DPM Modular Structure

`DPM` is, by the necessity of when and where it is used, modular.

This greatly reduces the dependency requirements at various stages of bootstrapping and allows for plenty more use cases than if all requirements and functionality were jammed into one single binary.

This also allows each piece of DPM to be a dedicated purpose component, allowing features to be more appropriately focused on.

Using a modular design in a C-based implementation essentially means that the dpm binary itself's primary concern is as a command and argument routing engine to shared objects with a common interface specification using reserved symbol names to determine entry point.

The modules themselves perform the work that DPM does. This provides the ability for 3rd party extension of DPM's capabilities without modification of the DPM source code, and allows different functionality of DPM to be managed by packages that manage the modules which perform that work.

DPM Core

The core DPM component is concerned primarily with command routing and module loading.

A sample implementation would strongly resemble this:

```
dpm/                          # Core DPM router/loader project
  ├── CMakeLists.txt
  ├── include/
  │   └── dpm/
  │       ├── module.h        # Module interface definitions
  │       └── types.h         # Common types/structures
  ├── src/
  │   ├── main.c             # dpm command entry point
  │   ├── module.c           # Module loading/management
  │   └── router.c           # Command routing logic
  └── tests/
```

```
    └── module_tests/    # Tests using dummy test modules
```

DPM Modules

The modules are the things that do what we are talking about when we are performing work with DPM. As you've no doubt noticed there are some commands in other parts of this documentation that refer to `dpm_create-repo` and `dpm_build-package` or similar. In future versions this will be a `repo` module and a `package` module, so that the command, instead of a separate script, is `dpm repo create` with various arguments and `dpm package build` with likewise arguments, such that users do not have to track the presence of various tools — it is simply a subcommand structure passed to the dpm tool which corresponds to the modules it has installed to determine which capabilities it has.

The Dark Horse Linux Project will provide a common set of modules for basic package management and repository interaction capabilities. For now these will be under one project structure to develop them in parallel, but these will eventually be separate repositories and projects. This allows the focus introduced by the separation of concerns to benefit the functionality provided. A sample core DPM modules project structure would resemble:

```
dpm-modules/              # Official modules repository
  ├── package/            # Package management module
  │   ├── CMakeLists.txt
  │   ├── src/
  │   └── tests/
  │
  ├── signature/          # Optional signing module
  │   ├── CMakeLists.txt
  │   ├── src/
  │   └── tests/
  │
  └── repo/              # Repository management module
      ├── CMakeLists.txt
      ├── src/
      └── tests/
```

This reduces the initial dependency requirement from, for example, the "gpgme" dependency tree, which is rather heavy, from being required at dpm install time

to only requiring libstdc++ and then the dependencies for any modules that are needed as the need is introduced, allowing you to start using the package manager to install needed functionality instead of manual compilation and installation, much earlier.

This also additionally allows anyone who can, to create modules that replace functionality of what is provided by the Dark Horse Linux Project in case some people can do it better and realize that they should — without needing to be making changes to the core project.

Command Syntax

It should look something like this:

```
dpm [<dpm args>] <module-name> [<module args>] <command> [<module command args>]
```