Software Design and Algorithms



Introduction into Programming Basics. Building blocks of OOP, part 2.

EPAM Systems Inc.

Learn & Development

Software Design and Algorithms

[www.epam.com](http://www.epam.com/)

Contents

[Introduction into Programming Basics. Building blocks of OOP, part 2. 1](#_Toc65082532)

[1. Composition 3](#_Toc65082533)

[1.1. What is the Composition? 3](#_Toc65082534)

[1.2. composition vs inheritance 3](#_Toc65082535)

[1.3. moving from inheritance to composition 4](#_Toc65082536)

[1.3.1. 1st refactoring: Composing a bicycle of parts: Creating a Parts Hierarchy 4](#_Toc65082537)

[1.3.2. 2nd refactoring: Composing a bicycle of parts: Creating a Part 6](#_Toc65082538)

[1.3.3. 3rd refactoring: Composing a bicycle of parts: Creating a Part Factory 8](#_Toc65082539)

[1.4. Composition: Accepting the Consequences of Inheritance 8](#_Toc65082540)

[1.4.1. Benefits of Inheritance 8](#_Toc65082541)

[1.4.2. Costs of Inheritance 9](#_Toc65082542)

[1.5. Composition: Accepting the Consequences of Composition 9](#_Toc65082543)

[1.5.1. Benefits of Composition 9](#_Toc65082544)

[1.5.2. Costs of Composition 9](#_Toc65082545)

[2. Duck typing 10](#_Toc65082546)

[2.1. if it walks like a duck and talks like a duck then it’s a duck 10](#_Toc65082547)

[2.2. Finding the Duck 11](#_Toc65082548)

[2.3. Writing code that relies on ducks 14](#_Toc65082549)

[3. mixins 15](#_Toc65082550)

[3.1. Sharing role behavior with mixins 15](#_Toc65082551)

[3.2. understanding roles 15](#_Toc65082552)

[3.3. Writing the Concrete Code 16](#_Toc65082553)

[3.4. mixins: writing inheritable code 18](#_Toc65082554)

[4. design by contract 21](#_Toc65082555)

[4.1. correctness 21](#_Toc65082556)

[4.2. preconditions and postconditions for inheritance 24](#_Toc65082557)

[4.3. covariance and contravariance 26](#_Toc65082558)

# Composition

## What is the Composition?

Composition is one of the fundamental concepts in object-oriented programming. It describes a class that references one or more objects of other classes in instance variables. This allows you to model a ‘has-a’ association between objects. You can find such relationships quite regularly in the real world. A car, for example, has an engine and modern coffee machines often have an integrated grinder and a brewing unit. Each part separately may not give us significant value, a musical note, for instance, does not contain a lot of information, while using composition we can create musical composition which can give us much more information than each note separately.

## composition vs inheritance

Comparing composition with inheritance we can say that inheritance models strong ‘is-a’ relation between classes, it means that mountain bike is-a bicycle, while composition models weaker ‘has-a’ relation – mountain bike has-a wheel. Let us go back to the example from the previous lecture.

abstract class Bicycle {  
 protected readonly defaultChain = '11-speed';  
  
 constructor(opts) {  
 this.style = opts.style;  
 this.chain = opts.chain || this.defaultChain;  
 this.tireSize = opts.tireSize || this.defaultTireSize;  
 }  
  
 spares() {  
 return {  
 chain: this.chain,  
 tireSize: this.tireSize  
 };  
 }  
}

class RoadBike extends Bicycle {  
 protected readonly defaultTireSize = '28';  
  
 constructor(opts) {  
 super(opts);  
 this.tapeColor = opts.tapeColor;  
 }  
  
 spares() {  
 return {  
 ...super.spares(),  
 tapeColor: this.tapeColor  
 };  
 }  
  
 protected get defaultChain() {  
 return '2-speed';  
 }  
}

class MountainBike extends Bicycle {  
 protected readonly defaultTireSize = '29';  
  
 constructor(opts) {  
 super(opts);  
 this.frontShock = opts.frontShock;  
 }  
  
 spares() {  
 return {  
 ...super.spares(),  
 frontShock: this.frontShock  
 };  
 }  
}

Currently we have abstract Bicycle class and two subclasses – RoadBike and MountainBike. If you need continuously extend existing functionality, you will notice that spare parts functionality may not change all the time or will require adjustments because of specific implementations of different subclasses. It becomes too difficult to extend spare parts functionality and we will use composition to solve this problem.

## moving from inheritance to composition

### 1st refactoring: Composing a bicycle of parts: Creating a Parts Hierarchy

Let us refactor current functionality, we will move spare parts to Parts class so inheritance will be replaced with composition. You can use UML diagram from Figure 1 to better understand what we will get after refactoring.

