Design Patterns

Episode 1

Design Patterns Series

- Saturday, October 21, 2017: **Episode 1**
  Introduction, `Composite`, `Singleton`, `Flyweight`, `Observer`
- Saturday, December 2, 2017: **Episode 2**
  `Decorator`, `Factory Method`, `Abstract Factory`, `Proxy`, `Command`
- Saturday, December 16, 2017: **Episode 3**
  `Adapter`, `Facade`, `Mediator`, `Strategy`, `Chain of Responsibility`
- Saturday, January 13, 2018: **Episode 4**
  `Bridge`, `Builder`, `Visitor`, `Prototype`, `Interpreter`
- Saturday, February 10, 2018: **Episode 5**
  `Memento`, `State`, `Template Method`, `Iterator`

Design Patterns - Episode 1

1. Introduction to Design Patterns
2. Composite
3. Singleton
4. Flyweight
5. Observer

Schedule

10:00 - 12:00: Introduction, Composite

12:00 - 13:00: Lunch break

13:00 - 16:00: Singleton, Flyweight, Observer

Questions? Ask right ahead.

Need a break? We'll take one.

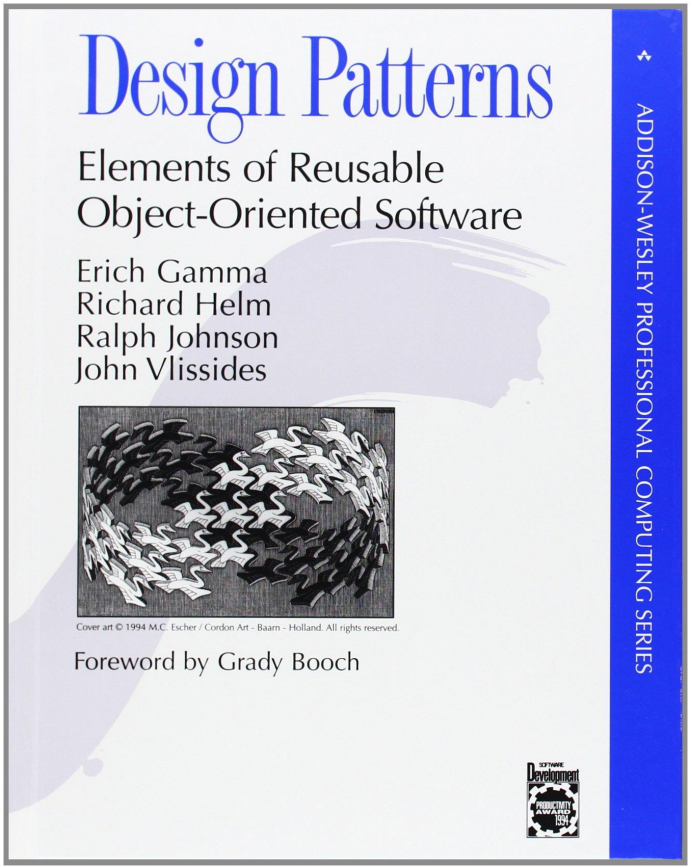
For each Pattern

1. The Problem
2. The Pattern
3. Use if...
4. Implement it in Java
5. Consequences, open discussion

Design Patterns

Gamma, Helm, Johnson, Vlissides:
Design Patterns. Elements of Reusable Object Oriented Software.
1995 Pearson Education

