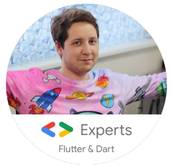
Google Developers

Design Patterns toolbox:
(not so) obvious patterns for Flutter



Mangirdas Kazlauskas
GDE for Flutter & Dart
@mkobuolys

Agenda

- OOP Design Patterns 101
- Example App
- Abstract Factory
- Composite
- Command
- Memento

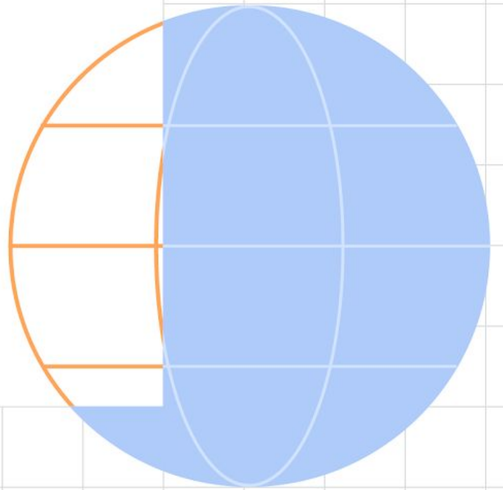
About me

- Software Engineer from Lithuania 🇱🇹
- Mobile Tech Lead @ Billo
- Google Developer Expert for Flutter & Dart 💙
- Organiser @ Flutter Lithuania
- Using Flutter since v0.10.2

billo

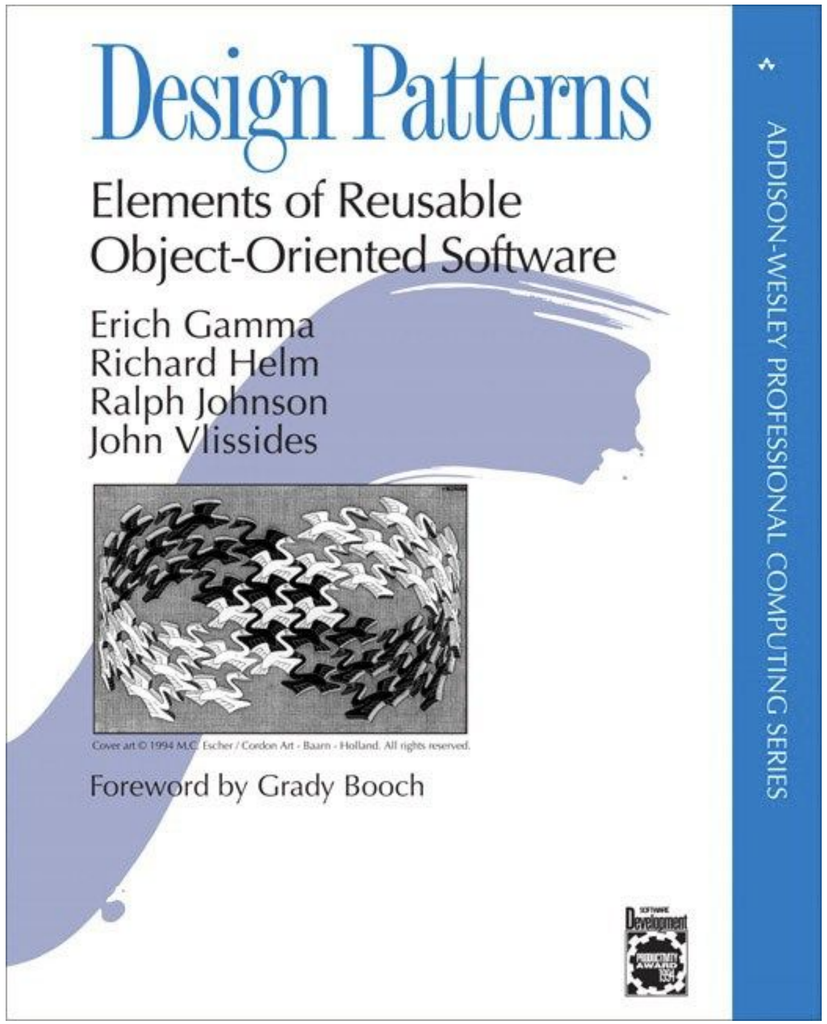
Experts

OOP Design Patterns 101



OOP Design Patterns 101

- Solves common code design problems
- (Only) provides a general idea, structure/blueprint on how to deal with a particular problem
- Speeds up the development process
- Improves code flexibility and reusability

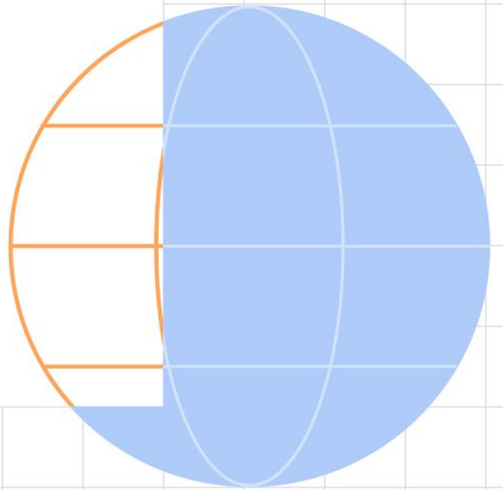


Example App: Patternify

- Music playlist management app
- State management - flutter_bloc
- Dependency Injection – provider

