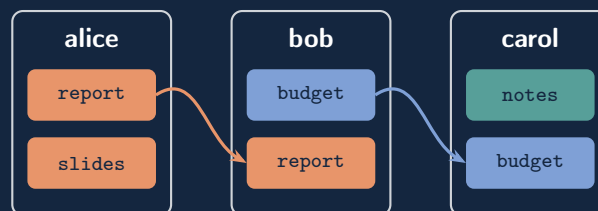




UNIVERSITÀ DI PISA

SECURE DATA CONTAINER

identity • access control • ownership



the color of an item is its first owner

```
interface SecureDataContainer<E>
    share(owner, passw, other, data)

    Item(owner, firstOwner, data)
```

Java • generics • data abstraction

Secure Data Container: a Generic, Access-Controlled Data Abstraction in Java

Project Report

Giuseppe Muschetta

B.Sc. in Computer Science

Course: Programming II • Course Instructor: Prof. Gian-Luigi Ferrari

Abstract

This report presents the Java part of the Programming II project: `SecureDataContainer<E>`, a generic collection that behaves like a small data storage service. Users register with an identifier and a password, store objects of any type, and share them with other users, while the container enforces access control: every operation on data authenticates the caller, a user sees only the data they own or that others shared with them, and the user who inserted an item, its *first owner*, keeps control over every copy they shared. The abstraction is written *specification first*: an interface documents each of its 18 operations with its preconditions, side effects, results and exceptions, in the style of Liskov and Guttag, and two separate classes implement it on completely different representations: a single list of labeled items, and one list per user. Each class states its abstraction function and its representation invariant. A test driver runs the same scenario on both implementations through the interface type only, covering registration, insertion, sharing, removal by the first owner and by a recipient, copies, clearing and account deletion: at every step the two implementations show every user exactly the same data.

Contents

1	Introduction	2
2	The specification	2
2.1	Overview of the type	2
2.2	The contracts	2
2.3	Operations and exceptions	3
3	The ownership model	4
4	First implementation: one labeled list	5
4.1	Representation	5
4.2	Abstraction function and representation invariant	5
4.3	Operations	5
5	Second implementation: one list per user	6
5.1	Representation	6
5.2	Equality as a search key	6
5.3	Operations	7
5.4	Two representations compared	7
6	Testing	7
6.1	The test driver	7
6.2	Results	8
7	Conclusion	9

1 Introduction

Many applications store data on behalf of several users at once: a cloud folder, a shared drive, a document service. Such a store needs more than a collection. It has to know *who* is asking, keep each user's data apart from everyone else's, and let the owner of a piece of data decide who else may see it. The first part of the Programming II project asks for exactly this abstraction, **SecureDataContainer<E>**: a container of objects of a generic type **E** that provides its own mechanism for user identities and an access-control mechanism that lets the owner of some data grant access to it selectively.

The project is also an exercise in method. Programming II teaches how to design a data abstraction before writing it: first a *specification* that says what the type means and what each operation does, then one or more *implementations*, each documented with the bridge between the concrete fields and the abstract value they represent [1]. The work was organized accordingly:

1. the interface **SecureDataContainer<E>**, with an overview of the type and a contract for each of its 18 operations (Section 2);
2. the ownership model that the contracts encode, built on a single label, the first owner (Section 3);
3. two implementations on different representations, **FirstSecureDataContainer** and **SecondSecureDataContainer**, each with its abstraction function and representation invariant (Sections 4 and 5);
4. a test driver that runs the same scenario on both (Section 6).

The code uses only the Java standard library (`java.util`) and is made of eight files: the interface, the two implementations, three checked exceptions, an immutable record type used as test data, and the driver. The project's second part, an interpreter for a small functional language written in OCaml, is described in a separate report.

2 The specification

2.1 Overview of the type

The interface opens with an *overview*, the informal description of the abstract values: a **SecureDataContainer<E>** is a container of objects of type **E** that behaves like a data storage service for storing and sharing data. It must guarantee data security through its own management of user identities, and an access-control mechanism through which the owner of some data can grant selective and restrictive access to it: some users may be authorized by the owner, while the others cannot access the data. The *typical element* associates each user with a collection of zero or more data elements.

2.2 The contracts

Every operation takes the caller's identifier and password, so that the container can authenticate the caller before touching any data. Each one is documented with four clauses:

- **REQUIRES**, the precondition: here, that no argument is `null`;
- **THROWS**, the exceptions and the conditions that raise them;
- **MODIFIES**, the parts of the state that the operation may change (omitted for observers);
- **EFFECTS**, what the operation does or returns when the precondition holds.

