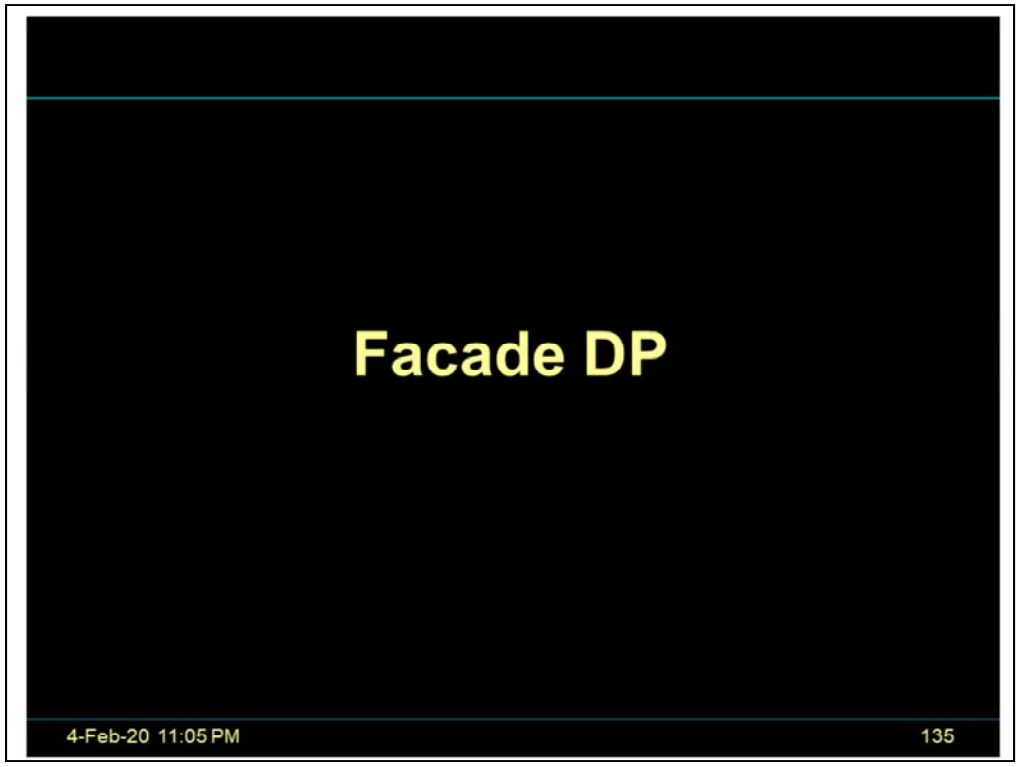
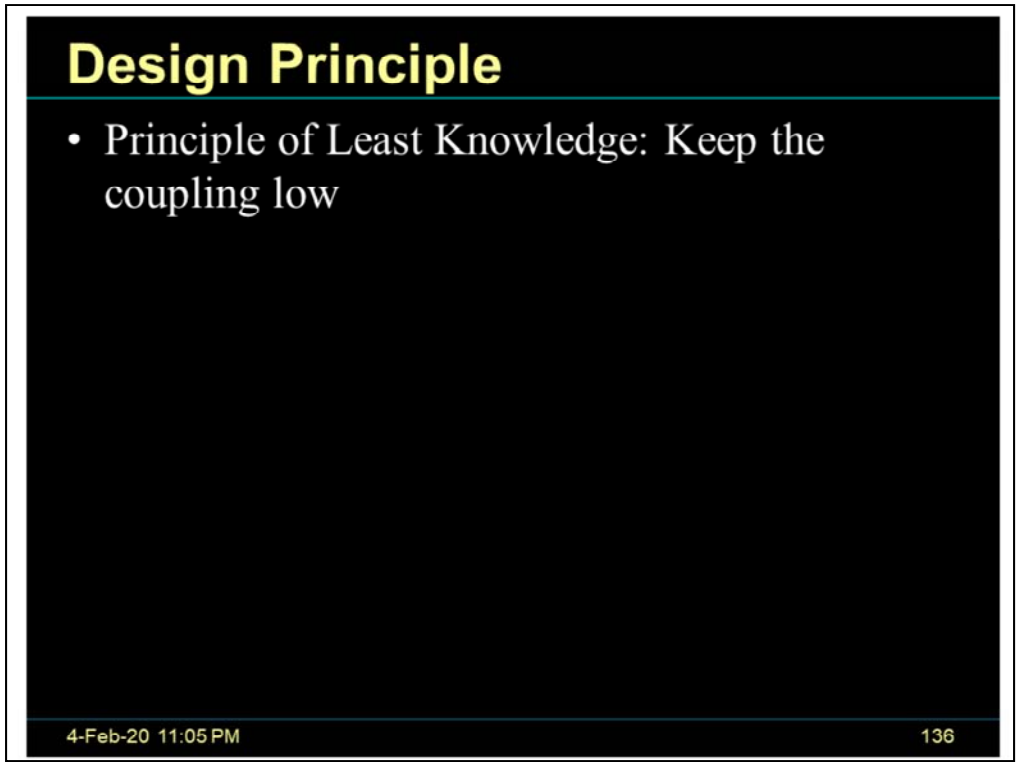

Facade DP
4-Feb-20 11:05 PM
135



Meaning:

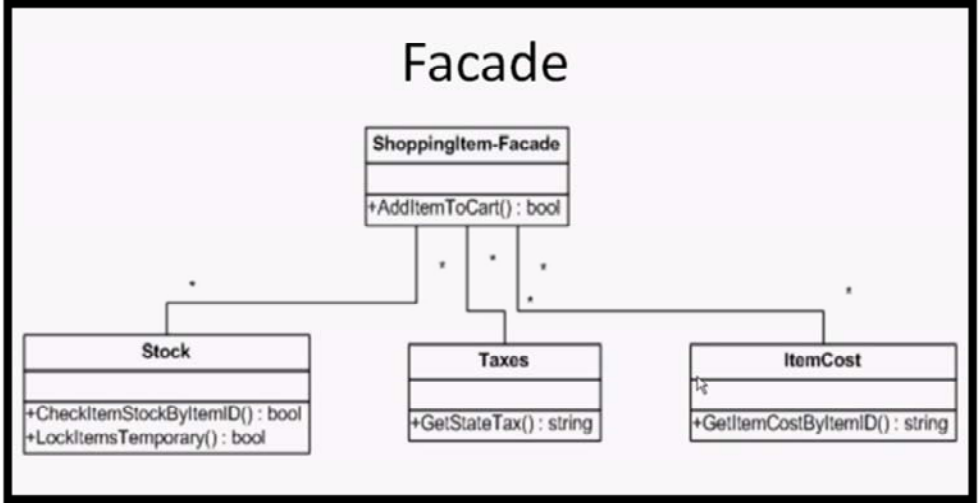
The number of classes, a class interacts with, should be few.

