Licensing

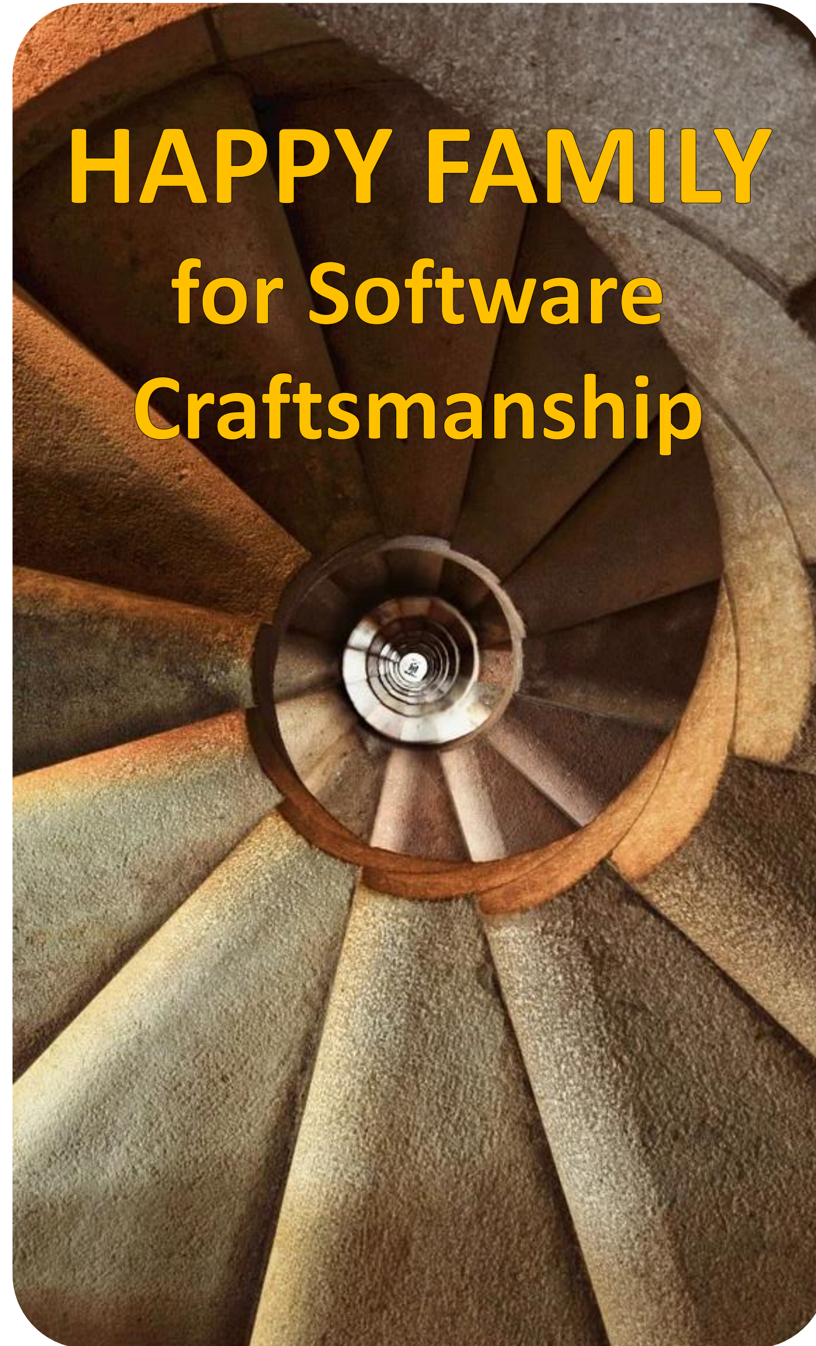
This work is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/

Credits

- Designed by Christophe Moustier
- U. Enzler (most texts)
- Back cards picture: https://pixabay.com/fr/photos/escalier-spirale-l-architecture-600468/
- Icons: Freepik & Kiranshastry (https://www.flaticon.com)

Disclaimer

- The CC4.0 does not apply to the materials referred to by the links
- The Manifesto for Software Craftsmanship may be freely copied in any form, but only in its entirety



Software Craftsmanship

This card set will renew your Dev Experience in a sustainable way.

To find more information on this card set, you can reach https://fan-de-test.fandom.com/fr/wiki/Happy_Family_for_Software_Craftsmanship. Out there some extra games and usages will be proposed, why not yours? ☺

To carry on discovering Clean Coding with Software Craftsmanship practices, you can also reach some books presented in the "Reference" card but **DON'T FORGET PRACTICING** with other craftsmen and organize coding dojos on a regular basis!

Software Craftsmanship

References

- Clean Code by Robert Martin
- McBreen's Software Craftsmanship
- The Pragmatic Programmer series
- The Craftsman by Richard Sennett
- Apprenticeship Patterns by Dave Hoover
- Mastery by George Leonard
- The Dunning-Kruger effect
- The Creative Habit by Twyla Tharp
- The Wikipedia page on Software Craftsmanship
- https://sourcemaking.com
- https://tinyurl.com/SC-cheat-list

You also should extend your quality knowledge through testing materials such as http://tinyurl.com/testagile-eni/ or https://tinyurl.com/testagile-safe-less-eni

Software Craftsmanship

Why?

Over time, *technical debt* reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

Technical debt can be compared to monetary debt. If technical debt is not repaid, it can accumulate 'interest', making it harder to implement changes later on. Unaddressed technical debt increases software entropy.

Software Craftsmanship intends to control inevitable entropy growth.

HAPPY FAMILY for Software Craftsmanship

Software Craftsmanship

Manifesto

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

- Not only working software, but also **well-crafted software**
- Not only responding to change, but also **steadily adding value**
- Not only individuals and interactions, but also **a community of professionals**
- Not only customer collaboration, but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

http://manifesto.softwarecraftsmanship.org

HAPPY FAMILY for Software Craftsmanship

USAGE #1

Happy Families (for 2 to 5 players)

The deck is composed of 5 families of 8 practices (5 relatives+3 foes).

The goal of each player is to collect the most completed families, incl. foes.

Shuffle the cards and distribute 8 to each player.

The dealer starts by asking another player for a card needed to complete a family. If the other player has the card he must give it to you. You may continue asking for cards until you make a mistake.

When a mistake occurs, the asker picks a card from the undistributed ones then the player who was asked for his card takes his turn to request cards.

When a player gathers a full family, he must put the 8 cards face down on the table in front of him. These cards can no longer be requested.

The games stop when all families are completed.

USAGE #2

Practices evaluation (collaborative)
Prepare 5 columns on a table named “We have” / “Let’s work on it” / “Future” / “Not applicable here” / “What is this?”.
Gather the team and evaluate together the appropriate column for this practice.
When using “Not applicable here”, rationale should also be provided.
Let them share the situation with the Product Owner about their concerns and plans.

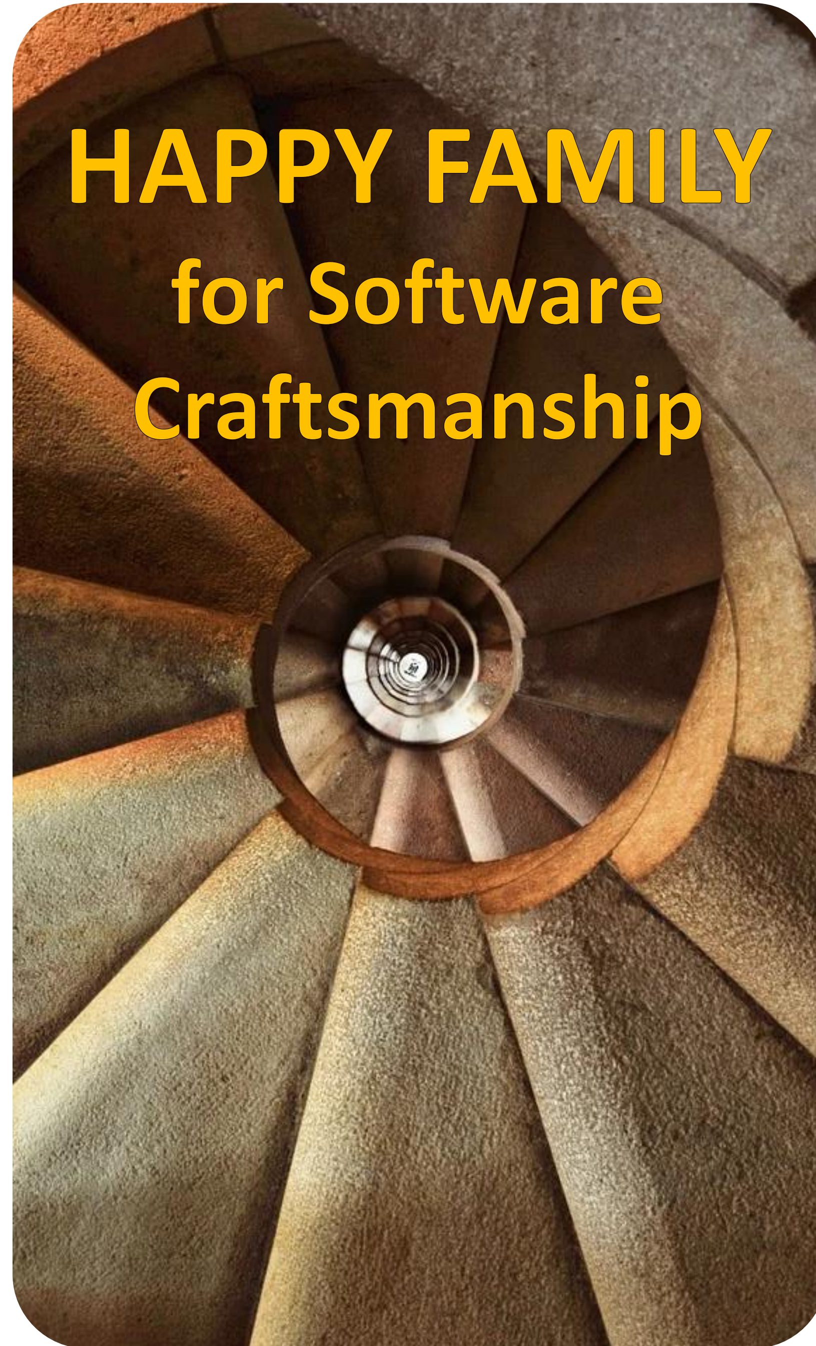
Tip#1: Don’t take too many card, 15 cards for a 15 minutes events works well.

Tip#2: Re-evaluate on a regular basis (say in retrospective) to monitor the progress.

Tip#3: Developers may assign a 0-5 score on a given card to refine the “We have” situation accuracy as follow: 0-Never / 1-Sometimes / 2-Always / 3-Documented / 4-Measured / 5-Optimizing

CRAFTSMANSHIP POWER CARD

Define a use in your own gameplay as a coding standard



1

Loose Coupling

Two classes, components or modules are coupled when at least one of them uses the other. The less these items know about each other, the looser they are coupled. A component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

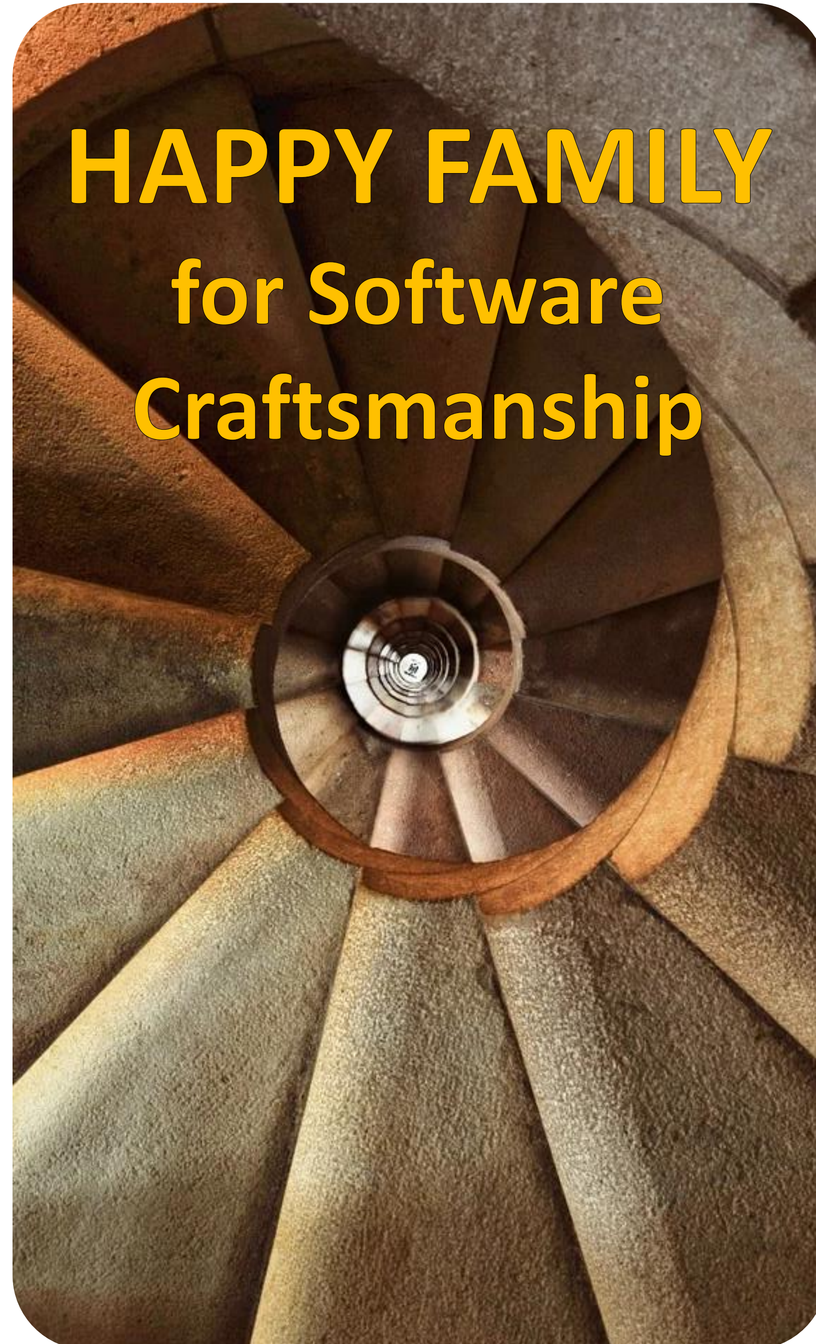
Relatives:

- *Loose Coupling*
- High Cohesion
- Name matches Level of Abstraction
- Mind-sized Components
- Structure over convention

Foes:

- Artificial coupling
- Over configurability
- Misplaced responsibility

DESIGN



2 High Cohesion

Cohesion is the degree to which elements of a whole belong together. Methods and fields in a single class and classes of a component should have high cohesion. High cohesion in classes and components results in simpler, more easily understandable code structure and design.

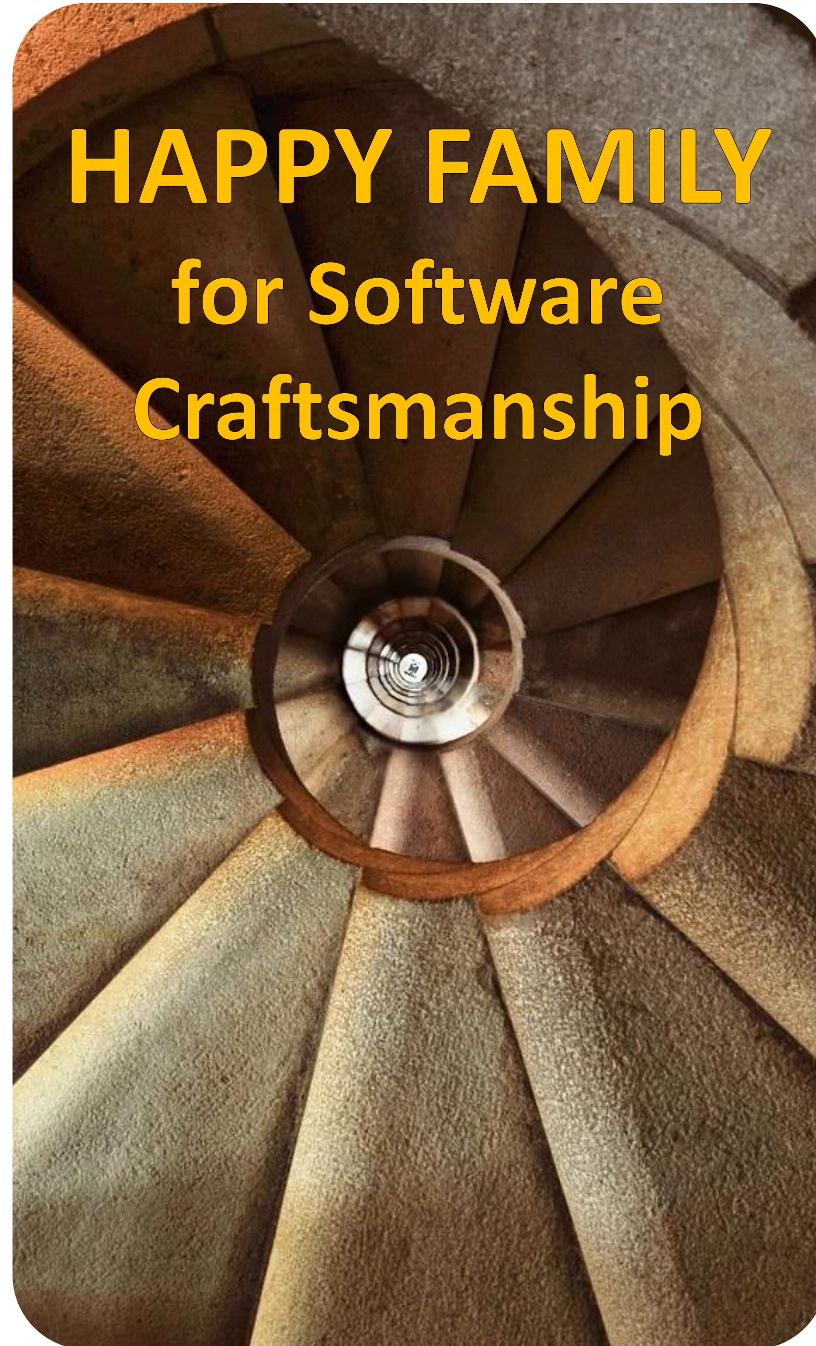
Relatives:

- Loose Coupling
- *High Cohesion*
- Name matches Level of Abstraction
- Mind-sized Components
- Structure over convention

Foes:

- Artificial coupling
- Over configurability
- Misplaced responsibility

DESIGN



3

Name matches Level of Abstraction

Choose names that reflect the level of abstraction of the class or method you are working in.

Relatives:

- Loose Coupling
- High Cohesion
- *Name matches Level of Abstraction*
- Mind-sized Components
- Structure over convention

Foes:

- Artificial coupling
- Over configurability
- Misplaced responsibility

DESIGN

4 Mind-sized Components

Break your system down into components that are of a size you can grasp within your mind so that you can predict consequences of changes easily
(dependencies, control flow, …).

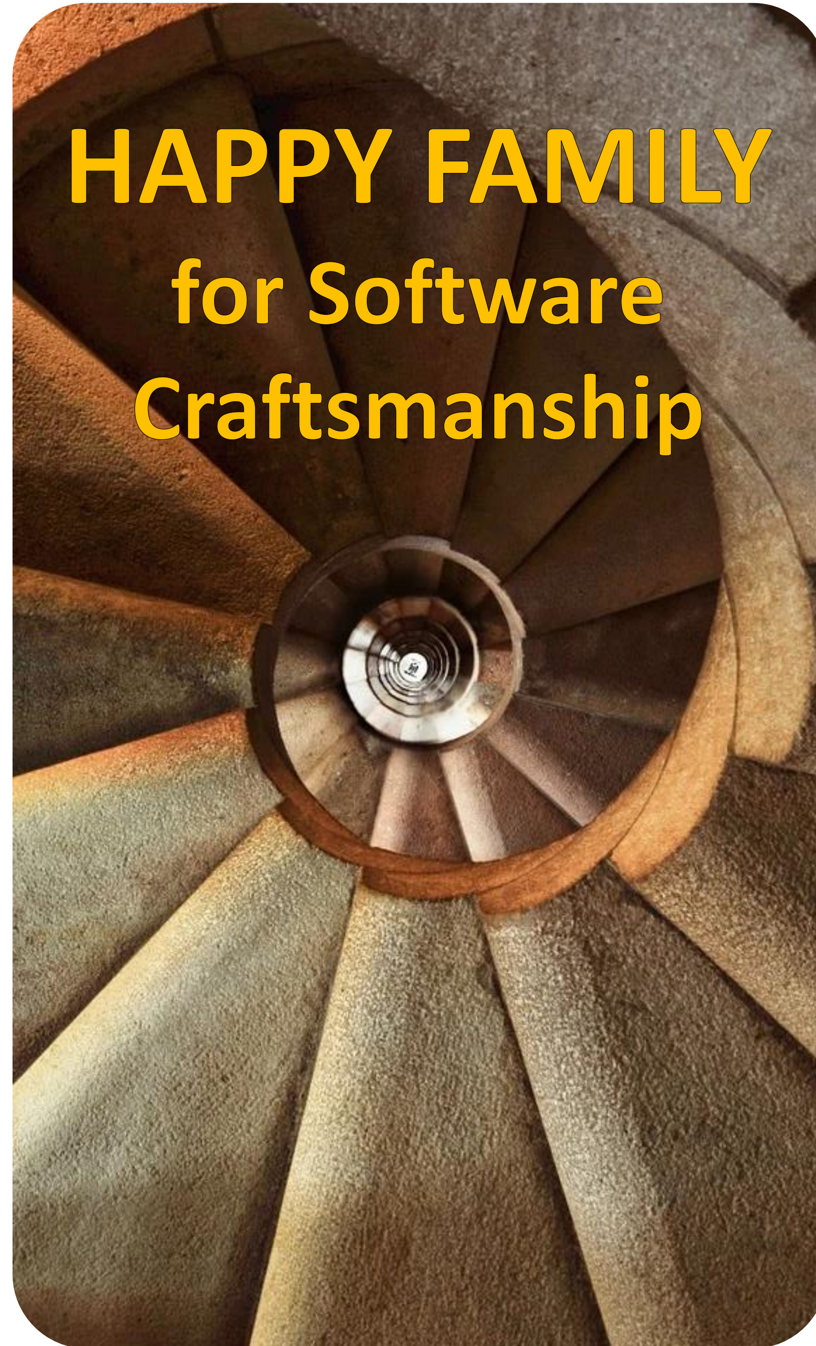
Relatives:

- Loose Coupling
- High Cohesion
- Name matches Level of Abstraction
- *Mind-sized Components*
- Structure over convention

Foes:

- Artificial coupling
- Over configurability
- Misplaced responsibility

DESIGN



5 Structure over convention

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance.

