Clean Code introduktion

Sebastian Lindgren

Välkomna till dagens föreläsning!

Clean Code - en introduktion!

Dagens agenda

- Välkomna!
  - Vilka är era utbildare?
- Vad ska vi gå igenom i den här kursen?
- Hur kommer kursen att gå till?
- Introduktion till Clean Code

Välkomna

Välkomna till kursen!

Utbildare

Föreläsare: Sebastian Lindgren

Handledare: Benjamin Österlund

Planering

Låt oss titta lite närmare på vad vi ska gå igenom i kursen mellan v22-24 och 34-36

Vecka 22

Denna vecka har vi en introduktion till kursen och till clean code som ämne. Vi kommer också att gå igenom lite repetition av teoretiska koncept inom objektorienterad programmering och hur vi får till skalbar kod. Vi kommer börja titta lite på mönster.

Vecka 23

Vecka 23 går vi igenom vad patterns är och kikar närmare på flera olika design patterns. Bland annat Singleton, Facade, Proxy, Factory, Iterator... Under veckan kommer ni också att få utdelad eran laboration.

Vecka 24

Vecka 24 tittar vi på TDD - testdriven utveckling (*eng: test driven development*). Vi kommer även att titta på BDD - beteendedriven utveckling (*eng: behavior driven development*)

Vecka 34

Vecka 34 tittar vi på patterns och anti-patterns. Hur kan vi komma undan problem med hjälp av patterns och hur kan vi undvika anti-patterns. Vi börjar även utforska SOA - Serviceorienterad arkitektur (*eng: service oriented architecture*) och Microservices.

Vecka 35

Vecka 35 fortsätter vi med serviceorienterad arkitektur och microservices. Vi tittar även lite på CI/CD i Azure.

Vecka 36

Vecka 36 blir det främst extra handledningstid och möjlighet att öva på det ni har lärt er. Ni kommer lämna in och redovisa era laborationer och skriva en tentamen.

Tillvägagångssätt

Kursen kommer att använda sig av ITHSdistans där allt material kommer att finnas tillgängligt.

- ITHS för på-platsgruppen [1]
- ITHS för distansgruppen [2]

Det kommer att finnas en del uppgifter som ges under föreläsningstid.

Tillvägagångssätt

Tillvägagångssätt för kunskapsevaluering:

- 1 laboration
- 1 tentamen

Laborationen och tentamen kommer att göras tillgängliga på ITHSdistans och även lämnas in på ITHSdistans.

Laboration

Laborationen kommer att handla om att med principer ni lär er inom clean code, testing och design patterns ta en fungerande men “smelly” applikation och förbättra den.

Ett labbdokument med mer specifik information om detaljerna kring laborationen kommer under nästa vecka.

Tentamen

Er tentamen kommer att ta upp de olika delarna ni lär er under kursens gång. I tentamen kommer ni att få ett antal flervals och skrivuppgifter om:

- clean code
- design patterns
- TDD och BDD
- SOA och Microservices
- CI/CD

Tentamen

De olika frågorna kommer för godkänt att handla mer eller mindre direkt om det vi går igenom och lär oss under föreläsningarna och uppgifterna. För väl godkänt kommer det vara samma ämnen men att det hjälper om ni läser på om ämnena i flera olika källor.

Tentamen

Tentamen ges under sista veckan som ett onlineprov.

- Onlineprovet kommer att öppnas den 7 september.
- Ni kommer att ha 5 timmar på er att skriva tentamen.

Mer info om tentamen kommer när den närmar sig.

Clean Code

Nu är det dags för en introduktion till Clean Code.

Kursboken

Mycket av innehållet såsom idéer, citat, exempel och illustrationer här kommer ifrån kursboken: Clean Code av Robert C. Martin [3].

There will be code

- Most attempts of getting rid of coding have failed.
- Coding is a craft, needing craftsmanship
- There are ways of proving program correctness logically/ mathematically, but it requires enormous amounts of work, i. e. not feasible for most code.
- Learning from mistakes is a good idea -Learning from other craftsmen is also good idea, maybe faster than by mistakes. This course is all about that.

Clean code will require work, maybe hard work

- Cleaning is easy, right? Just remove the dirt?
- Well . . . ?
- Some cleaning is easy, some harder
- Polishing the surface is not enough

Time collects dirt

Old code seems to have gotten dirty over time, why is that?

Dirty code is expensive

- Have you seen enough code to have come across “other people’s” dirty code?
- What aspects of your development does it slow down?
- Why does a small change in the code affect hundreds of lines, instead of just one?
- Clean code is a matter of professional survival!