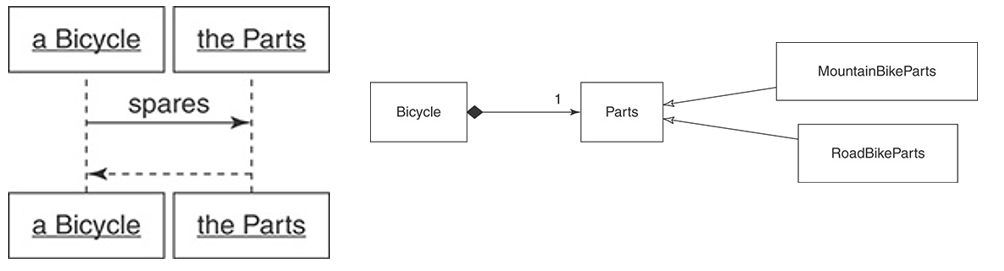


Figure 1. Bicycle asks Part for spares and Bicycle has a Parts

abstract class Parts {  
 protected chain: string;  
 protected tireSize: string;  
  
 constructor(opts) {  
 this.chain = opts.chain || this.defaultChain;  
 this.tireSize = opts.tireSize || this.defaultTireSize;  
  
 this.postInitialize(opts);  
 }  
  
 spares() {  
 return {  
 chain: this.chain,  
 tireSize: this.tireSize,  
 ...this.localSpares  
 };  
 }  
  
 protected abstract get localSpares();  
 protected abstract get defaultTireSize();  
 protected get defaultChain() {  
 return '11-speed';  
 }  
 protected postInitialize(opts) {};  
}

Now we have separate Parts class with all the spares functionality incapsulated inside it. With new approach Bicycle type depends on provided parts, when you call spares method inside Bicycle it is delegated to Parts instance and it decides which exact parts to return.

class RoadBikeParts extends Parts {  
 private tapeColor: string;  
  
 postInitialize(opts) {  
 this.tapeColor = opts.tapeColor;  
 }  
  
 protected get localSpares() {  
 return {  
 tapeColor: this.tapeColor  
 };  
 }  
  
 protected get defaultTireSize() {  
 return '28';  
 }  
}

class MountainBikeParts extends Parts {  
 private frontShock: string;  
  
 constructor(opts) {  
 super(opts);  
 this.frontShock = opts.frontShock;  
 }  
  
 protected get localSpares() {  
 return {  
 frontShock: this.frontShock  
 };  
 }  
 protected get defaultTireSize() {  
 return '29';  
 }  
}

After refactoring you need to pass bicycle size and Parts instance to create a new Bicycle, you can see it on example below:

const roadBike = new Bicycle(  
 {  
 size: 'M'  
 },  
 new RoadBikeParts({tapeColor: 'red'})  
);

const mountainBike = new Bicycle(  
 {  
 size: 'M'  
 },  
 new MountainBikeParts({frontShock: 'manitou'})  
);

Now Bicycle class is only responsible for its size and for which parts it can consume.

### 2nd refactoring: Composing a bicycle of parts: Creating a Part

But it is not a result, let us continue the refactoring and adjust Parts class to make it looks like typed collection. After refactoring our classes structure will look like on UML-diagram below:

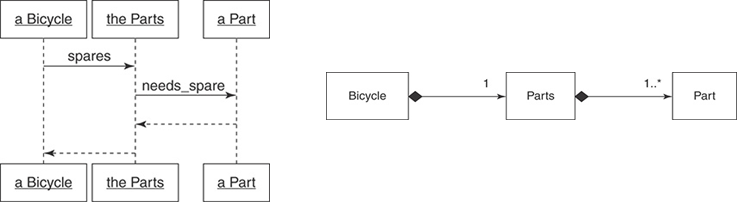


Figure 2. Bicycle, Parts and Part relations

class Bicycle {  
 constructor(private size: string, private parts: Parts) {}  
  
 spares() {  
 return this.parts.spares();  
 }  
}

class Parts {  
 constructor(private parts: Part[]) {}  
  
 spares() {  
 return this.parts  
 .filter(({needsSpare}) => needsSpare)  
 .reduce((spares, {name, value}) => ({  
 ...spares,  
 [name]: value  
 }));  
 }  
}

class Part {  
 constructor(  
 public name: string,  
 public value: string,  
 public needsSpare = true  
 ) {}  
}

