Recap

- How to recognize a bad design?
- How to come up with a good design?
  - Separation of concerns.
  - Consider expected extensions.
  - Design for testability.
- Two more behavioral patterns:
  - Iterator
  - Observer
- A concrete example for the MVC paradigm.

Iterator and observer

**Iterator pattern**
- Provides (sequential) access to a data structure.
- Does not reveal implementation details.

**Observer pattern**
- Models a “one to many” dependency.
- Decoupling of state and action:
  Notify registered observer(s) about state change.

Observer pattern

<<interface>>
Observer
+ update()

<<interface>>
Observable
+ register(o:Observer)
+ unregister(o:Observer)
+ notify()
+ getState():State

MyObservable
- state:State
The Observer pattern involves an Observer interface and an Observable interface. MyObservable is an example of Observable with a state property.

Variations exist that pass incremental changes or the state to the update method.

Today
- THE behavioral pattern: visitor pattern.
- Decoupling object creation from object uses.
- Creational design patterns:
  - Factory method
  - Singleton
  - Abstract Factory
  - Builder

Visitor pattern: motivation
Problem:
- Complex data structure.
- Different types of elements.
- Support many different operations.

Goal:
- Define operations independent of
  - concrete data structure.
  - data structure traversal.

Example: Abstract syntax tree
Visitor pattern

Decoupling of object structure and operations

Pros
- Adding new operations (visitors) is easy.

Cons
- Adding new visitable elements is hard.

Object creation challenges

```java
public class Foo {
    public void bar() {
        MyClass c = new MyClass();
        c.doSomething();
    }
}
```

What if Foo should not have control over the creation of MyClass?

How to decouple the creation of an object from its uses?

Factory method pattern
Factory method pattern

“Alternative constructor” or “virtual constructor”
- Can be implemented as a class or object method.
- Singleton pattern is a special case.

Pros
- Flexibility: no hard-coded constructor calls.
- Readability: allows descriptive names.

Cons
- Needs proper documentation (e.g., naming conventions: getInstance, createXYZ, constructXYZ, …).

What about factory methods for related objects?

Abstract factory pattern

Abstract factory pattern: the products
Abstract factory pattern: the factory

Abstract factory pattern

Generalization of factory methods

Pros
- Adding a new family of products is easy.
- Consistency: clients use products of one family.

Cons
- Adding a new product is hard.

Object creation challenges cont.

Example application: typesetting/text conversion

- Fixed set of types of content (header, paragraph, ...).
- Multiple output formats (html, latex, ...).
- Different representations of the same input.

How to separate the internal representation from the output format?
Builder pattern

Decoupling of internal representation and output format

Pros
● Adding a new format is easy

Cons
● Adding a new type (internal representation) is hard

Design patterns: summary

Pros
● Solve a recurring, common design problem.
● Provide a generalizable solution.
● Provide a common terminology.
● “toolbox” for novice developers.

Cons
● Risk of over-engineering.
● Impact on complexity and system performance.

Design patterns: categories

1. Creational
   ● Singleton
   ● Factory (method)
   ● ...

2. Structural
   ● Composite
   ● Decorator
   ● ...

3. Behavioral
   ● Template method
   ● Visitor
   ● ...

Just a subset of patterns!