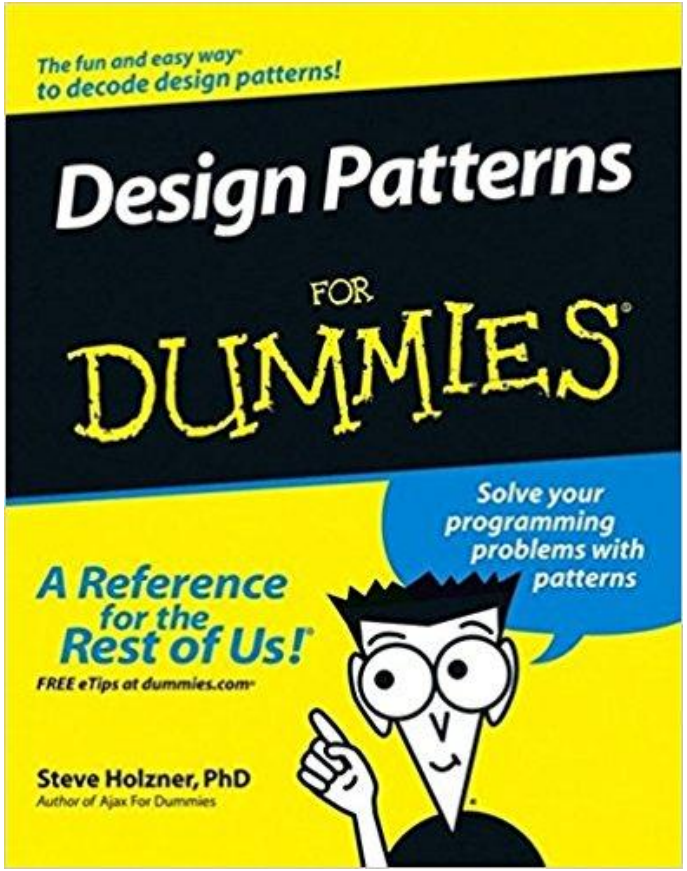
„Gang of Four“
„GoF design patterns“



Design Patterns

Holzner:
Design Patterns for Dummies.
2006 Wiley Publishing

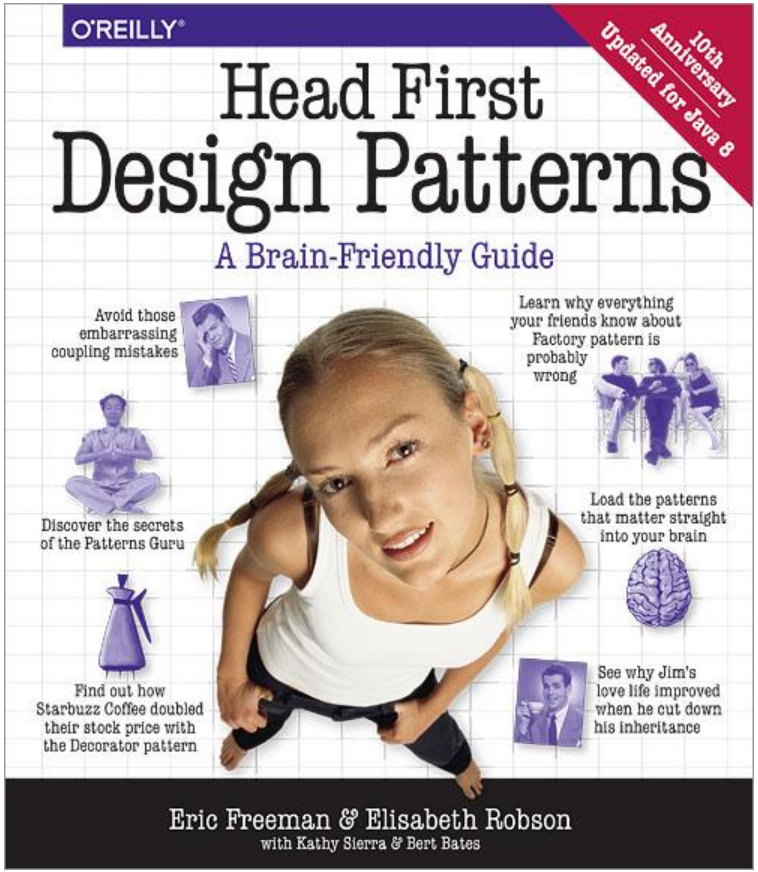
„Dummies“



Design Patterns

Freeman, Robson, Bates, Sierra:
Head First Design Patterns:
A Brain-Friendly Guide.
2004 O'Reilly

„Head First“



Design Patterns

What is a pattern?

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice“
>
> ~ GoF / Christopher Alexander

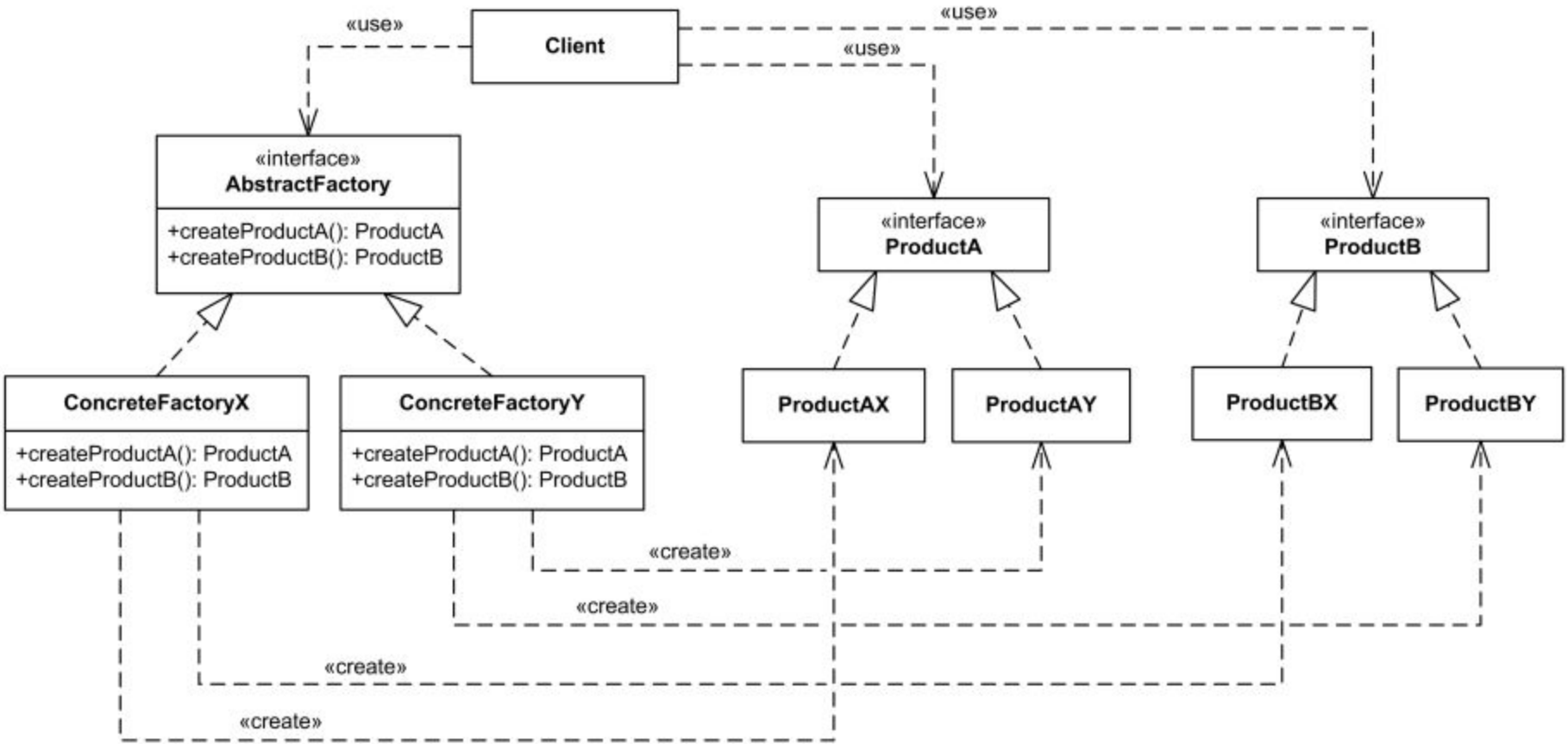
Design Patterns

What is a pattern?

> “Congratulations, your problem has already been solved.“
>
> ~ Dummies

Design Patterns

What is a pattern in object-oriented software?


