Edisyn

A Java-based Synthesizer Patch Editor, Version 15
By Sean Luke sean@cs.gmu.edu

Contents

1 About Edisyn

Edisyn is a no-nonsense synthesizer patch editor for the editing and parameter exploration of a variety of synthesizers. It is not skewmorphic or skinnable: its design is plain and consistent. Edisyn is free open source. Edisyn currently supports the following synthesizers:

- Kawai K1, K1m, and K1r (Single and Multimode)
- Kawai K4 and K4r (Single and Multimode)
- Kawai K5 and K5m (Single)
- Korg SG Rack (Single and Multimode)
- Korg Microsampler
- Korg Wavestation SR (Performance, Patch, and Wave Sequence)
- Oberheim Matrix 1000 (Single)
- PreenFM2 (Single)
- Waldorf Blofeld Desktop, Blofeld Desktop SL, and Blofeld Keyboard (Single and Multimode)
- Waldorf Microwave II, Microwave XT, and Microwave XTk (Single and Multimode)
- Yamaha DX7 Family (DX7, TX7, TX216/816, etc.) (Single)
- Yamaha TX81Z (Single and Multimode)

Edisyn does not support patches for global parameters, nor for wave, sample, or wavetable editing etc. Additionally, though it can do bulk downloads and save them as individual patches, Edisyn is **not at present a librarian tool**. You should use a good free librarian software program, such as SysEx Librarian on the Mac.

2 Starting Edisyn

If you're on a Mac, Edisyn will look like a standard application, just double-click on it. On other platforms, Edisyn comes as a single Java jar file. Just double-click on the jar file (you'll have to have Java installed) and Edisyn should launch.

You'll first be presented with the dialog at right, asking you to choose a synthesizer patch editor. You can either connect to a synth then and there, or run in **Disconnected Mode**, where you're not attached to MIDI. You can also quit immediately.

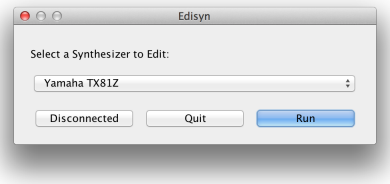

Figure 1: Initial Synthesizer Dialog

Edisyn will now build a patch editor for you and display it. But unless you chose **Disconnected Mode**, it'll first ask you to set up MIDI for this editor. The dialog at right presents you with up to 6 fields (5 are shown here):

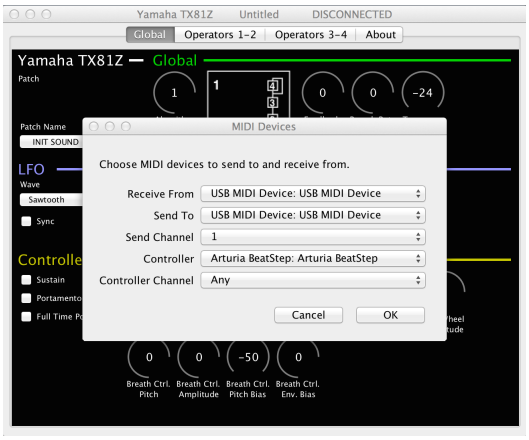

Figure 2: MIDI Dialog

- The USB MIDI Device from which you will **Receive** MIDI data sent by the synthesizer. Here we are sending to a Tascam US-2x2 interface, which presents itself as a generic, nameless device. (BTW, if you're on a Mac and you don't like a generic name for your device, go to the Audio MIDI Setup application in the Utilities folder, double-click on the device, and change its name.)
- The USB MIDI Device to which you will **Send** MIDI to ultimately be sent to the synthesizer. Here again we are sending to the Tascam US-2x2 device.
- The **Channel** on which the synthesizer is listening. (Here, 1).
- (Not visible here) The optional **ID** of the synthesizer. Some synthesizers require a special ID embedded in their sysex so they can tell that the message is for them rather than another copy of the same synthesizer. (The Yamaha TX81Z doesn't have an ID, so it's not displayed in this example).
- The USB MIDI Device from which you will receive MIDI data sent by a **controller**. This may be a controller keyboard to play test notes on the synthesizer, or it may be a control surface to send CC data to the synthesizer or to Edisyn itself. Here we are receiving from an Arturia Beatstep.
- The **Channel** over which you will receive MIDI data sent by a **controller**. This can be any specific MIDI channel, or (in this example) "Any", meaning any channel or OMNI.

If you are not connected to MIDI, or if you cancel, then Edisyn will inform you that you must continue in **Disconnected Mode**.

Important Note If your USB MIDI device is manually disconnected, Edisyn won't know until you ask Edisyn to send the synth something (perhaps changing a parameter or uploading a patch). At that point, Edisyn will get a clue and the patch editor window will change to **Disconnected Mode**.'

3 Edisyn Patch Editors

An Edisyn patch editor is a single window with multiple tabbed panes. You can switch tabs by clicking on them or via shortcuts (see the **Tabs Menu**). The far-right tab is the **About Tab**. It gives you information about the eccentricities of the synthesizer that require custom behavior in Edisyn (they all do!). You should read it carefully to understand how Edisyn will interact with your synthesizer.

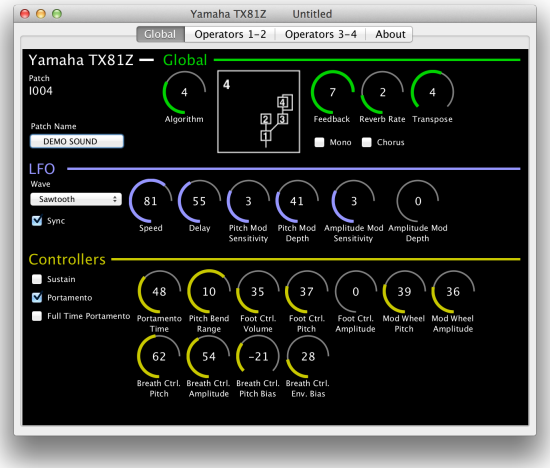

Figure 3: Typical Patch Editor Panel (TX81Z)

Categories At right is a typical tab pane. You'll note that various widgets are grouped together in regions (called **Categories**). There are four categories shown here. Three are arbitrary categories for this synthesizer: "Global", "LFO", and "Controllers". They're in various colors to differentiate them. Other categories will be found in other tab panes. But one category is special: the **Synthesizer Category**, always shown in white, here named "Yamaha TX81Z". It normally contains the patch name and bank/patch number.

Widgets Edisyn has a number of widgets. Here are some:

- The **Patch Display**, currently showing patch "I004". Sometimes this display will be inaccurate, particularly if you manually change the patch on the synthesizer while Edisyn is running; or if Edisyn has no idea what the patch should be (it'll usually display a default value like, in this case, "A001".
- The **Patch Name Button**, currently showing "DEMO SOUND". Click on this button to change the name of your patch. A dialog will pop up to let you change the sound, with an additional **Rules** button to explain the constraints the synthesizer places on patch names.
- Displays of **Keyboards**. Select a key!
- Various **Dials**.[1] These are semicircles in gray, partly in some other color, with a value in the center. You change these values by clicking on the dial and dragging vertically. You can also double-click on a dial to reset it to a default value (often zero). Finally, you can two-finger drag (on the Mac), or spin the mouse wheel to change the dial more subtly.

  Dials vary in orientation. Most look sort of like a "C", with the zero point at the bottom center. Other dials are symmetric, such as the "Breath Ctrl. Pitch Bias" dial (bottom row, second from right in the Figure), and have zero point at center top. Occasionally dials have other orientations: the goal is to keep the zero point centered (at top or bottom).
- Some **Checkboxes** (such as "Portamento") and **Pop-Up Choosers** or ComboBoxes (such as "Wave", set to "Sawtooth"). These are should be straightforward.
- Various **Pictorial Displays**. Here, changing the "Algorithm" dial will modify the Algorithm Display immediately to the right of it.
- Various **Envelope Displays**. Edisyn can draw envelopes using a variety of procedures. Consider the Waldorf Microwave envelopes in Figure **??** above, for example. The first two envelopes are ADSR envelopes, but the third is the Microwave's famous "Wave Envelope", an eight-stage envelope with two different looping intervals (shown below it), and with two special end times marked with vertical lines (here, the dashed line is where optional sustain occurs, and the solid line is the end of the wave). The last envelope is the Microwave's "Free Envelope", a four-stage envelope unusual in that it can have both positive and negative values: the dashed line is the axis.

  Mose Edisyn envelope displays are read-only – you can't draw the dots. But that's not always the case: for example the Kawai K5 harmonics display can be extensively edited by mouse.
- **Action Buttons**. Some patch editors have buttons on them which perform actions rather than edit or display values. For example many multimode-patch editors have buttons that pop up single patch editors for the various individual patches.

If you are connected to a synthesizer over MIDI, then changing a widget will modify the underlying patch parameter in real time, if the synthesizer supports this. Also, if you modify a parameter on the synthesizer, then Edisyn will update the corresponding widget or widgets (again, if the synthesizer supports this).

[1] Notably absent are **Scrollers**. Edisyn has software support for scrollers but to be honest they've not proven useful yet.

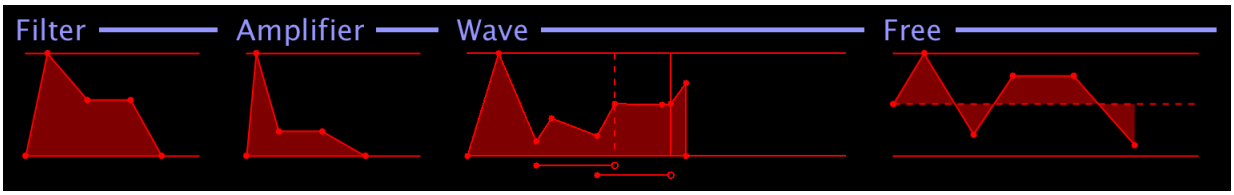

Figure 4: Envelope Displays of the Waldorf Microwave II, XT, and XTk.

4 Creating and Setting Up Additional Patch Editors

A patch editor is created by selecting one of the various **New...** menu options in the **File** menu. You have to create a new a patch editor before you can start loading a patch from a file or from the synthesizer. You can also **Duplicate** an existing patch editor (in the **File** menu). This will exactly duplicate the existing patch as well.

Whenever you create a new patch editor or duplicate one, you will once again be asked to set up MIDI as discussed in Section **??**, or to run in Disconnected Mode.

4.1 Persistence

Now would be a good time to mention an Edisyn feature you may never notice otherwise: many things are **persistent**. For example, if you choose "Arturia Beatstep" as the controller for your Blofeld patch, the next time you call up a Blofeld patch editor, "Arturia Beatstep" will be presented as the default choice in the MIDI Devices window, assuming your Arturia Beatstep is plugged in. This goes for everything in the MIDI Devices window. Furthermore, if you pop a new patch editor for a synthesizer you have never edited before, the Arturia Beatstep will be the default option for that one too (until you change it one time). And these options are per synthesizer type.

Persistence appears in other places too. For example, the Initial Synthesizer dialog will default to the last synth you chose in that dialog. And certain many choices are persistent as well.

