Introduction

Design patterns

Conceptual modelling

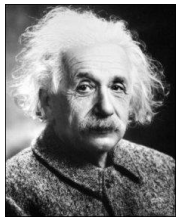
- What is the difference between **analysis** and **design**?
- These activities often end up with the same result, that is why many programmers consider them to be the same.
- Example on the analysis of the snooker (one use case):
  - Player hits with his cue the white ball. Then, white ball moves in a particular direction and speed and hit red ball under some angle. Red ball starts to move in a particular direction and speed.
- If we model only the specific situations present in the system (if the balls are distributed like this and I hit white ball like this, this will be the result), it is impossible to model the system completely.

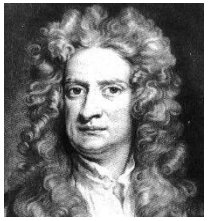
Conceptual modelling

- To fully understand the system, it is necessary a deeper investigation of the relations between movement and speed, mass and momentum, etc. After understanding of these rules, it is easier to create a simulator.
- It is easy to create a snooker simulator, since we know the laws of physics. However, if we wanted to be precise during the creation process, we would have to reveal them during the process.
- **Conceptual modelling** = mental model, that simplifies investigated system, what makes the system easier to understand.

Conceptual modelling

- To model the movement of the balls, we can use two movement models:
  1. absolutely (we do not have anything better) precise relativistic
  2. for our need suitable, but in general not precise Newtonian.





- Which model is better to utilize? The model would be absolutely precise using relativity, but is it necessary?
- **Models are not good or bad, they are more or less usable for particular situation!**
- More flexible the model is, the more complex and harder to understand and managing it is as well.

Conceptual modelling

- **Never add the functionality, you will not need!**
- Use as simple models as you can – they are easy to maintain. Often, these models will not be those, you come with first.
- Beware of thinking in a specific programming language!
- **Conceptual models are linked with interfaces, not with the specific implementations (classes)!!!**

History

- Christopher Alexander (professor of architecture, University of California, Berkeley) published several books about the patterns in architecture (the architecture that builds houses), the prototype of the books dedicated to the patterns in the software is considered his book [1].
- His idea of patterns was augmented by Kent Beck and Ward Cunningham.
- In early 90s a the conference OOPSLA was held. Bruce Anderson led the workshop oriented on the idea to create a „manual“ for software architects. Jim Coplien published such a book for C++ [2].
- These (and others) people formed a Hillside group – group that was engaged on the patterns problematics.

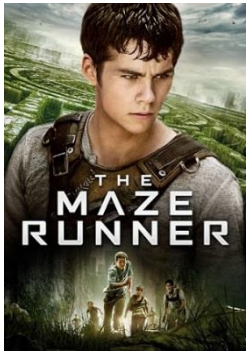
History

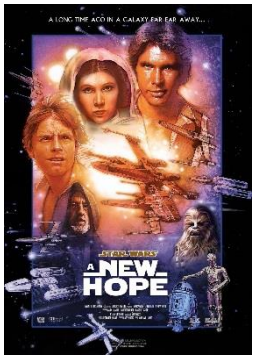
- **Gang of Four** (GoF) – book [3] is a breakthrough – patterns were brought to the common knowledge.
- Conference PLoP (Pattern Language of Programming), founded by Hillside group in 1994, helped as well. In the same year Portland Pattern Repository was founded.
- 1998 Martin **Fowler** publish his book [4].

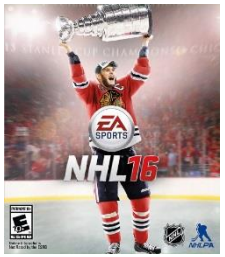
What is a (design) pattern?