«use»
Client
«use»
«use»
«interface»
AbstractFactory
+createProductA(): ProductA
+createProductB(): ProductB
«interface»
ProductA
«interface»
ProductB
ConcreteFactoryX
+createProductA(): ProductA
+createProductB(): ProductB
ConcreteFactoryY
+createProductA(): ProductA
+createProductB(): ProductB
ProductAX
ProductAY
ProductBX
ProductBY
«create»
«create»
«create»
«create»

Design Patterns

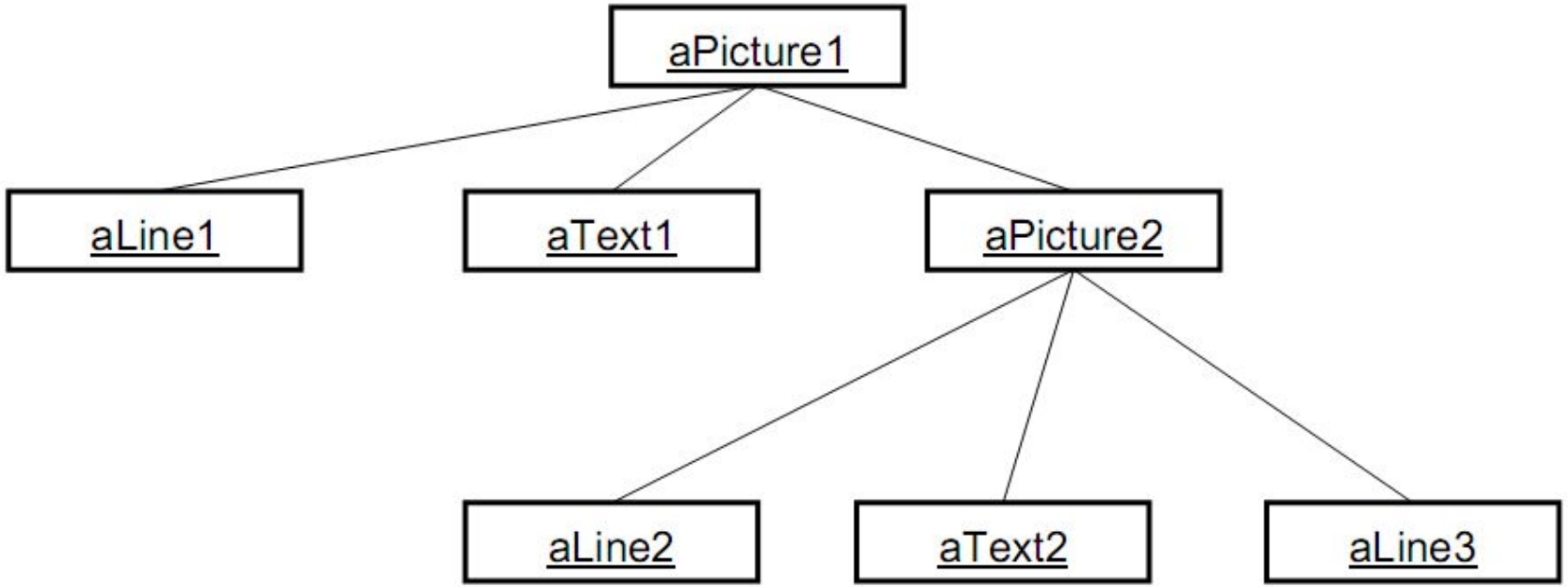
UML Diagrams

Creational Patterns	Structural Patterns	Behavioral Patterns
Singleton	Composite	Observer
Factory Method	Flyweight	Command
Abstract Factory	Decorator	Mediator
Builder	Proxy	Strategy
Prototype	Adapter	Chain of Responsibility
	Facade	Visitor
	Bridge	Interpreter
		Iterator
		Memento
		State
		Template Method

Creational Patterns	Structural Patterns	Behavioral Patterns
Singleton	**Composite**	**Observer**
Factory Method	**Flyweight**	Command
Abstract Factory	Decorator	Mediator
Builder	Proxy	Strategy
Prototype	Adapter	Chain of Responsibility
	Facade	Visitor
	Bridge	Interpreter
		Iterator
		Memento
		State
		Template Method