5 Loading and Saving Files

You can save your edited patch via the **Save** and **Save As...** options in the **File** menu, and you can load a patch via the **Load...** option. This is called *Load* and not *Open* because you can only load a file into an existing patch editor: you cannot create a new patch editor automatically on opening a file. If the sysex file is not for your patch editor, but Edisyn still recognizes its data, it'll ask if you want to load for a different synthesizer. If Edisyn doesn't recognize the data at all, it'll tell you it's best guess as to the manufacturer of the device which produced it.[2]

Most patch editor files are sysex dumps ending in the extension `.syx`. These files are usually exactly the same sysex data that you'd normally dump to your synthesizer using a patch librarian software program. There are exceptions however. For example, some synthesizers, like the PreenFM2, have no sysex to speak of at all: they exchange parameters entirely over NRPN. In this situation, Edisyn has invented a sysex file just for the PreenFM2. It obviously won't work in your librarian software.

[2]Thanks to the MIDI Association for updating their database to make this possible in Edisyn.

Batch Downoads Edisyn also has limited support for batch-downloading and saving patch files. To do this, choose **Batch Download...** from the **File** menu. You'll be asked to specify the directory in which to save patches, and also first patch and the final patch, and then downloading will commence. Note that if your final patch is "before" the first patch, then Edisyn will wrap all the away around to get to the final patch. For example, if your synth has ten patches 1....10, and you choose 8 as your first patch and 2 as your last patch, then Edisyn will download in this order: 8, 9, 10, 1, 2.

If Edisyn can't download a particular patch (the synth isn't responding), it'll try again and again until successful. So if it gets stuck, you can always stop batch-downloading at any time by choosing **Stop Downloading Batch** from the **File** menu. Note that you can still screw with knobs, etc. while Edisyn is busy downloading batches: but don't do that. You're just messing up the batches getting saved.[3]

Also note that as a failsafe Edisyn only allows the frontmost window to receive data over MIDI. This means that while you're batch-downloading, you can't go to some other patch editor: the downloading patch editor must stay in front. You can go to another application though (read a web browser say).

6 Communicating with a Synthesizer

First things first: if you're working in Disconnected mode, you'll need to set up MIDI before you can communicate with your synthesizer. This is done by selecting **Change MIDI** in the **MIDI** menu. (By the way, you can go Disconnected by selecting **Disconnect MIDI** in the **MIDI** menu as well). Remember that you have to connect USB devices to your computer *before* starting up Edisyn, or it won't see them, due to a bug in the MIDI subsystem.

Now that you're up and running, if you change widgets in the patch editor, many (not all) synthesizers will automatically update themselves. The opposite happens as well: changing a parameter on the synthesizer will update it in Edisyn. See the About pane to determine if your synthesizer can't do this.

By selecting **Request Current Patch**, you can also ask your synthesizer to send you a dump of whatever patch it is currently running. It is often the case that synthesizers respond in such a way that Edisyn cannot tell what the patch number or bank is. In these cases Edisyn will reset the patch number to some default (like A001).

Request Patch... will ask the synthesizer to send Edisyn a specific patch that you specify. Edisyn often (not always) does this by first asking the synthesizer to change to that patch and bank, and then requesting the current patch.

Send to Current Patch will dump Edisyn's current patch to the synthesizer, instructing it to only update its local working memory, and not to store the patch in permanent memory. This operation is primarily used to sync up certain synthesizers which do not update themselves in real-time in response to parameter changes you make.

Send to Patch... will ask the synthesizer to change to a new patch and bank which you specify, then dump Edisyn's current patch to the synthesizer in its working (not permanent) memory. This also isn't used all that much: but some synthesizers (like the PreenFM2 or TX81Z) cannot be permanently written to remotely. Instead you send to a patch, then store the patch manually on the synthesizer itself.

Sends Real Time Changes controls whether the Edisyn will send parameter changes to the synthesizer in real time in response to you changing widgets in the patch editor. This isn't necessarily determined by the synth model. For example, the default ROM for the Oberheim Matrix 1000 cannot handle real-time changes: but ROM versions 1.16 or 1.20 (later bug fixes by the Oberheim user community) allow real-time changes with no issue.

Write to Patch... will ask the synthesizer to change to a new patch and bank which you specify, then dump Edisyn's current patch to the synthesizer to its permanent memory.

Note that various synthesizers cannot do one or another of these tasks. When this happens, that feature will generally be disabled in the menu. As always, read the About Tab to learn more about what's going on

[3] I may change this in the future to something less fragile.

with that synthesizer model. See Section **??** for some information and griping about all this.

6.1 Test Notes

If you don't have a controller keyboard, you can send a test note to your synthesizer by choosing **Send Test Note**. You can also toggle whether Edisyn constantly sends a stream of test notes by choosing **Send Test Notes**. And you can shut off all sound on the synthesizer with **Send All Sounds Off** (this also turns off sending test notes).

Edisyn gives you various options for adjusting the test note you send (though it's always a "C"). You can change the length of the test notes you send in the **Test Note Length** submenu. You can change the pitch with **Test Note Pitch**. And you can change the volume with **Test Note Volume**.

Setting the **Pause Between Test Notes** will change how long Edisyn waits, beyond the note length itself, before it plays the next note if you have **Send Test Notes** on. It doesn't affect how fast you can play test notes on your own. One special setting is **Default**: this is defined as an additional pause equal to the note length if the note length is less than 1/2 seconds; or a pause of 1/2 second if the note length is greater than this.

6.2 Testing the Incoming Connection

If you're not sure if you have MIDI data coming to Edisyn from your synthesizer, select **Report Next Synth MIDI** from the **MIDI** menu. Then have your synth send any kind of MIDI message to Edisyn — a note, a sysex message, whatever. For example, you could have Edisyn request a sysex dump from the synth. At any rate, if Edisyn pops up a window telling you the message, then you have a live connection.

7 Communicating with a Controller

The MIDI Dialog (Section **??**) also lets you choose a device and MIDI channel for incoming messages from a control surface or controller keyboard. Using this keyboard you can:

- Play the synthesizer (through Edisyn).
- Control the synthesizer (CC and Program Change messages, etc.)
- Control widgets in Edisyn

7.1 Testing the Incoming Connection

If you're not sure if you have MIDI data coming to Edisyn from your controller, select **Report Next Controller MIDI** from the **MIDI** menu. Then have your controller send any kind of MIDI message to Edisyn — for example, play a note. If Edisyn pops up a window telling you the message, then you have a live connection.

7.2 Remote Control of your Synthesizer

If you play a note, do a pitch bend, etc., on your control surface, and **Pass Through Controller Data** is set, then Edisyn will route all of those MIDI messages directly to your synthesizer (changing the messages' channel to the one that Edisyn is using to talk to the synthesizer). Control Change (CC) and NRPN messages from your control surface are passed through only if you have *also* toggled **Pass Through All CCs** in the **Map** menu. Otherwise they *might* used to control Edisyn' via its parameter mapping (see the end of Section **??**).

If your controller is sending these messages on Edisyn's Controller Channel, Edisyn usually just routes them through unchanged, but it *might* route those messages to some other channel instead. This only happens in certain patch editors where it's appropriate. For example, the Kawai K4/Kr4 [Drum] Patch Editor needs to forward note messages like these to the Kawai K4's "Drum" channel to hear them. The

"Drum" channel is different from the Kawai's primary MIDI communication channel (which is what Edisyn's Send Channel is set to).

7.3 Remote Control of Edisyn

Edisyn is capable of *mapping* Control Change (CC) messages or NRPN messages from your control surface to parameters in your patch editor. Each patch editor type can learn its own set of CC and NRPN mappings.

Mapping a Parameter Mapping a parameter is easy:

1. Choose one of three MIDI mapping menu options discussed next. The title bar will say "LEARNING".
2. Select the widget you want to map, and modify it slightly. The title bar will change to "LEARNING *parameter*[*range*]", where *parameter* is Edisyn's name for the synthesizer parameter in question, and its values are in $(0...range - 1)$. The title bar might also tell you what the *previous* mapping was. If $range > 127$ then you should think about mapping with 14-bit NRPN instead of CC.
3. Press or spin the knob/button on your controller. You're now mapped!
4. If you have chosen an absolute mapping, you'll want to change your controller's range to $(0...range - 1)$.

Edisyn accepts any of the following MIDI Control commands.

- **Absolute CC** The value of the CC sent is exactly what the parameter will be set to (between 0...127). To map, choose **Map CC/NRPN** in the **Map** menu. This style is particularly useful for potentiometers or sliders. You are not permitted to map CC numbers 6, 38, 98, 99, 100, or 101, or Edisyn will think you're sending NRPN. So you only have 121 CCs to play with.
- **Relative CC** Here, the CC value you send indicates how much to *add to* or *subtract from* the existing parameter value.[4] This style is supported by a number of controllers and is useful for encoders.[5] For example, the Novation controller series calls this "REL1" or "REL2"[6], and the BeatStep calls this "Relative 1".[7] To map, choose **Map Relative CC** in the **Map** menu. Again, you're not permitted to map CC numbers 6, 38, 98, 99, 100, or 101.
- **NRPN** You are permitted to map any NRPN parameter you like. The value of the CC sent is exactly what the parameter will be set to: all 14 bits. If your controller can only send 7-bit NRPN, then you should configure it to send "Fine" or "LSB-only". Edisyn also supports the NRPN Increment and Decrement options, though those are rarely supported by hardware. To map, choose **Map CC/NRPN** in the **Map** menu.

[4]Specifically, a value $x = 64$ means 0 (add nothing), a value $x < 64$ means to subtract $64 - x$ from the current value, and a value $x > 64$ means to add $x - 64$ to the current value.

[5]You could also map a pair of pushbuttons to be up/down cursors using this method: set up the "down" pushbutton to send 63 and the "up" pushbutton to send 65.

[6]The difference being that in REL2 mode, if you spin the encoder rapidly, the amount added/subtracted is nonlinearly more than expected, whereas in REL1 the speed doesn't matter, all that matters is how far the encoder was turned. Novation controllers also have a relative CC mode called "APOT", which is not supported.

[7]The Beatstep also has relative CC modes called "Relative 2" and "Relative 3", which are not supported.

Mapping by Panel or by MIDI Channel By default, Edisyn only maps and responds to CCs (or NRPN etc.) if they are on Edisyn's controller channel. Each tab in a patch editor can have its own unique set of mappings: for example, the Oscillators tab might use CC#1 to change the Start Wave parameter, but the Envelopes tab might use CC#1 to change the attack of Envelope 1. The mapping being used at the moment depends on which tab is being displayed.

Alternatively, if you toggle **Do Per-Channel CCs**, you can ask Edisyn to instead remember the channel of mapped CCs (or NRPN). Then you can map CC#1, on (say) channel 4, to the Start Wave parameter on the Oscillators tab, and map CC#1 on channel 7 to the attack of Envelope 1 on the Envelopes tab. If CC#1 arrives on channel 4, the Start Wave parameter will be adjusted even if the Oscillators tab isn't being shown; similarly if CC#1 arrives on channel 7, then the attack of Envelope 1 will be adjusted even if the Envelopes tab isn't being shown.

If your controller can only send a few CCs (it only has a few knobs and buttons) I would use the first option (per-panel mapping). If your controller can send a vast number of CCs, or you're comfortable with it from experience with your DAW, you might use the second option.

