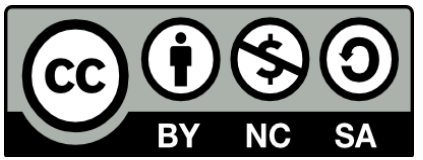
SW subsystem decomposed

On how-to manage SW subsystem with components

Samo Pogačnik

Škofja Loka, 6.12.2011



About

About the author and this work:

- I am not a computer science engineer, but I do have some computer and programming knowledge through my electro-technical education and about 20 years of private and on-job experience in SW development.
- This work is an attempt to align (merge) my personal experience and views on SW development, with some scientific knowledge on SW engineering.
- As said, this presentation's initial purpose is to clear-up my own mind, however comments and suggestions would have been very much appreciated.
- And of course, it would have been great, if someone finds this work useful.

Agenda

I. Abstract

II. Components

III. Provided interface

IV. Required interface

V. Composing components

VI. Compatible components

VII. Component identity

VIII. Name attributes (todo...)

IX. Team responsibilities (todo...)

X. Team planning (todo...)

XI. References

I. Abstract

- **Goals of this presentation:**
  - Tries to put Component-Based Development (CBD) paradigm into perspective of:
    - Individual component management from its in-system replacement capability point of view
    - Target SW subsystem build environment (component framework)
    - Defined development / integration team responsibilities
  - Provides implementation agnostic suggestion for component meta-information, which enables the above perspectives
  - Does not provide magic SW decomposition rules based on specific SW subsystem requirements

II. Components (1/3)

- **Component Based Development (CBD):**
  - Is a SW development paradigm focusing design on a "prefabricated" reusable and replaceable modules (components) through a well defined architecture of a SW system [1].
  - Components by definition must exist as explicit run-time (I'd rather say deployment-time*) incarnations (i.e. runnable code like binaries or scripts, configuration data, … - basically anything that can be defined as a replacement part of a specific runnable system). This definition differentiates components from design entities like classes in OOP [2]. However it is component's build-time (source) incarnation a subject of:
    - modification during its life cycle (component development and maintenance)
    - system composition / integration (a group of components can be defined as a higher level component and the whole system is a top-level component as well).
  - A method of component integration and replacement is specific for the target system build and deployment environment.

*By using term deployment-time incarnations, runtime entities like processes and threads are being excluded. Also the term deployment-time, just like the term runtime presumes, that a component is a pre-built unit. However there may be multiple levels of composition with specific deployment (non runtime) environments (any environment which allows its components to be replaced).

II. Components (2/3)

- **Component properties [3]:**

  1. *Black-box composability, substitutability and reusability*: there is no need to know the design and the implementation when composing a component with other parts of the system, substituting a component with another one or reusing it in another application.
  2. *Independent development*: components can be designed, implemented, verified, validated and deployed independently.
  3. *Interoperability*: components can be implemented in different programming languages and paradigms, but they can be composed, be glued together and cooperate with each another.

- **Those properties require that each component define black-box specification of:**

  - what it provides and
  - what a component requires from its run-time and build-time environment

II. Components (3/3)

- **Components and interfaces [4]:**
  - Interface is a description of component's visible behaviours.
  - Components communicate through their provided and required interfaces.
  - **Provided interface** specifies component's own visible behaviours, realized through component implementation.
  - **Required interfaces** of a component specify usage of other components via their provided interfaces. They help in implementation of component's provided interface.
- **Components and their interoperability:**
  - Interoperability is achieved via compatibility between connected required and provided interfaces [4].
  - The key is implementation conformance to interface specification at:
    - Syntax level
    - Semantic level
    - Behavioural level (protocol: sequences, timings, synchronism, constraints, …)
  - Connected provided and required interfaces adhere to the same interface specification.

III. Provided interface (1/2)

- **Provided interface is the main reason for each component's existence**
  - If there is nobody (other component or a different kind of user / actor) to require component's capabilities (its provided interface), then this component is mostly not needed.
  - User interface is a special (top-level) case of provided interface.
- **One or more provided interfaces per component:**
  - We could say, that a component exposes more provided interfaces and it is true (i.e. a compound component of more components exposing some provided interfaces of its internal components). Such perception may be also given through component composition drawings, emphasizing different exposed functionalities of a component, ...!
  - But, from our perspective of each component's replacement capability (in its own composition level - deployment environment), a union of all of its visible behaviours is a subject to change in a single component replacement transaction.
  - Thus we'd like to say, that **a component exposes all of its visible behaviours through a single commonly identified provided interface**.
    - I.e. a compound component as a replacement unit within a SW subsystem exposes its different capabilities (i.e. of some of its internal component's provided interfaces) as a single identification.
    - We may still talk about one component's provided interface as there were more of them, when its differences need to be emphasized.