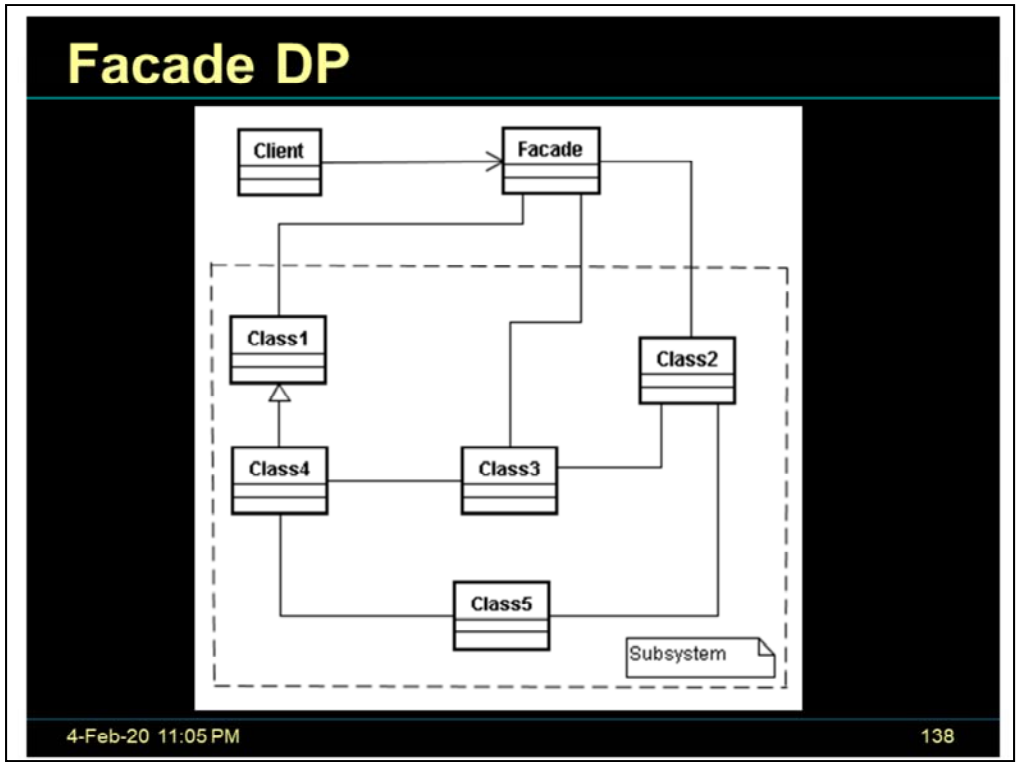
Changes to one class should not usually cause a huge cascade of changes in the system.

High coupling is fragile system.

It is costly to maintain and complex to understand.

Example


Facade
ShoppingItem-Facade
+AddItemToCart() : bool
Stock
+CheckItemStockByItemID() : bool
+LockItemsTemporary() : bool
Taxes
+GetStateTax() : string
ItemCost
+GetItemCostByItemID() : string



A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

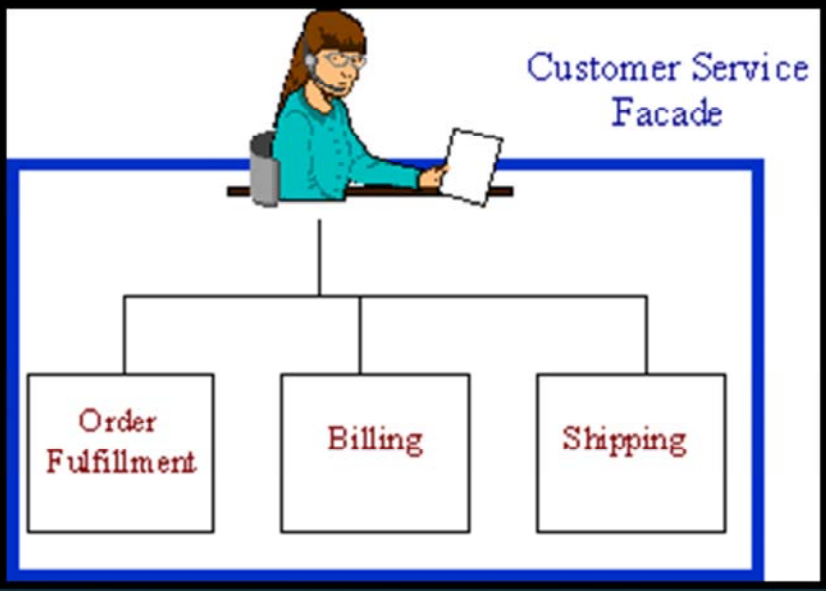
- If something is ugly, we try to hide it inside a good looking object.
- We always want to look good on the outside.

