



UNIVERSITÀ DI PISA

ADVANCED PROGRAMMING

components • reflection • types • decorators



```
@Testable @Specification(...)
```

```
data MSet a = MS [(a, Int)]
```

```
@bb_type_check(2)
```

Java • Haskell • Python

Components, Reflection, Types and Decorators: Four Exercises in Java, Haskell and Python

Project Report

Giuseppe Muschetta

M.Sc. in Data Science & Business Informatics

Course: Advanced Programming • Course Instructor: Prof. Andrea Corradini

Academic Year: 2024–2025

Abstract

The Advanced Programming course studies the mechanisms that sit behind a programming language: component models, run-time reflection, type classes and metaprogramming. This report presents complete solutions to the four programming exercises of the course, each in the language where its idea is most natural. In Java, the Matching Pairs game is built from JavaBeans that cooperate only through events: the rules of the game are enforced by vetoing changes of a constrained property, and the half-second delay between turns runs on the Swing event thread, so that no state is shared between threads. Still in Java, a unit-testing tool reads the `@Testable` and `@Specification` annotations through reflection, classifies every test in one of four outcomes and reports the methods in source order by reading the method table of the class file. In Haskell, a multiset type built on a list of pairs gets `Eq` and `Foldable` instances and a map function, with an argument for why it cannot be a `Functor`; anagram classes are then counted on four text files. In Python, three decorators use the `inspect` module to check type hints at run time and to block wrongly typed calls a bounded number of times. Every part was verified: the expected outputs provided with the exercises are reproduced byte for byte, the game was played through its real user interface by 43 automated checks, and 17 laws of the multiset type were verified on 76,494 cases.

Contents

1	Introduction	2
2	Java Beans: the Matching Pairs game	2
2.1	The task	2
2.2	The JavaBeans component model	3
2.3	Architecture	3
2.4	The rules are enforced by vetoes	3
2.5	A turn, step by step	4
2.6	One thread for everything	5
2.7	New rounds	5
2.8	Optional extensions and packaging	6
2.9	Verification	6
3	Java Reflection and Annotations: unit testing with arguments	6
3.1	The task	6
3.2	The pipeline	7
3.3	Reporting in declaration order	8
3.4	Verification	9
4	Haskell: multisets	9
4.1	Representation and invariant	9
4.2	Operations	9
4.3	Type-class instances	10
4.4	Testing with anagrams	11
4.5	Checking the laws exhaustively	11
5	Python: type-checking decorators	12
5.1	The task	12
5.2	Decorators and introspection	12
5.3	The three decorators	12
5.4	Verification	13
6	Verification summary	14
7	Conclusion	14

1 Introduction

Most programming courses teach how to *use* a language. Advanced Programming looks at the machinery behind it: how a component is discovered and connected at run time, how a program inspects its own classes, how a type system decides what an abstraction may promise, and how a dynamic language can be extended without changing its interpreter. The course follows these ideas across Java, C++, Haskell, Rust and Python, and its two programming assignments ask for four exercises, one for each idea:

1. **Java Beans** (Section 2): the Matching Pairs card game, built from JavaBeans that interact only through events, with bound and constrained properties.
2. **Java Reflection and Annotations** (Section 3): `RunTests`, a tool that loads a class, finds the methods annotated for testing and checks each of them against an annotated specification.
3. **Haskell** (Section 4): a multiset type with its operations and type-class instances, tested by counting anagram classes in text files.
4. **Python** (Section 5): three decorators that use the `inspect` module to check type hints at run time.

The four parts are independent and live in four folders of the same repository. Each folder holds three kinds of files: the *solution*, that is, exactly the files that the solution format of the exercise asks to deliver (21 files in total, including the five archives of the beans); the files *provided with the assignments*, included unchanged because the solutions need them to run or to be checked (for example, `RunTests` cannot be compiled without the class `Report`, and the decorators are exercised by the provided test programs); and the *documentation*, with a README in every folder that gives the role of each file. When the assignment provides an expected output, the output of the solution was compared with it byte by byte. Section 6 summarizes the verification, and Section 7 compares how the four languages approach programming at the meta level.

2 Java Beans: the Matching Pairs game

2.1 The task

In the Matching Pairs game (also known as *Concentration*) some cards lie face down and the player turns two of them face up at every turn; a pair of equal values is removed from play, two different values are turned face down again, and the round ends when every card has been removed. The assignment asks for a one-player version with eight cards, where every component is a JavaBean and the components interact according to the Publish–Subscribe (Observer) pattern, “using events and event listeners as much as possible”. The functional requirements fix the components:

- the **Board**, a `JFrame` that shows eight cards, a *Shuffle* and an *Exit* button, the **Controller** and the **Counter**;
- the **Card**, a `JButton` with a **value** and a **state** that must be *bound and constrained*: **excluded** (red), **face_down** (green) or **face_up** (showing the value); the state can change only through `setState`;
- the **Controller**, a label with the number of pairs found, which records the values v_1 and v_2 of the two cards of a turn and fires a **matched** event, delayed by half a second, whose value is $v_1 = v_2$; it is a `VetoableChangeListener` of every card and must forbid the changes triggered by clicking a face-up or excluded card;
- the **Counter**, a label with the number of cards turned face up, reset at every shuffle.

