Automated QA Testing at EA: Driven by Events

A discussion with Michael Donat, Jafar Husain, and Terry Coatta

To millions of game geeks, the position of QA (quality assurance) tester at Electronic Arts must seem like a dream job. But from the company's perspective, the overhead associated with QA can look downright frightening, particularly in an era of massively multiplayer games.

Hence the appeal of automated QA testing, which has the potential to be faster, more cost-effective, more efficient, and more scalable than manual testing. While automation cannot mimic everything human testers can do, it can be very useful for many types of basic testing. Still, it turns out the transition to automated testing is not nearly as straightforward as it might at first appear. Some of the thorniest challenges are considered here.

At EA, Michael Donat is an advocate of automation. His current focus is process improvement on the Player and Business Analysis team. He was previously the manager of QA at Silicon Chalk and ActiveState Corp., and has worked at Microsoft as a software design engineer.

Joining the discussion is Jafar Husain, a lead software developer for Netflix. Previously he worked at Microsoft, where one of his tasks involved creating the test environment for the Silverlight development platform. There he was introduced to MVVM (Model View ViewModel); he's a convert, he says, and now likes to spread the gospel of MVVM where applicable.

Terry Coatta, a member of the ACM Queue board, brought this group together to discuss the potential for automated QA testing. He and Donat once worked together at Silicon Chalk, where creating a sophisticated test environment was among their challenges. Coatta is now the CTO of Marine Learning Systems, developing a learning management system for marine workers.

TERRY COATTA In terms of your efforts so far to apply automated QA testing at EA, I gather you've found the going a little bumpy.

MICHAEL DONAT We started the journey thinking automation was a good idea, but then we tried it, and it failed. Still, we figured out what was wrong, and we fixed it. But, while we made it to a nice plateau, we realized there was still a long way to go. Our solution clearly wasn't going to get us everything we wanted—which was a way to broadly apply automated testing. To get there, and for some other reasons, several of us have concluded that what we really need is a new architecture along the lines of MVVM.

JAFAR HUSAIN What exactly was your driver for automating in the first place?

MD Our primary motivation had to do purely with the cost of manual testing, which has become quite significant given the complexity of our games. Basically, code changes that require us to retest everything manually can be incredibly expensive. By reducing those costs, we felt we would have an opportunity to redirect our testers away from what I call “stability testing”—which is something automation is capable of handling—so they can start focusing more on the authenticity and fun of our game experience.

TC In terms of stability testing, what did you see as your first opportunities for automation?

MD We started looking at this seriously when we were working on EA Sports' FIFA 10 [soccer game].

Initially, that involved 10 vs. 10 gameplay, which then became 11 vs. 11 with the addition of goalies. So we needed that many testers—either 20 or 22. But that’s not all, since we also needed to test for interactions between different matches to make sure the server wasn’t getting confused about what information to send to which match. So, in addition to the testers required to cover one match, we needed to have at least one other match in play at the same time—meaning we actually needed to have 40 or so testers involved at the same time.

Then, even after we’d managed to get everyone organized, we might end up running into some trivial bug just seconds into the match that would bring the whole thing down. Besides being wasteful, that was extremely frustrating for a lot of people who could have been doing something more productive during that time. All that came together to make a pretty strong argument for automation.

TC What were some of the problems you encountered as you worked toward that end?

MD First, setting up an OTP (online team play) match in FIFA 10 required the user to go through a few screens. There were 20 consoles and the script was time-based, meaning it sent commands to the consoles and then waited for some prescribed amount of time for all of them to get into the right state. Then it would send out the next batch of commands. The goal was to move the consoles in lockstep through a set of screen transitions in order to set things up for gameplay: participants chose which side they wanted to play, what jersey they wanted to wear, what position they wanted to play, and various other parameters. All those things needed to happen in concert just to keep the programming for the game as simple as possible.

At the time, our primitive test-automation system made navigating the front end problematic. Timing had to be just right, or tests would fail. As a result, I began advocating for a means of skipping the front end altogether, but I was forced to change my point of view. During manual testing of FIFA 10 OTP, a number of issues came up—so many, in fact, that the budget for manual testing had to be increased significantly. The question around the organization was, “How can we stop this from happening in the future?”

