Clean Code design patterns 2

Sebastian Lindgren

Välkomna till dagens föreläsning!

Clean Code - design patterns 2!

Dagens agenda

- Repetition av design patterns
- Fler design patterns
  - Facade
  - Proxy
  - Observer
  - Builder
- Övningsuppgifter

Repetition

- Vad är ett design pattern?
- Varför vill vi använda design patterns?
- Vad är definitionen för design patterns?

Repetition

Definitionen av design patterns:

A
named
pedagogical
solution
to a
recurring
problem
in a
certain context

This one is easy to forget

Figure 1: Definition av Design Pattern

Repetition

Vad minns ni om följande design patterns?

- Iterator
- Injection
- Singleton
- Factory

Facade

När man vill komma åt ett komplext system med ett mer lättanvänt interface.

The Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Facade

- Sometimes it is a good idea to hide the complexity of a complex system
- The complex system will often be "legacy code", code that has gathered complexity and code smells over years
- The code provides a needed service
- It is not feasible to rewrite it, for many reasons

Utan Facade

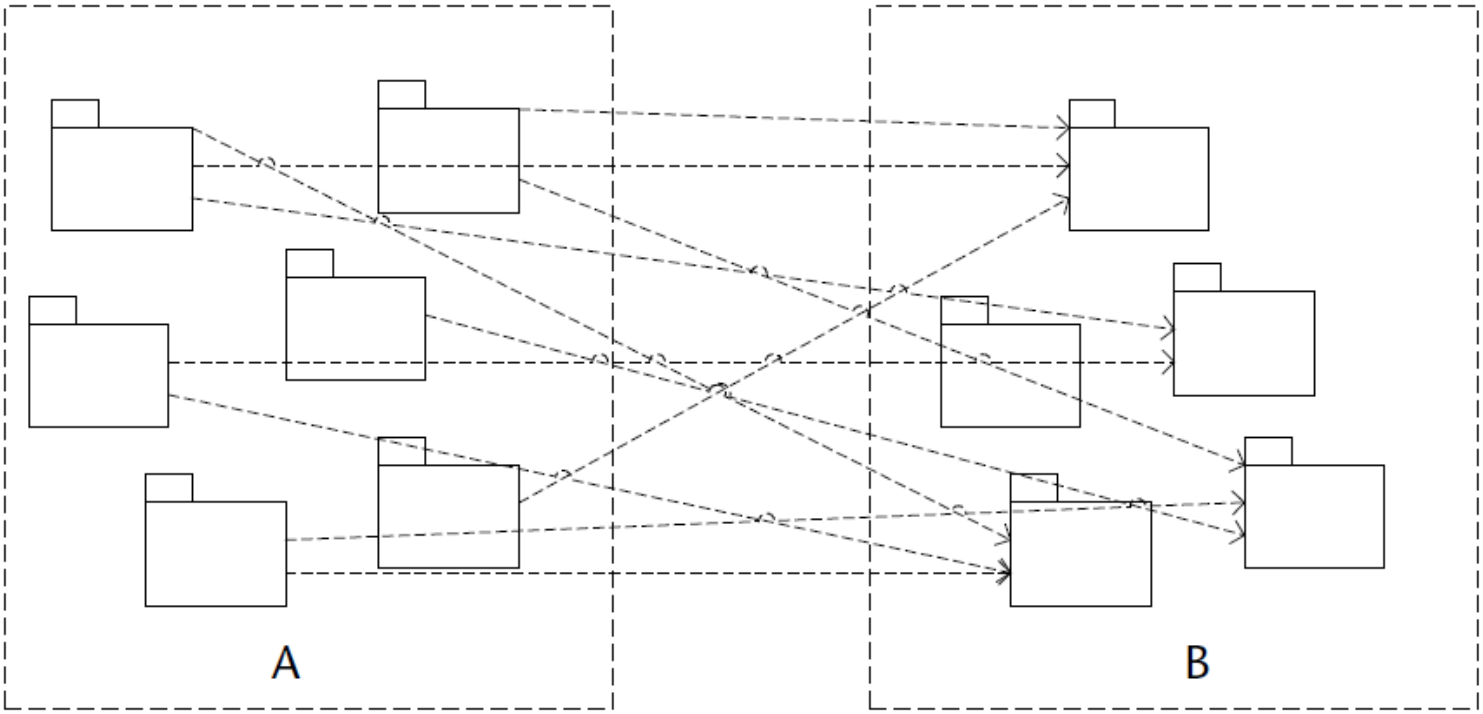


Figure 2: Många dependencies, väldigt svårt att komma runt detta!