Two optional extensions make the number of cards parametric ($2N$ cards, changing N by recompiling the Board only) and add a **Challenge** label with the best score. Both are implemented.

2.2 The JavaBeans component model

A JavaBean is a reusable software component that a builder tool can discover and connect without knowing its source code [1]. The conventions are few: a public no-argument constructor, serializability, and *properties* exposed by a pair of methods `getX/setX`. Components communicate through *events*: an event object, which extends `EventObject`, is delivered to the listeners that registered through `addXListener`, and each listener implements an interface that extends `EventListener`. Two kinds of property have a predefined event:

- a **bound** property fires a `PropertyChangeEvent` *after* every change, so that other components can follow its value;
- a **constrained** property fires the same kind of event *before* the change, to vetoable change listeners, which can reject it by throwing a `PropertyVetoException`: if a listener vetoes, the change does not happen.

The `Introspector` recognizes all of this by naming patterns alone: a setter that declares a `PropertyVetoException` marks a constrained property, and a method that registers property change listeners marks the properties as bound. Swing components already implement both mechanisms in `JComponent` (`firePropertyChange`, `fireVetoableChange` and the corresponding registration methods), so the beans of the game use them directly. Overriding the registration of property change listeners with a private support object, as is sometimes done, would also capture the listeners that the look-and-feel itself registers on the button and silently disconnect them from Swing's own events.

2.3 Architecture

Table 1 lists the beans and Figure 1 shows how they are connected. The **Board** creates all the other beans at startup and registers each of them as a listener of the events it needs; after that, the beans never refer to each other except through events.

Table 1: The beans of the game. All of them have a public no-argument constructor and are serializable.

Bean	Extends	Properties	Listens to / publishes
Board	JFrame	—	publishes <code>ShuffleEvent</code>
Card	JButton	<code>state</code> (bound, constrained), <code>value</code> (read-only), <code>index</code>	listens to <code>ShuffleEvent</code> , <code>MatchedEvent</code> ; publishes changes of <code>state</code>
Controller	JLabel	<code>pairsFound</code> , <code>gameFinished</code> (bound), <code>totalPairs</code>	vetoes and follows <code>state</code> ; listens to <code>ShuffleEvent</code> ; publishes <code>MatchedEvent</code>
Counter	JLabel	<code>flipCount</code> (bound)	follows <code>state</code> ; listens to <code>ShuffleEvent</code>
Challenge	JLabel	<code>bestScore</code> (bound)	follows <code>flipCount</code> and <code>gameFinished</code>

2.4 The rules are enforced by vetoes

The heart of the design is the constrained property `state`. A card never decides by itself what a click means: the click only *asks* for a change, exactly as the specification describes it. A face-down card asks to turn face up, while a face-up or excluded card asks to turn face down:

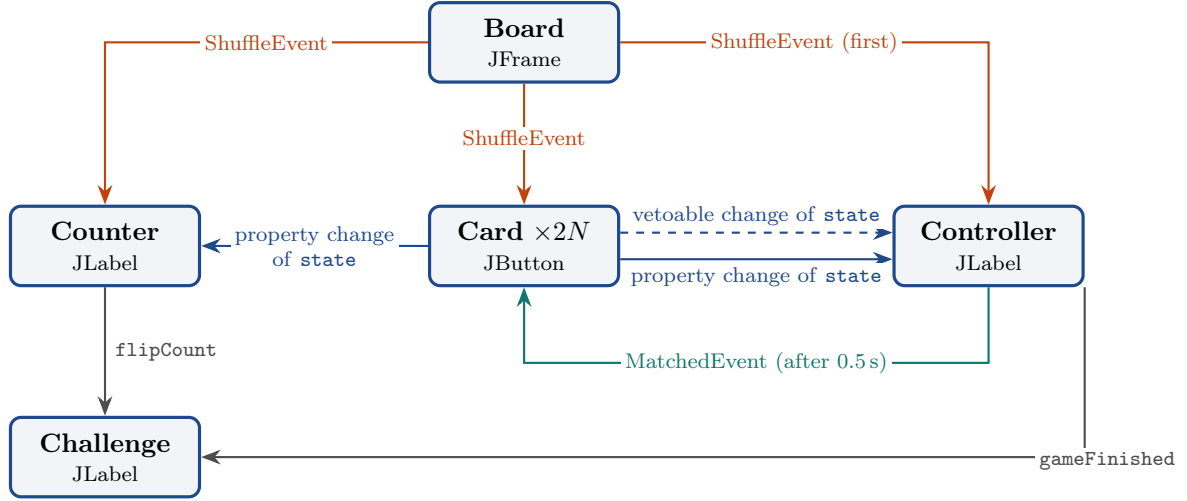


Figure 1: Event flow between the beans. Orange: the shuffle event published by the Board; blue: changes of the state of a card (the vetoable one is dashed); green: the matched event; black: bound properties followed by the Challenge.

```
private void clicked() {
    CardState requested = (state == CardState.FACE_DOWN) ? CardState.FACE_UP : CardState.FACE_DOWN;
    try {
        setState(requested);
    } catch (PropertyVetoException vetoed) {
        // The Controller rejected the move: the card keeps its current state.
    }
}

public void setState(CardState newState) throws PropertyVetoException {
    CardState oldState = this.state;
    if (newState == oldState) return;
    fireVetoableChange("state", oldState, newState); // constrained: listeners may veto
    this.state = newState;
    updateAppearance();
    firePropertyChange("state", oldState, newState); // bound: listeners are notified
}
```