Now we have three main classes: Bicycle class is the same as before the refactoring, Parts class composes separate Part instances, on diagram it is showed as ‘one-to-many’ relation between Parts and Part classes. With new approach, theoretically, we can just skip Parts class and just put typed collection directly inside the Bicycle class, but we have more than just a collection, we have some additional logic which is related to Parts functionality. Each Part instance has needs\_spare property which indicates if we need to take a spare for this instance of Part. If we will take a closer look on the MountainBike, we will see that it may have rear shock but we do not need a spare for it. This is logic which Parts class contains, without this part of functionality, we may choose the option described above. Below you can see an example how to use new approach.

const roadBike = new Bicycle(  
 'M',  
 new Parts([  
 new Part('chain', '11-speed'),  
 new Part('tireSize', '28'),  
 new Part('tapeColor', 'red')  
 ])  
);

const mountainBike = new Bicycle(  
 'L',  
 new Parts([  
 new Part('chain', '11-speed'),  
 new Part('tireSize', '29'),  
 new Part('readShock', 'fox', false),  
 new Part('frontShock', 'manitou')  
 ])  
);

### 3rd refactoring: Composing a bicycle of parts: Creating a Part Factory

The next step will help us to further adjust Parts creation, as we can use Factory pattern to unify parts creation. On the one hand, this will help us to simplify new Bicycle instances creation, but on the other hand, we need to understand that if some functionality will be changed or extended, this may require us to review our abstraction or design pattern. In the current case, Factory can incapsulate all the Parts creation inside, so it will look like on example below:

const roadConfig = new Set([  
 ['chain', '11-speed'],  
 ['tireSize', '28'],  
 ['tapeColor', 'red']  
]);

const mountainConfig = new Set([  
 ['chain', '11-speed'],  
 ['tireSize', '29'],  
 ['readShock', 'fox', true],  
 ['frontShock', 'manitou']  
]);

PartsFactory.build(roadConfig);  
PartsFactory.build(mountainConfig);

## Composition: Accepting the Consequences of Inheritance

“Inheritance is specialization.” — Bertrand Meyer, Touch of Class: Learning to Program Well with Objects and Contracts.

“Inheritance is best suited to adding functionally to existing classes when you will use most of the old code and add relatively small amounts of new code.” —Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software

### Benefits of Inheritance

* Use of inheritance results in code that can be described as open-closed; hierarchies are open for extension while remaining closed for modification.
* Correctly written hierarchies are easy to extend. The hierarchy embodies the abstraction and every new subclass plugs in a few concrete differences. The existing pattern is easy to follow and replicating. Hierarchies by their nature provide guidance for writing the code to extend them.
* Use Inheritance for is-a Relationships

### Costs of Inheritance

* Choosing inheritance to solve the wrong kind of problem.
* High cost of making changes near the top of an incorrectly modeled hierarchy. In this case, the leveraging effect works to your disadvantage; small changes break everything.
* Impossibility of adding behavior when new subclasses represent a mixture of types.

## Composition: Accepting the Consequences of Composition

* “Use composition when the behavior is more than the sum of its parts.” —paraphrase of Grady Booch, Object-Oriented Analysis and Design

### Benefits of Composition

* When using composition, the natural tendency is to create many small objects that contain straightforward responsibilities that are accessible through clearly defined interfaces.
* These small objects have a single responsibility and specify their own behavior. They are transparent; it is easy to understand the code and it is clear what will happen if it changes.
* Because composed objects deal with their parts via an interface, adding a new kind of part is a simple matter of plugging in a new object that honors the interface.
* Use Composition for has-a Relationships.

### Costs of Composition

* A composed object relies on its many parts. Even if each part is small and easily understood, the combined operation of the whole may be less than obvious.
* The benefits of structural independence are gained at the cost of automatic message delegation. The composed object must explicitly know which messages to delegate and to whom.
* As these costs and benefits illustrate, composition is excellent at prescribing rules for assembling an object made of parts but does not provide as much help for the problem of arranging code for a collection of parts that are very nearly identical.

# Duck typing

## if it walks like a duck and talks like a duck then it’s a duck

To better understand the meaning of this phrase let us analyze it using one of previous examples based on the UML-diagram below:

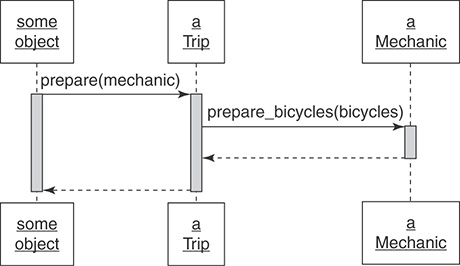


Figure 3. Trip – Mechanic interaction

In this example, Trip class earlier had to do few calls of Mechanic class method, now those calls are combined into single prepare\_bicycles method call, as you can see on Figure 3. But if we will need to prepare something besides bicycles, then our code may look like on example below:

class Mechanic {}  
class TripCoordinator {}  
class Driver {}  
  
class Trip {  
 bicycles;  
 customers;  
 vehicle;  
   
 prepare(prepares: object[]) {  
 return prepares.map((preparer) => {  
 switch (preparer.constructor) {  
 case Mechanic:  
 return preparer.prepareBicycles(this.bicycles);  
 case TripCoordinator:  
 return preparer.buyFood(this.customers);  
 case Driver:  
 preparer.fillTank(this.vehicle);  
 return preparer.fillWaterTank(this.vehicle);  
 }  
 });  
 }  
}