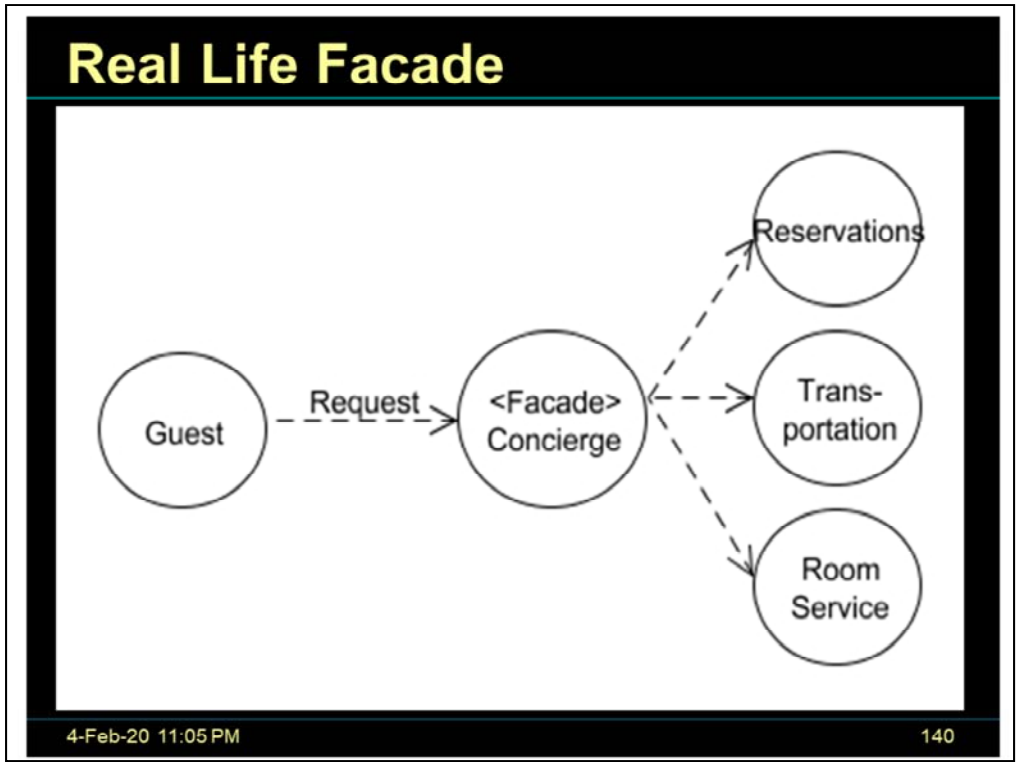
Facade design pattern provides a unified interface to a set of interfaces in a subsystem.

Facade defines a higher level interface that makes the subsystem easier to use.

- Facade alters the interface like Adapter.
- Facade's purpose is to simplify the Interface.
  - It hides all the complexity of one or more classes behind a clean simple interface.
- Facade reduces the learning curve necessary to use a large subsystem.

Real Life Facade



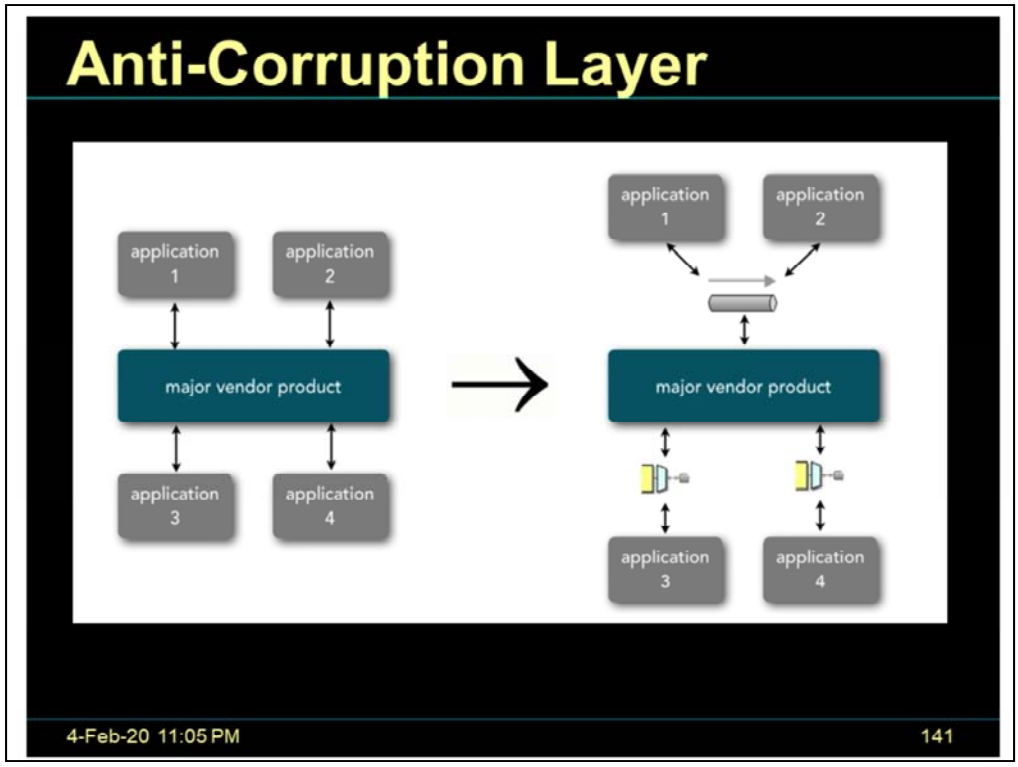


A hotel concierge (at an upscale hotel) stands between the hotel guest and all the various services that are needed to make for a successful, satisfying stay at the hotel.

