Agenda
Repetition
Design patterns
Iterator
Injection
Singleton
Factory
Summary

# Clean Code design patterns del 1

Sebastian Lindgren

# Välkomna till dagens föreläsning!



*Clean Code - design patterns!*

# Dagens agenda



- Utdelning av laborationen
- Repetition OOPs centrala delar och skalbarhet
- Introduktion till design patterns
- Design Patterns genomgång del 1
  - Iterator
  - Injection
  - Singleton
  - Factory

# Repetition 1/2: OOP centrala delar



Vi ser vad ni minns om OOPs centrala delar ifrån senaste tillfället!

- Vilka är de centrala delarna i OOP?

# Repetition 1/2: OOP centrala delar



Vi ser vad ni minns om OOPs centrala delar ifrån senaste tillfället!

- Vilka är de centrala delarna i OOP?
  - Abstraktion
  - Inkapsling
  - Arv
  - Polymorfism

*Vad mer vet vi om dessa?*

# Repetition 1/2: OOP centrala delar



Vi ser vad ni minns om OOPs centrala delar ifrån senaste tillfället!

- Vilka är de centrala delarna i OOP?
  - Abstraktion
    - Separationen av och fokuset på interfaces hos en klass och inte implementationen
    - Vi delar upp vår logik i interface och faktiskt utförande i klasser
  - Inkapsling
    - Ha data och funktioner kopplade till en entitet
    - Bara visa de detaljer som är viktiga att interagera med
  - Arv
    - Hierarkier där klasser bygger på andra klasser
  - Polymorfism
    - Enhetlig behandling av klasser i en hierarki. Så länge man kan hantera objekt i roten av hierarkin kan man hantera objekt som kommer senare också.

# Repetition 2/2: skalbarhet



Vi ser vad ni minns om skalbarhet ifrån senaste tillfället!

- När vi pratar om skalbarhet pratar vi om hur enkelt vi kan. . . ?
- Vad är det vi vill jobba mot när vi jobbar med skalbarhet?
- Vilken faktor är också bra att ta hänsyn till?

# Repetition 2/2: skalbarhet



Vi ser vad ni minns om skalbarhet ifrån senaste tillfället!

- Vad är det vi pratar om när vi pratar om skalbarhet?
  - Med skalbarhet i kod pratar vi om hur enkelt vi kan *anpassa oss till nya behov/förändring*.
- En fråga man kan ställa sig är: *om vi skulle få ett nytt behov, hur snabbt kan vi förändra koden så att det nya behovet går att tillfredställa?*
  - Ju färre nya rader kod som skulle behövas. Desto bättre!
- En faktor som också är bra att ta i hänsyn är hur många *repeating parts* man har.
  - Vi vill hellre bygga strukturer som hanterar repetition än att repetera kod.

# Design patterns



Nu ska vi gå in i vad design patterns är och varför vi vill använda dem.

# Design patterns



När vi använder design patterns så handlar det om att vi strukturerat vill använda oss av experters kunskaper och erfarenheter.

# Gang-of-Four boken



Det hela började med att fyra programmeringsforskare publiserade en bok om design patterns år 1995 kallad: Design Patterns: Elements of Reusable Object-Oriented Software

- Boken gav en bra startpunkt för en beskrivning av bra och återanvändbara idéer inom programmering.
- Det har funnits bra idéer långt före detta men dessa flöt mest bara omkring inom olika kretsar och på internet.
- GOF gav en taxonomi (i.e. studien om klassifikationsprinciper) för programming design patterns.
- När en duktig taxonomist har gjort sitt jobb bra så ser det väldigt obvious och enkelt ut (såsom: det här skulle jag kunnat komma på egen hand).

# DP - en definition



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

This one is easy to forget

Figure 1: Definition av Design Pattern

# Named



- A pattern needs a good name, GoF chose their names very carefully
- The name should describe the intent of the pattern
- The name will facilitate design communication: "I think a Visitor would do the job here!"
- "Model 2" is not a good name
- "Wrapper" only talks about structure, not intent
- "Adapter" and "Iterator" does!

