Edisyn

A Java-based Synthesizer Patch Editor
Version 10
By Sean Luke
sean@cs.gmu.edu

Edisyn is a no-nonsense synthesizer patch editor designed to be extensible for many different kind of of synthesizers, and to have unusual programming assistance capabilities. It is not skewmorphic and not skinnable: it's design is very no-nonsense and consistent. Edisyn is free open source.

Edisyn at present has patch editors for the following synthesizers, which strangely enough are the very ones that I own!

- Kawai K4 and K4r (Single and Multimode)
- Waldorf Blofeld Desktop, Blofeld Desktop SL, and Blofeld Keyboard (Single and Multimode)
- Waldorf Microwave II, Microwave XT, and Microwave XTk (Single and Multimode)
- Yamaha TX81Z (Single and Multimode)
- Oberheim Matrix 1000 (Single)
- Preen FM2 (Single)

Edisyn does not support patches for global parameters, nor for wave, sample, or wavetable editing etc. Additionally Edisyn is at present **not a librarian tool**. You should use a good free librarian tool, such as SysEx Librarian on the Mac.

1 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.

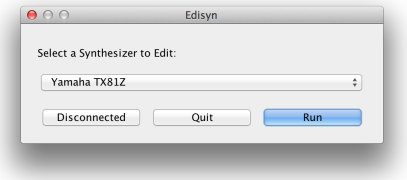

Figure 1: Initial Synthesizer Dialog

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.

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):

- 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.
- The USB MIDI Device to which you will **Send** MIDI to ultimately be sent to the synthesizer. Here we are sending to a Tascam US-2x2 interface, which presents itself as a generic, nameless device.

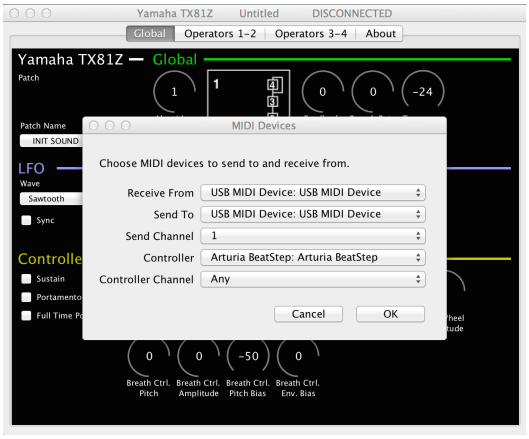

Figure 2: MIDI Dialog

- 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 At present, due to bugs in the MIDI library Edisyn relies on, if you connect a device to your computer *after* you have fired up Edisyn, Edisyn will not be able to see it. You'll need to restart Edisyn if you want to use that device. So connect your devices first.

2 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.

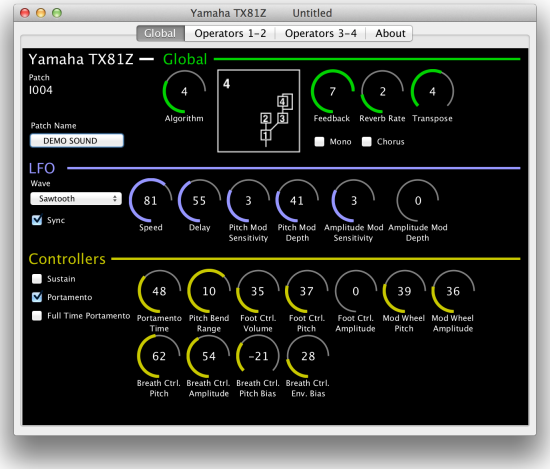

Figure 3: Typical Patch Editor Panel (TX81Z)

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 various random 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.

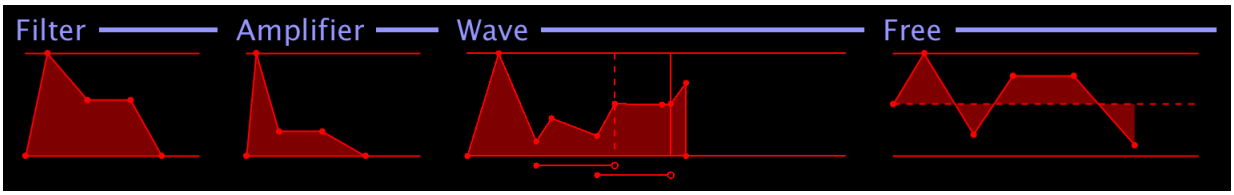

Figure 4: Envelope Displays of the Waldorf Microwave II, XT, and XTk.

- 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 of 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 4 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.

  At present Edisyn can only draw envelopes: you can't drag the dots.

- **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).

3 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 1, or to run in Disconnected Mode.

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

3.1 Persistence and Preferences

Now would be a good time to mention an Edisyn feature you may never notice: 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 menu choices are persistent as well.

4 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.

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.

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

Finally, 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**.