Where Data Goes Whether Edisyn will pass through data to your synth, or block it, or intercept it in order to map it, is as follows. If the data is CC/NRPN, then Edisyn must decide whether to *intercept and map* it. If you have selected **Pass Through All CCs**, Edisyn isn't permitted to intercept any CC/NRPN data at all. Otherwise Edisyn will intercept the CC/NRPN data if it is on your Controller Channel, or if the Controller Channel is OMNI, or if you have selected **Do Per-Channel CCs**.

If Edisyn isn't intercepting and mapping the data, or the data is something other than CC/NRPN, then Edisyn must then decide whether to *block* the data or *pass it through* to your synth. This is easy: the data is passed through only if **Pass Through Controller MIDI** is selected.

There are many situations where these combinations are useful. Here's a fun example. Suppose your synth doesn't respond to CC (only NRPN or Sysex) but you'd like to control it from your DAW, which *only* does CC, as is the case for many bad DAWs. You could set up the DAW as your Edisyn controller and map CCs to various synth parameters. Then you'd pass through non-CC data via Edisyn to the synth, but intercept CC data from the DAW to update parameters via Edisyn.

8 Communicating with a Software Synth or Digital Audio Workstation

In some situations you might wish to get Edisyn to communicate with a software synth: for example Dexed[8] is a nice DX7 emulator and works well with Edisyn. Or perhaps you might want to use Edisyn to translate CC messages from your DAW into Edisyn parameter changes, which then get forwarded to a synthesizer as sysex (see Section **??** to learn how to map CC messages to parameter changes).

You'd think it'd be easy to connect directly to another piece of software on your computer. But you'd be wrong! The problem is that Edisyn, because it's written in Java, can only connect to *MIDI devices*, and your software synthesizer or DAW has probably not registered itself as a device — it likewise probably is designed only to connect to MIDI devices.

To get around this, you need to make a *MIDI loopback*. This is where you create two *virtual devices* which are connected to one another. Edisyn and your software synth can see these devices. Consider Figure **??**. If Edisyn outputs to Virtual Device X (say) and your software synth is set up to *input* from Virtual Device X, then it will receive what Edisyn outputs.

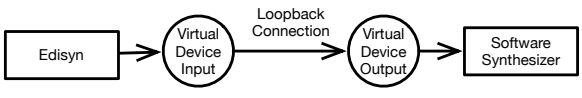

Figure 5: A MIDI loopback connecting Edisyn with a software synthesizer via a virtual device.

Similarly, if you need your software synth to respond to Edisyn, you need to make a *second* loopback and hook it up in the reverse order.

[8] https://asb2m10.github.io/dexed/

Making a Loopback Device This varies depending on your operating system.

- **On the Mac** First, open the application /Applications/Utilities/Audio MIDI Setup. Next, click on the "IAC Driver" icon to open the "IAC Driver Properties" window. Add a new port, named whatever you like. Check the box "Device is Online". This new port will be appear to Edisyn and to your software synth as the loopback device. You can add more ports to create more loopbacks. A loopback is only one-way: if you want Edisyn to send to *and* receive from a software synthesizer, you'll need to make two ports.
- **On Windows** There is no way to do this in Windows directly: instead you'll need to run a program which provides this service. Programs include loopMIDI, loopBe1, MIDIOx's MidiYoke, and so on. Googling for "loopback MIDI Windows" will get you there.
- **On Linux** In most flavors of Linux, to get virtual devices running you'll first need to type the command `sudo modprobe snd-virmidi` and then type in your password. If you're using something like Gentoo or any other distro that does not come with this kernel module, you'll need to custom compile your kernel to get it.

  This procedure will create a bunch of of virtual devices with names like `VirMIDI [hw:2,0,0]` or `VirMIDI [hw:2,1,8]`. Select a device whose third number is 0 (such as `VirMIDI [hw:2,0,0]` or `VirMIDI [hw:3,1,0]`, but not `VirMIDI [hw:2,1,1]`). Have Edisyn send to this device and have the software synthesizer listen from the same device. If you want to hook Edisyn and your synth up the other direction (so Edisyn receives from the synthesizer), you'll need to select a second virtual device.

9 Editing and Exploratory Patch Creation

Edisyn has a number of facilities to help you program your synthesizer, including tools to help you wander through the possible space of patches to hunt for the sound you want. Here's what you can do:

Undo and Redo Edisyn has infinite levels of undo and redo. When you change a parameter or do a wholesale modification, this can be undone, as can patch dumps and merges from the synthesizer. Individual parameter changes made manually on the synthesizer are not undoable even if they're reflected in Edisyn (it'd be too many). Loading and saving patches is not undoable. See the **Edit** menu.

Reset You can reset the patch editor to its "init patch". Just choose **Reset** in the **Edit** menu.

Category Cut/Paste, Distribution, and Reset Each category has a pop-up menu you get when you right-click or shift-click (or two-finger click on a Mac trackpad) on the category name. You can:

- **Copy Category** Marks the category to be pasted into other compatible categories.
- **Paste Category** Copies over all the parameters from the "copy" category, if it is compatible.
- **Distribute** Copies the last-modified parameter to all similar parameters in the category. For example, if you modified a step sequencer step, this might copy its value to all 15 other steps.
- You can also restrict your **Copy, Paste,** or **Distribution** to mutation parameters only.
- **Reset** This resets all the parameters in the category to their defaults.

Randomize (by some amount) You can add some randomness your patch parameters. Try a small value: values $\geq$ 50% are essentially full randomization. See the **Randomize** submenu in the **Edit** menu. Because it's so common to randomize, then undo and try again, you can also do undo-and-randomize-again as a single task: select **Undo and Randomize Again** in the **Randomize** submenu of the **MIDI** menu. See below for a discussion of how randomization (called **mutation**) works in Edisyn.

Nudge The nudge facility lets you push your patch to sound more and more like one of four other target patches you have chosen. You can use this, plus randomize, to wander about in the patch space. Before you can nudge, you have to first select patches to nudge towards. You can pick up to four patches by first setting up or loading the patch in your patch editor, then selecting one of **Set 1** ... **Set 4** in the **Nudge** submenu of the **Edit** menu. You don't have to ultimately select all four.

Above the **Set** options are four **Towards** and four **Away From** options, also in the **Nudge** submenu of the **MIDI** menu. When you set a patch, its current name will appear in the equivalent Towards/Away From option. The patch name is just a helpful reminder — it's entirely possible for four completely different patches to have the same name.

When you chose any of **Towards 1:...** through **Towards 4:...**, your current patch will get **recombined** with the target patch, by default by 25%, to move it towards that target. Similarly, when you chose any of **Away From 1:...** through **Away From 4:...**, your current patch will get adjusted (through a form of recombination) to *move away from* the target patch, by default by 25%. You can change the degree of recombination under the **Set Nudge Recombination** menu. Additionally you can add some automatic mutation whenever you nudge: just set its amount under the **Set Nudge Mutation** menu (by default it's 0%). If you did a nudge and didn't like it, you can try a slightly different one with **Undo and Nudge Again**.

A hint. It's a good idea to select target patches which don't have some radical difference creating a nonlinearity in the space between them: for example, if you were doing FM, I'd pick patches which all used the same operator Algorithm. See below for a discussion of how recombination works in Edisyn.

Merge (by some amount) Merging is a lot like nudging. But instead of nudging towards a predefined target patch, you are asking your synthesizer to load a given patch, which Edisyn will then directly **recombine** with your current patch to form a randomly merged patch. You specify the degree as a percentage: see the options in the **Request Merge** submenu of the **MIDI** menu.

Some patch editors may not be able to perform merges because the synthesizers can't load specific patches: if your synth can't do **Request Patch...**, it probably can't do a merge either.

Load and Merge This option, in the **File** menu, allows you to load a file and merge it with your current patch in more or less the same way that **Merge** works. The merge percentage is always 100% (that is, half-half).

Hill-Climb Hill-Climbing repeatedly presents you with sixteen sounds and asks you to choose your top three preferred ones. Once you have selected the three best, it performs various recombinations and mutations on those sounds to prefer sixteen new ones and the process repeats again. The idea is for your preferences to guide the hill-climber as it wanders through the space of synth parameters until it lands on something you really like.[9] For more on the Hill-Climber, see Section **??**.

9.1 Restricting Mutation and Recombination to Only Certain Parameters

You can restrict Mutation (Randomize) and Recombination (Nudge and Merge) to only affect a subset of parameters. To do this, choose **Edit Mutation Parameters** in the **Edit** menu. This will turn on *Mutation Parameters* mode (you'll see it in the window's title bar). You'll note that various widgets have now been surrounded with red frames. These widgets control synth parameters which are presently are being updated when you mutate a patch.

You can change these of course: just click on them and you can remove them from being updated (or add them back).[10] You can also turn on (or turn off) all of the parameters in a category by double-clicking on the category title. The categories in Figure **??** are *Yamaha TX81Z*, *Global*, *LFO*, and *Controllers*. Finally, you can turn on all the parameters in the entire patch editor by selecting **Set All Mutation Parameters** in the **Edit** menu, or conversely turn them all of by selecting **Clear All Mutation Parameters**.

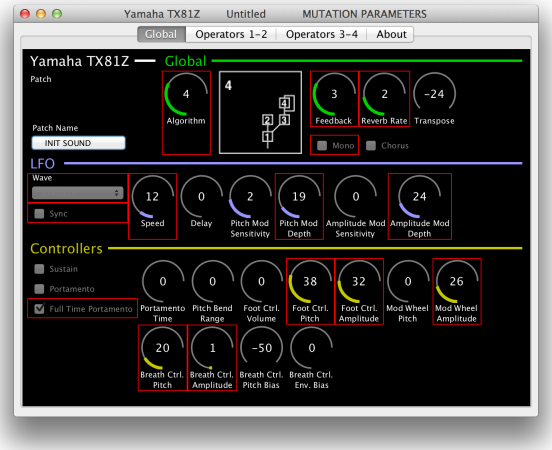

Figure 6: Editing Mutation Parameters

Some parameters, such as the patch name or certain other parameters, can't be mutated no matter what: these have been declared *immutable* by the patch editor. They will never have a red frame no matter how much you click them.

The parameters you have selected will be the only ones changed when you mutate (randomize) a patch. But if you turn on **Use Parameters for Nudge/Merge** in the **Edit** menu, then recombination (nudging, merging) will be restricted to these parameters as well.

Once you're done choosing parameters to mutate or recombine, just select **Stop Editing Mutation Parameters** in the **Edit** menu.

Note that your parameter choices, as well as using them for recombination, are persistent: they're saved in the preferences.

[9] If you're technically inclined, this is basically an evolution strategy (ES) with an elitism of 1 and a biased mutation procedure. If you'd like to learn more about evolutionary computation methods, google for the free online book *Essentials of Metaheuristics* by me.

[10] Note that due to an error in Java's design, you can't click directly on a Combo Box (a pop-up menu) such as the "Wave" combo box in Figure **??**. But you can click on its title (the text "Wave").