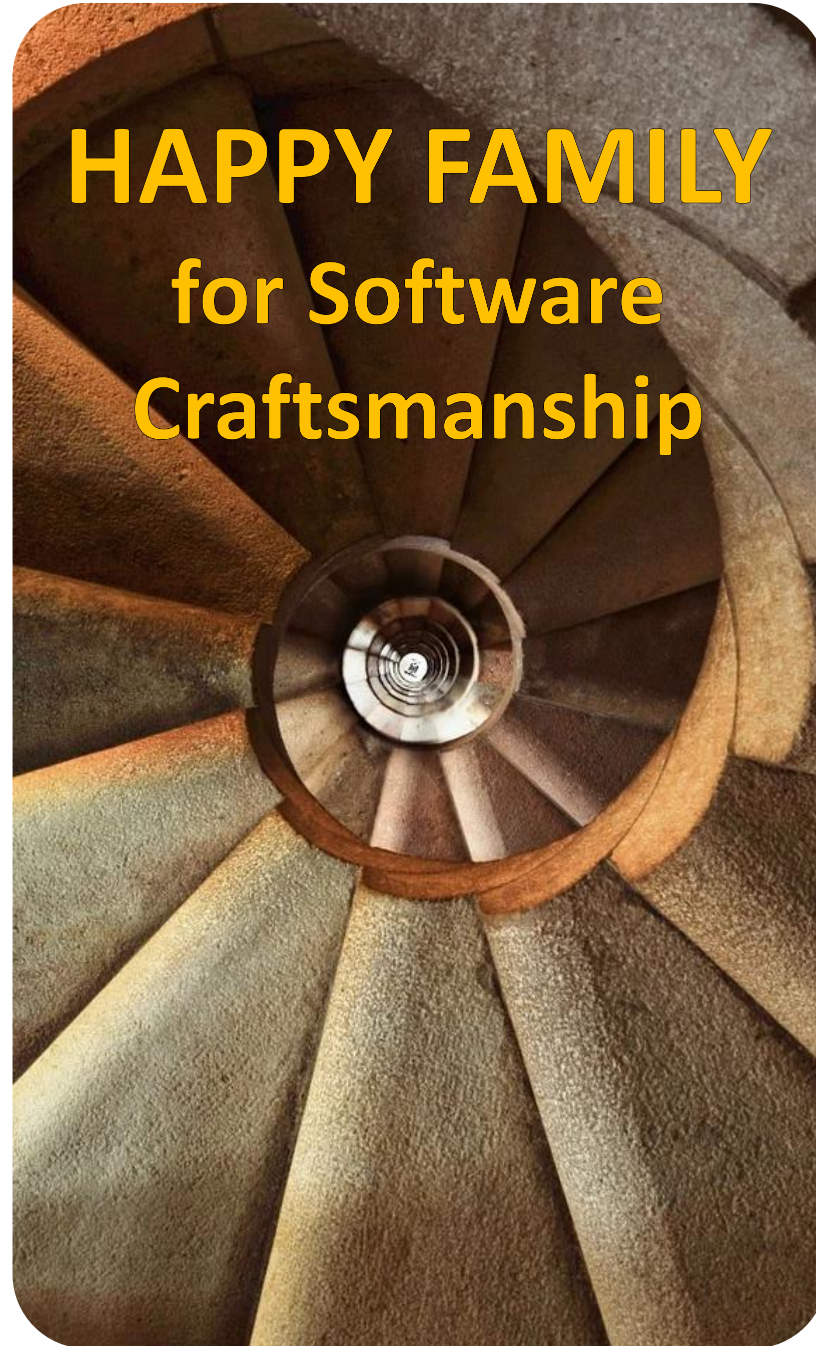
Relatives:

- Loose Coupling
- High Cohesion
- Name matches Level of Abstraction
- Mind-sized Components
- *Structure over convention*

Foes:

- Artificial coupling
- Over configurability
- Misplaced responsibility

DESIGN



6 Artificial coupling

An artificial coupling is a coupling which is only there for technical reasons and should not be coupled
Example : `Time` depending on `Watch` (Time should be meaningful without a Watch)

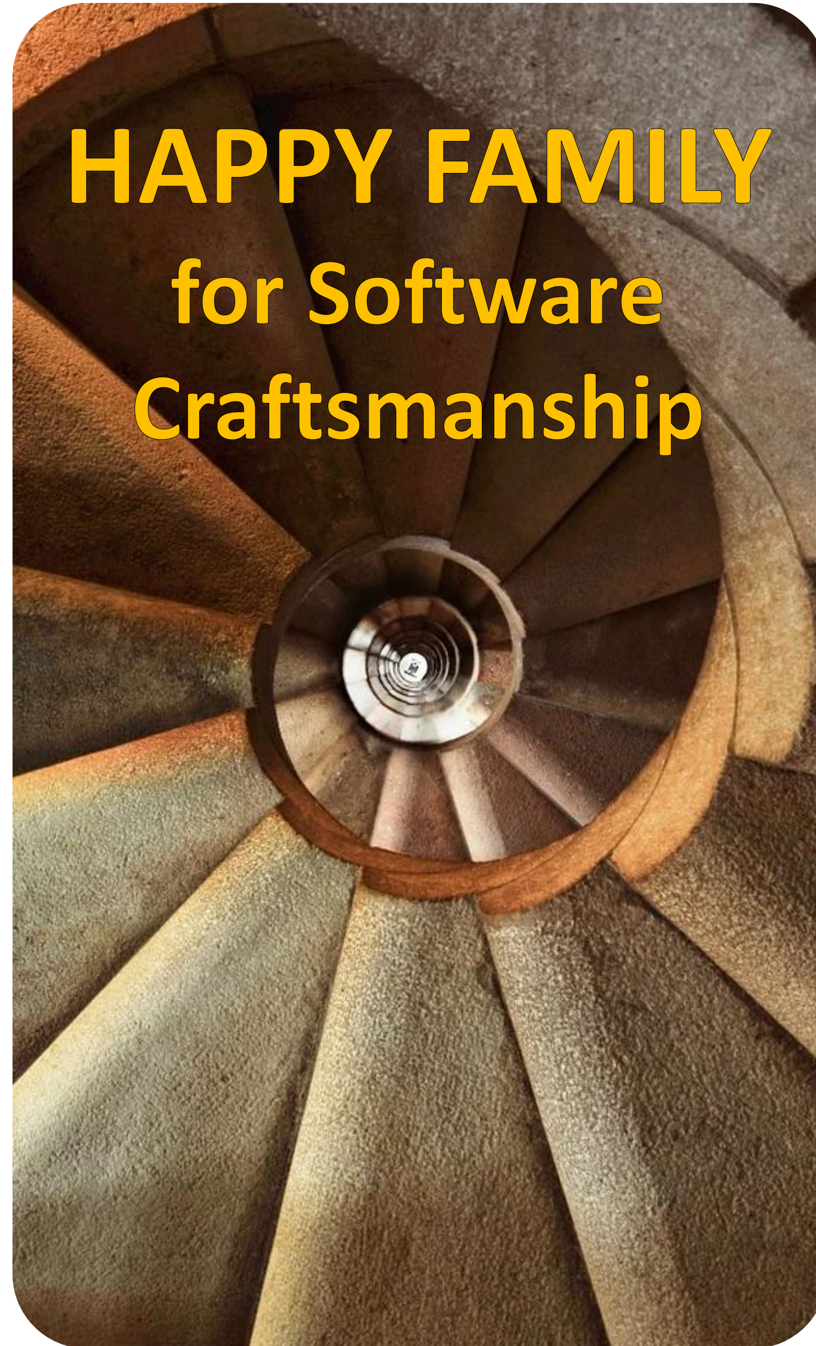
Relatives:

- Loose Coupling
- High Cohesion
- Name matches Level of Abstraction
- Mind-sized Components
- Structure over convention

Foes:

- *Artificial coupling*
- Over configurability
- Misplaced responsibility

DESIGN



7

Over configurability

Prevent configuration just for the sake of it – or because nobody can decide how it should be. Otherwise, this will result in overly complex, unstable systems.

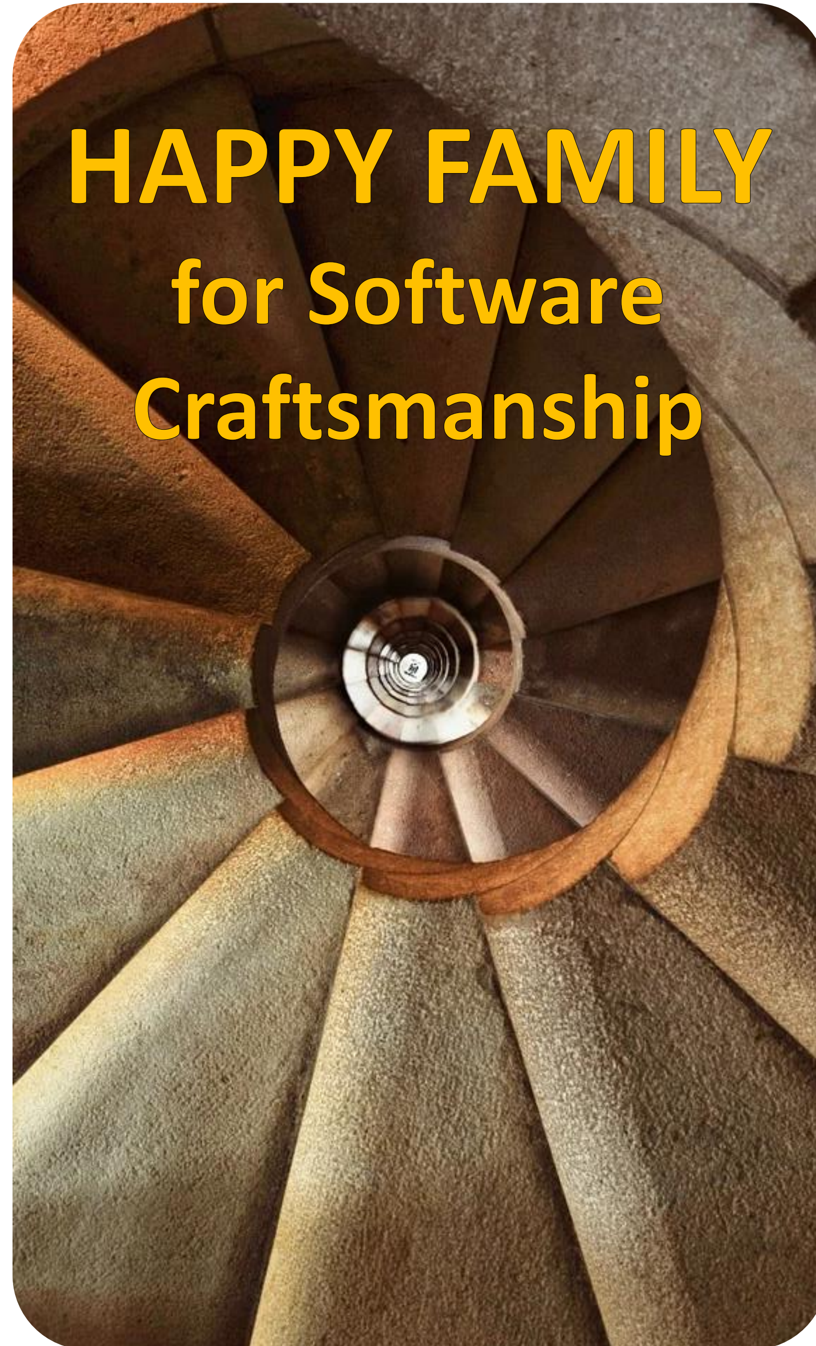
Relatives:

- Loose Coupling
- High Cohesion
- Name matches Level of Abstraction
- Mind-sized Components
- Structure over convention

Foes:

- Artificial coupling
- *Over configurability*
- Misplaced responsibility

DESIGN



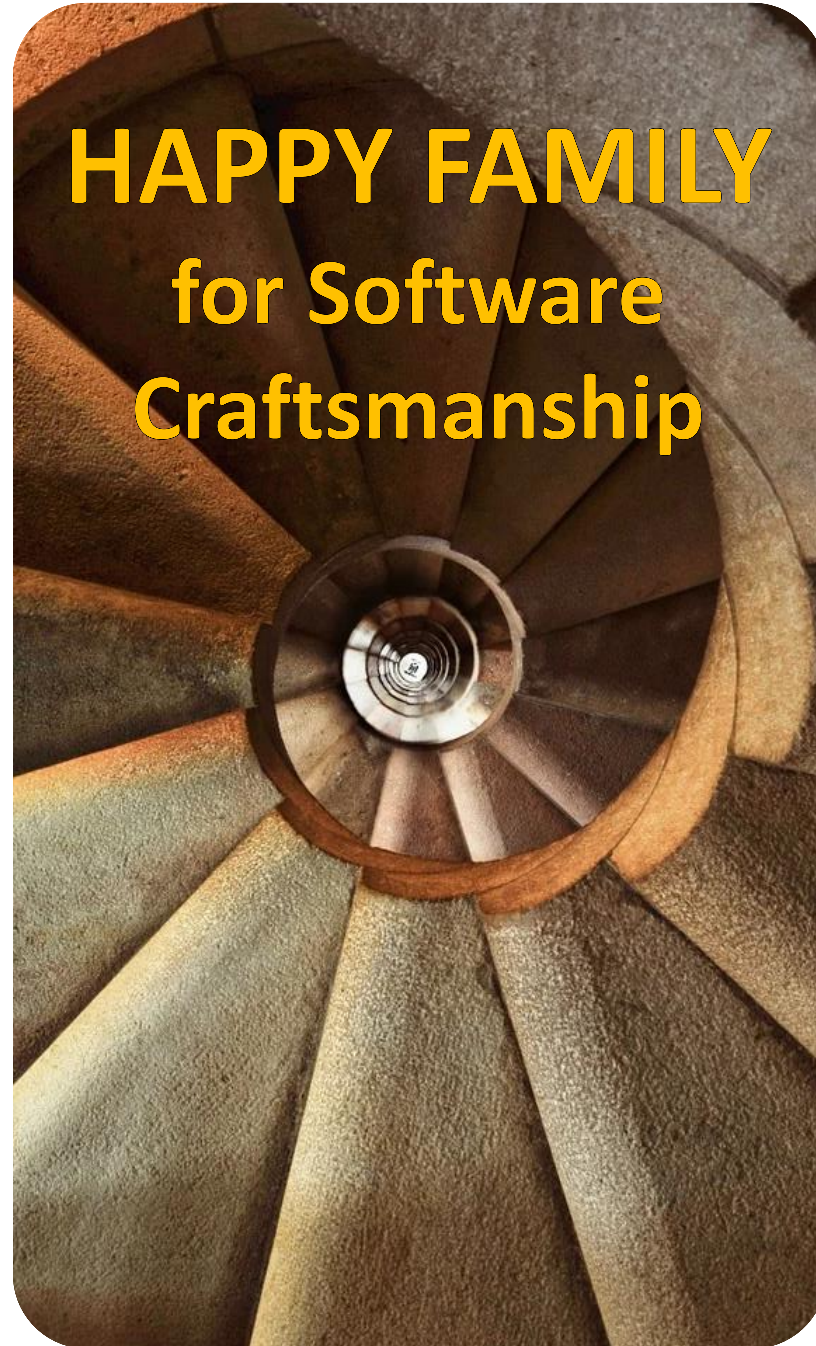
8 Misplaced responsibility

Something put in the wrong place.
Maybe the architecture is not “SOLID” and the Principle of “Least Astonishment” may be also violated.

Relatives:
- Loose Coupling
- High Cohesion
- Name matches Level of Abstraction
- Mind-sized Components
- Structure over convention

Foes:
- Artificial coupling
- Over configurability
- *Misplaced responsibility*

DESIGN



1 Automated ATDD

Use automated Acceptance Test Driven Development for regression testing and executable specifications.
ATDD should be defined in Sprint Refinement e.g. within a “3 Amigos” session especially on tricky US.

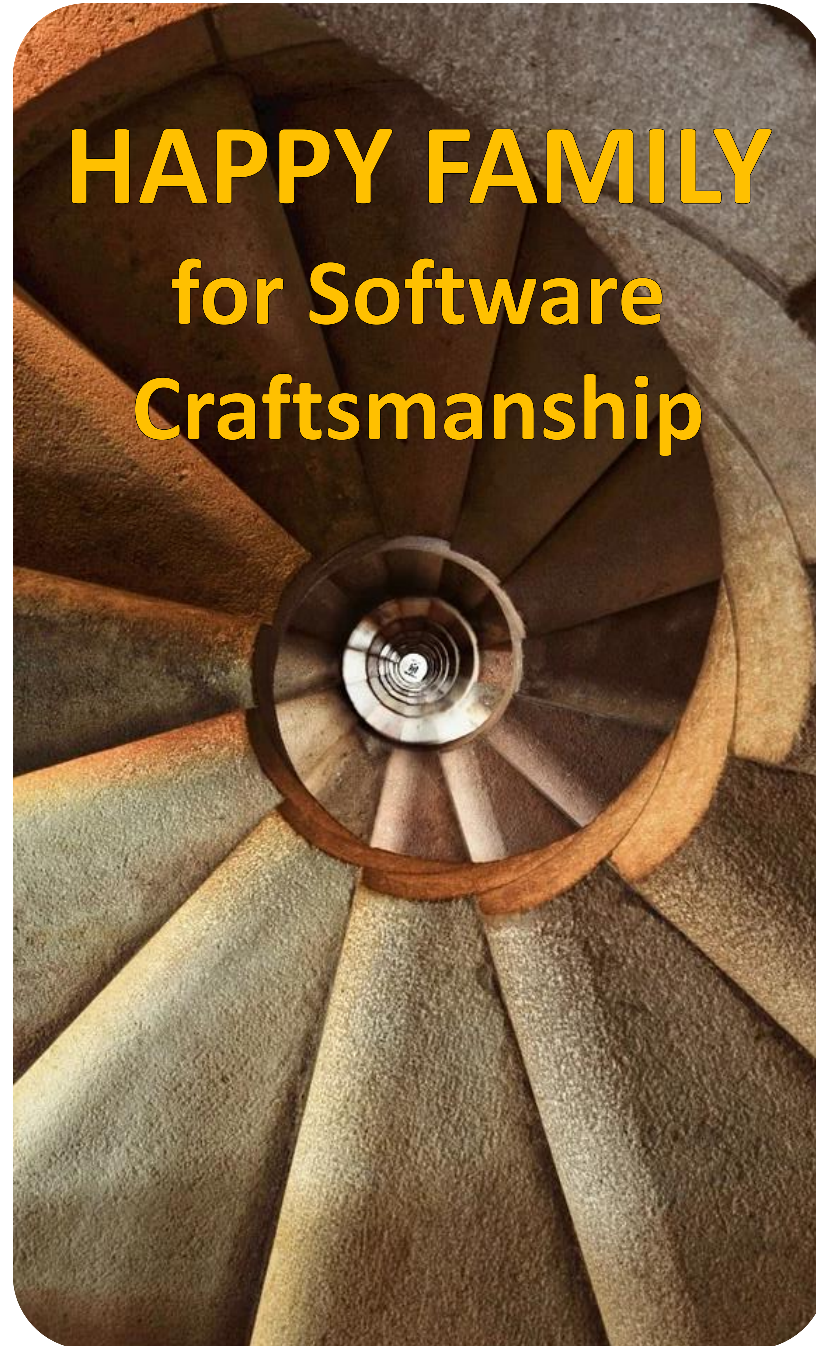
Relatives:

- *Automated ATDD*
- Pairwise Testing
- TDD with tiny steps
- FIRST
- Defect Driven Testing - DDT

Foes:

- Excessive Mock usage
- Incorrect Behavior at Boundaries
- Using Code Coverage as a Goal

TESTING



2 Pairwise Testing

A combinatorial method of software testing that, for each pair of input parameters to a system (typically, a software algorithm), tests all possible discrete combinations of those parameters.

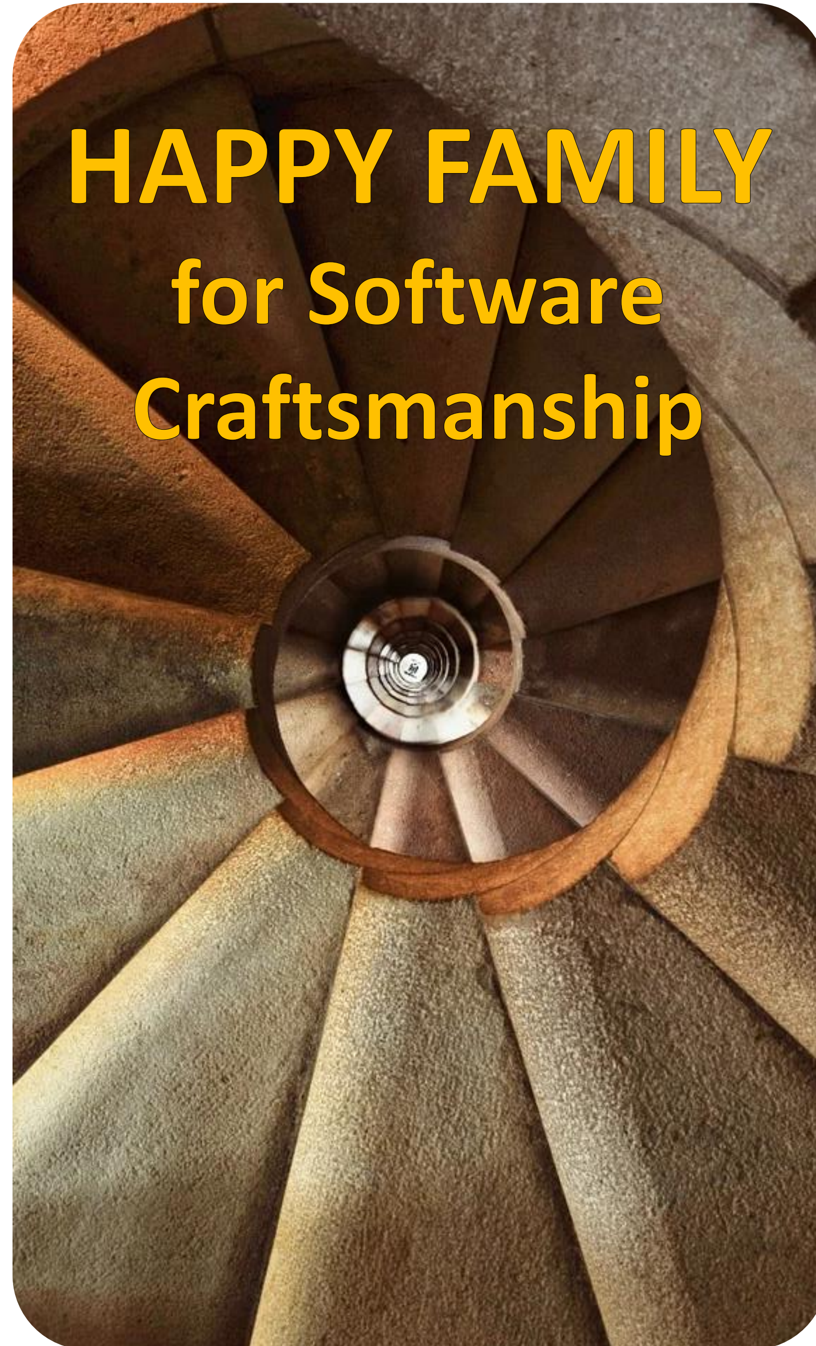
Relatives:

- Automated ATDD
- *Pairwise Testing*
- TDD with tiny steps
- FIRST
- Defect Driven Testing - DDT

Foes:

- Excessive Mock usage
- Incorrect Behavior at Boundaries
- Using Code Coverage as a Goal

TESTING



3

TDD with tiny steps

Start developing Unit Tests with tiny little steps. Add only a little code in test before writing the required production code. Then repeat. Add only one Assert per step.