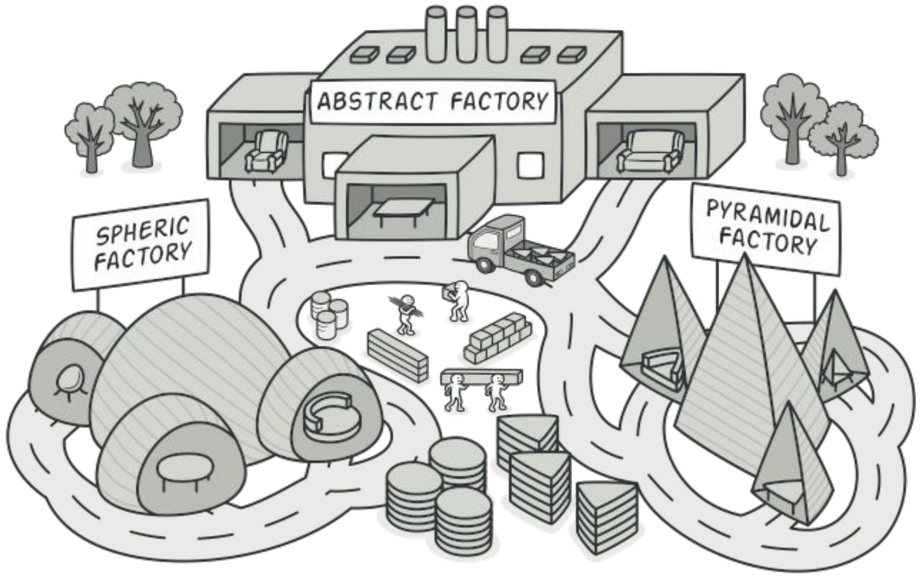
Experts

Abstract Factory



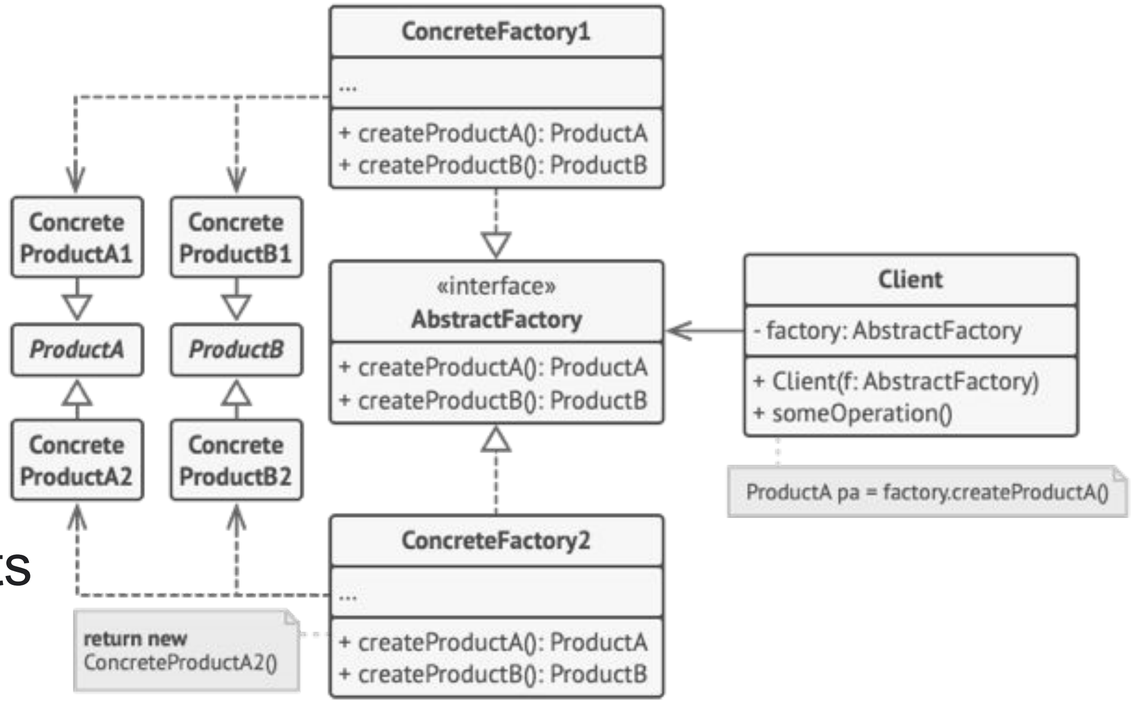
“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”

- GoF



Abstract Factory

- Creational design pattern
- *Abstract Factory* - declares an interface of operations that create abstract *Product* objects
- *Concrete Factory* – implements the operations to create *Concrete Product* objects
- *Product* – interface for a type of *Product* object
- *Concrete Product* – defines a product object to be created by corresponding *Concrete Factory*



Source: https://refactoring.guru/images/patterns/diagrams/abstract-factory/structure.png

Abstract Factory: abstract factory

```dart
abstract class PlatformWidgetsFactory {
  PreferredSizeWidget createAppBar({
    required String title,
    bool showSettingsButton = true,
  });
  Widget createBottomNavigationBar({
    required int currentIndex,
    required ValueSetter<int> onTap,
  });
  Widget createLoader();
  PageRoute createPageRouter({
    required WidgetBuilder builder,
  });
  Widget createSwitcher({
    required bool isActive,
    required ValueSetter<bool> onChanged,
  });
}
```

