Supporting Software Architecture Learning Using Runtime Visualization

John C. Georgas
Informatics and Computing Program
Northern Arizona University
Flagstaff, Arizona 86011
Email: john.georgas@nau.edu

James D. Palmer and Michael J. McCormick
Department of Electrical Engineering and Computer Science
Northern Arizona University
Flagstaff, Arizona 86011
Email: {james.palmer, michael.mccormick}@nau.edu

***Abstract*—Static diagrams are the most prevalent artifact used in visualizing component-and-connector architectures and supporting software architecture learning. The use of such artifacts exhibits a fundamental disconnect from the dynamic nature of software systems, deemphasizes the importance of architectural interactions with a focus on static structure, and does not support a high degree of learner engagement. This paper presents our work in addressing these challenges by developing runtime visualization techniques that better support learning through the use of visual vocabularies that leverage insights from computer graphics and human perception. We also present evaluative data drawn from user studies and associated insights, which provide positive indicators that our work is effective in supporting our target learning outcomes.**

***Keywords*-software architecture; runtime; computer science education**

I. Introduction

Design pervades the entire range of artifacts and layers of abstraction that software engineers work with and has a fundamental impact on the functional and non-functional properties of software systems. The area of software architecture, concerned with understanding and capturing the modular structure of software systems, has emerged as an important focal point for the design activity in software engineering, which informs the importance of effectively supporting software architecture learning.

Architectural models—along with techniques for their representation and visualization—are key learning artifacts in supporting software architecture learning. In the pedagogical context of component-and-connector architectures and architectural style learning, canonical software architecture depictions exhibit a fundamental drawback, namely the use of static representations for inherently dynamic software processes. In response, instructors typically center discussions with learners on how an architecture or architectural style behaves on static architectural depictions with *ad hoc* annotations for representing dynamic interactions and how these interactions change over time. This is particularly problematic for architectural style learning, as static models place undue attention on the structure of architectures rather than the interactions between elements that are critical in shaping the behavior of software systems. Finally, we feel that static models are not particularly effective in supporting a high level of learner interest and engagement.

In order to address these challenges, we are developing methods and tools that infuse dynamism in supporting software architecture learning. Our work is fundamentally focused on exploring novel visualization techniques that we aim to be engaging to learners and better align with the dynamic nature of software systems. Given the importance of communication in supporting architectural learning, our current focus is on visualizing inter-component communication rather than an exhaustive set of architectural characteristics. We achieve this by instrumenting running software systems and exposing their architectural behavior through real-time, interactive visualizations. Our work goes beyond decorating static visualizations of architectures or animating existing predominantly-static visualizations or viewpoints by leveraging graphics and human perception insights and using color, animation, shape, and spatial relationships to improve software architecture learning.

This paper is focused on describing our overall approach to supporting software architecture learning as well as the results from two studies involving learners from our undergraduate computer science program and contributes: (a) an overview of two distinct visual vocabularies for depicting runtime architectural behaviors; (b) details on two user studies involving our target learner population; and (c) lessons drawn from analyzing evaluative data and how these lessons inform our continued work in this area.

II. Background and Related Work

Our approach is broadly informed by learning theories, software architecture and architectural styles, runtime architectural visualization, and work in computer graphics and human perception.

A. Learning Theories

The pedagogical approach embodied by our focus on real-time visualization is centered on insights from constructivism [1], which focuses on the importance of learner interactions with each other and their environment to support knowledge gains. Our visualization-based perspective on supporting learning adopts this view by allowing learners to become active in creating hypotheses about *in situ* systems and actively

examining and interacting with both the running software and its associated architectural visualization. Naturally, this broad theory provides the foundation for more focused approaches: Situated learning [2] stresses the importance of providing a close match between the learning environment and the context in which knowledge will be applied, which resonates with our focus on linking software architecture instruction and learning to operational software systems. The interactive nature of our work, which has learners explore the runtime architectural characteristics of software behaviors, also draws from discovery-based learning [3]. These modes of learning support a high degree of learner attention and engagement, as does the high degree of variation provided by dynamic visualization—attention and engagement are identified in the ARCS motivational model [4] as key strategies through which to foster learner motivation and learning achievement.

B. Software Architecture and Modeling

The study of software architecture is underpinned by graphical and non-graphical models that capture the principal design decisions of a software system [5], particularly those relating to decomposing software systems into interconnected components. Our work is particularly concerned with a *component-and-connector* (C&C) view of software, with components capturing computational tasks associated with a system and connectors focused on facilitating component communication [6]. Patterns of composition for particular functional and non-functional characteristics are codified into architectural styles [7]. Architectural styles reinforce the importance of considering and representing connectors as first-class entities, which makes a C&C view of software architecture with first-class connectors ideal for the pedagogical context of our work as opposed to, for instance, an object-oriented view.

Modeling notations for C&C architectures fundamentally shape the manner in which users perceive and interact with software architectures, and vary widely in the level of detail they expose, the degree to which they leverage textual and graphical elements, and the sophistication of their associated toolsets [5]. Canonical visualizations associated with well-established architecture description languages (ADLs) [8] capturing C&C views are primarily static in nature and do not inherently support a visual vocabulary that incorporates runtime characteristics of a system's operation, which is a primary focus of our work. While the Unified Modeling Language (UML) [9] contains models useful for capturing behavioral aspects of software systems, the models themselves are also primarily static without special consideration of dynamism in the visual vocabulary they use. Furthermore, UML exhibits important drawbacks in the context of C&C architectures [10] particularly in (a) allowing ambiguity through multiple valid methods of representing elements of such architectures and (b) in overloading object-oriented visual design elements for C&C models; these drawbacks are particularly problematic in an instructional context.

