The Liskov Substitution Principle (LSP)

DEFINITION "If S is a declared subtype of T, objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T".

The LSP is all about expected behavior of objects.

One can only follow the LSP if one is clear about what the expected behavior of objects is.

Subtypes and Instances

SUBTYPES

For Java S is a ***Declared Subtype*** of T if
=> S is T,
=> S implements or extends T, or
=> S implements or extends a type that implements or extends T, and so on !

S is a ***Direct Subtype*** of T if
=> S extends or implements T.

INSTANCES

=> An object is a ***Direct Instance*** of a type T " if it is created by a "new T()" expression
=> An object is an ***Instance*** of T " if it is a direct instance of a declared subtype of T.

Expectations And Behavior

Behavioral specification

=> The behavioral specification of a class explains the "allowable behaviors" of the instances of a class.

=> *S is a behavioral subtype of T if " an instance of type S behaves only as allowed of type T objects.*

=>The ***LSP then says "declared subtypes should be behavioral subtypes***".

So where do these Expectations about behavior live?

=> In most language only a part of the expectations can be encoded in the language (for example types of parameters and results).

=> The rest of our expectations have to be expressed in the documentation (PreConditions and PostConditions).

They have to come (in part) from the documentation.
=> Such expectations can not come from the code, as method implementations may be abstract , even if not abstract, method implementations can be overridden

The Counterfeit Test (Different Way to Check Liskov Principle)

Here is a way to think about behavioral subtypes:
=> Suppose I promise to deliver you an object of class T, but instead I give you an object x of class S.
=>You can subject x to any series of method calls you like (chosen from T's signature).
=>If x behaves in a way that is not expected of a T object, then you know it is a counterfeit, x has failed the test.
=> If all S objects always pass every counterfeit test, then S is a ***Behavioral subtype*** of T.

Standard Requirements

Liskov's principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages.

Covariance of return types in the subtype.

```
class T { Object a()
 { … }
}

class S extends T {
@Override String a() { …
 }
}
```

This is allowed in Java. " More specific classes may have more specific return types " This is called "covariance.

Contravariance of method arguments in the subtype(JAVA doesn't support , could be incorporated using Wild Cards(generics) or Type Casting(equals method)).

A demonstration of Contra-variance ->

```
class T{
void c( String s ){
…}
 }
class S extends T{
@Override void c( Object s ){
 …}
 }
```

Logically it "could" be allowed for parameters to be "contra-variant"
However this is actually not allowed (in Java), as it would complicate the overloading rules.

Exception Rules

No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the super-type.

Every exception declared for the subtype's method should be a subtype of some exception declared for the super-type's method.

Every exception declared for the subtype's method should be a subtype of some exception declared for the super-type's method.

Additional Requirements.

In addition to the signature requirements, the subtype must meet a number of behavioral conditions. These are detailed in a terminology resembling that of design by contract methodology, leading to some restrictions on how contracts can interact with inheritance:

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the super-type must be preserved in a subtype.