Abstract Factory: concrete factories

```dart
class MaterialWidgetsFactory implements PlatformWidgetsFactory {
  const MaterialWidgetsFactory();

  @override
  PreferredSizeWidget createAppBar({
    required String title,
    bool showSettingsButton = true,
  }) =>
      MaterialAppBar(title: title, showSettingsButton: showSettingsButton);

  @override
  Widget createBottomNavigationBar({
    required int currentIndex,
    required ValueSetter<int> onTap,
  }) =>
      MaterialBottomNavigationBar(currentIndex: currentIndex, onTap: onTap);

  @override
  Widget createLoader() => const MaterialLoader();

  @override
  PageRoute createPageRouter({
    required WidgetBuilder builder,
  }) =>
      MaterialPageRouter(builder: builder);

  @override
  Widget createSwitcher({
    required bool isActive,
    required ValueSetter<bool> onChanged,
  }) =>
      MaterialSwitcher(isActive: isActive, onChanged: onChanged);
}
```

```dart
class CupertinoWidgetsFactory implements PlatformWidgetsFactory {
  const CupertinoWidgetsFactory();

  @override
  PreferredSizeWidget createAppBar({
    required String title,
    bool showSettingsButton = true,
  }) =>
      CupertinoAppBar(title: title, showSettingsButton: showSettingsButton);

  @override
  Widget createBottomNavigationBar({
    required int currentIndex,
    required ValueSetter<int> onTap,
  }) =>
      CupertinoBottomNavigationBar(currentIndex: currentIndex, onTap: onTap);

  @override
  Widget createLoader() => const CupertinoLoader();

  @override
  PageRoute createPageRouter({
    required WidgetBuilder builder,
  }) =>
      CupertinoPageRouter(builder: builder);

  @override
  Widget createSwitcher({
    required bool isActive,
    required ValueSetter<bool> onChanged,
  }) =>
      CupertinoSwitcher(isActive: isActive, onChanged: onChanged);
}
```

@mkobuolys

Abstract Factory: products

```dart
class MaterialSwitcher extends StatelessWidget {
  const MaterialSwitcher({
    required this.isActive,
    required this.onChanged,
    Key? key,
  }) : super(key: key);

  final bool isActive;
  final ValueSetter<bool> onChanged;

  @override
  Widget build(BuildContext context) {
    return Switch(value: isActive, onChanged: onChanged);
  }
}
```

```dart
class MaterialLoader extends StatelessWidget {
  const MaterialLoader({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Center(
      child: CircularProgressIndicator(
        valueColor: AlwaysStoppedAnimation(Theme.of(context).primaryColor),
      ),
    );
  }
}
```

```dart
class CupertinoSwitcher extends StatelessWidget {
  const CupertinoSwitcher({
    required this.isActive,
    required this.onChanged,
    Key? key,
  }) : super(key: key);

  final bool isActive;
  final ValueSetter<bool> onChanged;

  @override
  Widget build(BuildContext context) {
    return CupertinoSwitch(value: isActive, onChanged: onChanged);
  }
}
```

```dart
class CupertinoLoader extends StatelessWidget {
  const CupertinoLoader({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return const CupertinoActivityIndicator();
  }
}
```

@mkobuolys

Abstract Factory: creating the factory

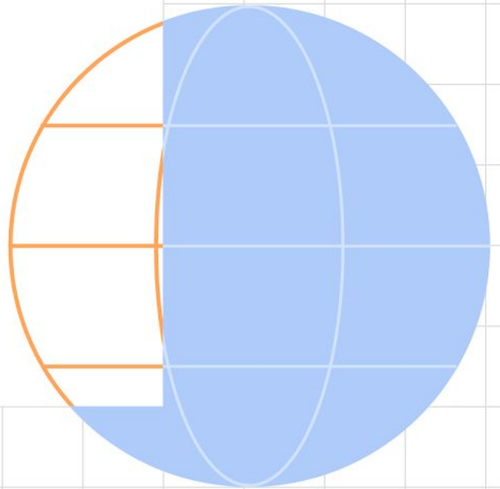
```dart
class App extends StatelessWidget {
  <...>

  PlatformWidgetsFactory _createPlatformWidgetsFactory() {
    switch (defaultTargetPlatform) {
      case TargetPlatform.android:
        return MaterialWidgetsFactory();
      case TargetPlatform.iOS:
        return CupertinoWidgetsFactory();
      default:
        return MaterialWidgetsFactory();
    }
  }

  @override
  Widget build(BuildContext context) {
    final widgetsFactory = _createPlatformWidgetsFactory();

    return Provider<PlatformWidgetsFactory>.value(
      value: widgetsFactory,
      child: MaterialApp(
        <...>
      ),
    );
  }
}
```

Abstract Factory: using the factory

```dart
class SettingsPage extends StatelessWidget {
  <...>

  void _onUseCupertinoWidgetsChanged(bool useCupertino) {
    <...>
  }

  @override
  Widget build(BuildContext context) {
    final l10n = context.l10n;
    final widgetsFactory = context.watch<PlatformWidgetsFactory>();

    return Scaffold(
      appBar: widgetsFactory.createAppBar(
        title: l10n.settingsTitle,
        showSettingsButton: false,
      ),
      body: Column(
        children: <Widget>[
          ListTile(
            title: Text(l10n.useCupertinoLabel),
            trailing: widgetsFactory.createSwitcher(
              isActive: defaultTargetPlatform == TargetPlatform.iOS,
              onChanged: _onUseCupertinoWidgetsChanged,
            ),
          ),
        ],
      ),
    );
  }
}
```