Every change of state, whether caused by a click, by a matched event or by a shuffle, goes through `setState`, as the specification requires. The **Controller** decides with the three rules of Table 2, which depend only on its own state: the number of cards face up in the current turn, and two flags that are true while it delivers a matched event (**resolving**) or while a new round starts (**newRound**). Figure 2 shows the resulting state machine of a card. Its `vetoableChange` method has no side effects: when a veto occurs, `VetoableChangeSupport` notifies the listeners that had already accepted the change with a reverse event, and a listener with side effects would have to undo them.

2.5 A turn, step by step

The interplay of the beans during one turn shows why the Observer pattern fits the problem:

1. The player clicks a face-down card. The card asks for `face_up`; the **Controller** sees no card face up and accepts; the card shows its value and fires the property change.
2. The **Controller** receives the property change and records v_1 ; the **Counter** receives the same event and counts one move.

Table 2: The veto rules of the Controller.

Requested change	Allowed when	Typical vetoed request
face_down → face_up	fewer than two cards are face up in the current turn	a third card clicked during the half-second delay
face_up → excluded / face_down	the Controller is delivering the matched event	a click on a face-up card
excluded / face_up → face_down	a new round is starting after a shuffle	a click on an excluded card

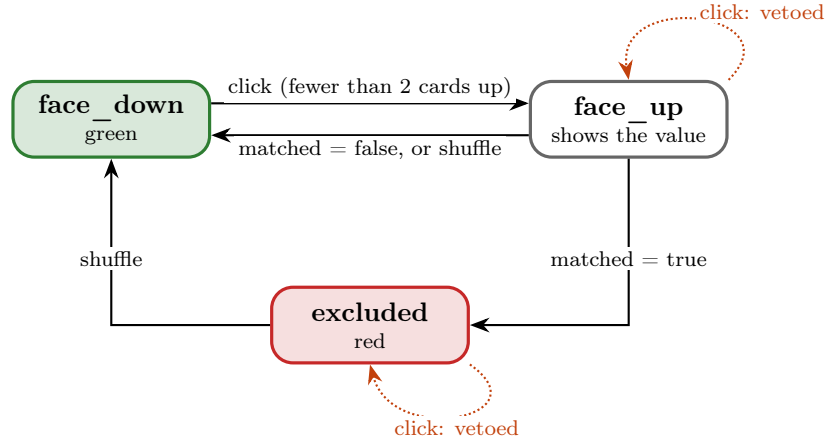


Figure 2: State machine of a card. Solid arrows are the changes that the Controller allows; dotted arrows are clicks whose request is vetoed, so the state does not change. When every card is excluded, only *Shuffle* and *Exit* react.

3. The player clicks a second card: the same sequence records v_2 , and the **Controller** starts a single-shot timer of 500 ms. Any click on a third card during the delay is vetoed, because two cards are already face up.
4. When the timer fires, the **Controller** compares v_1 and v_2 , updates its label, raises **resolving** and publishes the **MatchedEvent**. Every card receives it, and only the two face-up cards react, asking for **excluded** or **face_down**; the **Controller** accepts because **resolving** is true.
5. If the last pair was found, the **Controller** sets its bound property **gameFinished**; the **Challenge**, which has been following the **flipCount** of the **Counter**, compares the moves of the round with its best score.

2.6 One thread for everything

Swing components may be touched only from the Event Dispatch Thread (EDT) [3]. The delay before the matched event is therefore implemented with `javax.swing.Timer`, whose action runs on the EDT, rather than with `java.util.Timer`, whose task runs on a background thread. Since clicks, timers and shuffles all execute on the same thread, the beans share no state with other threads: there are no locks, no `volatile` fields and no calls to `invokeLater` to coordinate, and the order of the events is the order in which they happen.

2.7 New rounds

A shuffle must turn every card face down again, including excluded and face-up cards, whose change the rules would normally veto. The **Board** solves this with the order of notification,

which it controls because it registers the listeners: the **Controller** is the first subscriber of the shuffle event, so when the cards receive it the **Controller** has already reset its count, discarded any pending turn (a shuffle can arrive during the half-second delay) and opened the new round, during which a card may go back face down from any state. The **newRound** flag ends with the first card turned face up, when every card is necessarily face down.

2.8 Optional extensions and packaging

The number of pairs is a constant $N = 4$ of the **Board**, which creates $2N$ cards and passes N to the **Controller** at run time; since the other beans never read the constant, changing it requires recompiling the **Board** only, as was verified with $N = 6$. The **Challenge** label shows the smallest number of moves needed to complete a round since the start of the game, and it does so without ever calling another bean: it follows the bound properties **flipCount** and **gameFinished**.