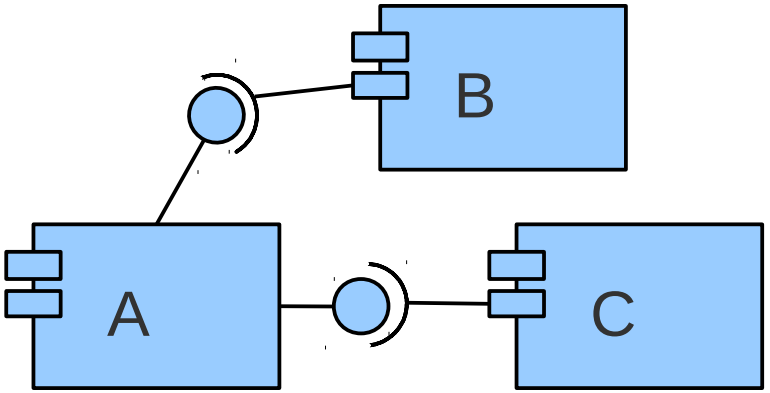
III. Provided interface (2/2)

➔ **More on provided interfaces:**

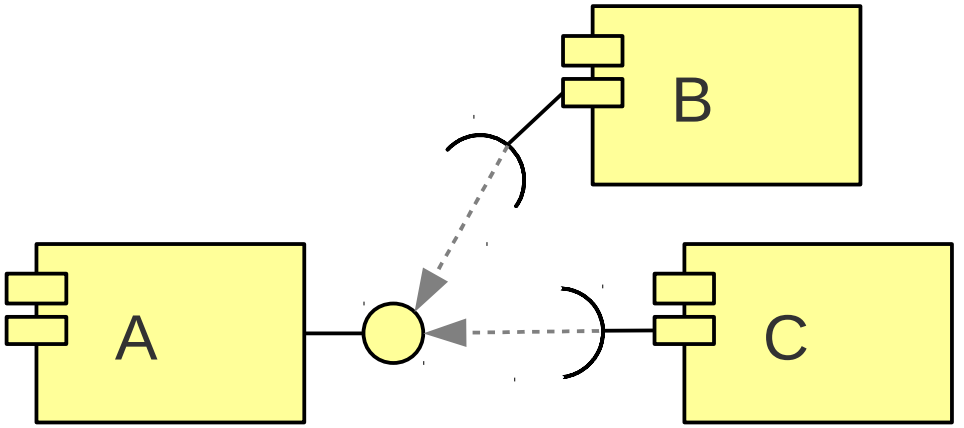
- Both drawings below may be equivalent, but:

  a) Emphasizes **who is using what** in another component (a low-level design oriented aspect – requires a lot more details)

  b) Focuses on just **who is using who** – which component uses which component (an architecture oriented perspective, which is of our interest)