That led me to analyze roughly 300 crash bugs for which we had obtained data in the QA cycle. Part of my goal was to see whether there was any significant ROI to be realized by continuing to pursue automation. I found that slightly more than half of our crash bugs were actually coming up in those initial screen transitions. It turned out I’d been telling the games developers exactly the wrong thing. That is, we really did need to do testing on the front end, as well as on the back end. We couldn’t make automation simpler just by getting rid of the front end.

TC That’s interesting. It seems like all that’s happening on the front end is that people are choosing things from menus, so how likely are you to find bugs there? In reality, what looks like a simple process of choosing menu items actually amounts to a distributed computation. You’ve got 20 different things going on, with input coming from all these different places, and now all of that has to be coordinated.

MD Exactly. It became clear we needed a different mechanism altogether. Just sending control inputs wasn’t going to be enough. We needed the test program to be aware of where it was on a particular console and then be able to move that forward in an error-correctable way.

The guys who had originally put together the test-automation framework for FIFA had realized this would be necessary, but the capability for handling it had rotted over the years and didn’t really exist by the time we were ready to tackle FIFA 11. So, one of the things we had to do was get the details we needed to see coming out of the UI so we’d be able to tell where things actually were.

JH I guess that instead of driving things from the view layer—that is, going through the controller and the views—you needed to bypass that view and go directly to the model itself.

MD Believe it or not, we were not at that stage yet. At that point, we were just happy to have scripts that were far more reliable, simply because they knew where they were in the state of the program.

TC That way, you could actually close the feedback loop. Before that, you would send a command and then have to wait and trust in God that something wasn't going to happen in the meantime, whereas now you don't need to have that trust since you can verify instead.

MD Right. We got to where we had more of a controlled state transition. Another big QA improvement we made on FIFA 11 was the addition of Auto Assist, whereby automation could be left to run the game itself while one or two manual testers drove the actual gameplay by supplying controller inputs for selected consoles. They didn't need to have 20 people on hand. That represented a huge improvement.

TC Some people might have just rested on their laurels at that point.

MD Maybe, but it was just one step for me. While introducing some test automation to specific applications like FIFA OTP is a wonderful thing, what I really want is a much broader application for stability purposes because that's what will make it possible for us to focus our testers on the overall game experience. That's the way to go about building a superior product.

The work on FIFA 11 helped convince EA of the potential benefits of automated testing, but accomplishing that end was clearly going to require a different architecture. The answer seemed to lie with the MVVM paradigm, an architectural pattern largely based on MVC. MVVM facilitates a clear separation between the development of the graphical user interface and the development of the back-end logic, meaning it should allow EA to separate OTP gameplay testing from UI testing.

TC Looking back on where things stood once you'd finished with FIFA 11 test automation, what did you see as your next steps?

MD As encouraging as FIFA 11 proved to be, the problem was that we had to spend a ton of time coding. Mostly that's because during game development, changes frequently would be made to certain screens, and then we would have to make corresponding changes in our test-automation script. That wasn't always properly coordinated, so things would end up breaking. As a result, we had a very fragile test-automation script that required a software engineer virtually dedicated to working on maintenance.

In the case of FIFA 11 OTP, that expense was justified, but I couldn't make the case for applying a similar level of test-automation effort across every other area of the game. We had to continue relying on a large number of manual testers to cover the full breadth of testing. Which made it pretty obvious we needed to figure out a way to encode our tests so that ongoing maintenance could be performed less often, using less expensive resources.

TC And that led you where exactly?

MD Basically, it meant the architecture would need to change. It should be easy to see how the game is laid out in terms of its screen transitions, but there should also be ready access to the data those screens act upon. In general, things should just be more accessible at a higher level of abstraction than is currently the case.

JH Is it fair to say you would like to focus on workflows independent of the actual UI controls?

MD That's absolutely right. Once that became clear, we realized we needed a different architecture—something more like MVVM. That isn't to say it has to be MVVM; it just needs to be something that can provide that sort of capability.

TC What is it about the MVVM paradigm that's important?

MD Essentially, it allows us to separate the data used by the screens from the screens themselves. We need that separation so automation systems can gain access to the things they need to tie into.