C. Runtime Visualization

Related work using xADL-based [11] depictions in the field of architectural visualization provides augmentations for conventional depictions of architecture while linked to runtime systems [12], much like our approach. Other work using custom notations [13] only somewhat integrates dynamism in its depictions and does not strongly represent C&C architectural elements. While this work satisfies some of our goals, it does not explore the pedagogical utility of visual vocabularies that adopt architectural dynamism by design.

Other related work explores the graphical representation of runtime software system characteristics: HotAgent [14] visualizes component interactions with an emphasis on a three-dimensional visual vocabulary. Jinsight [15] focuses on exposing the lifetime of objects in memory and object communication in multithreaded Java programs and uses custom visualizations for that purpose. Other work focuses on creating UML visualizations based on trace information collected from a target system [16] and relies on storing information during a system's execution to be visualized later. Javavis [17] shows how object and sequence diagrams change during runtime by using breakpoints, which overrides the system's native timing between activities. Our work is differentiated by a focus on C&C architectural views, which are particularly well-suited to the study of architectural styles, and by better bridging the temporal disconnect between visualization and the runtime behavior of target systems by focusing on runtime visualization.

D. Graphics and Perception

Our approach is also informed by work in studies of human perception, particularly Gestalt theory and motion perception, as they pertain to information visualization [18]. Gestalt theory is based on a number of principles used by the human visual system to comprehend patterns in images: We leverage the principle of proximity, for example, to lead learners to associate objects that appear closer together as more tightly associated.

Other visualization decisions have been influenced by the field of information design including Tufte's critiques and recommendations for the representation of graphical data [19], Cleveland's graphical specifier hierarchy [20], and a host of other work in information visualization including taxonomic approaches for transforming data, analytical abstractions, and visualization abstractions [21].

III. Approach

Our fundamental research goal is to support improvements in C&C architecture and architectural style learning by developing a new approach and accompanying toolset to support runtime architectural visualization of target systems that goes beyond augmentations of conventional architectural depictions. More specifically, by adopting visual vocabularies that embrace the inherently dynamic nature of software and an interactive toolset for student use we aim to improve learning outcomes across a number of important software

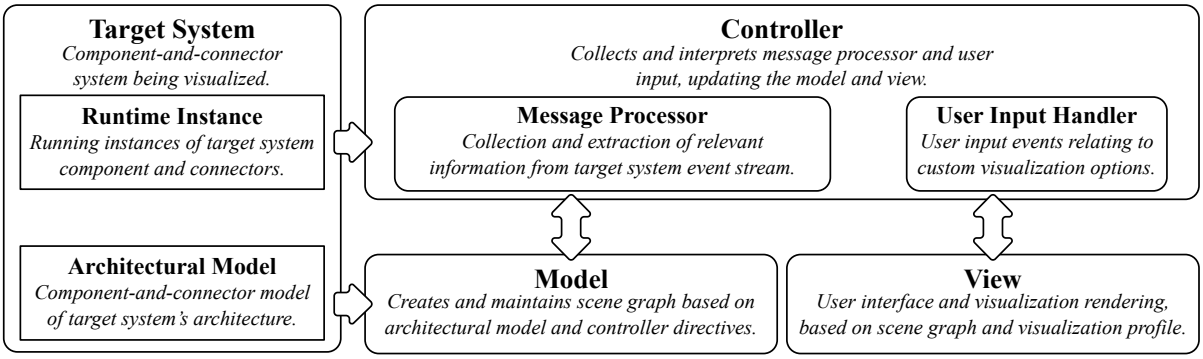

Fig. 1. High-level design of our toolset for runtime architectural visualization.

system characteristics that are not strongly supported through the instructional use of conventional architectural depictions:

- Investigate and interpret associations between externally observable software system behaviors and architectural element activities that drive such behaviors;
- Compare architectural elements and their relative activity, identifying their relative centrality to a software system's operation; and
- Identify and evaluate dynamic shifts in architectural element interactions over time.

A. Instructional Context and Motivation

Our work is primarily grounded in the context provided by our university's course in software architecture, delivered in a 16-week semester format. The course, intended for juniors and seniors in computer science, broadly addresses topics in software architecture with an emphasis on reusable design elements, namely C&C architectural styles and object-oriented design patterns. Specific topics covered in this course include fundamental architectural concepts and elements, multi-viewpoint modeling, applications of software architecture methodologies for developing software systems of various scales, architectural styles, first-class connectors and middleware, architectural modeling notations and ADLs, product-line architectures, architectural visualization, object-oriented modeling and design patterns, adaptive architectures, and domain-specific architectures. The module on architectural styles supports substantial coverage of basic and derived architectural patterns and styles, including the layered, data-flow, shared memory, implicit invocation, and hybrid families of architectural styles. A series of projects provide learners with the opportunity to reverse engineer and document the architecture of an open-source system, re-design this architecture according to the guidelines of a variety of architectural styles, and formally capture architectures using a variety of modeling approaches and languages.

The emphasis this course places on architectural styles is a key motivator for our work, as it brings to the foreground the fundamental mismatch between the static nature of canonical software architecture depictions and the dynamic and changing interactions at the heart of architectural styles. While behavioral models of systems, such as those found as part of UML, provide a link to the runtime behavior of software systems exemplifying a particular style, they are often not appropriate for modeling software architectures at the C&C level and they fall short of providing the engaging connection to a running software system that our work aims to support. As a result, our instructional methods to date have focused on introducing active learning activities [22]–[24], *ad hoc* annotated diagrams based on static architectural depictions to capture dynamic interactions and how these interactions change over time, and instrumented example systems for architectural styles of interest that provide log information on software events. Our work on runtime visualization is aimed at providing artifacts and tools for supporting software architecture and architectural style learning that embrace dynamism by design.