The solution format requires the commented source files and one JAR archive per bean. Each archive contains its bean and declares it in the manifest with the standard per-entry attribute; dependencies are declared with **Class-Path**, and **Card.jar** also holds the types shared by all the beans (the card states, the two events and their listener interfaces). The archive of the **Board** is executable:

```
Manifest-Version: 1.0
Main-Class: Board
Class-Path: Card.jar Controller.jar Counter.jar Challenge.jar

Name: Board.class
Java-Bean: True
```

The course instructions add a practical detail: the solutions are sent by email to a Gmail address, and Gmail filters out **.jar** attachments, even inside compressed archives, so the archives must be renamed with the suffix **.raj** before sending them. The repository keeps them in this delivered form, in the folder **jars**. To use them, the five files are renamed back to **.jar** (all of them, since the **Class-Path** entries refer to the original names), and the game starts with **java -jar Board.jar**; from the sources, it starts with **java Board** after compiling. The **main** method selects the cross-platform look-and-feel, because some native look-and-feels, such as the default one on macOS, ignore the background color of a button: with it, the green and red cards would look identical.

2.9 Verification

During development, an automated test played the game through the real **Board** without showing it: it finds the beans in the component tree and clicks cards and buttons with **doClick()**, so every move follows the same path as a mouse click. Its 43 checks cover the JavaBeans conventions, the initial deal (each of the N values exactly twice), a turn with different values, a turn with a matching pair, clicks on face-up and excluded cards, direct calls of **setState** that must be vetoed, the end of a round, a shuffle, a shuffle during the half-second delay, and the best score over several rounds. Each archive was also loaded in a class loader that sees only that archive and its **Class-Path**, and the **Introspector** reports **Card.state** as bound and constrained. Finally, the **Board** was recompiled alone with $N = 6$ and the 43 checks were repeated on 12 cards: all of them passed.

3 Java Reflection and Annotations: unit testing with arguments

3.1 The task

RunTests takes the name of a class from the command line, loads it with the Reflection API, creates an instance, and invokes on it every non-private instance method annotated with **@Testable**,

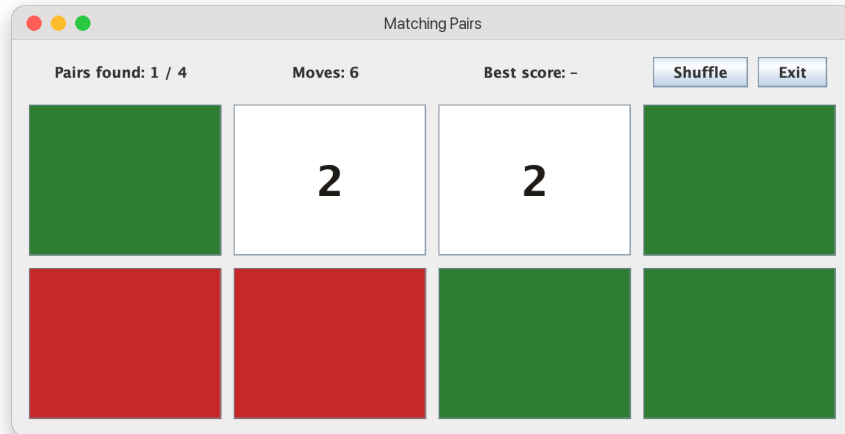


Figure 3: The game in the middle of a round: one pair has been found and excluded (red), two cards are face up during the half-second delay, the others are face down (green).

checking it against its `@Specification`. Both annotations are given by the assignment and kept at run time (`RetentionPolicy.RUNTIME`), which is what makes them visible to reflection:

```
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD)
public @interface Specification {
    String[] argTypes() default {}; // "int", "double", "bool" or "string"
    String[] argValues() default {}; // string representations, e.g. "42", "true"
    String resType() default ""; // "" means void
    String resVal() default "";
}
```

For each tested method `RunTests` must call `Report.report(...)`, a class provided with the assignment and used unchanged, exactly once, with one of four outcomes. The provided classes `MathOpsTests_simple` and `MathOpsTests`, with their expected outputs, define the required behavior.

3.2 The pipeline

Figure 4 shows the decisions taken for each method. The order of the checks matters: the specification defines `WrongResultType` only for methods that *could* be invoked with the given arguments, so the arguments are examined first; and in neither of the first two cases is the method invoked at all.

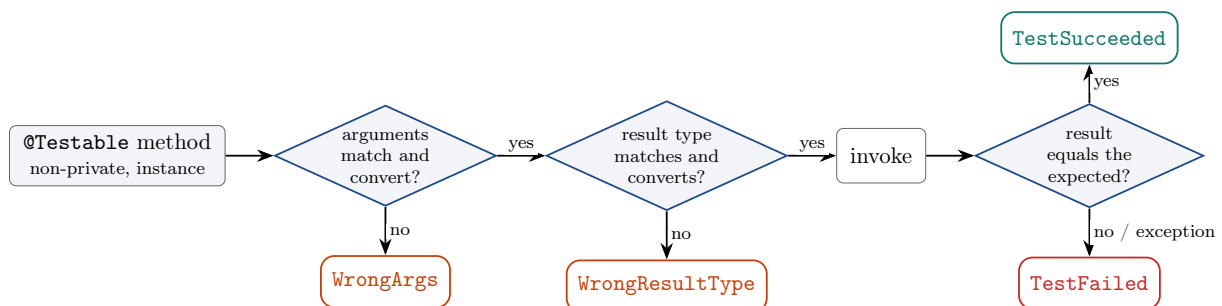


Figure 4: How `RunTests` classifies a method. The method is invoked only after both the arguments and the expected result have been validated.