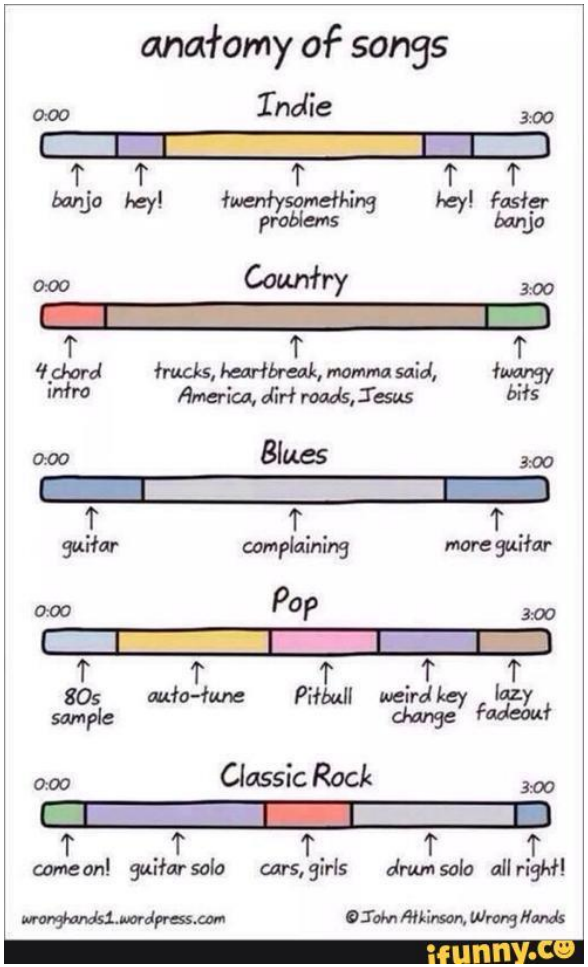
- *„Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“* Alexander
- *„The design patterns .. are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.“* GoF
- *„.. an idea that has been useful in one practical context and will probably be useful in others“.* Fowler

What is a (design) pattern?









What is a (design) pattern?

- Design patterns are not artificially created. They occur in praxis as the solutions of particular problems. If the proposed solution is applicable to other problem as well, we can talk about the pattern.
- Fowler divided patterns in two categories:
  - **Analysis** (65 patterns): group of concepts representing general conception in business modelling.
  - **Support** (21 patterns): describe how to apply analysis patterns (design patterns).
- **Design pattern is a beginning of the problem solving, not solution itself.**

Analytical vs. Design patterns

- Basic concepts are presented in [5] (here we present basic ideas).
- Analytical patterns come from business domain. They help to understand (and fill the gaps) in the system domain area – this is key for proper system design.
- Authors in [6] map several approaches of how to apply analytical and design patterns. They clasify two approaches to designing with patterns:
  - **Ad hoc application of patterns.**
    - Pattern is applied when a problem arise.
  - **Systematic application of patterns**.
    - There are systematic techniques specifying how to apply a pattern.

Systematic application of patterns

- **Pattern languages**:
  - provide set of patterns that solve problems in a specific domain.
  - capture not only the pattern, but the relationship to other patterns as well.
  - models are typically applied using method of fitting into final model (ad-hoc).
- **Development processes using general-purpose patterns**:
  - Patterns usage represent the core of the application model.
  - It defines steps, models, tools..

Ad hoc application of patterns

1. Create (analytical or design) model using standard techniques.
2. Find suitable candidates for the patterns.
3. Apply those candidates in proper parts of the model. Often, using a pattern, model will provide additional functionality, what leads to the increase of flexibility (careful – follow rule from this page!).
4. Finally, determine members to create a pattern, (what leads to instance of the pattern) and put it into application model.

Pattern Oriented Analysis and Desing

- POAD methodology [7] defines forementioned steps.
- There are three phases:
  1. **Analytical phase** – based on the requirements, patterns suitable for specific modelled area are selected.
  2. **Higher level design** – instances of the patterns selected from previous step are mutually connected using formalized principles.
  3. **Design specification** – expands (add details that were not covered) basic application model from previous step.

Analytical patterns

- Fowler categorized analytical patterns into 9 different application areas [4]:
  1. Accountability
  2. Observations and Measurements
  3. Observation for Corporate Finance
  4. Referring to Objects
  5. Inventory and Accounting
  6. Planning
  7. Trading
  8. Derivative Contracts
  9. Trading packages

Parts of design pattern

- Every pattern should consist of four basic parts:
  - **Name** is a handle we can use to describe a design problem, its
  - solutions, and consequences in a word or two
  - **Problem** describes when to apply the pattern.
  - **Solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation
  - **Consequences** are the results and trade-offs of applying the pattern. They include pattern‘s impact on a system's flexibility, extensibility, or portability.
- There are many classifications one can use: Alexander‘s, Fowler‘s, GoF, Portland form, Coplien form, POSA form..

GoF describing design patterns

- **Pattern Name and Classification** (see slide above).
- **Intent** - What does the design pattern do?
- **Also Known As -** Other well-known names for the pattern, if any
- **Motivation -** A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
- **Applicability** - What are the situations in which the design pattern can be applied?
- **Structure** – UML diagram.
- **Participants** - The classes and/or objects participating in the design pattern and their responsibilities.