B. Toolset Implementation

Our visualization toolset is developed in the ActionScript 3 language [25] and uses the Adobe Flash Starling framework [26] that provides a graphics processing unit accelerated two-dimensional scene graph application program interface accessible from ActionScript. The toolset architecture, outlined in Figure 1, follows the model-view-controller pattern with the dominating software concern being the maintenance of a two-dimensional scene graph capturing the visualization's state. Currently our work is built to support target systems built using the C2 framework [27], with information about the system collected by leveraging instrumented connectors that forward runtime events to our visualization toolset.

C. Dynamic Visual Vocabularies

Developing appropriate visual vocabularies, i.e. coherent sets of symbols used to visually depict related concepts, with features appropriate for runtime visualization of software architectures is fundamental to achieving our research goal of supporting learning improvements. Our work has thus far been focused on developing and deploying two such visual vocabularies, a *tethered-entity* and a *layer-based* visualization, with details on each appearing in the following sections.

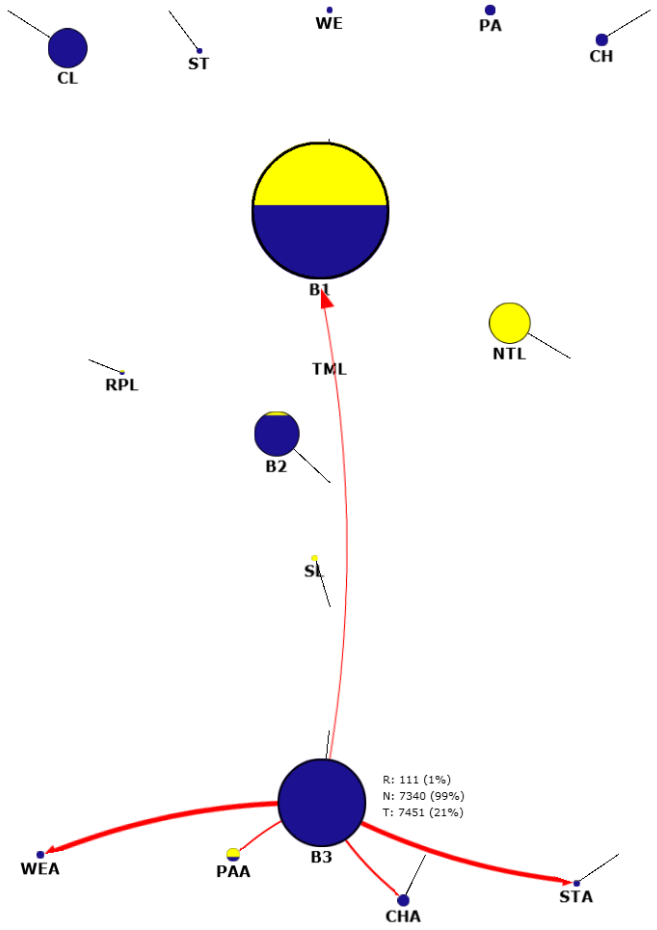

Fig. 2. In this tethered-entity visualization example, a user has clicked the B3 element to show extended connectivity information.

1) Tethered-Entity: The tethered-entity visualization, an example of which appears in Figure 2, uses labeled circle primitives to represent C&C architectural elements with initial positions that minimize connection crossings and make sure that component layers are preserved. Animations and visual attributes of these primitives are used to represent dynamic features of the runtime architecture:

- *Size*, relating the event activity of an architectural entity relative to others in the architecture;
- *Notification color*, shading the proportion of notification messages received by an architectural entity;
- *Request color*, shading the proportion of request messages sent by an architectural element;
- *Position*, relating architectural communication by adjusting the position of architectural entities relative to others in the architecture; and
- *Path*, defining line segments from the initial position of an architectural entity to its present position.

The area of a circle represents the number of messages sent from the entity relative to the number of messages sent from all entities, with an entity exhibiting its minimum size of a point if it sends no messages and its maximum size if it is the only entity sending out messages. In a system where many entities may be sending messages, the size of each entity is related to the overall number of actively communicating architectural entities, so circle area is a visual indicator of activity with larger circles representing elements that communicate more than entities represented with smaller circles.

Circles are colored with a notification color (blue) and a request color (yellow) and are shaded with each color proportional to the number of notification messages and request messages sent. In the example shown, these colors are vertically split to convey the idea of event directionality, which is a key aspect of the C2 style. For example, in Figure 2, the element labeled "NTL" is communicating requests almost exclusively, "B3" is communicating notifications almost exclusively, and "B1" is communicating an even mix of both.

The position of a circle is based on its communication with other architectural elements, so the line segment formed from an element's original and current position is an indicator of not just activity (captured by its length) but also what elements that activity is primarily associated with (captured by its direction). While the scene graph renders at approximately 60 frames-per-second, position calculations are expanded to consider a two-second window that helps emphasize short bursts of messages. The displacement of a circle from its initial position is then computed based on the current position of other components and weighted by the amount of overall system communication. The displacement magnitude is scaled to prevent degenerate animation results by applying the inverse tangent function to the number of messages, which emphasizes large fractional changes in the number of messages and dampens the effect of messages for already large message sets.

Interactive attributes provide learners with a mechanism to explore features not directly related in the primary visual vocabulary that only appear when elements are clicked. These properties include:

- *Request count summary*, which provides specific statistics on the number and percentage of requests sent per architectural entity;
- *Notification count summary*, which provides specific statistics on the number and percentage of notifications sent per architectural entity;
- *Message count summary*, which provides specific statistics on the number of messages sent per entity; and
- *Arrow links*, which capture the directionality of messages exchanged between entities within a small time window with a scaled thickness proportional to the number of overall messages exchanged.