Listing 1 shows the contract of **remove**, where the access rules are most visible.

Listing 1: The contract of `remove`, from `SecureDataContainer.java`.

```
E remove(String owner, String passw, E data) throws DataNotFoundException,
    UserNotFoundException;
/*
REQUIRES: owner != null && passw != null && data != null
THROWS: - NullPointerException if !(owner != null && passw != null) (UNCHECKED)
        or if data == null
        - UserNotFoundException if user is not found in the container (CHECKED)
        - DataNotFoundException if data is not found in the user owned items (CHECKED)
MODIFIES: this
EFFECTS: If user is the firstOwner the method will delete his data and all the same data
        that he shared to others, otherwise (if user is not the firstOwner) it will only
        delete his data
*/
```

The first-owner rule is part of the contract, not an implementation detail: every implementation must respect it, and a client can rely on it without knowing how the data are stored.

2.3 Operations and exceptions

The interface has 18 operations, listed in Table 1: nine required by the assignment and nine more, marked as optional in the interface.

Table 1: The 18 operations of `SecureDataContainer<E>`. Every operation authenticates the caller first.

Operation	Effect
<i>Required</i>	
<code>createUser(id, passw)</code>	registers a new user; the identifier must not exist yet
<code>removeUser(id, passw)</code>	removes the user, their data and every copy of their data shared with others
<code>getSize(owner, passw)</code>	number of items the user can access, their own and those shared with them
<code>put(owner, passw, data)</code>	inserts an item; the caller becomes its first owner
<code>get(owner, passw, data)</code>	returns an item the caller can access
<code>remove(owner, passw, data)</code>	applies the first-owner rule of Listing 1
<code>copy(owner, passw, data)</code>	adds to the caller's data a copy of an item they can access
<code>share(owner, passw, other, data)</code>	gives another registered user access to an item
<code>getIterator(owner, passw)</code>	iterates over the caller's items; removal is not supported
<i>Optional</i>	
<code>removeAll(owner, passw, data)</code>	removes from the caller's collection all the elements equal to data
<code>clear(owner, passw)</code>	removes the caller's data and every copy they shared
<code>isEmpty, isIn</code>	whether the caller has no items, or has a given one
<code>printTotalData</code>	prints every item the caller can access
<code>printFirstOwnerData</code>	prints only the items the caller inserted
<code>printSharedFromOthersData</code>	prints only the items shared with the caller
<code>printUsersId, totalUsers</code>	the registered identifiers, sorted, and their number

Errors are divided by kind. A `null` argument violates the precondition: it is a programming error of the caller, signaled by the unchecked `NullPointerException`. The situations that can legitimately happen at run time, and that a client is expected to handle, are signaled by three *checked* exceptions defined by the project:

- `ExistingUserException`, when `createUser` is called with an identifier already in use;
- `UserNotFoundException`, when the identifier is unknown or the password does not match it: the two cases are reported in the same way, so a caller cannot use the error to tell which identifiers exist;

- `DataNotFoundException`, when the caller has no access to the requested item.

Because they are checked, the compiler forces every client to either handle them or declare them, which keeps the access-control failures visible in the code that uses the container.

3 The ownership model

All the access rules follow from a single piece of information attached to every stored item: its **first owner**, the user who inserted it. A user can access an item either because they inserted it or because its first owner shared it with them; in both cases the item remembers its first owner. Figure 1 shows the rules on a small example, and the list below states them.