a) Who is using what



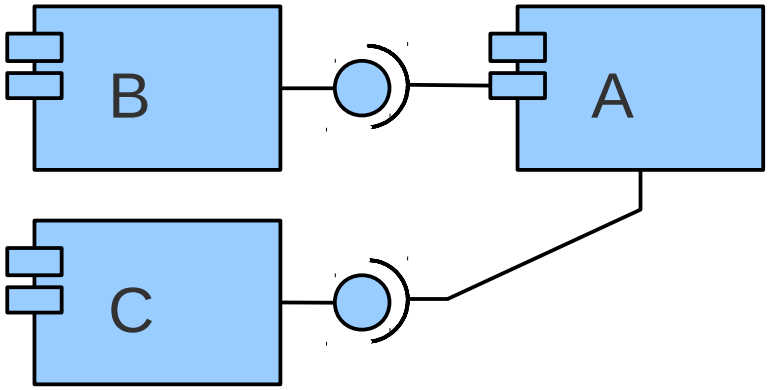
b) Who is using who

IV. Required interface (1/2)

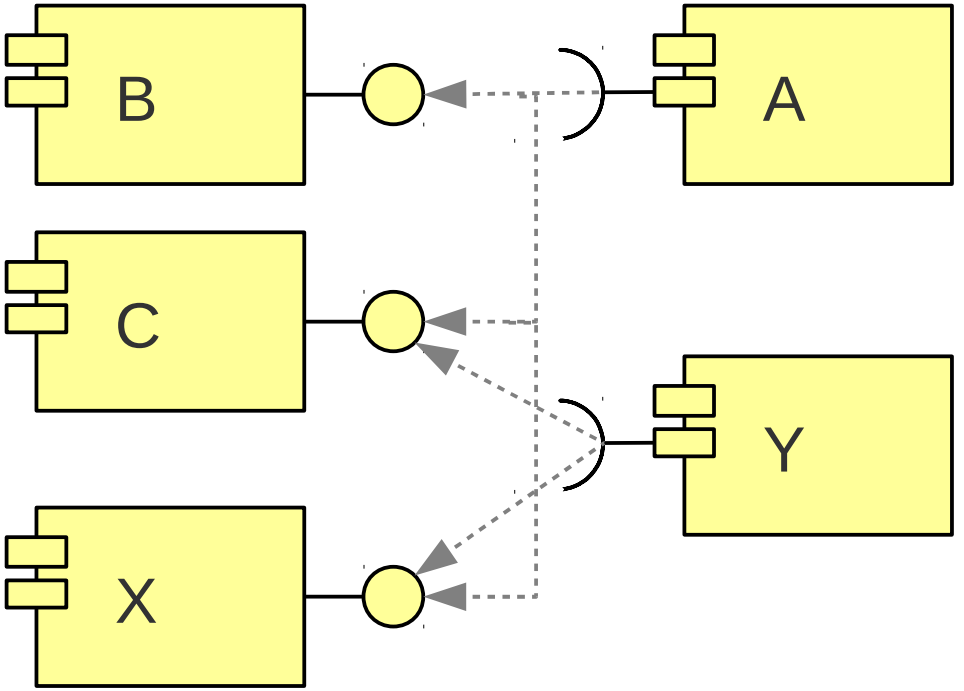
- **A required interface of a component defines usage of another component through its provided interface.**
  - There is probably just one component in each SW subsystem without a required interface (independent of any other SW component within the system).
- **One or more required interfaces per component:**
  - Typically a component defines one required interface per each used component.
  - What is of our interest:
    - **Again we focus on who is using who** and not which part of a provided interface of another component has been used by our component ....
    - Instead of a common identification of a union of required interfaces, **a reference list to all used provided interfaces** would've been suggested to represent a common declaration of all component's required interfaces.
    - We need to know, if existing running (and build) environment of a component is sufficient for the new replacement component or not.

IV. Required interface (2/2)

- **More on required interface**
  - Regarding of how many components get used by a component, different drawing techniques may be used:



a) Required interface drawn per each used component.