JH It might be useful to contrast the MVVM approach with other patterns many developers might be more familiar with—MVC, for example. In an MVC architecture, both the controller and the view know about each other and send messages to each other directly. In an MVVM architecture, instead of a controller, you have a view model, which is just that—a model of the view. The view model stores the state of the view, and the view object decides how the state of the view model ought to be presented.

Unlike in the MVC pattern, the view model has no direct knowledge of the view. Instead of sending messages to the view directly, the view model updates the view indirectly via the observer pattern. When the view model is changed, it broadcasts those changes, and the view responds by updating itself. The main advantage of this is that it's possible to test that the view models are in the correct state without even instantiating the view objects, which would add many asynchronous operations (usually related to rendering) that in turn would have to be coordinated.

Testing new models this way is easy since your models expose methods that can be directly invoked. Testing logic through the view layer is much more prone to error since it requires waiting for buttons to load and relies on the delivery of brittle messages such as simulated mouse clicks and key events.

Anyway, as you've moved beyond FIFA 11, what additional steps have you taken toward an MVVM sort of architecture?

MD I should point out that improved test automation is only one benefit of MVVM. Several other groups at EA are also moving this way for a variety of reasons. The steps we've taken so far have mostly been to make the separation of the data from the screens more apparent. Unfortunately, FIFA has so many screens that we can't just go in and rewrite everything. What we can do, however, is to work the new paradigm into new features.

JH It's interesting that, in the face of so many challenges, you've chosen to evolve your architecture in this stepwise manner toward MVVM. It seems you've found it easier just to add new events or extra components that follow this new pattern and then start using those as you can. I presume that at some point the plan is to make a more wholesale transition to MVVM—or something like it—as that opportunity presents itself.

MD That is the plan because it's the only way we can actually go about it. It's going to be a while before we can achieve the full breadth of automation I'm pushing for, but at least we're moving in the right direction.

Our next challenge is figuring out how to specify our tests, since we now have an architecture that lets us access that stuff. But we still don't know what those tests ought to look like, how they should be packaged, or how to contain the information such that it's easy to maintain and makes sense to the people who maintain it.

TC What's the pushback on arguments for an MVVM-like environment? Are people afraid the transition would be too hard?

MD There's no doubt it would be hard. What makes it worse is that the software engineers would have to make those changes in lieu of adding some new features—which can be a very difficult sacrifice to justify. I can't even say exactly how much they would be able to save as a consequence of automation. The truth is that they probably wouldn't save all that much since we're just talking about moving manual resources from one kind of testing to another.

TC Do you think it would actually be more expensive to build in MVVM? Or is this really more about resistance on the part of the software engineers to making any changes to the way they're accustomed to working?

MD That depends on the underlying code involved. Also, we sometimes make incremental changes to existing features. That is, we sometimes need to rewrite features because they need to evolve beyond the original design. If we're about to rewrite a feature anyway, that certainly presents an opportunity to take the newer approach.

On the other hand, if we're putting in a new twist for an existing game mode or adding a small feature to something that's already there, it would be very difficult to do that the new way while all that old stuff is still around. That would only make those incremental changes all the more expensive.

JH It seems that, in order to get to a place where you've actually got something useful, you're going to need to move an entire workflow to MVVM. I suppose that's going to be difficult to accomplish incrementally.

MD That's right.

JH We've run into this at Netflix. So I think you've touched on something that's worth pointing out—namely, that it's one thing to have two different but similar libraries in a code base, while it's quite another to have two different paradigms within the same code base. When you're in that situation, it can be very difficult for onboarding developers to figure out exactly what to do. Have you found this to be a stumbling block? And has that caused any friction?

MD Absolutely. There are many FIFA developers all over the world, so the idea of unifying all of them in support of moving in more of an MVVM direction is pretty hard to imagine.

JH I wonder if the current attitude of those developers toward MVVM reflects the fact that the benefits you're touting will only be realized downstream. Beyond that, though, are they also aware that MVVM might be a better architecture in general for development, quite apart from any testing benefits?

MD Actually, I've been really impressed with the software engineers I've worked with here. They all seem to know what the right thing to do is. But time is also an issue.

JH Is it fair to say the developers don't have any objection to MVVM, and might even be very much in favor of making the necessary changes to use MVVM?