Composite

Composite

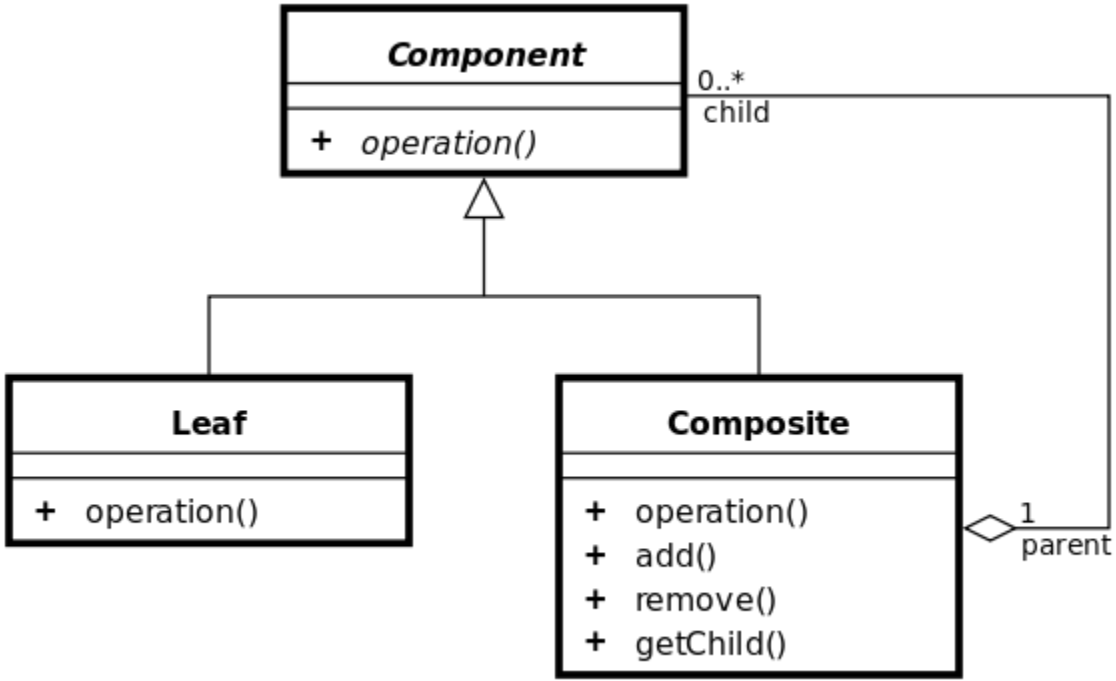


Composite

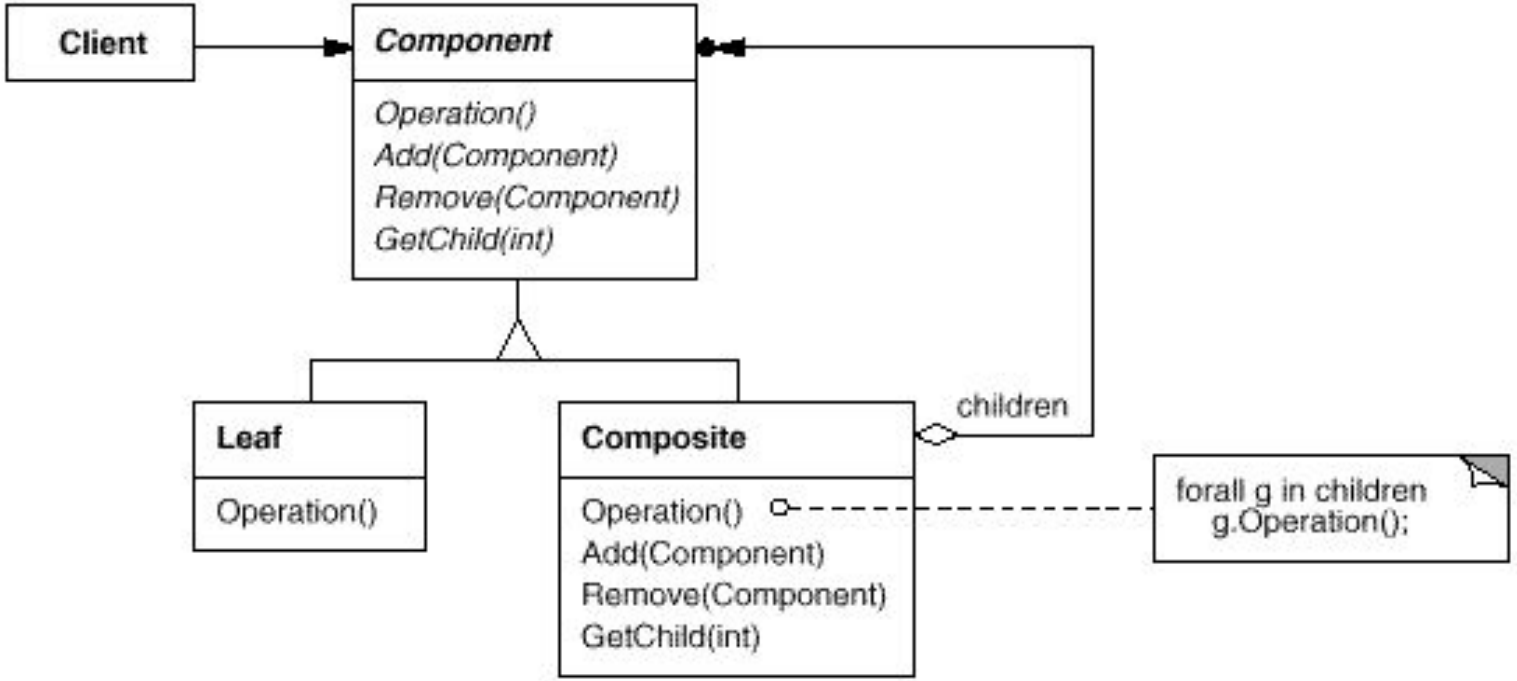
The Problem:

> You want to create a structure that has recursive properties: parent objects that can have children that again are parents. You need to build a tree of objects, where objects can be both nodes and leaves.
>
> Necessary e.g. for building composed graphics elements, or for building nested layout components.

Composite


Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

Composite


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

Composite

The Pattern:

1. Create either an abstract class / interface, which defines the methods that both leaf and node have in common.
2. Create a class for the leaf that implements this abstract class / interface.
3. Create a class for the node that also implements the abstract class / interface. The node has a list of your abstract class / interface as member.
4. If you implement the methods, iterate over the children and call their methods *recursively*.

Composite

Use if…

> … you want objects to be connected in a tree structure.
>
> … the client should be able to use the composed parent object the same way as leaf objects.

Composite

Let’s touch some code.

Composite

Consequences:

- Allows for class hierarchies from primitive and composite objects.
- Makes the access for a client object easier.
- Simplifies adding of new composite or leaf objects.

Drawback:

- It’s hard to limit the breadth or depth of the hierarchy (needs runtime checks).

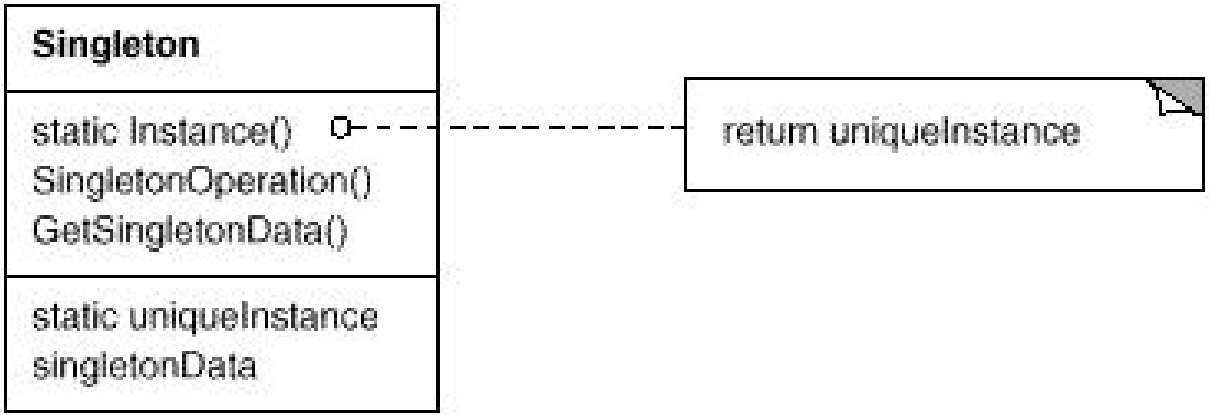
Singleton

Singleton

The Problem:

> You have one very large class that holds many other objects, e.g. a database class. You need exactly one instance of it, because more would compromise your application's logic, or would be unnecessary allocation of memory.

