant user stories related to the Platform/Language issue. Subsequent architectural decisions and their impact upon the personas are shown.

related to platform portability, programming language of the framework and components, and the plug-and-play ability of TraceLab were identified. In this section we explore these issues as part of the architectural design process. Our approach loosely follows SEI's Attribute Driven Design (ADD) process [5] which is an incremental scenario-driven approach to design that involves identifying quality attribute scenarios, and then proposing and evaluating candidate architectural solutions. ASP-List captures relevant user stories, architectural decisions, specific risks, and the impact of various decisions upon persona roles in an

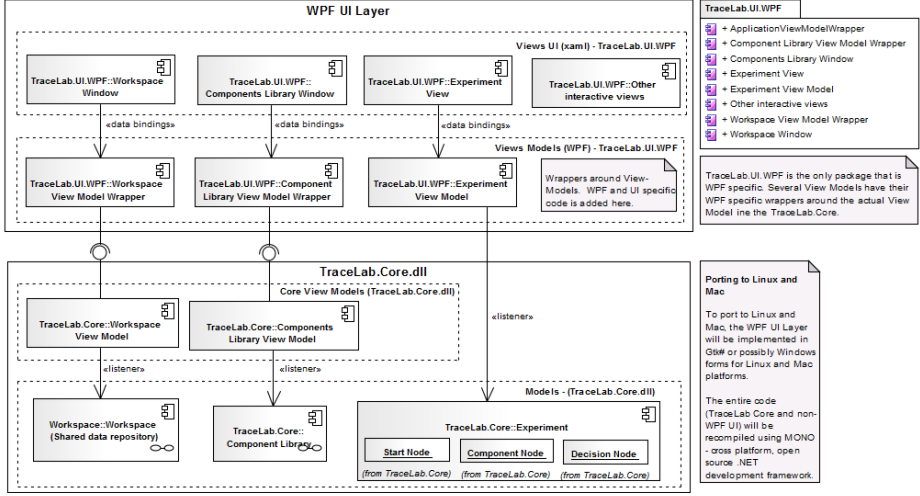


Fig. 5. TraceLab high level architectural design using the MVVM architectural pattern. This architectural diagram captures design decisions related to the platform/language issues that emerged through analyzing persona needs.

architectural issue template, illustrated in Fig. 4. Two examples of architectural issues are provided in the following discussion.

5.1 Architectural Issue # 1: Platform Language Portability

The personas' user stories highlighted the need for TraceLab to run on multiple platforms and to allow components to be developed in a variety of languages and then incorporated into TraceLab's plug-and-play environment at runtime. The following user stories, depicted in Fig. 4, were found to be particularly relevant:

1. The system must run on multiple platforms (*Tom, Janet, Karly, Mary*)
2. Users must be able to integrate components written in a wide variety of languages (*Tom, Janet, Karly, Mary*)
3. The source language of each component should be invisible at runtime (*Jack*)
4. The selected language and platform must support rapid prototyping (Rationale: As a research project we need the freedom to explore variations and to backtrack if and when necessary) (*Jack*)
5. Razzle dazzle (our metaphor for a GUI development environment which provides a high quality presentation to the user) (*Janet, Jack, Wayne*)

Team members met over a period of 2-3 weeks to brainstorm potential architectural solutions for addressing these quality concerns. The extended discussion period was needed to accommodate a series of architectural spikes in which proposed solutions were prototyped and evaluated. Serious consideration was given to three different framework languages: C++ (as this was the preferred language of at least one of our developers, Java (which would be intrinsically portable),

and C# (which from the perspective and experience of the overall development team was the easiest language for development). C++ was discarded due to the learning curve needed by most of the developers and its anticipated lower productivity.

A series of architectural spikes were created to test the benefits of using a C# framework versus a java framework to support the integration of components from multiple source languages. The results from this phase showed that it was far simpler to make calls from C# to components written in other languages, than vice versa, which suggested developing the TraceLab framework in C# and then later compiling it to Mono so that it could run on other platforms. Future portability issues were addressed through a series of architectural decisions. For example, the VisualStudio.net environment provides intrinsic support for the MVVM (model view view model) architectural pattern and integrates closely with WPF. WPF supports rapid prototyping of professional GUIs, while the use of MVVM provides a clear separation between view and model and facilitates future reimplementations in GTK# or Windows Forms for porting to Linux and Mac platforms. Our design also separated out the WPF code in the views layer (which would need to be rewritten for porting purposes) from the non-WPF code which could be compiled using Mono.