MD Often, I'll be talking to a group of game developers about some idea and they'll say, "Oh yeah, we already know we should go that route," but when it comes to implementation, they aren't able to follow through because of time constraints.

JH In terms of how you move forward, I gather you still have some questions regarding the architecture and that you're also still trying to figure out what the API for your testers ought to look like.

MD That's right, although I'd put the emphasis on specification rather than API, because programming is expensive. We're trying to determine how we can specify these things in such a

way that they'll be understandable, maintainable, and robust in the face of change. That is, in its purest form, you'd like to run an OTP test where you have 22 consoles, with 11 being assigned to one team and the other 11 going to the other side, along with the ability to associate all appropriate parameters with each.

Then the question becomes: how can you specify that in such a way as to cover a broad range of tests? And that's, of course, because each time you run a test, you would like to be able to do different things. If you've got a multiple-match situation, for example, you might want to roll through all the different teams, stadiums, and jerseys so that over the course of many weeks of testing, you would wind up cycling through as many different combinations as possible—and all of that by essentially specifying only one test. That's our goal, anyway, but it's not entirely clear at this point how we're going to manage that.

JH There really are two questions here: (1) Is it possible? (2) Does it scale? There are also some more advanced approaches you could use to build asynchronous tests, but would those then be accessible to junior developers or test engineers?

MD Right. There's no point in doing this unless we can do it in a low-cost manner.

The transition to automated testing has a significant cost dimension when it comes to the use of human resources. First, software engineers accustomed to doing things one way must be convinced to change. They must learn a new paradigm, move from synchronous to asynchronous programming, and perhaps even learn a DSL (domain-specific language) for writing event-based tests.

Second, it's essential to strike the right balance between the work done by lower-cost QA testers and that which is reserved for higher-paid specialists. This means taking advantage of the asynchronous nature of the game by emphasizing declarative tests that are started and gated by events, while designing tests orchestrated to play off those events. This could allow for large numbers of inexpensive coders to write the declarative tests, while a much more select set of expensive coders are left to focus on the more sophisticated orchestration issues.

JH Have you explored different languages that might make it easier for lower-skilled developers to write event-based tests?

MD I've been considering the possibility of using a DSL. What worries me, however, is that there was a time when we had to encode game information in the test code, and I'm afraid we might end up going back to encoding information in some other type of code if we were to choose the wrong DSL.

One of the properties of the DSL we'd be looking to use is a container for the game information that needs to be transparent enough so people can easily access that information. It's important the information can be accessed using vocabulary that both the QA people and the game producers are familiar with.

JH Understood. The line between where a DSL begins and a library ends can be somewhat blurry. But a DSL can also be embedded as part of the general-purpose programming language you already use.

MD At the moment I don't think we're really going to be looking at any if-then-else loop coding. We're probably talking only about tests at the level of stimulus and response—that is, "When the program responds in this particular way, then provide this sort of stimulus."

TC Jafar, have you had any experience with DSLs at Netflix?

JH We're actually currently struggling to make a similar transition—not so much for testing but

more for finding a better way to coordinate the concurrency in our application. What we're using for that now is the Rx (Reactive Extensions) library. The interesting thing about this library is that it has actually been integrated into C#, meaning you can use it at a much higher level than if-then-else to coordinate asynchronous processes. There's also a JavaScript version of Reactive Extensions, which is what we're using now.

While this should make things very easy, in practice we're finding that it's a very new way of thinking for developers—particularly those who have come from a background of if-then-else, imperative, top-down programming. And this is despite the fact that the Rx abstraction is at a much higher level and is, in fact, quite declarative and obviously flexible, powerful, and capable enough to handle all sorts of complex asynchronous operations. It's not so much a matter of this new language being any more or any less difficult to work with; it's just that when you come from a synchronous way of thinking, making the transition to programming asynchronously can be very challenging.

Asynchronous programming requires a significant investment in terms of learning something new and a whole different way of thinking about your code. Which is to say I'm skeptical you'll manage to find a DSL out there that can transform a synchronous programmer into an asynchronous programmer in a few weeks, or even over the course of a product cycle.

MD That's my fear as well. There's going to be a need for people in the loop who are skilled in asynchronous programming. Whoever is coding up these exotic OTP tests where we have two or three matches going on at the same time is definitely going to need those skills.