Refactoring: the art of cleaning code

- Refactoring is about modifying code to make it cleaner
- But without changing the functional behaviour (at least not correct behaviour, e g cleaning up error handling maybe changes the exception thrown, etc.)
- So: refactoring is not about fixing bugs
  - only refactor code that is correct and passes all tests

Refactoring: small/large, quick/longterm

- Some changes are completed in a second or two
- Some may be a two-year project
  - likely due to code that is old, has not been refactored well/at all, or is badly maintained

Refactoring should be incremental

- Refactoring works best as an incremental activity
  - A little at a time
  - Done as a completely integral part of development
  - All agile methodologies include refactoring as a totally natural activity in all development
  - Old Scout rule: leave the camp site cleaner than when you arrived

A practical structure of the cleaning craftsmanship

- Naming
- Comments
- Formatting
- Code structure: functions/methods, classes, interface, object
- Error handling
- Layering/boundaries
- Testing/testability Yes, some of these overlap.

Almost all of them have to do with reducing dependencies!

We can also organise by the “code smells” that we fix

- Naming: *Non-informative Name, Disinformation . . .*
- Comments: *Redundant Comment, Obsolete Comment . . .*
- Formatting: *Non-aligned grouping, Magic numbers. . .*
- Code structure: *functions/methods, classes, interface, objects: Too many arguments, Implementation Dependency, Feature Envy . . .*
- Error handling: *Misplaced Responsibility. . .*
- Layering/boundaries: *Artificial Coupling. . .*
- Testing/testability: *Slow Test . . .*

Naming: intent

Names should reveal intent

```
int d; // elapsed time in days
```

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Naming: intent

Names should reveal intent

```
public List<int[]> getThem() {
List<int[]> list1 = new List<int[]>();
foreach (int[] x in theList)
if (x[0] == 4)
list1.add(x);
return list1;
}
```

```
public List<Cell> getFlaggedCells() {
List<Cell> flaggedCells = new List<Cell>();
foreach (Cell cell in gameBoard)
if (cell.isFlagged())
flaggedCells.add(cell);
return flaggedCells;
}
```

Naming: don't be stupid or too smart

Names should be clear, readable, searchable, non-cute, non-pun, pronounceable and distinguishable

```
int a = l;
if ( O == 1 )
a=O1;
else
l=01;
```

```
class XYZControllerForEfficientHandlingOfStrings {...
class XYZControllerForEfficientStorageOfStrings {...}
```

Naming: Hungarian notation

- Invented a loooong time ago
- prefix the name with type-, usage-, whatever-information
- e.g. all fields prefixed with underscore: private int _size;
- maybe useful in the “old days”,
- strongly typed languages really do not need it (opinions vary!)

```
long lSum;
int iNumberOfPersons;
interface IBlockService {...}
// useful?
```

Naming: don't overspecify


```
List <Account> accountList;
// will it always be stored as a List?
List <Account> accounts;
// keep it straightforward!
```

Naming: follow conventions


```
public class Customer {...}
// class names are nouns
```

```
public int CalculateSum() {...}
// methods start with verbs
public float ToCelsius() {...}
// convention with implied "Convert" verb
```

Naming: Use solution and problem domain names

If you are using a programmer's concept, e.g. a design pattern, or a specific algorithm, tell the future programmer/ reader about it, certainly if the caller needs to know!

```
public interface GraphicsVisitor {...}
// design pattern Visitor is used
```

If the problem domain has a specific term for something, use it!

```
public double CalculateDeterminant(...) {...}
```

Comments

Writing code that is so clear that comments are totally unnecessary seems to be the ultimate goal

```
// Check to see if the employee is
// eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65)) {...}
```

```
if (employee.isEligibleForFullBenefits()) {...}
```

Comments: good ones

Not all comments are evil:

- XML/API comments to document a library class for others to use (but only public stuff)
- A URL to a description of an advanced algorithm you are using
- TODO comments (usually marked/coloured by your IDE)
- Clarifications that simply are better as comments than code (optimise for the future code reader!)

Comments: bad ones

- Autogenerated (by IDE or lazy programmer), unnecessary and not removed
- Describing what should be readable in proper code
- Misleading, obsolete, incorrect etc.
- Commented-out code. Should be handled by Git or similar.

Bottom line: we will never agree totally on this topic

Formatting

Don't invent your own formatting

- (although: code that stretches over multiple lines usually need a bit of manual invention)
- Your IDE will help you

```
Example: ISO: 2021-11-16
              11/16/21
              16.11.2021
```

Methods/functions

Methods should be small

- A method should do one thing and do it well.
- One thing can be done in one line or forty lines. But it should be clear that it is doing one thing. It probably means “one level of abstraction”.
- Ignore the “cost” of a method call. It can almost always be optimised away by the compiler.
- The method name should describe what it does, not how.