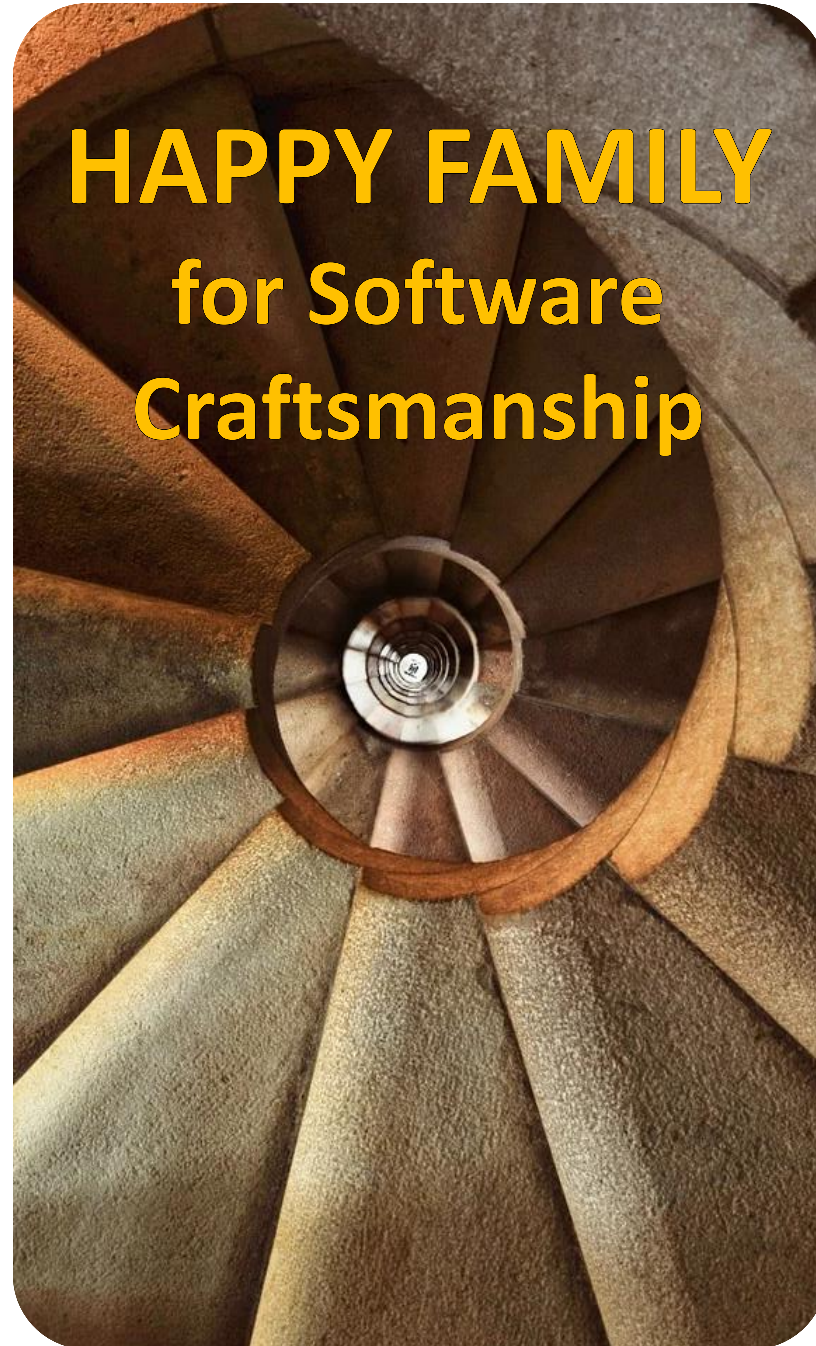
Relatives:

- Automated ATDD
- Pairwise Testing
- *TDD with tiny steps*
- FIRST
- Defect Driven Testing - DDT

Foes:

- Excessive Mock usage
- Incorrect Behavior at Boundaries
- Using Code Coverage as a Goal

TESTING



4

FIRST

Automated unit tests must be

- *Fast*: in order to be executed often
- *Isolated*: No dependency between tests.
- *Repeatable*: No assumed initial state, nothing left behind
- *Self-Validating*: No interpretation or intervention.
- *Timely*: Tests are written at the right time (TDD, DDT, Plain old Unit Tests)

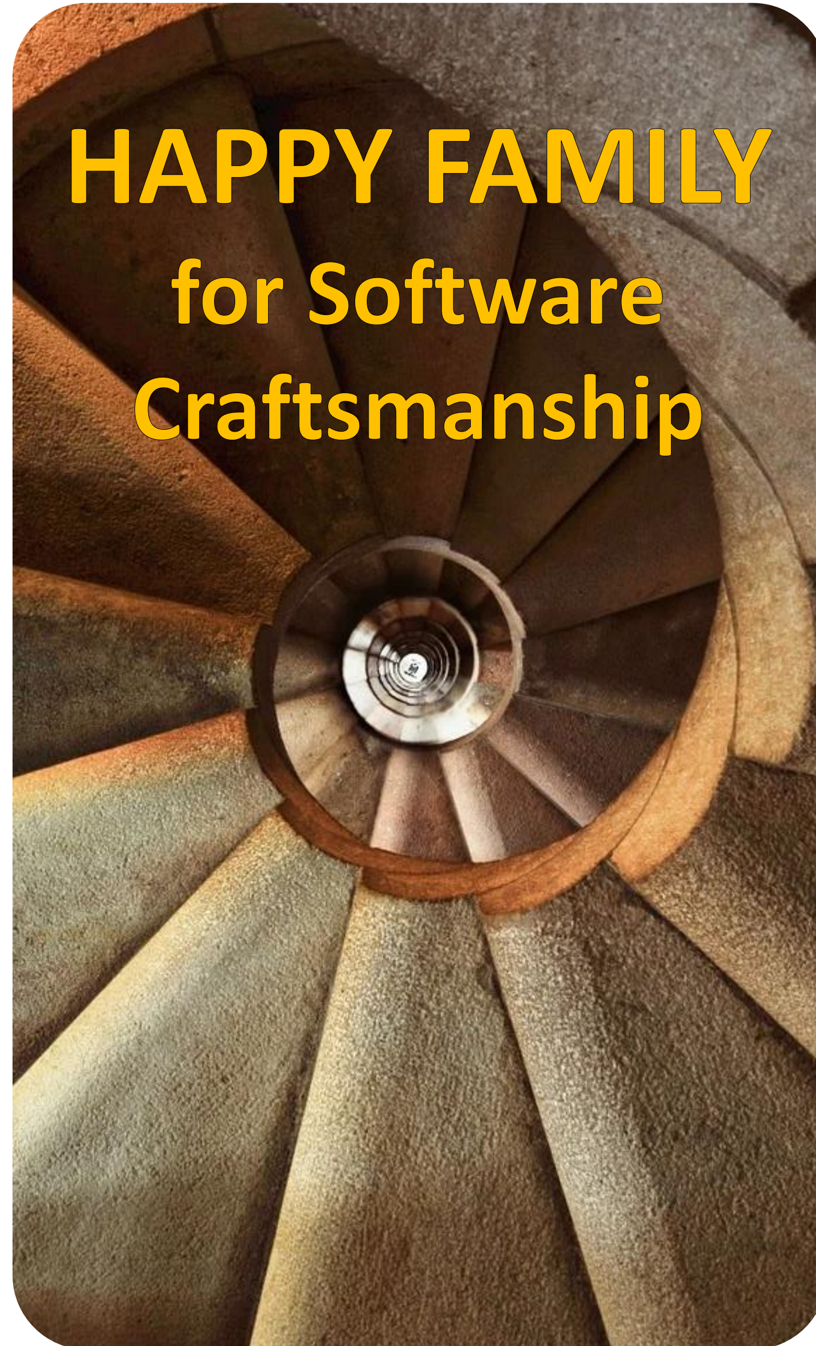
Relatives:

- Automated ATDD
- Pairwise Testing
- TDD with tiny steps
- *FIRST*
- Defect Driven Testing - DDT

Foes:

- Excessive Mock usage
- Incorrect Behavior at Boundaries
- Using Code Coverage as a Goal

TESTING



5 Defect Driven Testing - DDT

This happens whenever a case was not addressed in a unit test, then you should write a unit test that reproduces the defect – Fix code – Test will succeed – Defect will never return

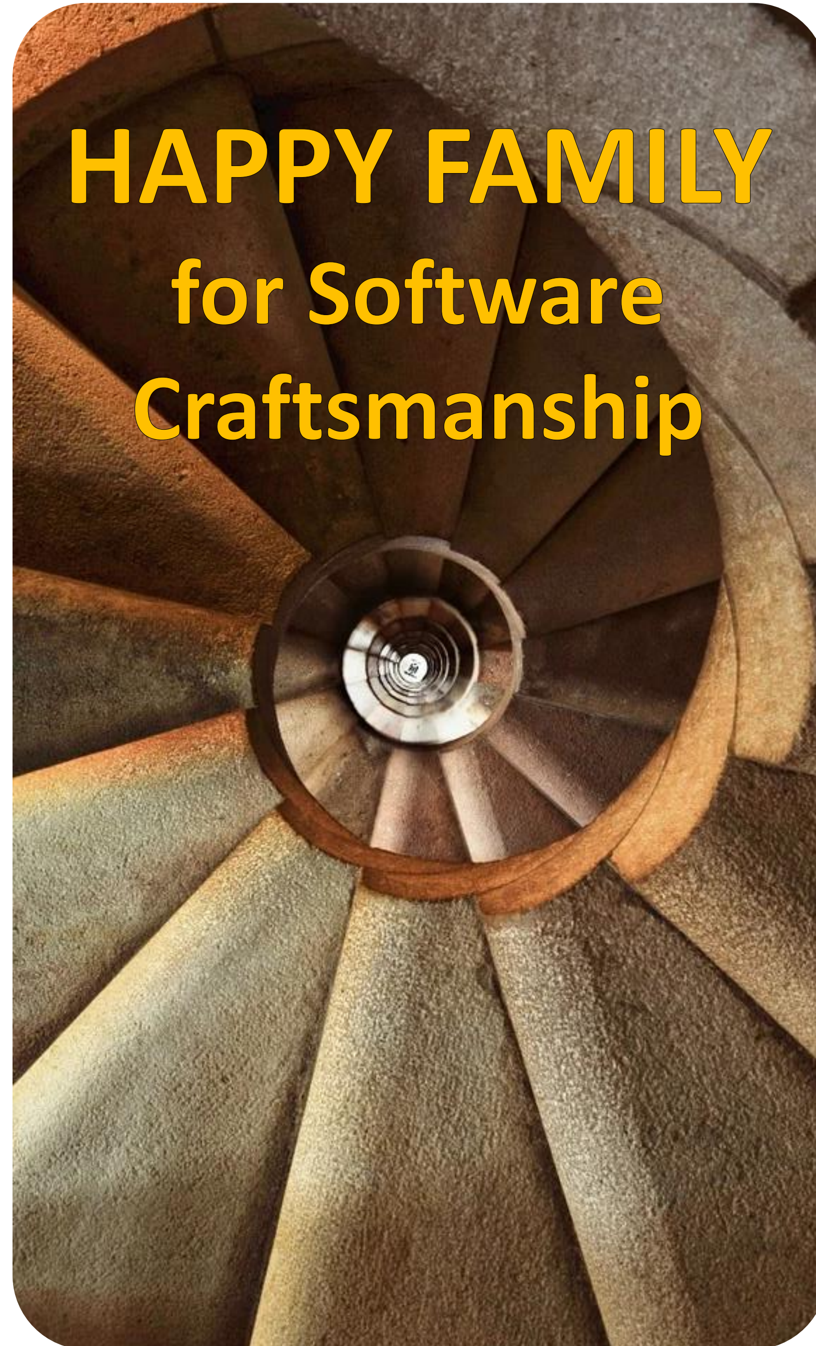
Relatives:

- Automated ATDD
- Pairwise Testing
- TDD with tiny steps
- FIRST
- *Defect Driven Testing - DDT*

Foes:

- Excessive Mock usage
- Incorrect Behavior at Boundaries
- Using Code Coverage as a Goal

TESTING



6 Excessive Mock usage

When unit testing, dependencies must be faked to isolate behaviors. If your test needs a lot of testDoubles (mocks, stubs, fakes,...), then consider splitting the testee into several classes or provide an additional abstraction between your testee and its dependencies.

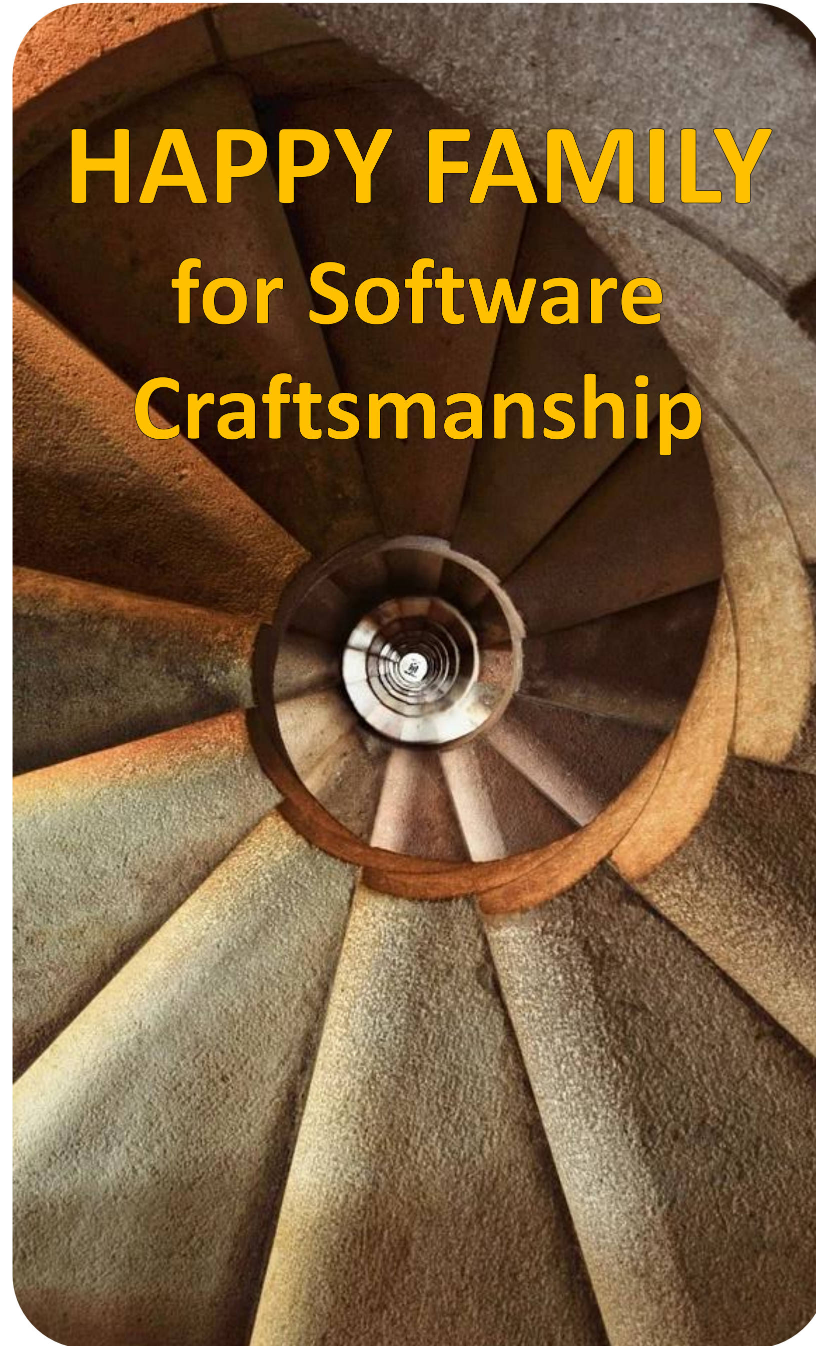
Relatives:

- Automated ATDD
- Pairwise Testing
- TDD with tiny steps
- FIRST
- Defect Driven Testing - DDT

Foes:

- *Excessive Mock usage*
- Incorrect Behavior at Boundaries
- Using Code Coverage as a Goal

TESTING



7

Incorrect Behavior at Boundaries

Always unit test boundaries. Do not assume behavior. Parameters may have boundaries from business (require them from the Product Owner) values and also from code (e.g. 0-255 for a byte). Intervals between boundaries should be tested on valid AND invalid intervals. Look for “Pairwise testing” when facing too many combinations.

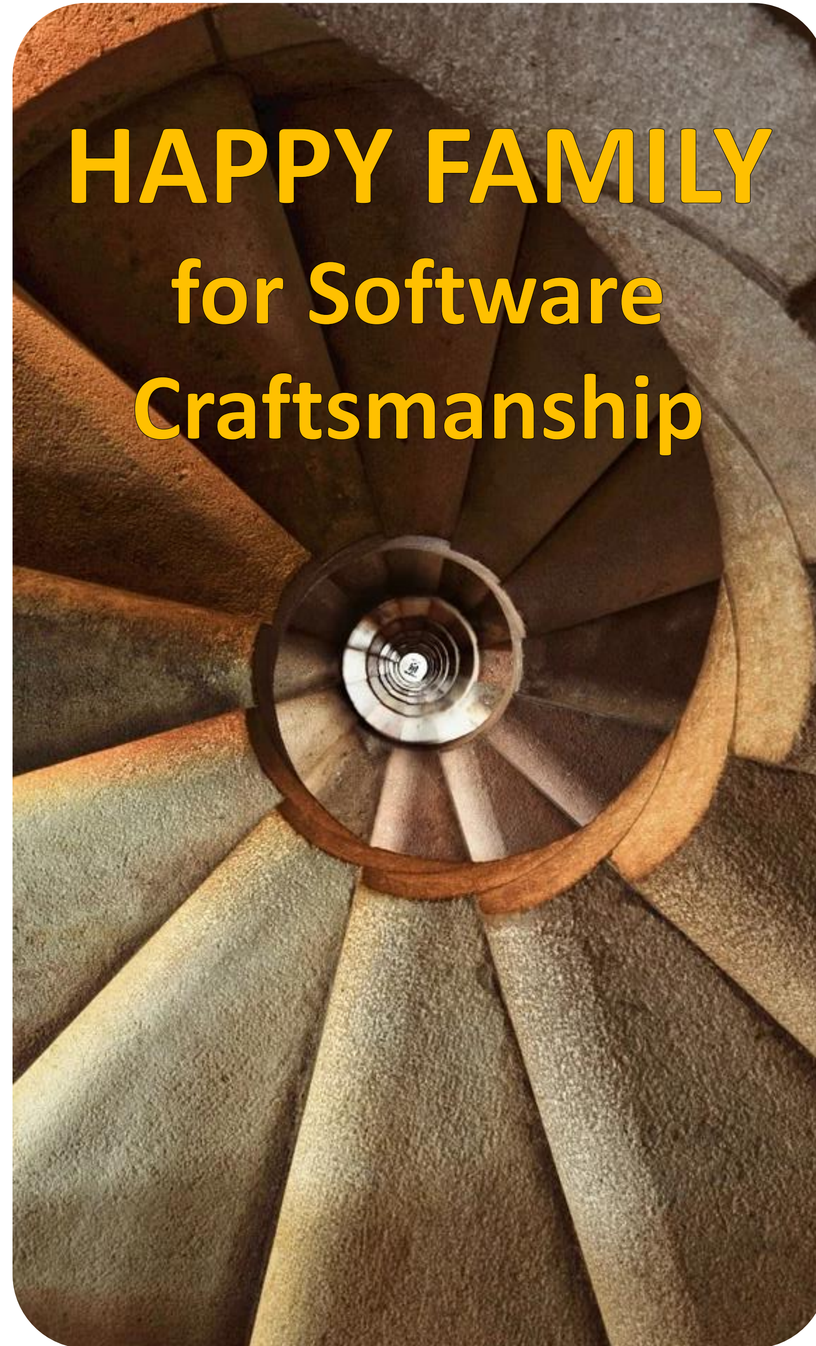
Relatives:

- Automated ATDD
- Pairwise Testing
- TDD with tiny steps
- FIRST
- Defect Driven Testing - DDT

Foes:

- Excessive Fake usage
- *Incorrect Behavior at Boundaries*
- Using Code Coverage as a Goal

TESTING



8

Using Code Coverage as a Goal

Use code coverage to find missing tests but don’t use it as a driving tool. Otherwise, the result could be tests that increase code coverage but not certainty. Code coverage is much weaker than Branch cov., Decision cov. (see Pairwise Testing) or Path cov. (that late one is usually impossible). Coverture should be adapted with failure impacts severity.

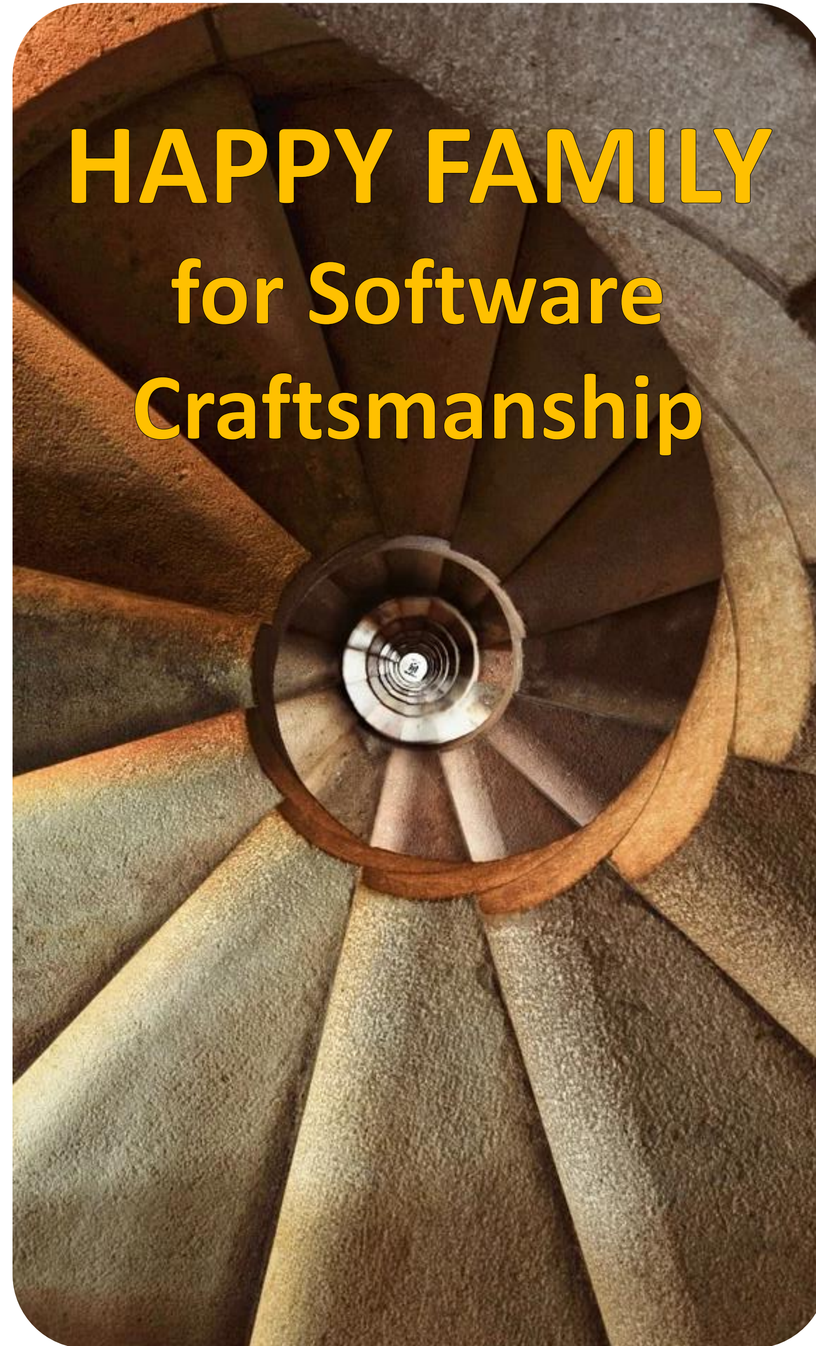
Relatives:

- Automated ATDD
- Pairwise Testing
- TDD with tiny steps
- FIRST
- Defect Driven Testing - DDT

Foes:

- Excessive Fake usage
- Incorrect Behavior at Boundaries
- *Using Code Coverage as a Goal*

TESTING



1

Always have a Running System

Change your system in small steps, from a running state to a running state. Isolation of the area to refactor is key, so prepare fakes, TDDs and possible new interfaces at refactoring boundaries before reengineering.