6 Communicating with the Controller

The MIDI Dialog (Section 1) 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

6.1 Remote Controlling your Synthesizer

If you play a note, do a pitch bend, etc., on your control surface, 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). You can also pass through Program Change messages, MIDI clock, etc. Control Change (CC) messages from your control surface are passed through only if you have toggled **Pass Through All CCs** in the **Map** menu.

6.2 Remote Controlling Edisyn

Edisyn is capable of *mapping* Control Change (CC) messages or NRPN messages from your control surface to specific parameters in your patch editor. Each patch editor type can learn its own unique 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. The title bar might also tell you what the *previous* mapping was.
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. Note that 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.

- **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.
- **Relative CC "64"** Here, the CC value you send indicates how much to *add to* or *subtract from* the existing parameter value. In this form of Relative CC, 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. This style is supported by a number of controllers and is useful for encoders. To map, choose **Map Relative CC[64]** in the **Map** menu.
- **Relative CC "0"** Here, the CC value you send again indicates how much to *add to* or *subtract from* the existing parameter value. In this form of Relative CC, 0 means 0 (add nothing), a high value $64 < x < 128$ means to subtract $128 - x$ from the current value, and a low value $0 > x \geq 64$ means to add x to the current value. This style is also supported by a number of controllers and is useful for encoders. To map, choose **Map Relative CC[0]** in the **Map** menu.
- **NRPN** You are permitted to map any NRPN parameter at all. 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 rare. To map, choose **Map CC/NRPN** in the **Map** menu.

Mapping by Panel or by MIDI Channel In Edisyn, 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. CCs on MIDI channels which do not match Edisyn's controller channel are passed through to the synthesizer.

Alternatively you can turn off this behavior and instead send all CCs to Edisyn regardless of MIDI Channel: now two panels can both use CC#1 only if it's on different MIDI channels. This is a common mapping approach used in DAWs. For example, CC#1 on MIDI Channel 1 might change the Start Wave parameter , while CC#1 on MIDI Channel 2 changes the Detune parameter (in the same tab!) and CC#1 on MIDI Channel 3 changes the Envelope 1 Attack (in the Envelope panel).

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.

You turn on per-MIDI-channel CCs by toggling **Do Per-Channel CCs** in the **Map** menu (and conversely choose per-Panel CCs by toggling it off).

7 Creative Programming

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.

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

Now when you chose any of **Towards 1:...** through **Towards 4:...**, your current patch will get **recombined** with the target patch, currently by 50%, to move it towards that target.

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.

7.1 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 the back).[2] 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 5 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**.

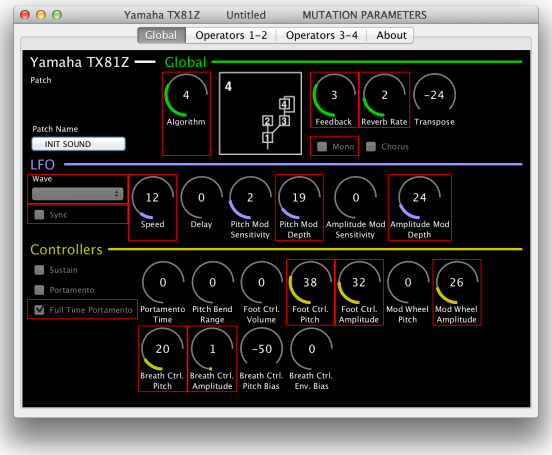

Figure 5: Editing Mutation Parameters

[2]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 5. But you can click on its title (the text "Wave").

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.

7.2 How Recombination and Mutation Work

One unusual feature of Edisyn is its support for various ways to recombine (merge) and mutate (partially randomize) patches. These are used in a variety of tools designed to help you hunt for new and unusual sounds. Many systems let you randomize sounds, but Edisyn goes a lot further. Just so you know, here are the recombination and mutation procedures:

Recombination When your current patch **A** is recombined with another patch **B**, you produce a patch which is somewhat in-between the two. You will specify the degree to which this should be done as a percentage 0...100%. This number specifies what percentage of parameters in **A** may (randomly) get merged with their compatriots in **B**. Different kinds of parameters merge in different ways.

- A *metric parameter*, such as a number range from 0...127, will merge with another by picking a random number somewhere between the two. Many dials are metric parameters, for example.
- A *non-metric parameter* is just a set of arbitrary values, such as the Filter types HP, LP, BP, and Bit Crush. A non-metric parameter will merge with another by choosing one or the other of them with a 50/50 chance. Choosers and checkboxes are generally non-metric parameters.
- Some parameters have both metric and non-metric regions: for example, a MIDI Channel dial might have the values 1...16, plus *off* and *omni*. Here the values 1...16 are metric, and off and omni are non-metric. If both **A** and **B** have parameter values which are in the metric region, then they are merged in the metric way. If they're both non-metric, or one is metric and the other is not, then one or the other value is selected.
- Occasional parameters are *immutable*: they will refuse to be merged. These are typically things like patch names.