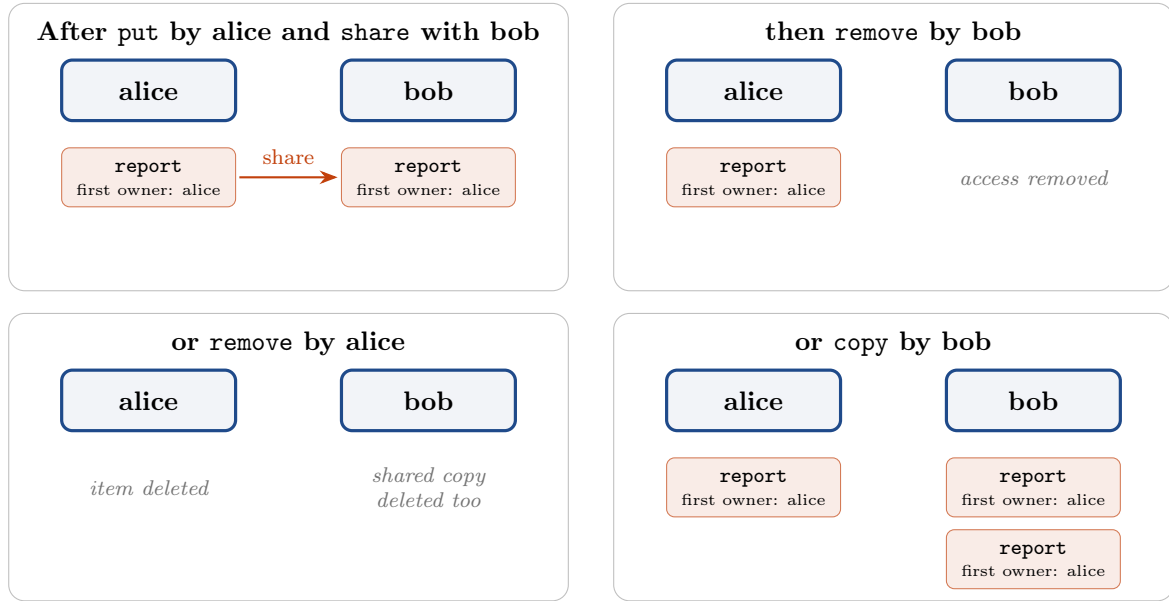


Figure 1: The first-owner rules on one item. From the state at the top left, the other panels show the effect of `remove` by the recipient, of `remove` by the first owner and of `copy` by the recipient.

- **Insertion.** `put` stores the item with the caller as its first owner.
- **Sharing.** `share` gives a registered user a copy of an item the caller can access; the copy keeps the original first owner.
- **Removal.** If the caller is the first owner, `remove` deletes the item and every copy of it that they shared; otherwise it removes only the caller's own access, and the first owner keeps the item.
- **Copies.** `copy` duplicates an item in the caller's collection; the duplicate still belongs to the original first owner, so copying never turns a recipient into an owner.
- **Clearing and leaving.** `clear` and `removeUser` delete the caller's items together with every copy shared with others, so nothing outlives its owner's decision.

The consequence is a simple guarantee for the owners of the data: whatever the recipients do, sharing never gives away control. A recipient can drop a shared item for themselves, but cannot delete it for anyone else, and cannot keep a copy once the first owner removes the original.

4 First implementation: one labeled list

4.1 Representation

`FirstSecureDataContainer<E>` keeps the users and their passwords in a `HashMap<String, String>` and *every item of every user* in a single `LinkedList`. Each element of the list is an instance of the private inner class `Item`, which stores the data together with two labels: `owner`, the user who can access this copy, and `firstOwner`, the user who inserted the data (Listing 2). When a user inserts an item, both labels hold their name; sharing adds a new `Item` with the recipient as `owner` and the same `firstOwner`.

Listing 2: The representation of the first implementation (comments condensed).

```
private class Item {
    private String owner;           // who has access to this copy
    private String firstOwner;      // who inserted the data
    private E data;
}
private Map<String, String> users;  // user ID -> password, a HashMap
private List<Item> container;      // every item of every user, a LinkedList
```

4.2 Abstraction function and representation invariant

The **abstraction function** maps a concrete state to the abstract value it represents: every registered user, that is every key of `users`, is associated with the multiset of the `data` of the items whose `owner` is that user. The **representation invariant** lists the conditions that every concrete state must satisfy:

$$\begin{aligned} & \text{users} \neq \text{null} \wedge \text{container} \neq \text{null} \\ & \wedge \forall i. 0 \leq i < \text{container.size()} \Rightarrow \text{container.get}(i) \neq \text{null} \\ & \wedge \text{the keys of users are unique and not null} \wedge \text{its values are not null,} \end{aligned}$$

and each `Item` has its own invariant, `owner` $\neq$ `null` $\wedge$ `firstOwner` $\neq$ `null` $\wedge$ `data` $\neq$ `null`, enforced by its constructor, which throws `NullPointerException` otherwise. The operations check their arguments before changing the state, and every new item goes through that constructor, so the invariant is preserved.

4.3 Operations

With all the items in one list, each operation is a scan that selects items by their labels: `getSize` counts the items whose `owner` is the caller, `getIterator` collects their data, `share` appends a relabeled `Item`, and `clear` and `removeUser` delete every item where the caller appears as `owner` or as `firstOwner`. Listing 3 shows how `remove` applies the first-owner rule on the labels.