9.2 Hill-Climbing

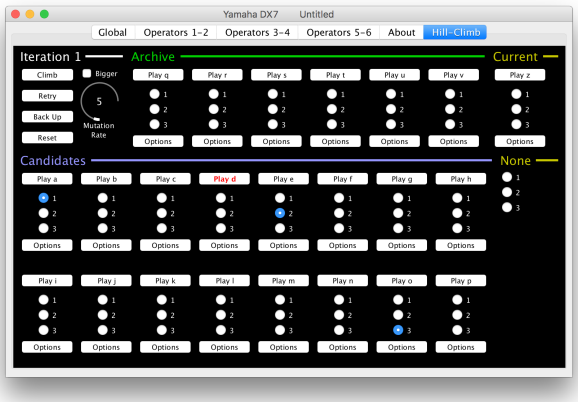

Figure 7: The Hill-Climber Panel

Edisyn's *hill-climber*[11] is another of its patch-exploration tools. You select it by choosing **Hill-Climb** in the **Edit** menu. This will add a new tab to your patch editor labelled **Hill-Climb**. When you fire up the Hill-Climber, it appears as an additional tab after your **About** tab. Whenever you select a tab other than the hill-climber tab, the hill-climber will pause; when you go back to the hill-climber tab it will resume. You get rid of the hill-climber by selecting **Stop Hill-Climbing** in the **Edit** menu.

Edisyn's Hill-Climber repeatedly offers you sixteen new versions of patches and asks you to choose your top three preferences. After you have chosen them, the Hill-Climber will try to build a new set of sounds whose parameters are similar to your choices in various ways. The Hill-Climber builds new sounds by recombining your top three sound choices in certain ways and adding some degree of noise (mutation) to them. If you're lucky, Edisyn will head towards regions of the synthesizer's parameter space which make sounds you like. **Hint: You will have more success with the hill-climber if you restrict the parameters being mutated to just those you want to explore.** Different kinds of synthesizers will also benefit from different amounts of exploration (mutation and recombination) rates. You'll need to tweak those as necessary.

Candidates This region holds the current candidate patches. When you start up the Hill-Climber, it will begin playing each candidate patch in turn. If you don't want Edisyn to play automatically, you can turn it off by choosing **Send Test Notes...** in the **MIDI** menu. Either way, you can manually play a patch by pressing its **Play** button, or by typing the key located on that button. You can of course also choose which three patches you like the most (in order).

Iterations After you have selected your preferred patches, you can build a new set of patches from them by pressing the **Climb** button. This will also increment the iteration number. If you don't like the patches that were built, you can try again by pressing the **Retry** button. If you'd like to back up to a previous iteration, press **Back Up**. Finally, to reset back to the very first iteration, press **Reset**.

The amount of mutation noise used when generating new patches is specified by the **Mutation Rate** dial. Typically you'd select something around 5–10. Additionally, by default the Hill-Climber lets you select from 16 candidates. But if you press **Bigger**, this set will expand to 32 candidates.

Archive If you like a patch and you'd like to hold onto it even after hill-climbing to the next iteration, you can place it in the *archive*. The archive consists of six patch locations: to copy a patch to a given archive location, click on its **Options** button and select one of **Archive 1** through **Archive 6**. Archived patches can also be selected to participate in hill-climbing: just pick a number under a given patch.

[11] Important Note. The Hill Climber is a work in progress and as testing and research continues on it, it may change significantly

Current and None The *Current* category holds the patch currently being edited in your editor, and it too can be selected to participate in hill-climbing at any time. Finally, if you select a number in the *None* category (number 2, say) then no patch is selected for number 2 at this time.

Additional Options The **Options** button holds some additional features besides just copying to the archive.

- **Keep Patch** Loads the patch into the current patch in your patch editor.
- **Edit Patch** Creates a new patch editor and loads the patch into that. Note that if you edit the patch in its editor, the *changes you make will be automatically reflected here*.
- **Save to File** Saves the patch to a file.
- **Load from File** Changes the patch to one loaded from a file.
- **Nudge Candidates to Me** Nudges all the candidates towards the given patch.

9.3 How the Hill-Climber Makes New Sounds

The Hill-Climber builds new sounds by recombining your top three sound choices and adding some degree of noise (mutation) to them. Let's say that your top three sounds were sounds **A, B,** and **C**, and that your *previous* "A" from last time is called **Z**. You can also select only two top choices (A and B), or single choice (A). Figure **??** explains the mechanism by which the 16 new patches are generated through various recombination and mutation mechanisms for the first 16 patches. If you have chosen to use 32 candidate patches, then the second 16 are done exactly like the first 16, except with twice the mutation rate.

The hill-climber tries to balance coming up with mixtures of your selections as well as things well outside the searched space (balancing so-called *exploration* and *exploitation*). It also attempts to move further in the direction you had just moved, in the hopes that you'll like even more of what you had selected.

When you fire up the Hill-Climber the first time, it doesn't have three candidates yet, nor does it have a Z. So instead what it does is take your current patch and build sixteen sounds as follows:

- 1–4: Four mutated versions of the current patch
- 5–8: Four twice-mutated versions of the current patch
- 9–12: Four thrice-mutated versions of the current patch
- 13–16: Four 4x mutated versions of the current patch

Again, if you have chosen to do 32 patches rather than 16, then the second 16 are done in the same way, but with twice the mutation rate.

9.4 How Recombination and Mutation Work

Edisyn's patch merging, patch mutation (randomization), nudging, and hill-climber all rely on certain low-level patch manipulation operations to do their magic. These operations are **mutation**, **recombination**, and **opposite-recombination**.

All three of these operations take a **weight** (a value between 0.0 and 1.0) which specifies how strong an effect the operation will have. Sometimes this weight specified the *probability* that a patch parameter will change. Other times the weight influences the *amount* the patch parameter will change by.

Patches are lists of parameters. Each parameter takes one of several forms:

- A *metric parameter* can take on a value from number range, such as from 0...127.
- A *non-metric* parameter can take on a value from a set of unordered elements: for example, a set of waves, or a set of filter types.

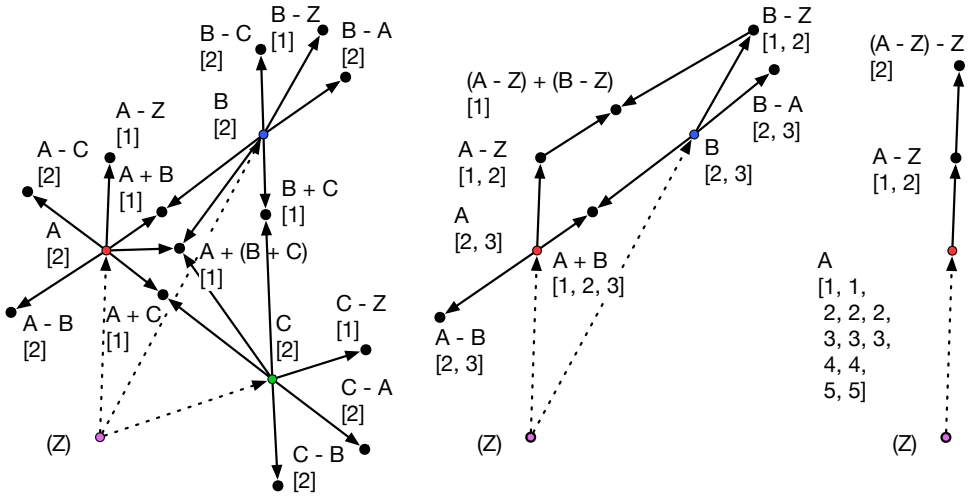

Figure 8: Recombination and Opposite-Recombination in the Hill-Climber. Shown are (left to right) basic operations performed when the user has made three (A, B, C), two (A, B), or only one (A) selections. Z is the top selection the previous iteration (the previous iteration's A). Operations are combinations of basic Edisyn recombination procedures. Specifically, $A + B$ means ordinary recombination between A and B, weighted towards A. Whereas $A - B$ means opposite-recombination, finding a point on the other side of A from the location of B. Each operation is accompanied by numbers in brackets, such as [2, 3]. In this example, this means that two children will be produced using this particular operation: one (2) will be then mutated twice, and the other (3) three times. Recombination and Opposite-Recombination are always done with a weight of 0.75; but mutation is done with a weight specified by the user on the panel.

- A *boolean* parameter can take on only two values (0/1, or on/off, of triangle/saw, etc.). Boolean parameters are assumed to be non-metric parameters.
- A *semi-metric* parameter has both a metric range and a non-metric range. For example, a MIDI Channel parameter might have the metric values 1...16 plus the non-metric options OMNI and OFF.
- Some parameters are *immutable*: they will refuse to be modified. These are typically things like patch names. You can make other parameters immutable by turning whether they can be edited on or off (see Section **??**).

Mutation When a patch **A** is *mutated*, each of its parameters is modified with some degree of random noise, or with some probability. Figure **??** shows the procedure for mutating a single parameter P in a patch. The parameter has the current value v and the user-specified *weight*. The new parameter value is returned. Mutation is used in patch randomization in nudging, and in Hill-Climbing.

Notice that if v is a metric value, and the algorithm has decided it will stay a metric value, then it uses a special sub-procedure, *MetricMutate*, to mutate the value to something relatively near to the original; the distance is determined by the weight. Otherwise generally new (metric, non-metric) values are chosen completely at random with a certain probability based on the weight.

Recombination This takes two values v and w for a given parameter P and returns a new value for v. If both values are metric, then a value is selected between them with a certain probability. Else v either stays as v or changes to w with a certain probability. See Figure **??**. Recombination is used in the patch-merge operation, as well as forming the "plus" operation in the Hill-Climbing procedure (see Figure **??** and Table **??**).

Algorithm 1 MutateM(Parameter P with value v; *weight*)

$l, h \leftarrow$ metric min and metric max of P, respectively
repeat
  $\delta \leftarrow N(0,\ 0.5 \times (\frac{1}{1-weight} - 1)) \times (h - l) \mod 2$ $N(\mu, \sigma)$: *Normal*
until $l - 0.5 < v + \delta < h + 0.5$
return $v + \delta$ rounded to the nearest integer

Algorithm 2 Mutate(Parameter P with value v; *weight*)

if P is metric **then**
  return MutateM(P, v, *weight*)
else if P is semi-metric and v is a metric value **then**
  if probability 0.5 **then**
    return MutateM(P, v, *weight*)
  else if probability *weight* **then**
    return a random $p_{\text{non-metric}} \in P$
else if P is semi-metric and v is not a metric value **then**
  if probability *weight* **then**
    if probability 0.5 **then**
      return a random $p_{\text{metric}} \in P$
    else
      return a random $p_{\text{non-metric}} \in P$
else if probability *weight* **then**
  return random $p_{\text{metric}} \in P$
return v

Figure 9: The **Mutate** function, which calls the **MetricMutate** sub-function as necessary for metric parameters or semi-metric parameters in the metric range. The "metric min" and "metric max" are the maximum and minimum values in the metric range of parameter P.

Opposite Recombination This is used to find a new patch Q which is on the *other side* of patch P from where patch S is. There are two reasons you might want to do this. One obvious reason is because you want to essentially move P away from S (so P becomes Q). This is called *fleeing*: it's used in Edisyn for *nudging away* from a target patch.

The other reason is because you have recently moved from S to P, and now you'd like to move P even *further* in that direction. This is the "minus" operation done the Hill-Climbing procedure (see Figure **??**). You'll note from Figure **??** that the Opposite procedure takes a *flee* argument to distinguish between your two reasons. This largely influences what will happen if the two parameter values are the same.[12].