Singleton


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

Singleton

The Pattern:

1. In the class that you want to make a singleton, create a `private static` member of the same class that you call `instance`.
2. Make the constructor private.
3. Create a `static` method called `getInstance()` that returns your singleton instance. Check if it has been initialized, and if not, create it.

Singleton

Use if…

> … there should be exactly one instance of an object.

Singleton

Let’s touch some code.

Singleton

Consequences:

- Access control to only instance by singleton class.
- No need to hold instances of big object as global variables.

Drawback:

- Makes it hard to test a class that uses the singleton instance.
- Alternative: Dependency Injection frameworks (e.g. Dagger)

Flyweight

Flyweight

Singleton: explicitly only one object instance

Flyweight: Let one object look like it was many

Instead of creating many big objects, you create a set of small reusable objects: *flyweights*.

Flyweight

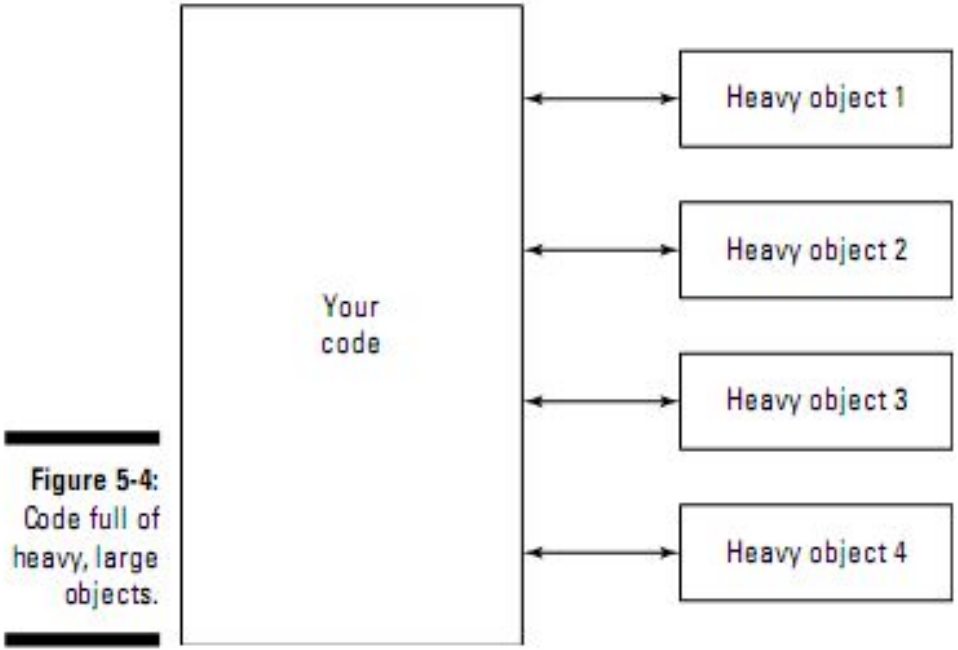


Figure 5-4: Code full of heavy, large objects.

Flyweight

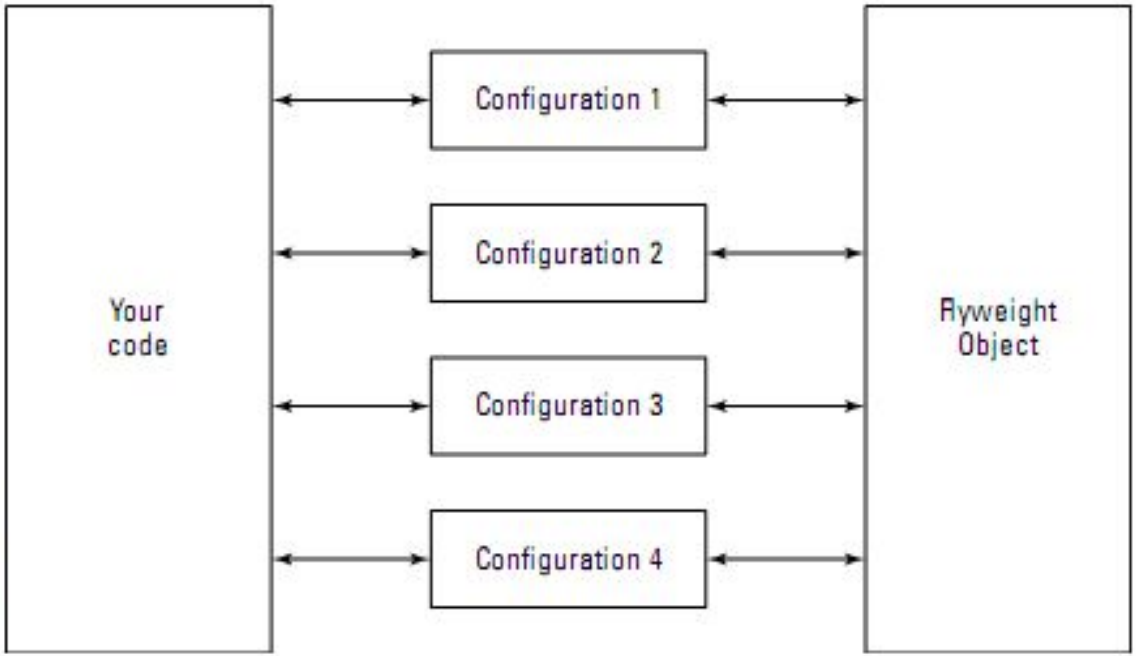


Figure 5-5: Using a flyweight object.

Flyweight

The Problem:

> You have a large number of big objects, many of which are duplicates. You want to reduce their number and their weight.

Flyweight

The Pattern:

1. Identify objects or parts of a class that are repeated many times.
2. Create a pool class that can hold all items that are repeatedly used.
3. Use these items by getting the same instances from the pool repeatedly.