0% recombination means nothing changes in patch **A**. 100% recombination means that every parameter is recombined: but this doesn't imply that patch **A** is entirely replaced by patch **B**: just that the two are fully merged together.

Mutation When a patch **A** is *mutated*, it is modified with some degree of random noise. Again, you specify a percentage 0...100%, but this means something different than in recombination. Every single parameter will potentially get mutated, but percentage specifies the *degree* to which noise will impact on a parameter. Specifically:

- If your parameter is *metric*, then the percentage specifies the width of a uniform distribution centered at the current value. For example, if your parameter range is 0...127 (and so has size 128), and your current value is 30, and your percentage is 25%, then we will select a new random value within the region $[30 - 128 \times 0.25, 30 + 128 \times 0.25]$, which reduces to $[-2, 62]$. Obviously, -1 and -2 are not valid, so if we choose them, we try again repeatedly until we get a valid number.

0% mutation means none at all. 100% mutation is guaranteed to be complete randomization over the entire range. But other values may be counterintuitive: for example, 25% sounds small but it's not: it's potentially selecting numbers over half of the range. Generally if you intuitively want to do a mutation of D%, cut your D down by half.

- If your parameter is *non-metric*, the percentage simply specifies the probability that it will be entirely randomized to a new value.
- If your parameter is has both metric and non-metric regions, then we first decide whether to jump the parameter value from metric to non-metric (or vice versa). This is half of the percentage. So if your percentage is 25%, then this happens with a $0.25 \times 0.5 = 0.125$ likelihood. If the value jumps then we pick a totally new random value for it (in its new metric or non-metric region). If the value doesn't jump, then it is mutated just like elements in its region (metric or non-metric) as above.
- Occasional parameters are *immutable*: they will refuse to be mutated. These are typically things like patch names.

8 Synthesizers are Inconsistent

The dirty truth is that synthesizer companies don't really care about MIDI control, else they'd put more effort into it. The Waldorf Blofeld's multimode is entirely 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 the 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). 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. And of course the infamous Korg Microsampler is such a MIDI disaster that Edisyn may never release its patch editor.

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

	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	✓	✓	✓	✓[3]	✓	✓	✓	✓		✓
Waldorf Blofeld	✓[4]	✓[4]	✓	✓[3]	✓	✓	✓			✓
Kawai K4/K4r	✓[5]		✓		✓	✓	✓		✓	✓
Yamaha TX81Z	✓[4]			✓	✓					✓[6]
PreenFM2	✓	✓	✓	✓	✓	✓				
Oberheim Matrix 1000	✓	✓	✓		✓	✓	✓			✓
Korg Microsampler	✓[7]	✓[7]								

9 Writing a Patch Editor

So you want to write a patch editor? They're not easy. But they're fun! Here are some hints. Let's say you're adding a single (non-multimode) patch editor for the Yamaha DX7. Here's what you'd do.

9.1 Step One: Create Files

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"; }
```

9.2 Step Two: Get the UI Working

This is mostly the 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.

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.

[3]Incoming patch won't have patch number or bank information.
[4]Not in Multimode.
[5]Not in Multimode. Also, certain parameters cannot be set (source mutes).
[6]File contains two concatenated sysex transmissions.
[7]A limited number of them only.

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 should always 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 general: your internal parameters should have contiguous ranges and make sense from a user perspective and not the synthesizer's weird worldview.

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 m̂ax 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.[8] 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.

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")),
```

[8] 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.

```
    ... // 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 a single pixel is 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 with it! 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[9]), or just play "C 0" in simple MIDI notation. You can specify this by calling the method **setOctavesBelowZero(...)**.

- **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"):

  ```
  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.[10]

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

[10] *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.

```
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.

  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 4.

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 4.

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 4.

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 Waldorf Blofeld's code for an ADSR envelope. Before we go on, goo to the Blofeld patch editor and play with the Attack, Decay, Sustain, and Release settings. Notice that 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*.[11] 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 };
```

[11] Actually, the envelope as displayed in the Blofeld synth itself is drawn as if the Blofeld uses length. But it does not: it's an error. Edisyn's patch editor is right and Waldorf's UI designers are wrong!

```
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, rates);
```

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, rates)
    {
    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).

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.

9.3 Step Three: 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.[12]

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.[13] So if you do something fancy with emit(...) later, you may need to revise your parse(...) implementation.

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(...).

[12] 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.

[13] 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.

- **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()** should extract the patch name from the model (probably via `getModel().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** 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(...)** method 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.

9.4 Step Four: 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()** or bad things will 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.

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 If your bulk dumps are emitted by you as sysex messages, then again congratulations, you already have file writing working. If not, as discussed earlier, you will need to *invent* a bulk sysex format and implement it in the **emit(...)** and **parse(...)** methods. That way you can still load and save files.

9.5 Step Five: 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();
```

9.6 Step Six: Other Niceties

You're almost done! Two other niceties you might want to do. First, 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.

9.7 Step 7: 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.