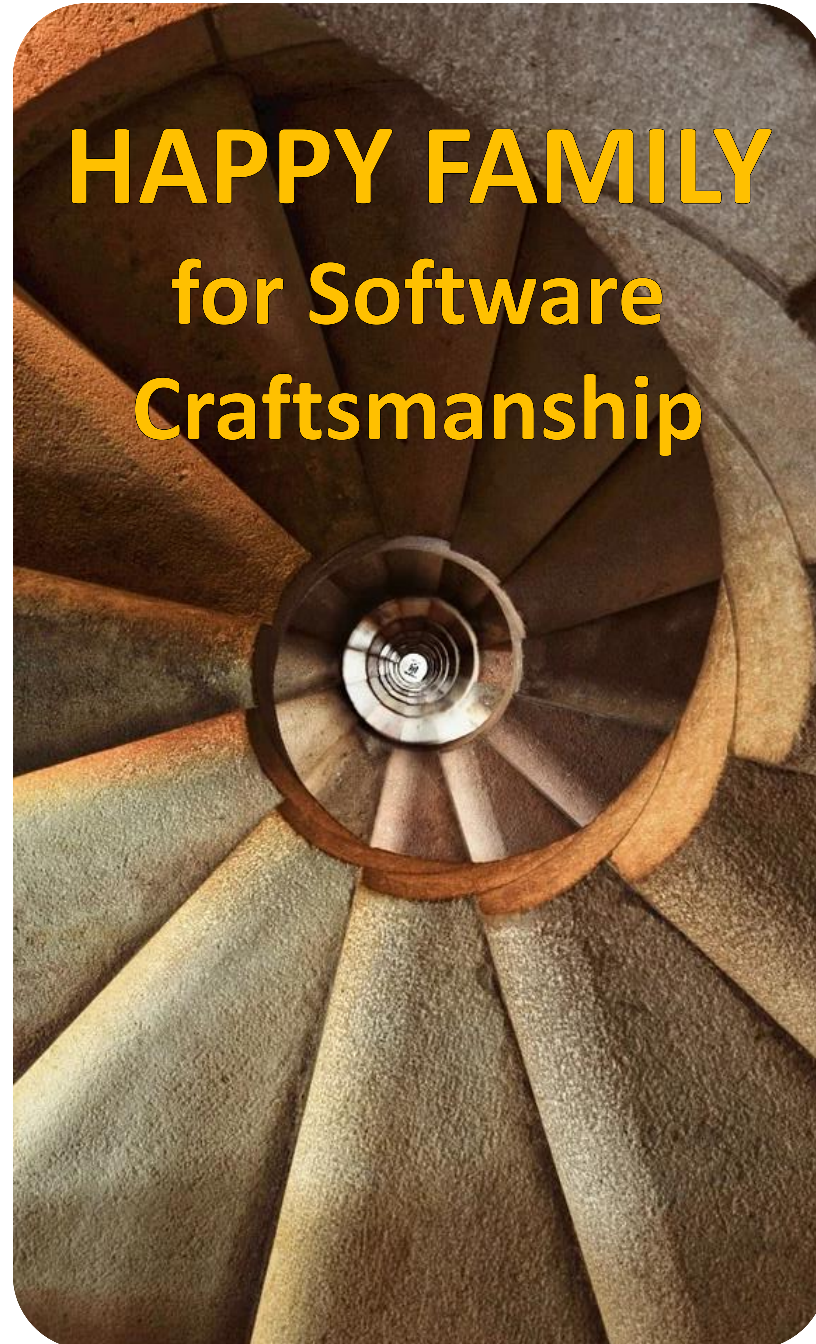
Relatives:

- *Always have a Running System*
- The Three Laws of TDD
- Migrate Data
- Small Refactoring's
- It is Easy to Remove

Foes:

- Duplication
- Rigidity
- Needless Complexity

REFACTORING



2 The Three Laws of TDD

1 - You are not allowed to write any production code unless it is to make a failing unit test pass.
2 - You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3 - You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

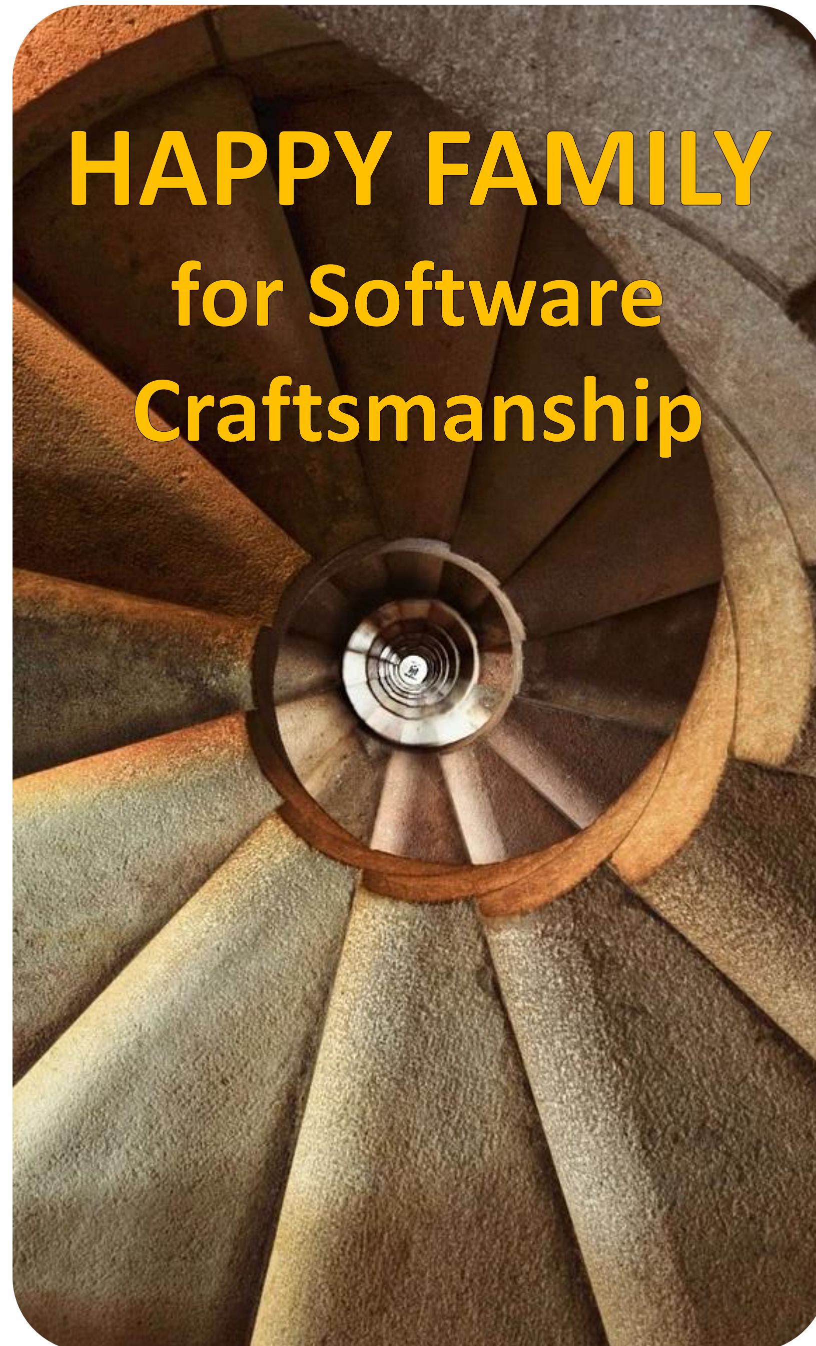
Relatives:

- Always have a Running System
- *The Three Laws of TDD*
- Migrate Data
- Small Refactoring's
- It is Easy to Remove

Foes:

- Duplication
- Rigidity
- Needless Complexity

REFACTORING



3 Migrate Data

Move from one representation to another by temporary duplication of data structures. Consider small and frequent refactoring rather than wide range refactoring area for data migration could also be tremendous!

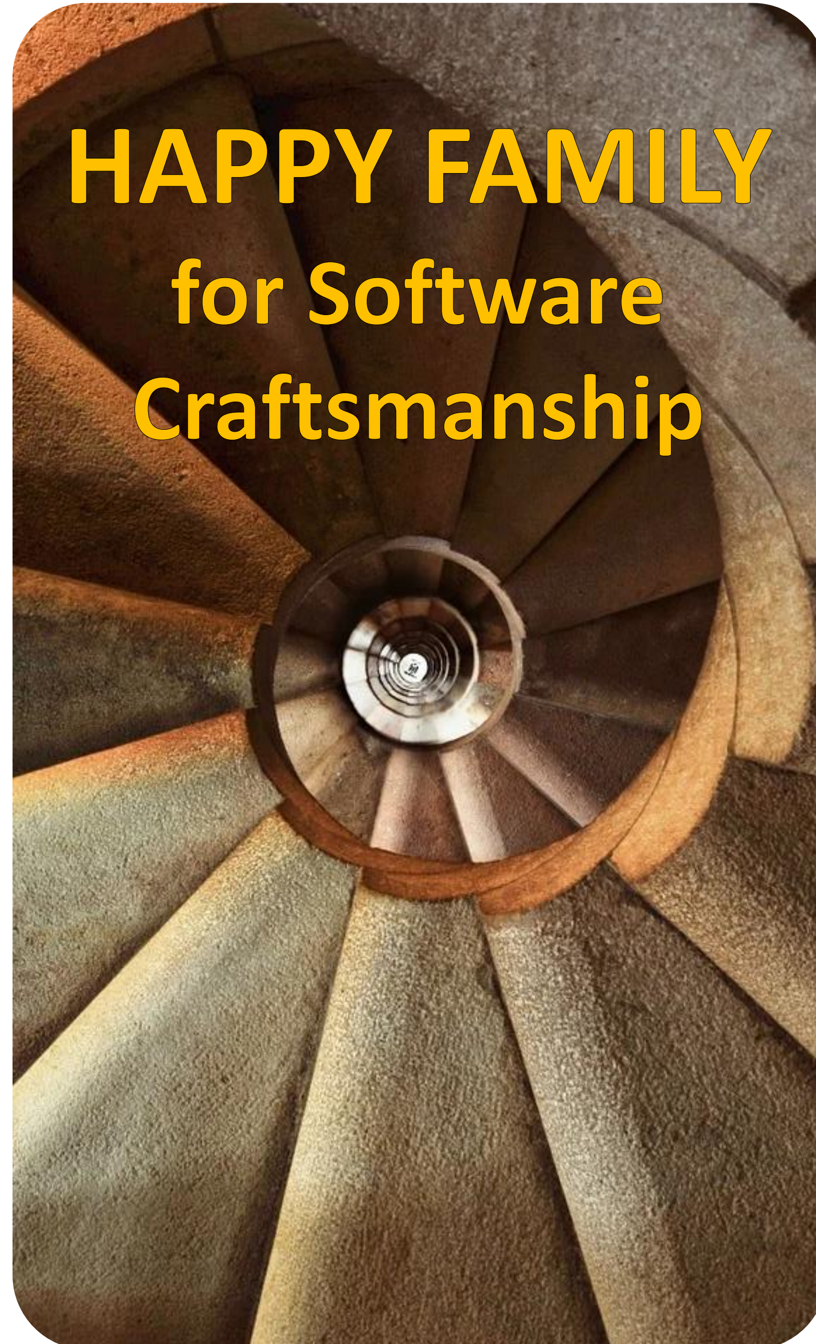
Relatives:

- Always have a Running System
- The Three Laws of TDD
- *Migrate Data*
- Small Refactoring's
- It is Easy to Remove

Foes:

- Duplication
- Rigidity
- Needless Complexity

REFACTORING



4 Small Refactoring's

Only refactor in small steps with working code in-between so that you can keep all loose ends in your head. Otherwise, defects sneak in and data migration becomes tremendous.

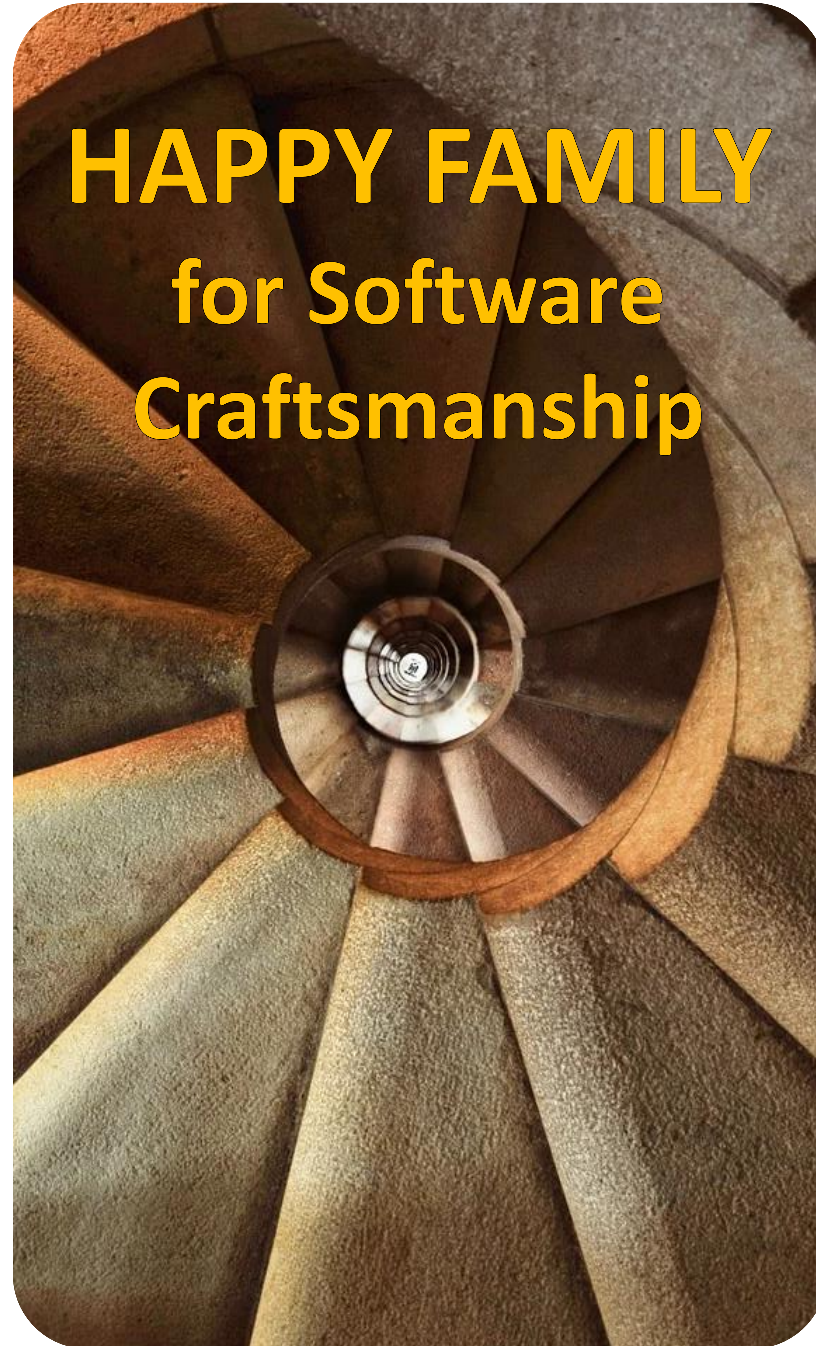
Relatives:

- Always have a Running System
- The Three Laws of TDD
- Migrate Data
- *Small Refactoring's*
- It is Easy to Remove

Foes:

- Duplication
- Rigidity
- Needless Complexity

REFACTORING



5 It is Easy to Remove

We normally build software by adding, extending or changing features.
However, removing elements is important so that the overall design can be kept as simple as possible. When a block gets too complicated, it has to be
removed and replaced with one or more simpler blocks.

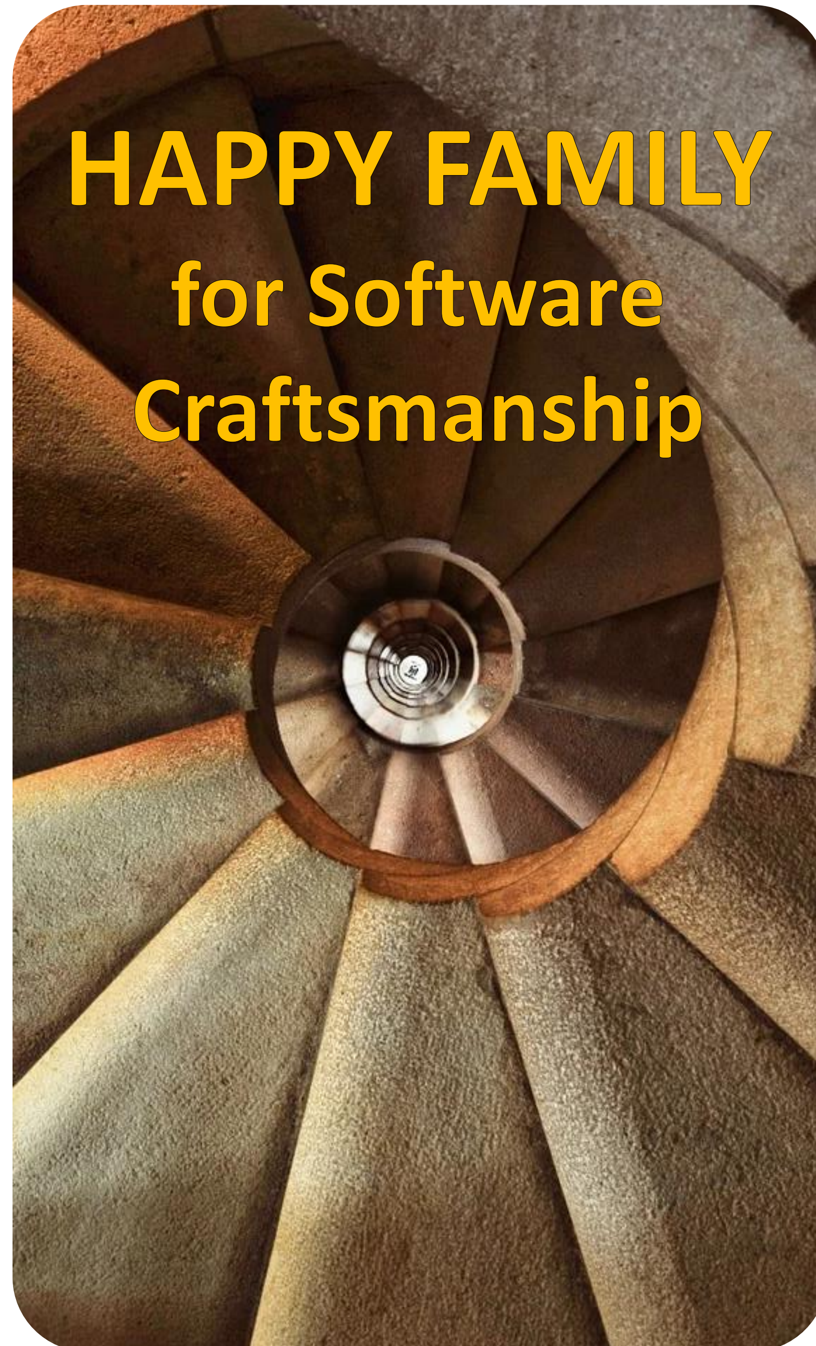
Relatives:

- Always have a Running System
- The Three Laws of TDD
- Migrate Data
- Small Refactoring's
- *It is Easy to Remove*

Foes:

- Duplication
- Rigidity
- Needless Complexity

REFACTORING



6 Duplication

Code contains duplication or design duplicates (doing the same thing in a different way). Making a change to a duplicated piece of code is more expensive and more error-prone. Eliminate duplication. Violation of the "Don't repeat yourself" (DRY) principle

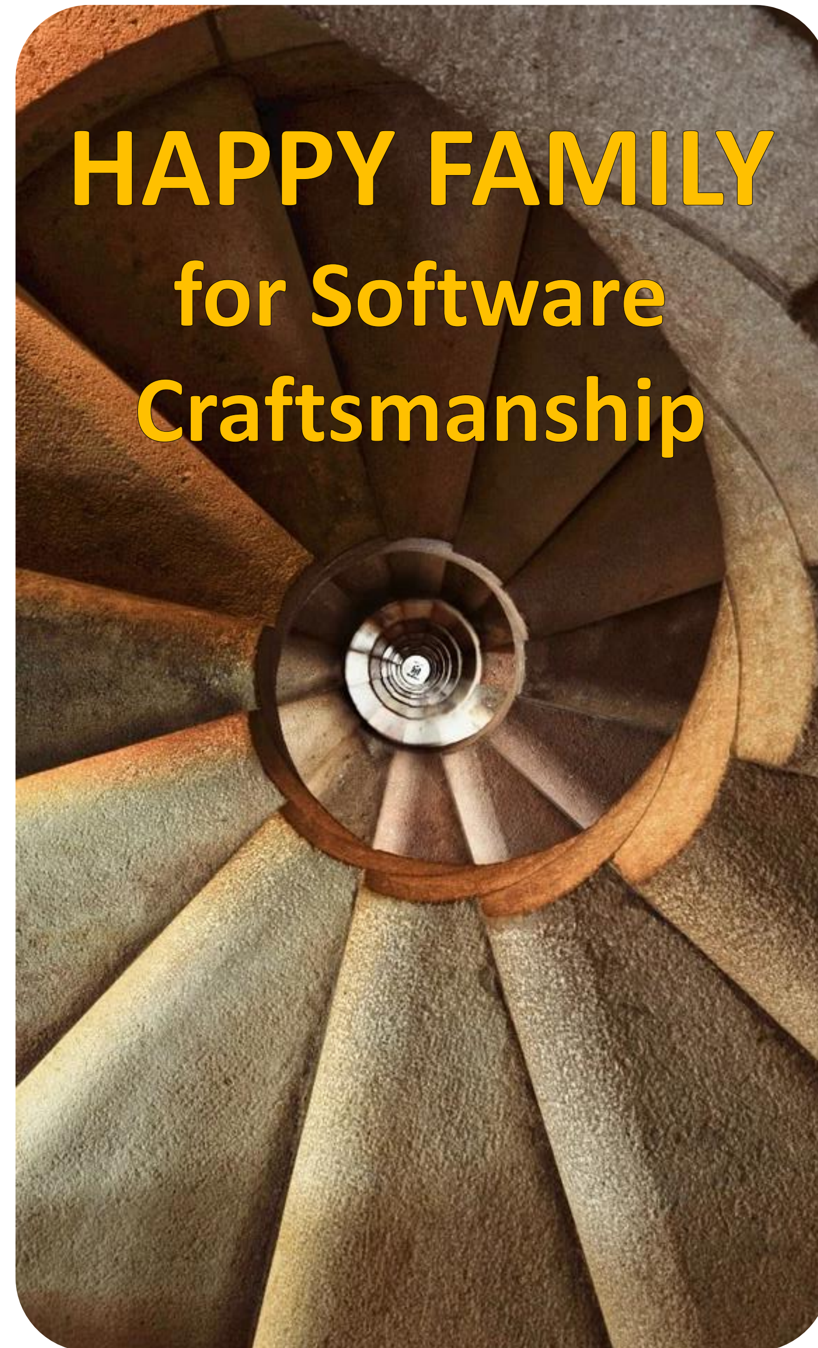
Relatives:

- Always have a Running System
- The Three Laws of TDD
- Migrate Data
- Small Refactoring's
- It is Easy to Remove

Foes:

- *Duplication*
- Rigidity
- Needless Complexity

REFACTORING



7

Rigidity

Software code (unit tests included) is difficult to change. A small change causes a cascade of subsequent changes.

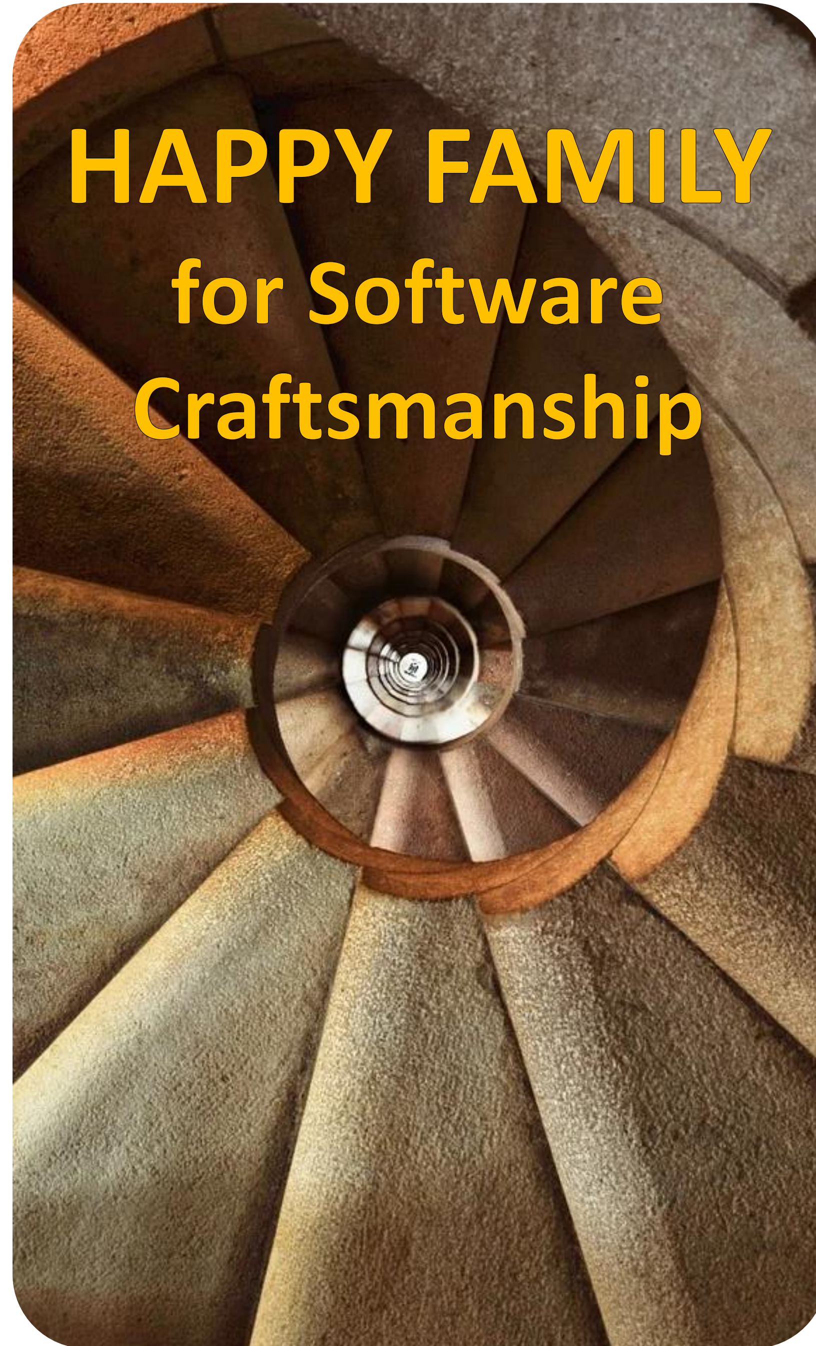
Relatives:

- Always have a Running System
- The Three Laws of TDD
- Migrate Data
- Small Refactoring's
- It is Easy to Remove

Foes:

- Duplication
- *Rigidity*
- Needless Complexity

REFACTORING



8 Needless Complexity

The design contains elements that are currently not useful. The added complexity makes the code harder to comprehend. Therefore, extending and changing the code results in higher effort than necessary.