Arguments. `argTypes` and `argValues` must have the same length as the parameter list; each

type name must be one of the four allowed and correspond to the parameter type; each value must convert to that type. Table 3 lists the correspondence, which accepts both primitive types and their wrappers. Booleans must be written exactly `"true"` or `"false"`: `Boolean.parseBoolean` would silently turn any other string into `false`.

Expected result. An empty `resType` requires a `void` method and an empty `resVal`; otherwise the type must correspond to the return type and `resVal` must convert to it (`"3a"` is not an `int`).

Invocation. The method is made accessible, so that protected and package-private methods are tested too, and invoked. An exception thrown by the method is reported as `TestFailed`: the method was invoked but did not return the expected result. The result is compared with `equals` on the boxed values.

Defaults. A method annotated with `@Testable` but not with `@Specification` is tested with a default specification (an anonymous implementation of the annotation interface whose attributes take their default values), so that `Report.report` still receives a specification object.

Table 3: Correspondence between the type names of a specification and the Java types.

Name	Java types	Conversion	Example
"int"	int, Integer	Integer.valueOf	"42" → 42
"double"	double, Double	Double.valueOf	"42" → 42.0
"bool"	boolean, Boolean	only "true" / "false"	"yes" → error
"string"	String	identity	"42" → "42"

3.3 Reporting in declaration order

The expected outputs list the tests in the order in which the methods are declared in the source, but `Class.getDeclaredMethods()` returns the methods “in no particular order” [4], and on a recent JVM the order indeed differs. Sorting by name would give a deterministic output, but not the expected one. The solution is to read the order from the bytecode: `javac` writes the method table of a class file in declaration order, and the class file format is fully specified [5]. `RunTests` opens the `.class` resource of the tested class and walks its structure (Figure 5):

magic 0xCAFEBAFE, minor and major version
constant pool: <code>count</code> , then 17 kinds of entries (the Utf8 strings are kept)
access flags, this class, super class
interfaces: <code>count</code> × 2 bytes (skipped)
fields: flags, name, descriptor, attributes (skipped)
methods: flags, name, descriptor, attributes ⇒ position i
class attributes (not read)

Figure 5: The parts of a class file read by `RunTests`. Entries of type Long and Double take two slots of the constant pool.

Each method of the table is identified by its name and its JVM descriptor, for example `testMin(II)I` for `int testMin(int, int)`; the same key is computed for every `Method` object from its parameter and return types, and the testable methods are sorted by their position in the table. The parser handles all 17 kinds of constant-pool entries defined by the JVM specification, and if the class file cannot be read (for example, a class generated at run time) it falls back to sorting by name, so the output remains deterministic.

3.4 Verification

`RunTests` was run on the two provided classes and, during development, on a third class written to exercise what the provided ones do not: a `@Testable` method without `@Specification`, protected and package-private methods, boxed parameter and result types, the string "42" read as a `double`, a method that throws an exception, malformed booleans and numbers, an unknown type name, a `void` method with an expected value, and static and private methods that must be skipped. Table 4 summarizes the outcomes; for the two provided classes the complete output is identical, byte by byte, to the expected one.

Table 4: Outcomes of `RunTests`. S: succeeded, F: failed, WA: wrong arguments, WR: wrong result type.

Class	Tested	S	F	WA	WR	Output
<code>MathOpsTests_simple</code> (provided)	7	5	2	0	0	identical to the expected one
<code>MathOpsTests</code> (provided)	11	5	2	2	2	identical to the expected one
edge cases (development)	10	4	1	3	2	as intended for each case

```
==> hello: Tests ready to run
=> Testing "hello" with args [] of type [] and result :...
=> "hello" tested with SUCCESS
...
=> Testing "fail1" with args [3.14] of type [double] and result 3.14:double...
=> "fail1" not tested: wrong argument list
=> Testing "fail3" with args [3.14] of type [double] and result 3:int...
=> "fail3" not tested: wrong result type
```

4 Haskell: multisets

4.1 Representation and invariant

A multiset, or bag, is a collection in which the order of the elements does not count but each element can occur several times. The assignment fixes the concrete representation, a list of pairs (*element*, *multiplicity*), and forbids library modules such as `Data.MultiSet`:

```
data MSet a = MS [(a, Int)]
    deriving (Show)
```

An `MSet` is **well-formed** if every multiplicity is positive and no element appears in two pairs. The representation does not enforce this invariant, so every operation that returns an `MSet` must preserve it. For example, the word *abracadabra* corresponds to `MS [('a',5),('b',2),('r',2),('c',1),('d',1)]`, while `MS [('a',3),('a',2)]` denotes the same collection of letters *a* but is not well-formed.