Med Facade

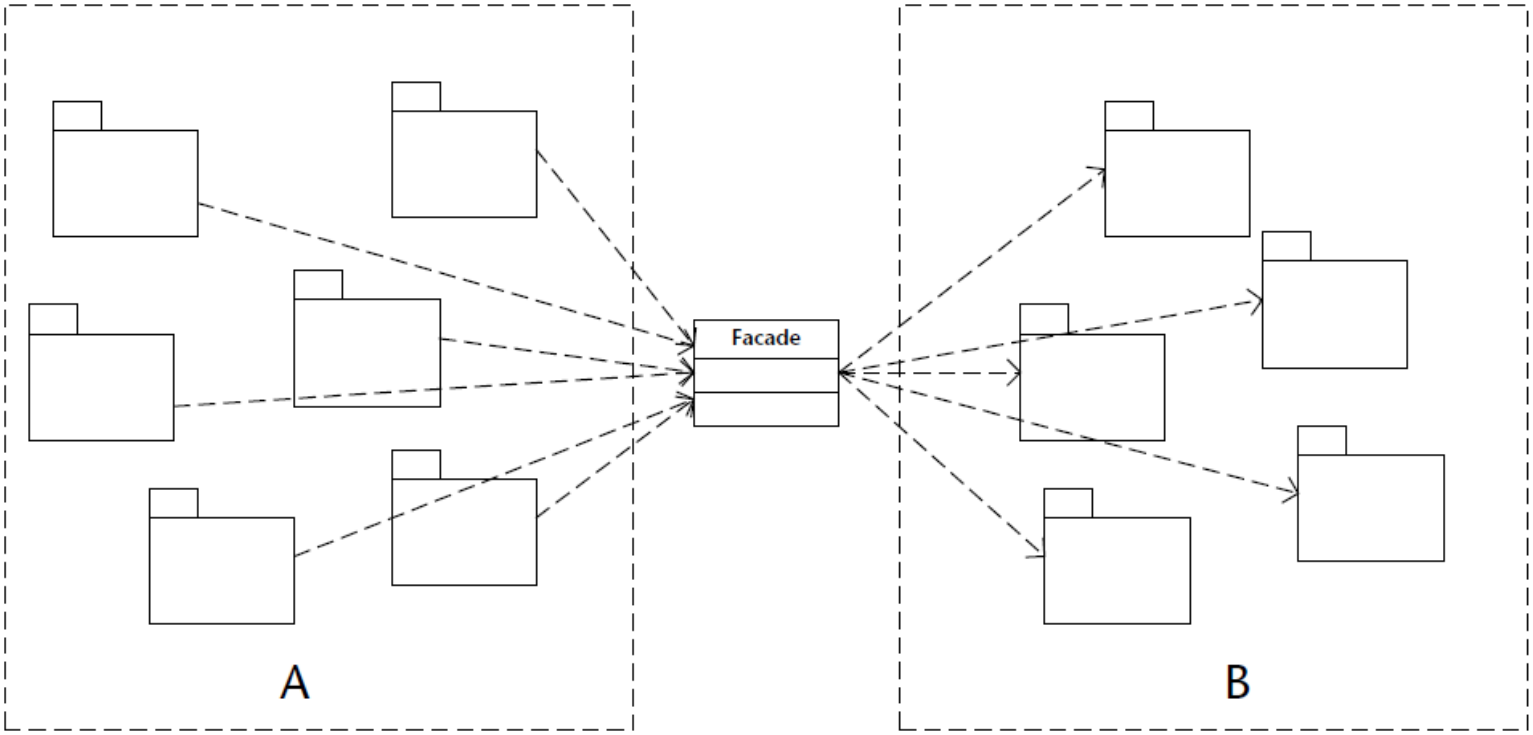


Figure 3: Massor av arbete att konvertera hela A till att använda det förenklade Facade interfacet. Vad händer om vårt Facade interface inte är väl designat?