[12]Note that we assume that all metric parameters are integers, not real-valued

Algorithm 3 Recombine(Param P with values v, w; *weight*)

if both v and w are metric values in P **then**
  $q \leftarrow v - weight \times (v - w)$, rounded towards w
  return a random uniform selection from $[v, q]$
else if probability *weight* **then**
  return w
return v

Algorithm 4 Opposite(Param P with values v, w; *weight*)

if both v and w are metric values in P **then**
  $q \leftarrow v + weight \times (v - w)$, rounded away from w
  $q \leftarrow \min(\max(q, \text{ metric min of } P), \text{ metric max of } P)$
  return a random uniform selection from $[v, q]$
return v

Figure 10: The **Recombine** and **Opposite** functions. Each take a parameter with *two* values v and w (one from each patch being recombined), plus a user-specified *weight*. Additionally the Opposite function takes a boolean argument, *flee*, which indicates whether Opposite is being presently used to flee away from w. The "metric min" and "metric max" are the maximum and minimum values in the metric range of parameter P.

10 Writing a Patch Editor

So you want to write a patch editor? They're not easy. But they're fun! Here are some hints.

10.1 Step One: Understand What You're Getting Into

Make sure you understand that Edisyn can only go so far to help you in writing a patch editor: but synthesizer sysex world is an inconsistent, buggy mess.

For example, the Waldorf Blofeld's multimode sysex is undocumented and must be reverse engineered. The PreenFM2 bombs when it receives out-of-range values over NRPN, but happily sends them to you. The PreenFM2 has sysex files for its patches, but they are undocumented and are basically unusable memory dumps of IEEE 754 floating-point arrays. The Yamaha TX81Z requires not one but *two* separate sysex patch dumps in a row, in order to be backward compatible with an earlier synth family nobody cares about: it also is incapable of writing a patch (likewise the PreenFM2). And it too bombs if you send it invalid data. The Kawai K4's sysex documentation is riddled with incredible numbers of errors. The Matrix 1000 accepts patch names but doesn't store or emit them: it just ignores them. Synths often pack multiple parameters into the same byte, making it impossible to update just a single parameter: you have to update five at a time. There are multiple different strategies for packing data of size 8 bits and up. Some synths, like the Futuresonus Parva, and Yamaha DX7, are highly regular in their format, while others, like the infamous Korg Microsampler, require custom tables for nearly every parameter.

Below is a little table of the current patch editors for Edisyn, and various Edisyn capabilities that they can or cannot take advantage of.

	Send Parameter	Receive Parameter	Request Specific Patch	Request Current Patch	Send to Current Patch	Send to Specific Patch	Write to Specific Patch	Change Mode	Receive Error or Ack	Standard Sysex File
Waldorf Microwave II/XT/XTk	✓	✓	✓	✓*	✓	✓	✓	✓		✓
Waldorf Blofeld	✓*	✓*	✓	✓*	✓	✓	✓			✓
Kawai K1/K1r/K1m	✓*		✓				✓		✓	✓
Kawai K4/K4r	✓*		✓		✓	✓	✓		✓	✓
Kawai K5/K5m	✓	✓	✓			✓	✓		✓	✓
Yamaha DX7	✓	✓				✓				✓*
Yamaha TX81Z	✓*			✓	✓					✓
PreenFM2	✓	✓	✓	✓	✓	✓				
Oberheim Matrix 1000	✓	✓	✓		✓*	✓*	✓			✓
Korg Microsampler	✓*	✓*								
Korg SG Rack			✓	✓	✓	✓	✓	✓		✓
Korg Wavestation	✓		✓	✓*	✓*	✓	✓	✓*		✓

** With significant caveats or restrictions*

I particularly love how the Korg Microsampler and the Korg SG are disjoint in their abilities; yet they're from the same company. Long story short, you'll probably have to do a lot of customization. I've tried to provide many customization options in Edisyn. If you need something Edisyn doesn't provide, contact me.

10.2 Step Two: Set Up the Development Environment

Still not scared away? Okay, we'll start by getting Edisyn set up for development. Probably the easiest way to fire up Edisyn for purposes of testing is as a build directory. You just need to add two items to your CLASSPATH:

1. The `coremidi4j-1.1.jar` file, located in the `libraries/` folder (you can move it where you like). This jar file contains the CoreMidi4J library, which enables sysex to work properly on Macs (you'll need it for non-Macs too).

2. The `trunk` directory. This parent directory holds the `edisyn` package. Or if you like, make some other directory `foo` and move (or link) `edisyn` into that directory, then add `foo` to your CLASSPATH.

Now you can compile Edisyn with `javac edisyn/*.java edisyn/*/*.java edisyn/*/*/*.java`
You can then run Edisyn as `java edisyn.Edisyn`

10.3 Step Three: Create Files

Let's say you're adding a single (non-multimode) patch editor for the Yamaha DX7.

Make a directory called `edisyn/synth/yamahadx7`. This directory will store your patch editor and any auxiliary files. Next copy the file `edisyn/synth/Blank.java` to `edisyn/synth/yamahadx7/YamahaDX7.java`. That'll be your patch editor code. Also copy the file `edisyn/synth/Blank.html` to `edisyn/synth/yamahadx7/YamahaDX7.html`. This will be the "About" documentation for your file. You'll eventually fill it out.

Modify the `YamahaDX7.java` file to have the proper class name and package.

Edit the `edisyn/Synth.java` file. In that file there is an array called:

```
public static final Class[] synths
```

Add to this array your class:

```
edisyn.synth.yamahadx7.YamahaDX7.class,
```

Now Edisyn knows about your (currently nonexistent) patch file.

Finally, implement the `getSynthName()` and `getHTMLResourceFileName` methods in your class file, along these lines:

```
public static String getSynthName() { return "Yamaha DX7"; }
public static String getHTMLResourceFileName() { return "YamahaDX7.html"; }
```

10.4 Step Four: Get the UI Working

This is mostly writing the class constructor and subsidiary functions. Typically you will create one `SynthPanel` for each tab in your editor. A SynthPanel is little more than a JPanel with a black background: you can lay it out however you like. However Edisyn typically lays it out as follows:

1. At the top level we have a **VBox**. This is a vertical Box to which you can add elements conveniently. You can also designate an element to be the **bottom** of the box, meaning it will take up all the remaining vertical space.

2. In the VBox we will place one or more **Categories**. These are the large colorful named regions in Edisyn (like "LFO" or "Oscillator").

3. Typically inside a Category we'd put an **HBox**. This is a horizontal box to which you can add elements. You can also designate an element to be the **last item** of the box, meaning it will take up all the remaining horizontal space. By doing this, the Category's horizontal colored line nicely stretches the whole length of the window.

4. Inside the HBox you put your widgets. You might lay them out with additional VBoxes and HBoxes as you see fit. It's particularly common to one or more small widgets (check boxes, choosers) in a VBox, which will cause them to be top-aligned rather than vertically centered as they would if they were stuck directly in the HBox. It's helpful to look at existing patch editors to see how they did it.

5. If you need multiple rows, you should put a VBox in the Category, and then put HBoxes inside of that.

6. You might have multiple Categories on the same row. To do this, just put them in an HBox. Make sure the final Category is designated to be the Last Item of the HBox. You'd put this HBox in the top-level VBox instead of the Categories themselves.

The first category is the **Synth Category**. It is typically named the same as `getSynthName()`, its color is `edisyn.gui.Style.COLOR_GLOBAL`, and contains the patch name and patch/bank information, and perhaps a bit more (for example, Waldorf synthesizers have the "category" there too).

To the right of the first category is usually (but not always) various global categories. They're usually `edisyn.gui.Style.COLOR_A`.

If you have additional categories, you might distinguish them using `edisyn.gui.Style.COLOR_B`, and eventually `edisyn.gui.Style.COLOR_C`.

You can lay out the rest of the categories as you see fit.

Think about Parameters Synthesizer parameter values will be stored in your Synth object's **Model**. These parameters will be stored in your synth's **Model** object. Each parameter has a **Key**. Edisyn traditionally names the keys all lower case, plus numbers, with no spaces or hyphens or underscores, and tries to keep the keys fairly similar to how your synth sysex manual calls them. They're usually described with a category descriptor (such as `op3` and then the parameter name proper (such as *envattack*), resulting in the final key name *op3envattack*. Various global parameters are just the parameter name: for example, it's standard in Edisyn that the patch name be just called *name*, the patch number is called *number*, and the bank number is called *bank*.

Often parameters (as set by widgets) are exactly the same as the various elements you send and receive to the synthesizer. But sometimes they're not. Many synthesizers pack multiple parameters (like LFO Speed + Latch) together into a single variable, which is very irritating. You want to lay out what the *real* parameters of your synthesizer are, that the user would be modifying, not what you'd be packing and sending to the synth.

Another issue is how your synthesizer interprets values sent over sysex or NRPN. Consider BPM for a moment. Perhaps your synthesizer has BPM values of 20...300, and there are missing values (for example, there's no 21). The actual values are mapped to the numbers 0...127. What values should you store? In my patch editors, I store the values in the model as 0...127, which makes it easy to emit them. But then I have to have an elaborate conversion function to map them to 20...300 for display on-screen.

Also some synthesizers have holes in their ranges. For example, they might permit the values 0...17 and the values 20...100, but do not permit 18 and 19. What to do then? You probably ought to compact them to be contiguous between some min and max: for example, you might compact it to 0...98. When displayed, use a custom displayer, and when emitting or parsing them, you'll have to map them to your internal representation accordingly.

In summary: your internal parameters ought to have contiguous ranges and should make sense from the user's perspective and not the synthesizer's weird parsing perspective.

So how to set parameters? You usually don't add the key yourself, though you could. Instead, normally you tell the widget the name of the parameter it's modifying (the key), and it adds it to the model on its own. Parameters are either **strings** or are **numbers**. Numerical parameters all have a **min** and a max value, inclusive: usually the widget will set those for you. They also may have a **MetricMin** and **MetricMax** value, and you may need to set those manually.

MetricMin and MetricMax work like this. Some numerical parameters are **metric**, meaning they're a range of numbers where the order matters, such as 0–127. Other numerical parameters are **categorical** (or "non-metric"), meaning that the numbers just represent an ID for the parameter. For example, a list of wavetables is categorical: it doesn't *really* matter that wavetable 0 is "HighHarm3": it's just where it's stored in your synth.

Edisyn is smart about mutating and recombining metric parameters, but for non-metric ones it just picks a new random setting. Sometimes your parameters are *both* metric and non-metric. For example, some parameter might have the values 1–32 plus the non-metric values "off" (0), "uniform" (17), and "multi" (18), or whatever. In this case, your min is 0 and your max is 18. But your *metric min* is 1 and your *metric max* is 16. This tells Edisyn that values outside the metric min / metric max range should be treated as non-metric.[13] If you have this situation, you'll need to set the Metric Min and Metric Max manually.

Parameters can be declared **immutable**, meaning Edisyn can't mutate them or cross them over at all. Also, all string parameters are automatically immutable. You'll need to declare the others.

[13] What if your synth has metric values on the outside and non-metric value on the inside? Edisyn can't handle that. Thankfully I've not seen it yet.