Flyweight

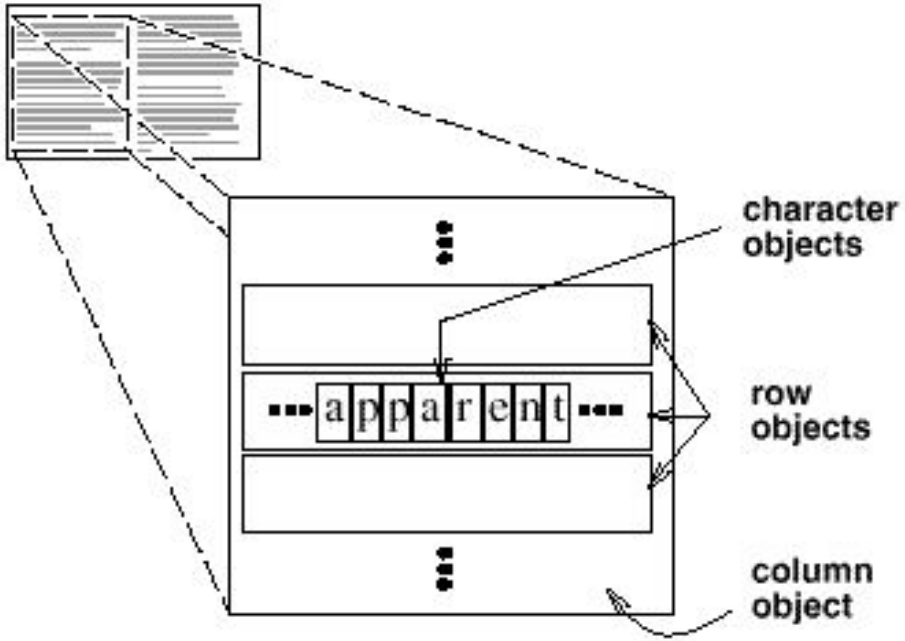
Example: Editor

We want to display many characters at different positions (row, column).

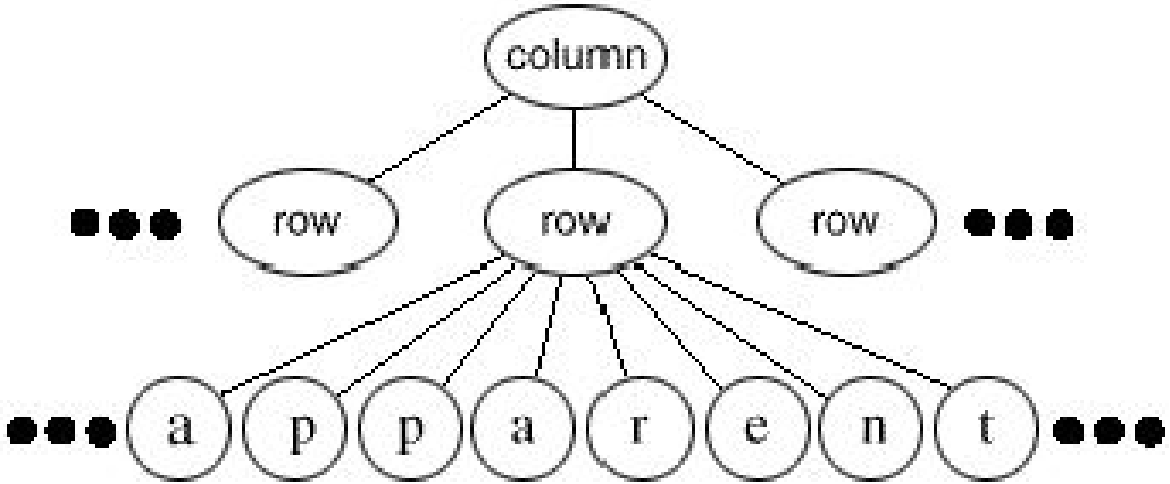
It would be a waste of memory if we created a single object for every character.

Instread, we create a pool of all characters (flyweights), and display them at the respective positions.

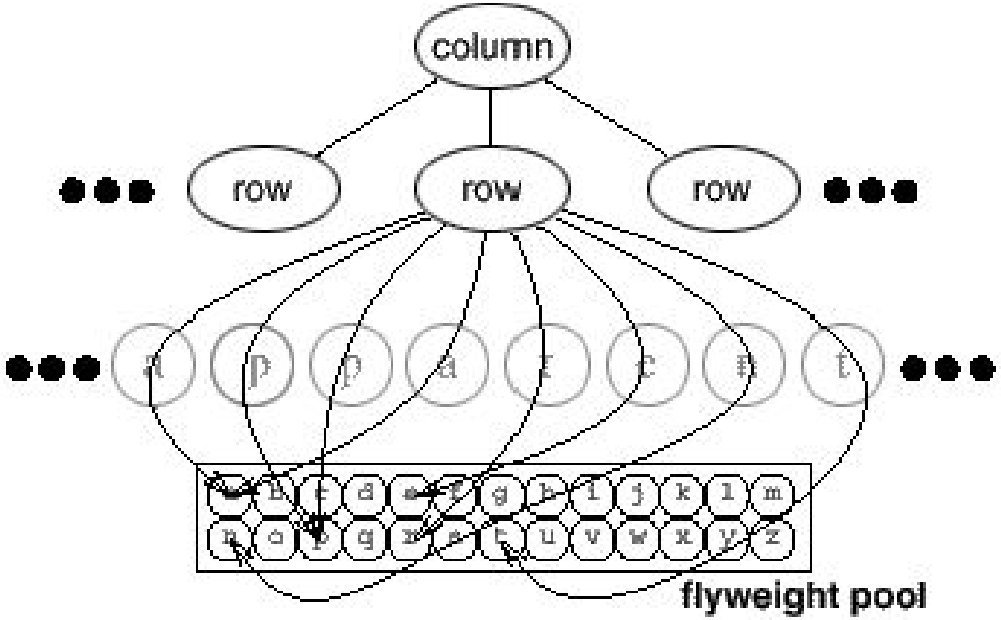
Flyweight



Flyweight



Flyweight



Flyweight

A Flyweight is a shared object: **extrinsic state** vs. **intrinsic state**

Intrinsic state:

- Similarities of objects, properties that can be shared
- E.g.: shape of letter A

Extrinsic state:

- Properties that depend on context
- E.g.: the position (row, column) where letter A is

Flyweight

Extract the *intrinsic* state: **flyweight object**

Flyweight

Use if …

… you have large duplicated intrinsic state,

… and you want to save resources.