Facade applicability

Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.

Hur man implementerar

1. Check whether it's possible to provide a simpler interface than what an existing subsystem already provides. You're on the right track if this interface makes the client code independent from many of the subsystem's classes.
2. Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.

Hur man implementerar

3. To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code. For example, when a subsystem gets upgraded to a new version, you will only need to modify the code in the facade.
4. If the facade becomes too big, consider extracting part of its behavior to a new, refined facade class.

Facade sammanfattning

- The idea of the pattern is trivial to understand
- Solves a chaotic situation, requiring lots of invested work
- Great risk that we do not pick the optimal interface
- In reality: can be extremely hard to implement!

Proxy - ett “stand-in” objekt

- Sometimes it is a good idea not to talk directly to an object, but do it via a stand-in object – a Proxy
  - When we talk to a remote object over a network, we talk to a remote proxy (since it is impossible to have a reference across a network!)
  - We look a thumb-nail in a photoalbum, i.e. a lazy proxy (that delays the creation of the real object until necessary)
  - We may talk to a security proxy (that limits our access to the real object until we have authorised ourselves)
- In all cases, the Proxy has the same interface as the Real Object, so the user does not need to know that it is talking to a Proxy

Lazy Proxy exempel

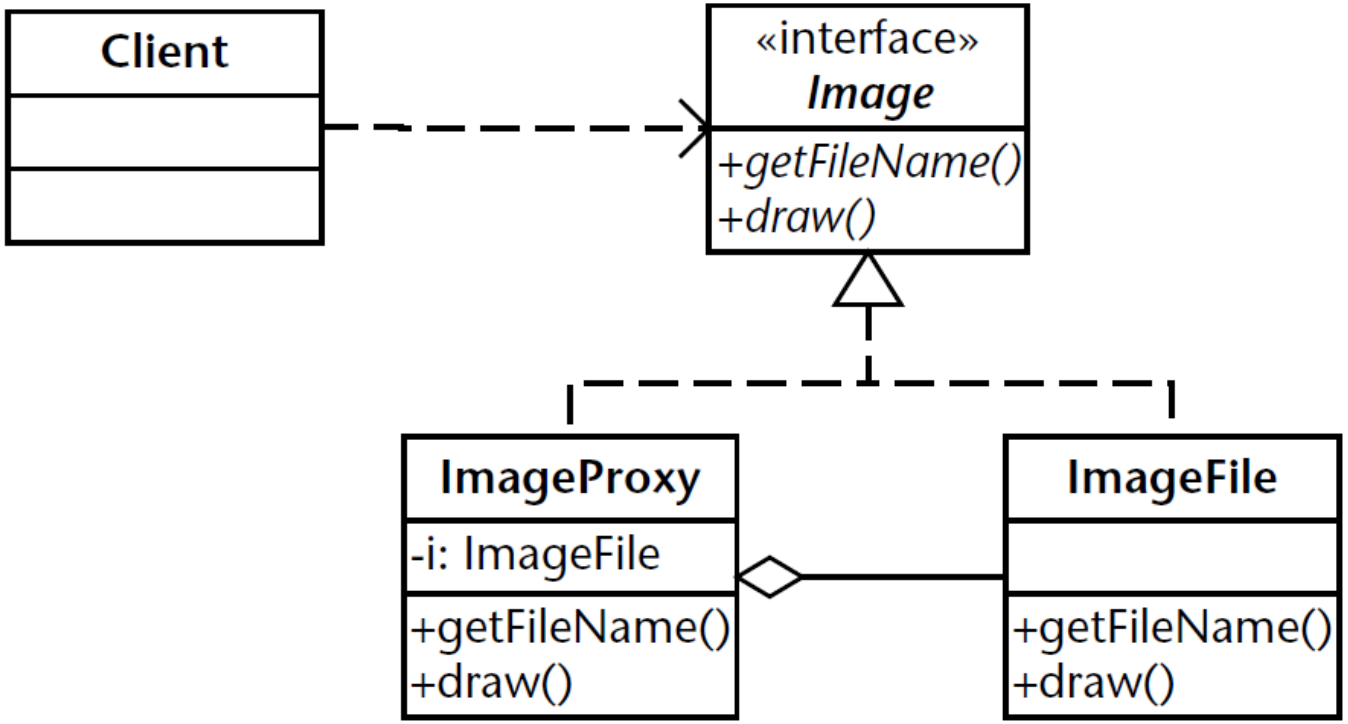


Figure 4: ImageProxyn innehåller allt innehåll som inte tar mycket plats. Den undviker att hämta/skapa en ImageFile tills vi faktiskt behöver alla 50Mpixels.

Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Real world analogy

If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.

The publisher maintains a list of subscribers and knows which magazines they're interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

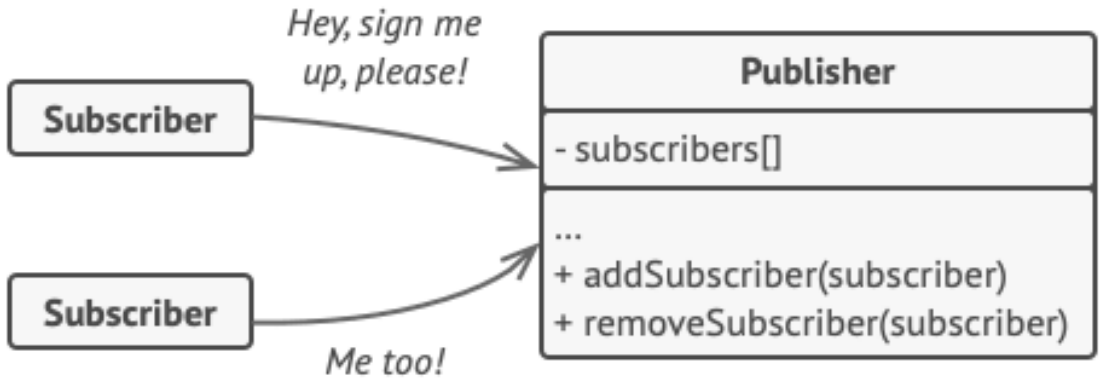
Observer solution 1/2

The object that has some interesting state is often called subject, but since it's also going to notify other objects about the changes to its state, we'll call it publisher. All other objects that want to track changes to the publisher's state are called subscribers.

Observer solution 2/2

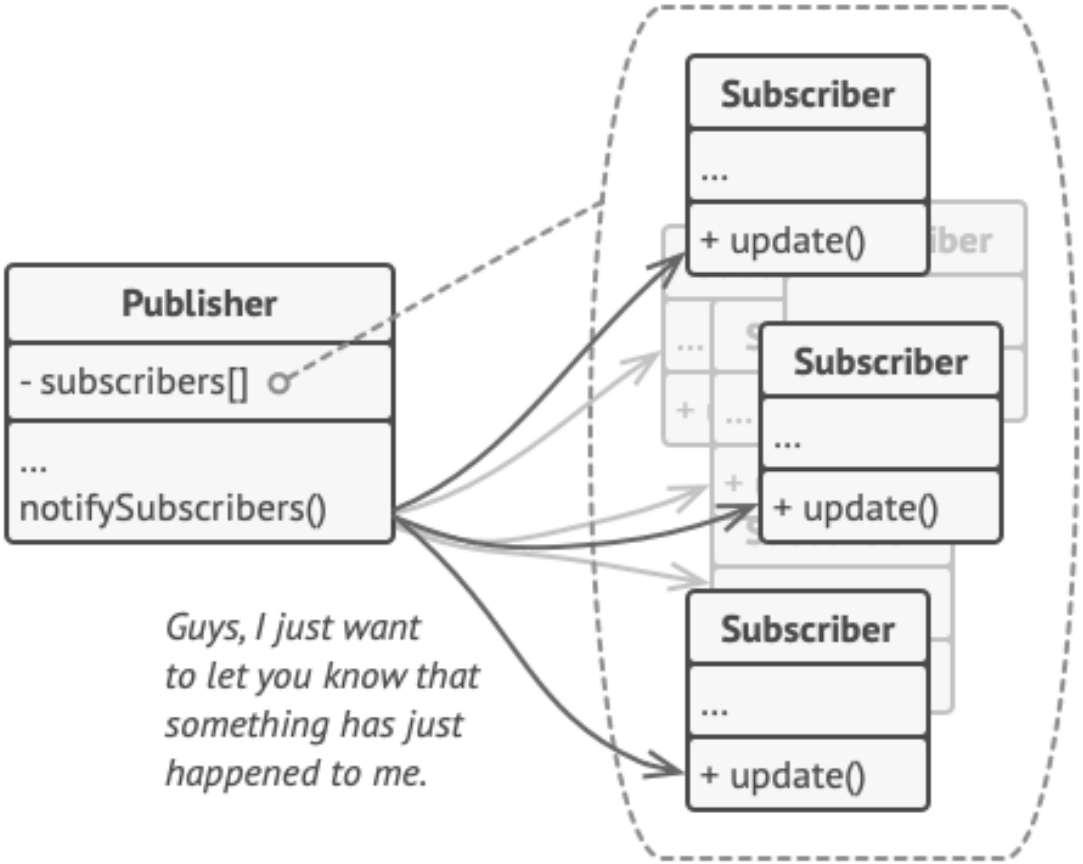
The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. Fear not! Everything isn't as complicated as it sounds. In reality, this mechanism consists of 1) an array field for storing a list of references to subscriber objects and 2) several public methods which allow adding subscribers to and removing them from that list.