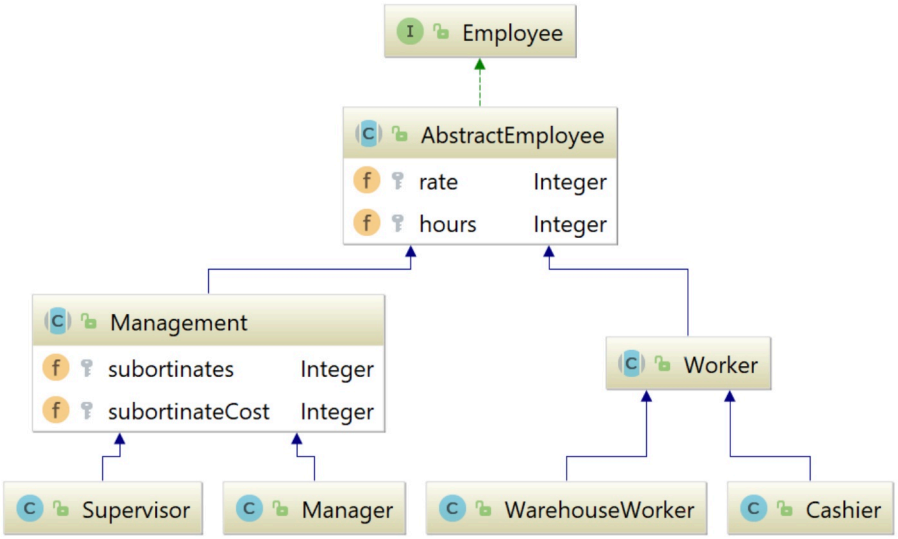
Documentation Rules of Liskov Principle

Any concrete method that may be overridden should be documented twice:
"The ***"Expected Behavior"*** documents what the client can expect from all instances (direct or indirect) of the class.

"The ***"Direct Behavior"*** documents what the client can expect from direct instance of the