Diego Bortolussi
Matthieu Candalot
Argan Da Costa
Gael Garrido Moran

Diagrams Explanations

Diagram Use Case Explanation:

This diagram shows what a user can do with the EasySave system and how the system interacts with other components (like the file system).

1. Actors

- User: the person who uses the software.
- FileSystem: the computer's file system, where backups and logs are stored.

The stick figures represent the actors.

2. The Blue Rectangle

The large blue box represents the EasySave system.

Everything inside it shows the features provided by the software.

3. Use Cases (Ovals)

Each oval represents an action that the user can perform.

Examples:

- Create Job: create a backup task
- Modify Job: modify a task
- Delete Job: delete a task

In short: what the user can do with the software.

4. Relationships (<<include>>)

When a use case includes another one (<<include>>), it means:

This action automatically uses another action.

Example:

- Start single or multiple jobs includes Execute backup, Write logs, and Write real-time status.

5. Extern actos :

The FileSystem is used to:

- Store backups,
- Save logs,
- Store job configurations.

Cryptosoft is used to :

- Encrypt the different files

Sequence Diagram Explanation

This diagram shows how different parts of the EasySave system communicate with each other over time when a user launches a backup.

1. Participants (Vertical Lines)

Each vertical column represents a component of the system, such as:

- User
- ConsoleRunner or ExecuteBackupMenuViewModel
- BackupController or BackupAppService
- Etc…

They show who is involved in the process.

2. Time Flow (Top to Bottom)

Time flows from top to bottom.

This means:

- What is at the top happens first,
- What is lower happens later.

3. Messages (Arrows)

The arrows show method calls and responses between components.

Example:

- The User starts a backup.

- Files are copied, logs are written, and progress is updated.

4. Main Steps of the Process

In simple terms, the diagram describes:

1. The user launches a backup job.
2. The controller selects the backup type (full or differential).
3. The system retrieves the list of files to copy.
4. Files are compared (for differential backup).
5. Files are copied to the destination folder.
6. The system updates progress in real time.
7. Logs and status files are written and updated.
8. The backup ends and confirms completion.

5. Loops and Conditions

- Loops show repeated actions, such as copying each file and each subfolder.
- Conditions (if / else) show different behaviors depending on the backup type (full or differential).

Activity Diagram Explanation:

An activity diagram shows how a process works step by step, from start to end. It helps you understand what happens, in which order, and under which conditions.

- Start and End
  The process begins with a filled black circle (start) and ends with a black circle surrounded by a ring (end).
- Actions (rounded rectangles)
  Each rounded rectangle represents an action or task that the system or the user performs.
- Arrows
  Arrows show the flow of the process, meaning what happens next after each action.
- Decisions (diamonds)
  Diamonds represent a choice or condition (for example: Yes / No).
  Depending on the answer, the flow follows different paths.

- Swimlanes (vertical or horizontal sections)
  Swimlanes divide the diagram into areas to show who is responsible for each action (for example: User vs System).
- Loops
  If arrows go back to a previous step, it means the process repeats until a condition is met.

Overall, you can read an activity diagram like a flowchart: start at the top (or left), follow the arrows, make decisions when needed, and continue until the end.

Component Diagram :

A component diagram shows the structure of a system by presenting its main parts (called components) and how they interact with each other. It focuses on what the system is made of:

- Components (rectangles):
  Each rectangle represents a software component, such as a module, service, or application part.
  A component usually has a specific responsibility (for example: user interface, backup engine, logging).
- Interfaces (lollipop or socket symbols):
  Interfaces show how components communicate.
  - A provided interface means what the component offers.
  - A required interface means what the component needs from another component.
- Connections (lines):
  Lines between components indicate dependencies or communication.
  They show that one component uses or depends on another.

Overall, a component diagram helps to understand how the system is organized, which parts exist, and how they depend on each other, without going into implementation details.

Class Diagram Explanation:

Architectural Layers

A. EasyLog

This layer is responsible for low-level data persistence regarding application activity.

- EasyLogService: Implements the Singleton pattern to provide a global access point for logging.
- JsonLogWriter: Decouples the logging logic from the file system, specifically handling JSON serialization.

B. Domain

The core of the system. It contains the business rules and entities without any dependency on the UI or database.

- Models: Entities like BackupJob (stores configuration) and BackupProgress (tracks real-time state).
- Services: Logic for file manipulation and backup orchestration.
- Interfaces: Define the "contracts" that the other layers interact with.

C. Application

This layer acts as an intermediary. It translates user actions from the UI into business logic calls.

- Controllers: Manage the flow of data.
- DTOs (Data Transfer Objects): Lightweight objects used to pass data to the UI without exposing the internal logic of the Domain entities.

D. Console

The user interface. It handles display logic and user input.

- Resources: Manages internationalization (i18n).
- ConsoleUI: Contains the visual structure of the menus.
- Commands: Orchestrates the interaction loops.

E. GUI

- ViewModels : Manage the link between the GUI and the application

- Views : Contains the visual structure of the views

D. Cryptosoft

Contain the logic for the encryption and the decryption

Relationships

The strength of this diagram lies in the precise use of UML connectors to represent code structure.

A.1 Realization (Dashed line with hollow triangle ……|>)

This is used between Interfaces and Classes.

- Logic: It signifies that a class "fulfills the contract" of an interface.
- Example: BackupService ……|> IbackupService.
- Impact: The rest of the application (like the BackupController) only knows about the interface. This allows you to replace the BackupService with a new version without changing a single line of code in the Controller.

B. Generalization / Inheritance (Solid line with hollow triangle ——|>)

This represents an "is-a" relationship between two classes.

- Logic: A child class inherits all properties and methods from a parent class.
- Example: CreateBackupMenu ——|> GeneralContent.
- Impact: All menus share the same Header() and Footer() logic defined in GeneralContent, reducing code duplication.

C.1. Association (Solid line with open arrow ——>)

This represents a structural relationship where one class "has" or "knows" another as a field/property.

- Logic: Indicates a long-term relationship.
- Example: ConsoleRunner ——> BackupController.

- Impact: The ConsoleRunner holds a reference to the controller to delegate user commands. In the code, this is usually initialized via the constructor.

C.2. Association Roles (_runner, _menu, etc.)

On many association lines, you will see labels at the arrowheads. These represent Property Names (Fields) in the code.

- _runner : ConsoleRunner: Found in the Commands package. It indicates that every Interaction class (e.g., CreateBackupMenuInteraction) holds a reference to the central engine to execute logic.
- _menu: Found in interactions. It indicates the specific UI class the command is currently "driving."
- _texts : ITextProvider: Found in GeneralContent. This ensures all UI elements have a shared reference to the translation dictionary.

D. Dependency (Dashed line with open arrow - - - - - >)

This represents a transient relationship.

- Logic: A class "uses" another temporarily (as a parameter in a method or a local variable).
- Example: BackupController - - - - - > BackupJobDTO.
- Impact: The controller creates or returns a DTO, but it doesn't "own" it as a permanent member.

E. Multiplicity (Cardinality)

Multiplicity defines how many instances of one class are associated with another.

- 1 (Exactly One): Used for mandatory dependencies. For example, ConsoleRunner is linked to exactly 1 BackupController.
- 0..* or * (Zero to Many): Indicates a collection (List). For example, BackupManagerService is linked to * BackupJob, meaning it manages a list of tasks.
- 1..1: A strict one-to-one relationship, often used for injected services.