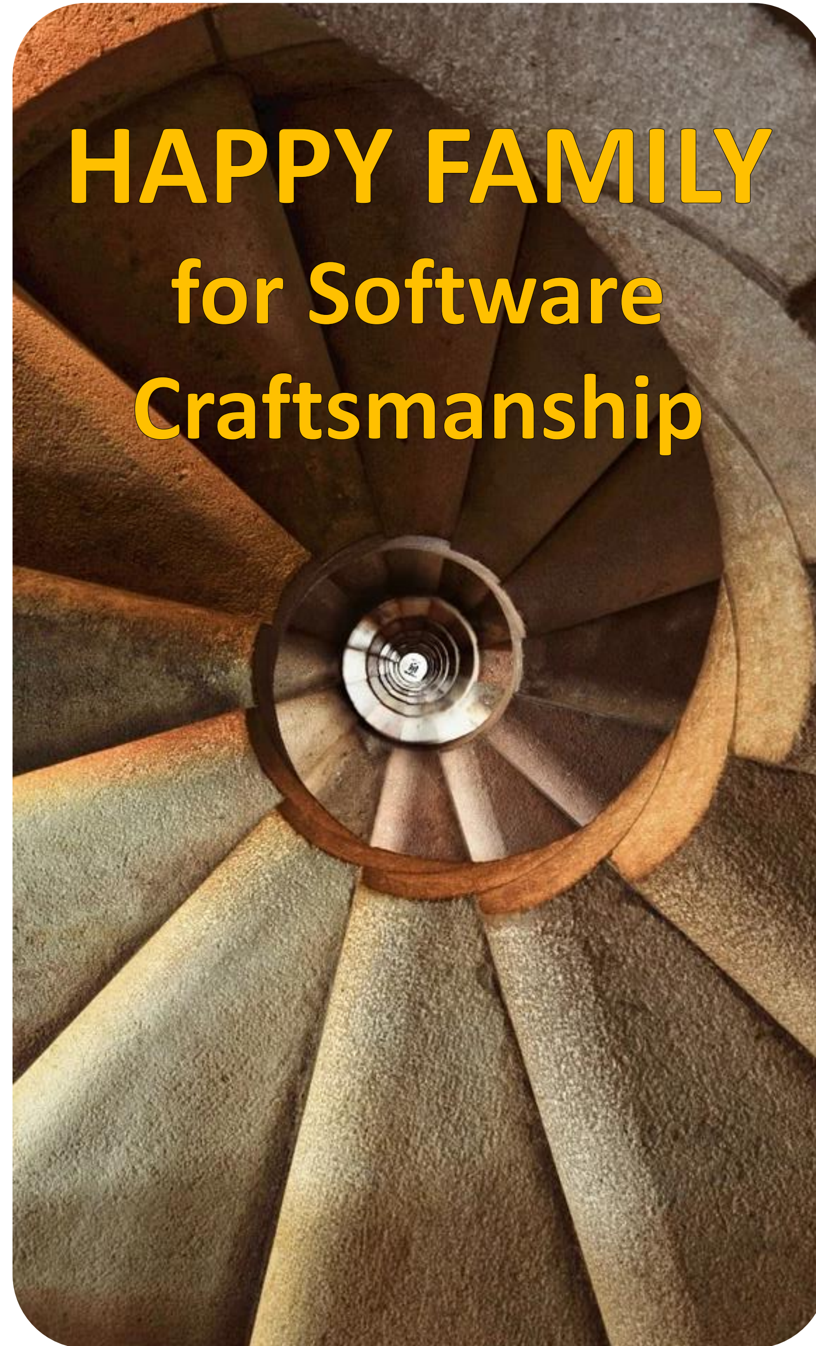
Relatives:

- Always have a Running System
- The Three Laws of TDD
- Migrate Data
- Small Refactoring's
- It is Easy to Remove

Foes:

- Duplication
- Rigidity
- *Needless Complexity*

REFACTORING



1 Naming is a process

The easiest approach I've yet found for finding good names is to progress along a series of regular steps. The steps a name goes through are:
1- Missing → 2- Nonsense → 3- Honest → 4- Honest and Complete → 5- Does the Right Thing → 6- Intent → 7- Domain Abstraction

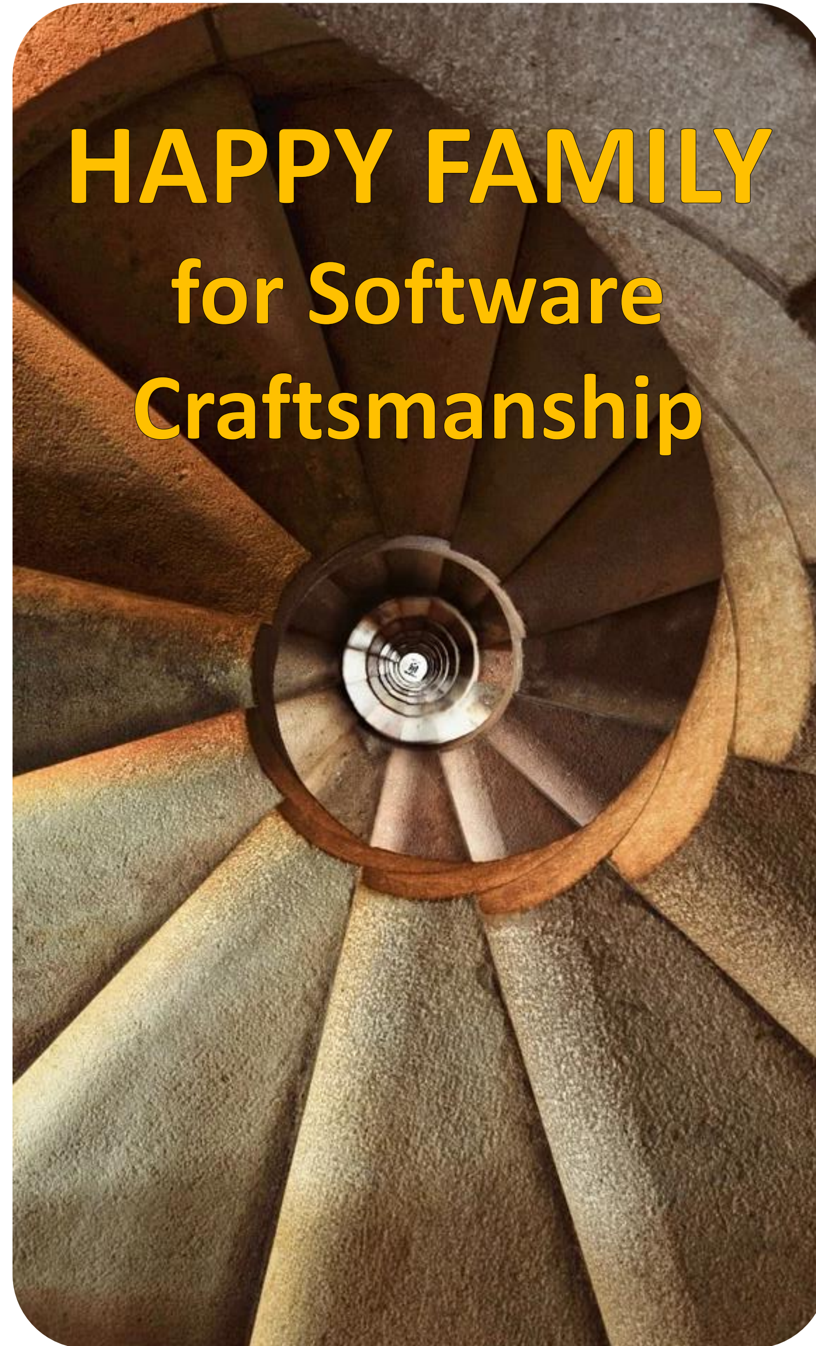
Relatives:

- *Naming is a process*
- Package Cohesion
- Do stuff or know others, but not both
- Knowing code smells
- Fail Fast

Foes:

- Magic Numbers / Strings
- Rigidity
- Data Class

CODING



2 Package Cohesion

- Release Reuse Equivalency Principle (RREP): The granule of reuse is the granule of release.
- Common Reuse Principle (CRP): Classes that are used together are packaged together.
- Common Closure Principle (CCP): Classes that change together are packaged together.

Relatives:

- Naming is a process
- *Package Cohesion*
- Do stuff or know others, but not both
- Knowing code smells
- Fail Fast

Foes:

- Magic Numbers / Strings
- Rigidity
- Data Class

CODING

3

Do stuff or know others, but not both

It is a simple approach to Domain Driver Design (DDD) tactical patterns. The role of the tactical patterns in DDD is to manage complexity and ensure clarity of behavior within the domain model.

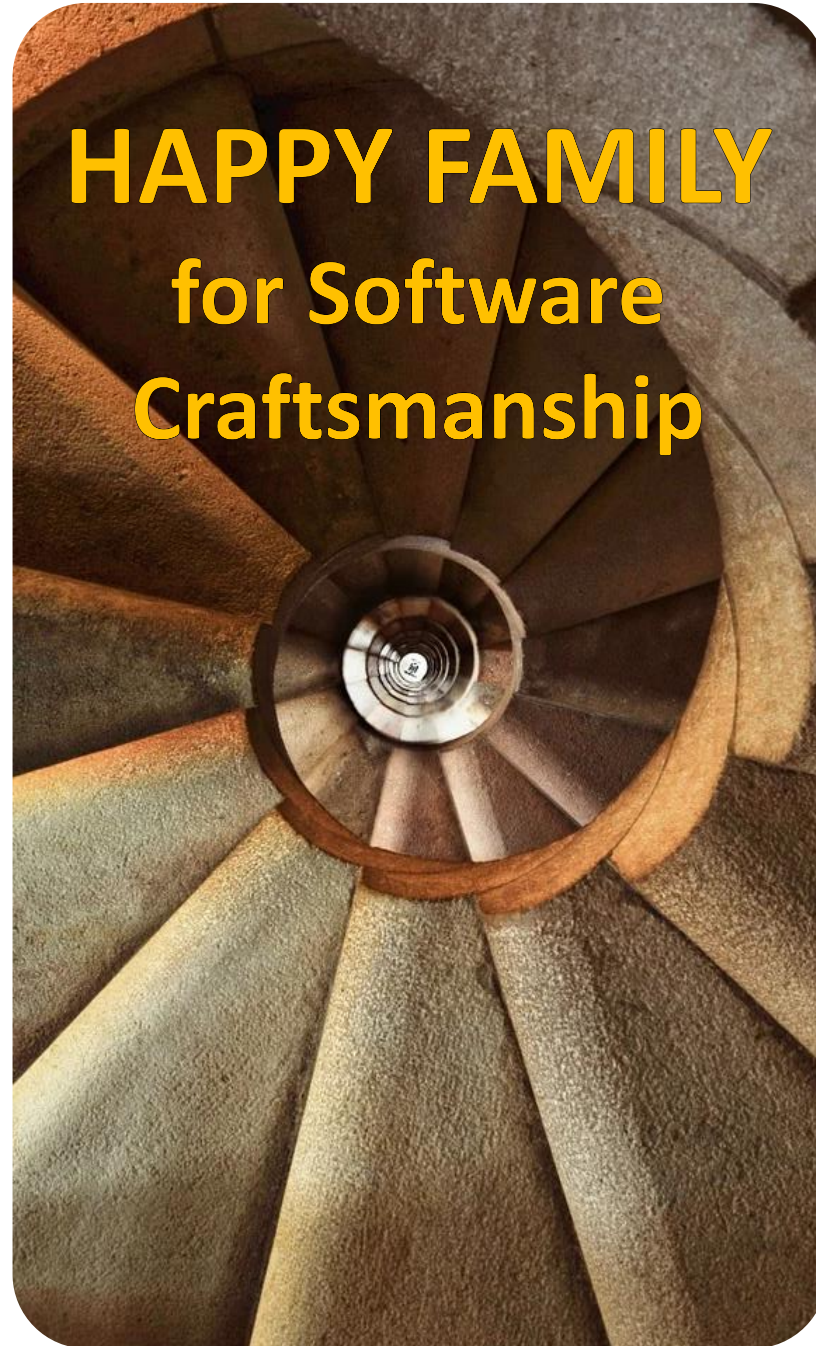
Relatives:

- Naming is a process
- Package Cohesion
- *Do stuff or know others, but not both*
- Knowing code smells
- Fail Fast

Foes:

- Magic Numbers / Strings
- Rigidity
- Data Class

CODING



4 Knowing code smells

Code smell, also known as bad smell, in computer programming code, refers to any symptom in the source code of a program that possibly indicates a deeper problem.
A Code smell deserves a refactoring to remove it. Each code smell has a subset of applicable refactoring techniques

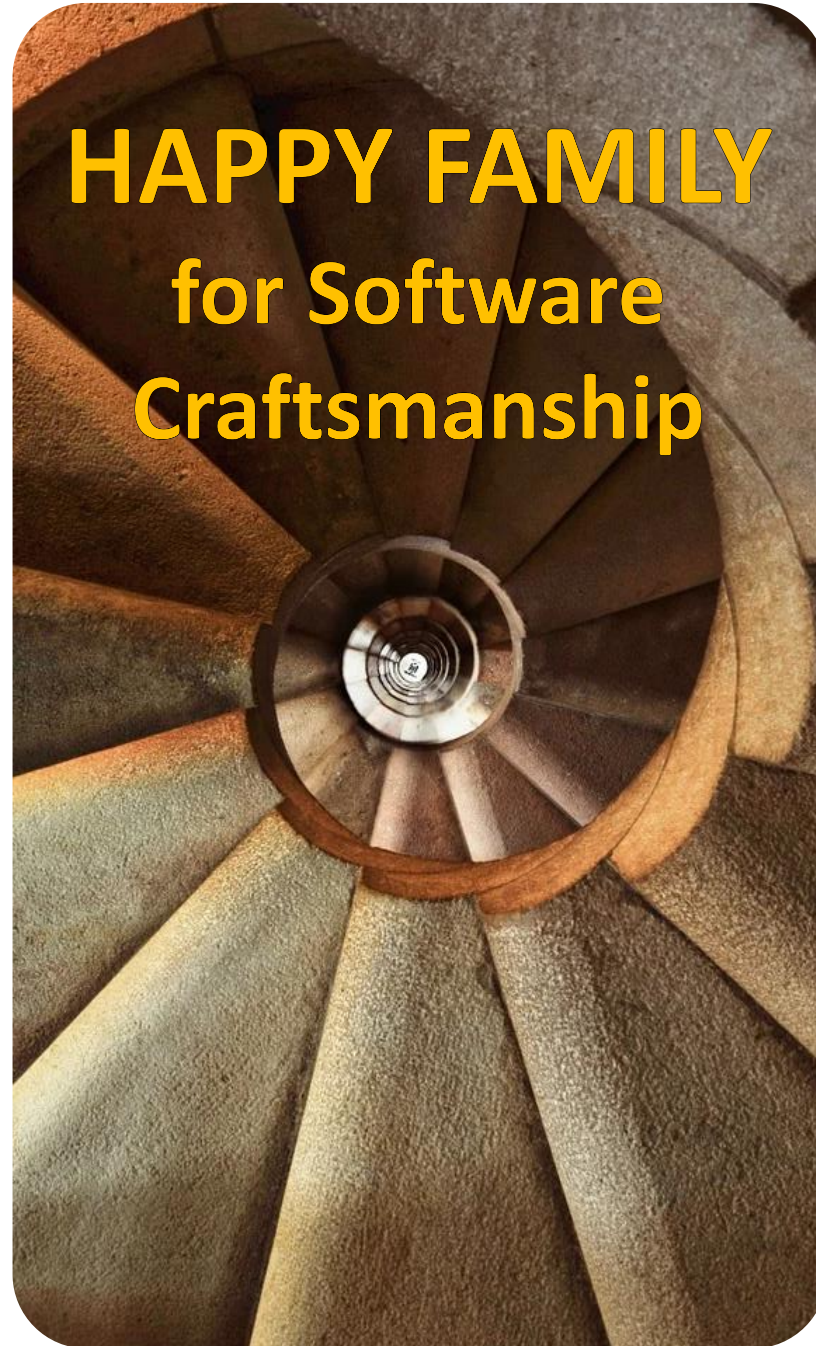
Relatives:

- Naming is a process
- Package Cohesion
- Do stuff or know others, but not both
- *Knowing code smells*
- Fail Fast

Foes:

- Magic Numbers / Strings
- Rigidity
- Data Class

CODING



5

Fail Fast

Exceptions should be thrown as early as possible after detecting an exceptional case (e.g. undefined behavior or invalid values). This helps to pinpoint the exact location of the problem by looking at the stack trace of the exception.
An Error Handler service could also be triggered to track the exception with value and calling object or service.

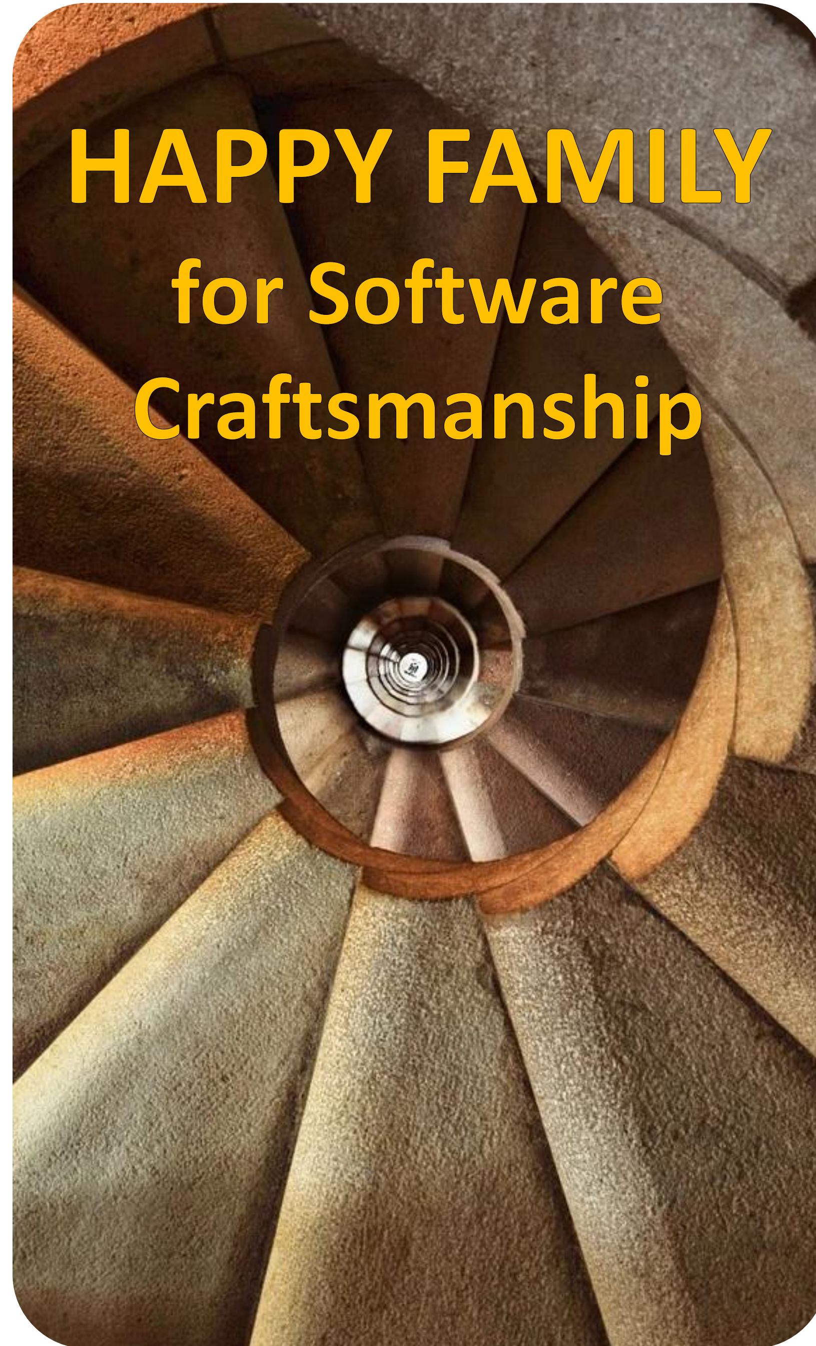
Relatives:

- Naming is a process
- Package Cohesion
- Do stuff or know others, but not both
- Knowing code smells
- *Fail Fast*

Foes:

- Magic Numbers / Strings
- Rigidity
- Data Class

CODING



6 Magic Numbers / Strings

Replace Magic Numbers and Strings with named constants to give them a meaningful name when meaning cannot be derived from the value itself.

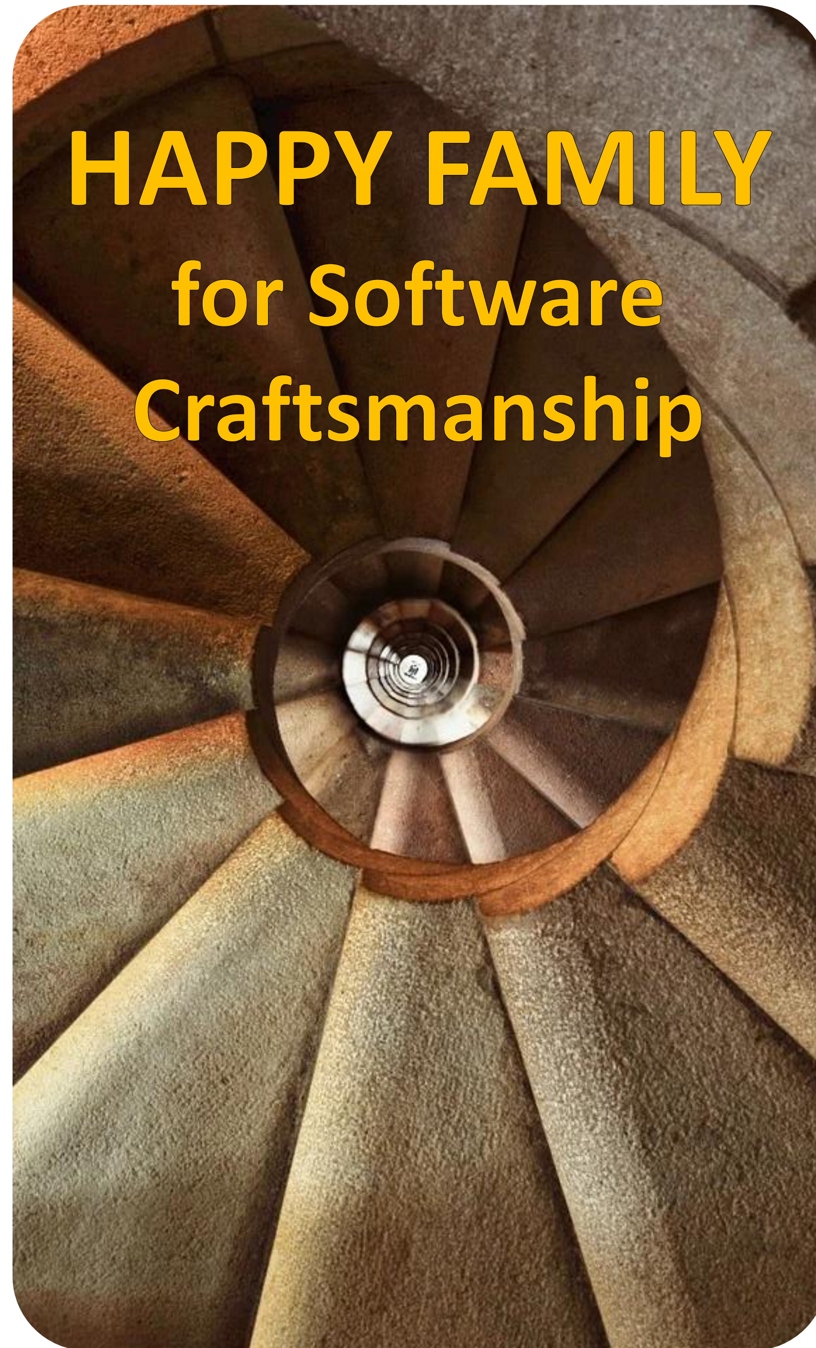
Relatives:

- Naming is a process
- Package Cohesion
- Do stuff or know others, but not both
- Knowing code smells
- Fail Fast

Foes:

- *Magic Numbers / Strings*
- Rigidity
- Data Class

CODING



7

Rigidity

The software is difficult to change. A small change causes a cascade of subsequent changes.