Within the visualization toolset, learners can click on a circle or label to get this more detailed message flow information in the form of text boxes and arrows. For example, Figure 2 illustrates how these annotations appear when the "B3" entity is clicked, showing arrows with scaled thicknesses based on the strength of communication with connected elements and a text box showing summary information.

2) Layer-based: The layer-based visualization, an example of which appears in Figure 3, uses a variety of visual attributes

to represent static and dynamic features of the architecture with an emphasis on the layered arrangement of architectural elements in the architecture.

The following attributes relate to static features of the architecture:

- *Rectangular widgets*, which correspond to components;
- *Layer numbers*, which indicate the vertical position of components in a static architectural arrangement; and
- *Solid lines*, which indicate how connectors are linked to components and other connectors.

The initial positions of these visual elements are based on the layered arrangement of architectural elements in the architectural description. Connectors are represented by solid lines connecting widget layers. If a component is linked to a connector above it in the architecture diagram, then the corresponding component widget is adjacent to a solid line above it in the visualization. A similar concept applies to components with links below them in the architecture diagram. The semi-circle arcs on the right side of the visualization capture links between connectors.

This visualization uses a collection of visual attributes to represent dynamic features of the architecture with each of these updated during runtime:

- *Bar height*, which indicates the number of messages sent from a component;
- *Bar color*, which indicates the type of messages sent from a component;
- *Text display*, which indicates the number of messages sent from a component in text format; and
- *Rectangular widget horizontal position*, which indicates the ranking of components in a layer according the number of messages of a certain type sent by those components.

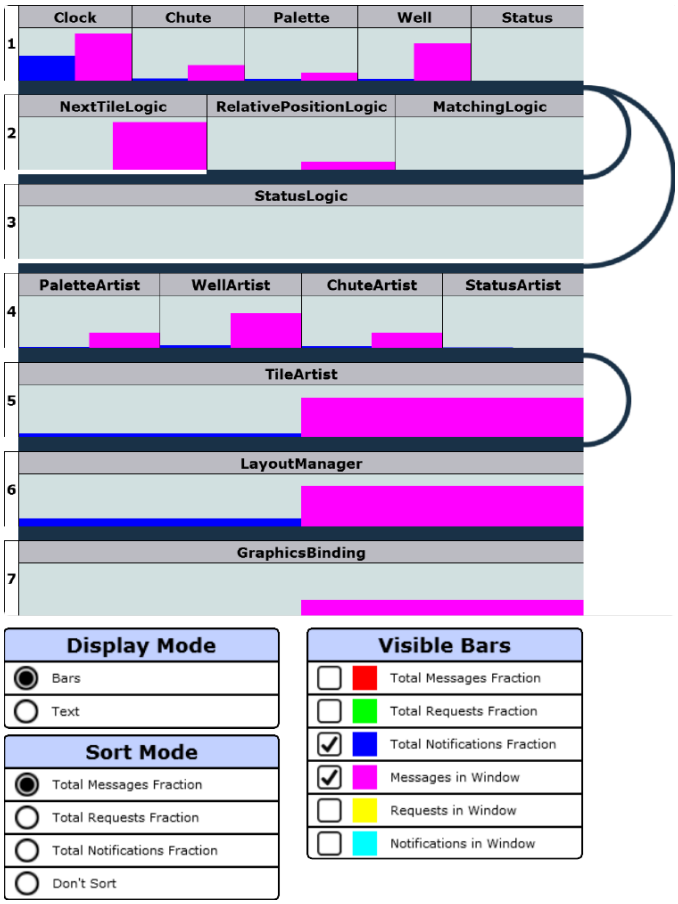

Fig. 3. In this layer-based visualization example, a user has selected to display additional information on the fraction of total notifications sent from each component and total number of messages sent in a two-second time window.

Each rectangular widget can contain up to six colored bars below the component label. The height of a bar from the bottom of the widget is based on the number of messages of a certain type sent from the component within some amount of time. The width of each widget is evenly divided among the widgets in a layer, and the bar widths are evenly divided among bars in a widget.

There are two main types of bars: fraction bars and window bars, which are distinguished by the amount of time that they keep track of messages sent. For fraction bars, the fractional height corresponds to the ratio of messages of a certain type sent from the component to the number of messages of that type sent from all components since the beginning of system execution. A fraction bar for requests has no height if it has sent no requests and achieves the maximum height if it is the only component that has sent requests. For window bars, the bar height corresponds to a function of the number of messages of a certain type sent from the component in a two-second time window. This function is based on the inverse tangent of the number of messages, which limits the maximum height of the bar and emphasizes large fractional message changes.

Optionally, the horizontal position of each widget within each layer can change based on sorting criteria such as the total number of requests, notifications, or requests plus notifications sent from components since the beginning of the visualization. The widgets animate their position changes every second so that they are sorted from left to right within a layer based on the sorting criteria.

Interacting with the layer-based visualization is accomplished through the following controls:

- *Pan and zoom*, which allows the user to isolate and center certain parts of the visualization;
- *Display mode*, which allows the user to change between the bar view and text view for widgets;
- *Sort mode*, which allows the user to change the sort method for widgets within each layer;
- *Visible bars*, which allows the user to select which bars should be visible in the widget bar view; and
- *Layer view*, which allows the user to show or hide a summary view of information in a particular layer.

IV. Deployment and Evaluation

We deployed our toolset with students from our target learner population in the context of two studies that allowed us

to qualitatively and quantitatively evaluate our work in runtime architectural visualization and its utility as an instructional tool. One study focused on gathering formative feedback on early work with the tethered-entity visualization as compared to static architectural depictions, with the second larger-scale study focusing on a comparison between the use of dynamic event log messages and the layer-based visualization.