When introducing new preparers in addition to Mechanic, like TripCoordinator and Driver you will notice how dramatically increased the number of dependencies in prepare method. Now it knows every class name, classes` methods` names and their arguments because it needs to prepare some specific things before the trip. What is even worse, is the fact that this type of code only will increase its size and dependencies number with time, it is the easiest way for developer to add another switch case to already existing cases.

## Finding the Duck

We have identified the problem which we need to solve, and now we need to minimize dependencies number to make Trip functionality easily extensible without usage of switch-case operator and other similar approaches. Analyzing existing functionality, we may notice something common between all the preparers, something, that each of them does, but at the same time what they are not. To understand what the instance is we are talking about, let us look on the UML-diagram on Figure 4:

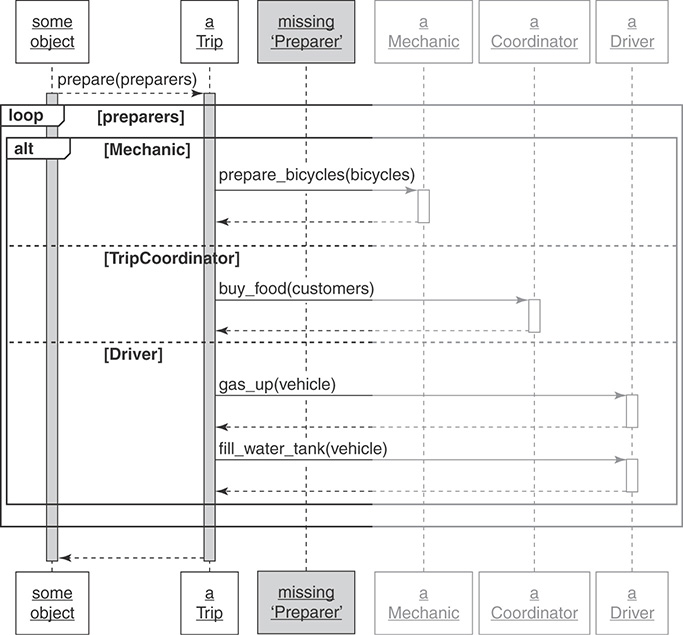


Figure 4. Missing Preparer type

Every preparer class is responsible for preparing something for the Trip, so we can try to extract some Preparer abstraction and call it a duck type. As a result we have something similar to an interface, but actually it is just a role which can be applied to some specific class in some specific moment of time, and we cannot say that every Preparer class is a part of some types hierarchy. This is the exact moment when we can extract our duck types, the next step is to review changes in code, which is required to extract the duck type. Our refactoring will be based on the UML-diagram from Figure 5:

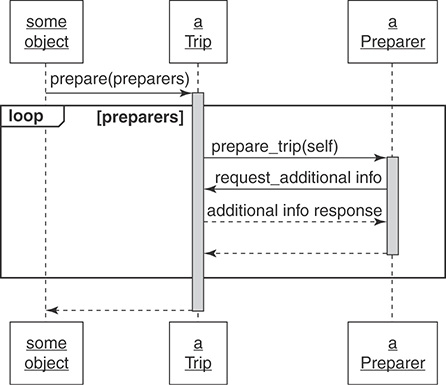


Figure 5. Trip – Preparer interaction

When we will finish the refactoring, every Preparer will have prepareTrip method which takes Trip instance as an argument so every preparer can take needed data from the instance. Below you can see the refactoring result:

class Trip {  
 prepare(prepares: *Array*<{ prepareTrip(trip: Trip) }>) {  
 prepares.map((preparer) => {  
 preparer.*prepareTrip*(this);  
 });  
 }  
}

class Driver {}  
class Mechanic {  
 prepareTrip(trip: Trip) {  
 trip.bicycles.map((bicycle) => {  
 this.prepareBicycle(bicycle);  
 });  
 }  
}

class TripCoordinator {  
 prepareTrip(trip: Trip) {  
 this.buyFood(trip);  
 }  
}

Trip class changed the most, we have removed all the dependencies on specific implementations of other classes, now every Preparer only need to have prepareTrip method so Trip class will not change anymore with addition of new preparers.