But the open question for me is: How can you do that and still have the QA people specify most of the tests? It would be fantastic if we could just get to the point where 80 percent of the game code could be covered by tests written by the QA people. And then if the other 20 percent of the OTP tests had to be written by highly paid specialists, so be it. I would be cool with that just so long as we could get a large proportion of the code covered in a lower-cost manner.

JH Those specialized developers might be expensive, but if they're using the right set of tools or languages or frameworks or paradigms, then you have the potential to squeeze a lot more out of them. There's real value in identifying those individuals who are naturally inclined toward asynchronous programming and intensively training them. Beyond that, I think we're starting to see more frameworks and tools that have the potential to yield some tremendous savings once you start leveraging them such that those specialists can produce six or seven or even eight tests a day instead of just two.

TC Initially I got the sense, Michael, that you were hoping to find a DSL that would let you take better advantage of QA personnel by enabling them to execute a reasonably broad set of tests. Meanwhile, Jafar, it sounds like your experience so far is that the asynchronous stuff is sufficiently complex that the real win lies in finding those people who have some natural talent for it and then making them super-efficient.

How is this going to play out long-term? Is asynchronous programming just so difficult that it's always going to be the province of power people? Or is there anything to suggest this can be made more accessible to less-sophisticated programmers?

MD I think we're going to see a mix of the two. There's going to be some significant portion of any product that will remain fairly straightforward, where the coding is likely to be the kind that can be handled by lower-cost individuals once you've got the right framework in place. But that framework is going to need to be set up by someone who understands asynchrony and who has the training and

experience to deal with other reasonably complex requirements. There's definitely going to be a role for some highly trained and talented individuals, but you also want to make sure you can leverage those efforts to make their contributions broadly applicable.

JH I'm a little pessimistic about that. We recently were looking to build some asynchronous frameworks on the server at Netflix, and I think some of our developers started out with a similar attitude, based on the assumption that maybe 80 percent of our asynchronous problems could be easily solved with a few helper methods. The idea was to provide some simple APIs for a few of the more common concurrency patterns so junior developers would be able to tackle most of the asynchronous problems. We discovered that simple APIs solved only about 10-15 percent of our end-user cases—not 80 percent. That's because it was very easy to fall off a cliff, at which point it became necessary to revert to dealing with primitives such as semaphores or event subscriptions.

It turns out that even seemingly trivial async problems are actually quite complicated. For example, if you're making a remote request, it will invariably require some error handling like a retry. If two operations are executing concurrently, you'll need a way to specify different error-handling policies for each operation. What's more, each of these operations might be made up of several other sequential and concurrent operations. In reality, you need a compositional system to be able to express such rich semantics.

I admit it's possible that some simple helper APIs might prove more useful for building tests since the requirements are less stringent than for app development. So maybe you're right, Michael, to think you can mix low-skilled programmers with highly skilled ones. What exactly that mix looks like remains to be seen, though.

MD I couldn't agree more. I think that's the big question.

TC On a somewhat different note, my group has been developing for asynchronous environments, and finite-state machines have worked really well in that regard. We've found them to be a stunningly good way to capture information about events and transitions and stuff like that. So what are your thoughts about using state machines and some kind of language built around that? Are state machines simple enough for less-skilled developers, like QA people, to use them effectively?

MD I certainly think state machines describe the mathematics well enough. A transition effectively amounts to a stimulus-response pair. So, yes, you can describe what we're talking about as hierarchical-state machines. And yes, that's the perfect mathematical paradigm to use for discussing this. But you can't present that to low-cost personnel and expect them to be able to do anything with it. What you can do, though, is to use those same mathematics to create the tools and the machinery that drives all this stuff. In terms of what you put in front of the QA people, however, that can't be anything more than what they already recognize as stimuli to the responses they're looking for.

JH I completely agree. It's true that the primitives are simple enough that everyone can understand how to hook up to an event, set a variable, and then move from state to state. In practice, however, those simple primitives don't mean the overall program itself is going to be simple. In fact, it's going to be quite complex because there are so many different moving parts.