Copying and Distributing Parameters If your synth has multiple copies of the same category (for example, multiple LFOs), you can *copy* parameters wholesale from one category to another. To do this, parameters must obey a certain convention. Specifically, parameters in a category must all start with the same *preamble*, which must contain no digits, followed by a *category number*, which must be all digits. After that, you can have whatever you like. For example, `lfo1rate` or `osc14attack`. If you have four LFO categories, their category numbers might be 1...4, say. After you have set up your parameters appropriately, you can turn on copy and paste in a given category by calling **makePasteable(*preamble*)**, passing in the preamble (not the category number).

Categories also often contain multiple instances of a given parameter. For example, a step sequencer category might contain 16 steps. You can *distribute* values to all such parameters if you follow the a similar convention, specifically, parameters should start with a *preamble*, and then the first string of digits will refer to the index of the parameter. For example, if your step sequencer had `seq` as its preamble, perhaps you might have `seqstep1` through `seqstep16`. You an have additional text, such as `seqstep1attack` or whatnot. After you have set up your parmaters, you can turn on distribution by calling **makeDistributable(*preamble*)**, passing in the preamble.

Your category can also do both of these things. In this case, all parameter names should obey the copy/paste convention, and distributable parameters should have a *second* string of digits somewhere later in the parameter name which refers to the parameter index. For example, you might have `seq1step1` through `seq1step16` for sequencer 1, and `seq2step1` through `seq2step16` for sequencer 2.

Finally, by default categories can be *reset*. It's probably wise to turn this off in the global category. This is done by calling **makeUnresttable()**.

Common Widgets Edisyn has a number of widgets available. Most widgets are associated with a single parameter (a "key"). There is no reason you can't have multiple widgets associated with the same key: when that parameter is updated, all associated widgets are updated.

The most common widgets are:

- **StringComponent** This is the only String widget. It's used for patch names. For a patch name, you typically implement it like this:

```
String key = "name";  // the key in the model
String instructions = "Name must be up to 10 ASCII characters.";
JComponent comp = new StringComponent("Patch Name", this, "name", maxLength, instructions)
    {
    public String replace(String val)
        {
        return revisePatchName(val);
        }
    public void update(String key, Model model)
        {
        super.update(key, model);
        updateTitle();
        }
    };
```

  In conjunction with this, you will want to override the **revisePatchName(...)** method in your Synth subclass. This method modifies a provided name and returns a corrected version. The default version, which you might call first (via super), removes trailing whitespace. You can then revise incorrect characters, length, and so on.

- **Chooser** This is a pop-up menu or combo box, and it's a numerical component. You provide it with an array of strings representing the parameter values 0...*n*. For example, you might set up a wavetable chooser as:

```
String key = "wave";  // the key in the model
String[] params = WAVE_OPTIONS;  // this is an array of wave names elsewhere
JComponent comp = new Chooser("Wave", this, key, params);
```

There's an option to add images to the chooser's menu:

```
public static final ImageIcon[] MY_WAVE_ICONS =
    {
    new ImageIcon(YamahaDX7.class.getResource("Wave1.png")),
    new ImageIcon(YamahaDX7.class.getResource("Wave2.png")),
    ... // and so on
    };
String key = "wave";  // the key in the model
String[] params = WAVE_OPTIONS;  // this is an array of wavetable names elsewhere
JComponent comp = new Chooser("Wave", this, key, params, MY_WAVE_ICONS);
```

These PNG files would be stored in your `edisyn/synth/yamahadx7/` directory. They should be no taller than 16 pixels high: OS X refuses to display comboboxes with icons taller than that.

- **Checkbox** This is a simple checkbox. By default it's on, but there's a setting to have it by default be off. On is 1 and Off is 0 as stored in the model.

```
String key = "arpeggiatorlatch";  // the key in the model
JComponent comp = new CheckBox("Arpeggiator Latch", this, key);
```

The's a bug in OS X which mis-measures the width of the string needed, so you might see "Arpeggia..." instead of "Arpeggiator Latch" on-screen. To fix this, just add a tiny bit to the width: usually one or two pixels are enough:

```
String key = "arpeggiatorlatch";  // the key in the model
JComponent comp = new CheckBox("Arpeggiator Latch", this, key);
((CheckBox)comp).addToWidth(1);
```

- **LabelledDial** This is a labelled dial representing a collection of numbers from some min to some max.

```
int min = 1;
int max = 16;
Color color = edisyn.gui.Style.COLOR_A;  // Make this the same color as the enclosing Category
JComponent comp = new LabelledDial("MIDI Channel", this, "midichannel", color, min, max);
```

It's common that you need more lines in your label. Perhaps you might say:

```
int min = 1;
int max = 16;
Color color = edisyn.gui.Style.COLOR_A;  // Make this the same color as the enclosing Category
JComponent comp = new LabelledDial("Incoming", this, "midichannel", color, min, max);
((LabelledDial)comp).addAdditionalLabel("MIDI Channel");
```

You can add additional (third, fourth, ...) labels too. Note that you can change the first label text later on (with **setLabel(...)**) but you can't change the label text of additional labels.

It is very common to need a custom string display for certain numbers in the center of the dial. You can do it like this:

```
int min = 0;
int max = 17;
Color color = edisyn.gui.Style.COLOR_A;  // Make this the same color as the enclosing Category
JComponent comp = new LabelledDial("MIDI Channel", this, "midichannel", color, min, max)
    {
    public String map(int val)
        {
        if (val == 0) return "Off";
        else if (val == 17) return "Omni";
        else return "" + val;
        }
    };
```

Note that if you're just trying to subtract a certain amount from the dial, for example, to display the values 0...127 as the values -64...63, then there's a constructor option on LabelledDial for this:

```
new LabelledDial("Pan", this, "pan", color, 0, 127, 64)  // subtracts 64 before displaying
```

This brings us to the discussion of *symmetry*. Sometimes you want the dial to be symmetric looking, and sometimes not. Edisyn tries hard to see to it that, whenever possible, the "zero" position on the dial is vertically directly above or directly below the center of the dial. For example, a symmetric dial going from -100 to $+100$ would have zero at the top: and a dial going from 0 to 127 would have zero at the bottom (this second case results in Edisyn's unusual "C"-shaped dials). The "zero" position doesn't always mean 0: it should be the notional identity for the dial. For example, a Keytrack dial might have 100% be the identity position.

By default Edisyn's dials assume that the zero position is at the beginning of the dial, resulting in the "C" shape. Because a great many synthesizers go from 0...127 or from 0...100, if you use the aforementioned constructor option to subtract either 64 or 50 from the dial, Edisyn will automatically make it look symmetric.

Sometimes you need to customize the orientation in order to keep the zero position vertically centered. For example Blofeld's Arpeggiator has a variety of dials which aren't *quite* symmetric, because there are some unusual options at the start, as shown on the top figure at right. But even worse: the Kawai K4 Effects patch has a number of dials which look like a *reversed* "C" because of so many additional options loaded at the end of the dial, as shown on the bottom figure.

You can customize the orientation in two ways. First, if you override LabelledDial's **isSymmetric()** method to return **true**, then the dial will display itself as fully symmetric. Second, you could override LabelledDial's **getStartAngle()** method to return the desired angle of the start (leftmost) position of your curve. The default is 270 (the "C"), and when fully symmetric it's 90 + (270 / 2).

When the user double-clicks on a LabelledDial, try to have the LabelledDial go to some default position. This is often the "zero" position: but sometimes it's not. At any rate, it's almost always most common position the user would want, whatever that is. By default the "default position" is the first position if asymmetric, and the center position if symmetric. You can change the default position by overriding LabelledDial's **getDefaultValue()** method to return a different value.

Last but not least! If you have a mixture of metric and non-metric values (for example, 0="Off", 1...32 = 1...32, and 33="Uniform"), you will need to modify the MetricMin and MetricMax declarations. Normally LabelledDial declares MetricMin to be the same as Min and MetricMax to be the same as Max. But in this example, your minimum metric value is 1 and your maximum metric value is 32.

```
getModel().setMetricMin("whateverkey", 1);
getModel().setMetricMax("whateverkey", 32);
```

It sometimes happens that *none* of the LabelledDial values should be thought of as metric. For example, a previous code example, we were using the LabelledDial to select the MIDI Channel. Now, channels aren't metric: they're just 16 unique labels for channels which happen to be numbers. In this case, we should remove the metric min and max entirely, so Edisyn considers the entire range to be non-metric. To do this, we say:

```
getModel().removeMetricMinMax("midichannel");
```

- **IconDisplay** This displays a different icon for each value in your model. You can't change the values by clicking or dragging on an IconDisplay: instead, use a separate LabelledDial or Chooser.

  ```
  ImageIcon icons = MY_ALGORITHM_ICONS;
  JComponent comp = new IconDisplay("Algorithm Type", icons, this, "algorithmtype");
  ```

  Your images can be PNG or JPEG files: I suggest PNG. You might create an instance variable like this:

  ```
  public static final ImageIcon[] MY_ALGORITHM_ICONS =
      {
      new ImageIcon(YamahaDX7.class.getResource("Algorithm1.png")),
      new ImageIcon(YamahaDX7.class.getResource("Algorithm2.png")),
      ... // and so on
      };
  ```

  These PNG files would be stored in your `edisyn/synth/yamahadx7/` directory.

- **KeyDisplay** This displays a keyboard. You specify the min and max keys (which *must* be white keys), and a transposition (if any) between keys and the underlying MIDI notes actually generated. When the user chooses a key, the KeyDisplay will update a value 0...127 corresponding to the equivalent MIDI note value.

  The KeyDisplay can update *dynamically* or *statically*. When dynamic, then every time you scroll through the display and a note is highlighted, the model is updated. When static, the model is only updated when a note is finally chosen and the user has released the mouse button. To set this, use **setDynamicUpdate(...)**.

  You will probably want your KeyDisplay to update in concert with a LabelledDial. This is easy: just set them to the same key in the model. but synthesizers are inconsistent in how they describe notes, because MIDI didn't specify a notation. For example, MIDI note 0 is "C -2" in Yamaha's notation (also adopted by Kawai and some others), or it is "C -1" in *Scientific Pitch Notation* (or SPN[14]), or just play "C 0" in simple MIDI notation. You can specify this by calling the method **setOctavesBelowZero(...)**.

  In some cases you might wish to be notified whenever the user *clicks* on a key, or drags to it, rather than when the key actually is updated (which might only happen on button release). Typically this happens because you want to actually play the note so the user gets some feedback. To be notified of this, just override the method **userPressed(...)**.

- **PushButton** This doesn't maintain a parameter at all: it's just a convenience cover for JButton. You see it in Multimode patches where pressing it will pop up an equivalent Single patch (it's usually called "Show"):

[14]... or *American Scientific Pitch Notation*(ASPN), or *International Pitch Notation* (IPN). They're all pretentious names.

```
JComponent comp = new PushButton("Show")
    {
    public void perform()
        {
        // do your stuff here
        }
    };
```

Popping up new synth panels from a multimode panel is complex. Take a look at how `edisyn/synth/waldorfmicrowavext/WaldorfMicrowaveXTMulti.java` does it.