Observer



A subscription mechanism lets individual objects subscribe to event notifications.

Observer



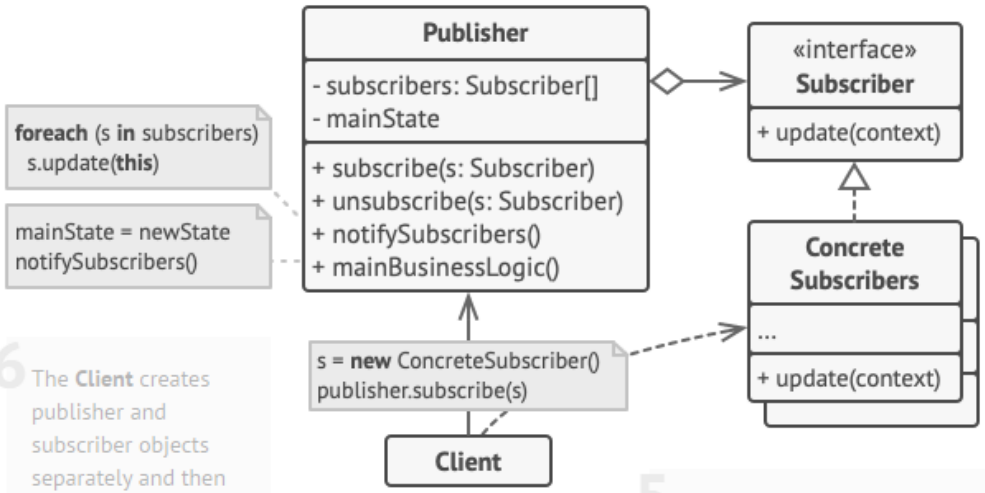
Publisher notifies subscribers by calling the specific notification method on their objects.

Observer

1 The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

2 When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.

3 The **Subscriber** interface declares the notification interface. In most cases, it consists of a single `update` method. The method may have several parameters that let the publisher pass some event details along with the update.



4 **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

6 The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

Observer

Du kan se en observer såsom att det vore ett kontinuerligt promise.

Builder

Med builder pattern så kan vi skapa mer komplexa objekt steg för steg.