Experts

Composite

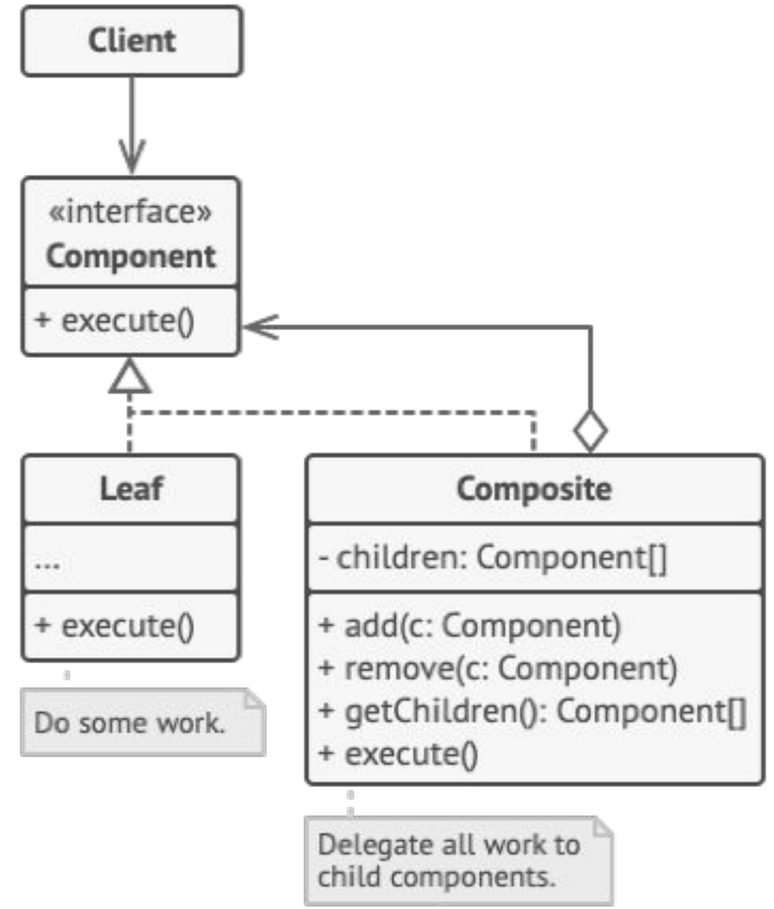


“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”

- GoF

Composite

- Structural design pattern
- *Component* – declares the interface for objects in the composition
- *Leaf* – represents leaf objects in the composition
- *Composite* — stores sub-elements (children) and implements child-related operations in the Composite interface.



Source: https://refactoring.guru/images/patterns/diagrams/composite/structure-en.png

Composite: abstract component

```
1  abstract class MusicLibraryItem {
2    int getItemsCount();
3    int getDuration();
4    Widget build(BuildContext context);
5  }
```

Composite: concrete components

Leaf node

```dart
class MusicLibrarySong extends StatelessWidget implements MusicLibraryItem {
  const MusicLibrarySong({
    required this.data,
    Key? key,
  }) : super(key: key);

  final Song data;

  @override
  int getItemsCount() => 1;

  @override
  int getDuration() => data.duration;

  @override
  Widget build(BuildContext context) {
    <...>
  }
}
```

Composite node

```dart
class MusicLibraryCollection extends StatelessWidget
    implements MusicLibraryItem {
  MusicLibraryCollection({
    required this.data,
    Key? key,
  }) : super(key: key);

  final MusicCollection data;
  final List<MusicLibraryItem> _items = <MusicLibraryItem>[];

  void addItems(List<MusicLibraryItem> items) => _items.addAll(items);

  @override
  int getItemsCount() => _items.fold<int>(
        0,
        (prev, item) => prev + item.getItemsCount(),
      );

  @override
  int getDuration() => _items.fold<int>(
        0,
        (prev, item) => prev + item.getDuration(),
      );

  @override
  Widget build(BuildContext context) {
    <...>
  }
}
```

@mkobuolys

Composite: building the composition

```dart
class MusicLibraryService {
  const MusicLibraryService({
    required this.repository,
  });

  final MusicLibraryRepository repository;

  Future<List<MusicLibraryItem>> getMusicLibraryItems() async {
    final collections = await repository.getCollections();
    final songs = await repository.getSongs();

    await Future.delayed(const Duration(seconds: 2));

    return _buildMusicLibraryItems(collections, songs);
  }

  <...>
}
```

```dart
List<MusicLibraryCollection> _buildMusicLibraryItems(
  List<MusicCollection> collections,
  List<Song> songs,
) {
  final musicLibraryCollectionsMap = <int, MusicLibraryCollection>{
    for (final collection in collections)
      collection.id: MusicLibraryCollection(
        data: collection,
      )
  };

  for (final musicLibraryCollection in musicLibraryCollectionsMap.values) {
    final musicCollection = musicLibraryCollection.data;
    final parentId = musicCollection.parentId;

    if (musicLibraryCollectionsMap.containsKey(parentId)) {
      musicLibraryCollectionsMap[parentId]!.addItems(
        [musicLibraryCollection],
      );
    }
  }

  _addSongsToCollections(musicLibraryCollectionsMap, songs);

  return musicLibraryCollectionsMap.values
      .where((collection) => collection.data.parentId == null)
      .toList();
}
```

@mkobuolys

Composite: rendering

```dart
class _MusicLibraryView extends StatelessWidget {
  const _MusicLibraryView({
    required this.musicLibraryItems,
    Key? key,
  }) : super(key: key);

  final List<MusicLibraryItem> musicLibraryItems;

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Expanded(
          child: ListView.builder(
            itemBuilder: (context, index) => musicLibraryItems[index].build(
              context,
            ),
            itemCount: musicLibraryItems.length,
          ),
        ),
        SummaryText(
          songsCount: musicLibraryItems.fold<int>(
            0,
            (prev, item) => prev + item.getItemsCount(),
          ),
          duration: musicLibraryItems.fold<int>(
            0,
            (prev, item) => prev + item.getDuration(),
          ),
        )
      ],
    );
  }
}
```