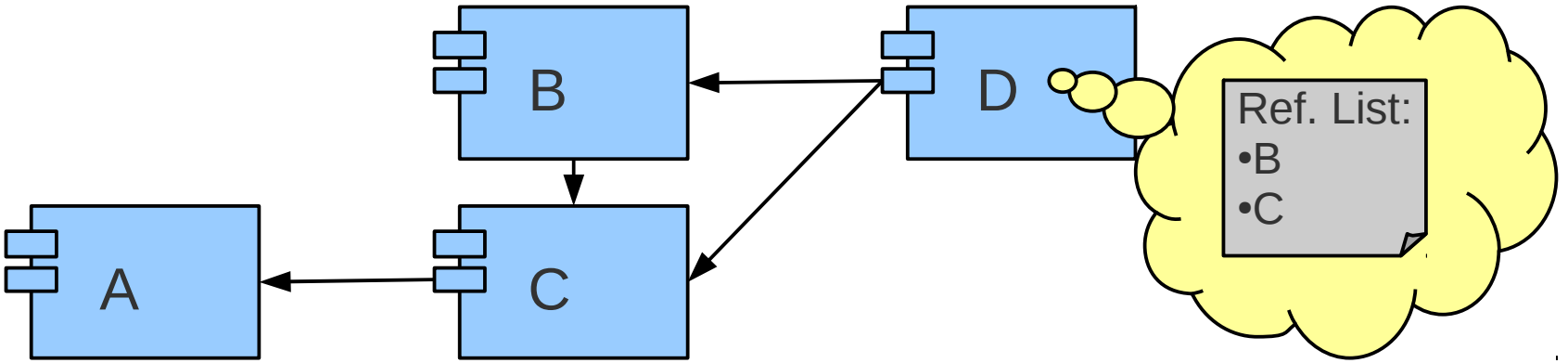
b) A single required interface drawn

V. Composing components (1/6)

- **Before we describe types of component composition, let's try:**
  - to limit our interest in a composition and frame our abstraction layer around relevant information:
    - **In short we'd say, we only care who is using who, as long as they are compatible.**
    - By that we mean which component uses which other component through any of their interfaces. For that purposes we simplify drawings and connect components with arrows only (no interfaces drawn)
    - That said, we don't care (at least not at this level) what's the purpose of composition nor how we end-up doing a SW subsystem out of components, …
    - However we do care about all compatibility information, that can be provided by each incarnation of a component (will be discussed later).
- **There are two generic types of composition:**
  - *Chaining* components together (programming related techniques)
  - *Grouping* (chained) components into a new *component* (integration related techniques)

V. Composing components (2/6)

- Chaining components together:
  - It is a natural way to compose components by chaining the provided operations of one component to the required operation of the other[3].
  - This is the basic level of composition based on which other components are being explicitly used by each component.
    - Chaining components together is the integral part of each component development requiring other components for its operation.
  - A kind of **reference list** must exist for each component specifying used components (provided interfaces used via own required interfaces).



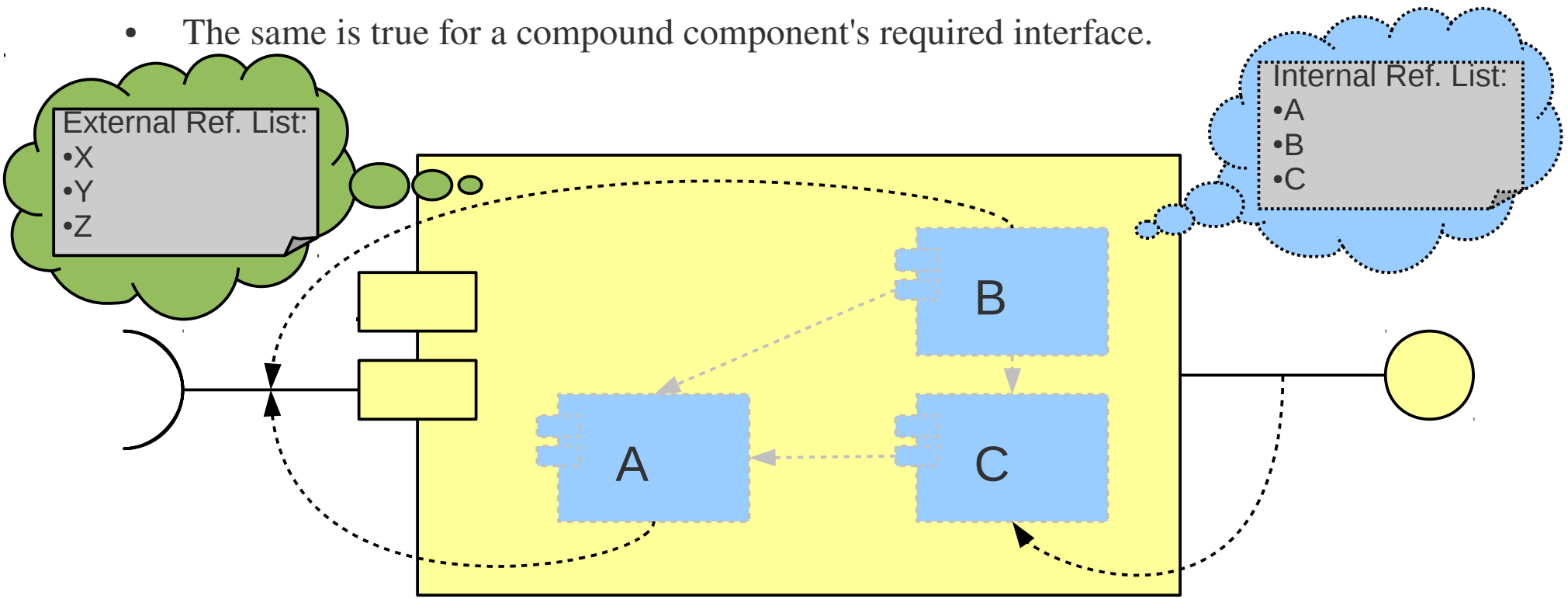
Chained components (automatically through their reference list declaration)

V. Composing components (3/6)

- *Grouping* (chained) components into a new *compound component*:
  - This level of composition is based on architectural and other non direct component usage dependency requirements of a SW subsystem.
  - There are two basic ways of component grouping possible:
    - *Encapsulation* results into a *compound component* which completely hides internal composition of (encapsulated) components (which components and their internal connections).
    - *Collection* results into a *virtual component*, which only refers to all needed (collected) components.
  - A kind of reference lists may exist to specify any encapsulated or collected components.