# pedagogical



- The whole point of a pattern is that it should be reused
- A pattern should inspire the user
- It needs a description that any user can recognise, so that is solves a problem that a user has experienced or will experience
- All pattern descriptions tend to start with a concrete situation, rather than an abstract description

# solution



- The obvious part
  - There are usually wide variations
  - There is no "correct", "kosher" or "canonical" version of a pattern
  - A pattern is not a "fixed plug-in", avoid-thinking idea
  - It is a starting point, but from a view standing on the shoulders of the pattern description

# recurring problem



- The problem should not be a once-off, obviously.
- The problem description should easily trigger thoughts about similar situations, i.e. similar code smells
- There also is a general when-to-use description

# certain context



- A pattern is not good by nature
- It has a benefit and a cost
- There are situations when a pattern should not be used – a pattern gives a clear description of this.
- Don't stop reading before you get to that part
- This is the easiest part of a pattern to forget!

# Patterns are very different



- Patterns vary in many dimensions:
  - Some are used all the time, while some more seldom
  - Some are general, some very specific
  - Some are easy, some difficult to understand
  - Some have to do with object creation, some with data structure, some with behaviour.
  - Some are specialised for certain frameworks

# Patterns, code smells and refactoring

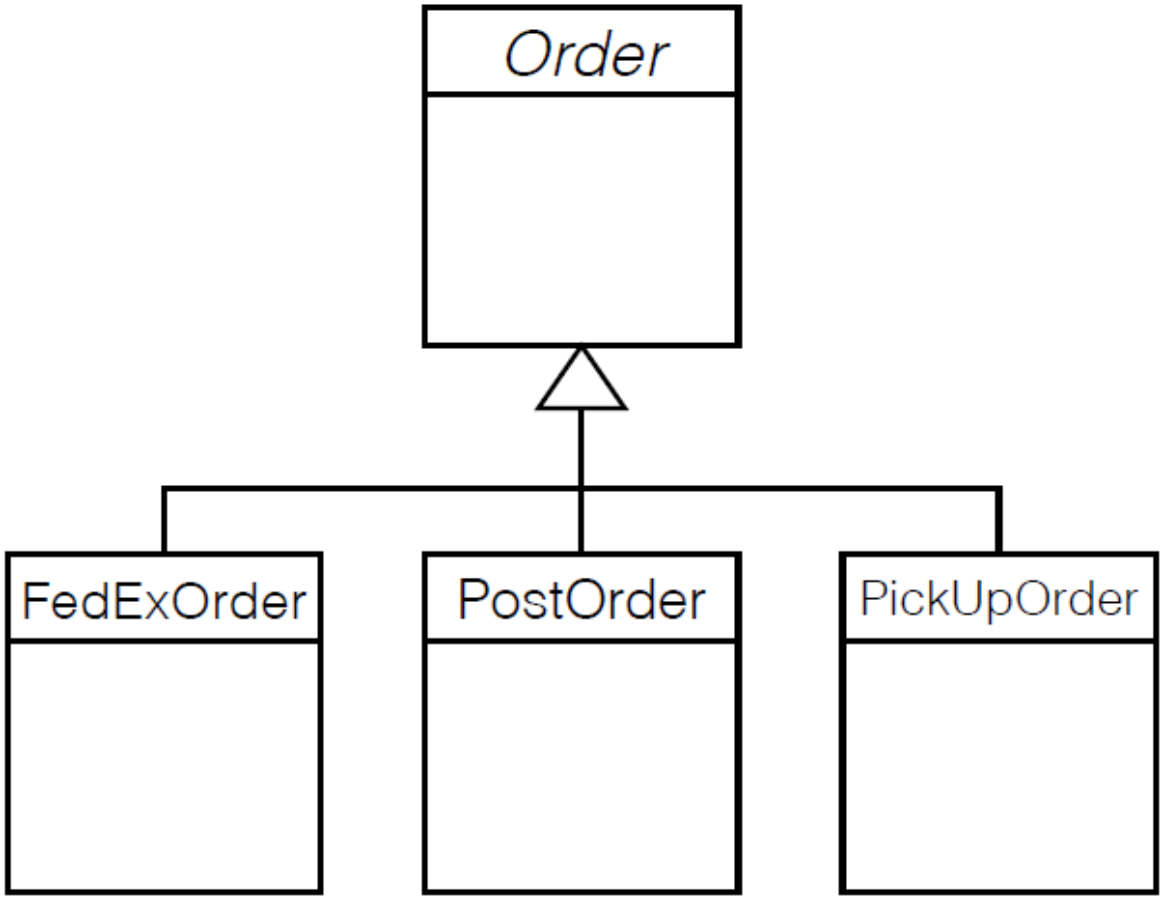


- A correctly behaving program that could be improved without change of functionality has a "code smell"
- Code smell examples: bad naming, duplicated code, bad usage of inheritance, too large method or class, too many comments, too few comments
- Refactoring is the process of removing code smells, without change of functionality
- The goal of a refactoring can often be a design pattern

# Example: bad inheritance



What is wrong with this?

