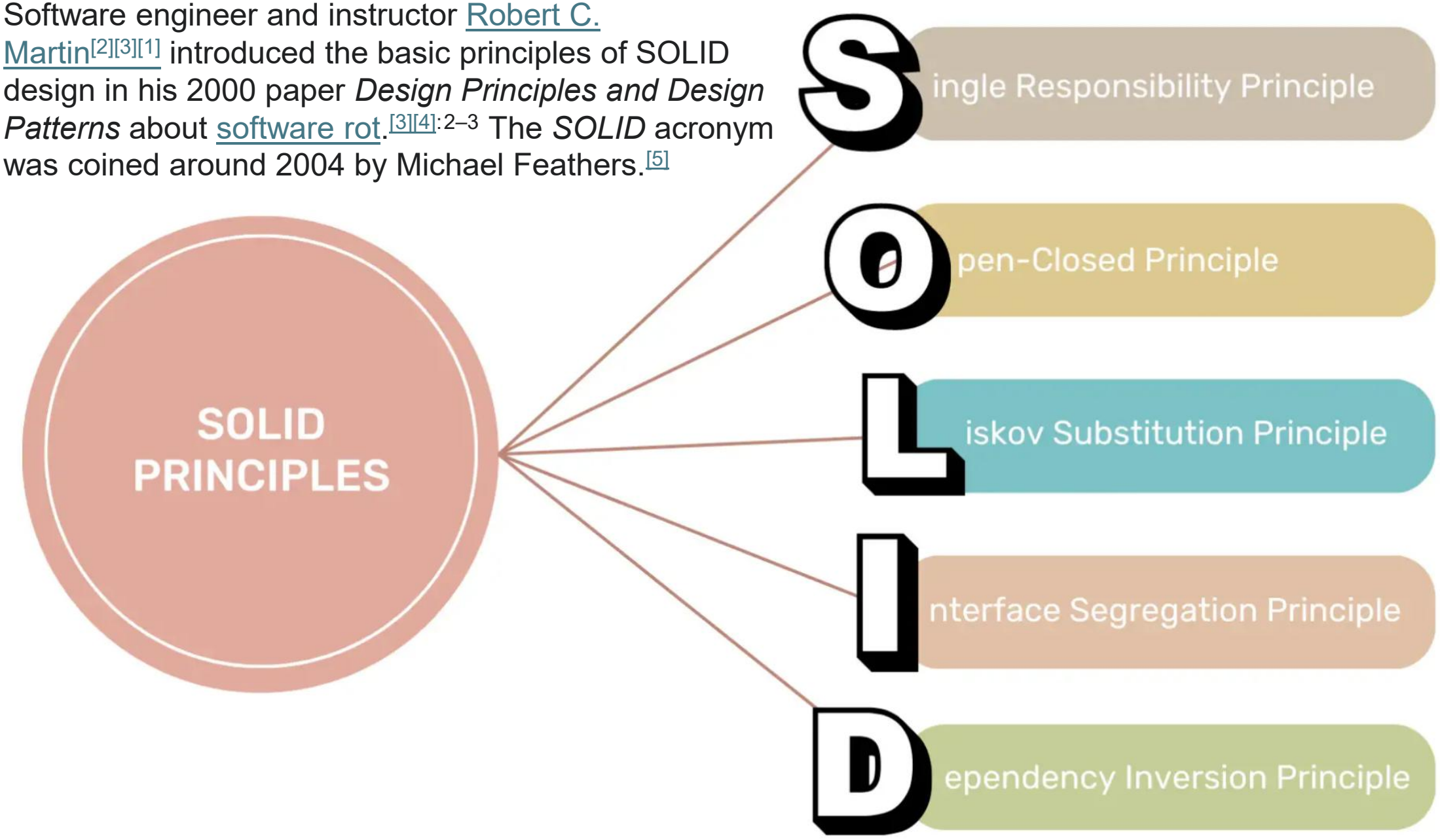
Software engineer and instructor Robert C. Martin[2][3][1] introduced the basic principles of SOLID design in his 2000 paper *Design Principles and Design Patterns* about software rot.[3][4]:2–3 The *SOLID* acronym was coined around 2004 by Michael Feathers.[5]



