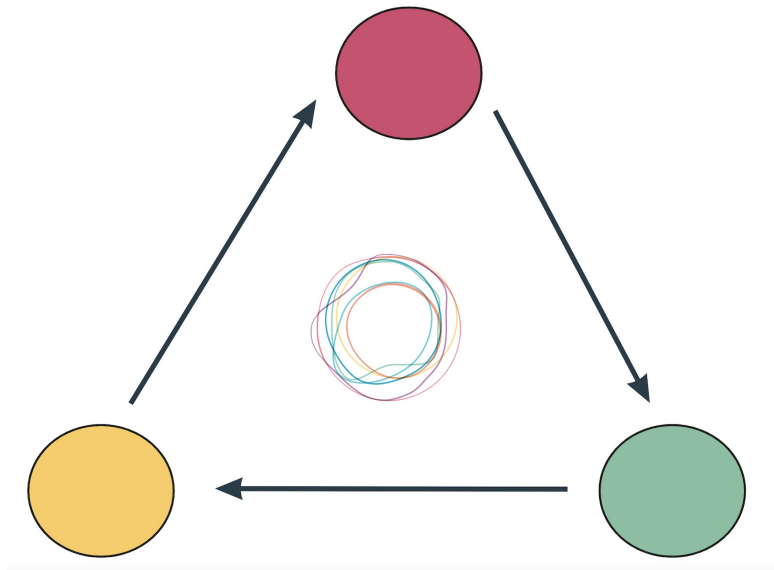


TASK LIST

–Meet-up Kata–

TDD | Software-Design | Refactoring | SOLID Principles | Starter

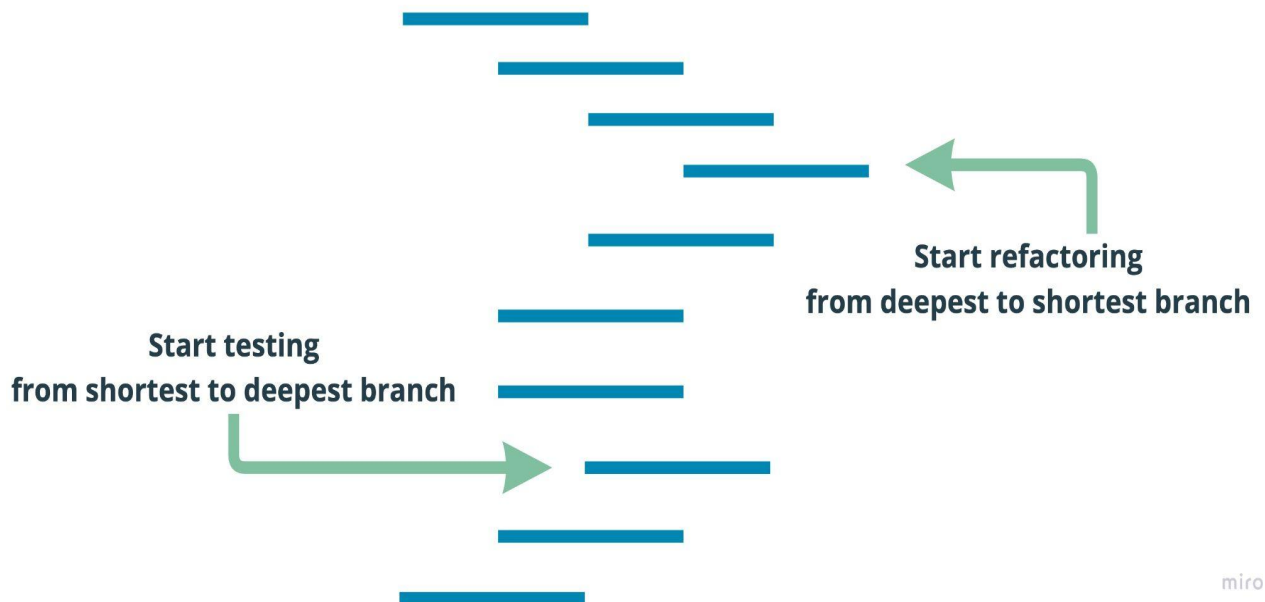


PURPOSE

This is an example of code obsessed with primitives.