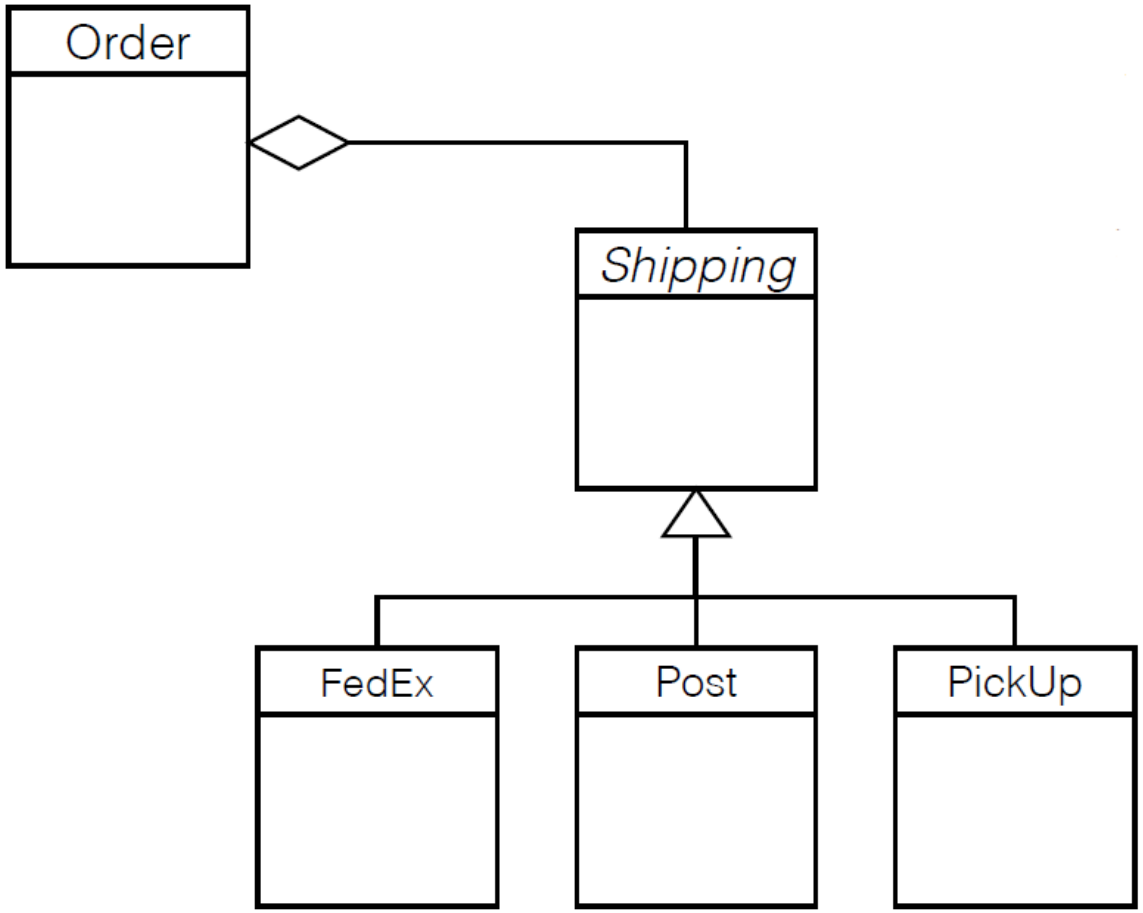


# Refactoring DP Strategy



Encapsulate partial behaviour in a replaceable object

The varying shipping behaviour is now in a replaceable object



# DP Strategy



Ett till exempel:

The varying comparison behaviour is encapsulated in a replaceable object

```
List<String> list =
      new List<String>();
// put stuff in the list

list.Sort(list, new StringLengthComparer());
// or
list.Sort(list, (s1,s2) => s1.Length-s2.Length);
```

# Iterator (C#: IEnumerator)



- We would like to go through the elements of a collection of objects
- We don't know the structure of the collection, so we don't know the best way to step through it (and we really don't want to know).
- Let a specialised object do the stepping
- Let's give the idea a very good name: Iterator
- Let's reuse the idea every time we need it

# Patterns are language neutral ideas



## Java

```
Iterator<Object> it = myList.iterator();
while (it.hasNext()) {
Object o = it.next();
//...
}
```

## C#

```
IEnumerator enumerator = myList.GetEnumerator();
while (enumerator.MoveNext())
{
object item = enumerator.Current;
// ...
}
```

# Patterns are language neutral ideas



C++

```
list<Thing>::iterator iter = mylist.begin();
while (iter != mylist.end()) {
Thing t = *iter++ ;
//...
}
```

# Dependency Injection



- This is a Design Pattern that favours loose coupling between objects, particularly at creation time. (It is also called Inversion of control – IoC)
- Many .NET frameworks is using DI
  - Basically:
    - do not let an object create its own dependent/needed objects and connect to them
    - but: let some "external magic" create all objects and connect them.

# Inte dependency injection



Utan dependency injections har vi ingen möjlighet att lägga till extra, och ingen möjlighet att välja vilken Service implementation vi vill använda.

```
public class User
{

   IService s;
   //...

   public User()
{
      //...
      s = new ServiceImpl();