The Architectural Issues Template shown in Fig. 6 also documents specific risks and their mitigations. For example, the decision to defer porting to the Linux/Mac environments is potentially impacted by the ability of Mono to compile framework code correctly. This risk was partially mitigated through testing Mono on a variety of projects, and through frequent compiles of the growing TraceLab framework into Mono.

Finally, the proposed architectural decisions were evaluated against the ability of the delivered architecture to meet each of the persona goals. In this case, four of the personas would be fully satisfied with the solution, while Tom and Mary would need to wait until later in the project for the port to Linux and Mac environments. However, this solution was determined to be an acceptable trade-off in light of the tight delivery constraints of the project, the need to build rapid prototypes in order to address the difficulty of potentially changing requirements in such a novel research instrumentation project, and the ease by which C# code was able to invoke components written in other languages.

5.2 Architectural Issue #2: Experimental Workflow

A second major architectural decision pertained to the requirements and design of the TraceLab experiments themselves. These experiments are composed from a series of pre-defined and/or user defined components, and therefore the TraceLab architecture needs to support communication between components and to control their execution. Relevant user stories included the following:

1. Experiments that I run using TraceLab must take about the same amount of time to run as my existing experiments. (*Tom, Janet, Karly*)
2. The TraceLab environment must incorporate plug-and-play. (*Tom, Janet, Karly, Wayne*)







Decision:	Workflow Architecture						
Pertinent user stories:	US 1. The TraceLab environment must support plug and play.	●	●	●			●
	US 2. The performance penalty of using TraceLab must be low (i.e. close to runtime of non-TraceLab experiments).	●	●	●			●
	US 3. Components should be reusable across research groups and experiments.	●				●	
Architectural Decisions	AD 1. Utilize a blackboard architecture.						
	AD 2. Create standard data types for exchanging data between components.						
	AD 3. Construct the experiment around the concept of a workflow.	½	✓	✓			✓
	AD 4. Support concurrent execution of components.						
	AD 5. Trust the TraceLab users to create a viable workflow. Provide basic type checking only.						
Risks	R 1. Performance may suffer as data is exchanged between components via shared memory.	Keep the data cache in the same App space as the experiment to avoid excessive data marshalling . Stream only critical data not entire data structure class.					
	R 2. If TraceLab users proliferate the creation of data types, then plug-and-play ability will be lost.	Use community governance to increase the likelihood of shared use of data types.					
Personal Impacts	PI 1. All personas are satisfied with the plug-and-play solution.						
	PI 2. The performance penalty will be felt more by Tom, as he already has a functioning tracing environment. For other researchers the benefits of the plug-and-play environment and the use of previously defined tracing components far outweighs the slight performance penalty.						

Fig. 6. A second architecturally significant issue related to the way in which components should be built into an experimental workflow

3. Components should be reusable across experiments and research groups. (*Mary, Tom*)
4. Components should run concurrently whenever feasible. (*Tom*)

These user stories and associated architectural decisions are documented in Figure 6. Three different high-level architectural patterns were considered for connecting components in an experiment. A service oriented approach was proposed by an early consultant to the project based on his industrial experience as a SOA architect. However, this option was ruled out because we anticipated that some individual experiments might include over 50 fine-grained components (a supposition which has since proven to be correct). The overhead of calling so many services in a SOA environment was deemed to be prohibitively expensive. The second somewhat intuitive candidate architecture was the pipe-and-filter architectural pattern [5]. However, while this approach seemed to initially fit the concept of data flowing through the experiment, an initial analysis demonstrated that many filters (i.e. components) would in fact be assigned responsibility for the task of transferring data that they did not actually use. While this problem could be partially mitigated by having all components accept a composite message (containing self-describing datasets), this approach has the known flaw of

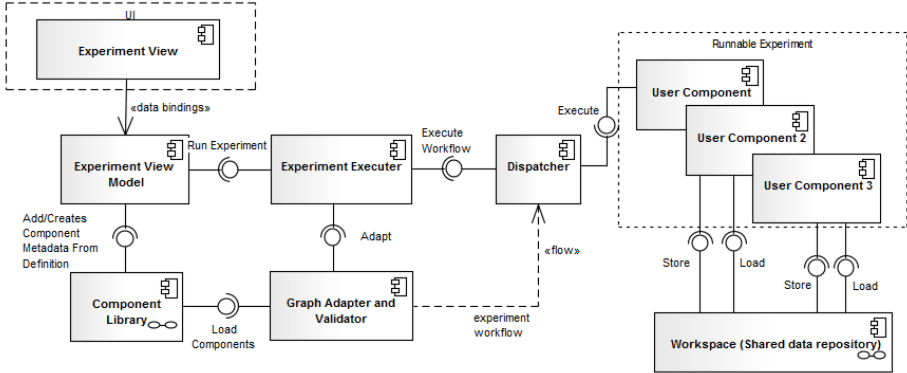


Fig. 7. The Architecture for the TraceLab workflow engine showing the execution control of components and the exchange of data through the blackboard workspace

creating ambiguous interfaces which cannot be understood without looking at the inner workings of the code. Furthermore, this approach would pass on the complexity of handling data typing to the component builders and could result in relatively large amounts of data being passed from one component to another. For these reasons, the pipe-and-filter approach was rejected.