V. Composing components (4/6)

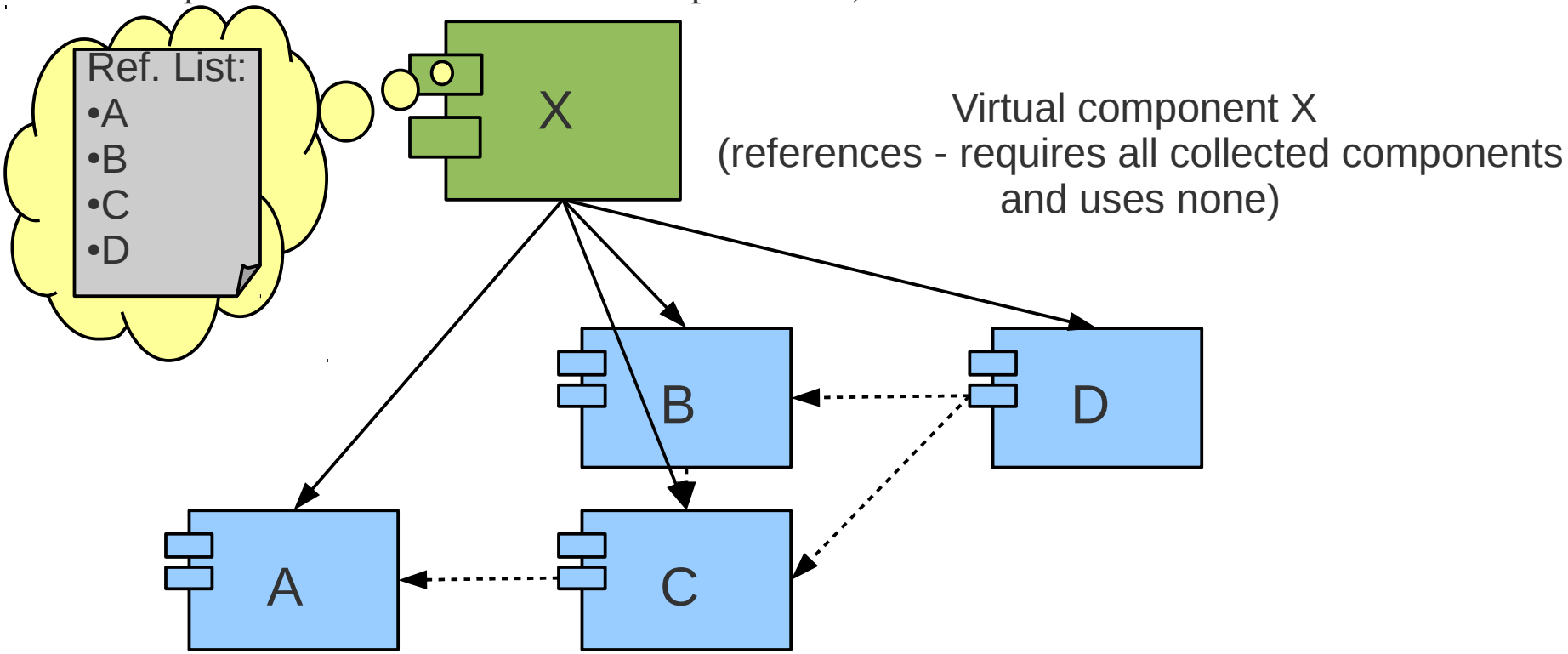
- ***Encapsulation*** (a new ***compound component***):
  - The *compound component* physically encapsulates at least one component.
  - Such a compound component defines its own provided interface:
    - as a subset of a union of all provided interfaces of contained components,
    - and / or through additional adaptation code
  - The same is true for a compound component's required interface.