Both studies addressed the fundamental research question driving our work by focusing on core dimensions of interest: the usefulness of runtime visualization for learner ability to identify, describe, and compare architectural behaviors and their changes over time. Additionally, we examined learner perceptions regarding the value of runtime visualization in architecture learning. Overall, the results provide strong positive indicators that our work is effective and useful in supporting architectural learning and is perceived as helpful to the learning process by our students.

A. Klax

In an effort to provide an engaging context for participating students, both studies used a C&C-centric port of the Klax video game developed as part of the C2 project [27]. From a pedagogical perspective, this interactive video game is engaging and exhibits a rich set of architectural behaviors that are linked to user interface elements and externally observable behaviors. The game involves matching colored tiles: Tiles drop through a chute at the top of the screen, are caught by players by moving a palette in the middle, and are dropped into wells at the bottom of the screen. The object of the game is to not miss any dropped tiles while also matching tiles of the same color in the wells. The C2 implementation of this game exhibits a layered architecture, with all communication facilitated through explicit connectors and components arranged in groups that handle game state, game logic, and the game's user interface.

B. Case Study: Tethered-Entity Evaluation

Our aim in this study was to provide a foundation for better understanding how the tethered-entity visualization compares to conventional static means of understanding a software system's architecture, namely through an examination of the system's implementation and a static visualization of its C&C structure.

A focus group discussion was central to this study, so we kept the participant group small. The user group consisted of eight students randomly selected from a set of 18 volunteers all drawn from our undergraduate computer science program. Five participants were male and three were female. Five participants were in their third year of study in the program and were enrolled in an introductory course in software engineering that includes a two week module in software architecture at the time of the activity. Three participants were in their fourth year of study and, in addition to previous study in the context of a software engineering course, had completed a semester-long course specifically devoted to software architecture. Participants were randomly assigned into pairs for this study to better distribute varying levels of architectural expertise.

1) Study Protocol: Each participant pair was provided the following artifacts and tools: (a) a running instance of the Klax game; (b) our visualization toolset with a tethered-entity visualization active and linked to the Klax game; (c) an instance of the Eclipse development environment[1] with the complete Java-based source code of the Klax game; and (d) a static visualization of the C&C architecture of the Klax game.

The study started with participants having 30 minutes to use provided artifacts and answer worksheet-guided questions about the Klax system and its runtime behavior. Participants were free to use any of the artifacts available to them, as we wanted to allow them the freedom to gravitate toward whatever artifact they felt was most appropriate. The worksheet was aimed to provide a structured experience for learners, as might be provided to them as an assignment in an instructional context, and asked them questions pertaining to dynamic runtime behaviors of the Klax game and relative architectural element activity (e.g. "which entity sends the most messages overall?") and connections between game events and architectural activity (e.g. "which entities send messages when a tile lands on the palette?").

Participants then completed an anonymous nine-question Likert-type questionnaire: The first part of this instrument asked participants to assess their perceived learning gains from the experience of using our runtime architectural visualization toolset and is based on the Student Assessment of their Learning Gains (SALG) methodology [28]. The second part asked participants to gauge their perceptions regarding the usefulness of the visualization as compared to an examination of the target system's static architectural visualization and source code in answering worksheet questions. All participant pairs completed this instrument and—to minimize social desirability bias—study moderators were not present while participants completed this questionnaire.

The study concluded with a 40-minute open focus group discussion with participants. While the discussion was allowed to be open-ended, moderator prompts were aimed at eliciting feedback on the usefulness of core visualization design decisions (discussed in Section III-C1) in exposing static and dynamic aspects of software architecture.

2) Study Results: The SALG-oriented questionnaire posed questions aimed at gathering indicators as to the perceived benefits of engaging in the worksheet-guided exploration of runtime visualization for learners: Q1 through Q3 addressed their perceptions of whether the activity enhanced their understanding of Klax, the C2 style, and software architecture in general. Q4 through Q6 probed at perceptions regarding their perceptions on increased understanding of matches and connections between the Klax source code, the game's architecture, and the game's runtime behaviors. Finally, Q7 through Q9 asked them to rate the usefulness of Klax's static architectural depiction, the game's source code, and the

[1] www.eclipse.org

TABLE I
PARTICIPANT RESPONSES TO A QUESTIONNAIRE FOCUSED ON PERCEPTIONS OF LEARNING GAINS THROUGH THE USE OF RUNTIME ARCHITECTURAL VISUALIZATION.

	Group Identifier				
Question	**G1**	**G2**	**G3**	**G4**	**Average**
Q1: *Understanding of Klax*	3	4	4	4	3.75
Q2: *Understanding of the C2 style*	5	5	4	3	4.25
Q3: *Understanding of software architecture*	3	4	4	5	4.00
Q4: *Match between Klax implementation and architecture*	3	5	4	1	3.25
Q5: *Match between Klax architecture and runtime*	4	5	4	2	3.75
Q6: *Match between Klax implementation and runtime*	2	1	2	1	1.50
Q7: *Usefulness of static depiction in worksheet*	2	3	5	3	3.25
Q8: *Usefulness of Klax implementation in worksheet*	1	1	2	1	1.25
Q9: *Usefulness of tethered-entitity visualization in worksheet*	4	5	4	4	4.25

tethered-entity visualization in answering worksheet questions. Table I truncates the full questions that participants saw—Q4, for example, was phrased as "How much did the experience help to improve your understanding of how the Klax code matches up with the Klax architecture?" The instrument uses a five-point scale, with a score of one indicating that students perceived the activity as being "no help" and five being perceived as "very good help."