## Writing code that relies on ducks

To sum up the information about duck types we will try to make a list of main points which helps us to write a code using duck types.

1. Recognizing Hidden Ducks. You need to timely understand where the duck types are hidden and how to extract them, pay attention to the next places in the code:
   1. Case statements that switch on class.
   2. `Instanceof` operator.
   3. Checking the method exists (if (obj.someMethod) { obj.someMethod() } );
2. Placing Trust in Your Ducks. Let client code trust the duck type, in lack of the trust client code means the next: “I know who you are, so I know what you do”. Such knowledge transforms into tight coupling between classes which results into non extensible code. Flexible applications built on top of objects which works on trust – your goal as a developer is to make those objects reliable, to let the trust work.
3. Documenting Duck Types. Preparer duck type and its open interface is a specific part of the design, but at the same time it is a virtual part of code, because it is neither a class nor a real interface. Preparers are an abstraction, just a convention which gives you the powerful system design tool, but this abstraction makes code less obvious. When you create a duck type, you must document and cover it with tests.
4. Sharing Code between Ducks. In our example shared is only prepareTrip method, but when you start using duck types, you may notice that some part of the functionality is common for all the types. Share such functionality using mixins and other available approaches.
5. Choosing Your Ducks Wisely. The last point, as always, tells us that you do not need to create duck types just to have them. You need to find a balance between resources required for the refactoring, benefit gained, support simplicity and code clarity.

# mixins

## Sharing role behavior with mixins

Mixins is an another OOD tool, inheritance is not the only way to share a behavior, every problem which we solve using inheritance also can be solved using other tools, and one of such tools are mixins. But each tool has its pros and cons so let us look on what mixins can give us and what the price we must pay to use them.

## understanding roles

Sometimes you need to share some behavior between non-related objects, such case is a direct opposite to inheritance hierarchy (‘is-a’ relation). It is rather a role, which object can play on some state of its lifecycle. Let us review some basic example below:

const sayMixin = {  
 say(phrase) {  
 *alert*(phrase);  
 }  
};

const sayHiMixin = {  
 \_\_proto\_\_: sayMixin,  
 sayHi() {  
 super.say(`Hello ${this.name}`);  
 },  
 sayBye() {  
 super.say(`Bye ${this.name}`);  
 }  
};

class User {  
 constructor(name) {  
 this.name = name;  
 }  
}

We have sayMixin with say method which prints given phrase on the screen, we also have its extended version – sayHiMixin. When you need to use mixin functionality on some object, you can just add it to the object’s prototype, and this will give us the possibility to use mixins methods on this object.

Object.assign(  
 User.prototype,  
 sayHiMixin  
);  
  
new User('Dave').sayHi();

But such approach has some consequences, prototypes functionality is not the most convenient in usage, apart from that, Object.assign only makes a shallow copy so you can use existing JS libraries to make mixins usage easier.

## Writing the Concrete Code

To better understand how mixins work and what they can give us we will go back to bicycles example. Let us look on the case when we need to make our bicycles schedulable, this functionality will give as an opportunity to schedule a single bicycle on a specific period. We also need to add leadDays property which will store number of days required to prepare the bicycle.

class Schedule {  
 isScheduled(schedulable, starting, ending) {  
 console.log(`Checking if ${schedulable.constructor.name}`  
 + `is available on ${starting} - ${ending}`);  
  
 //do the checks  
  
 return true;  
 }  
}

class Bicycle {  
 leadDays = 1;  
  
 constructor(parts, schedule = new Schedule()) {  
 this.schedule = schedule;  
 // ...  
 }  
  
 isSchedulable(starting, ending) {  
 const withLeadTime = starting - this.leadDays;  
  
 return this.schedule.isScheduled(this, withLeadTime, ending);  
 }  
}

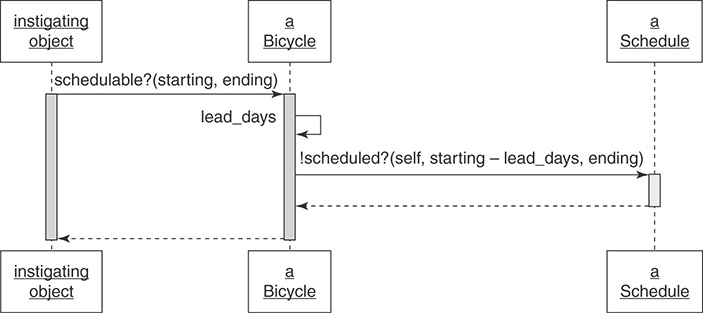


Figure 6. Bicycle classes know if they are schedulable

Based on the result above and UML-diagram from Figure 6 we can say that major part of logic is stored in Schedule class, this class is used in isSchedulable method, it needs two dates, one for the start and one for the end of booking, Schedule instance receives start date with subtracted leadDays so we will always have enough time to prepare the bicycle for a trip. Now we can schedule a bicycle, but we also have other classes like Mechanic and Driver and each of them has its own leadDays value. Extracting common logic, we can create Schedulable duck type.