- **PatchDisplay** This displays your patch and bank in a pleasing manner.[15]

```
String numberKey = "number";  // typically or null if you have no patch numbers
String bankKey = "bank";  // typically, or null if you have no bank numbers
int numberOfColumns = 10;  // for example
JComponent comp = new PatchDisplay(this, "Patch", bankKey, numberKey, numberOfColumns)
    {
    public String numberString(int number) { "" + number}  // format as you like
    public String bankString(int bank) { "" + bank}  // format as you like
    };
```

- **EnvelopeDisplay** This displays a wide variety of envelopes. Envelopes are drawn as a series of points, and between every successive pair of points we draw a line. You will provide the EnvelopeDisplay with several arrays defining the coordinates of those points.

  There are two main kinds of envelopes your synthesizer might employ. First, your synthesizer might define parameters (like attack) in terms of the *height* of the attack and also the *amount of time* necessary to reach that height. This is intuitive to draw, but in fact many synthesizers don't do it that way. Instead, some define it in terms of the *height* of the attack and the *rate of change* (or slope, or angle). In the first case, the height of the attack has no bearing on how long it takes to reach it. But in the second case, the amount of time to reach the attack depends on both the height and on the rate. This is even further complicated by some synthesizers (like Yamaha's) which use rate, but compute it not in terms of angle, but in terms of (essentially) 90 degrees *minus* the angle. Thus a steeper rate is a *lower number*. You will need to figure out what your synthesizer does exactly.

  Let's say your synth does the easy thing and computes stuff in terms of height and amount of time. Then you set up an Envelope Display with four elements:

  - An array of keys (some of which can be null) of the parameters which define the *amount of time* for each segment. If a key is null, the parameter value is assumed to be 1.0.
  - An array of keys (some of which can be null) of the parameters which define the *height* for each segment. If a key is null, the parameter value is assumed to be 1.0.
  - An array of constant doubles which will be multiplied against the time parameters. You want these constants to be such that, when the time parameters are at their maximum length, their values, multiplied by these constants, will sum to no more than 1.0
  - An array of constant doubles which will be multiplied against the height parameters. You want these constants to be such that, when the any given height parameter is maximum, when multiplied against the constant it will be no more than 1.0.

[15] *Why is PatchDisplay so elaborate? Why not just use a JLabel or something?* Originally PatchDisplay did other complex things like change color. Now it doesn't.

Here's how you'd make an Envelope Display for an ADSR envelope where each of the values varies 0...127:

```
String[] timeKeys = new String[] { null, "attack", "decay", null, "release" };
String[] heightKeys = new String[] { null, "attackheight", "sustain", "sustain", null };
double[] timeConstants = new double[] { 0.0, 0.25 / 127, 0.25 / 127, 0.25, 0.25 / 127 };
double[] heightConstants = new double[] { 0.0, 1.0 / 127, 1.0 / 127, 1.0 / 127, 0.0 };
JComponent comp = new EnvelopeDisplay(this, Color.red, "ADSR",
                timeKeys, heightKeys, timeConstants, heightConstants);
```

Notice that `"sustain"` is used twice: thus the line stays horizontal; and furthermore its time constant is fixed to 0.25 so it always takes up 1/4 of the envelope space. Also notice that in this example the beginning and end of the ADSR envelope are fixed to 0.0 height. That doesn't have to be the case. And maybe you don't have an attack height: it's always full-on attack. Then you'd say:

```
String[] heightKeys = new String[] { null, null, "sustain", "sustain", null };
double[] heightConstants = new double[] { 0.0, 1.0, 1.0 / 127, 1.0 / 127, 0.0 };
```

It's possible that your envelope isn't always positive: it can go negative. The EnvelopeDisplay assumes that your parameters are all positive numbers (like 0–127), but it does allow to draw a line indicating where the X axis should be, via the **setAxis(...)** method. See the fourth example in Figure **??**.

You also can also tell the EnvelopeDisplay to draw a vertical line at some key position and a dotted line at another, using the methods **setFinalStageKey(...)** and **setSustainStageKey(...)** respectively (these are named after their use in the Waldorf Microwave XT). These keys should specify the *stage number* (the point) where the line is drawn. For example, if the sustain stage key's value is 4, then the line should be drawn through point number 4 (zero-indexed) in the envelope. See the third example in Figure **??**.

You can also specify two intervals with start and stop keys respectively. At present the EnvelopeDisplay supports two intervals. These are set up with **setLoopKeys(...)**. These keys should specify the *stage number* (the point) where the intervals are marked. For example, if the interval end's key value is 4, then the end should be marked exactly at point number 4 (zero-indexed) in the envelope. Again, see the third example in Figure **??**.

You can also postprocess the sustain stage, final stage, or loop keys with **postProcessLoopOrStageKey(...)**. This function takes a key and its value, and returns a revised value, perhaps to add or subtract 1 from it.

What if your synth uses angles/rates/slopes rather than time intervals? For example, the Waldorf Blofeld does this. To handle this situation, we add an additional array of double constants called *angles*. It works like this. The height keys and height constants are exactly as before. And `timeConstants[0]` still defines the x position of the first point in the envelope, as before. But the other time constants work differently.

Specifically, to compute the X coordinate of the next point, we take its key value and multiply it by the corresponding angle, and then take the absolute value. This tells us the *positive angle* of the line. Angles can never be negative: whether the line has a positive or negative slope is determined entirely by the relative position of the height keys.

Since angles can and will create very strung-out horizontal lines, the remaining time constants tell us the *maximum length* of a line: these again should sum to 1.0.

Angles/rates create weird idiosyncracies you'll have to think about. For example, below is the code for the Blofeld's ADSR envelope. As the Sustain gets higher, the Release gets longer but the Decay gets

shorter, because the synth is basing this envelope on *rate* and not *time*.[16] One consequence of this is that the Decay and Release together are as long as the Attack, because if you're basing on rate, then the amount of time to go up is the same as the total amount of time to go *down*, and both Decay and Release go down. Thus we have a *max width* of 1/3 for all four portions: but at any time they can only sum to 1/3 [attack] + 1/3 [sustain] + 1/3 [decay + release].

In the Blofeld ADSR, all the values go 0...127, and the angles are displayed by Edisyn to go from vertical to $\pi/4$ (we don't want them too flattened out). See if the code below makes sense now:

```
String[] timeKeys = new String[] { null, "attack", "decay", null, "release" };
String[] heightKeys = new String[] { null, null, "sustain", "sustain", null };
double[] timeConstants = new double[] { 0, 0.3333, 0.3333,  0.3333, 0.3333};
double[] heightConstants = new double[] { 0, 1.0, 1.0 / 127.0, 1.0/127.0, 0 };
double[] angles = new double[] { 0, (Math.PI/4/127),   (Math.PI/4/127), 0, (Math.PI/4/127) };
JComponent comp = new EnvelopeDisplay(this, Color.red, "ADSR",
                timeKeys, heightKeys, timeConstants, heightConstants, angles);
```

This *still* might not be flexible enough for you. For example, the Yamaha TX81Z has, shall we say, an unusual approach to defining angles. You can do further post processing on the $\langle x, y \rangle$ coordinates of each of the points (where both X and Y vary from 0...1) by overriding the **postProcess(...)** method like this:

```
JComponent comp = new EnvelopeDisplay(this, Color.red, "ADSR",
                timeKeys, heightKeys, timeConstants, heightConstants, angles)
    {
    public void postProcess(double[] xVals, double[] yVals)
        {
        // modify xVals and yVals as you see fit.
        }
    };
```

Envelopes generally stretch to fill all available space: they're particularly good to put as the "last" element in an HBox via addLast(). But you might want to add them elsewhere and fix them to a specific width. In this case, just call **setPreferredWidth(...)**.

- **Spacers** Occasionally you might need to add some fixed space to separate widgets. See the `Strut` class for factory methods that can build some struts for you.

Dynamically Changing Widgets One gotcha which shows up in a number of synthesizers (particularly in effects sections) is that if you change (say) the effect type, the number of available parameters, and their names, will change as well. Eventually Edisyn will have a widget that assists in this, but for now you'll have to manually add and remove widgets.

Edisyn's patch editors usually do this by defining a bunch of HBoxes, one for each effect type, and then remove the current HBox and add the correct new one dynamically in response to the user changing types. You can see a simple example of this in the Waldorf Microwave XT code, and a more elaborate version in the Blofeld code (where different effects actually share specific widgets).

[16] Edisyn no longer displays this way for the Blofeld, because although the Blofeld indeed follows angle/rate, for large values the Blofeld's functions start getting close to following time. The problem is that while the Blofeld documentation acknowledges that it follows angle/rate, the Blofeld's *screen* incorrectly displays envelopes following time! When I wrote this documentation I was using angle/rate for the Blofeld because it's the "true" underlying behavior, but I've since changed the patch editor back to displaying time because using something other than what's on the Blofeld screen would really confuse owners, and in the Blofeld's case it's a subtle difference.

You'll have to manually remove and add these widgets or HBoxes. But when should you do so? That's pretty easy: when the effect type has been updated. Typically the effect type is shown as a Chooser, and when it is updated, the Chooser's **update(...)** method is called:

```
JComponent comp = new Chooser("Effect Type", this, "effecttype", types)
    {
    public void update(String key, Model model)
        {
        super.update(key, model);  // be sure to do this first
        int newValue = model.get(key, 0);  // 0 is the default if the key doesn't exist, but it will.

        // now do something according to the value newValue
        }
    };
```

You'll see various patch editors have implemented update(...) for various purposes.

Hand in hand with this: in some cases you want the update(...) method to be called not only when the widget's key is updated, but when *some other key* is updated. To do this, you can *register* a widget to be updated for that key as well. This is done as follows:

```
model.register("keyname", widget);
```

For example, in the Yamaha TX81Z, the operator frequency is computed as a combination of three widgets: and in the final widget ("Fine") the final frequency is displayed. To do this, we have registered the "Fine" widget to revise itself (via the map(...) method) whenever any of three different parameters is updated.

10.5 Step Five: Get Input from the Synth (and File Loading) Working

There are two ways the synth can send you information: as a bulk sysex patch dump and as individual parameters. We'll start with the bulk sysex patch dump.

Bulk Dumps First, you need to implement the **recognize(...)** method. This method tells Edisyn that you recognize a bulk dump sysex message. You should verify the message length and the header to determine that it's a bulk dump and in fact meant is for your type of synthesizer *and* is probably correct. This method will also be called when loading a sysex file from disk.[17]

Next, you need to implement the **parse(...)** method. In this method you will be given a data array and your job is to set the model parameters according to your parsing of this array. You set parameters using the **set(...)** methods in the model, like this:

```
getModel().set(numericalKey, 4.2); // or whatever new value
getModel().set(stringKey, "newValue"); // or whatever new value
```

It is possible that the parse(...) method will actually contain multiple sysex messages, if you loaded from a file. For example, the Yamaha TX81Z's patch isn't a single sysex messages, it's *two* messages, to be backward compatible with an unimportant earlier synthesizer for some ridiculous Yamaha reason. When you receive a dump via the synth, it'll only be one or other other of these messages. But if you receive a TX81Z dump from a file, it'll be both messages. Thankfully, the parse() method will tell you whether you're receiving from a file or not.[18] So if you do something fancy with emit(...) later, you may need to revise your parse(...) implementation.

[17] In fact the primary purpose of this method is to recognize sysex data loaded from disk: and so other sysex messages don't have their own recognize(...) method.

