Design by Contract vs Design by Capability

- Languages like Java, C++, and C# lead us to Design by Contract.
- Smalltalk and Ruby lead us to Design by Capability.
- Groovy provides both of these. Groovy doesn't force you to use Design by Contract, unlike Java, C++.
- Statically typed languages often lead us to Design by Contract. Dynamically typed languages often lead us to Design by Capability.

Interfaces

- Interfaces are abstractions. It's a collection of methods.
- Interfaces often don't allow any implementations in them.
- In C++, interfaces come as pure abstract base classes, where they have only pure virtual functions.
- In Java and C#, there interfaces defined by the keyword `interface`.
- When a class inherits from an interface, it's called an interface based inheritance. A class can inherit from multiple interfaces.
- In Java and C#, a class can inherit from multiple interfaces, but can implement at most from one class. So, we say that these languages support multiple interface based inheritance but only support a single implementation inheritance.

Design by Contract

- Interfaces serve as a contract. Before you make a call, or even compile, you are asking for this contract.
- We cannot imagine fruitfully implementing plug-ins, especially in large enterprise applications, if we don't have the Design by Contract capability.

- With Design by Contract, a complier can verify that a piece of code meets certain expectations.

The benefit of contracts

- Streamlines writing code very nicely.
- Contract is very powerful. Very useful for compile time verification. There won’t be any runtime error because the complier will make sure that the contract is not being violated.
- Eg. when you have a interface based inheritance — the interface will require you to implement methods. This is the contract.
    - As a service provider, you can dictate terms. As a service receiver, you can abide by those terms.
- Useful for tools to figure out what the contract is and help us to implement the code.
    - Tools can assist in putting two parties together so they can lock in and work with a certain predefined contract.
- Becomes easier for programmer to communicate with each other and dictate these contracts across.

The burden of contract

- A contract is enforced all the time. We don’t the luxury to ignore the contract or to apply it selectively.
- The functions need to be placed at the abstraction at the right level.
- We may have to create interfaces to get things to work properly only to abide by the contract.

Design by Capability

- Example in Groovy-

```
// we don't need to specify the type of the parameter.
// As long as the parameter object has a method called methoc
// This is known as Design by Capabililty
public void functionName(helper) {
    helper.methodName()
}
```

The benefit of Design by Capability

- The design is very lightweight. No need to build a complex hierarchy.
- The hierarchy is a lot more flat rather than deep, which arises when we use Design by Contract.
- Becomes easier to evolve a design.

The downside of Design by Capability

- We don't know if the code/object we're passing around is actually capable. The object may not support a particular function, it may not be there, could be removed, rearranged.
- In Design by Contract, we have a complier that verifies this contract.
- In dynamically typed languages, either there's no complier or the complier is very lenient. Therefore, it's the programmer's responsibility to send the right things.
- One way to make sure that the code isn't wrong is a lot of automated testing.