Questionnaire data indicates that participants found the experience of using our visualization tool useful in improving their understanding of Klax, software architecture, and C2, with all groups scoring Q1 through Q3 above the scale midpoint. Notably, the highest average rating among participants was assigned to the perceived usefulness of our toolset in supporting their understanding of the C2 architectural style, with styles being a critical element of software architecture learning. Participants also perceived the experience as enhancing their understanding of the connections between source code and architecture and architecture and running system. Given our work's focus on software architecture and design-level elements, it is not surprising that participants felt our toolset was not very helpful in understanding how runtime behaviors relate to source code.

Most importantly, we note that participants strongly identified our runtime visualization toolset as the most useful artifact in answering their worksheet questions. Our worksheet was oriented toward exploring runtime behaviors, so this result provides a positive indicator that our work is perceived by learners as effective and useful in supporting learning on dynamic aspects of software architecture.

The focus group discussion concluding this study suggested that our approach to runtime visualization is perceived as valuable by learners as a way to convey architectural information. A number of comments, such as "[t]here's a lot of connections being made, and you can't really see that through the [static architectural depiction]," focused on the utility of a runtime view of architecture and how this view had higher utility than either the static architectural depiction or the target system's source code in understanding the behavior of the system during runtime. Participants also indicated that using entity size as a visual vocabulary element was a good technique for capturing differences in the number of messages an architectural element is sending and that animating an element's position was a good technique for indicating the directionality of message flows. More importantly, the combination of these visual elements was perceived as being particularly effective, with one student noting that "it was helpful for me to look at the buses in terms of the size of the circles to show the traffic going between components and then just looking at the components just to see what direction they were changing to." Participants also suggested that adding end-to-end tracking, pause, playback, and frame tracing would be very helpful in providing the means through which to address more fine-grained and detailed questions about the target system's runtime operation.

C. Case Study: Layer-based

One of the simplest and most common techniques for augmenting instruction with information on the runtime behavior of a software architecture is the use of log messages that accompany component and connector actions and information exchanges. Our aim in the activity described in this section was to explore how runtime architecture visualization compares to this use of log messages for architectural behaviors and to enable more concrete conclusions by expanding the sample size of study participants. The central activity of this study was a randomized order 15-question worksheet that asked participants questions about runtime characteristics of the target system.

The user group for this study consisted of 51 student volunteers drawn from our undergraduate computer science program. Of these participants, 46 were male and five were female. For this work, we drew on our sophomore population, with 36 student participants enrolled in our data structures course—a course that immediately follows our two-course introduction to programming and computer science and precedes our software engineering and software architecture courses—at the time of this study and the remaining 15 students having already completed the course.

Participants were randomly assigned to the control or experimental groups, with the control group comprising 21 of our participants. The experimental group was further sub-divided into two groups, with one group using the tethered-entity visualization and the other using the layer-based visualization. Our data analysis thus far and the focus of this discussion

will center on a comparison between the control group and the experimental group using the layer-based visualization, which comprised ten of our participants. In this subgroup, nine participants were male and one female, with seven enrolled in our data structures course and three already having completed the course.

1) Study Protocol: Each group's activities started with a ten-minute review of event-based systems, the C2 architectural style, and Klax. We added this instructional element in this study to account for the fact that, given their sophomore status, participants lacked in-depth exposure to software architecture concepts. This review included an overview of the artifacts available to them while working through the study's worksheet. Participants had up to 50 minutes during which to complete this worksheet, which was built using SurveyGizmo[2], and recorded both participant answers and timing information about their responses—correct responses for each worksheet question were collaboratively determined by the authors who are experts in software architecture and the implementation of Klax used in the study.

Each participant in the control and experimental groups was provided with the following artifacts and tools: (a) a running instance of the Klax game and (b) a static depiction of Klax's C&C component and connector architecture. The control group was also provided with (a) an instance of the Eclipse development environment with the complete Java-based source code of the target system and (b) a text-based display within the Eclipse environment displaying the type of message and the names of the sending and receiving architectural elements for each event during the game's runtime operation. Instead of the source code and text-based display, experimental group participants were provided with our visualization toolset with a layer-based visualization active and linked to the running instance of Klax. Control group materials were intended as an approximation of a low-overhead approach that an instructor might reach for to infuse an in-class presentation of an architecture with runtime information rather than only rely on static architectural depictions—a technique we have previously used in our own instructional activities.

2) Study Results: The questions contained in the worksheet were designed to be answerable by all participants in both the control and experimental groups, since our focus was to examine differences in performance based on the set of available artifacts. Each worksheet question appeared in a random order for each participant to better isolate any training effect dependencies between questions. Worksheet questions were equally categorized into three general types:

- *Entity comparison*, comprised of questions that asked participants to rank specific components or connectors according to the relative number of messages they sent (e.g., "Rank components in layer 2 from highest (rank of 1) to lowest (rank of 3) number of messages sent.");
- *Entity identification*, which asked participants to indicate which components or connectors are sending messages or have sent messages during certain game states (e.g., "Identify the component that only sends out messages while clicking the arrow keys."); and
- *Game event identification*, which ask participants to indicate which game events are occurring while certain components or connectors send messages (e.g., "Identify the game event that occurs when the Relative Position Logic component sends notifications.").

[2]www.surveygizmo.com

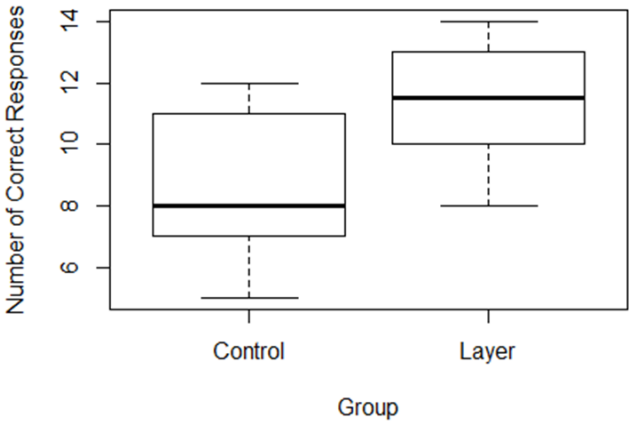

Fig. 4. Box plot comparing correct responses for the control and layer-based experimental groups.