[18] Though in fact the TX81Z implementation — and in fact all Edisyn's parse editors to date — don't change their parse(...) behavior when receiving from a file.

You also need to implement the **gatherPatchInfo(...)** method. This method is nontrivial to implement. Its function is to work with the user to determine the patch number, bank number, etc. necessary to ask the synthesizer for a given patch. I suggest you take a look at existing patch editors to see how they have implemented it, and largely copy that. You'll notice that patch-gathering code usually pops up a dialog box with a bunch of rows in it. How is this done? Edisyn's Synth.java class has a special method to make this easy: **showMultiOption(...)**.

Additionally, you need to override methods which issue a dump request to the synth:

- **performRequestDump(...)** or **requestDump(...)** Override *one* of these methods to request a dump from the synth of a specific patch. requestDump(...) is simpler: you just return bytes corresponding to a sysex message to broadcast to the synth. performRequestDump(...) lets you manually issue the proper commands.

  In the second case, the edisyn.Midi class, instantiated in the **midi** instance variable, has several methods for constructing MIDI messages: you can send them, or send sysex messages (as byte arrays) via the **tryToSendMIDI()** or **tryToSendSysex()** methods. Also you'll have to handle changing the patch: see the information in Blank.java's documentation on this method for an example.

  Both of these methods take a Model called **tempModel** which will hold information concerning the patch number and bank number that you should fetch. This model was built by gatherPatchInfo(...).

- **performRequestCurrentDump(...)** or **requestCurrentDump(...)** Override *one* of these methods to request a dump from the synth of the current patch being played. These methods are basically just like performRequestDump(...) and requestDump(...), but they don't take a model (there's no patch number).

You will also probably need to implement **changePatch(...)** to issue a patch change (it'll be called as part of performRequestDump(...)). It's possible that your synthesizer must pause for a bit after a patch change (the Blofeld, for example, requires almost 200ms). You may want to implement the **getPauseAfterChangePatch()** method to slow Edisyn down. If your synth can't change patches to whatever you're editing, that's okay, but you'll need to handle the right behavior later on when you emit a patch to it.

If your synth cannot load the current patch you can avoid implementing some of these methods by saying the following:

```
receiveCurrent.setEnabled(false);  // turns of the "Request Current Patch" menu option
```

You should do this in an overridden version of the **sprout()** method (be absolutely sure to call super.sprout() first).

You will also want to override some other methods. First **getPatchName(model)** should extract the patch name from the provided model (probably via `model.get("name", "foo")`). Second, you also will want to override the **revisePatchName(...)** method if you've not already done so for the StringComponent widget. This method modifies a provided name and returns a corrected version. The default version, which you might call first (via super), removes trailing whitespace. You can then revise incorrect characters, length, and so on. Third, if your synthesizer uses an ID to distinguish itself from other synthesizers of the same type (the Waldorf synths do this for example), you should override the **reviseID(...)** method to correct provided IDs. If this method returns `null` (the default), the ID won't even appear as an option.

Finally, you will probably want to override the **revise()** method to verify that all the model parameters have valid values, and tweak them if not. The default version, which you can call via super, does most of the heavy lifting: it bounds the values to between their min and max. You might also verify that the patch name is correct here. See the Waldorf Blofeld code as an example of what to do.

See also the description of these methods in `edisyn/synth/Blank.java`

Individual Parameters [If your synth doesn't send out individual parameters, or you don't want to be bothered right now in handling this, you can just ignore this section for now]. Individual parameters might come in as sysex messages, as CC messages, or as NRPN. Here are your options:

- **Sysex Messages** Here, override the method **parseParameter(...)**. Note that the provided data might be something else sent via sysex besides just a parameter change. You can test for that too (and handle it here if you like).
- **NRPN or Cooked CC messages** A cooked CC message is one which doesn't violate any of the RPN/NRPN rules (it's not 6, 38, 97, 98, 99, 100, or 101). At present Edisyn does not recognize 14-bit CC. If your messages are always cooked or are NRPN, then you can handle them via **handleSynthCCOrNRPN(...)**, which takes a special **MIDI.CCData** argument that tells you about the message (see the Midi.java class).
- **Raw CC Messages** A raw CC message is any message number 0...127 just sent out willy-nilly, not respecting things like RPN/NRPN or 14-bit CC. If your synth sends out raw CC messages, you need to override **getExpectsRawCCFromSynth()** to return **true**. Then you handle the messages via **handleSynthCCOrNRPN(...)** as discussed above.

Again, you update one or more parameters in response to these messages using one of:

```
getModel().set(numericalKey, 4.2); // or whatever new value
getModel().set(stringKey, "newValue"); // or whatever new value
```

Note on File Loading If your bulk dumps come in as sysex messages, then congratulations, you already have file loading working. If not, you will need to *invent* a bulk sysex format and implement it in the **parse(...)** (and later **emit(...)** methods even if your synthesizer never sends stuff via sysex (such as is the case in the PreenFM2). That way you can still load and save files.

You probably ought to use the "educational use" wildcard MIDI sysex ID (0x7D). Edisyn's made-up sysex header for the PreenFM2 currently looks like this: `0xF0, 0x7D, P, R, E, E, N, F, M, 2,` *version*`.` Presently *version* is `0x0`. You might do something similar.

10.6 Step Six: Get Output to the Synth (and File Writing) Working

If you've gotten this far, writing is simpler than parsing and requesting, because you've already written a lot of the support code. You can write out both bulk dumps and individual parameters (as you tweak widgets).

Bulk Dumps You will need to implement *one* of the following two methods: either **emitAll(Model, ...)** or **emit(Model, ...)**. The emit(Model, ...) method is simpler: you just build data for a sysex message and return it. In emitAll(Model, ...), you build an array consisting of *either* javax.sound.midi.SimpleMessage objects *or* byte[] arrays corresponding to sysex messages, or a mixture of the two. These will be emitted one by one. Most commonly you just override emit(Model, ...).

Both emit(Model, ...) and emitAll(Model, ...) receive a temporary model. This model will contain a small bit of data sufficient to inform you of the patch and bank number are that the patch is going to be emitted to (via Edisyn's "write" procedure). Alternatively if the *toWorkingMemory* argument is TRUE, then you're supposed to emit to current working memory (Edisyn's "send" procedure).

You may not be able to write, or you may not be able to send to a specific patch, or to the current patch, depending on your synthesizer. If so, you can do any of:

```
transmitTo.setEnabled(false);  // turns of the "Send to Patch..." menu option
transmitCurrent.setEnabled(false);  // turns of the "Send to Current Patch" menu option
writeTo.setEnabled(false);  // turns of the "Write to Patch..." menu option
```

Again, these should be set in an overridden version of the **sprout()** method. Be sure to call **super.sprout()** first, or bad things will probably happen.

Note that emit(Model, ...) and emitAll(Model, ...) are also used to write out files. If you implemented emitAll(...), be aware that Edisyn will strip out all of the javax.sound.midi.SimpleMessage messages and just pack together then remaining sysex messages. This is what will result in multiple sysex messages being read in in a single parse(...) dump, as discussed earlier.

Some synthesizers need a bit of time to rest after receiving a dump before they can do anything else. You can tell Edisyn to pause after a dump by overriding **getPauseAfterSendAllParameters()**.

Individual Parameters In response to changing a widget, Edisyn will try to change a parameter on your synthesizer. This is similar to the bulk dump. Specifically, there are two methods, **emit(String)** and **emitAll(String)**, which work like their bulk counterparts, except that they are tasked to emit a *single parameter* to the synthesizer. Implement only *one* of these methods. If you don't want to do this, just don't implement these methods.

If your synthesizer accepts NRPN (such as the PreenFM2), the Midi.java file has some utility methods for building NRPN messages easily.

It's possible that your synthesizer can only accept messages at a certain rate. You may want to implement the **getPauseBetweenMIDISends()** method to slow Edisyn down.

Bulk Dumps Via Individual Parameters Some synthesizers, such as the PreenFM2, do not accept a bulk dump method at all. Rather you send a "bulk dump" as a whole lot of individual parameter changes. If your synthesizer is of this type, you should override the method **getSendsAllParametersInBulk()** method to return **false**.

Note on File Writing See the earlier note at the end of Section **??** about File Loading: as discussed there, if your synth doesn't read or write sysex, you'll still need to *invent* a bulk sysex format, and implement it in the **emit(...)** and **parse(...)** methods, so you can save and load files to disk.

10.7 Step Seven: Create an Init File

Now that you've got everything coded and working (hah!) it's time to create an Init file. To do this, either request an init patch from the synthesizer, or create an appropriate one yourself. Then save it out as a sysex file.

Next, move that file and rename it to `edisyn/synth/yamahadx7/YamahaDX7.init`. Edisyn will load this file to initialize your patch editor. To do this, add to the very bottom of your constructor the following line:

```
loadDefaults();
```

10.8 Step Eight: Get Batch Downloads Working

Edisyn can download many patches at once. To support this, you need to implement a few methods.[19] First, there's **getPatchLocationName(...)**, which returns as a String a short version of the patch address (bank, name) to be used in a saved filename. Next, there's **getNextPatchLocation(...)** which, given a Model containing a patch address, returns a model with the "next" patch address (wrapping around to the very first address if necessary). Finally you need to implement **patchLocationEquals(...)**, which compares two patches to see if they contain the same patch address.

A few synthesizers (notably the PreenFM2) don't send individual patches as single sysex patch dumps, but rather send them as multiple separate NRPN or CC messages. Edisyn needs to know this so it can make a better guess at whether a patch dump has arrived and is ready to be saved. To let Edisyn know that your

[19] Until you implement **getPatchLocationName(...)** to return something other than `null`, Edisyn will keep the Batch Downloads menu disabled. So when you implement this method, be sure to also implement the other methods here at the same time.

patch editor is for a synthesizer of this type, override the method **getReceivesPatchesInBulk()** to return false.

Compared to the other stuff, this step is easy.[20]

10.9 Step Nine: Other Stuff

You're almost done! Some other items you might want to do. First, you may need to tweak the mutability of parameters. No string parameters are mutable, but by default all numerical ones are (including checkboxes and choosers). Occasionally you'd want to make some of those immutable so they will not be modified during merge, hill-climbing, etc. To do this, you can call **setStatus(..., Model.STATUS_IMMUTABLE)** on the model.

Second, whenever your patch editor becomes the front window, the method **windowBecameFront()** will be called. You could override this to send a special message to your synth to update it somehow. For example, the Waldorf Microwave XT patch editors send a message to the Microwave XT to tell it to switch from single to multi-mode (or back) as appropriate.

Finally when the user clicks on the close box, the method **requestCloseWindow()** is called. You can override this to query the user about saving the patch etc. first, and then finally return the appropriate value to inform Edisyn that the window should in fact be closed. Though in fact currently no patch editors implement this method at all.

10.10 Step Ten: Submit Your Patch Editor!

- Clean up the editor code, make it really polished, well documented, and good looking.
- Test it well.
- Copyright your editor code at the top of the file. License the editor code under Apache 2.0 (I don't accept anything else).
- Send the whole directory to me! I'd love to include it.

[20] Note that lots of synthesizers have sysex facilities to dump the entire patch memory, or dump an entire bank, etc. Edisyn doesn't use these; it requests patches one by one. This is slower but saves you a lot of coding and is consistent across synthesizers. So you're welcome.