A primitive is any concept technical in nature, and not relevant to your business domain. This includes integers, characters, strings, and collections (lists, sets, maps, etc.), but also things like threads, readers, writers, parsers, exceptions, and anything else purely focused on technical concerns. By contrast, the business concepts in this project, “task”, “project”, etc. should be considered part of your domain model. The domain model is the language of the business in which you operate, and using it in your code base helps you avoid speaking different languages, helping you to avoid misunderstandings. In our experience, misunderstandings are the biggest cause of bugs.

Tip

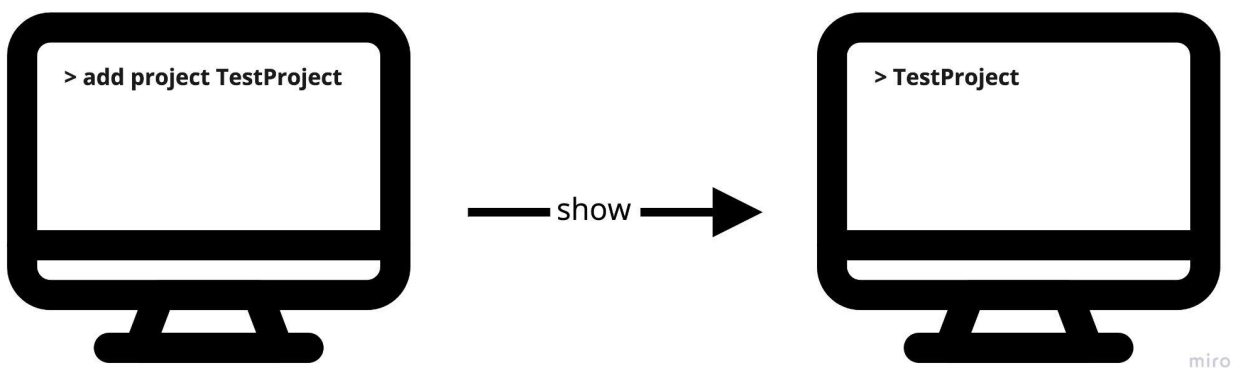


miro

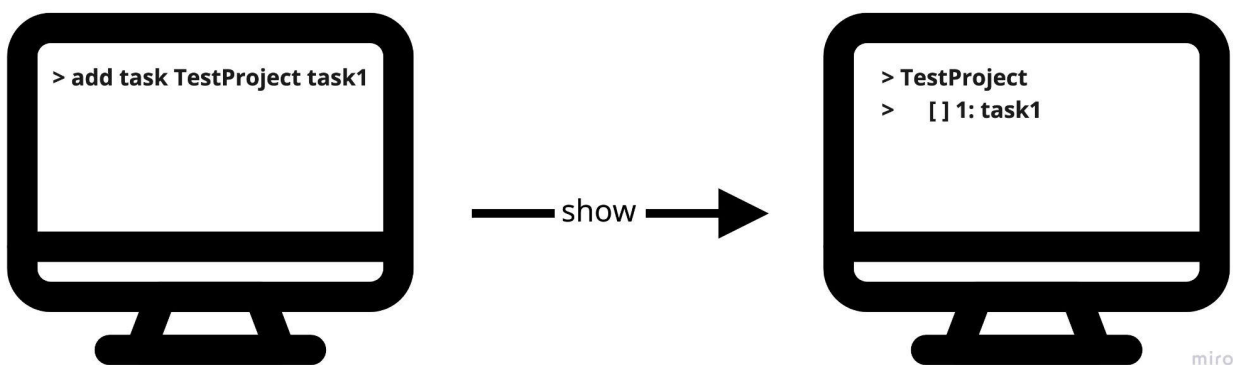
FUNCTIONALITY

This console application manages tasks. For this purpose, you can execute a list of commands, such as show, add <"project" project name>, add <"task" "project" task name>, check <task number>, uncheck <task number>, help. The following are examples of the execution of these commands:

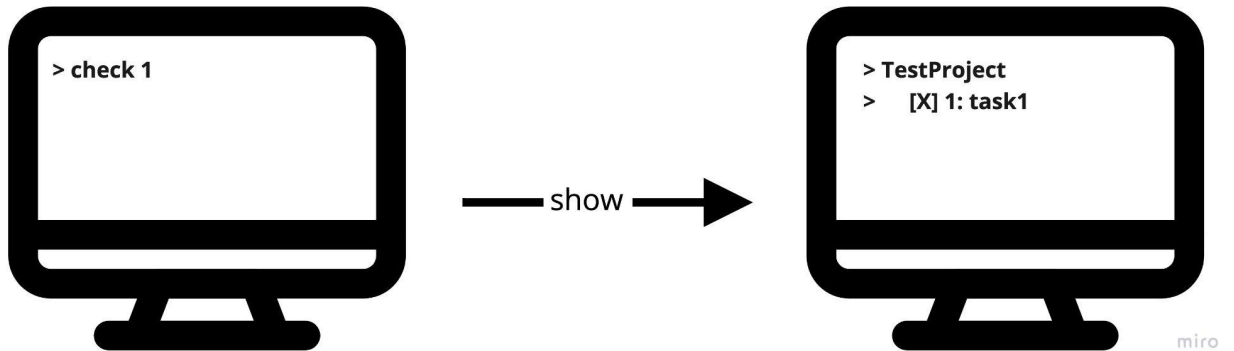
add <"task" "project" task name>



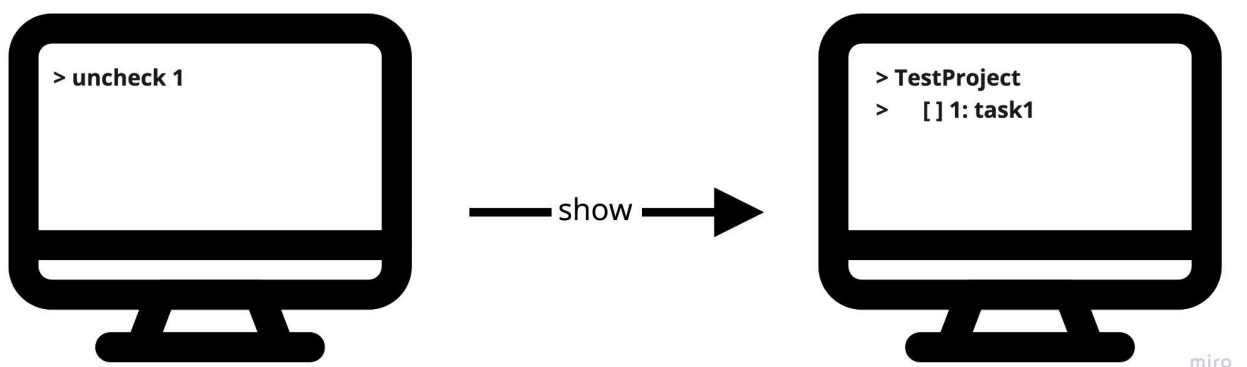
add <"task" "project" task name>



add <"task" "project" task name>



add <"task" "project" task name>



help: shows all commands

show: display the complete task list: projects, task and task status.



EXERCISE

Try implementing the following features, refactoring primitives away as you go. Try not to implement any new behaviour until the code you're about to change has been completely refactored to remove primitives, i.e. Only refactor the code you're about to change, then make your change. Don't refactor unrelated code.