Flyweight

Let’s touch some code.

Flyweight

Consequences:

- Reduces the number of object instances.
- Reduces an object’s intrinsic state to an object.
- The more effective a flyweight is, the more resources are saved.
- Often combined with composite.

Observer

Observer

The Problem:

Objects need to update their state depending on certain events.

Coupling them directly would make things complicated and reduce their reusability.

We want each object to update themselves if an event occurs.

Observer

The Solution:

A 1..* relationship between objects, where the objects can be automatically notified and updated when the single object changes (event).

Implementations: e.g. EventListener, RxJava’s Observer

Observer

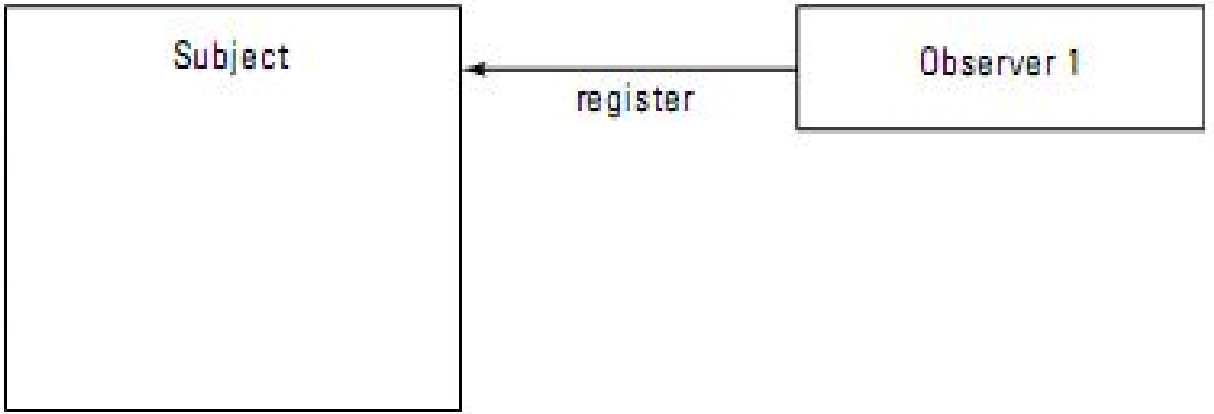


Figure 1-8: The Observer pattern lets observers register with subjects.

Observer

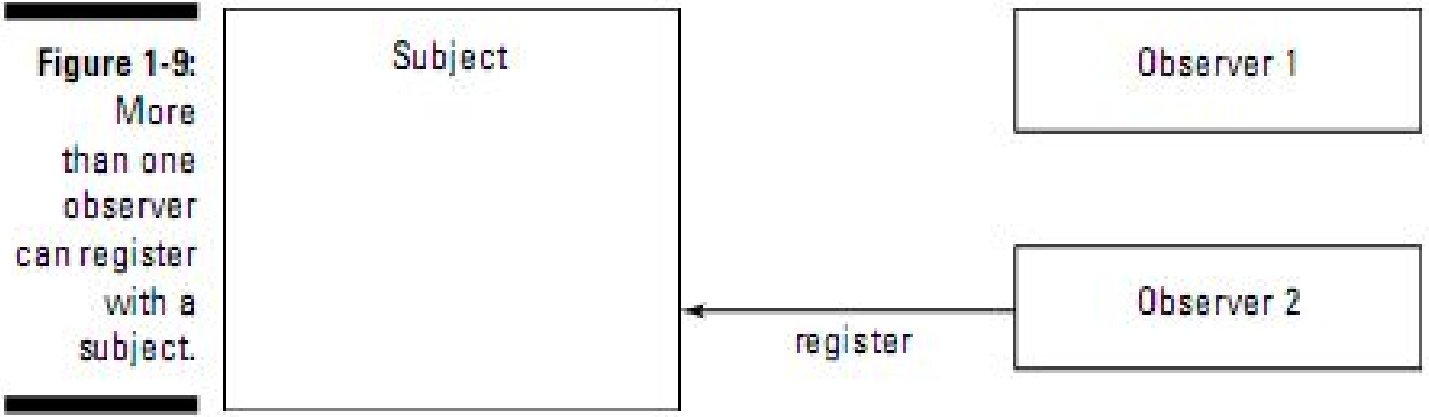


Figure 1-9: More than one observer can register with a subject.