Another approach to asynchronous programming that's gaining a following in the JavaScript world is around an abstraction called Promises that's being integrated into common JS and F#. That gives you an async type that provides for composability while letting you build asynchronous programs in a stateless way. That might be the model you end up having to embrace—a declarative

stateless way of describing an asynchronous computation—since that’s a level of abstraction that lower-skilled developers might be able to take advantage of.

TC Could you provide an example of that?

JH What it comes down to is that there’s a new way of thinking about asynchronous programs. The move away from GOTOs to structured programs raised the level of abstraction. Today we build asynchronous programs with callbacks and state machines, and these programs suffer many of the same disadvantages of the old GOTO-based programs: logic tends to be fragmented into many different pieces. We can resolve this problem the same way we resolved it earlier—by raising the level of abstraction. Instead of using callbacks and state machines to build asynchronous programs, we can model them as sequences of data. An event, for example, can be seen as a sequence of data—one, in fact, that has no end. There’s no way a mouse-move event is going to be able to say, “Hey, I’m done.” It just goes on and on forever.

It’s interesting to note we already have a means for modeling sequences in synchronous programming: the familiar iterator is a synchronous way of moving through a data structure, from left to right, simply by continuing to request the next item until the iterator finally reports there’s no more data. Erik Meijer, when he was at Microsoft, turned the iterator pattern inside out and found the observer pattern fell out. In mathematical terms, the observer pattern is the dual of the iterator pattern. This is a very important unification since it means anything we can do to an iterator can also be done to observers such as event listeners.

The significance here is that we have several high-level languages for manipulating data structures that can be expressed as iterators. The most relevant example is SQL, which I would argue is a very successful high-level language because it allows developers to create complex queries that are both easy to understand and powerful to use. Now, based on the discovery that the observer and iterator patterns are dual, Erik has managed to build a framework that allows an SQL-like language to be used to create asynchronous programs.

The idea is that events and asynchronous requests for data are collections, just like arrays. The only difference is that asynchronous collections arrive over time. Most operations that can be performed on a collection in memory can also be performed on collections that arrive over time. Hence we find that a DSL originally built into C# to compose synchronous sequences of data can also be used to compose asynchronous sequences. The result is a high-level language for building asynchronous programs that has the expressiveness and readability of SQL.

MD That’s a step in the right direction. I’m definitely going to look into this further.

JH We’re using this technology on our Xbox platform right now. It seems to be just what you’re looking for, Michael.

TC Can you describe how Erik Meijer’s Reactive Extensions work applies in a QA environment? Say you’ve got a bunch of consoles you need to drive through some sequences so you can verify that certain things are happening in the game you’re testing. Where does Rx fit into that? What would you be querying in that circumstance and how would you be able to turn that into a test result?

JH That’s a great question, since some people have difficulty seeing the connection between querying a database and creating a test. A test in its own way is actually a query along the lines of: “Did this stream fire and did that stream fire before some particular event fired, which then led to some other thing happening?” That’s really no different from querying a table to see whether a certain condition is true.

TC We would still need some mechanism to drive the system through different states. Perhaps Rx could even be used for that. At each stage the query is going to come back as either true or false. If it comes back false, then we'll know the test didn't pass since the sequence of events we had been expecting didn't match the query we issued.

JH Exactly right. But this can be partitioned into two steps. The first is the one Michael already mentioned: transitioning the system so as to make it more observable, and by and large that's simply a matter of adding events that fire when interesting things occur. The second step involves building queries over those events. Those queries would be very, very declarative—they wouldn't be state machines at all—so you would be able to confirm that certain conditions are met as you drive through the system.

TC It sounds like you're applying this approach to a product now. Has that experience proved to be positive? Are you finding that the Rx syntax or the query syntax is something non-experts might be able to use to capture information about the system?

JH Thus far, I don't think the syntax has really helped as much as I'd anticipated. The real challenge is in making the leap to thinking about events as collections. Most people have spent their careers thinking about events very mechanistically. Although thinking about events as collections might be conceptually simpler, it may also prove difficult to make the transition at the organizational level, if only because it's so hard to break bad old habits. My sense, however, is that if you can find some developers who are already inclined toward functional programming, then when you give them these powerful new tools for asynchronous programming, you're going to be able to realize the sorts of economies we're talking about.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

© 2014 ACM 1542-7730/14/0400 \$10.00