Liskov Substitution principle
LSP

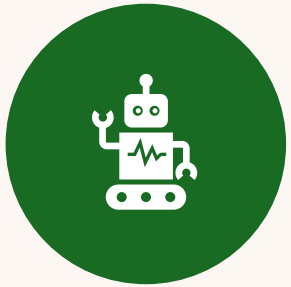
Agenda

01	Historia
02	Przykłady
03	Definicja
04	Drama/Foch
05	Q/A dyskusja!!!

- Barbara Liskov (ur. 7 listopada 1939 r.) jest amerykańskim naukowcem zajmującym się informatyką i jedną z najbardziej wpływowych postaci w dziedzinie podstaw współczesnej inżynierii oprogramowania. Jest najbardziej znana ze swojej pracy nad abstrakcją danych, systemami rozproszonymi i projektowaniem języków programowania. Liskov uzyskała doktorat z informatyki na Uniwersytecie Stanforda w 1968 roku, stając się jedną z pierwszych kobiet w Stanach Zjednoczonych, które uzyskały doktorat z informatyki. Większość swojej kariery spędziła w MIT, gdzie kierowała dużymi projektami badawczymi i była mentorką wielu pokoleń informatyków. Nagroda Turinga 2008 r.

1987.

OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications



Coroczna konferencja organizowana przez ACM (Association for Computing Machinery) organizowana od 1986 roku.

Addendum to the Proceedings

- O czym to jest:
  - Abstrakcja danych
  - Specyfikacja
  - Dziedziczenie i hierarchia typów
  - Polimorizm

Data Abstraction and Hierarchy

Barbara Liskov
MIT Laboratory for Computer Science
Cambridge, Ma. 02139

Abstract

Data abstraction is a valuable method for organizing programs to make them easier to modify and maintain. Inheritance allows one implementation of a data abstraction to be related to another hierarchically. This paper investigates the usefulness of hierarchy in program development, and concludes that although data abstraction is the more important idea, hierarchy does extend its usefulness in some situations.

This research was supported by the NEC Professorship of Software Science and Engineering.

Wczesna definicja

3.3. Type Hierarchy

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra. What is wanted here is something like the following substitution property [6]: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T. (See also [2, 17] for other work in this area.)

We are using the words "subtype" and "supertype" here to emphasize that now we are talking about a semantic distinction. By contrast, "subclass" and "superclass" are simply linguistic concepts in programming languages that allow programs to be built in a particular way. They can be used to implement subtypes, but also, as mentioned above, in other ways.

The above examples ignored a simple difference between the pairs of types, namely related operations. A subtype must have all the operations[3] of its supertype since otherwise the using program could not use an operation it depends on. However, simply having operations of the right names and signatures is not enough. (A operation's *signature* defines the numbers and types of its input and output arguments.) The operations must also do the same things. For example, stacks and queues might have operations of the same names, e.g., *add_el* to push or enqueue and *rem_el* to pop or dequeue, but they still are not subtypes of one another because the meanings of the operations are different for them.

- Przykłady

1994 r.

Later on Jeannette Wing, who actually had been my master's student I think and then John Guttag's PhD student, approached me and said, "Why don't we try to figure out precisely what this means?" So we worked together on this in some papers that got published a bit later.

A Behavioral Notion of Subtyping

BARBARA H. LISKOV
MIT Laboratory for Computer Science
and
JEANNETTE M. WING
Carnegie Mellon University

A Behavioral Notion of Subtyping

Invariants

Abstract Function

Pre-conditions

Constraints

Post-conditions

History properties

Behavioral Subtyping Using Invariants and Constraints

Barbara H. Liskov[a] Jeannette M. Wing

July 1999
CMU-CS-99-156

1999 r.

We present a way of defining the subtype relation that ensures that subtype objects preserve behavioral properties of their supertypes. The subtype relation is based on the specifications of the sub- and supertypes. Our approach handles mutable types and allows subtypes to have more methods than their supertypes.

x: T := E

is legal provided the type of expression E is a subtype of the declared type T of variable x. Once the assignment has occurred, x will be used according to its "apparent" type T, with the expectation that if the program performs correctly when the actual type of x's object is T, it will also work correctly if the actual type of the object denoted by x is a subtype of T.

3 Model of Computation

We assume a set of all potentially existing objects, *Obj*, partitioned into disjoint typed sets. Each object has a unique identity. A *type* defines a set of *values* for an object and a set of *methods* that provide the only means to manipulate that object. Effectively *Obj* is a set of unique identifiers for all objects that can contain values.

by relying on the helping function, *mk_elems*, that maps sequences to multisets in the obvious manner. (We will revisit this abstraction function in Section 5.3.) The **subtype** clause also lets specifiers relate subtype methods to those of the supertype. The subtype must provide all methods of its supertype; we refer to these as the *inherited* methods.[3] Inherited methods can be renamed, e.g., *push* for *put*; all other methods of the

[3] We do not mean that the subtype inherits the code of these methods but simply that it provides methods with the same behavior (as defined below) as the corresponding supertype methods.