As composition

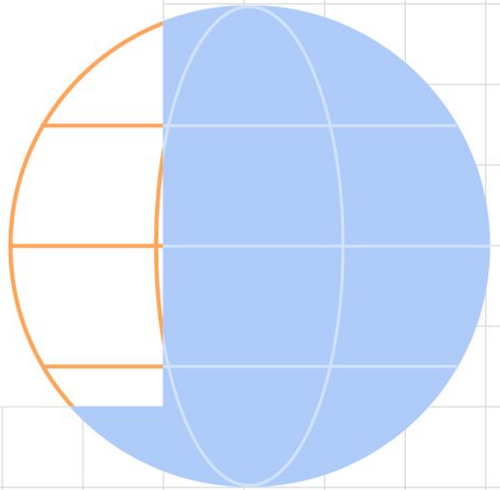
@mkobuolys

```dart
class PlaylistPage extends StatelessWidget {
  static const route = '/playlist';

  const PlaylistPage({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final playlist = context.select<PlaylistCubit, Playlist>(
      (cubit) => cubit.state.playlist,
    );

    return Column(
      children: [
        Expanded(
          child: ReorderableListView(
            children: [
              for (final song in playlist.songs)
                MusicLibrarySong(key: ValueKey(song), data: song),
            ],
            <...>
          ),
        ),
        SummaryText(
          songsCount: playlist.songs.length,
          duration: playlist.songs.fold<int>(
            0,
            (prev, song) => prev + song.duration,
          ),
        )
      ],
    );
  }
}
```

As stand-alone widget

Experts

Command

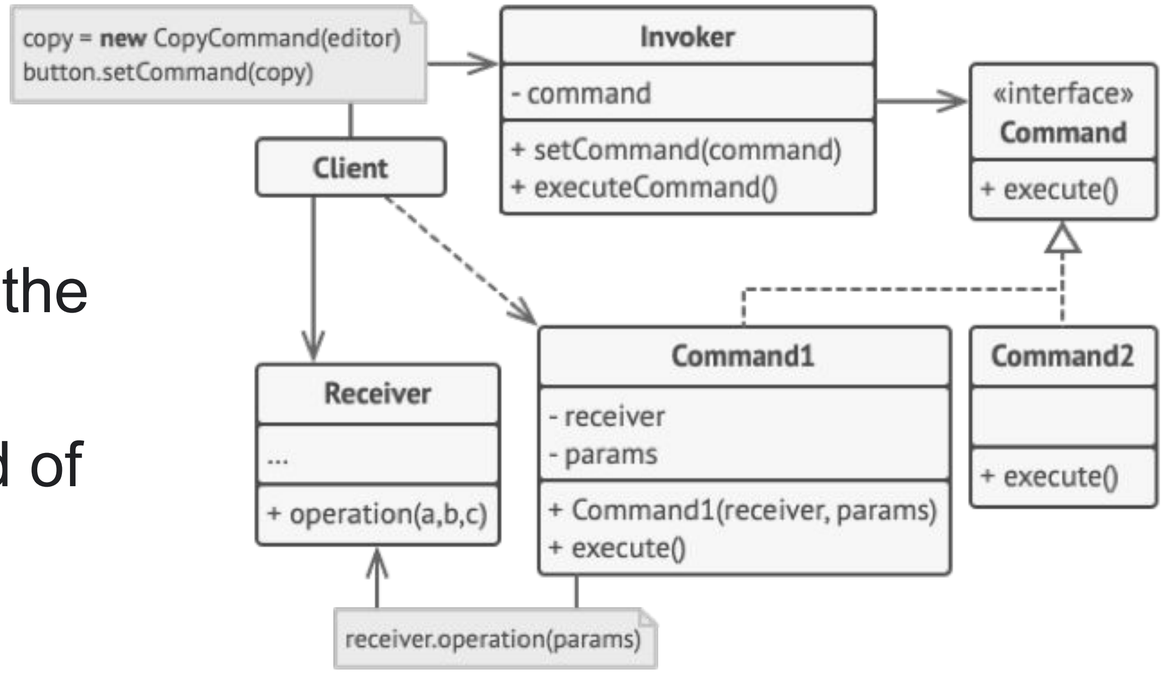


“Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”

- GoF

Command

- Behavioural design pattern
- *Command* – declares an interface for executing an operation
- *Command1*, *Command2* – implements the specific execution logic
- *Invoker* – triggers the command instead of sending the request directly to the *Receiver*
- *Receiver* – knows how to perform the operation



Source: https://refactoring.guru/images/patterns/diagrams/command/structure.png

Command: abstract command and base

```dart
abstract class IPlaylistCommand {
  Playlist execute();
  Playlist undo();
}
```

```dart
abstract class PlaylistCommand implements IPlaylistCommand {
  PlaylistCommand(Playlist playlist)
      : playlist = playlist.copyWith(),
        _songsBackup = [...playlist.songs];

  @protected
  final Playlist playlist;
  final List<Song> _songsBackup;

  @override
  Playlist undo() => playlist.copyWith(songs: _songsBackup);
}
```

Command: state (receiver)

```dart
class Playlist {
  Playlist({
    required this.songs,
  });

  final List<Song> songs;

  Playlist copyWith({List<Song>? songs}) => Playlist(
        songs: songs ?? this.songs,
      );

  @override
  String toString() {
    return 'Playlist { songs: $songs }';
  }
}
```

