Patterns

An Easier Way to Think About
Common Software Designs

Patterns make it easier to talk about, and think about software.

Patterns aren't revolutionary design ideas.

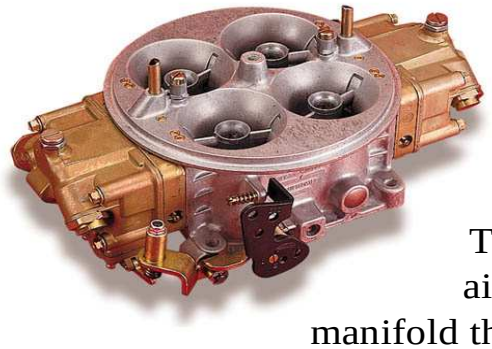
You are probably writing patterns already.

Patterns are common designs that naturally appear in software.

If we are sufficiently clear, patterns should feel familiar, like an old pair of sneakers.

Pattern are just a common language for talking and thinking about software.

The Carburetor Pattern



The thing between the air filter and the intake manifold that mixes fuel and air and regulates the flow of the air/fuel mixture.

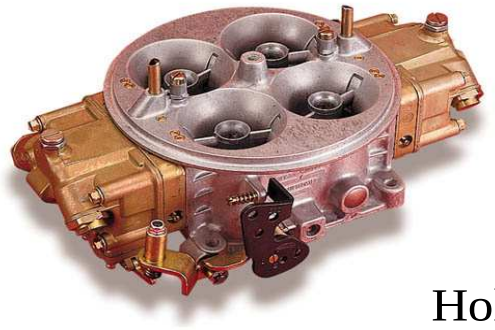
Here's our first pattern. It's the fuel/air regulation system for a car.

We could talk about this thing by detailing its purpose.

"To fix your car we need to rebuild the thing between the air filter and the intake manifold that mixes fuel and air and regulates the flow of the air/fuel mixture."

OK, I know what it is, but that's quite a mouthful.

The Carburetor Pattern



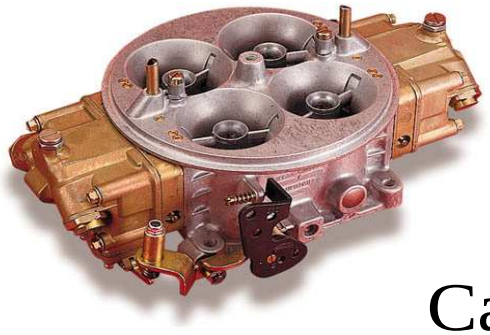
Holley Street/Strip
750 CFM Four Barrel Carb

Another way to talk about this thing is to use the name of this particular version.

“To fix your car we need to rebuild your Holley seven fifty four barrel.”

So now we have a less cumbersome name, but we've lost the information about what it does.

The Carburetor Pattern



Carburetor

The easiest way to talk about it is to use its pattern name.

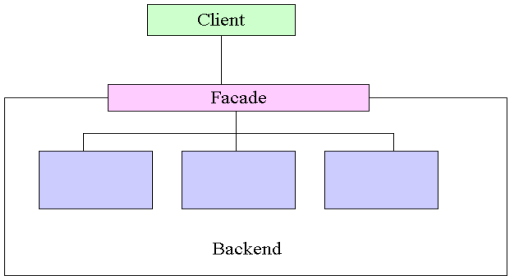
"To fix your car, we have to rebuild the carburetor."

This kind of shorthand can be called jargon, or classification, or patterns.

The word "carburetor" captures the idea of a device for mixing fuel and air.

The Facade Pattern

The class between the client and the backend classes that hides the complexity of the backend classes to make it easier for the client to use the backend classes.