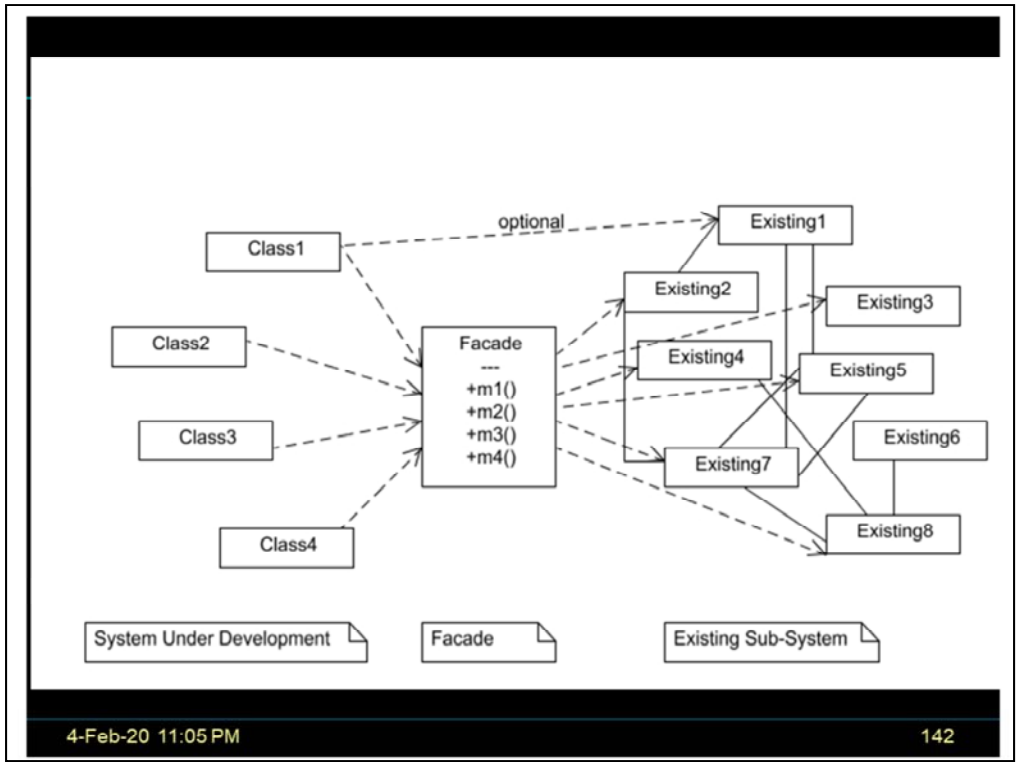
The guest has goals, which are expressed from the guest's point of view. The concierge translates these goals into needed interactions with the concrete services needed to achieve them.

For example, the guest may say "we'd like to go out to an evening dinner, and a show, and they return to the hotel for an intimate dessert in our room".

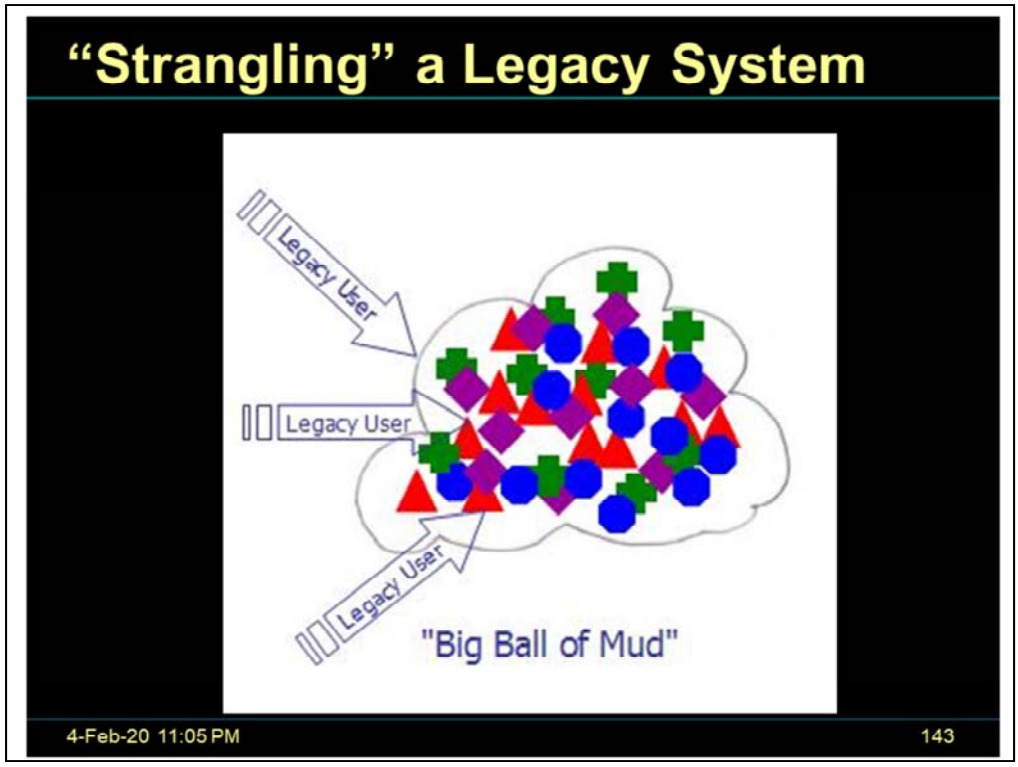
The concierge handles all the nitty-gritty details (a taxi, restaurant reservations, theatre tickets, housekeeping prep of the room, the kitchen preparing the dessert, room-service delivery, etc...)



Product-agnostic Architecture: Isolate products to avoid vendor lock-in.
The left-side diagram is “Vendor King Anti-Pattern”


optional
Class1
Existing1
Existing2
Existing3
Class2
Facade

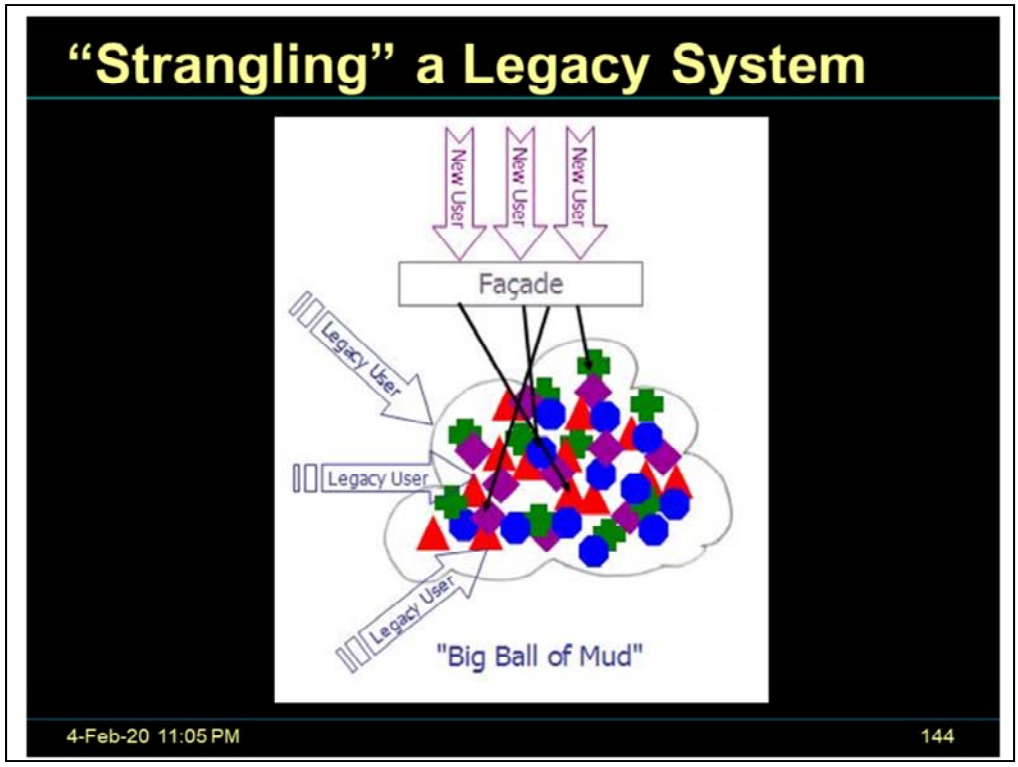
+m1()
+m2()
+m3()
+m4()
Existing4
Existing5
Class3
Existing6
Existing7
Existing8
Class4
System Under Development
Facade
Existing Sub-System
4-Feb-20 11:05 PM
142