One set of criteria to identify when primitives have been removed is to only allow primitives in constructor parameter lists, and as local variables and private fields. They shouldn't be passed into methods or returned from methods. The only exception is true infrastructure code—code that communicates with the terminal, the network, the database, etc.

Infrastructure requires serialisation to primitives, but should be treated as a special case. You could even consider your infrastructure as a separate domain, technical in nature, in which primitives are the domain.

You should try to wrap tests around the behaviour you're refactoring. At the beginning, these will mostly be high-level system tests, but you should find yourself writing more unit tests as you proceed.

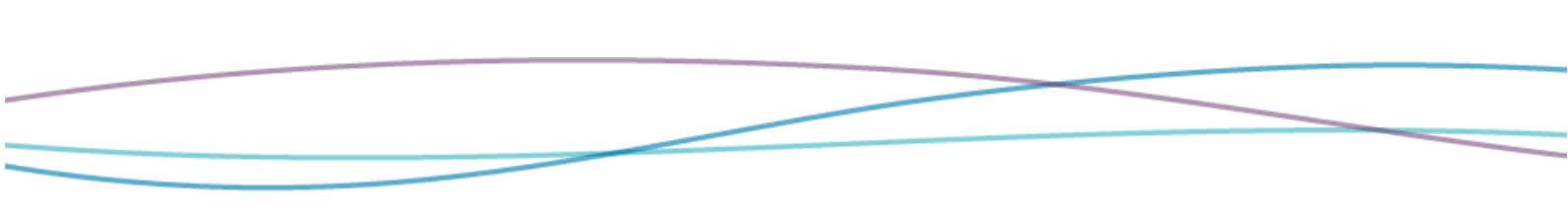
FEATURE

1. Deadlines

1. Give each task an optional deadline with the `deadline <ID> <date>` command.
2. Show all tasks due today with the `today` command.

2. Customisable IDs

1. Allow the user to specify an identifier that's not a number.

- 
2. Disallow spaces and special characters from the ID.
 3. Deletion
 1. Allow users to delete tasks with the `delete <ID>` command.
 4. Views
 1. View tasks by date with the `view by date` command.
 2. View tasks by deadline with the `view by deadline` command.
 3. Don't remove the functionality that allows users to view tasks by project, but change the command to `view by project`.

CONSIDERATIONS AND APPROACHES

Think about *behaviour attraction*. Quite often, you can reduce the amount of behaviour that relies upon primitives from the outside world (as opposed to internal primitives stored as private fields or locals) simply by moving the behaviour to a *value object* which holds the primitives. If you don't have a value object, create one. These value objects are known as *behaviour attractors* because once they're created, they make it far more obvious where behaviour should live.

A related principle is to consider the type of object you've created. Is it a true value object (or *record*), which simply consists of `getFoo` methods that return their internal primitives (to be used only with infrastructure, of course), or is it an object with behaviour? If it's the latter, you should avoid exposing any internal state at all. The former should not contain any



behaviour. Treating something as both a record and an object generally leads to disaster.

Your approach will depend on whether you learn toward a functional or an object-oriented style for modelling your domain. Both encourage encapsulation, but *information hiding* techniques are generally only used in object-oriented code. They also differ in the approach used to extract behaviour; functional programming often works with closed sets of behaviour through *tagged unions*, whereas in object-oriented code, we use *polymorphism* to achieve the same ends in an open, extensible manner.

Separate your commands and queries. Tell an object to do something, or ask it about something, but don't do both.

Lastly, consider SOLID principles when refactoring:

- Aim to break large chunks of behaviour into small ones, each with a single responsibility.
- Think about the dimensions in which it should be easy to extend the application.
- Don't surprise your callers. Conform to the interface.
- Segregate behaviour based upon the needs.
- Depend upon abstractions.

this Kata has been extracted from [Kata-logs.rock](https://kata-logs.rock)