## Package Oriented Design

Package Oriented Design allows a developer to identify where a package belongs inside a Go project and the design guidelines the package must respect. It defines what a Go project is and how a Go project is structured. Finally, it improves communication between team members and promotes clean package design and project architecture that is discussable.

## Links

[Design Philosophy On Packaging](https://www.ardanlabs.com/blog/2017/02/design-philosophy-on-packaging.html)    
[Package Oriented Design](https://www.ardanlabs.com/blog/2017/02/package-oriented-design.html)

## History

In an interview given to Brian Kernighan by Mihai Budiu in the year 2000, Brian was asked the following question:

**“Can you tell us about the worse features of C, from your point of view”?**

This was Brian’s response:

**“I think that the real problem with C is that it doesn’t give you enough mechanisms for structuring really big programs, for creating "firewalls" within programs so you can keep the various pieces apart. It’s not that you can’t do all of these things, that you can’t simulate object-oriented programming or other methodology you want in C. You can simulate it, but the compiler, the language itself isn’t giving you any help.”**

## Language Mechanics

* Packaging directly conflicts with how we have been taught to organize source code in other languages.
* In other languages, packaging is a feature that you can choose to use or ignore.
* You can think of packaging as applying the idea of microservices on a source tree.
* All packages are "first class," and the only hierarchy is what you define in the source tree for your project.
* There needs to be a way to “open” parts of the package to the outside world.
* Two packages can’t cross-import each other. Imports are a one way street.

## Design Philosophy

* **To be purposeful, packages must provide, not contain.**
  + Packages must be named with the intent to describe what it provides.
  + Packages must not become a dumping ground of disparate concerns.
* **To be usable, packages must be designed with the user as their focus.**
  + Packages must be intuitive and simple to use.
  + Packages must respect their impact on resources and performance.
  + Packages must protect the user’s application from cascading changes.
  + Packages must prevent the need for type assertions to the concrete.
  + Packages must reduce, minimize and simplify its code base.
* **To be portable, packages must be designed with reusability in mind.**
  + Packages must aspire for the highest level of portability.
  + Packages must reduce setting policy when it’s reasonable and practical.
  + Packages must not become a single point of dependency.

## Project Structure

Kit Application

├── CONTRIBUTORS ├── cmd/

├── LICENSE ├── internal/

├── README.md │ └── platform/

├── cfg/ └── vendor/

├── examples/

├── log/

├── pool/

├── tcp/

├── timezone/

├── udp/

└── web/

* **vendor/**  
  Good documentation for the vendor/ folder can be found in this Gopher Academy [post](https://blog.gopheracademy.com/advent-2015/vendor-folder) by Daniel Theophanes. For the purpose of this post, all the source code for 3rd party packages need to be vendored (or copied) into the vendor/ folder. This includes packages that will be used from the company Kit project. Consider packages from the Kit project as 3rd party packages.
* **cmd/**  
  All the programs this project owns belongs inside the cmd/ folder. The folders under cmd/ are always named for each program that will be built. Use the letter d at the end of a program folder to denote it as a daemon. Each folder has a matching source code file that contains the main package.
* **internal/**  
  Packages that need to be imported by multiple programs within the project belong inside the internal/ folder. One benefit of using the name internal/ is that the project gets an extra level of protection from the compiler. No package outside of this project can import packages from inside of internal/. These packages are therefore internal to this project only.
* **internal/platform/**  
  Packages that are foundational but specific to the project belong in the internal/platform/ folder. These would be packages that provide support for things like databases, authentication or even marshaling.

## Validation

**Validate the location of a package.**

* Kit
  + Packages that provide foundational support for the different Application projects that exist.
  + logging, configuration or web functionality.
* cmd/
  + Packages that provide support for a specific program that is being built.
  + startup, shutdown and configuration.
* internal/
  + Packages that provide support for the different programs the project owns.
  + CRUD, services or business logic.
* internal/platform/
  + Packages that provide internal foundational support for the project..
  + database, authentication or marshaling.

**Validate the dependency choices.**

* All
  + Validate the cost/benefit of each dependency.
  + Question imports for the sake of sharing existing types.
  + Question imports to others packages at the same level.
  + If a package wants to import another package at the same level:
    - Question the current design choices of these packages.
    - If reasonable, move the package inside the source tree for the package that wants to import it.
    - Use the source tree to show the dependency relationships.
* internal/
  + Packages from these locations CAN’T be imported:
    - cmd/
* internal/platform/
  + Packages from these locations CAN’T be imported:
    - cmd/
    - internal/

**Validate the policies being imposed.**

* Kit, internal/platform/
  + NOT allowed to set policy about any application concerns.
  + NOT allowed to log, but access to trace information must be decoupled.
  + Configuration and runtime changes must be decoupled.
  + Retrieving metric and telemetry values must be decoupled.
* cmd/, internal/
  + Allowed to set policy about any application concerns.
  + Allowed to log and handle configuration natively.

**Validate how data is accepted/returned.**

* All
  + Validate the consistent use of value/pointer semantics for a given type.
  + When using an interface type to accept a value, the focus must be on the behavior that is required and not the value itself.
  + If behavior is not required, use a concrete type.
  + When reasonable, use an existing type before declaring a new one.
  + Question types from dependencies that leak into the exported API.
    - An existing type may no longer be reasonable to use.

**Validate how errors are handled.**

* All
  + Handling an error means:
    - The error has been logged.
    - The application is back to 100% integrity.
    - The current error is not reported any longer.
* Kit
  + NOT allowed to panic an application.
  + NOT allowed to wrap errors.
  + Return only root cause error values.
* cmd/
  + Allowed to panic an application.
  + Wrap errors with context if not being handled.
  + Majority of handling errors happen here.
* internal/
  + NOT allowed to panic an application.
  + Wrap errors with context if not being handled.
  + Minority of handling errors happen here.
* internal/platform/
  + NOT allowed to panic an application.
  + NOT allowed to wrap errors.
  + Return only root cause error values.

**Validate testing.**

* cmd/
  + Allowed to use 3rd party testing packages.
  + Can have a test folder for tests.
  + Focus more on integration than unit testing.
* kit/, internal/, internal/platform/
  + Stick to the testing package in go.
  + Test files belong inside the package.
  + Focus more on unit than integration testing.

**Validate recovering panics.**

* cmd/
  + Can recover any panic.
  + Only if system can be returned to 100% integrity.
* kit/, internal/, internal/platform/
  + Can not recover from panics unless:
    - Goroutine is owned by the package.
    - Can provide an event to the app about the panic.