Imagine an old, very decayed, and poorly-implemented legacy system. You've probably had to work with one or more of these in your practice. We often think of them as a "big ball of mud" because they are incomprehensible, brittle, and "dense“.

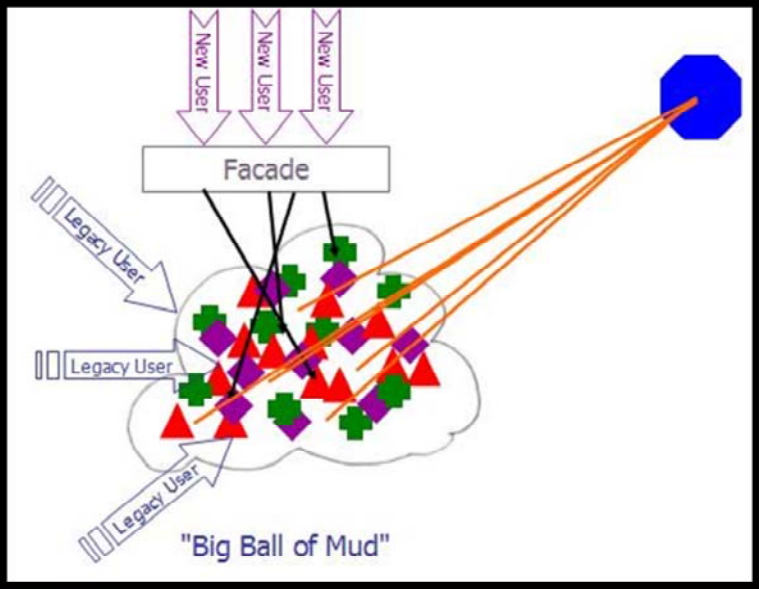
Job one is to stop using the system in the old way when writing new code that has to interact with it. If we could, we'd stop here and refactor or rewrite the system entirely, but often that is simply not an option: we have to live with it, at least for now.

However, using the old system directly (as the "legacy users" do) will mean writing code that is similar in nature to the code in the system, and at the very least we want our *new* code to be clean, tested, and object-oriented. So, we create a Façade with the interface we "wish we had" and then delegate from there into the various routines in the legacy system.

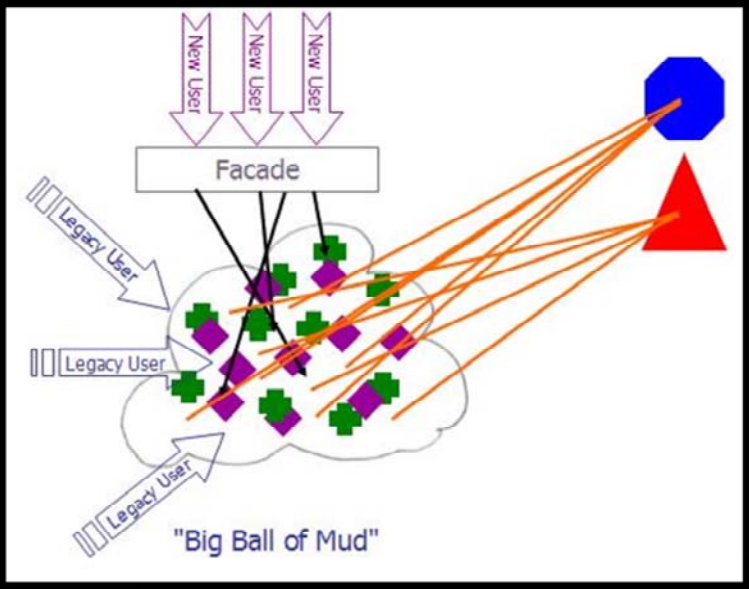


Now that we have this in place, the "new users" can be developed more cleanly and with higher quality. Now, over time, and without significant time pressure on the project, we can incrementally pull out behaviors from the legacy system into more clean, tested, object-oriented code, and delegate from the old routines into the new ones:

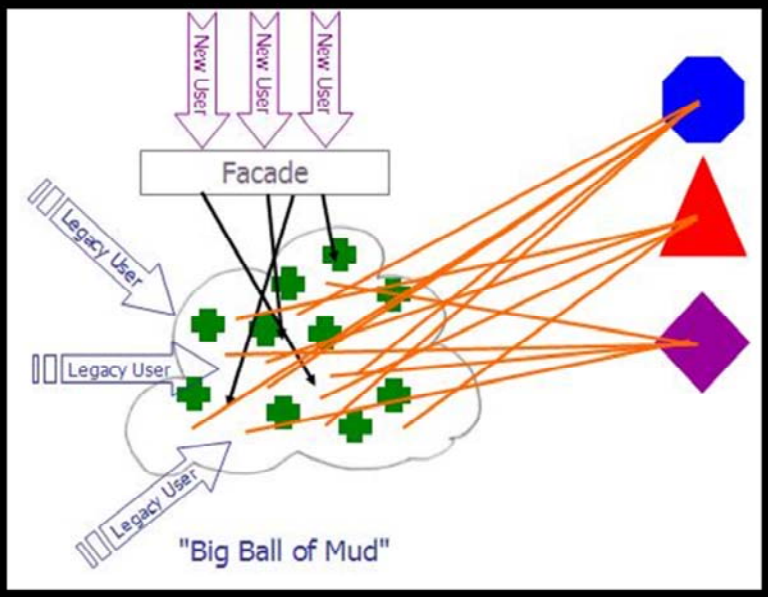
“Strangling” a Legacy System

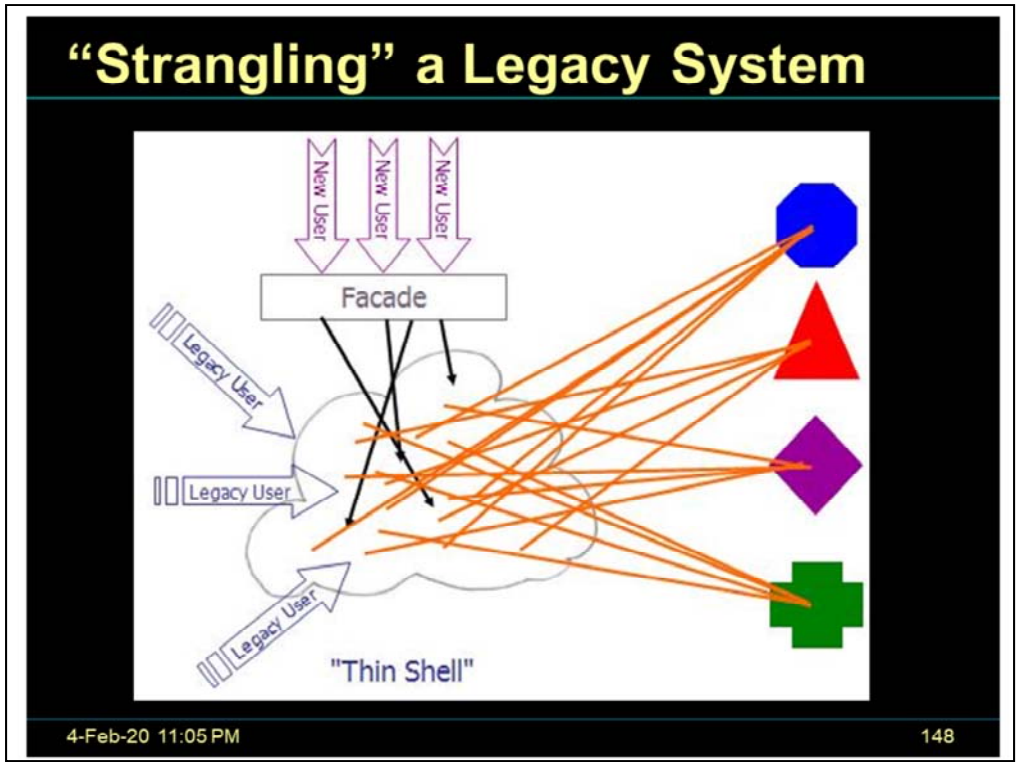


“Strangling” a Legacy System



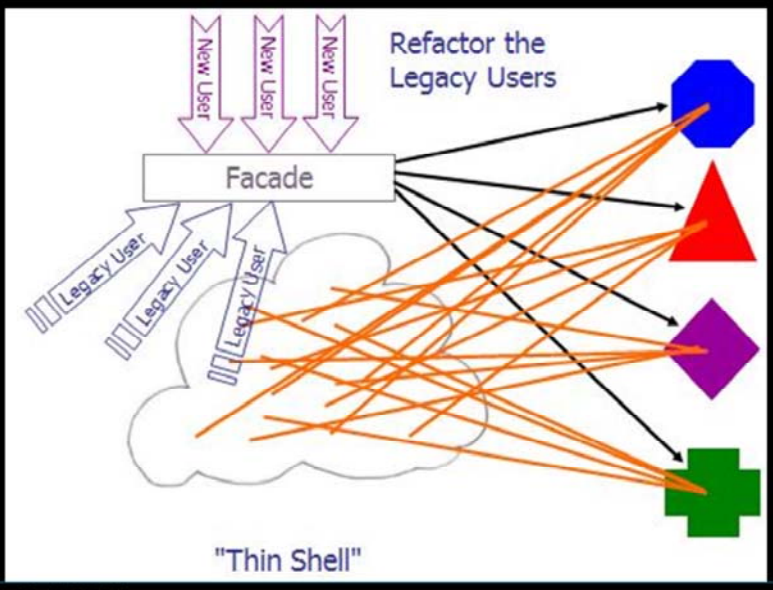
“Strangling” a Legacy System

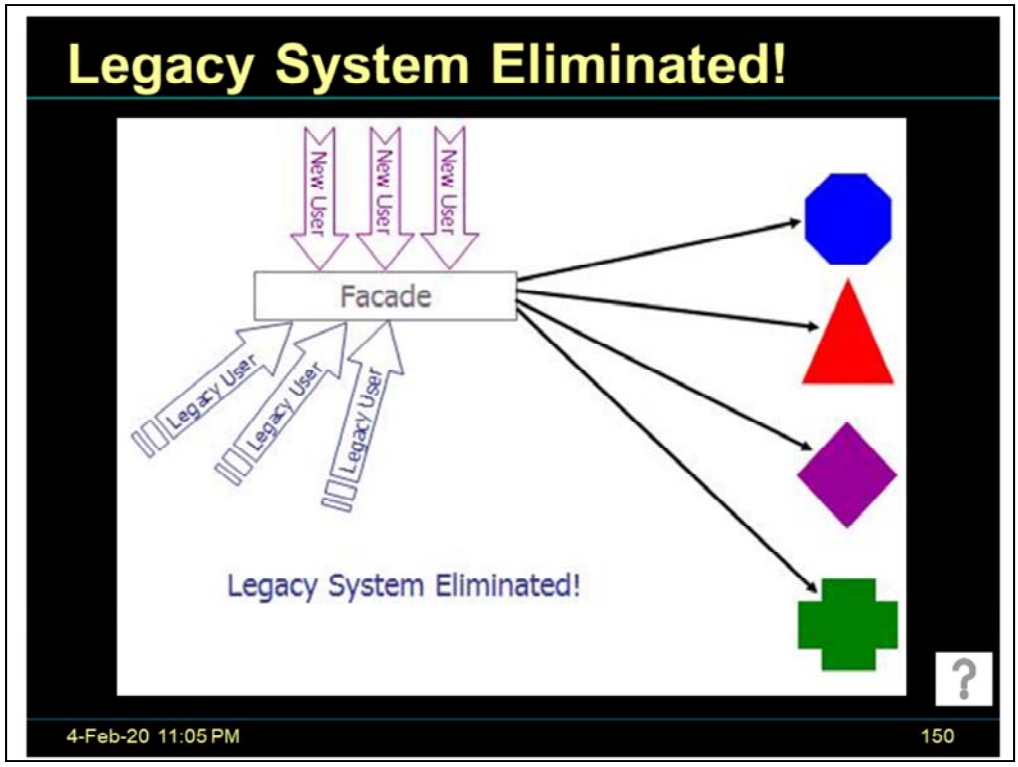




Once we reach the point that the legacy system remains only as a series of delegation points, nothing but a thin shell around the new behaviors, we can begin to refactor the legacy users to use the Façade instead of the delegating routines in the older system. The Façade itself can also be refactored to use the new code directly. Again, this can be done incrementally, as time