Command: concrete commands

```dart
class AddToPlaylistCommand extends PlaylistCommand {
  AddToPlaylistCommand({
    required Playlist playlist,
    required this.song,
  }) : super(playlist);

  final Song song;

  @override
  Playlist execute() {
    playlist.songs.add(song);

    return playlist;
  }
}
```

```dart
class RemoveFromPlaylistCommand extends PlaylistCommand {
  RemoveFromPlaylistCommand({
    required Playlist playlist,
    required this.song,
  }) : super(playlist);

  final Song song;

  @override
  Playlist execute() {
    playlist.songs.remove(song);

    return playlist;
  }
}
```

```dart
class ReorderPlaylistCommand extends PlaylistCommand {
  ReorderPlaylistCommand({
    required Playlist playlist,
    required this.song,
    required this.oldIndex,
    required this.newIndex,
  }) : super(playlist);

  final Song song;
  final int oldIndex;
  final int newIndex;

  @override
  Playlist execute() {
    final insertAtIndex = newIndex > oldIndex ? newIndex - 1 : newIndex;

    playlist.songs.removeAt(oldIndex);
    playlist.songs.insert(insertAtIndex, song);

    return playlist;
  }
}
```

Command: execute

```dart
class PlaylistCubit extends Cubit<PlaylistState> {
  PlaylistCubit() : super(PlaylistState(playlist: Playlist(songs: [])));

  final Stack<IPlaylistCommand> _commandHistory = Stack();

  void executeCommand(IPlaylistCommand command) {
    _commandHistory.push(command);

    final playlist = command.execute();

    emit(PlaylistState(playlist: playlist, isCommandHistoryEmpty: false));
  }

  <...>
}
```

```dart
class PlaylistPage extends StatelessWidget {
  static const route = '/playlist';

  const PlaylistPage({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final playlist = context.select<PlaylistCubit, Playlist>(
      (cubit) => cubit.state.playlist,
    );

    return Column(
      children: [
        Expanded(
          child: ReorderableListView(
            children: [<...>],
            onReorder: (oldIndex, newIndex) {
              final command = ReorderPlaylistCommand(
                playlist: playlist,
                song: playlist.songs[oldIndex],
                oldIndex: oldIndex,
                newIndex: newIndex,
              );

              context.read<PlaylistCubit>().executeCommand(command);
            },
          ),
        ),
        <...>
      ],
    );
  }
}
```

@mkobuolys

Command: undo

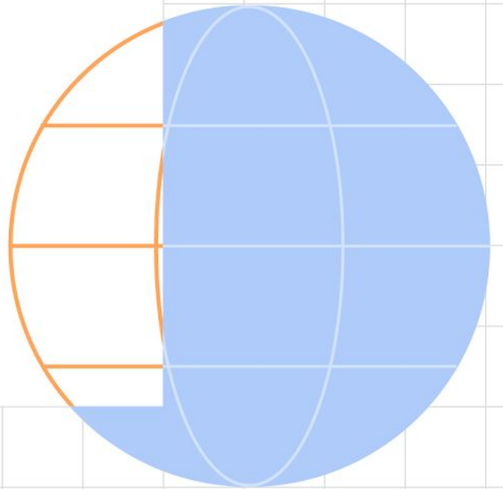
```dart
class PlaylistCubit extends Cubit<PlaylistState> {
  PlaylistCubit() : super(PlaylistState(playlist: Playlist(songs: [])));

  final Stack<IPlaylistCommand> _commandHistory = Stack();

  <...>

  void undoLastCommand() {
    if (_commandHistory.isNotEmpty) {
      final playlistCommand = _commandHistory.pop();
      final playlist = playlistCommand.undo();

      emit(
        PlaylistState(
          playlist: playlist,
          isCommandHistoryEmpty: _commandHistory.isEmpty,
        ),
      );
    }
  }
}
```

```dart
class MaterialAppBar extends StatelessWidget with PreferredSizeWidget {
  const MaterialAppBar({
    required this.title,
    this.showSettingsButton = true,
    Key? key,
  }) : super(key: key);

  final String title;
  final bool showSettingsButton;

  @override
  Widget build(BuildContext context) {
    final isCommandHistoryEmpty = context.select<PlaylistCubit, bool>(
      (cubit) => cubit.state.isCommandHistoryEmpty,
    );

    return AppBar(
      backgroundColor: Colors.black,
      title: Text(title),
      actions: [
        if (!isCommandHistoryEmpty)
          IconButton(
            icon: const Icon(Icons.replay),
            onPressed: () => context.read<PlaylistCubit>().undoLastCommand(),
          ),
      ],
    );
  }

  @override
  Size get preferredSize => const Size.fromHeight(kToolbarHeight);
}
```