      //..
   }

   //...
}
```

```
public interface IService
{
   //useful interface

}
```

```
public class ServiceImpl
           : IService
{
   //useful implementation

}
```

# Med dependency injection (property injection)



```
public class User
{
   public IService {get; set;}
   //...
   public User()
   {
      //...
   }


}
```

```
public interface IService
{
   //useful interface

}
```

```
public class ServiceImpl
                  : IService
{
   //useful implementation

}
```

# Med dependency injection (property injection)



Some "external" will now create the User and ServiceImpl objects, and call the setService method to "glue" them together. The whole point is that the User does not itself create/connect the Service it needs!

```
User u = new User();
IService s = new ServiceImpl();
u.Service = s;
// start using u
```

# Med dependency injection (constructor injection)



```
public class User {
   Service s;
   //...
   public User(IService s)
   {
      this.s = s;
      //...
   }


   //...
}
```

```
public interface IService {
   //useful interface

}
```

```
public class ServiceImpl
                 : IService
{
   //useful implementation

}
```

# Med dependency injection (constructor injection)



Instead of a set method, the constructor can be used to do the injection. The whole point is that the User does not itself create the Service it needs!

```
Service s = new ServiceImpl();
User u = new User(s);
// start using u
```

# Dependency Injection (annat)



```
public class User {

   Service s;
   //...

   public User(){
      //...
   }

   //...
}
```

```
public interface Service {
   //useful interface

}
```

```
public class ServiceImpl
            implements Service{
   //useful implementation

}
```

# Dependency Injection (annat)



Injection can also happen with an outside mechanism, knowing dynamically by name that User needs an IService, or it can know this from info in a configuration file.