permits, without a lot of pressure. Naturally, once all the legacy users and the Façade no longer use the thin-shell legacy system, it can be retired.

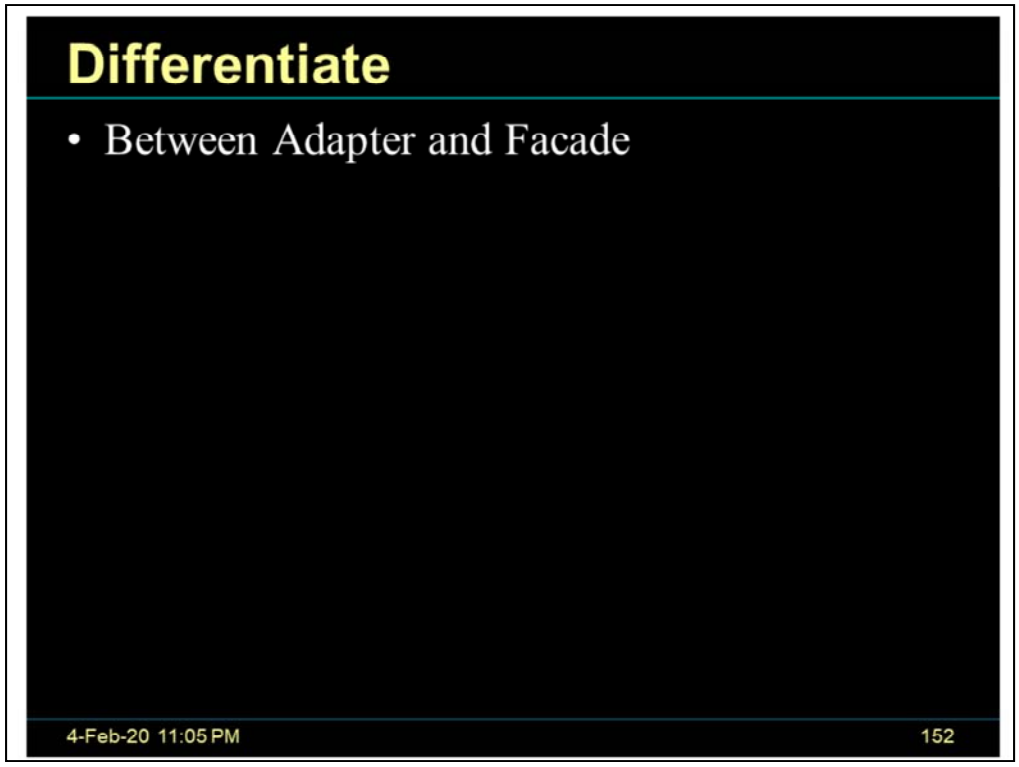
“Strangling” a Legacy System




Legacy System Eliminated!
New User
New User
New User
Facade
Legacy User
Legacy User
Legacy User
Legacy System Eliminated!
?
4-Feb-20 11:05 PM
150

Usage of Facade

- The packaging features in Java attempt to accomplish the Facade interface.
  - Outside of the package, we can only create and use **public** classes
  - All the non-**public** classes are only accessible within the package.
- Facade's is used in libraries