Listing 3: The body of `remove` in the first implementation (comments translated from Italian).

```
if (userIsInTheContainer(owner, passw)) {
    if (userOwnsTheData(owner, data)) {
        int index = indexOfUserOwnsTheData(owner, data);
        E copy = container.get(index).data;
        if (container.get(index).firstOwner.equals(owner)) {
            // I am the first owner: delete the copies shared with others
            for (int i = 0; i < container.size(); i++) {
                if (container.get(i).firstOwner.equals(owner)
                    && container.get(i).data.equals(data)
                    && !(container.get(i).owner.equals(owner))) {
                    container.remove(i);
                    i--; // to compensate the removal
                }
            }
        }
    }
}
```

```

    }
    }
    int finalIndex = indexOfUserOwnsTheData(owner, data);
    container.remove(finalIndex);
} else {
    // I am not the first owner: drop only my own access
    container.remove(index);
}
return copy;
} else
    throw new DataNotFoundException("Element has not been found!");
} else
    throw new UserNotFoundException("User not found in the container");

```

The iterator is built on a snapshot: `getIterator` copies the caller's data into a new `ArrayList` and returns an anonymous `Iterator<E>` over it, whose `remove` throws `UnsupportedOperationException` as the contract requires. Since the snapshot is private to the iterator, a client cannot use it to reach the container's internal list.

5 Second implementation: one list per user

5.1 Representation

`SecondSecureDataContainer<E>` organizes the same information the other way around. The container is an `ArrayList` of users, and each user, an instance of the inner class `User`, carries their identifier, their password and *their own* list of items. Since an item now lives in the list of the user who can access it, it no longer needs an `owner` label: the inner class `Dato` (Italian for *datum*) stores only the data and its first owner (Listing 4).

Listing 4: The representation of the second implementation (comments condensed).

```

private class Dato<E> {
    private String firstOwner;
    private E data;
}
private class User<E> {
    private String id;
    private String passw;
    ArrayList<Dato<E>> container; // the items this user can access
}
private ArrayList<User<E>> users;

```

The abstraction function associates each `User` in `users`, recognized by identifier and password, with the multiset of the `data` in its `container`. The representation invariant requires `users` and its elements to be non-null and the identifiers to be pairwise distinct; each `User` requires non-null fields and elements, and each `Dato` a non-null first owner and data.

5.2 Equality as a search key

The second implementation delegates its searches to the methods of `ArrayList`, by giving the inner classes the right notion of equality. Two `User` objects are equal when their identifiers are equal, so the position of a user is found with `users.indexOf(new User<>(id, ""))`; two `Dato` objects are equal when their data are equal, whatever their first owner, so an item is found in a user's list with `indexOf(new Dato<>("", data))`. A temporary object with a dummy field acts as the search key, and `indexOf` and `contains` do the scan.

5.3 Operations

Most operations now touch one list only, the caller's: `put` appends a `Dato` labeled with the caller, `getSize` and `isEmpty` read the size of that list, `share` appends a copy of the `Dato` to the recipient's list, and the iterator walks the caller's list directly. Only the operations that act on copies held by other users visit all the lists: the first owner's `remove`, `clear` and `removeUser`. For `remove`, a private helper deletes the shared copies from the other users once the caller's own item is gone:

Listing 5: Removing the copies shared by the first owner, in the second implementation.

```
private void removeSharedData(String owner, E data) {
    for (User<E> user : users) {
        for (int i = 0; i < user.container.size(); i++) {
            if (user.container.get(i).firstOwner.equals(owner) && user.container.get(i).data.
                equals(data)
                && !(user.id.equals(owner))) {
                user.container.remove(i);
                i--;
            }
        }
    }
}
```

5.4 Two representations compared

Table 2 puts the two designs side by side. They store the same abstract value, but distribute the information differently: the first keeps both labels on every item and finds a user's data by filtering one shared list; the second encodes the `owner` label in the position of the item, inside the list of the user who can access it. The second design makes the common operations on one user's data independent of how much the others have stored, at the price of a slightly more complex structure; the first keeps the structure flat and every rule visible on the labels.

Table 2: The two representations of the same abstraction.