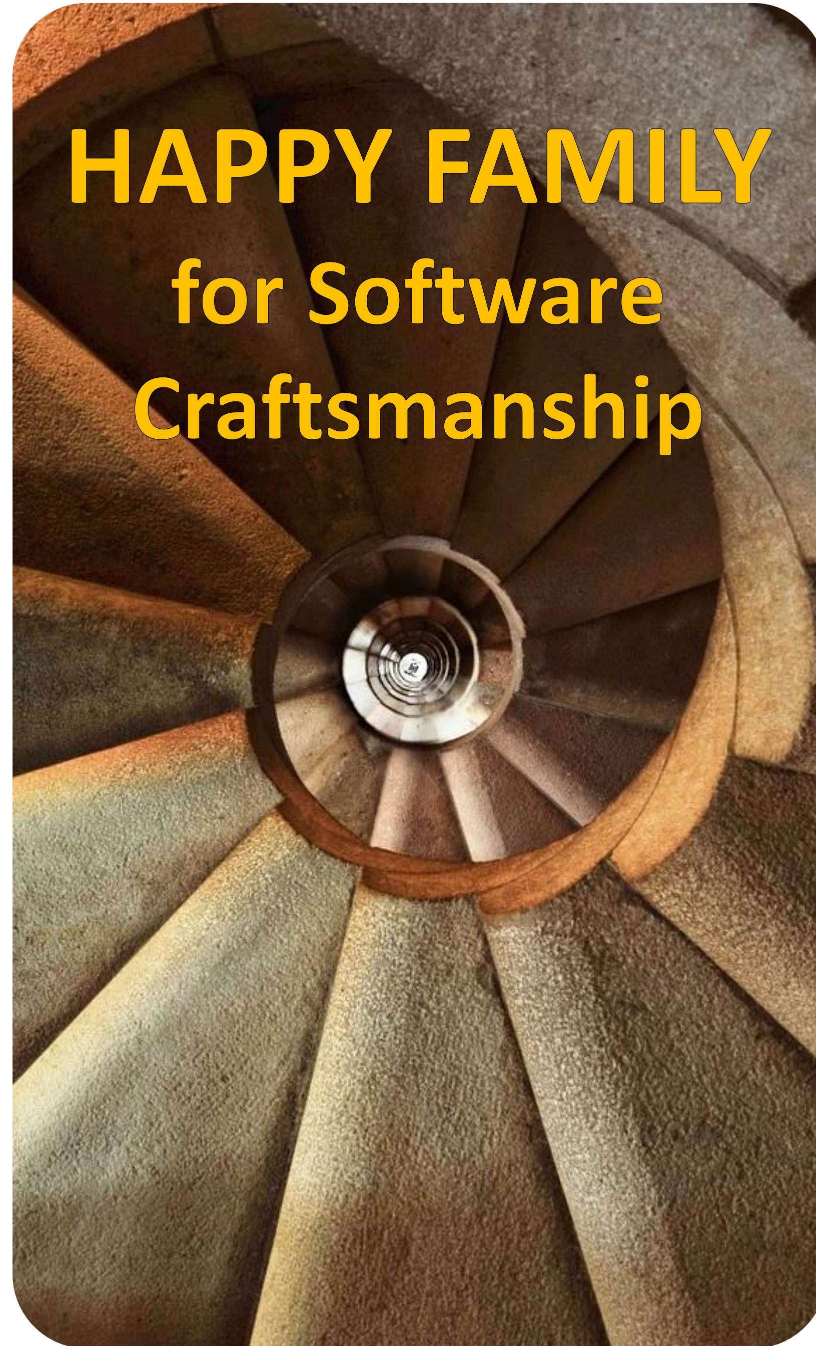
Relatives:

- Naming is a process
- Package Cohesion
- Do stuff or know others, but not both
- Knowing code smells
- Fail Fast

Foes:

- Magic Numbers / Strings
- *Rigidity*
- Data Class

CODING



8

Data Class

A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own.

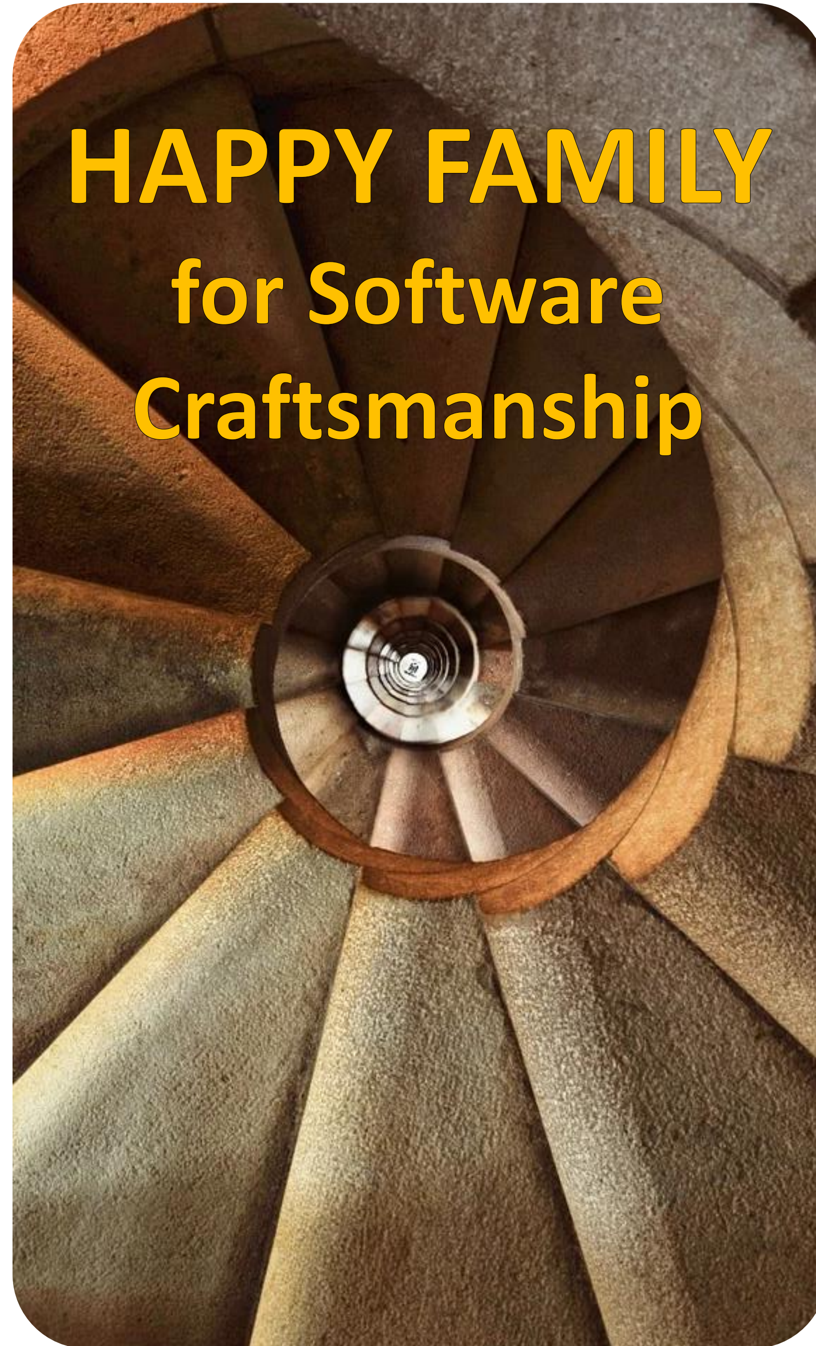
Relatives:

- Naming is a process
- Package Cohesion
- Do stuff or know others, but not both
- Knowing code smells
- Fail Fast

Foes:

- Magic Numbers / Strings
- Rigidity
- *Data Class*

CODING



1 Uncle Bob's Programmer's Oath

- *1-I will not produce harmful code.*
- *2-The code that I produce will always be my best work.*
- *3-I will not knowingly allow code that is defective [...]*

This oath includes 9 items - Read the whole oath at https://tinyurl.com/programmers-oath

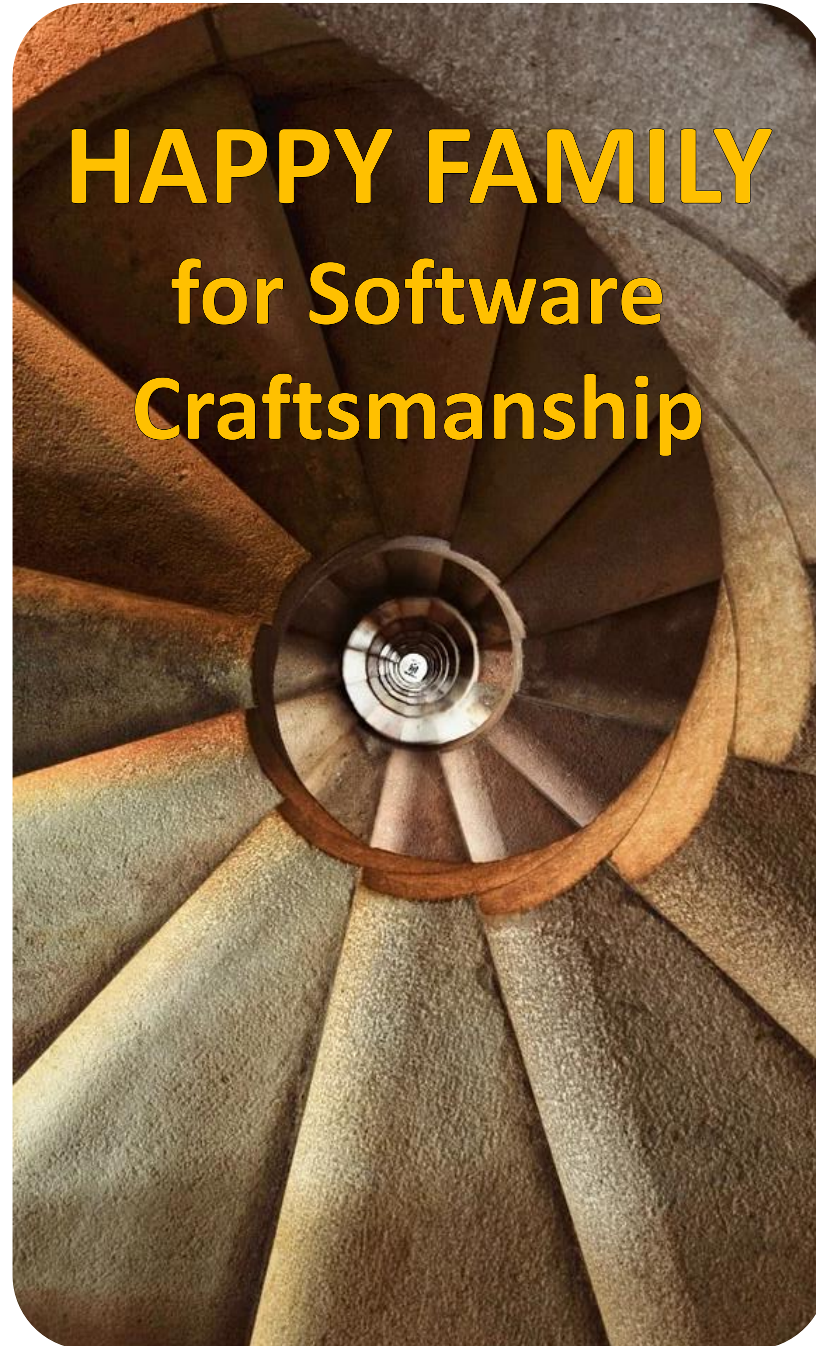
Relatives:

- *Uncle Bob's Programmer's Oath*
- Keep it Simple, Stupid (KISS)
- Boy Scout Rule
- Continuous Learning Process
- Humility

Foes:

- Being Arbitrary
- Violate the principle of Least Astonishment
- Inconsistency

MINDSET



2 Keep it Simple, Stupid (KISS)

Simpler is always better. Reduce complexity as much as possible.
This often goes along with YAGNI (*You Ain't Gonna Need It*).

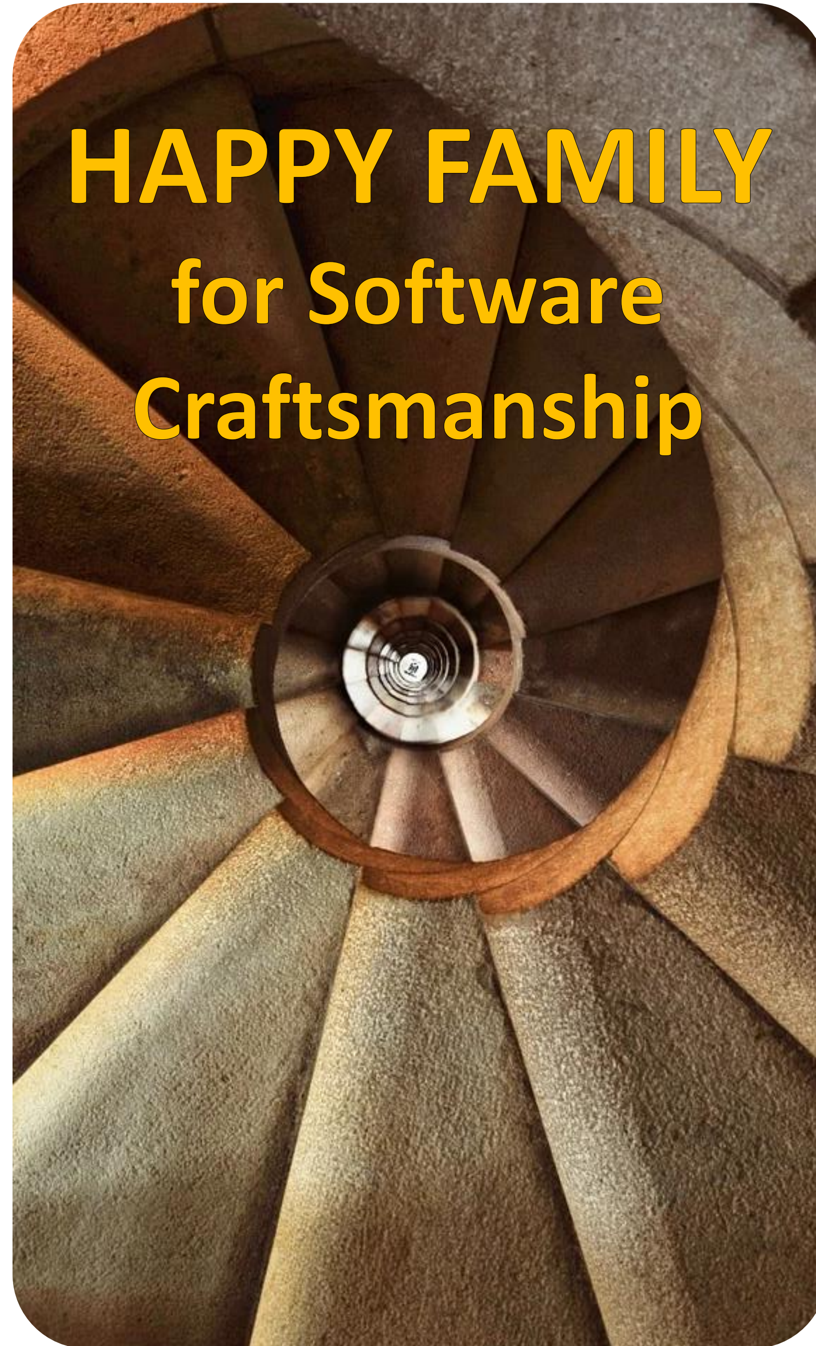
Relatives:

- Uncle Bob's Programmer's Oath
- *Keep it Simple, Stupid (KISS)*
- Boy Scout Rule
- Continuous Learning Process
- Humility

Foes:

- Being Arbitrary
- Violate the principle of Least Astonishment
- Inconsistency

MINDSET



3 Boy Scout Rule

«There are 2 things a boy scout leaves behind him: nothing and "thank you" »
It means leaving the campground cleaner than you found it; it does not need to be perfect but at least the cumulated efforts, even the smallest, will make the place nicer!

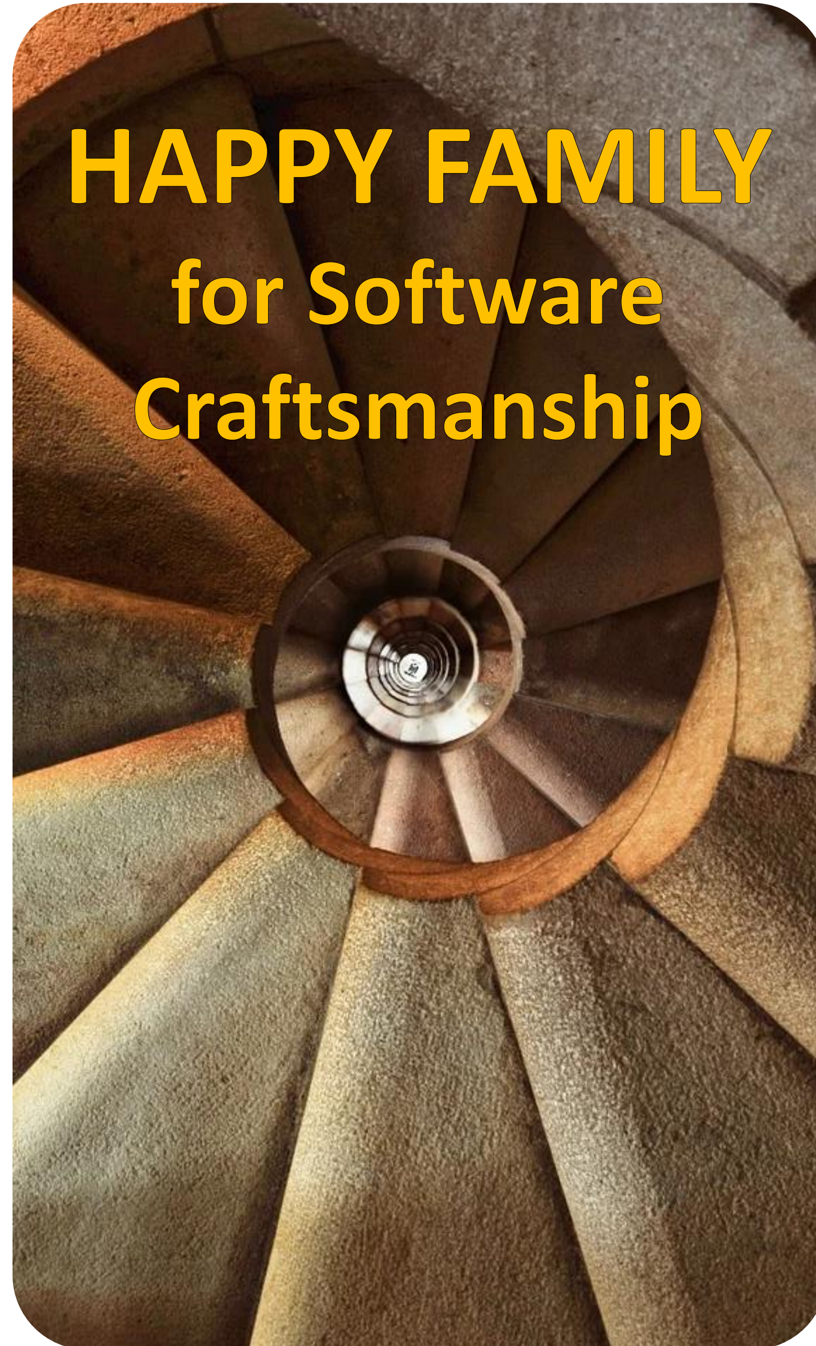
Relatives:

- Uncle Bob's Programmer's Oath
- Keep it Simple, Stupid (KISS)
- *Boy Scout Rule*
- Continuous Learning Process
- Humility

Foes:

- Being Arbitrary
- Violate the principle of Least Astonishment
- Inconsistency

MINDSET



4 Continuous Learning Process

As a “Knowledge Worker”, continuing innovation is part of your work. This is your responsibility.
You also need to develop a proper environment (attending dojos, reading books, …) and unlock your intrinsic motivation.

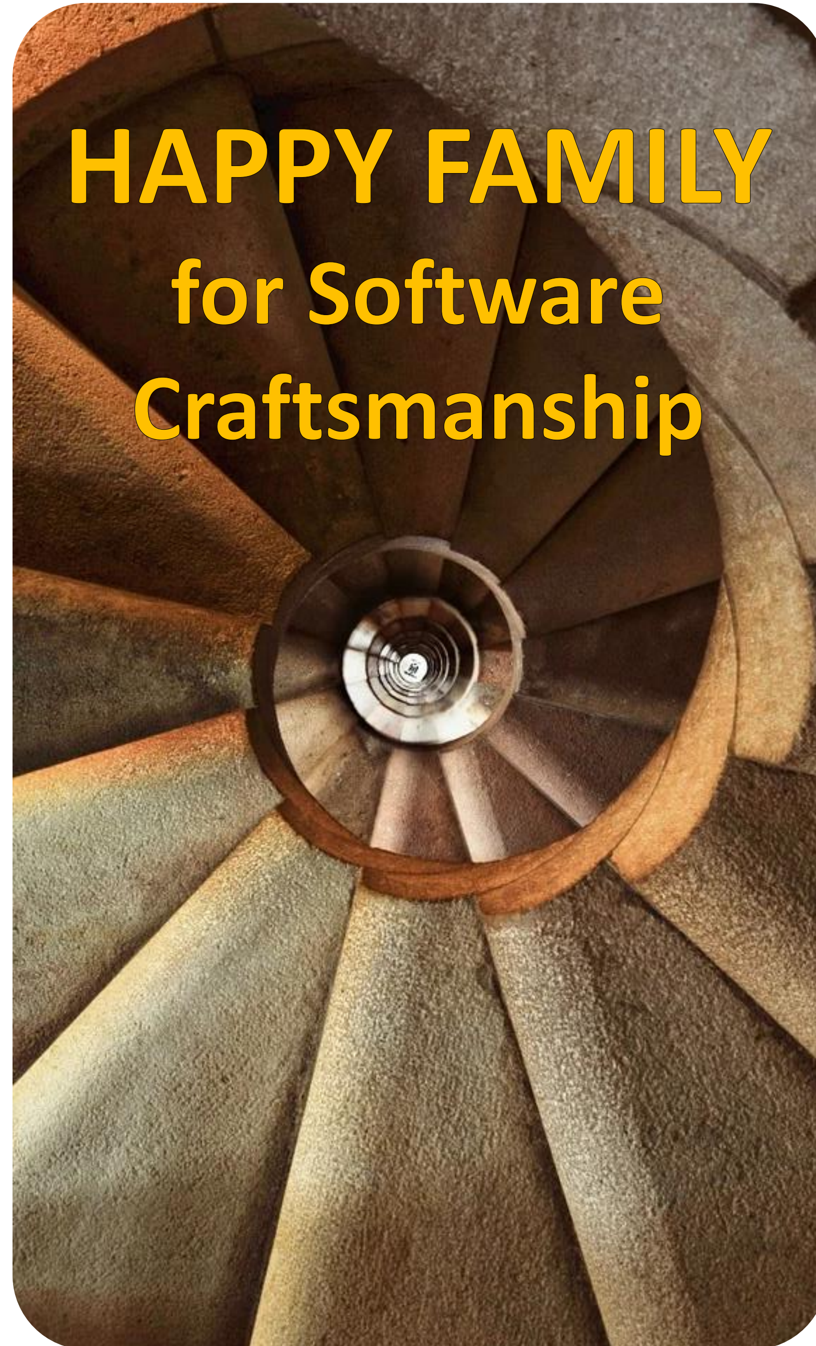
Relatives:

- Uncle Bob’s Programmer’s Oath
- Keep it Simple, Stupid (KISS)
- Boy Scout Rule
- *Continuous Learning Process*
- Humility

Foes:

- Being Arbitrary
- Violate the principle of Least Astonishment
- Inconsistency

MINDSET



5 Humility

Issues found today come from solutions found yesterday so don't blame bad solutions.
Fix what needs to be fixed and try to improve legacy as much as possible.
Good Samaritan and comprehension provide the proper collaborative environment for a Team.