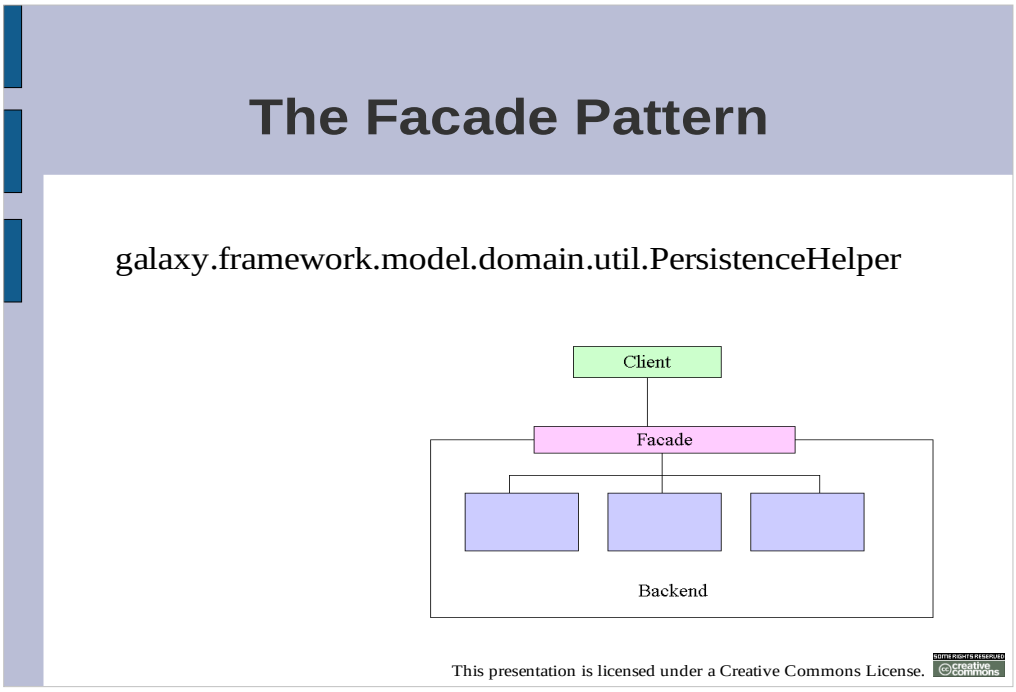


The same can be done with software patterns.

We can detail the purpose.

“To make it easier to persist objects, we wrote a class between the client and the backend that hides the complexity of the backend and makes it easier for the client to use the persistence engine.”

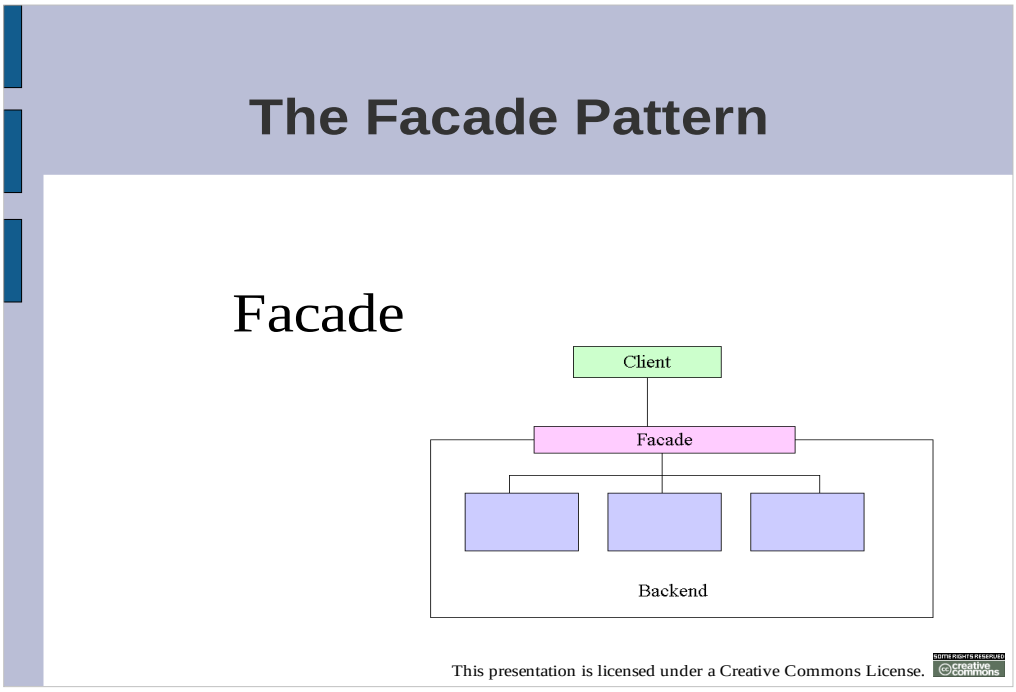
We've made it clear what we're doing, but it takes too long to say.



Another way would be to use the actual class name when talking about it.

“To make it easier to use the persistence engine, we wrote Persistence Helper.”

Hopefully the class name gives an idea of what is going on, but the developer you're talking to has to make an educated guess.



Shorthand for this kind of class is “Facade”.

“To make it easier to persist objects, we wrote a facade.”

It's short and simple, and it explains what the role is.

Patterns are software jargon.

The facade pattern captures the idea of putting a simplified interface on a complex system.

Patterns don't belong to any one language.

The original software pattern book <hold up GoF, ISBN: 0201633612> had examples in SmallTalk, but the patterns work just as well in Java, C++, or C Sharp.

The book that started it all <hold up A Pattern Language, ISBN: 0195019199> isn't about software at all, it's about architecture and city planning.

This, by the way, is a really fun book.