4.2 Operations

All the operations are written with Prelude functions only. A single helper, `addMany`, inserts $k > 0$ occurrences of an element: it increases the multiplicity of the element if present and appends a new pair otherwise, so it preserves the invariant by construction, and `add`, `union` and `mapMSet` are all defined through it.

```

addMany :: Eq a => MSet a -> a -> Int -> MSet a
addMany (MS xs) v k = MS (insert xs)
  where
    insert [] = [(v, k)]
    insert ((x, n) : rest)
      | x == v = (x, n + k) : rest
      | otherwise = (x, n) : insert rest

union :: Eq a => MSet a -> MSet a -> MSet a
union (MS xs) m2 = foldl (\acc (x, n) -> addMany acc x n) m2 xs

```

Table 5 lists the interface. As required, every definition is preceded by its type exactly as inferred by the compiler; this was checked by removing all the signatures and asking GHCi for the inferred types, which coincide up to the names of the type variables and the synonym `String = [Char]`. With d distinct elements, `add` and `occs` cost $O(d)$ and `union` costs $O(d_1 d_2)$: the price of a list representation, which cannot rely on an order or a hash of the elements, since only `Eq` is assumed.

Table 5: The interface of the module `MultiSet`.

Name	Type	Meaning
<code>empty</code>	<code>MSet a</code>	the empty multiset
<code>add</code>	<code>Eq a => MSet a -> a -> MSet a</code>	one more occurrence of an element
<code>occs</code>	<code>Eq a => MSet a -> a -> Int</code>	multiplicity of an element (0 if absent)
<code>elems</code>	<code>MSet a -> [a]</code>	all the elements, with repetitions
<code>subeq</code>	<code>Eq a => MSet a -> MSet a -> Bool</code>	inclusion, multiplicities included
<code>union</code>	<code>Eq a => MSet a -> MSet a -> MSet a</code>	sum of the multiplicities
<code>mapMSet</code>	<code>Eq b => (a -> b) -> MSet a -> MSet b</code>	map, merging equal images
<code>isWellFormed</code>	<code>Eq a => MSet a -> Bool</code>	the invariant, used by the tests

4.3 Type-class instances

Equality. Two multisets are equal if they contain the same elements with the same multiplicities, regardless of order. The instance normalizes both sides first (merging repeated elements and dropping non-positive multiplicities), then checks that the two lists of pairs have the same length and that every pair of one occurs in the other. Thanks to the normalization, the comparison is correct even for values built directly with the constructor `MS`, which may not be well-formed: `MS [(‘a’,1),(‘a’,1)] == MS [(‘a’,2)]`.

Foldable. The assignment asks that folding “apply the function to the elements of the multiset, ignoring the multiplicities”. A minimal complete definition of `Foldable` is either `foldMap` or `foldr` [7]; the instance defines `foldr`, visiting each pair once and passing only its element to the function:

```

instance Foldable MSet where
  foldr f z (MS xs) = foldr (\(x, _) acc -> f x acc) z xs

```

All the other methods of the class are derived from it. On `MS [(‘a’,3),(‘b’,1)]`, `length` returns 2 and `toList` returns `"ab"`, while `elems` returns `"aaab"`: the two views are deliberately different, one over the distinct elements, the other over every occurrence.

Why not a Functor. A function `mapMSet :: Eq b => (a -> b) -> MSet a -> MSet b` applies `f` to every element; when `f` maps two distinct elements to the same value, their multiplicities must be added, otherwise the result would not be well-formed. It cannot serve as `fmap`,

because the class fixes the type

$$\text{fmap} :: (a \rightarrow b) \rightarrow f a \rightarrow f b \quad \text{for every type } b,$$

with no constraint on b , whereas merging the images requires comparing values of type b , that is, the constraint `Eq b`. Dropping the merge is not an option: it would break the invariant of the type, since `fmap (const 0) (MS [(1,1),(2,1)])` would be `MS [(0,1),(0,1)]`. The same obstacle prevents `Data.Set` from being a `Functor` in the standard library: a structure whose invariant depends on comparing its elements needs a constrained map.

4.4 Testing with anagrams

The *ciao* of a string (*characters in alphabetical order*) is the string with the same characters, in lower case and sorted: the ciao of “Hello” is “ehllo”. Two words are anagrams exactly when they have the same ciao, so the multiset of the ciao of the words of a text counts its anagram classes. The file `TestMSet.hs` implements:

- `readMSet :: FilePath -> IO (MSet String)`, which reads a file and adds the ciao of each word, forcing the whole contents before returning so that the file is closed at once;
- `writeMSet :: MSet String -> FilePath -> IO ()`, which writes one line `<elem> - <multiplicity>` per element;
- `main`, which loads the four provided files into $m_1, \dots, m_4$, checks the two facts required by the assignment and writes m_1 and m_4 to `anag-out.txt` and `gana-out.txt`.

To decide whether two multisets have the same elements, the test uses `all` and `elem` on the multisets themselves: both come from the `Foldable` instance, which ignores the multiplicities, so `all (`elem` m2) m1 && all (`elem` m1) m2` expresses the property directly. Table 6 reports the results.

Table 6: The four multisets of ciao strings built by `TestMSet.hs`.