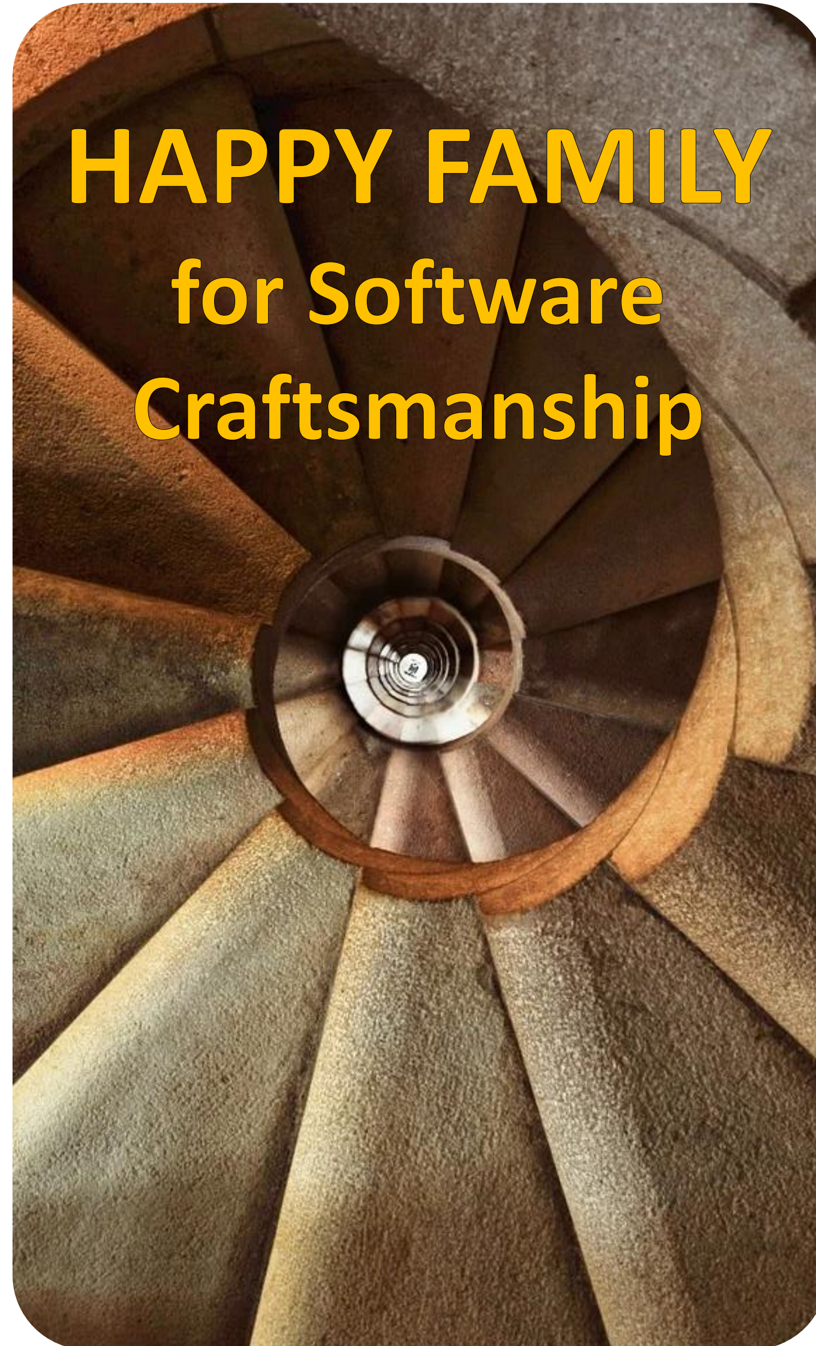
Relatives:

- Uncle Bob's Programmer's Oath
- Keep it Simple, Stupid (KISS)
- Boy Scout Rule
- Continuous Learning Process
- *Humility*

Foes:

- Being Arbitrary
- Violate the principle of Least Astonishment
- Inconsistency

MINDSET



6 Being Arbitrary

Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it.

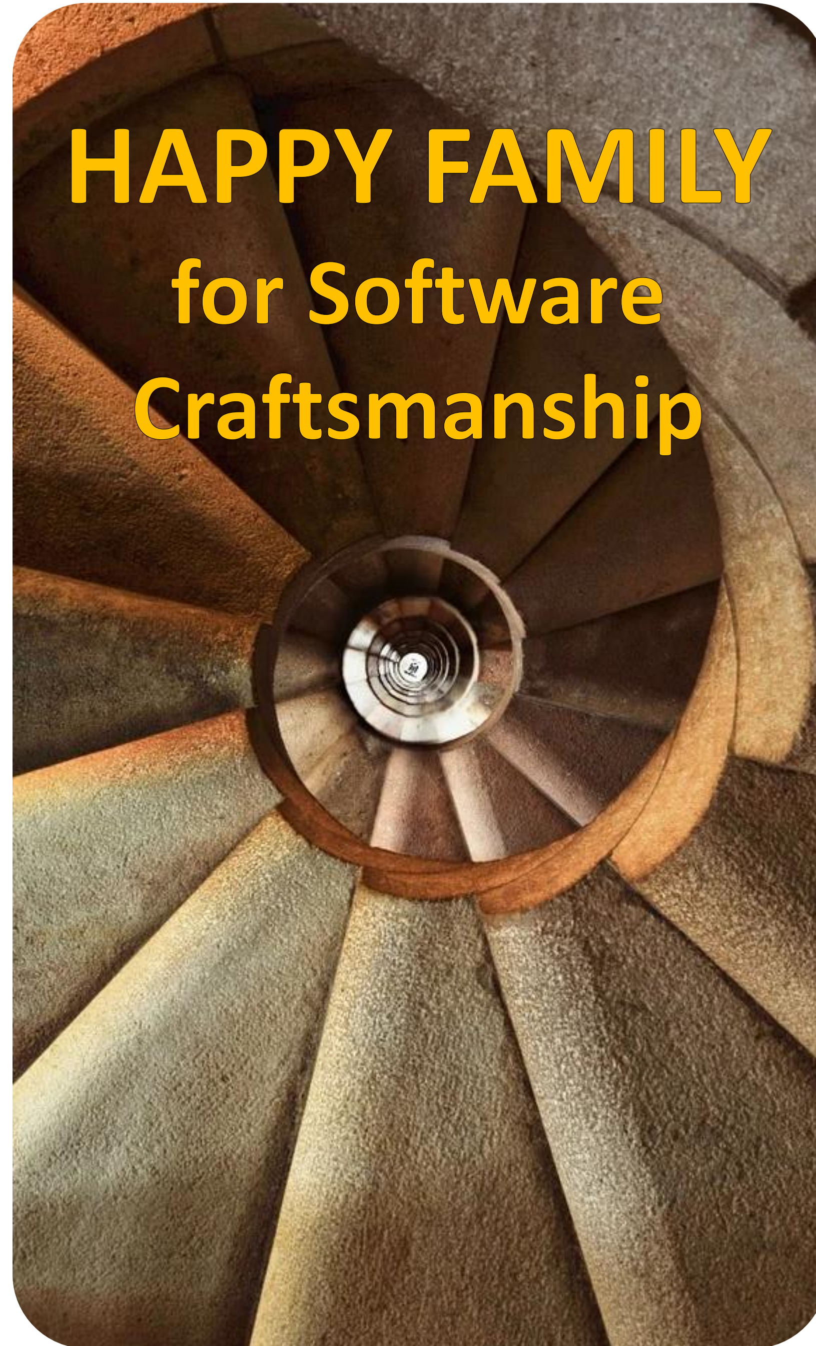
Relatives:

- Uncle Bob's Programmer's Oath
- Keep it Simple, Stupid (KISS)
- Boy Scout Rule
- Continuous Learning Process
- Humility

Foes:

- *Being Arbitrary*
- Violate the principle of Least Astonishment
- Inconsistency

MINDSET



7 Violate the principle of Least Astonishment

If a component of a system should behave in a way that most users will expect it to behave; the behavior and structure should not astonish or surprise users.

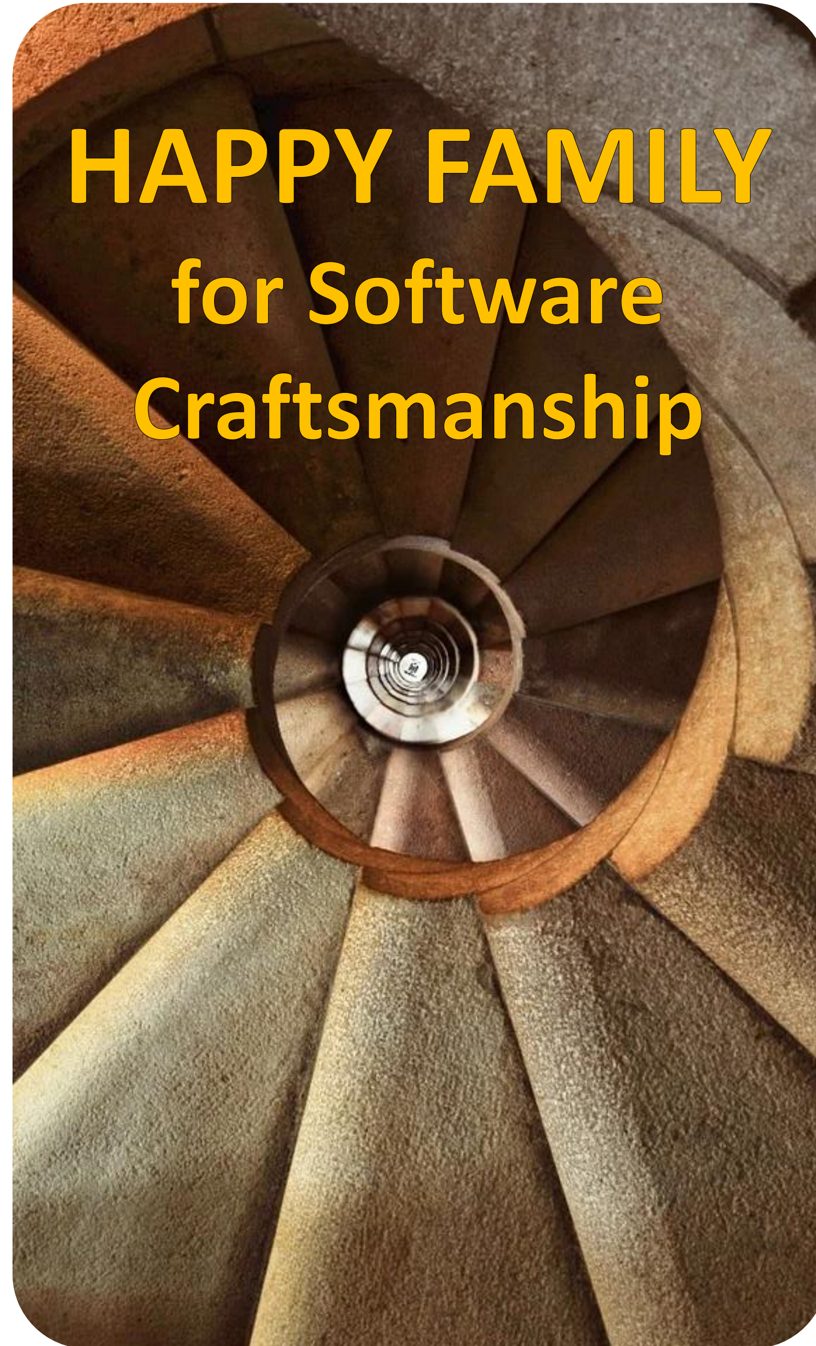
Relatives:

- Uncle Bob's Programmer's Oath
- Keep it Simple, Stupid (KISS)
- Boy Scout Rule
- Continuous Learning Process
- Humility

Foes:

- Being Arbitrary
- *Violate the principle of Least Astonishment*
- Inconsistency

MINDSET



8 Inconsistency

If you do something a certain way, do all similar things in the same way: same variable name for same concepts, same naming pattern for corresponding concepts.

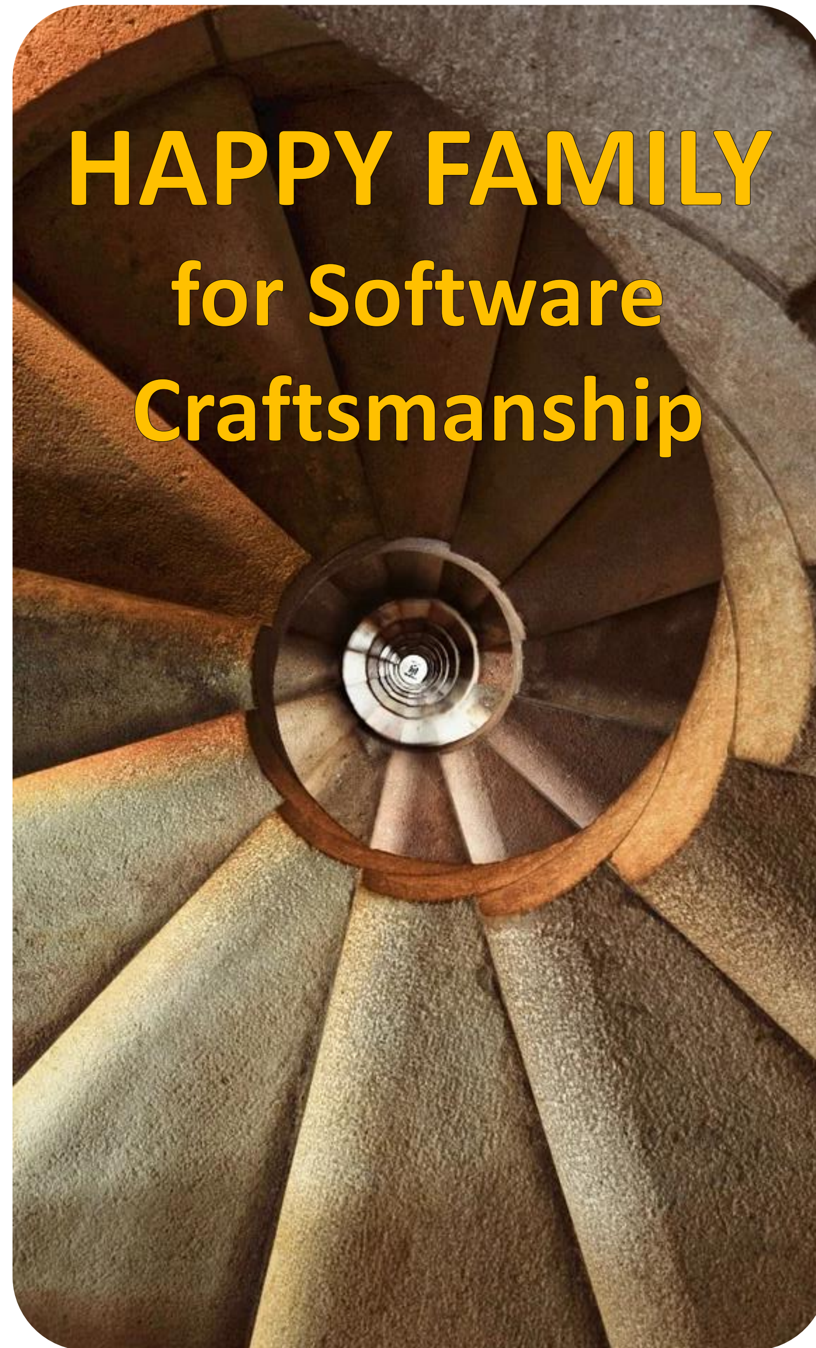
Relatives:

- Uncle Bob's Programmer's Oath
- Keep it Simple, Stupid (KISS)
- Boy Scout Rule
- Continuous Learning Process
- Humility

Foes:

- Being Arbitrary
- Violate the principle of Least Astonishment
- *Inconsistency*

MINDSET



1 Single responsibility principle (SRP)

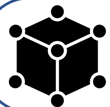
A class should have one, and only one, reason to change.
Associated smells: Large Class (more than 30 methods) / Duplicated Code / "Shotgun surgery" to change application / Divergent Change (the class changes more frequently than other classes in the application)

Relatives:

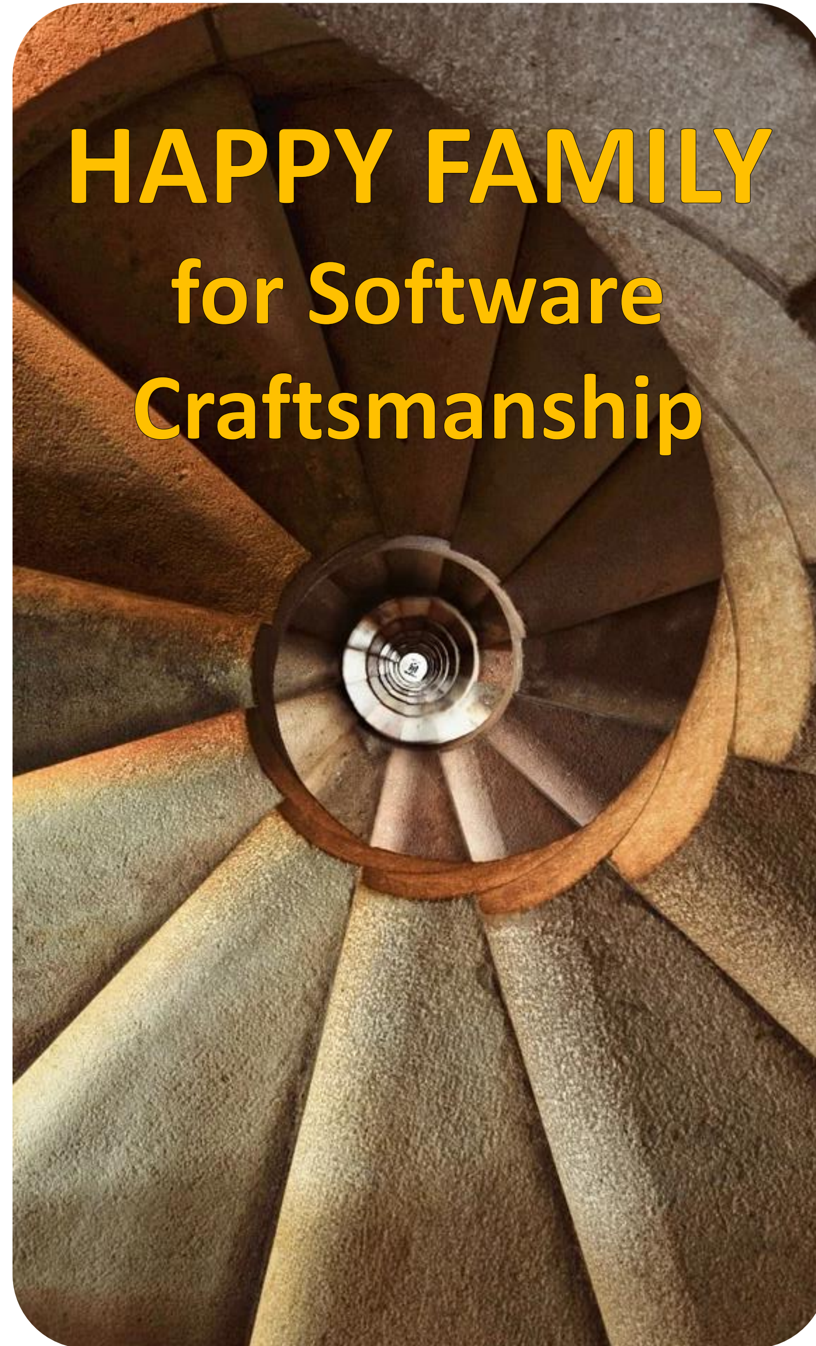
- *Single responsibility principle (SRP)*
- Open–closed principle (OCP)
- Liskov's substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Foes:

- Shotgun surgery
- Abrupt API Change
- Feature Envy



SOLID



2 Open–closed principle (OCP)

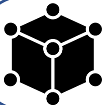
Software entities should be open for extension, but closed for modification.
You should be able to extend a classes behavior, without modifying it.
You should not alter an existing API behavior but rather extending the component with new APIs.
The class needs to change for more than one reason.

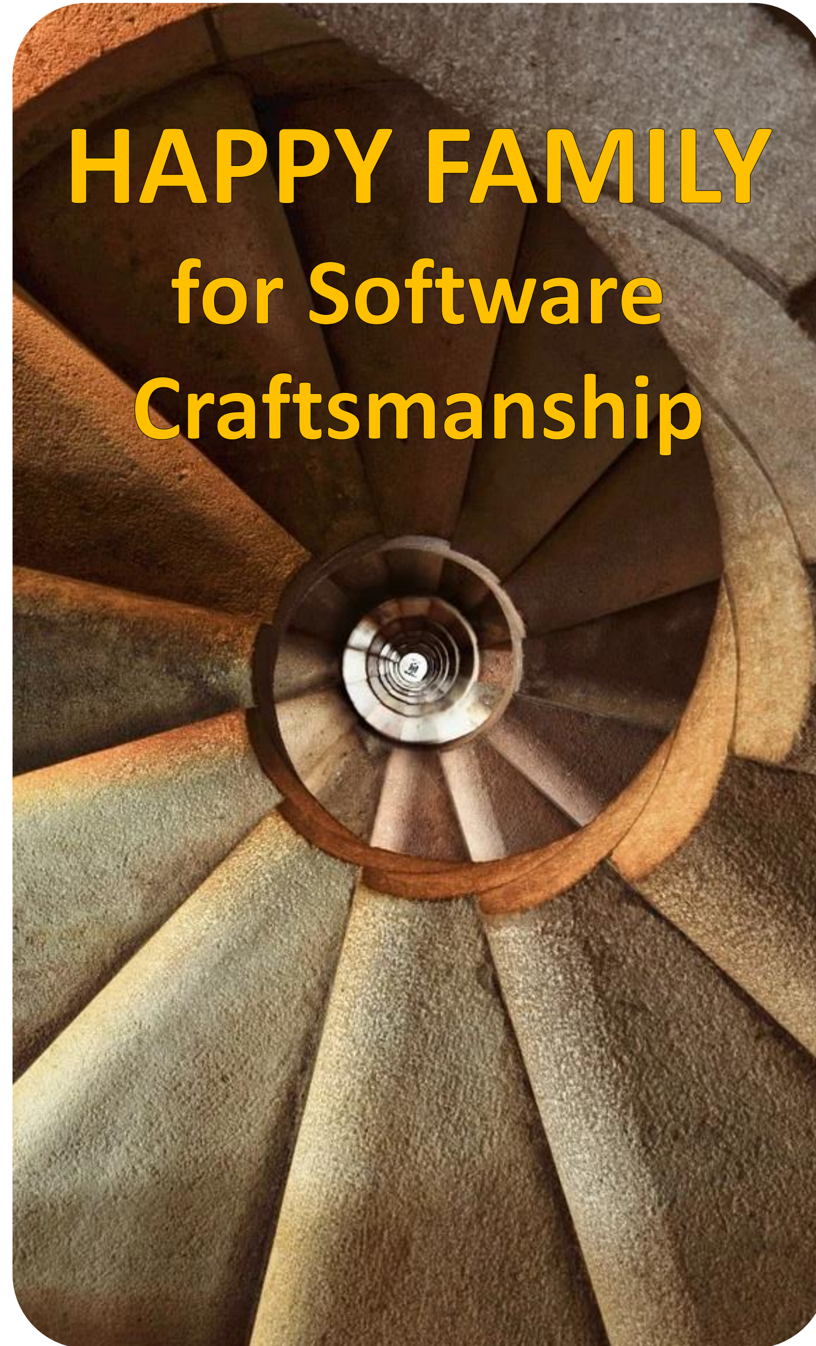
Relatives:

- Single responsibility principle (SRP)
- *Open–closed principle (OCP)*
- Liskov's substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Foes:

- Shotgun surgery
- Abrupt API Change
- Feature Envy


SOLID



3

Liskov substitution principle (LSP)

Functions that use references to base classes must be able to use objects of derived classes without knowing it.
Associated smells: explicit casting / You are not using the base class without knowledge of the derived classes / Preconditions cannot be strengthened in a subtype / Postconditions cannot be weakened in a subtype.