	First implementation	Second implementation
Users	HashMap from identifier to password	ArrayList of User objects
Items	one LinkedList of Item for all users	one ArrayList of Dato per user
Who can access	the <code>owner</code> label of each item	the list that contains the item
Who inserted	the <code>firstOwner</code> label	the <code>firstOwner</code> label
A user's own data	selected by filtering the shared list	the user's list itself
Searches	explicit loops on the labels	<code>indexOf</code> and <code>contains</code> with <code>equals</code>

6 Testing

6.1 The test driver

The class `Main` runs the same scenario on each implementation. The container is always declared through the interface type, `SecureDataContainer<Esame>`, so the scenario can only use the operations of the specification: if it runs unchanged on both classes, both implement the same abstraction. The test data are instances of `Esame` (Italian for *exam*), an immutable type with its own abstraction function and representation invariant, which represents the result of an exam: the name of the exam, a student number and a grade between 0 and 30. Its `equals` compares

all three fields.¹

The scenario involves three users, **paolo**, **giuseppe** and **francesco**, and prints every user's view of the container after each step. Table 3 reports the steps and the views that were actually printed when the driver was run for this report.

Table 3: The test scenario and the data each user can access after every step, as printed by the driver. The same output is produced by both implementations.

Step	giuseppe	paolo	francesco
three users register; each inserts two records	pr2, elaboratori	pr1, algoritmica	logica, analisi
giuseppe shares pr2 with paolo and francesco; francesco shares logica and paolo shares algoritmica with giuseppe	pr2, elaboratori, logica, algoritmica	pr1, algoritmica, pr2	logica, analisi, pr2
giuseppe, first owner, removes pr2	elaboratori, logica, algoritmica	pr1, algoritmica	logica, analisi
giuseppe removes logica, shared with him	elaboratori, algoritmica	—	logica, analisi
giuseppe copies algoritmica	elaboratori, algoritmica, algoritmica	—	—
paolo calls <code>clear</code>	elaboratori	<i>none</i>	—
giuseppe shares elaboratori with francesco	—	—	logica, analisi, elaboratori
giuseppe deletes his account	<i>removed</i>	—	logica, analisi

A dash means that the driver does not print that user's view at that step.

Each step checks one rule of Section 3. The removal of **pr2** by its first owner deletes the copies held by **paolo** and **francesco**, while the removal of **logica** by **giuseppe**, a recipient, leaves the original with **francesco**. The copy of **algoritmica** gives **giuseppe** two entries of a record that still belongs to **paolo**: when **paolo** calls `clear`, both disappear together with **paolo**'s own data. Finally, when **giuseppe** deletes his account, the record **elaboratori** that he had shared disappears from **francesco**'s view too. Before the data operations, `printUsersId` lists the three users in alphabetical order: **francesco**, **giuseppe**, **paolo**.

6.2 Results

The project was compiled and the driver run with both scenarios enabled (the second one sits in a comment block of `Main`, so that the two can be run one at a time). The two outputs list exactly the same items, in the same order, at every step; the only difference between them is cosmetic, in how an empty collection is printed (`[none]` by the first implementation, an empty line by the second). Two representations with little in common, one observable behavior: this is what the specification promises, and what the driver shows. The driver also contains, commented out, the calls that trigger the checked exceptions, such as registering an existing identifier, using a wrong password or copying an item the caller cannot access.

¹In the scenario the records are named after B.Sc. exams, in Italian as in the source: **pr1** and **pr2** (Programming I and II), **algoritmica** (Algorithms), **logica** (Logic), **analisi** (Calculus), **elaboratori** (Computer Architecture).

7 Conclusion

The Secure Data Container shows the main idea of data abstraction at work: a client of the container depends only on its specification, and two implementations with different data structures can be exchanged without the client noticing. Writing the contracts first made the design decisions explicit before any code existed: what counts as a programming error and what as a recoverable failure, what a recipient may do with a shared item, and what happens to the copies when their owner leaves. The single first-owner label then turned out to be enough to express all the access rules, and each implementation could focus on representing it well, the first with two labels on every item of one list, the second with one list per user. The abstraction functions and representation invariants documented in both classes are the link between the two levels, and the shared test scenario is the practical evidence that they describe the same abstraction.

References

- [1] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.
- [2] J. Bloch. *Effective Java*, third edition. Addison-Wesley, 2018.
- [3] Oracle. *The Java Tutorials: Generics*. <https://docs.oracle.com/javase/tutorial/java/generics/>.
- [4] Oracle. *Java SE API Specification*: `java.util` (`HashMap`, `LinkedList`, `ArrayList`, `Iterator`).