The final architectural pattern we considered, and adopted, was the blackboard architecture. In this approach all data is transferred in standard datatypes, representing domain specific concepts such as trace matrices, artifact collections, and/or metric containers. Each component retrieves a copy of the data from a shared memory space (i.e. the blackboard), processes the data, and then returns the results in standard data formats back to the blackboard for use by other components. The TraceLab experimental graph represents a precedence graph, and the blackboard controller is responsible for dispatching components as soon as their precedence nodes in the graph complete execution. This design supports parallel computation and therefore also addresses performance concerns. In fact, once deployed we found that performance was still below expectations, but were able to modify the data marshaling functions to achieve Tom’s performance goals. Some of the architectural decisions that contributed to satisfying the workflow requirements are shown in Figure 7.

6 Effectiveness of Our Approach

The ASR-aware personas we have created, and the supporting process in which they are deployed, brings several unique contributions to the current state of the art and practice in handling ASRs in agile projects. First, it has shown how personas can be used to capture quality concerns from the perspective of different user groups. Secondly, unlike the previous use of personas which have focused on interaction design and/or requirements elicitation, our approach is designed to drive and evaluate architectural design. Third, our approach introduces a light-weight approach to handling ASRs, which is particularly appropriate in an

agile development process. Not as time-consuming as existing approaches such as Win-Win [7], ADD [5], or i* [26] techniques, our use of personas facilitates a meaningful exploration of the quality concerns of related stakeholder groups. Finally, the *architecture issues template* provides a useful means of exploring architectural options within the context of persona goals, and then of visualizing and documenting the extent to which a set of architectural decisions meets the quality requirements of the system. Like ADD [5], our approach is somewhat ‘greedy’ in nature, as it addresses one set of architectural issues at a time, and then accepts the constraints of that decision upon future decisions. This greedy approach is somewhat softened by the ability to backtrack whenever necessary. This introduces an enormous benefit to the agile process, because it allows backtracking to be performed during the design stage instead of relying upon more expensive backtracking (i.e. refactoring) following deployment.

In comparison to alternate techniques such as Win-Win [7] and i* [26], ASP-Lite facilitates the careful deliberation of competing quality concerns without introducing unnecessary upfront modeling activities. Using either win-win or a modeling approach such as i* would have slowed the project pace in a way that was unacceptable to the project team. In contrast, ASP-Lite fit seamlessly into the adopted agile process, and was perceived by all participants to deliver value to the project.

Formally evaluating a process such as ASP-Lite can be difficult and costly, and can effectively only be accomplished through examining the impact of the new process on practice [25]. For example, our approach could be evaluated by measuring the impact of the process upon the reduction in refactoring costs in a greenfield or even a long-lived project, and through making statistical comparisons between projects that use the approach versus those that do not. However this is an extensive process which can best be performed once a process is adopted in an industrial setting.

On the other hand, the purpose of this experience report is to describe a process which we found anecdotally to be effective in our own development project. The architectural decisions in the TraceLab project were made carefully in light of the persona user stories, and two and a half years into the project, the architectural decisions have all proven to be basically sound. For example, the decision to build first for Windows and then later to port to Linux and Mac environments allowed us to get TraceLab into the hands of our users much faster than would otherwise have been possible. Furthermore, the architectural decisions that were made to separate out WPF code are now paying dividends as we have already compiled TraceLab to run on multiple platforms and are in the process of completing a new multi-platform GUI in GTK#. Similarly, the decision to adopt a blackboard architecture has also proven successful, and early adopters have no problem creating components in multiple languages, integrating them into TraceLab’s plug-and-play environment, and reusing them across experiments.

To evaluate whether ASP-Lite could be generalized to a broader set of systems, we also applied it as a reverse-engineering exercise against an enterprise level Mechatronics traceability project we conducted with a major Systems

Engineering Company. Five distinct personas were identified. Related user stories were written and analyzed which ultimately led to the identification of several architecturally significant requirements pertinent to the design of the system. Interestingly, the ASRs that were identified were significantly different from those discovered for the TraceLab project and focused more upon access control and confidentiality of data as well as usability and performance issues, suggesting that our approach is effective for identifying project-specific quality concerns. Based on this very initial analysis of our approach across an entirely different system, we conclude that it is applicable to a range of projects in which quality concerns need to be explored in a more incremental and agile environment.