Builder - Problemet

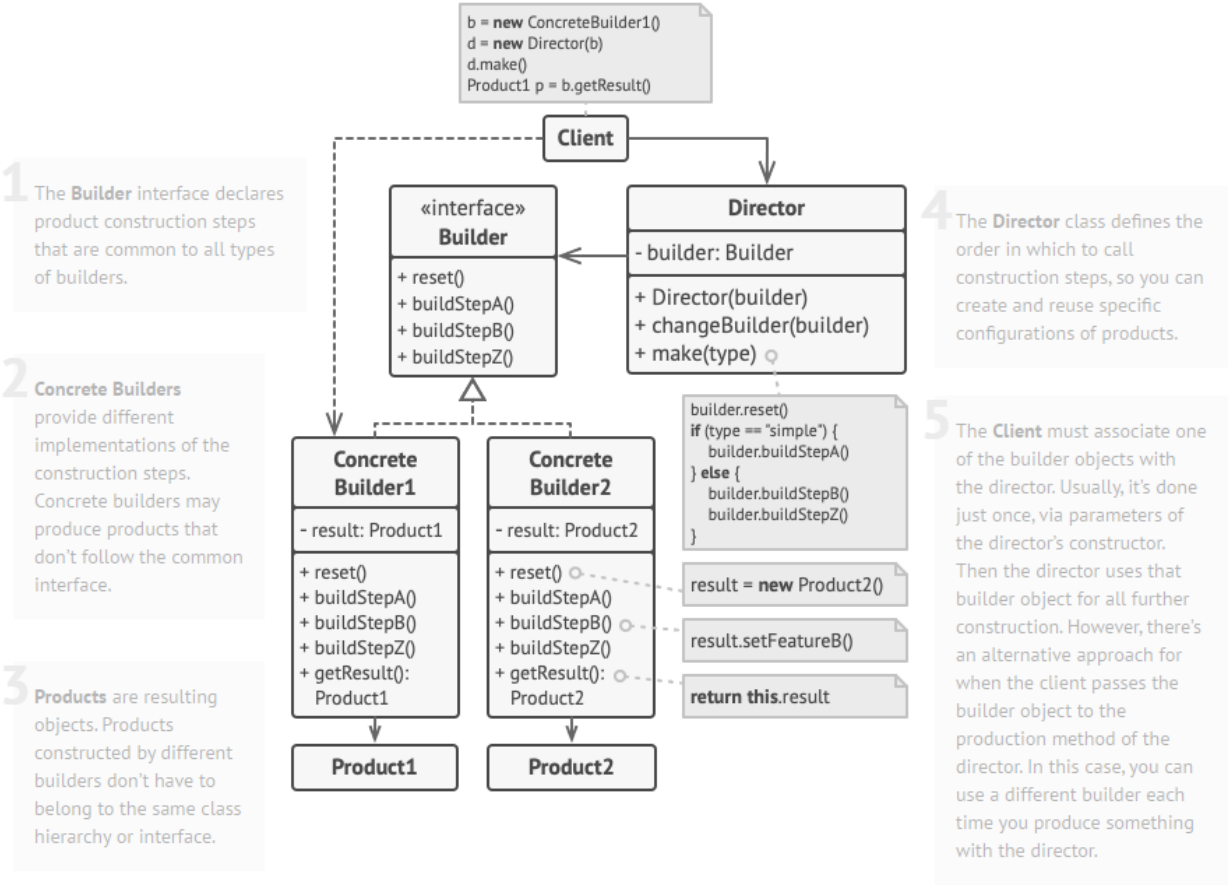
Tänk er att vi skapar ett objekt för “Hus”. Detta objekt har t.ex. 7 olika paramterar i sin konstruktor. Ofta kommer många av dessa sättas till något “null” eller “tomt värde”. Detta kan göra våra konstruktorkallerser ganska fula.

Builder - Lösningen

Man delar upp uppbyggandet av sitt objekt i olika builder-steg. Det viktiga är att man inte behöver köra alla steg utan bara det eller de steg som behövs för att bygga upp det objektet du vill ha. Man kan sedan skapa olika builder-klasser som använder sig av dessa builder steg på lite olika sätt, i Hus analogin kan man tänka sig att en Builder klass bygger ett trähus och en annan ett slott.

Man kan även gå ett steg längre och använda sig av en Director klass som ger ett enkelt interface för klienten och som kallar på olika builders. Builders står i sin tur för implementationen av själva buildandet.

Builder översikt



Builder - slutord

Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, but more complicated).

Builder focuses on constructing complex objects step by step. Abstract Factory specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product.

You can use Builder when creating complex Composite trees because you can program its construction steps to work recursively.

Teoretisk uppgift

Läs på lite mer om de olika design patterns vi har gått igenom hittills och fler om du hittar några du tycker verkar intressanta.

https://refactoring.guru/design-patterns/catalog

Extra

Testa att använda några av de olika design patterns du läser mer om praktiskt i den föregående uppgiften eller i egna små exempel!