The Bumper Pattern

Here's another pattern with which you are probably familiar.

The bumper pattern is shorthand for a device that absorbs, deflects, or repels impacts.

There are slightly different instances of the bumper pattern on the front and back of every car.

The pattern doesn't specify color, width, or stopping power.

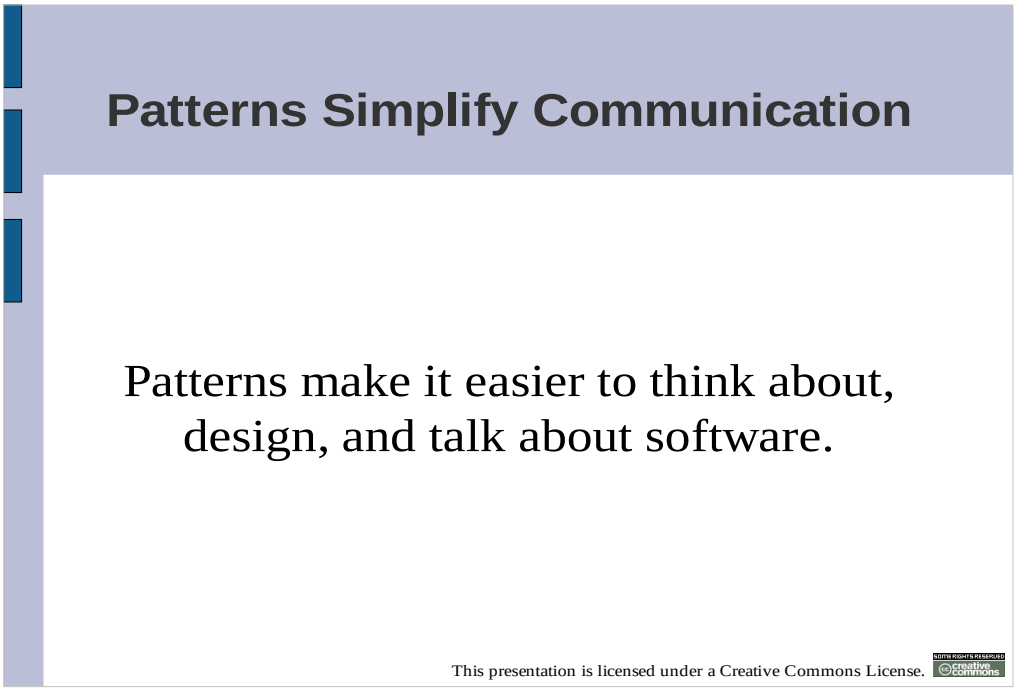
The Bumper Pattern

The bumper pattern is platform independent.

Much like software patterns, the bumper pattern is platform independent.

Bumper cars, bumper pool, and pinball bumpers are very diverse implementations of the bumper pattern.

The idea of absorbing, deflecting, or repelling an impact is retained in the bumper pattern, regardless of the specific implementation.



Patterns make it easier to think about software.

Patterns allow you to focus on the role that a component plays when designing a system.

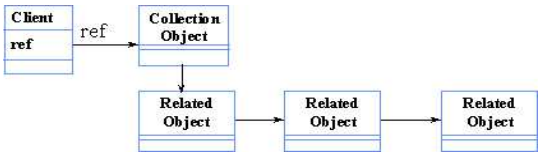
Patterns free you from thinking about the implementation details.

Patterns make it easier to describe a complex system in bite-sized chunks.

The Collection Pattern

Intent

How do you model a one to many relationship?



Solution

1. Make a Collection Object
2. Usually encapsulates a linked list or queue
3. May be ordered or unordered

Consequences

* Usually used to implement uni-directional relationships
* Most common form of Relationship Object

Here's a few software patterns to give you a quick feel for them.

This continuing series of lunches will go into much more detail on individual patterns.

This is the collection pattern.

Does this make sense to you? I find it really confusing.

The Collection Pattern

Two “Car Trunk” Implementations
of The Collection Pattern

This makes sense to me.

A collection is just a group of things that have something in common.

In these two examples, we have collections of “things in the trunk”.

The objects themselves may or may not be related otherwise.

They're part of the same collection of objects.

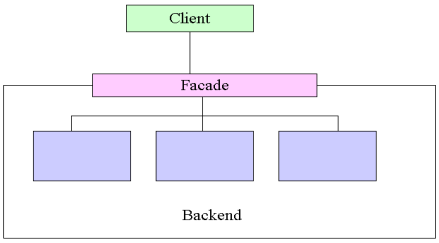
The Facade Pattern

Intent

Encapsulate a subsystem using a high-level interface, simplifying subsystem usage and hiding structural details.