@mkobuolys

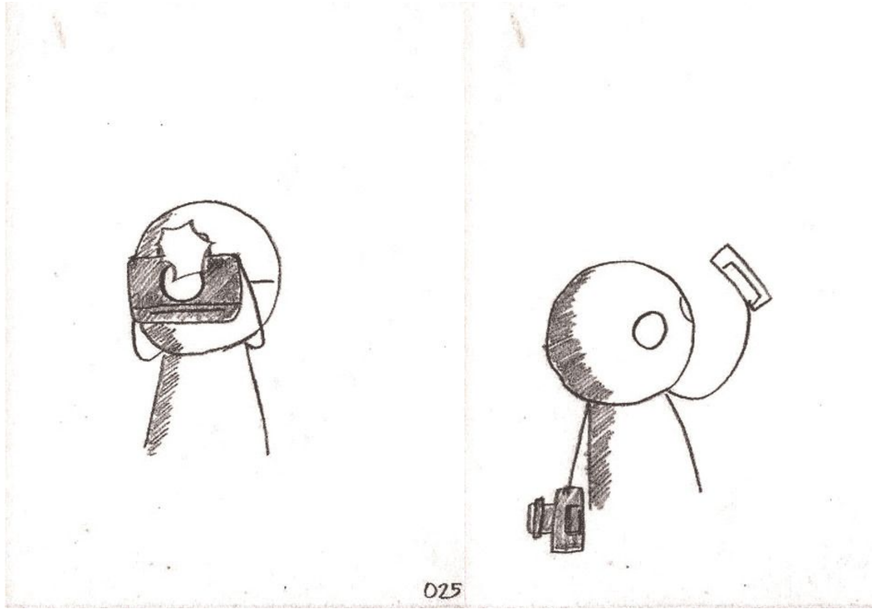
Experts

Memento



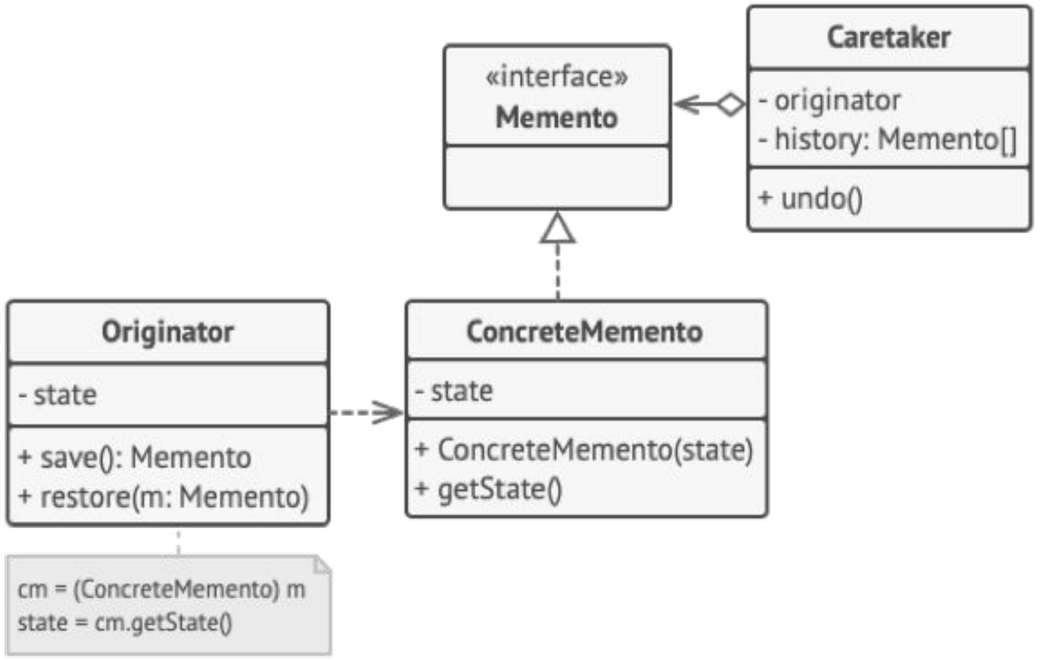
“Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.”

- GoF



Memento

- Behavioural design pattern
- *Memento* – restricts access to the *ConcreteMemento's* fields
- *ConcreteMemento* – stores *Originator's* internal state
- *Caretaker* – only keeps the *Memento*, but never operates or examines its data
- *Originator* - creates a *ConcreteMemento* containing a snapshot of its current internal state.



Source: https://refactoring.guru/images/patterns/diagrams/memento/structure2.png

Memento: components

```dart
abstract class Memento {
  Playlist getState();
}
```

```dart
class Originator {
  Playlist _state;
  Playlist get state => _state;

  Originator(Playlist playlist) : _state = playlist;

  Memento createMemento() => PlaylistMemento(_state);

  void restore(Memento memento) {
    _state = memento.getState();
  }
}
```

```dart
class PlaylistMemento implements Memento {
  final Playlist _state;

  PlaylistMemento(Playlist playlist) : _state = Playlist.copy(playlist);

  @override
  Playlist getState() => _state;
}
```

Memento: usage

```dart
abstract class PlaylistCommand implements IPlaylistCommand {
  PlaylistCommand(this.originator) : backup = originator.createMemento();

  @protected
  final Originator originator;
  @protected
  final Memento backup;

  @override
  Playlist undo() {
    originator.restore(backup);

    return originator.state;
  }
}
```

```dart
class PlaylistPage extends StatelessWidget {
  static const route = '/playlist';

  const PlaylistPage({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final playlist = context.select<PlaylistCubit, Playlist>(
      (cubit) => cubit.state.playlist,
    );

    return Column(
      children: [
        Expanded(
          child: ReorderableListView(
            children: [<...>],
            onReorder: (oldIndex, newIndex) {
              final command = ReorderPlaylistCommand(
                originator: Originator(playlist),
                song: playlist.songs[oldIndex],
                oldIndex: oldIndex,
                newIndex: newIndex,
              );

              context.read<PlaylistCubit>().executeCommand(command);
            },
          ),
        ),
        <...>
      ],
    );
  }
}
```