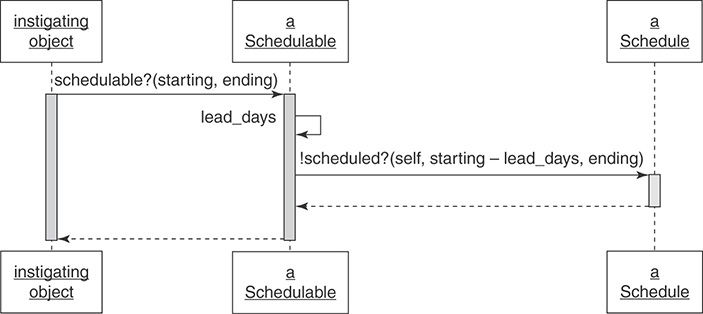


Figure 7. Schedulable duck type

Let us review UML-diagram from Figure 7, we have new Schedulable instance, but relation between Bicycle and Schedulable is not ‘is-a’, because Bicycle should not be schedulable under normal conditions. This duck type rather describes Bicycle specific behavior when it is required to be scheduled for the trip, or under specific conditions in other words. Other parts of our system should not even know that Bicycle is schedulable, these classes relations will be better to describe as ‘behaves-as’.

const *SchedulableMixin* = (superclass) => class extends superclass {  
 private schedule: Schedule;  
 protected leadDays = 0;  
   
 set schedule(schedule) {  
 this.schedule = schedule;  
 }  
   
 get schedule(schedule) {  
 return this.schedule || new Schedule();  
 }  
   
 isSchedulable(starting, ending) {  
 const withLeadDays = starting - this.leadDays;  
   
 return this.schedule.isScheduled(this, withLeadDays, ending);  
 }  
}

class Bicycle extends *SchedulableMixin*(Object) {  
 protected leadDays = 1;  
}

class Vehicle extends *SchedulableMixin*(Object) {  
 protected leadDays = 3;  
}

class Mechanic extends *SchedulableMixin*(Object) {  
 protected leadDays = 4;  
}

We have extracted common logic to SchedulableMixin with isSchedulable method and now we can easily mix it to any class in our hierarchy, either Bicycle, Mechanic or Driver so we will have schedulable behavior when we only need it. In this case decorators could be used as an alternative solution, you may know about them from TypeScript and they solve this problem in as similar manner, so mixins are not the only solution to add a behavior. The only consequence in Schedulable mixin is that we need to store leadDays property in a target class so it can be used in mixin.

## mixins: writing inheritable code

To better understand which consequences mixins have in general let us review the scheme from Figure 8.

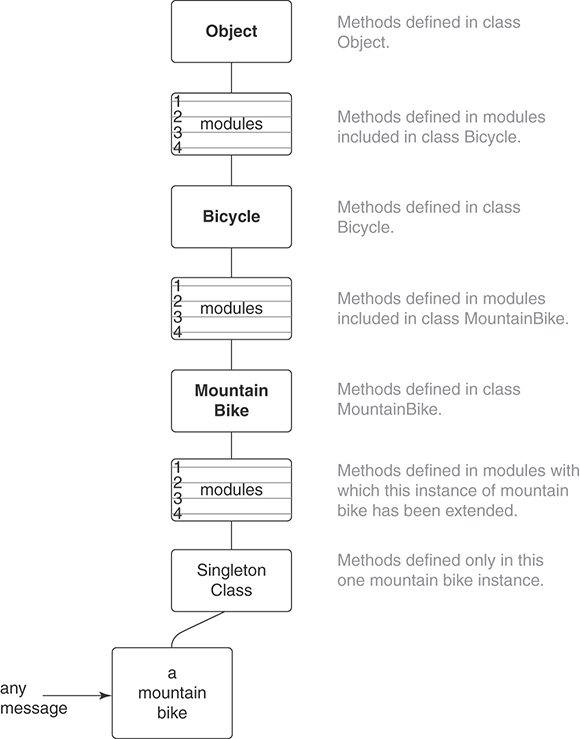


Figure 8. Mixins call stack

Taking a closer look at this scheme we can notice that mixins added additional levels to the call stack, this makes understanding and debugging of the application less obvious so you need to keep this in mind and use mixins only when they are really needed.