Multiset	File	Words	Distinct ciao
m_1	<code>anagram.txt</code>	870	245
m_2	<code>anagram-s1.txt</code>	450	208
m_3	<code>anagram-s2.txt</code>	420	198
m_4	<code>margana2.txt</code>	1,740	245

Both facts required by the assignment hold:

- $m_1 \neq m_4$, but they have the same elements: every multiplicity of m_4 is exactly twice that of m_1 ;
- $m_1 = m_2 \cup m_3$: the two halves of the word list add up to the whole, $870 = 450 + 420$ words.

The files `anag-out.txt` and `gana-out.txt` contain 245 lines each, one per anagram class.

4.5 Checking the laws exhaustively

The anagram test exercises the module on one data set. To check it in general, a separate program written during development verified 17 laws on *every* multiset built from the 121 lists over the alphabet $\{a, b, c\}$ of length 0 to 4, on all their 14,641 ordered pairs and, for associativity, on the 2,197 triples of lists of length up to 2 (Table 7). Small exhaustive domains of this kind catch the same mistakes as random testing (a missed merge, an off-by-one multiplicity, an order-dependent equality) without any test library. All the laws hold on all 76,494 cases.

Table 7: Laws verified exhaustively during development. $\#_x(\ell)$ is the number of occurrences of x in the list ℓ , and $M(\ell)$ the multiset obtained by adding the elements of ℓ one by one.

Law	Cases
$M(\ell)$ is well-formed; <code>occs</code> $M(\ell)$ $x = \#_x(\ell)$; <code>elems</code> returns every occurrence	3×121
$M(\ell) = M(\text{reverse } \ell)$; $M(\ell_1) = M(\ell_2) \iff \#_x(\ell_1) = \#_x(\ell_2)$ for all x	$121 + 14,641$
$\text{subeq } M(\ell_1) M(\ell_2) \iff \#_x(\ell_1) \leq \#_x(\ell_2)$ for all x	$14,641$
<code>union</code> is well-formed and adds the multiplicities; it is commutative	$2 \times 14,641$
<code>union</code> is associative; <code>empty</code> is its identity	$2,197 + 121$
each argument is <code>subeq</code> of the union	$14,641$
<code>Foldable</code> visits each distinct element once, in order; <code>sum</code> ignores multiplicities	2×121
<code>mapMSet</code> merges colliding images and preserves well-formedness	2×121
equality normalizes values built with <code>MS</code> ; a naive <code>fmap</code> result is ill-formed	$2 + 1$
Total: 17 laws	76,494

5 Python: type-checking decorators

5.1 The task

Python ignores type hints at run time: `def f(x: int)` accepts a string without complaint. The assignment frames the exercise as a small mystery: a lecturer who likes decorators wrote three of them to perform simple type checks with the `inspect` module, the file was lost, and only three test files with their logs survived. The task is to rewrite `print_types`, `type_check` and `bb_type_check` so that `python3 testN.py` produces “essentially the same output” as `logN.txt`. The solution reproduces the three logs exactly, character by character.

5.2 Decorators and introspection

A decorator is a function that receives a function and returns another one, usually a *wrapper* that adds behavior around the call [8]. The wrapper of these decorators needs the type hints of the decorated function, which `inspect.signature` exposes as a `Signature` object: an ordered mapping of `Parameters`, each with its `annotation`, plus a `return_annotation`. A parameter without a hint has the annotation `inspect.Parameter.empty`, whose string form is the `<class 'inspect._empty'>` that appears in the logs.

Figure 6 shows the structure shared by the three decorators. The signature is computed once, when the function is decorated, and captured by the closure of the wrapper; `functools.wraps` copies the name and the docstring of the original function onto the wrapper.

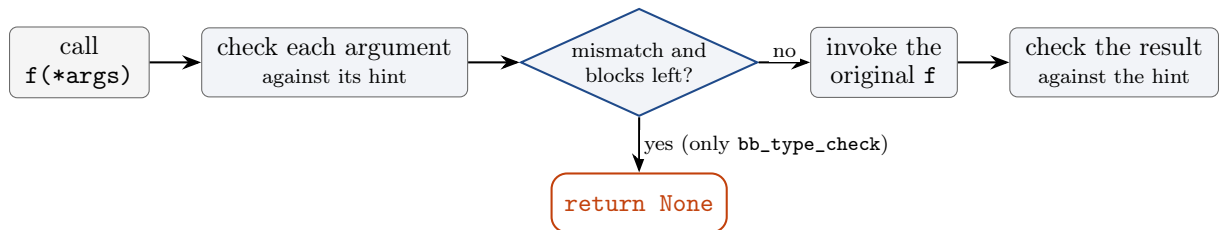


Figure 6: The wrapper of the three decorators. `print_types` prints every argument and the result; `type_check` prints only the mismatches; `bb_type_check` can also block the call.