Methods: switch statements

Switch statements (and its cousin: the if else if else chain) are a bit evil by nature

- repeated switch statements are really bad
- Always consider a way to replace them: polymorphism, table controlled code, Strategy design pattern, etc.
- If switch is appropriate, hide it in a single method

Methods: parameters

Parameters/arguments are hard to read and hard to understand and hard to test

- Methods with no parameters are ideal
- Methods with one parameter (monadic) are OK
- Methods with two parameters (dyadic) are less OK
- Methods with three or more parameters are bad
- Methods that modify its arguments (“output arguments”) are almost always a pitfall

Methods: repeated code

- Repeated code is common and painful
  - A horror to maintain, modify, extend
  - Copy/paste is just to easy... just for now... fix it later...
- Extract Method is a refactoring, often supported in IDEs
- Template Method is a design patters helping to remove duplicated code in overriding subclass methods

Classes

- Classes should be small!
  - SRP – "Single Responsibility Principle" is how Uncle Bob expresses that
  - Not so easy to design – easier to see when a class has multiple responsibilities
- "High cohesion – low coupling" is another way of expressing it
  - everything inside the class has "togetherness" – cohesion
  - it has as little as possible to do with the rest of the world – low coupling

Classes: constructors

Calling a constructor with new may not always be the best option

- There are a number of design patterns that handle various situations, the most common one is Factory Method

```
Temperature temp1 = new Temperature(37.2);
Temperature temp2 = Temperature.CreateFromCelsius(37.

Person p1 = new Person("John", 17);
//will always create a Person
Person p2 = PersonFactory.Create("John", 17);
//could create a Teenager
```

Classes: forward compatibility

- How on earth do you future-proof a class?
  - If you knew what the future requires, you would add it now, right?
- You may not know what the future will require, but you may make smart guesses where it will be added. And at least, do not block the possibility of adding things there!
  - Many design patterns are about this
  - Still not easy, sorry.

Error “handling”

- Errors unfortunately tend to happen in an unstructured way and in places where we do not want them to happen
- Exceptions were invented to relieve some part of this: the error reporting
  - Exceptions **do not handle** errors
- “Checked exceptions” is a feature in Java, i e exceptions that the developer “ought to” handle.
- It might be a good idea to think like this also in C#, though no one will force you!

Don't use null

- Using null as a value, argument or return value is a bad idea
- Tony Hoare, inventor of null: "I call it my billion-dollar mistake . . . "
- Lots of code and libraries still use null.
- Since C# 8 there is Nullable to designate values that might be empty
- It avoids NullReferenceException and "forces" programmer to consider non-existence

```
Person? customer = repository.FindById(56);
// can be a Person or not-a-Person
// Person? is the same as Nullable<Person>
//...
if (customer.HasValue()) {
customer.Value.SendGift();
}
```

Exceptions

Exceptions still are a good way to report errors

- Particularly when the code that throws it does not know what to do (How serious is it that the file can't be opened?)
- Return codes or success check methods can easily be ignored or forgotten
- A try-catch is "one thing" so it should probably be the only thing in a method

Ganska mycket

Vi har nu gått igenom ganska mycket. Trots detta finns det mycket mer att lära sig i boken! Jag rekommenderar att läsa kapitel 1-7 + 10. Det är även en bra idé att oavsett hur mycket ni hinner med det andra ta en titt på kapitel 17!

Kapitel 9 blir mer aktuellt vid avsnittet om TDD.

Övningsuppgift

För att komma igång ska vi börja med en liten övning som ni kommer att få jobba med under ca en vecka framåt.

Ni kommer att få ett litet program som ser ganska hemskt ut. Ni ska försöka göra programmet snyggare och förbättra de olika punkterna som vi just gått igenom såsom namngivning, kommentarer, formattering, kodstruktur, error-hantering och layering.

Övningsuppgift

Ni behöver inte oroa er, i uppgiften kommer det viktigaste vara att förbättra utifrån era nuvarande kunskaper och jag rekommenderar att fokusera på uppdelningen av metoder och klasser, så att man jättelätt kan lägga till nya kalkylatorfunktioner.

Vi kommer senare att titta på hur vi skulle kunna lägga till olika patterns och tester!

Avslut

Det var det för idag!

[1] Sebastian Lindgren. *ITHSDistans: Clean-Code*. URL: https://www.ithsdistans.se/course/view.php?id=1181.

[2] Sebastian Lindgren. *ITHSDistans: Clean-Code distans*. URL: https://www.ithsdistans.se/course/view.php?id=1173.

[3] Robert C. Martin and James O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009. ISBN: 9780132350884 0132350882. URL: https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00.