class ! or from instances of subclasses that do not override the method.

Abstract methods: " Only need Expected Behaviors.
Final methods (methods that can't be overridden) " Only need Direct Behaviors.
Otherwise if the Direct Behavior is undocumented,
It is considered equal to the Expected Behavior

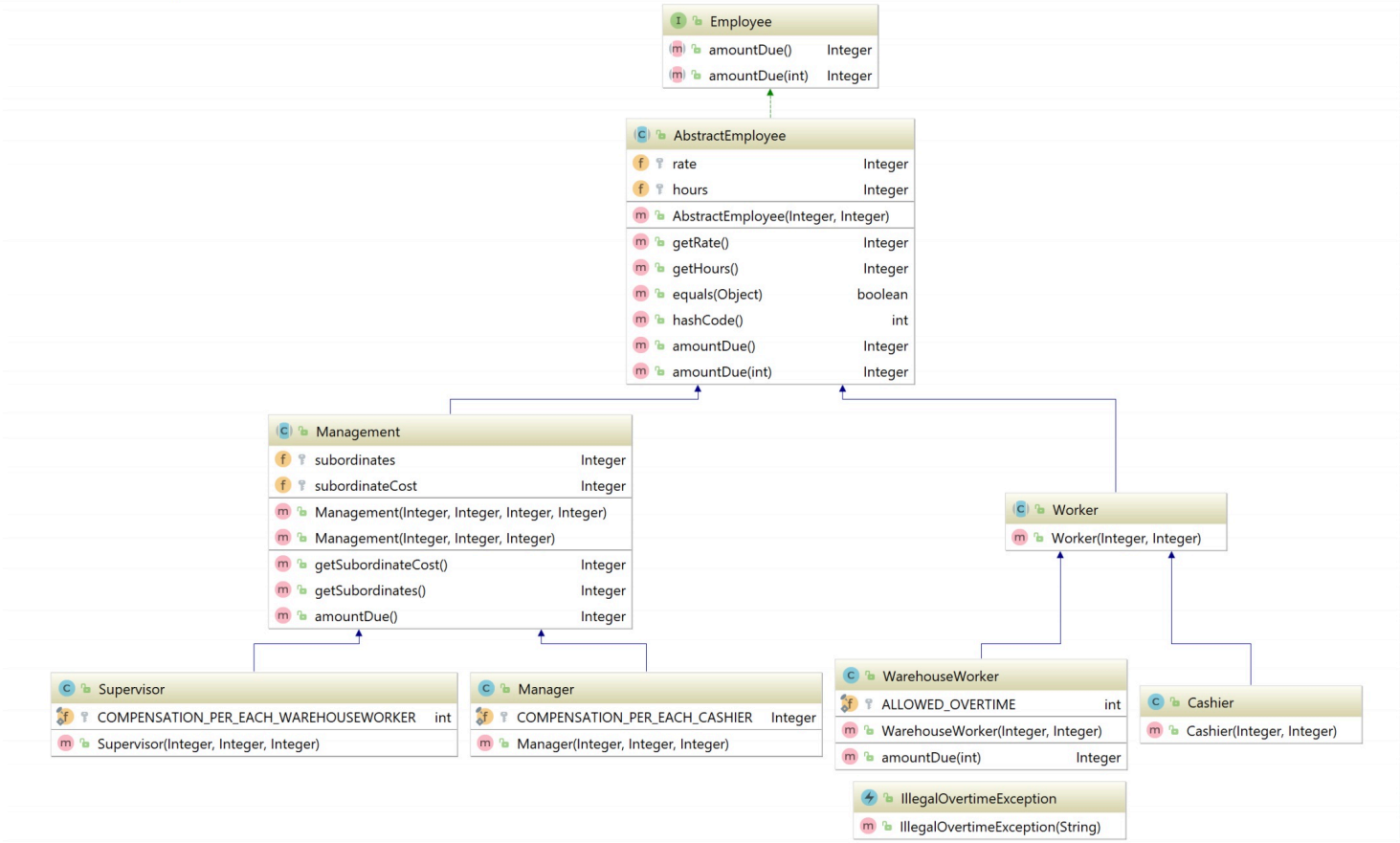
Design One
Problem a.