**The whole point is that the User itself does not create the Service it needs!**

# Dependency Injection – why?



- Testing the User gets easier, it can be isolated with dependency (a MockService) that the test needs, rather than production dependency
- The injection framework may pool the dependencies
- The injection framework may add extra functionality "on the way" from User to Service framework, e.g. transaction behaviour.

# Dependency Injection – for everything and everywhere?



- No, but
- Very likely across layer boundaries
- When a mock implementation needs to replace the real one for testing
- When a framework needs to dynamically insert functionality "on the way", e.g. transaction behaviour.
- When a framework needs to pool resources
- "Should I create my needed object here or is it better that someone else is doing it for me?"

# Singleton



- Sometimes we would like to have only one single object of a class, e.g.
- UniqueIDGenerator
- ConnectionManager
- It seems like a reasonable idea.
- However...

# Singleton



- A Singleton object is reachable and used globally
- We do not want many of these objects
- So: all ideas usually have benefits vs. costs
- But if we need one, let's implement it properly
- Easy?

# Singleton implementation



```
// Please, create only one of these
public class QueueTicketDispenser {
   private int nr;

   public QueueTicketDispenser() {
      nr = 1;
   }

   public int GetNextNumber() {
      return nr++;
   }
}
```

# Singleton implementation



```
public class QueueTicketDispenser {

   static private int objectCount = 0;
   private int nr;

   public QueueTicketDispenser() {
      if (objectCount > 0)
         throw new SingletonException();
      nr = 1;
      objectCount++;
   }

   public int GetNextNumber() {
      return nr++;
   }
}
```

# Singleton diskussion



- How many variants, bad and good, did we just see?
- Many of the improvement and pitfalls have been discovered after GoF included Singleton in their book.
- Would the research process have happened without them?
- Conclusion: Every design pattern becomes a research topic!

Det finns fler varianter av Singleton som ni kan läsa mer om här [2].

# Factory



Factory method – not saying new Xxxx(...) - Sometimes it is a bad idea to use new to create an object - We don't know what the class name should be - We don't want to know what the class name should be - We don't want to know whether we can use a secondhand recycled object or need a new one

# Factory



```
Person p = new Person(18);
// not Factory: this creates a Person with NO CHOICE!

Person p = PersonFactory.Create(18);
// create(int age) knows which kind of Person to create

Greeting g = GreetingFactory.Make();
// different kind of Greetings due to time, date, Locale etc.

IEnumerator<String> en = myList.GetEnumerator();
// creates the perfect Enumerator/Iterator for myList, .NET

Connection c = ConnectionManager.connect("DB86Plus");
// use a recycled or shared connection, user does not need to know
```

# Design Patterns conclusions



DP – some conclusions

- Some patterns are almost trivial, some require study
- They do not guarantee that anything gets better, but there is a very good chance
- Each pattern tends to become a research topic, i.e. scrutinised and improved by the community of programmers

# Summary



- Patterns are different in many ways
- Easy – hard to understand
- Easy – hard to implement
- Very common – rare in usage
- General – very specific
- In the GoF book – not in the book
- You don't have to remember everything about a pattern, just a faint whiff of a code smell should trigger you to look for a solution, maybe with a pattern as a goal.

# Hur man blir en duktig programmerare



A very good way to become a better programmer is to study the experience, knowledge and mistakes of others – in a structured way. See gärna mer här: [1].

# Nästa gång



Nästa gång pratar vi mer om design patterns och bland annat om observer, builder m.fl.

[1] Administrator. URL: https://hillside.net/patterns/.

[2] Jon Skeet. *Implementing Singleton*. URL: https://csharpindepth.com/articles/singleton.