Overriding core limitations : how to handle it the right way

A bit of theory

When talking about managing an open-source technology integration, there are a ton of debates about “to do” or “no to do” adaptations to the code. Many of the exposed reasons often fall into a dogmatic discourse, the originating motivation is understandable : the lack of architectural mastering that set many of the moodle administrators and technical crew in a “black box” situation, i.e., having between hands an “opaque thing” we do not really know how it works. But if we respect the given rules, it should work.

All is about trouble of the technical crew assigned to making moodle work.

Directly opposed to this “prudent” way of managing the platform lifecycle, is usage and requirement for doing “better”, “clearer”, more “user friendly” user experience. Often this view stucks on core limitations, i.e., unforecasted consequences of code design, unexpected or not imagined uses cases, etc. Moodle has a LOT of tuning and overriding possibilities and techniques, thus those cases of hanging on a “really tricky override case” are few.

The purpose of this chapter is to demonstrate that locking on a Moodle Intrinsic limitation is not a pure fatality. All depends on a fair evaluation of the balance between Maintenance Trouble amount vs. Usage Benefice. Most of the reluctance factors to accept core overrides are measured by the “factor of Trouble occasionned for Moodle tech mainteners crew'.

We propose further some theorical approach of the concern:

First will we propose a theorical definition of the “Technical Impact” of a design decision. The **technical impact** (Imp) roughly determines how amount of effort will be necessary to maintain properly a working application:

$$Imp = \frac{Nop * Xop * Of}{Aut}$$

The impact unit should be not far from some Energy measurement.

It can be modelized roughly as a function of increasing factors: Number of maintenance operations - Nop, complexitxy of operations - Xop, and probably an organisational factor that increases when both number and complexity grows. Organisational factor can include f.e. an increasing of the number of actors in the operations and the loss in coordination.

Opositely, there is one big factor that helps reducing the technical impact : Automation (Aut). The more automated you are (including also power tools, not necessary fully automated), the less the technical impact will be a trouble.

Now we can address the notion of “Trouble”. “Trouble” is a subjective notion, and wil not affect all actors the same way. Maintainers will have “Trouble” in having to maintain complex things and make it available for users. Users will be troubled in not having friendly enough or unefficiant tools,

stakeholders will be troubled to get informed and harassed that thing do NOT WORK well.

Now let propose a structural definition of the Trouble:

$$Tb = k * \frac{Imp}{\sum_{r=1}^n{dCUi * NUi}}$$

Let's explain now…

The trouble raises when the Technical Impact raises. We just add a k constant factor to adjust units and scales.

Oppositely, the trouble decreases when the “gain in process” (dCU) per user in a certain role and the role population (NU, as number of users) increases. In other words, the more users can gain in their own process (gain of time, gain of mental load), the more affordable the Technical Impact will be for the organisation.

Customcripts

Customscripts are a “soft” way to get over the moodle core limitations. Customscripting is a standard mechanism to get core code replaced by a customized version, WITHOUT interfering with core code from the official Git or distribution.

By defining a directory (often `/customscripts`) at moodle code's root (or outside) and adding a key:

```
$CFG->customscripts = '<moodlepath>/customscripts';
```

In your config file, then any request to a php script say: `https://<mymoodle>/a/b/c.php` that encounters an existing file named `/customscripts/a/b/c.php` will be derouted at end of the `setup.php` initialisation to that file, includes the file, then continue normal processing.

Customscripts MUST NOT include `config.php`, because it has already be done before routing to customscript.

By ending the customscript by a `die()` or `exit()`, the customscripted script REPLACES the standard behaviour.

Customscripts MUST NOT redefine classes, functions or any “compilation time definitions”, as the standard version has already be loaded into memory.

Good cases

Customscripts work fine when the change is very superficial, and resides in the moodle page itself (vs. a deeper library).

Less good cases

When the change is deeper, customscripting will force you to deroute deeper calls to derouted APIs,

so the code path may need a lot of work to reach the changing point.

Too complicated derouting will lead to replacing a LOT of standard code. As the standard code is occulted in your Moodle, it will not reveal upgrades or bugfixes, so your replaced code will diverge slowly from moodle standard state of the art. this can be problematic.

Core patchs

Core patchs are the last extremity of overriding core code. All moodle developers are conscious that patching **IS BAD**, but sometime are the only reasonable solution we have to make things work as they *should work*.

Core patches rais a real issu of middle term and long term maintenance, because upgrading a moodle instance results in a non trivial operation:

- **Trivial way (no patchs):** Get the moodle Zip archive, unpack it, copy all other third party plugins in it. check third party compatibility level, and upgrade, and that's it.
- **Non trivial way (some patchs):** Do the some way than trivial **PLUS** report all patchs from the previous code base to the new codebase.

and THIS can be the tough part, specially if:

- Patchs are **not documented**
- Patchs are **not properly tagged** and identified in the code
- There is **no tool** to help the process

Tracking patches and customscriptings needs in plugin design

In our plugins, a common structure to self-document and provide out-of-the-band info about moodle alteration we need to get the plugin run correctly is given as '__' prefixed directories in the plugin.

- **__patch:** contains core files with patches serving this plugin.
- **__customscripts:** contains files to add to the moodle local 'customscript' folder, as defined in `config.php` with the key `$CFG→customscripts`.
- **__reference:** contains core files who serve as core source reference for the patchs, and are snapshot when the plugin is reviewed and packaged. Your installed core files may be somewhat different, if your minor version is not the exact one we use in our development faciity.

What happens if several plugins need patching/customscripting the same files

This is a more tricky situation that can happen, as we aim to keep the modularity concept of Moodle as clean as possible, which means, many plugin do not have to car another plugin exist, moreover if there is no functional relation between them.

As a consequence, a general packing rule has been to only pack whate was relative to the current plugin (or pluginset in case of real unbreakabke bundles).

So it may be possible that assembling several unrelated plugins might need to overload the same core files.

In case of patchs, all patchs are bounded strictly in `// PATCH+` and `// PATCH-` comments, so they are easy to locate in the code. In the future, we will have to make some effort in generalizing the @see tag so each patch refers to the plugin that required it.

In case of customscripts, the case of collision is also possible. In most case we try to use `// CHANGE +` and `// CHANGE-` comments to locate them.

Back to general index