DETAILED OUTLINE.

INTRO. The Problem.

Depending on the statistics you pick, hospitals harm between 3.7% and 16% of their patients. In the US, medical error kills 94,000 people per year -- equivalent to a fully laden Boeing 747 crashing every second day. Failures of communication are a major factor in 40% of adverse patient events, and 70% of the ones that caused patients serious harm. One study attributed communication as a major factor in 1,500 deaths per year in Australia alone.

But, while doctors have very sophisticated simulators for practicing operations, there’s not much technology to support their communication skills. Doctors are very used to looking at scientific visualizations (like X-Rays, fMRIs, ECGs, etc), so why not start to “X-ray” medical communication and see if we can see what’s happening? What is good communication, and how does bad communication happen?

While we can’t tackle all medical miscommunication at once, because it’s not all “communication skills”. (Eg, forgetting to write something down in the patient notes is a failure to communicate). But we can try to tackle the kinds of communication that affect every patient. “Patient history taking” (where the doctor talks to the patient to find out what’s wrong) and “ward handover meetings” (where an outgoing shift meets with an incoming shift) are particularly important.

Like many scientists, we start by looking at training scenarios because they are more controlled conditions. So, we start by recording and analyzing doctors consultations not with real patients but with actors (it’s called a “standardized patient exercise” and trainee medics already undergo these; we’re adding scientific analysis to them).

(In the conference videos, the doctor is also played by an actor, for ethical clearance reasons.)

PART 1. DEMO

NB: I haven’t included a real screenshot because we do need to make the CSS/styling more attractive. (I don’t want to kill a proposal with an ugly screenshot that a reviewer might assume was the finished product. Ugly concept art is less damning.) We can make real screenshots available to you on request, before the decision in May if necessary.

In this illustration (original concept diagram) “Track views” are the ones that line up like tracks along the bottom. “Playing views” (or “Frame views”) are the ones that sit in frames like the video frame and the concept map.
• First, just a video of a doctor talking to a patient.
• Add a transcript.
• Show a very simple visualization (a track view) colouring who is speaking on the sound waveform. Now you can see who’s doing all the talking.
• (Close that visualization.)
• Show a track showing what concepts are in each utterance (Latent semantic analysis)
• Add a graph next to it of how hard the patient is thinking (voice stress analysis). Now you can start to see the tricky topics.
• (Close them.)
• Show the concepts for the conversation as a whole, and mark where they occur. Now you can see what parts of the conversation are the most relevant.
• (Close it.)
• Show an automatically generated topic map that changes as the conversation progresses (Leximancer). Now you can see who is talking about what, and who “owns” which topics.
• (Close it.)
• Show a track view of how relevant each utterance is to the previous one (LSA again), colour-coded by speaker. Now you can see if the doctor and patient are responding to each other, or talking at cross purposes.

We can zoom in, zoom out, and we can navigate using any and all of the plug-ins.

The audience should get from the demo:
1. By using different visualizations, we can see different things about the conversation.
2. Every time there’s a new kind of analysis, we want to be able to plug it in
3. It’s easy to see what the app itself is doing.
4. This is a serious app -- many different kinds of data, many plug-ins, all dynamically loaded, linked as needed. It’s not just a toy example. But it’s all about rich interaction.

PART 2. HOW IT WORKS.

Then we start to dissect the demo.

Point out which of the track views are Swing, which ones are JavaFX (it needs pointing out -- they mix almost indiscernibly well)

Each of the plug-ins (the different tracks, and the different playing views) represent different scientific analyses. As new analysis methods become available, we want to be able to plug them in. Better, we want the teams that produce them to be able to plug them in. So, of course we have a plug-in model.
OSGi in 3 minutes. (So as not to bore those who are familiar with it, and not to lose those who are not.) It lets us load plug-ins dynamically, and separates their class visibility, so that two plug-ins can happily co-exist even though they use different versions of the same library.

Code sample: JavaFX code to load Apache Felix.

Explain that the JavaFX license doesn’t let us redistribute the JavaFX JARs (as you’ll see mentioned on the Web often), but we don’t have to -- if the JNLP launches a JavaFX app, then the client machine will take care of getting those JARs from Sun directly. We just have to mark them as being exported (so OSGi will let the plug-ins access them).

In academia (where new analysis methods will come from) there are still more Swing programmers than JavaFX programmers. So, we make sure they can just use Swing if that’s what they know. But we’d like them to be able to use JavaFX because it is so much quicker.

Code sample: TrackView is an interface, so it can be implemented by a JComponent subclass or a Node subclass. TrackSpace accepts a TrackView and wraps it in a SwingWrapper if necessary.

So that let’s our plug-in writers use either technology. We said the JavaFX version is shorter. Let’s see:

Code comparison (we’ll just compare length): The same plug-in, in Swing vs in JavaFX.

So it wasn’t much work to get the OSGi loaded in the client. How much work is it to produce it in the first place? Not much. We just need to add code to advertise that the plug-in is there.

Code sample: An Activator advertising a plug-in.

Code sample: Using iPOJO annotations to do the same thing.

So now we have a JavaFX GUI that can accept Swing and JavaFX plug-ins, and we have a way of writing those plug-ins. But we should also mention how we build it all.