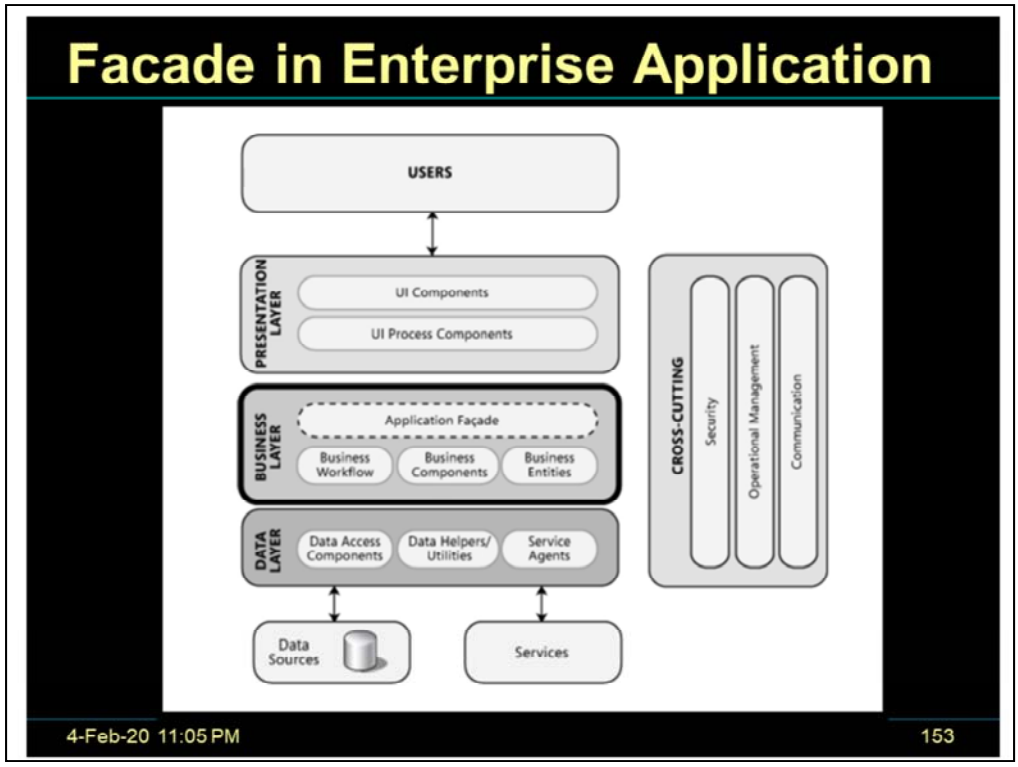


Facade defines a new interface, whereas Adapter reuses an old interface.

The main difference between Facade and Adapter is the INTENT.

> When we need to use an existing class an its interface is not the one we need, we use an adapter.
>
> When we need to simplify and unify a large interface or complex set of interfaces, we use a Facade.
>
> > By this definition, WindowAdapter, etc classes in Java are actually a Facade and not Adapters.

The Adapter pattern is often confused with the Facade pattern [DP]. Both patterns make code easier to use, yet each operates on different levels: Adapters adapt objects, while Facades adapt entire subsystems.



Application Facade provides a coarse-grained interface to fine grained Business domain classes.

==========

Imagine we are working on an application with a 3-tiered architecture (UI, Middle, Data), and we want to work in a test-driven way. Unfortunately, our teams are all blocked:

Our teams are blocked:

UI Tier – I can't start until the middle tier is done, they provide the interface for the services I will consume.

Middle Tier – I can't start until the data-layer is done, because I need to access it in order to provide services to the UI

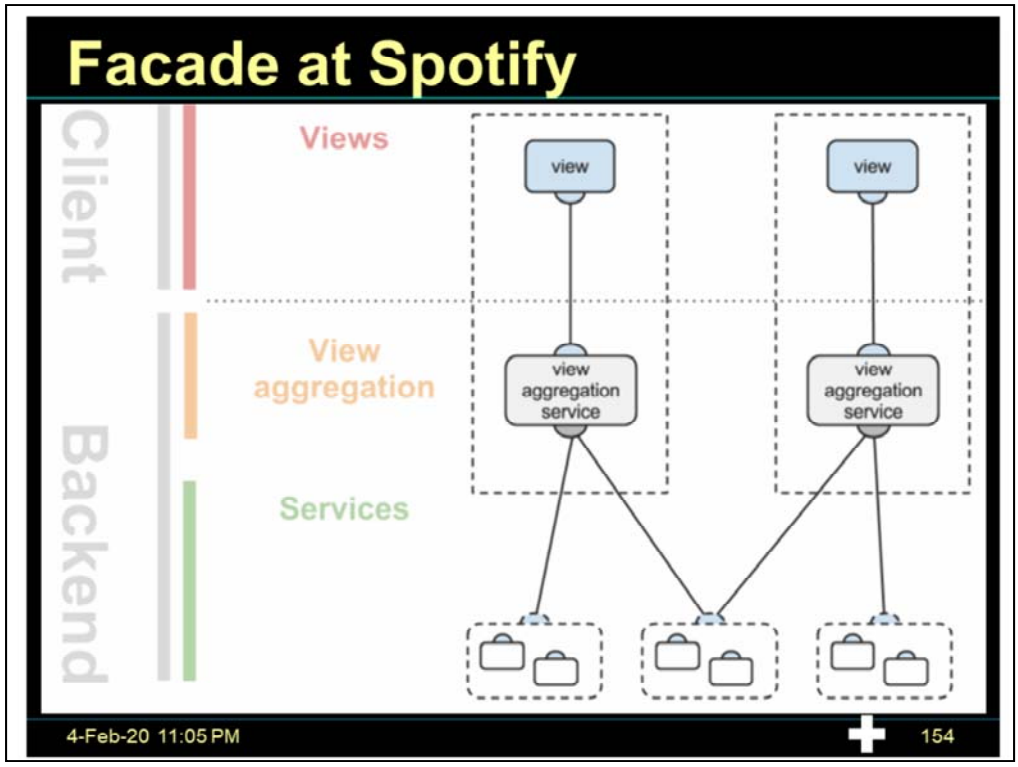
Data Layer – I can't start until I know what the data needs are. Also, I am involved in day-to-day business, which must always take priority over new development. I'll get to it when I get to it.

We can use Façades to separate these layers. The Façades will present the interface of the consuming layer, and later can be coded to the implementation of the service layer in each case, when this becomes available (it can be stubbed out, initially).

Also, each Façade will be a mockable entity, and so the layers can be test-driven:

==========

OOAD slides that can be used: Encapsulation, Compare (getEmployees), Null Pattern, Guideline(Don’t return a string a client has to parse), Compare (Junit with JUnit4), Improve (Varagrs / Param), Compare (min), Java Date and Time, Problems in Java API, Principle of least Astonishment, Find the Flaw (checkpassword), Guidelines(keep command and queries segregated), Law of Demeter violated, Reduce coupling, Rule (one-dot per line), Guidelines(An interface should be designed so that it is easy to use and difficult to misue) + Improve code + example. Tell don’t ask, Avoid getters and setters, Improve (getStatus), Example (myThing), Interface Segregation Principle, ISP implemented, ISP violated, ISP implemented, ISP violated, ISP implemented,



Earlier, at Spotify (a music service company), different views (mobiles, desktop, game boxes) used to make multiple call to different microservices.

Now, with Facade, only one call is needed between view and Facade for most interactions. The Facade makes multiple calls to different microservices.

This reduces latency and lowers coupling for the view.

Exercise - Facade Pattern

- Take a package like java.util.stream
  - How many interfaces/abstract classes?
  - How many classes that are final / non-final?
  - How many factories?