Our principal goal was to determine whether our toolset enables higher learning, as determined by assessing differences in the number of correct responses to the runtime questions for the control and experimental groups and differences in how much time was spent on each question. A box plot of the number of correct responses for each group is shown in Figure 4. After accounting for the division into sub-groups within our experimental group, we found strong positive evidence of the merits of our approach to runtime visualization.

Our data set did not meet the assumptions of equal variance and normally distributed error terms that are necessary for running an ANOVA test. Additionally, we observed a number of low scores that pulled the mean number of correct answers down significantly for groups that did not have the layer-based visualization. In this context, we elected to focus on the median as the best measure of the center value for the number of correct responses by group, since the median value is less affected by outlier values and perform permutation-based t-tests to assess differences in median score between groups.

More specifically, we performed a one-sided hypothesis test to see if the median score for the layer-based visualization group was higher than the median score for the control group with a significance level of $\alpha = 0.05$. Since the median total score for the control group was 8 and the median total score for the layer group was 11.5, we sampled 10,000 permutations to find the probability of observing a difference of 3.5 or greater in the median score for a sample, assuming there is no difference in the population median score. After the Bonferroni correction was applied to address the problem of multiple

comparisons, we found a $p = 0.03$. This p-value suggests there is statistically significant evidence that the median score for the layer visualization group was higher than the median score for the control group, indicating that our layer-based visualization better supports learners when examining runtime characteristics of software architecture.

The range and spread of the scores also provide strong evidence of the effectiveness of the layer-based visualization. The highest score in the layer-based experimental group was 14, and was higher than the highest score in the control group, which was 12. This suggests that the layer-based visualization provides increased opportunity to build better runtime architectural comprehension. The lowest score for the layer-based group at 8 was higher than the lowest score for the control group, which was 5. In fact, no participant in the layer-based group scored below the median score of the control group. Looking at the spread of the data, the scores are much tighter for the layer-based experimental group than the control group—the interquartile range for the layer group was 3, while the interquartile range for the control group was 4. These characteristics suggest that there is significant utility by visualizing runtime architectural information using the layer-based visual vocabulary as compared to examining raw event log information.

We also analyzed the total amount of time participants used to complete worksheet questions for each group. The median completion time for the control group was 1026 seconds, while the median completion time for the layer group was 1249 seconds. By performing another permutation-based test with a Bonferroni correction, we did not find statistically significant evidence for differences between the median completion times for each group. In practice, however, we note that the question of time investment may require additional investigation as there are more interaction features provided with the layer-based visualization, giving users more options when exploring runtime architectural characteristics.

V. Discussion and Future Work

Results from our evaluative studies provide significant positive indicators that validate our overall approach and reinforce important visual vocabulary design decisions. However, our work with target population learners also brings to the foreground a number of interesting research questions and future work directions. In addition to introducing visual refinements, such as shading and textures that accommodate color-blind users, important issues relate to developing more sophisticated visual vocabularies, identifying sudden or subtle changes during runtime visualization, and incorporating complex event dependency analysis.

A. Designing Visual Vocabularies

Our work with learners provides positive indicators that our focus on using animation as a key element of visual vocabularies is an effective way to depict runtime architectural behaviors. Study participants pointed out that animated relative sizes of architectural elements were particularly effective in depicting differences and changes in architectural entity activity levels. In addition to validating a core design decision, this also motivates further investigation into other visual techniques that use animation to capture architectural activity and inter-component communication. One such future direction we are investigating, for example, is to animate component communication patterns by depicting sets of components merging together into aggregate graphical entities should they almost exclusively communicate with each other and drifting back apart once more when this relationship no longer holds, i.e. logically-dependent "blobs."

B. Infrequent Events

Student participants also pointed out that some changes in the visualization can be difficult to notice, with frequent and intense architectural activity visually overwhelming infrequent and transient activity. In the Klax system, for example, certain high-level system events such as tile matching happen infrequently and only trigger a limited number of events over a short time window. Architectural entities involved in these activities are animated only for a short time, which may make noticing these activities challenging.

In order to compensate, participants in our studies described a number of techniques: One student commented that "you've got to do it like multiple times in order to see where the spikes are occurring," indicating an iterative approach where behaviors of interest can be repeatedly triggered so that the architecture can be observed. Other students pointed out that the static architectural diagram can be used in a complementary way, commenting that "we couldn't tell from just the visualization itself, so we kind of made our own hypothesis based on the architecture and then looked for any drastic changes between what the possibilities could be and then looked at it through the visualization."

While the learners that worked with us found ways to identify these infrequent architectural events, this challenge motivates an investigation of visual elements aimed at providing a more effective depiction of short-lived but possibly important architectural behaviors. Techniques to mitigate this challenge might include adding pause-playback functionality or the addition of graphical trails to the movement of visual vocabulary elements, so that both the current and recent past position is visible for each element—this, of course, introduces the possibility of overloading the visualization with too much information to remain an effective communication tool.

C. Event Dependencies

Participating learners indicated that they sometimes had difficulty in determining causality when faced with software behaviors that inherently involve multiple architectural events over a short amount of time. The tight timing between the visual representation of these behaviors makes it more difficult to decompose the aggregate visual behavior in a way that allows event dependency inferences. Without limiting our toolset to architectural models that explicitly capture and represent event dependencies, we are investigating introducing

complex event processing [29] into our visualizations that can better identify—and therefore visualize—complex event chains. Given our focus on runtime visualization, this also motivates considering probabilistic models that can build event dependency inferences as the system operates and the available data set grows.