We do our builds with Maven, as probably do many of you. We like Maven. It’s like everybody else has done our work for us -- and sure enough other people have done the work of building both OSGi and JavaFX modules in Maven.

Code sample: pom.xml file of a plug-in, using the Felix OSGi bundle Maven plug-in (pointing out the import of the JavaFX classes)

At the moment, we use JFrog’s Maven plug-in for JavaFX, and we’ll show how that’s set up (with system scope dependencies on the JavaFX jars so we don’t have to put them in a repository). However, I’m aware Alex Ruiz of FEST has been working on a “simpler” Maven plug-in for JavaFX, so we might swap the example to that if it proves successful.
Then explain how we have a multi-module build. Core API, Swing-only GUI, JavaFX GUI, plus one module per plug-in. (Actually plus modules for wiring scripts in JavaFX and Groovy -- that set the client up in different ways for different experiments.)

Code sample: parent pom.

Code comparison: (colour coding the sections we are referring to, just showing presence of blocks and not examining syntax) Highlight the reference to the parent pom in a child pom. Note that only plug-ins have the OSGi bundle section. Only JavaFX modules have the JavaFX invocation.

The point here is that we have JavaFX and Java modules, and that imports between them work.

With Maven, we can easily build the application for deployment on the web with a JNLP. But if you’re a NetBeans user, you probably like just being able to click “Play” to try your app out locally.

Tip: Make a Maven assembly project that gathers all the core modules (not the OSGi plug-ins you want to load dynamically). Create a non-Maven NetBeans JavaFX project that just contains a Main.fx that calls your assembly. Add the assembly jar (inside ~/.m2/repository/... as a library.) Now you can click Play to run locally.

PART 3. LESSONS TO MAKE LIFE EASIER, AND TRAPS TO AVOID.

I’ll illustrate this with our “Reorientable” interface.

Demo changing the orientation of the track GUI (horizontal to vertical and back). That’s our “Reorientable” interface at work, that says a component can be flipped from a leftToRight to a topToBottom orientation. As you might guess, we have a lot of components whose orientations all need to be bound together. (ScrollPane, Axis, TrackSpace, TrackCarriage, TrackViews...)

Implementing a Java interface as a JavaFX mix-in (giving it a variable to bind to and validity checks).

Code sample: Reorientable is wrapped as FXReorientable, which also checks that orientation is not null (not even through failing to set it). That saves a lot of boilerplate code.

Trap to avoid 1: Container.fx calls getPrefHeight() in its init{}. But if the container’s getPrefHeight() checks its orientation, either orientation needs to be set before superclass init, or it mustn’t complain about nulls.

Ok, so that sounds like we should push all our set-up into the declarative matter that is called before init{} where possible? Not quite.

Trap to avoid 2: “override var = “ changes initialization value but not order of declaration

Code sample: class A { var v1 = “1”; }
class B extends A { var v2 = “2”; override var v1 = v2; }.
new B().v1 ends up null. v1 is declared first, so it is set first. But it is set to v2 (by the
override), which hasn’t been declared yet, so it is null.

Code sample (code-folded to short): setting the onMousePressed event on a custom
Control to point to a method in it’s Skin’s Behavior. Fails because the “override var
onMousePressed = ...” happens before the custom Skin is created.

Life lesson: JavaFX is easy after initialization has taken place. But object initialization can
be hard to debug. (Video of stepping through it in NetBeans; looks like the execution point
is jumping all over the place). And it’s tricky to put log messages in before the init{} call.

Rule of thumb: Don’t refer to any other member variable in “override var x = “; not even
other member variables from the superclass (you don’t know what their order of
declaration was).

We found 90% of our debugging time when we started working with JavaFX was issues
that came about because of initialization order. Be careful about that, and everything else
will be much easier for you.

PART 4. TWO EXAMPLES WHERE UNDERLYING JAVA IS REALLY USEFUL.

1 Visible bounds. Pop up the demo, and a track view that just has little charts for chunks
of time (what they are charting doesn’t matter here; we could use thumbnail images of the
video otherwise). Zoom in really close. Across the length of a two hour video, there would
now be tens of thousands of charts. We don’t want to keep all that live, so instead our GUI
tells each of the TrackViews the bounds they need to cache. We provide abstract
JComponent and Node classes that will update their visible content from what’s put into
the cache. Handily, the JRE includes an LRU cache.


2 Consuming custom streams. The data comes from an Apache Sling server, where we
store all kinds of data generated by all kinds of server-side plugins (from ffmpeg to
research-grade Matlab math modules). We’ll use sound as a straightforward example.

We keep an Ogg Vorbis version of the audio for graphing the waveform (because JMC
doesn’t allow direct access to the waveform). Because it’s Java under the hood, we’ve
plugged in JOOrbis, and writing a class to put data chunks out to a temp file was also a
straightforward task -- that let us write our own stream cache that copes with us jumping all
over the file at all sorts of different resolutions (scroll back and forth, zoom in and out)

WRAP-UP

You’ve seen how we can visualize “the science of communication”. Technically, you’ve
also seen a real-world fairly complex example of Swing and JavaFX OSGi plug-ins
working together, produced from Maven builds, deployable on the Web, and locally
executable from NetBeans. And you’ve seen that computer scientists do have something
to say about communication skills after all!