7 Conclusions and Future Work

This paper presents an experience report of utilizing architecturally-aware personas within an agile project environment. As such, it provides a viable light-weight solution for addressing quality concerns in agile projects and for potentially reducing the need to refactor later in the project. Based on our initial experiences with ASP-Lite we are currently developing a tool which will make the process more accessible to practitioners.

Acknowledgments. This work was supported by National Science Foundation grants CCF-0959924 and CCF-1265178.

References

1. Abrahamsson, P., Babar, M., Kruchten, P.: Agility and architecture: Can they coexist? *IEEE Software* 27(2), 16–22 (2010)
2. Ambler, S.W.: Agile modeling: A brief overview. In: *pUML*, pp. 7–11 (2001)
3. Ameller, D., Ayala, C.P., Cabot, J., Franch, X.: How do software architects consider non-functional requirements: An exploratory study. In: *RE*, pp. 41–50 (2012)
4. Anton, A.: Goal Identification and Refinement in the Specification of Software-Based Information Systems. Georgia Institute of Technology, Atlanta (1997)
5. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*. Addison Wesley (2003)
6. Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley (2000)
7. Boehm, B.W., Egyed, A., Port, D., Shah, A., Kwan, J., Madachy, R.J.: A stakeholder win-win approach to software engineering education. *Ann. Software Eng.* 6, 295–321 (1998)
8. Bosch, J., Dvorak, D.: Traversing the twin peaks. *IEEE Software* (2012)
9. Chung, L.: *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishers, Norwell (2000)
10. Cohen, M.: Non-functional requirements as user stories. Mountain Goat Software, mountaingoatsoftware.com
11. Cooper, A.: The inmates are running the asylum. *Software-Ergonomie*, 17 (1999)
12. Davis, A.: *Software Requirements - Objects, Functions, and States*. Prentice Hall, Englewood Cliffs (1993)

13. Denne, M., Cleland-Huang, J.: The incremental funding method: Data-driven software development. *IEEE Software* 21(3), 39–47 (2004)
14. Dotan, A., Maiden, N., Lichtner, V., Germanovich, L.: Designing with only four people in mind? – a case study of using personas to redesign a work-integrated learning support system. In: Gross, T., Gulliksen, J., Kotzé, P., Oestreicher, L., Palanque, P., Prates, R.O., Winckler, M. (eds.) *INTERACT 2009*. LNCS, vol. 5727, pp. 497–509. Springer, Heidelberg (2009)
15. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pp. 109–120. IEEE Computer Society Press, Washington, DC (2005)
16. Kazman, R., Klein, M., Clements, P.: *Atam: A method for architecture evaluation*. Software Engineering Institute (2000)
17. Keenan, E., Czauderna, A., Leach, G., Cleland-Huang, J., Shin, Y., Moritz, E., Gethers, M., Poshyvanyk, D., Maletic, J.I., Hayes, J.H., Dekhtyar, A., Manukian, D., Hossein, S., Hearn, D.: Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In: *ICSE*, pp. 1375–1378 (2012)
18. Mirakhorli, M., Cleland-Huang, J.: Tracing Non-Functional Requirements. In: Zisman, A., Cleland-Huang, J., Gotel, O. (eds.) *Software and Systems Traceability*. Springer, Heidelberg (2011)
19. Madison, J.: Agile architecture interactions. *IEEE Software* 27(2), 41–48 (2010)
20. Madison, J.: Agile architecture interactions. *IEEE Software* 27(2), 41–48 (2010)
21. Nielsen, L.: *Personas - User Focused Design*. Human-Computer Interaction Series, vol. 15. Springer (2013)
22. Putnam, C., Kolko, B.E., Wood, S.: Communicating about users in ICTD: leveraging hci personas. In: *ICTD*, pp. 338–349 (2012)
23. Rising, L., Janoff, N.S.: The scrum software development process for small teams. *IEEE Software* 17(4), 26–32 (2000)
24. Robertson, S., Robertson, J.: *Mastering the Requirements Process*. Addison Wesley (2006)
25. Unterkalmsteiner, M., Gorschek, T., Islam, A.K.M.M., Cheng, C.K., Permadi, R.B., Feldt, R.: Evaluation and measurement of software process improvement - a systematic literature review. *IEEE Trans. Software Eng.* 38(2), 398–424 (2012)
26. Yu, E.S.K.: Social modeling and i*. In: *Conceptual Modeling: Foundations and Applications*, pp. 99–121 (2009)