Memento: abstract command without vs with memento

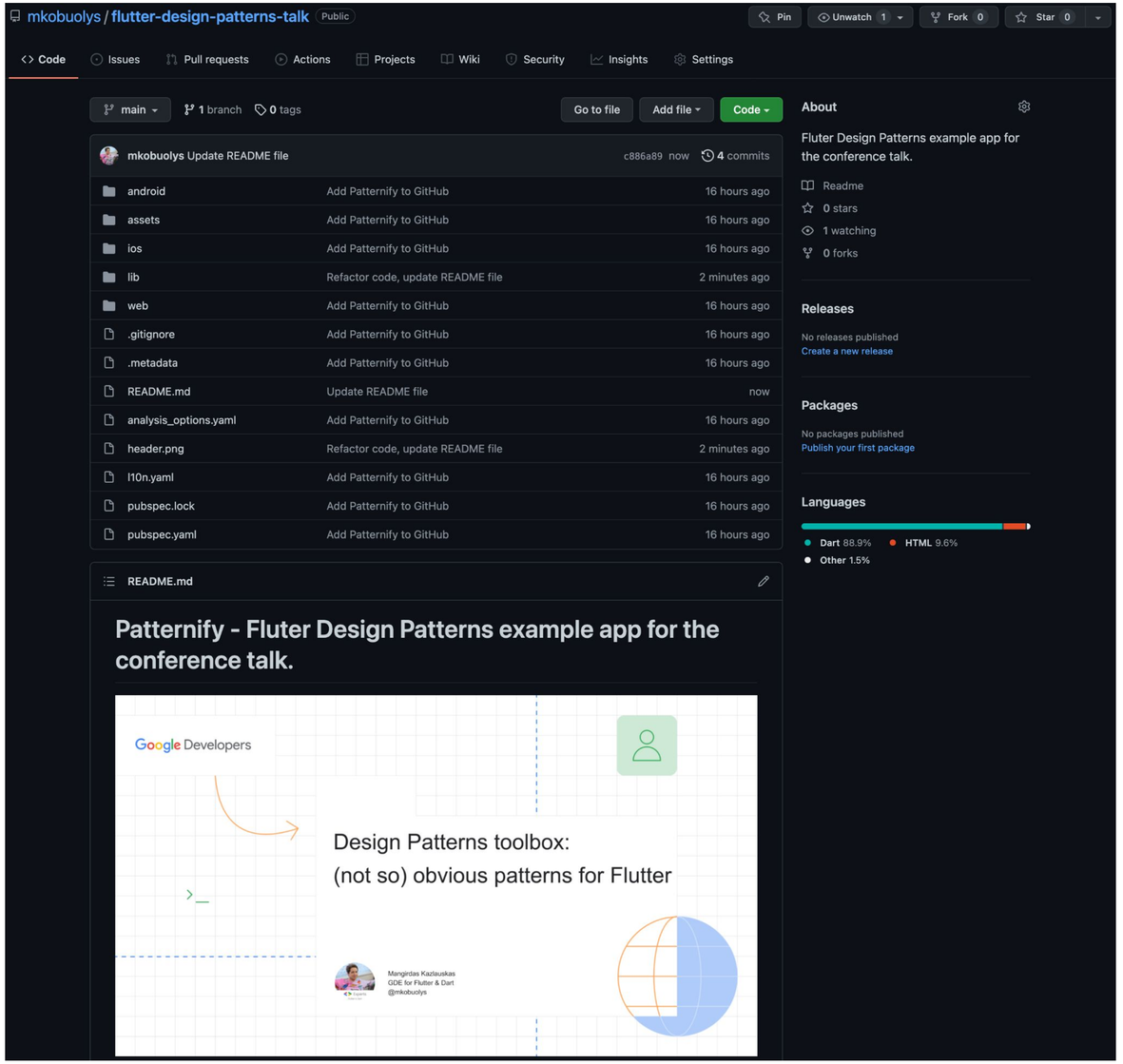
```
abstract class PlaylistCommand implements IPlaylistCommand {
  PlaylistCommand(Playlist playlist)
      : playlist = playlist.copyWith(),
        _songsBackup = [...playlist.songs];

  @protected
  final Playlist playlist;
  final List<Song> _songsBackup;

  @override
  Playlist undo() => playlist.copyWith(songs: _songsBackup);
}
```

```
abstract class PlaylistCommand implements IPlaylistCommand {
  PlaylistCommand(this.originator) : backup = originator.createMemento();

  @protected
  final Originator originator;
  @protected
  final Memento backup;

  @override
  Playlist undo() {
    originator.restore(backup);

    return originator.state;
  }
}
```

Memento: concrete command without vs with memento

```dart
class AddToPlaylistCommand extends PlaylistCommand {
  AddToPlaylistCommand({
    required Playlist playlist,
    required this.song,
  }) : super(playlist);

  final Song song;

  @override
  Playlist execute() {
    playlist.songs.add(song);

    return playlist;
  }
}
```

```dart
class AddToPlaylistCommand extends PlaylistCommand {
  AddToPlaylistCommand({
    required Originator originator,
    required this.song,
  }) : super(originator);

  final Song song;

  @override
  Playlist execute() {
    final playlist = backup.getState();

    return playlist.copyWith(songs: [...playlist.songs, song]);
  }
}
```



https://github.com/mkobuolys/flutter-design-patterns-talk

Mangirdas Kazlauskas

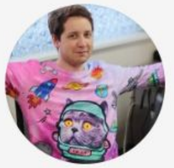
Home Lists About

Flutter Design Patterns

An overview of all 23 Design Patterns from the GoF book "Design Patterns: Elements of Reusable Object-Oriented Software" implemented in Dart & Flutter.

Jul 27, 2021 · 24 stories

Add a note...



Mangirdas Kazlauskas

2.1K Followers

Google Developer Expert for Flutter & Dart | Software Engineer - https://twitter.com/mkobuolys

Edit profile

Lists

Flutter Design Patterns
24 stories

View All

https://mkobuolys.medium.com/

Flutter Design Patterns

Created with Flutter and ❤

Creator: Mangirdas Kazlauskas

Creational

5 patterns

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton

Ensure a class only has one instance, and provide a global point of access to it.

Structural

7 patterns

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade

Provide an unified interface to a set of interfaces in a subsystem.

Behavioral

11 patterns

Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

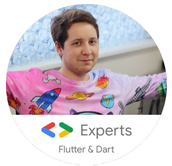
Mediator

https://flutterdesignpatterns.com/

Google Developers

Experts

Save trees. Stay SOLID. Thank you!



Mangirdas Kazlauskas
GDE for Flutter & Dart
@mkobuolys