Observer

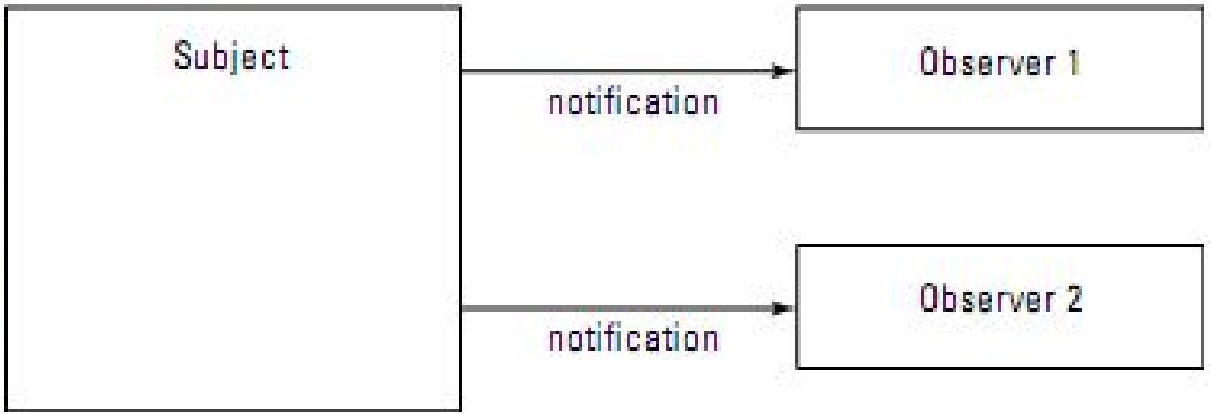
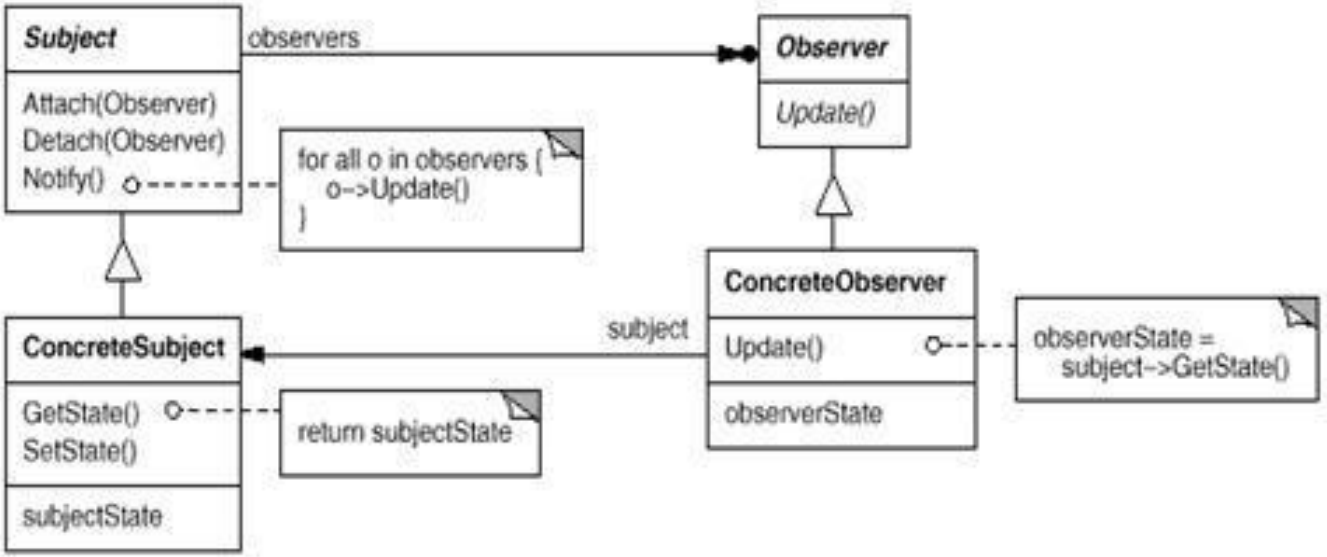


Figure 1-10: When events occur in the subject, registered observers are notified.

Observer


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Update()
ConcreteObserver
ConcreteSubject
subject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

Observer

The Pattern:

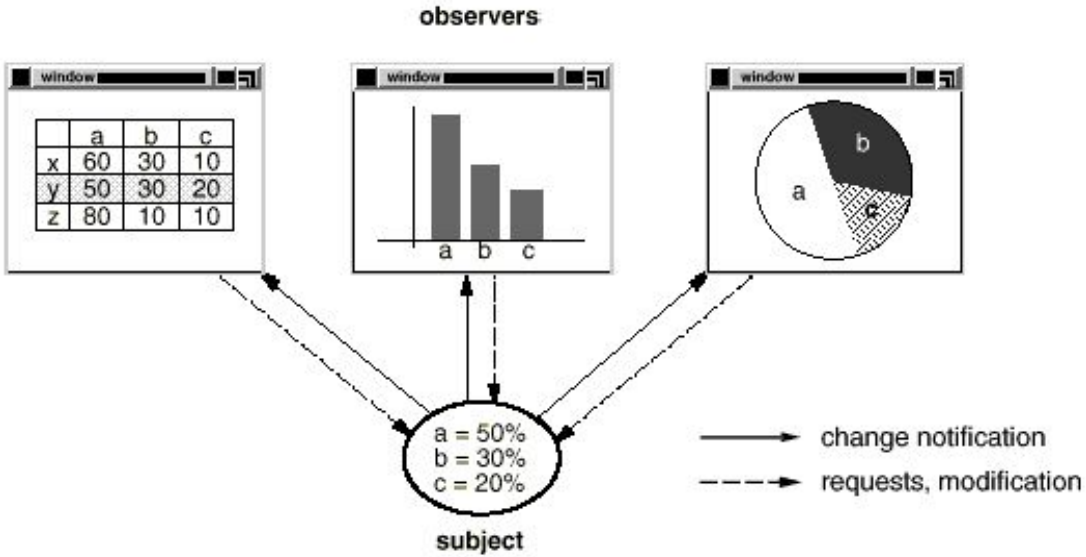
1. Create a `Subject` class and an `Observer` interface with a `notify()` method.
2. The Subject class holds a list of Observers. They are added to it by a method `subscribe(Observer observer)` and removed by `unsubscribe(Observer observer)`.
3. A method `accept()` calls the `notify()` method on each subscribed observer.

Observer

4. All objects that want to be notified to update themselves need to implement the Observer interface, and subscribe to the subject.
5. If all objects should be updated, call the `accept()` method of the Subject.

Observer

Example: Graphic User Interface



Observer

Example: Graphic User Interface

- Graphic displays depend on the same data source
- If data changes, all graphics change
- **publish** - **subscribe**

Observer

User if…

> … two or more objects should depend on each other and be *synchronized*.
>
> … if a change of one object should should cause change in another object, without necessarily knowing how many there are.
>
> … an object should be able to notify other objects without knowing their identity.

Observer

Let’s touch some code.

Observer

Consequences:

Broadcast: The messages sent by subject don’t need a receiver. Registered observers decide for themselves if they act upon the received message or ignore it.

Drawback:

Unwanted updates: the object calling the accept() method of the subject doesn’t know about its observers. Consequences for sending the event can be underestimated.

Design Patterns - Episode 1

- `Composite`
- `Singleton`
- `Flyweight`
- `Observer`

Any questions?

Next Episodes:

- Saturday, December 2, 2017: **Episode 2**
  `Decorator`, `Factory Method`, `Abstract Factory`, `Proxy`, `Command`
- Saturday, December 16, 2017: **Episode 3**
  `Adapter`, `Facade`, `Mediator`, `Strategy`, `Chain of Responsibility`
- Saturday, January 13, 2018: **Episode 4**
  `Bridge`, `Builder`, `Visitor`, `Prototype`, `Interpreter`
- Saturday, February 10, 2018: **Episode 5**
  `Memento`, `State`, `Template Method`, `Iterator`

Thank you!

Hamburg Coding School
www.hamburgcodingschool.com