5.3 The three decorators

`print_types` prints, for every argument, the name of the formal parameter with its hint and the actual value with its type, then the return hint and the actual result:

```

Formal par 'x':<class 'int'>; actual par '5':<class 'int'>
Formal par 'y':<class 'inspect._empty'>; actual par ' is ':<class 'str'>
Result type <class 'str'>; actual result '5 is similar to 5.0':<class 'str'>

```

type_check prints a message only for a disagreement, identifying the parameter by its position. A single helper decides whether a value agrees with a hint: a missing hint accepts anything; the hint `None` (as in `-> None`) stands for `type(None)`; union hints such as `int | str` are accepted by `isinstance` directly since Python 3.10 [10]; a hint that cannot be checked at run time, such as a string or a parameterized generic like `list[int]`, is skipped rather than raising an error.

```

def _matches(value, hint):
    if hint is _EMPTY:
        return True
    if hint is None:
        hint = type(None)
    try:
        return isinstance(value, hint)
    except TypeError:
        return True

```

bb_type_check (*bounded-blocking type check*) takes a parameter, so it is a decorator *factory*: `bb_type_check(2)` returns the decorator, which returns the wrapper. A call with at least one wrong argument is blocked (the function is not invoked and the call returns `None`) as long as blocks remain; afterwards the function runs anyway and its result is checked. The counter of remaining blocks belongs to the decorated function, so it lives in the closure of the decorator and is updated with `nonlocal`:

```

def bb_type_check(max_block):
    def decorator(func):
        signature = inspect.signature(func)
        params = list(signature.parameters.values())
        remaining_blocks = max_block

        @wraps(func)
        def wrapper(*args):
            nonlocal remaining_blocks
            if _check_arguments(params, args) and remaining_blocks > 0:
                remaining_blocks -= 1
                print(f"Function blocked. Remaining blocks: {remaining_blocks}")
                return None
            result = func(*args)
            _check_result(signature, result)
            return result
        return wrapper
    return decorator

```

Each decorated function gets its own counter. In `test3.py`, the function `add`, decorated with `bb_type_check(2)`, is blocked twice and then runs, while `add1`, decorated with `bb_type_check(0)`, is never blocked.

5.4 Verification

The three test programs are run and their output, preceded by the command line as in the logs, is compared with `log1.txt`, `log2.txt` and `log3.txt`: the 127 lines are identical. The interplay of the nested calls in `test2.py` is a good check of the design: in `add1(3, add1(2.0, 5))` the

inner call reports its own mismatch and result before the outer call checks its arguments, exactly as in the log.

6 Verification summary

Table 8 summarizes how each part was verified. Whenever the assignment provides an expected output, the output of the solution was compared with it byte by byte, with `diff`: “essentially the same output” is met in the strongest possible sense. Since the expected outputs, the test programs and their logs are included in the repository, these comparisons can be repeated by anyone with the commands given in the README.

Table 8: How the four parts were verified.

Part	Checks	Result
Java Beans	compilation with <code>-release 17</code> , with no warnings; 43 game rules through the real <code>Board</code> , with $N = 4$ and with $N = 6$ recompiling only the <code>Board</code> ; each archive loaded on its own	all passed
Reflection	<code>RunTests</code> on the two provided classes, outputs compared with the expected ones; ten further edge cases	2 / 2 identical
Haskell	compilation with <code>-Wall -Werror</code> ; the two facts of the anagram test; 17 laws on 76,494 cases	all passed
Python	three test programs, outputs compared with the three logs	3 / 3 identical

The sources target Java 17, use only the Prelude (plus `Data.Char` and `Data.List` in the test program) in Haskell, and only the standard library in Python, which must be version 3.10 or newer to accept union hints written `X | Y`.

7 Conclusion

The four exercises approach the same question, how a program can reason about its own structure, from four directions. JavaBeans rely on *conventions* that a tool can discover by introspection: naming patterns turn ordinary methods into properties, and a vetoable event turns a property into a place where rules are enforced. Java reflection goes one step further and treats classes and annotations as data, to the point that the order of the methods had to be recovered from the bytecode itself. Haskell takes the opposite route: what an abstraction may do is decided statically by its type, and the impossibility of a `Functor` instance for multisets is a property of types, not of code. Python, finally, has full access to its own functions at run time, and decorators use it to add a discipline, type checking, that the language does not impose.

On the engineering side, the project shows the value of making every claim checkable: each expected behavior of the assignments was encoded in a check and verified automatically, and two small defects that inspection alone would have missed (a label cut short at the end of a round, and a look-and-feel that hides the colors of the cards) were found by looking at the program actually running.

References

- [1] Sun Microsystems. *JavaBeans API Specification*, version 1.01, 1997.
- [2] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

- [3] Oracle. *Concurrency in Swing*. The Java Tutorials, <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/>.
- [4] Oracle. *Java SE 21 API Specification*: `java.lang.Class`, `java.lang.reflect`, `java.beans`.
- [5] T. Lindholm, F. Yellin, G. Bracha, A. Buckley and D. Smith. *The Java Virtual Machine Specification*, Java SE 21 Edition, Chapter 4: The class File Format. Oracle, 2023.
- [6] S. Marlow (editor). *Haskell 2010 Language Report*, 2010.
- [7] The `base` package: `Data.Foldable`. <https://hackage.haskell.org/package/base/docs/Data-Foldable.html>.
- [8] K. D. Smith, J. Jewett, S. Montanaro and A. Baxter. PEP 318: Decorators for Functions and Methods, 2003.
- [9] G. van Rossum, J. Lehtosalo and Ł. Langa. PEP 484: Type Hints, 2014.
- [10] P. Prados and M. Moss. PEP 604: Allow writing union types as `X | Y`, 2019.
- [11] Python Software Foundation. *The Python Standard Library*: `inspect`, Inspect live objects.