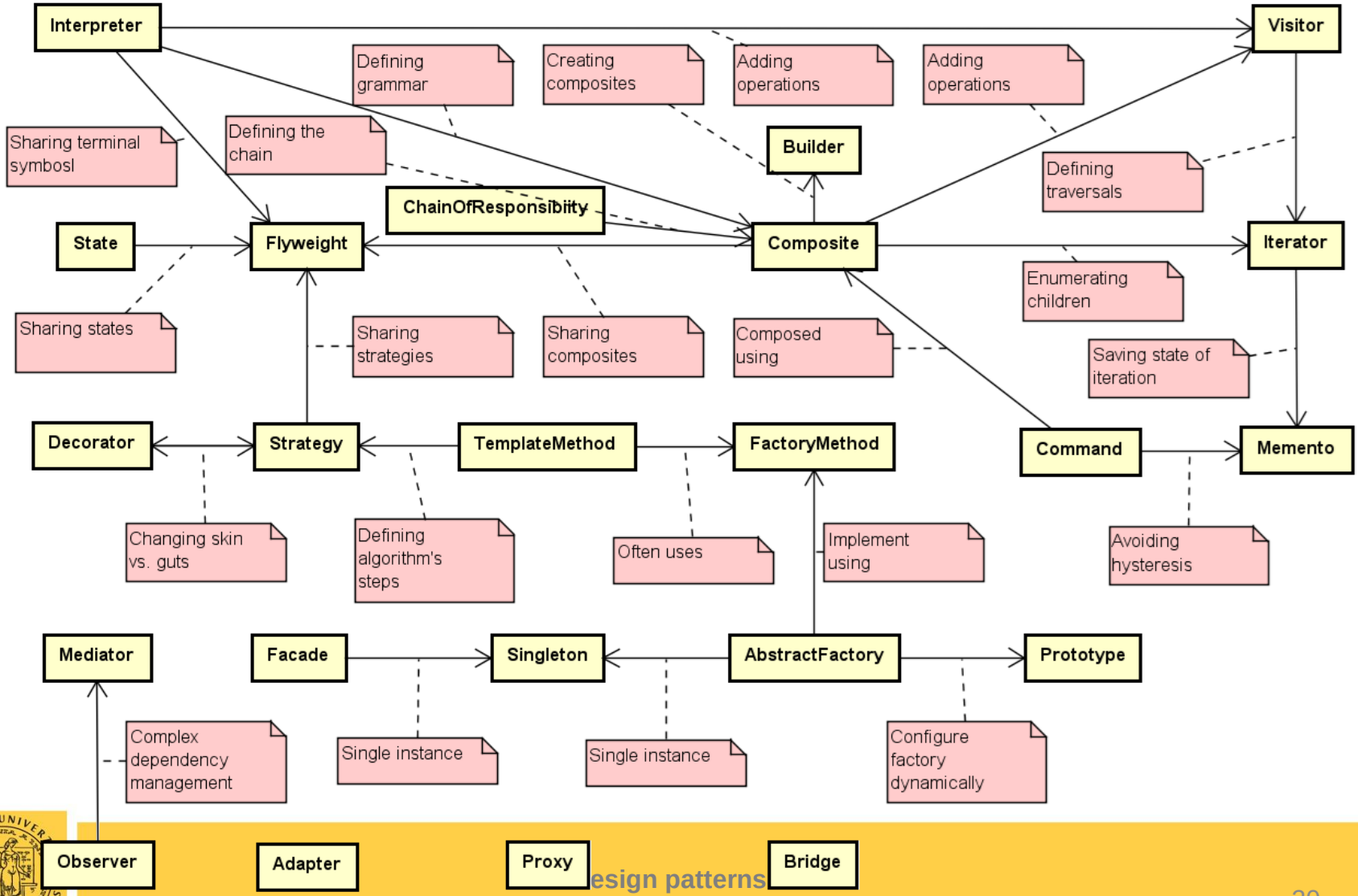
GoF describing design patterns

- **Collaborations** - How the participants collaborate to carry out their responsibilities.
- **Consequences** - (see slide above).
- **Implementation** - What pitfalls, hints, or techniques should you be aware of when implementing the pattern?
- **Sample Code** – C++, Smalltalk.
- **Known Uses** - Examples of the pattern found in real systems.
- **Related Patterns** - What design patterns are closely related to this one?

Design patterns proposed by GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory method	Adapter	Interpreter Template method
	Instance	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design patterns relationships (GoF)



Common causes of redesign 1

- **Creating an object by specifying a class explicitly**
  - `Car c = new Car()`
  - **It is not** a programming agains interface – if there will be a need in the future to create `Car` in a specific way a lot of code will need modifications.
  - *Abstract factory, Factory method, Prototype*.