* Recognize the Antipatterns. There are two antipatterns which may indicate that you can gain a benefit from inheritance. First of them is using variables with a type/category to determine a type of object and send it a message. The second is usage of direct object type checking or switch-case operator – then you rather missed a duck type. Duck types may have not only common interface but also common behavior, which is recommended to extract to mixins.
* Insist on the Abstraction. All the code in an abstract superclass which should be used in every subclass, superclasses should not contain a code which is only applied for some subclasses. This limitation is also applicable to mixins, all the mixin functionality should be used in every place where it is mixed in. If you cannot identify an abstraction, then probably it is not existing, and inheritance cannot be applied to solve this problem.
* Honor the Contract. Subclasses must honor the contact so they can be easily replaced with superclasses without any change in a system behavior. This means that they need to answer on the same messages receiving the same input data and returning the same result data. Thereby they can not do something which will force a client code to check them for a type to understand what to wait from them. Subclasses which do not honor the contract can not work synchronously thus making all the inheritance hierarchy unpredictable. This also violates Liskov Substitution Principle which you will learn about in the lecture about SOLID.
* Preemptively Decouple Classes. Try to avoid super method call, use template method pattern and so-called hooks instead, they give subclass a possibility to specify the common algorithm which is controlled by superclass. Remember that it is not a ‘silver bullet’ and do not follow this approach blindly.
* Create Shallow Hierarchies (Figure 9). Try to create as compact hierarchies as it is possible. Shallow hierarchies are easy to understand, shallow and wide are slightly more difficult, but they still are easy to understand. Deep and narrow hierarchies tend to become wider and much difficult to understand and maintain. You should avoid deep and wide hierarchies, they create a long path to target method or property which is missing in a target class. Such hierarchies are difficult to maintain, and they create a high risk of application failure.

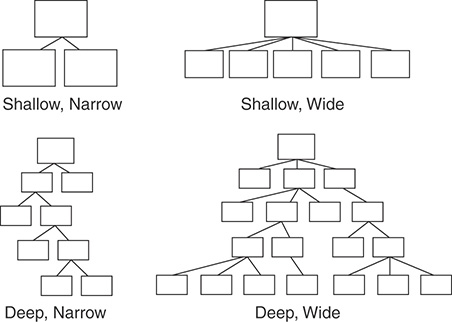


Figure 9. Hierarchies come in different shapes

# design by contract

## correctness

Design by contract (DbC), also known as contract programming, programming by contract and design-by-contract programming, is an approach for designing software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

Before we will proceed to contracts, we need to understand why we need to learn this theme, and the first term we need to understand is ‘system correctness’.



This function by itself is neither correct nor incorrect, correctness can be applied when we will talk about expected results. This function is correct if we say that ‘Returned value is half the size of the argument’, but it is not correct if we will say that ‘Returned value should be positive’ because there are no guarantees that the function will not receive a negative number as the argument.

This example makes it clear that correctness can be applied to the specification, but not to the function code.

The next part we need to review is a Hoare triple from examples two and three. The first triple is correct as before the operation x ^ 2 precondition is met and x is equal to 5, then after the operation postcondition (x is greater than 0) also will be met (subject to correct implementation of integer arithmetic). In this example postcondition is not the strongest one, the strongest possible postcondition for this precondition is {x = 25} and the weakest one is {x > 0}. We always can create a new triple from existing one making precondition and postcondition weaker or stronger.

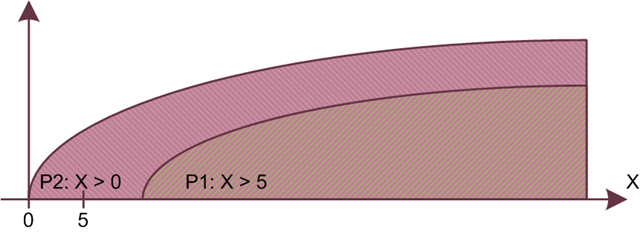


Figure 10. x > 5 is stronger than x > 0

The concept of ‘stronger’ and ‘weaker’ came from logic. It is said that the condition P1 is stronger than P2, and P2 is weaker than P1 if the fulfillment of P1 implies the fulfillment of P2, but they are not equivalent. For example, x > 5 (P1) is stronger than x > 0 (P2), since if P1 is fulfilled, P2 is fulfilled as well (after all, if x is greater than 5, then, naturally, x is greater than 0), and they are not equivalent.



Each function has strict semantic properties that reflect what the function does, regardless of how it does it.

In this case, preconditions define properties that must be met every time before function is executed, and postconditions - those properties that must be met after its execution.

The precondition binds the calling code: the conditions are defined under which the program call by the client is legitimate (for example, x > 0 for the Sqrt function or Count ! = 0 for the Pop function of the Stack class). In this case, the client's obligations benefit the provider class, since the class performing the operation does not need to worry about what to do if a precondition is violated: return a default value or error code, throw an exception, save information about the error to the I / O stream, or interrupt the program execution.