DEFINITION OF THE SUBTYPE RELATION, $\preceq$: $\sigma = \langle O_\sigma, S, M \rangle$ is a *subtype* of $\tau = \langle O_\tau, T, N \rangle$ if there exists an abstraction function, $A : S \to T$, and a renaming map, $R : M \to N$, such that:

1. Subtype methods preserve the supertype methods' behavior. If m_τ of τ is the corresponding renamed method m_σ of σ, the following rules must hold:

   - *Signature rule.*
     - *Contravariance of arguments.* m_τ and m_σ have the same number of arguments. If the list of argument types of m_τ is α_i and that of m_σ is β_i, then $\forall i \ . \ \alpha_i \preceq \beta_i$.
     - *Covariance of result.* Either both m_τ and m_σ have a result or neither has. If there is a result, let m_τ's result type be α and m_σ's be β. Then $\beta \preceq \alpha$.
     - *Exception rule.* The exceptions signaled by m_σ are contained in the set of exceptions signaled by m_τ.
   - *Methods rule.* For all $x : \sigma$:
     - *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$

2. Subtypes preserve supertype properties. For all computations, c, and all states ρ and ψ in c such that ρ precedes ψ, for all $x : \sigma$:

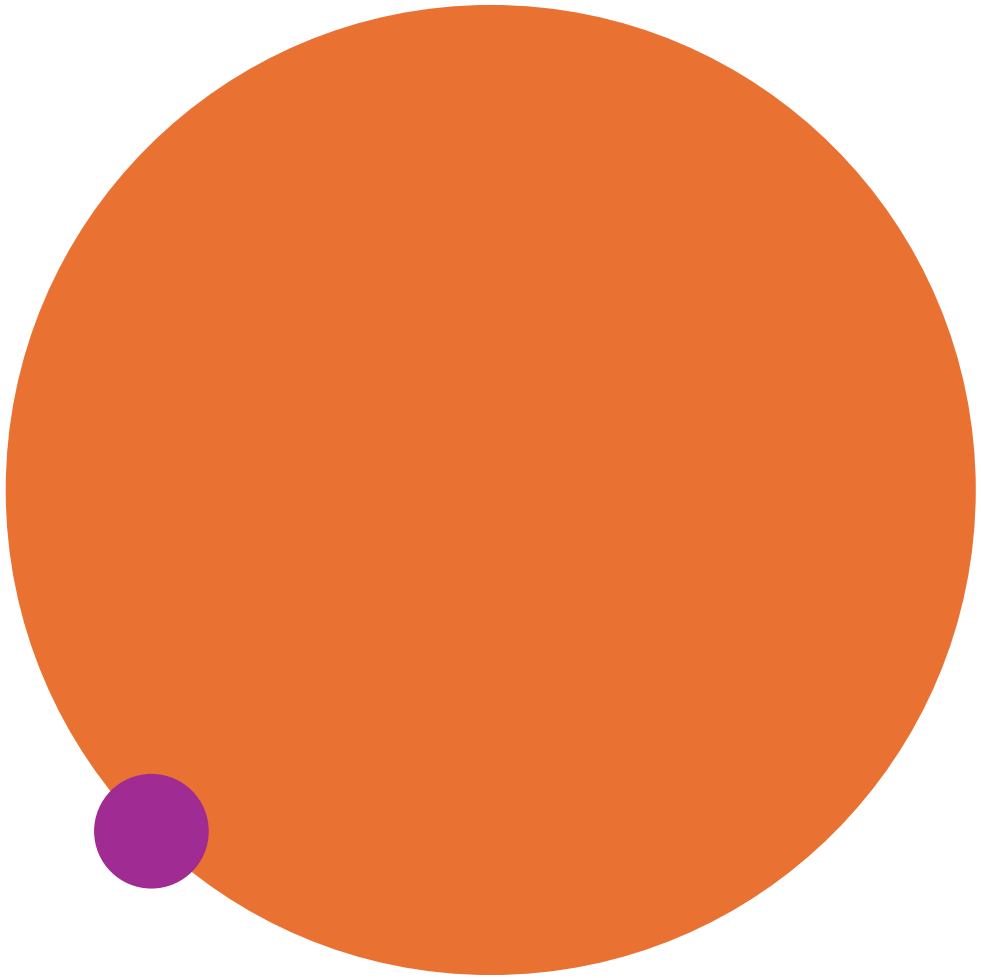
   - *Invariant Rule.* Subtype invariants ensure supertype invariants.