Solution

1. Clients communicate with subsystem objects by calling methods in Facade
2. Clients never (or as seldom as possible) directly accessing objects in subsystem -- any such access weakens the encapsulation
3. Subsystem objects usually retain no knowledge of client
4. Subsystem objects do not normally maintain any reference to Facade



This is the facade pattern we looked at earlier.

Once again, there's a whole lot of data here, but you have to know what a facade is to be able to understand it.

If I already knew what a facade was, I wouldn't be reading a description of what a facade is.

The Facade Pattern

The “Give Me Money” and “Start the Car” implementations of the facade pattern.

Here's a couple of facades we use all the time.

The ATM hides the complexity of handling banking transactions – you just want your money.

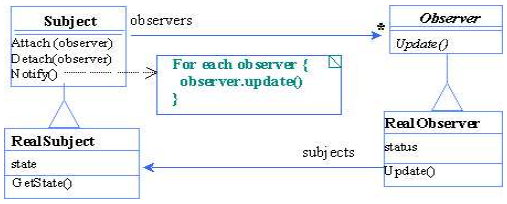
You don't have to know how your engine works to be able to start your car.

Facades hide the complexity that the user doesn't care about.

The Observer Pattern

Intent

Define relationship between a group of objects such that whenever one object is updated all others are notified automatically.



Applicability

* Changes to one object in a group requires changes to the others in the group
* The number of objects in the group may vary
* Loose coupling is desired between the objects in the group

This is an extremely influential pattern.

Virtually every modern graphical program uses the observer pattern.

You may have heard of MVC (model view controller); the observer pattern is critical to good MVC design.

I have written half a dozen observers this month, and I still find this diagram baffling.

The Observer Pattern

The “Security Guard” implementation of the observer pattern.

I love this picture.

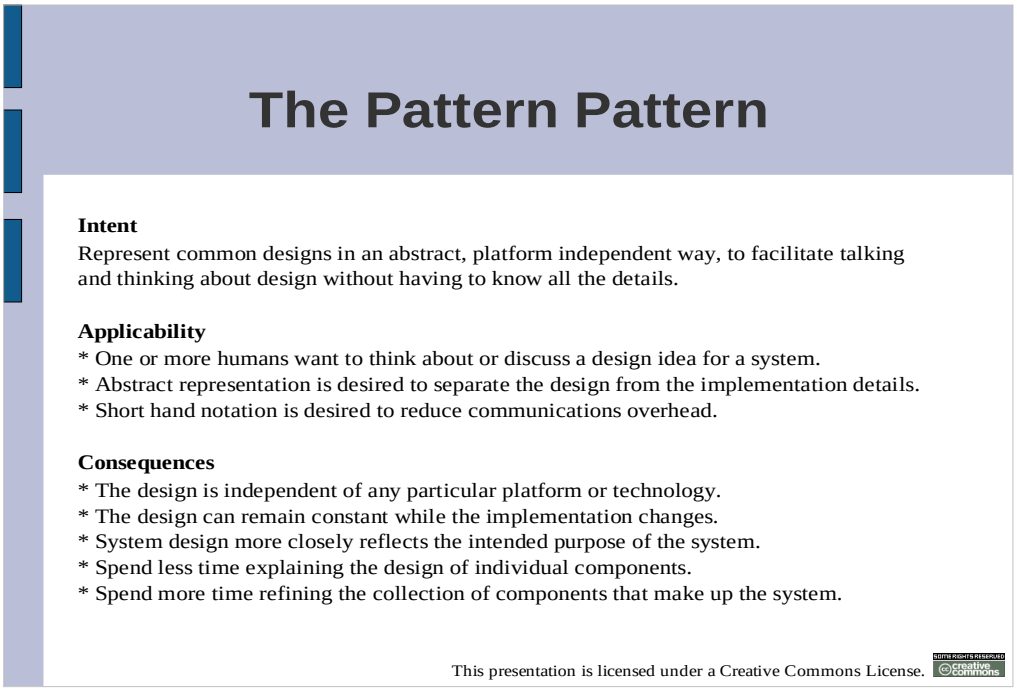
This guy is an observer, he watches you in case you shoplift.

See the whistle hanging from his shoulder?

If your state changes from “customer” to “shoplifter”, he blows that whistle, which notifies all the interested parties.

An observer watches other objects.

If the watched object changes, the observer tells everyone who wants to know.



This is the most important pattern, take a minute to look it over.

<wait 2 minutes>

That last consequence might be a little optimistic, but one can hope, right?

So that's an introduction to patterns. I hope that it serves to take some of the fear, uncertainty, and doubt away. Patterns are like a good pair of sneakers – once you've used them for a while, they'll fit like a glove.

You can get a copy of this presentation at: http://traxel.com/patterns/