The postcondition binds the class: the conditions that must be met upon completion of the operation are determined (the Stack class must provide an increase in the number of elements by 1 after the Push function is executed). Here, the client's benefit (the result of performing the function) turns into the supplier's obligations (he can no longer fail to fulfill his obligations, since they are spelled out in the contract).

## preconditions and postconditions for inheritance

Let us review an example from Figure 11 where we have C class which contains link to a B class. Due to dynamic linking D class (or any other B subclass) can be used instead of B class after the start of program execution.

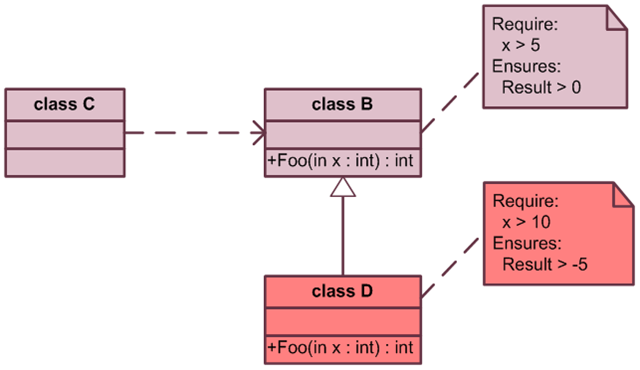


Figure 11. B – C – D classes relations





B class has a public ‘foo’ function with a precondition of x > 5 (pre\_b) and a postcondition Result > 0 (post\_b). By checking the precondition, class C fulfills its part of the contract and can expect class B (or one of its subclasses) to fulfill the contract. According to the Liskov substitution principle, the behavior of the given code fragment should not change if we substitute any B subclass.

Let us assume that the function Foo (int x) in class D starts to require more (contains a stronger precondition like x > 10), and guarantees less (contains a weaker postcondition like x > -5)

In this case, although the client of class B fulfills its part of the contract and provides an input value to the function Foo that satisfies the precondition, it may not get the expected result. Strengthening the precondition means that the data correct for the base class will become incorrect for its subclass (in our example, it can be the value x equal to 6), and weakening the postcondition means that the result that the client of the base class expects may not be returned by the subclass (in our example, this could be the result of the function Foo equal to -1).

Hence, we can conclude that when overriding methods, the precondition can be replaced only by an equal or weaker one (require less), and a postcondition - only equal to it or stronger (guarantees more). The new version of the method should not reject calls allowed in the original, and should, at a minimum, provide guarantees equivalent to those of the original version. It is free, although not obligated, to allow more calls or provide stronger guarantees.

## covariance and contravariance

Based on the previous example, we can come to terms such as covariant and contravariant, when replacing the base class with its subclass, the input values for its methods must be contravariant, that is, the precondition must be the same or weaker, and the output values must be covariant, that is, the same or stronger. Consider the following example for better understanding:







We have inheritance structure built of three classes, Locality as a base class and City and NewYork as more specific classes. Function with name ‘covariance’ accepts City instances as an argument but it also can accept NewYork instances as the more specific subclass and at the same time it can not accept City superclass as the argument – Locality because covariance will be violated in such case. Everything is vice versa for the contravariance function – it can accept City and its supertype as the argument but cannot accept City subclass.

Let us look on another example, with a more detailed implementation, to understand how contravariance works on input values:



In this example, we also have an inheritance hierarchy of three classes, the base Destination, its USADestination subclass and the even more specific TexasDestionation class. If we want to create a ShippingCalculator class that will accept USADestination as input and return the Price to us, and then create its InternationalShippingCalculator subclass which will already accept any Destination as input, then this will be correct since the contravariance of the input values ​​is observed and we can replace the ShippingCalculator with InternationalShippingCalculator and it will work. But if after that we want to extend the functionality of the ShippingCalculator by creating its TexasShippingCalculator subclass which will only accept TexasDestionation as input, then in this case we can already say that the contravariance is not observed, since the input value is more specific, and in this case we cannot replace the ShippingCalculator on TexasShippingCalculator while keeping the system working.

Based on the next example we will find out how to meet contravariance for the output values:



We have the base Animal class and its three subclasses, while the Snake and Wolf classes are its direct subclasses, and the CanisLupus class is a Wolf subclass. If we want to create a ZooCage class that will return the contents of the cage and the expected output value will be the Wolf class, then to maintain covariance in the CanisLupusCage class, we need to return the Wolf or CanisLupus class. If we create a Terrarium class with an output value of the Animal class, then the covariance will not be met, since the Animal class is a superclass of the Wolf class.