     $I_\sigma \Rightarrow I_\tau[A(x_\rho)/x_\rho]$
   - *Constraint Rule.* Subtype constraints ensure supertype constraints.

     $C_\sigma \Rightarrow C_\tau[A(x_\rho)/x_\rho, A(x_\psi)/x_\psi]$

Figure 4: Definition of the Subtype Relation

- Przykłady



- Drama

Programming concepts: substitution principle

Far from me any wish to under-represent the seminal contributions of Barbara Liskov, particularly her invention of the concept of abstract data type on which so much relies. As far as I can tell, however, what has come to be known as the "Liskov Substitution Principle" is essentially contained in the discussion of polymorphism in section 10.1 of in the first edition (Prentice Hall, 1988) of my book *Object-Oriented Software Construction* (hereafter OOSC1); for example, *"the type compatibility rule implies that the dynamic type is always a descendant of the static type"* (10.1.7) and *"if B inherits from A, the set of objects that can be associated at run time with an entity* [generalization of variable] *includes instances of B and its descendants"*.

Perhaps most tellingly, a key aspect of the substitution principle, as listed for example in the Wikipedia entry, is the rule on assertions: in a proper descendant, keep the invariant, keep or weaken the precondition, keep or strengthen the postcondition. This rule was introduced in OOSC1, over several pages in section 11.1. There is also an extensive discussion in the article *Eiffel: Applying the Principles of Object-Oriented Design* published in the Journal of Systems and Software, May 1986.

The original 1988 Liskov article cited (for example) in the Wikipedia entry on the substitution principle says nothing about this and does not in fact include any of the terms "assertion", "precondition", "postcondition" or "invariant". To me this absence means that the article misses a key property of substitution: that the *abstract* semantics remain the same. (Also cited is a 1994 Liskov article in TOPLAS, but that was many years after OOSC1 and other articles explaining substitution and the assertion rules.)

Liskov's original paper states that *"if for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for oz, then S is a subtype of T."* As stated, this property is impossible to satisfy: if the behavior is identical, then the implementations are the same, and the two types are identical (or differ only by name). Of course the **concrete** behaviors are different: applying the operation *rotate* to two different figures o1 and o2, whose types are subtypes of *FIGURE* and in some cases of each other, will trigger different algorithms — different behaviors. Only with assertions (contracts) does the substitution idea make sense: the **abstract** behavior, as characterized by preconditions, postconditions and the class invariants, is the same (modulo respective weakening and strengthening to preserve the flexibility of the different version). Realizing this was a major step in understanding inheritance and typing.

I do not know of any earlier (or contemporary) exposition of this principle and it would be normal to get the appropriate recognition.

[Some contributions - Bertrand Meyer's technology+ blog]

1. Język Eiffel (1985) - obiektowy język programowania z wbudowaną obsługą kontraktów (`require`, `ensure`, `invariant`)
2. Design by Contract - metodologia definiowania formalnych "kontraktów" dla komponentów:
•Warunki wstępne (preconditions)
•Warunki końcowe (postconditions)
•Niezmienniki klas (invariants)
3. "Object-Oriented Software Construction" (1988) - klasyczna książka o programowaniu obiektowym (~1400 stron)
4. Zasada Open-Closed - "Oprogramowanie powinno być otwarte na rozszerzenia, zamknięte na modyfikacje" (później "O" w SOLID)
5. Command-Query Separation - metody powinny albo zmieniać stan, albo zwracać dane, nigdy jedno i drugie

Wieloletni profesor ETH Zürich, założyciel Eiffel Software, laureat nagrody Dahl-Nygaard (2006). Wciąż aktywny - prowadzi blog bertrandmeyer.com.

constraints not just on the signatures of an object's methods, but also on their behavior.

Our work is most similar to that of America [Ame91], who has proposed rules for determining based on type specifications whether one type is a subtype of another. Meyer [Mey88] also uses pre- and post-condition rules similar to America's and ours. Cusack's [Cus91] approach of relating type specifications

The End