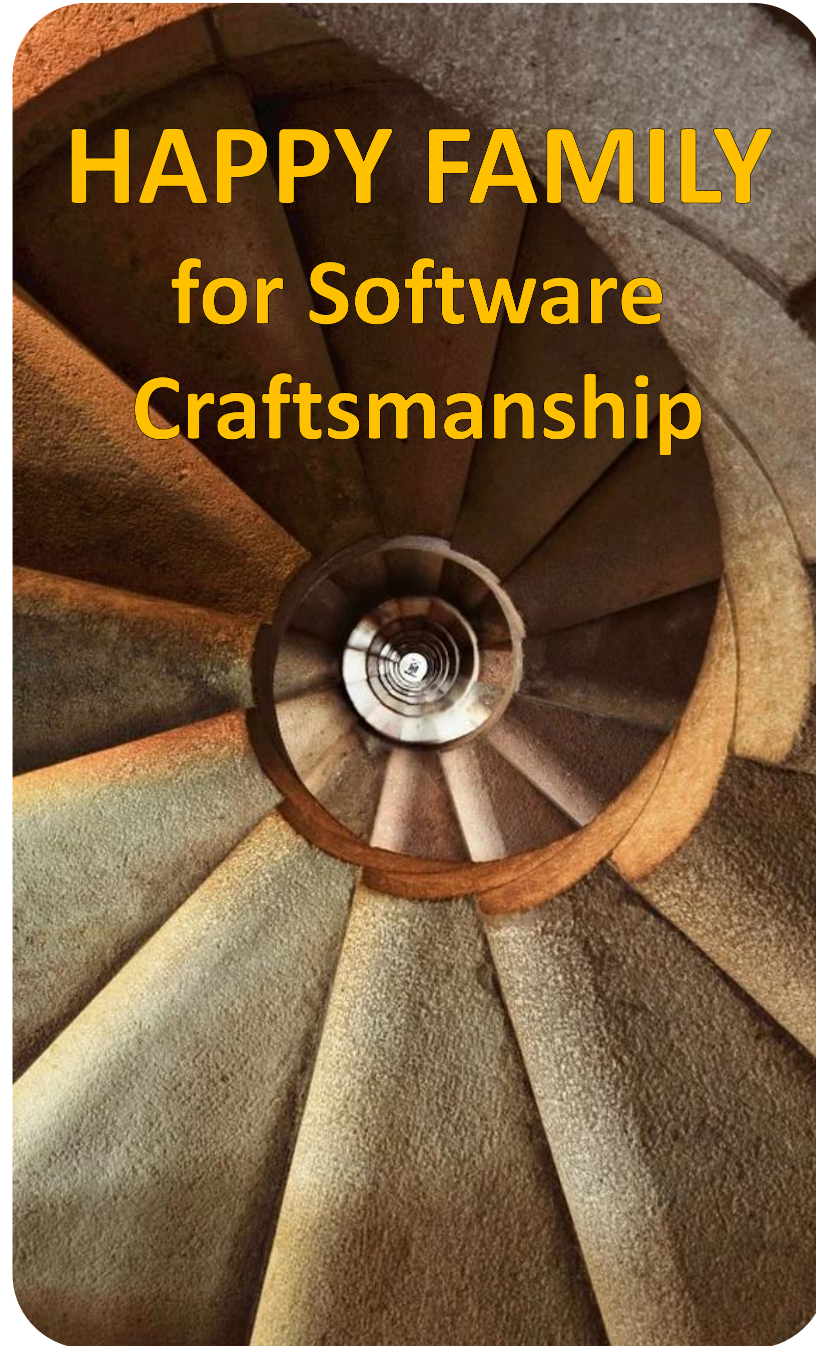
Relatives:

- Single responsibility principle (SRP)
- Open–closed principle (OCP)
- *Liskov's substitution principle (LSP)*
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Foes:

- Shotgun surgery
- Abrupt API Change
- Feature Envy

SOLID



4 Interface segregation principle (ISP)

Many client specific interfaces are better than one general purpose interface. Some code that violates this principle will be easy to identify due to having interfaces with a lot of methods on. This principle compliments SRP, as you may see that an interface with many methods is actually responsible for more than one area of functionality.

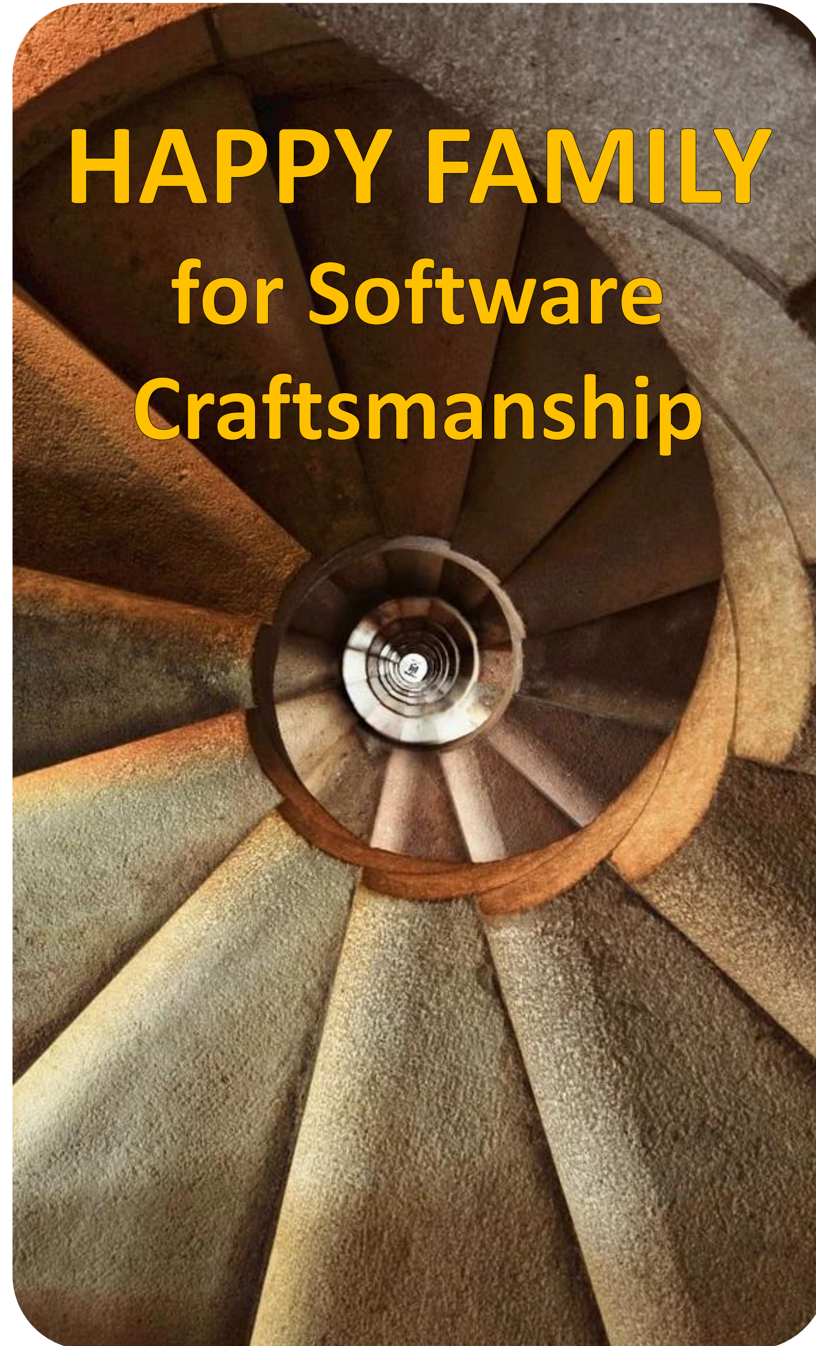
Relatives:

- Single responsibility principle (SRP)
- Open–closed principle (OCP)
- Liskov's substitution principle (LSP)
- *Interface segregation principle (ISP)*
- Dependency inversion principle (DIP)

Foes:

- Shotgun surgery
- Abrupt shared API Change
- Feature Envy

SOLID



5 Dependency inversion principle (DIP)

Depend on abstractions (module or duck type), not on concretions (class).

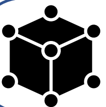
Do not depend on things that change less often than you do.

Relatives:

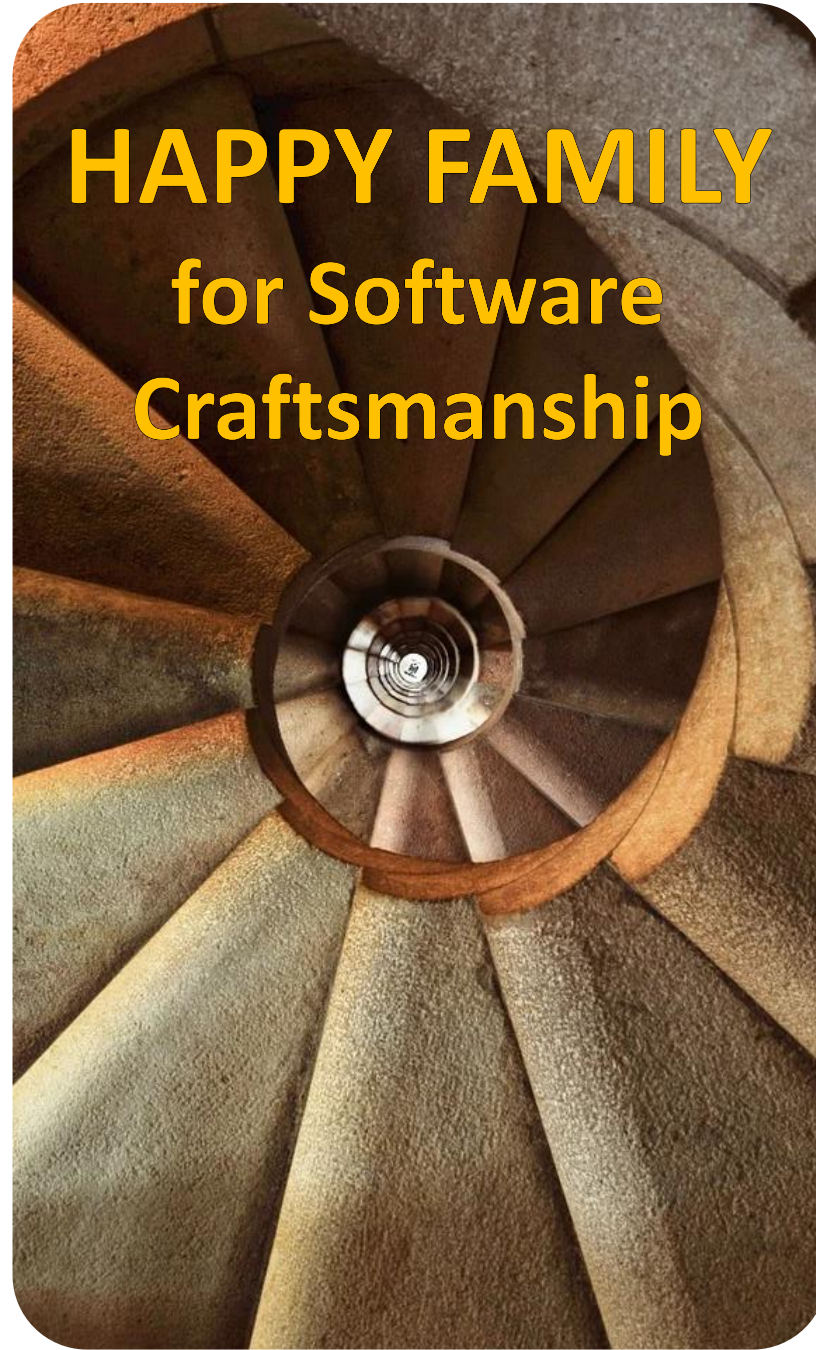
- Single responsibility principle (SRP)
- Open–closed principle (OCP)
- Liskov's substitution principle (LSP)
- Interface segregation principle (ISP)
- *Dependency inversion principle (DIP)*

Foes:

- Shotgun surgery
- Abrupt shared API Change
- Feature Envy



SOLID



6 Shotgun surgery

Making any modifications requires that you make many small changes to many different classes.

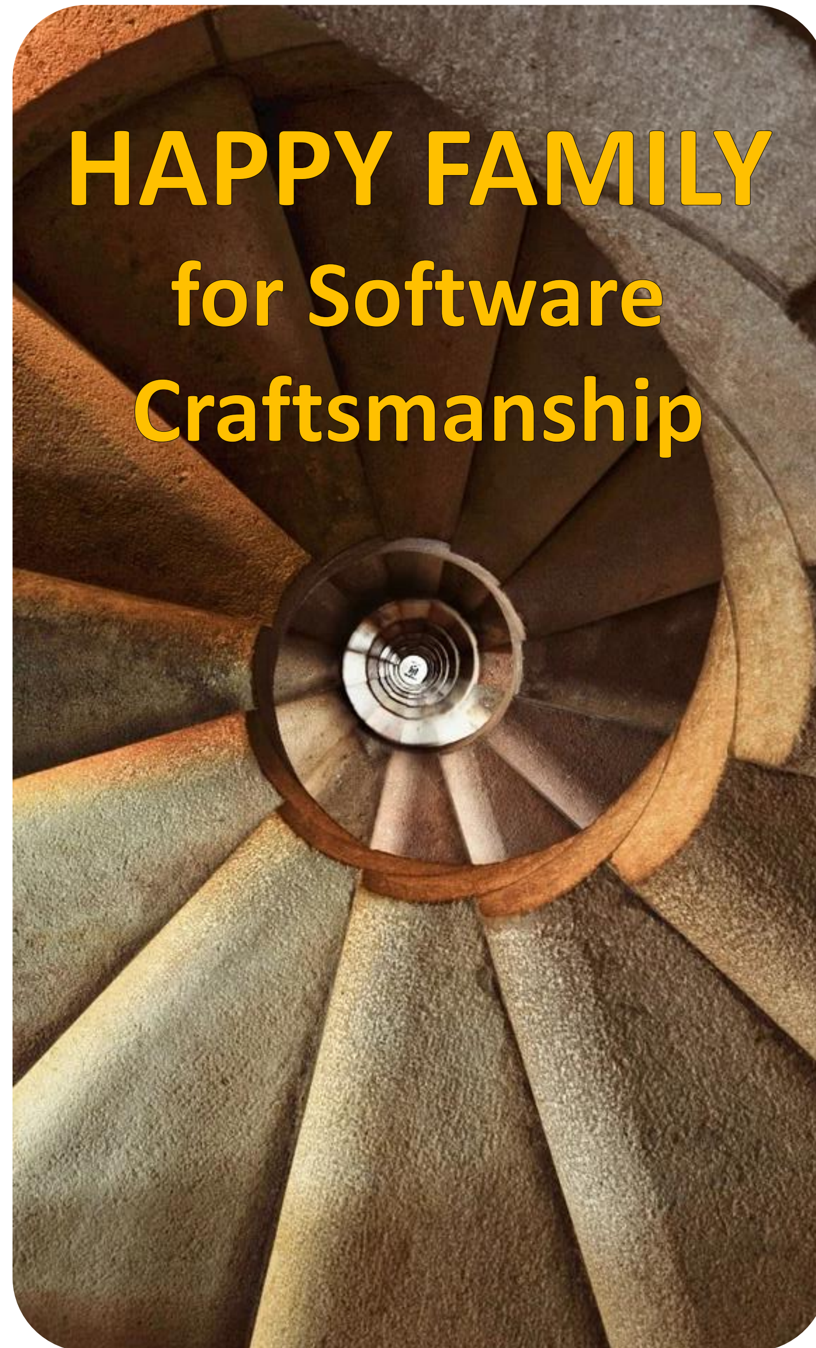
Relatives:

- Single responsibility principle (SRP)
- Open–closed principle (OCP)
- Liskov’s substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Foes:

- *Shotgun surgery*
- Abrupt shared API Change
- Feature Envy

SOLID



7 Abrupt API Change

Changing the behavior of an API or its signature without a decommissioning policy (such as deprecation notice with some delay or use measurement before being removed) will generate a lot of debugging efforts waste of time on the client side

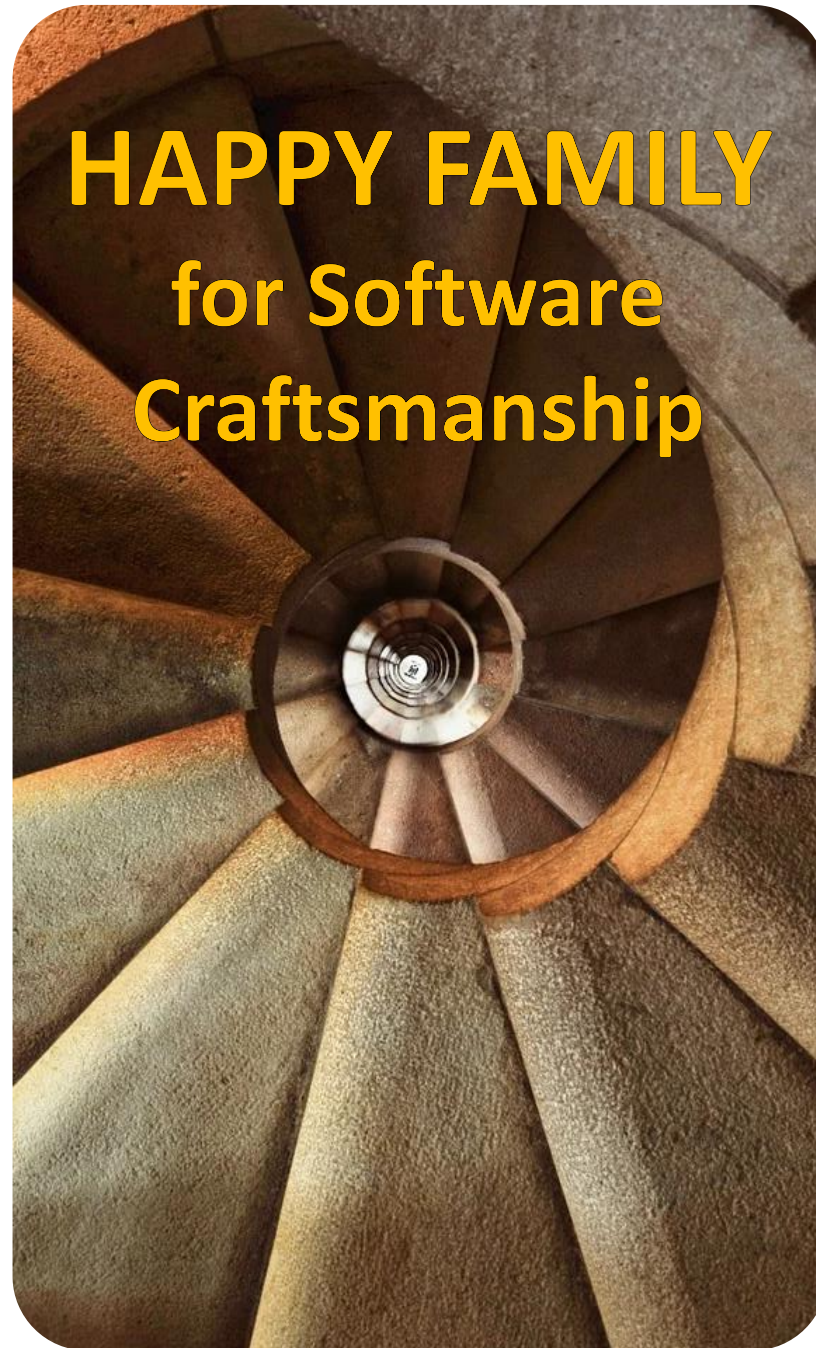
Relatives:

- Single responsibility principle (SRP)
- Open–closed principle (OCP)
- Liskov's substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Foes:

- Shotgun surgery
- *Abrupt API Change*
- Feature Envy

SOLID



8 Feature Envy

A method accesses the data of another object more than its own data. This smell may occur after fields are moved to a "data class". If this is the case, you may want to move the operations on data to this class as well.

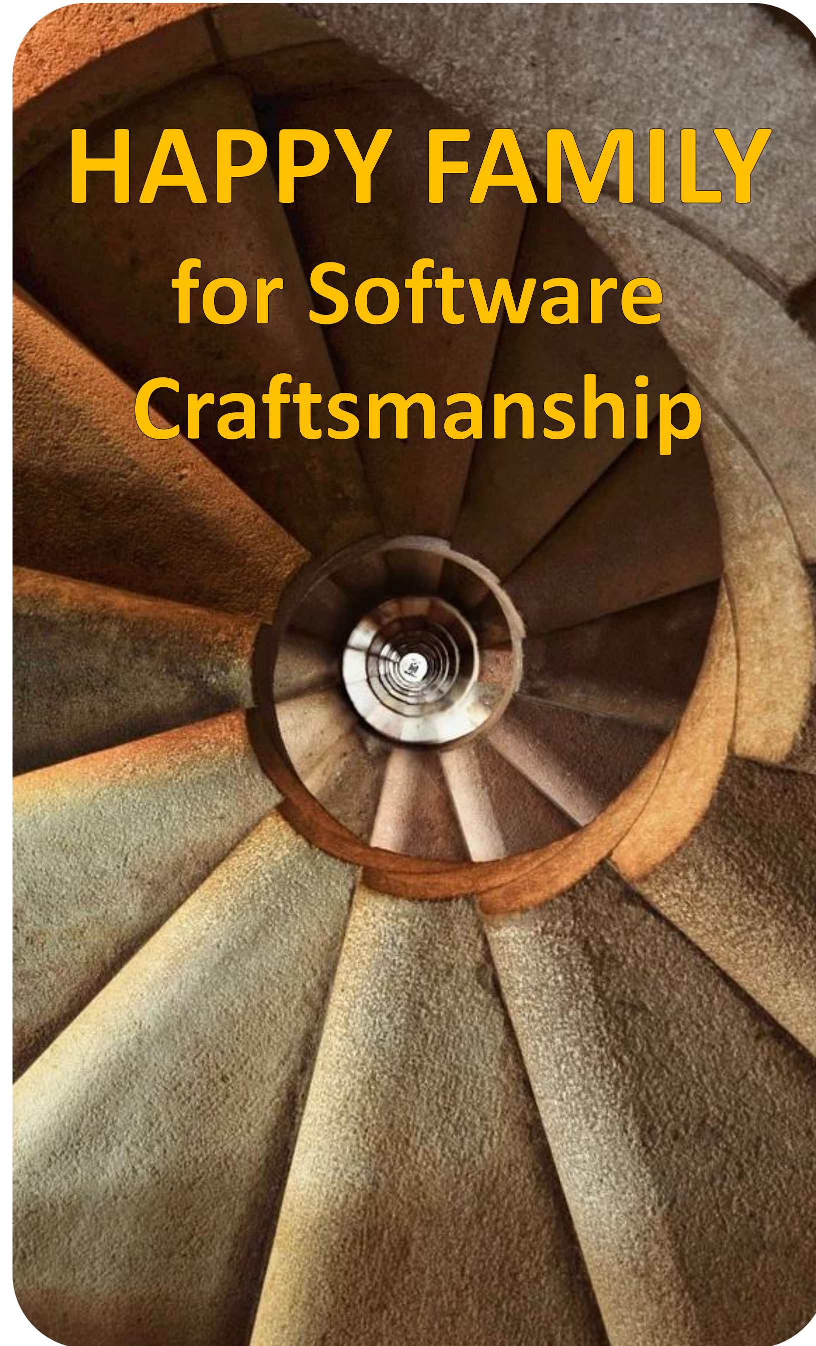
Relatives:

- Single responsibility principle (SRP)
- Open–closed principle (OCP)
- Liskov's substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

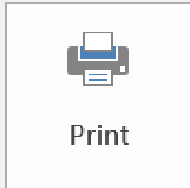
Foes:

- Shotgun surgery
- Abrupt API Change
- *Feature Envy*

SOLID



Print



Copies: 1

Printer

Global_Follow-Me_Color on nc...
Ready: 29 documents waiting

Printer Properties

Settings

Print All Slides
Print entire presentation

Slides:

Full Page Slides
Print 1 slide per page

Print One Sided
Only print on one side of the p...

Collated
1,2,3 1,2,3 1,2,3

No Staples

Color

Edit Header & Footer

\\nceprtslnxspm01\Global_Follow-Me_Color Properties

Frequently Used Settings | Detailed Settings | Configuration/About

Current Setting
User Setting

Document Size:
A4 (210 x 297 mm)
Print On:
Same as Original Size
Reduce/Enlarge:
Fit to Print Size

Settings Summary

Register Current Settings...

One Click Preset List:
Factory Default

Expand list >>

Manage...

Job Type: Normal Print — Details...

Document Size: A4 (210 x 297 mm)

Print On: Same as Original Size

Orientation: Portrait / Landscape

Paper Type: Plain & Recycled

Input Tray: Auto Tray Select

Layout: 2 Pages per Sheet

Page Order: Left-Right/Top-Bottom

2 sided: Off

Booklet: Off

Staple: Off

Punch: Off

Color/ Black and White: Color

Copies:(1 to 999) 1

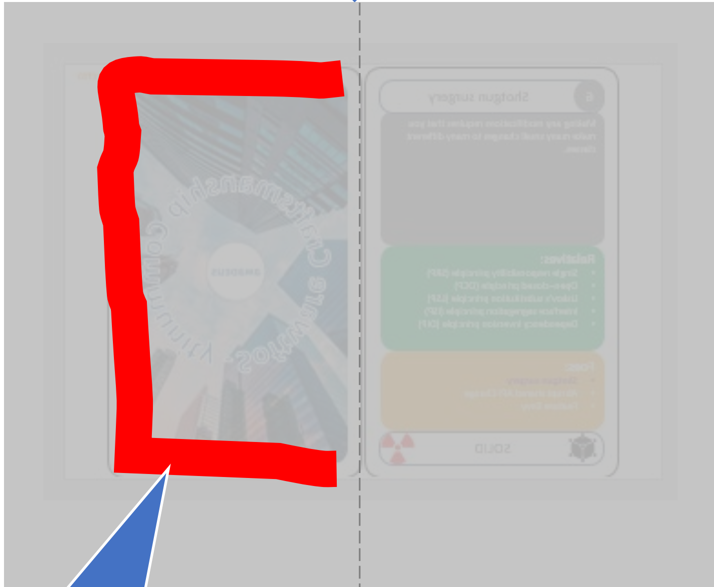
OK | Cancel | Help

1

Fold here

2

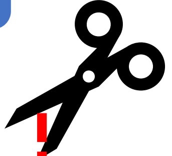
Fold here



Glue only here

3

Cut here



Note: the printing will provide 2 cards per sheet