Common causes of redesign 2

- **Dependence on specific operations**
  - When you specify a particular operation, you commit to one way of satisfying a request.
  - By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.
  - *Chain of responsibility*, *Command*.

Common causes of redesign 3

- **Dependence on hardware and software platform**
  - Software that depends on a particular platform will be harder to port to other platforms.
  - It may even be difficult to keep it up to date on its native platform.
  - *Abstract factory*, *Bridge*

Common causes of redesign 4

- **Dependence on object representations or implementations**
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes.
  - `FMSI.DI.varga.doSomething();` what if I change my department?
  - *Abstract factory*, *Bridge*, *Memento*, *Proxy*.

Common causes of redesign 5

- **Algorithmic dependencies**
  - Changes of an algorithm (extension, optimization, replace..) are very often during development.
  - If object relies on the specific form of algorithm, it will have to be modified in such a case as well.
  - *Builder*, *Iterator*, *Strategy*, *Template method*, *Visitor*.

Common causes of redesign 6

- **Tight coupling**
  - Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other
  - Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes.
  - *Abstract factory*, *Bridge*, *Chain of responsibility*, *Command*, *Facade*, *Mediator*, *Observer*.

Common causes of redesign 7

- **Extending functionality by subclassing**
  - For a proper functionality of descendant, ascendant‘s implementation must be known.
  - Overriding specific method may have sense only with overriding some other method.
  - Combinatorial explosion of classes.
  - Composition is often better solution, however overuse will make the code harder to understand.
  - *Bridge*, *Chain of responsibility*, *Composite*, *Decorator*, *Observer*, *Strategy*.

Common causes of redesign 8

- **Inability to alter classes conveniently**
  - Source codes are not available.
  - Modification triggers a lot of changes.
  - *Adapter*, *Decorator*, *Visitor*.

How to select a design pattern (GoF)

- Consider **how** design **patterns solve** design **problems.**
- Scan **intent** of the pattern.
- Study how patterns **interrelate**.
- Study patterns of like **purpose**.
- Examine a **cause of redesign**.
- Consider **what should be variable** in your design.

- **Always use common sense!**

Variable aspects of „creational“ patterns

Design pattern	Aspect(s) that can vary
Abstract factory	families of product objects
Builder	how a composite object gets created
Factory method	subclass of object that is instantiated
Prototype	class of object that is instantiated
Singleton	the sole instance of a class

Variable aspects of „structural“ patterns

Design pattern	Aspect(s) that can vary
Adapter	interface to an object
Bridge	implementation of an object
Composite	structure and composition of an object
Decorator	responsibilities of an object without subclassing
Facade	interface to a subsystem
Flyweight	storage costs of objects
Proxy	how an object is accessed; its location

Variable aspects of „behavioral“ patterns

Design pattern	Aspect(s) that can vary
Chain of responsibility	object that can fulfill a request
Command	when and how a request is fulfilled
Interpreter	grammar and interpretation of a language
Iterator	how an aggregate's elements are accessed, traversed
Mediator	how and which objects interact with each other
Memento	what private information is stored outside an object, and when
Observer	number of objects that depend on another object; how the dependent objects stay up to date
State	states of an object
Strategy	an algorithm
Template method	steps of an algorithm
Visitor	operations that can be applied to object(s) without changing their class(es)

„Another“ design patterns

- There exist not only the patterns presented by GoF – however some object puritans consider only those patterns to be the true patterns.
- Some of the other design patterns are:

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Dependency injection Simple factory method	Twin	
	Instance	Multiton Object pool RAII	Delegation Marker	Null object

Literature

[1] ALEXANDER, Christopher, et al.: *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure)*. 1977.

[2] COPLIEN, J.O.: *Advanced C++ Programming Styles and Idioms.* Reading, MA: Addison-Wesley, 1992.

[3] GAMMA, E., HELM R., JOHNSON, R., VLISSIDES , J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[4] FOWLER, Martin.: *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.

[5] BUHNOVÁ, Barbora. *Analytické vzory*. [online] s.d. [cit. september. 2016]- Dostupné z http://www.fi.muni.cz/~buhnova/PV167/Analyticke-vzory.pdf

[6] YACOUB, S., AMMAR, H.: *Pattern-Oriented Analysis and Design – Composing Patterns to Design Software Systems*. USA, Boston: Addison-Wesley, 2003.

Warning

- These materials can be used exclusively by students of the course Design patterns on Faculty of Management Science and Informatics, University of Žilina.
- Duplicating, publishing (even some parts) of materials without written permision of the author is not allowed.

Ing. Michal Varga, PhD.
Department of Informatics
Faculty of Management Science and Informatics
University of Žilina
Michal.Varga@fri.uniza.sk