VI. Conclusion

In the context of supporting software architecture learning, particularly for C&C architectures and architectural styles, conventional instructional methods do not adequately capture the dynamism inherent to the runtime operation of software systems. Our work is aimed to address this challenge by providing runtime visualizations that learners can use to actively explore the association between architectural events and externally observable system behaviors, compare architectural elements and their relative activity, and evaluate dynamic runtime shifts in architectural element interactions over time.

Evaluative studies and activities with students provide strongly positive indicators that our visualization toolset is effective at allowing learners to more accurately identify runtime architectural characteristics, when compared to conventional instructional methods. Furthermore, learners themselves perceive their experience with our toolset as beneficial to their learning gains, particularly regarding concepts relating to architectural styles and the dependencies between a system's architecture and its runtime behavior.

Our future work targets refinements in our currently used visual vocabularies while also exploring novel vocabularies that incorporate additional concepts from graphics and perception. We are also investigating techniques that more clearly depict infrequent and sudden architectural events and incorporate runtime complex event processing for visualizing and supporting learning on event dependencies.

Acknowledgment

The authors are grateful to participating Northern Arizona University computer science students. This research is supported in part by the National Science Foundation under Grant number DUE-1245427.

References

[1] L. Vygotsky, *Mind In Society: The Development of Higher Psychological Processes*. Cambridge, MA, USA: Harvard University Press, 1978.

[2] J. Lave, *Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life*. Cambridge, UK: Cambridge University Press, 1988.

[3] M. Prensky, *Digital Game-Based Learning*. New York, NY, USA: McGraw-Hill, 2001.

[4] J. Keller and K. Suzuki, "Use of the ARCS Motivation Model in Courseware Design," in *Instructional Designs for Microcomputer Courseware*, D. Jonassen, Ed. Hillsdale, NJ, USA: Lawrence Erlbaum, 1988.

[5] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[6] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[7] M. Shaw and P. Clements, "A field guide to boxology: Preliminary classification of architectural styles for software systems," in *Computer Software and Applications Conference*, August 1997, pp. 6–13.

[8] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[9] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd ed. Reading, MA, USA: Addison-Wesley Professional, 2005.

[10] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva, "Documenting Component and Connector Views with UML 2.0," Carnegie Mellon University/Software Engineering Institute, Tech. Rep. CMU/SEI-2004-TR-008, 2004.

[11] E. Dashofy, A. van der Hoek, and R. Taylor, "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 2, pp. 199–245, April 2005.

[12] J. Ren and R. N. Taylor, "Visualizing Software Architecture with Off-The-Shelf Components," in *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, July 2003, pp. 132–141.

[13] J. Grundy and J. Hosking, "High-level static and dynamic visualisation of software architectures," in *Symposium on Visual Languages*. Seattle, WA: IEEE, September 2000, pp. 5–12.

[14] L. Martin, A. Giesl, and J. Martin, "Dynamic component program visualization," in *Proceedings of the Ninth Working Conference on Reverse Engineering*. IEEE, 2002, pp. 289–298.

[15] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the Execution of Java Programs," in *Revised Lectures on Software Visualization, International Seminar*. London, UK: Springer-Verlag, 2002, pp. 151–162.

[16] T. Systä, K. Koskimies, and H. Müller, "Shimba—an Environment for Reverse Engineering Java Software Systems," *Software: Practice & Experience*, vol. 31, no. 4, pp. 371–394, April 2001.

[17] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in *Software Visualization*, ser. Lecture Notes in Computer Science, S. Diehl, Ed. Springer Berlin Heidelberg, 2002, vol. 2269, pp. 176–190.

[18] C. Ware, *Information Visualization: Perception for Design*, 3rd ed. Waltham, MA, USA: Elsevier, 2012.

[19] E. R. Tufte, *The Visual Display of Quantitative Information*. Cheshire, CT, USA: Graphics Press, 1986.

[20] W. S. Cleveland and R. McGill, "Graphical Perception and Graphical Methods for Analyzing Scientific Data," *Science*, vol. 229, pp. 828–833, 1985.

[21] E. H. Chi, "A Taxonomy of Visualization Techniques Using the Data State Reference Model," in *Proceedings of the IEEE Symposium on Information Vizualization 2000*, ser. INFOVIS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 69–75.

[22] J. C. Georgas, "Supporting Software Architectural Style Education Using Active Learning and Role-playing," in *Proceedings of the American Society for Engineering Education Annual Conference & Exposition (ASEE 2013)*, Atlanta, GA, USA, June 2013.

[23] T. V. Wilkins and J. C. Georgas, "Drawing insight from student perceptions of reflective design learning," in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, Florence, Italy, May 16-24 2015.

[24] J. C. Georgas, "Teams battling teams: Introducing software engineering education in the first-year with robocode," in *Proceedings of the 118th American Society for Engineering Education Annual Conference & Exposition (ASEE 2011)*, Vancouver, BC, June 26-29 2011.

[25] Adobe, "ActionScript 3.0 Reference for the Adobe Flash Platform," http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html. [Online]. Available: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html

[26] Gamua, "Starling—The Cross Platform Game Engine," http://gamua.com/starling/.

[27] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow, "A component-and message-based architectural style for GUI software," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[28] E. Seymour, D. Wiese, and A. Hunter, "Creating a better mousetrap: Online student assessment of their learning goals," in *National Meetings of the American Chemical Society*, San Francisco, CA, USA, March 2000.

[29] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley, 2001.