Liskov Substitution Principle holds for the proposed solution of the question 1. Supervisor, manager, warehouse worker and cashier are subclasses of Employee, the objects of Employee can be replaced with objects of 4 classes above since there is no desirable properties in Employee now.

Design Two

Problem c



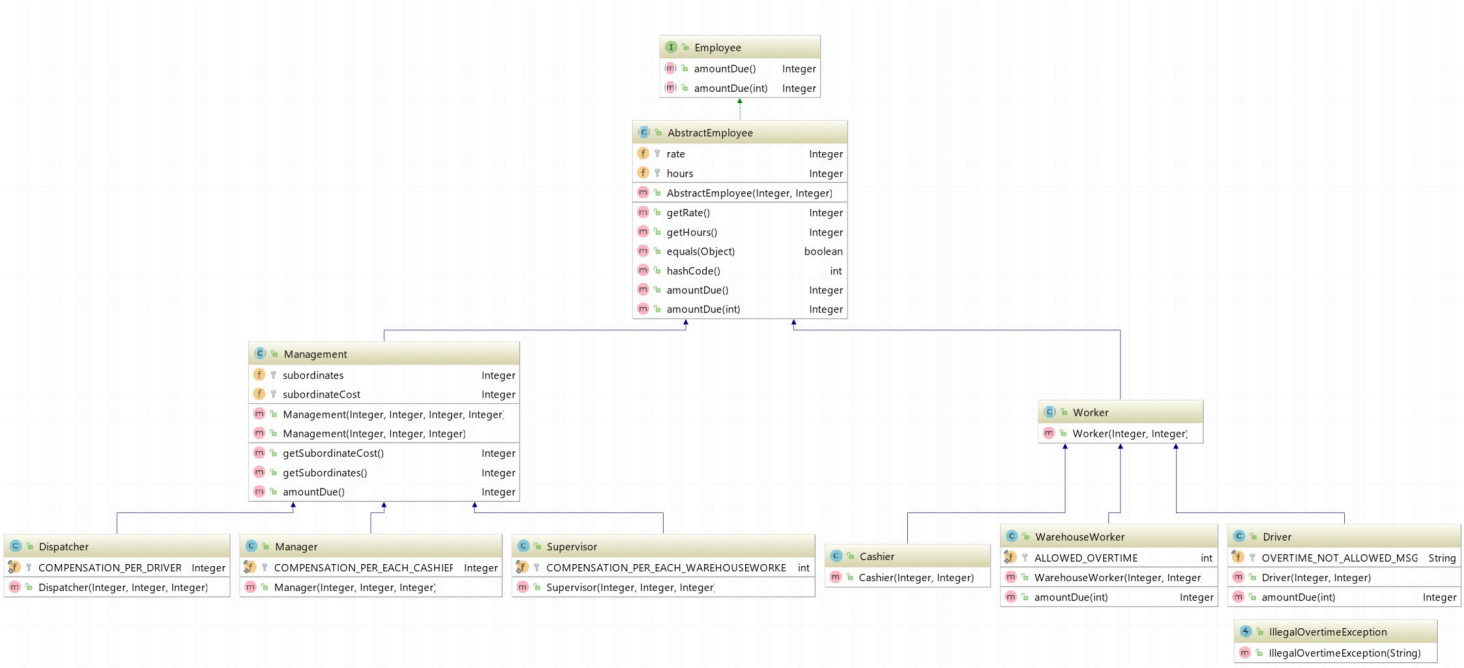
Liskov Substitution Principle does not hold for the proposed solution of the question c.
Let us focus on the amountDue(int) method in Employee and Warehouse.
The amountDue(int) method in Employee:

Precondition: int
Postcondition: Integer
The amountDue(int) method in Warehouse:
Precondition: int (not beyond ALLOWED_OVERTIME)
Postcondition: Integer

Since Warehouse worker will throw an exception while using amountDue(int) method when the int is larger than ALLOWERD_OVERTIME, which does not happen in its subclass. That means the precondition of Warehouse is larger than its subclass Employee, which violets Liskov Substitution Principle. The method amountDue(int) in the Employee should be removed. We should add an interface with amountDue(int) for warehouse worker to implement.

DESIGN THREE

Problem E



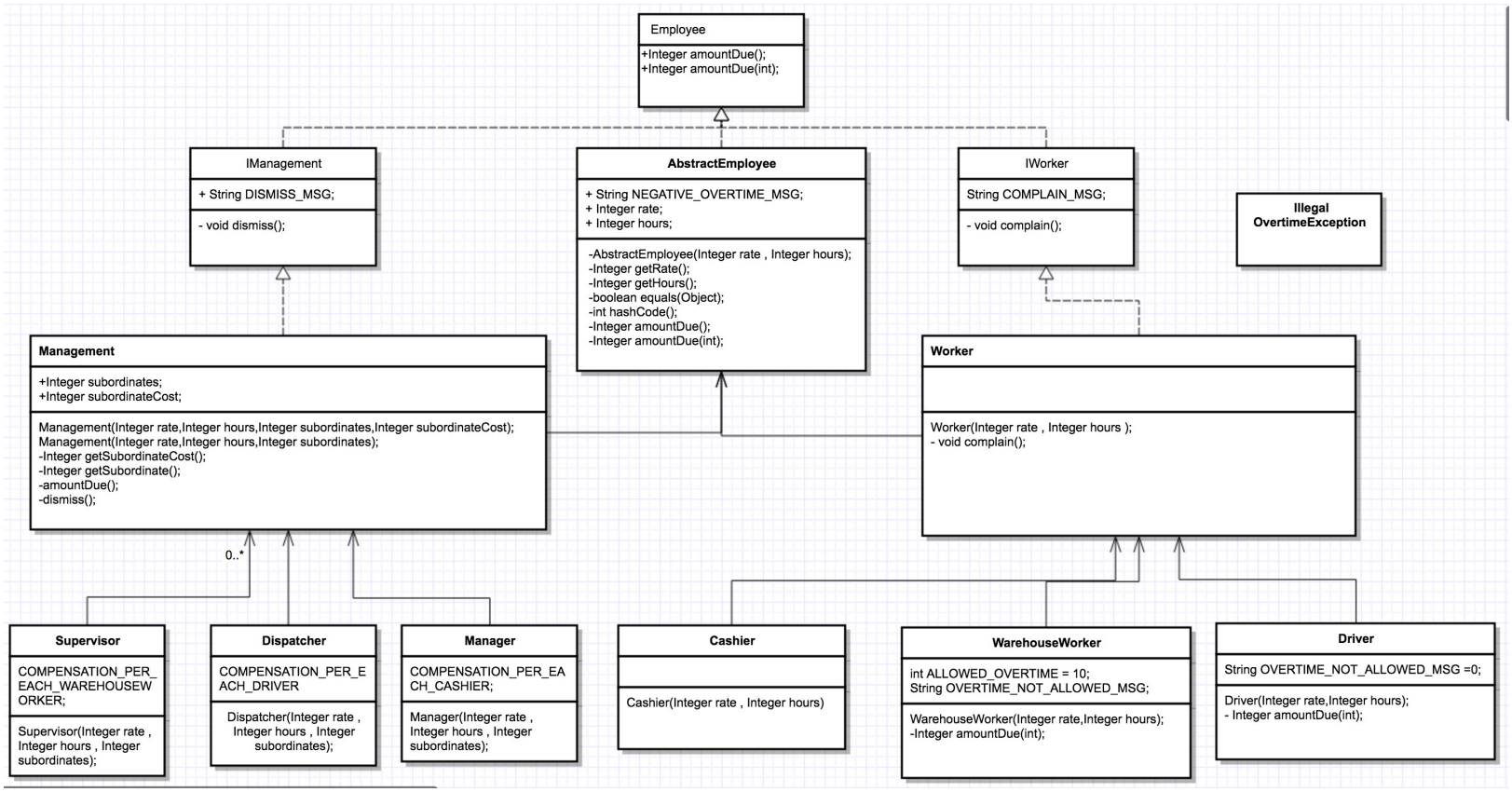
Liskov Substitution Principle does not hold for the proposed solution of the question e.
Let us focus on the amountDue(int) method in Employee and Driver.
The amountDue(int) method in Employee:

Precondition: int
Postcondition: Integer

The amountDue(int) method in Driver:
Precondition: int (actually any ints are not accepted)
Postcondition: Integer
Since Driver always throws an exception while using amountDue(int) method whatever the input is, the precondition of Driver is larger than its subclass Employee, which violets Liskov Substitution Principle. The method amountDue(int) in the Employee and Driver should be removed. We should add an interface with amountDue(int) for warehouse worker to implement.

Design Four

Problem f



Liskov Substitution Principle holds for the proposed solution of the question e.

First, let us focus on the dismiss() method in Dismiss and its subclasses. The dismiss() method in Dismiss:
Precondition: none
Postcondition: String

The dismiss() method in subclasses of Dismiss: Precondition: none
Postcondition: String

Secondly, let us focus on the complain() method in Complain and its subclasses. The complain() method in Complain:
Precondition: none
Postcondition: String

The complain() method in its subclasses: Precondition: none
Postcondition: String

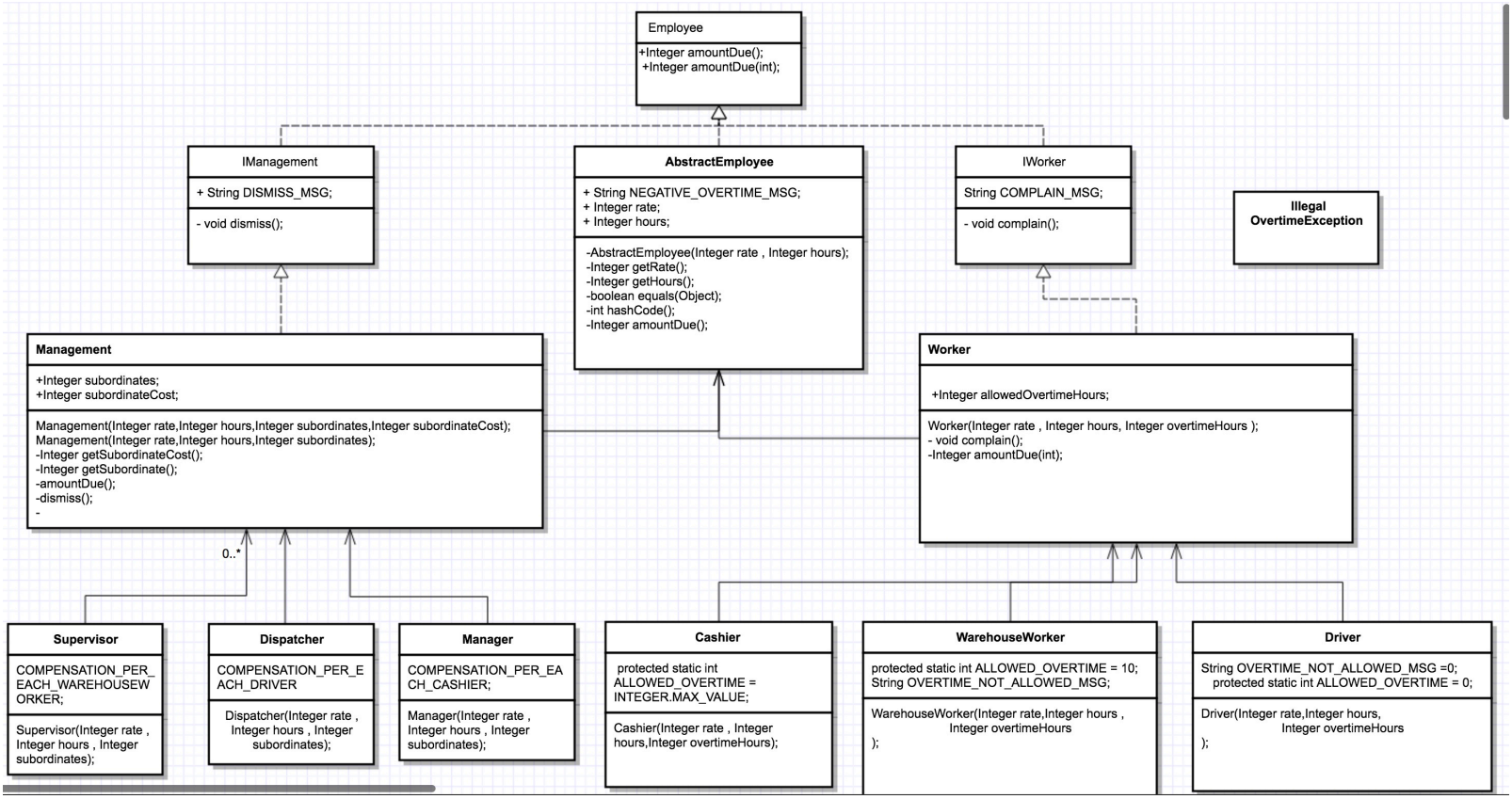
Since dispatcher, supervisor and manager all have the behavior dismiss, the

Dismiss interface is added correctly to let management implement so that Liskov Substitution Principle does not break.

Similarly, the interface complaint is added correctly to let worker implement so that Liskov Substitution Principle does not break, either.

Design Five

Proposed Design



Abstract class Worker class can have additional field allowedOvertimeHours (similar to subordinateCost in Management) that will be updated upon construction (that means the constructor of Worker will have an additional field, similar to Management) .

In such case, the implementation of amountDue(int overtimeHours) in Worker will be:

@Override public amountDue(int overtimeHours)
{ if (overtimeHours > allowedOvertimeHours) throw new
IllegalOvertimeException(ILLEGAL_OVERTIME_MSG);

return getRate() * overtimeHours + amountDue(); }
we will need to set constants in each concrete class of workers.

1. In Cashier: protected static int ALLOWED_OVERTIME = Integer.MAX_VALUE;
2. 2. In Warehouse: protected static int ALLOWED_OVERTIME = 10;
3. 3. In Driver: protected static int ALLOWED_OVERTIME = 0;
4. This design is definitely better, given the new restriction on Drivers and yields less code duplication.