Compound component (internal structure invisible to the outside world)

V. Composing components (5/6)

- ***Collection*** (a new ***virtual component***):
  - *Collection* results into a *virtual component*, which only refers to all needed (collected) components.
    - The virtual (also called empty or dummy) component collects other components in the same way as chaining defines usage of other components (it does not physically contain them).
    - However a virtual component does not really use referred components (does not implement own required interfaces but does define dependencies).



V. Composing components (6/6)

- **There are two generic types of composition, however:**
  - To compose final SW subsystem several steps of component composition might be exercised and each step might define its own set of chaining and grouping of components.
  - It is also possible for a component to encapsulate some components as well as to collect and use some other components.
    - Such a component uses all composing techniques at once:
      - Implements own behaviour through encapsulation of some other components, hiding their internal relations and via optional adaptation code
      - Requires some components present in the system, used by its own implementation and encapsulated components
      - Requires some components to be present in the system (collected) just because this component is present in the system

VI. Compatible components (1/4)

- **Backward compatibility**
  - Only new **backward compatible component may individually replace** current component within a given version of SW subsystem (which may be a component itself).
  - A **backward compatible component** implements a backward compatible provided interface and backward compatible required interfaces.
    - A **backward compatible provided interface** implementation may extend its capabilities, as long as they are being implemented as a superset of capabilities provided by previous version of component's provided interface.
    - Implementation of component's **backward compatible required interfaces** may only reduce the need for other component's provided capabilities:
      - A new component is being able to use less capable provided interface of another component. Such a component becomes more independent. A maximum reduction of a required interface represents its removal (a component in its new implementation is capable of providing its capabilities without previous help of another component).
      - A new component may increase use of another component's provided interface to its full extent as long as other's components additional capabilities weren't required.
    - It seems, that requirement for each backward compatible required interface is right opposite to requirement for a backward compatible provided interface.

VI. Compatible components (2/4)

- **Incompatibility**
  - Any other kind of component modification results in its **backward incompatibility** (inability to be individual replacement part of an existing SW subsystem).
  - **A backward incompatible component** which implements a backward compatible provided interface with new (incompatible) required interface implementations (i.e. uses a different set of other components) **has adapter capability** (could have been called **component alternative**).
  - **Adapter capable components** provide important compatibility roles as:
    - **infrastructure components** defining a clean separation between levels of layered SW architecture hierarchy (middleware)
    - **adhoc adaptation components** connecting two incompatible interfaces

VI. Compatible components (3/4)

- **Providing compatibility information**
  - We are interested in component's backward compatibility capabilities given by each subsequent implementation of a component.
  - There may be two ways of providing component's compatibility information:
    - **Implicit** provision of compatibility information through lists of interface specification identifiers, a component claims to comply to (won't be discussed in this presentation):
      - Indirect link to other dependent components
      - Additional tools required for human inspection
      - Good for standard interface compliance for the unknown target framework
      - Additional complexity via mapping between interface specifications and components
    - **Explicit** provision of compatibility information via component identifiers:
      - Directly links to other required components
      - Visible information without additional tools
      - If conforming to some external standard interface specification, a component identity most likely does not show that
      - Easier set-up of an adhoc component framework

VI. Compatible components (4/4)

➔ Providing compatibility information

- There are two sources of component's (in)compatibility information:
  - ✔ Provided side
    - ✗ Should tell what kind of change has been made to the provided interface (none, compatible or incompatible)
    - ✗ Implicit provision: a list of interface specifications covered by provided interface
    - ✗ Explicit provision: part of component identification – i.e. name and version (covers type of provided interface changes)
  - ✔ Required side
    - ✗ Should answer the question what kind of change has been made to the required interface (none, compatible or incompatible)
    - ✗ Implicit provision: a list of interface specifications covered by required interfaces
    - ✗ Explicit provision: part of component identification (shows compatibility info of the required interface changes)
- We are going to focus on explicit provision of compatibility info through the rest of the presentation and leave implicit provision for independent exploring of its potential.

VII. Component identity (1/5)

- **What about component identity?**
  - Through a component life cycle many incarnations of a component exist.
    - In development
      - Development versions of a component need to be identified by a component framework and by developers.
    - In system (product) integration
      - A small subset of development versions become system integration versions, which need to be identified by a component framework, by developers and by system integrators.
  - At least “integration” versions need to be uniquely identified at the component composition level.
    - A “Component Identifier” is used to identify each version of a component
    - We might invent an UCI (Uniform Component Identifier – likewise URI, URL, URN:)

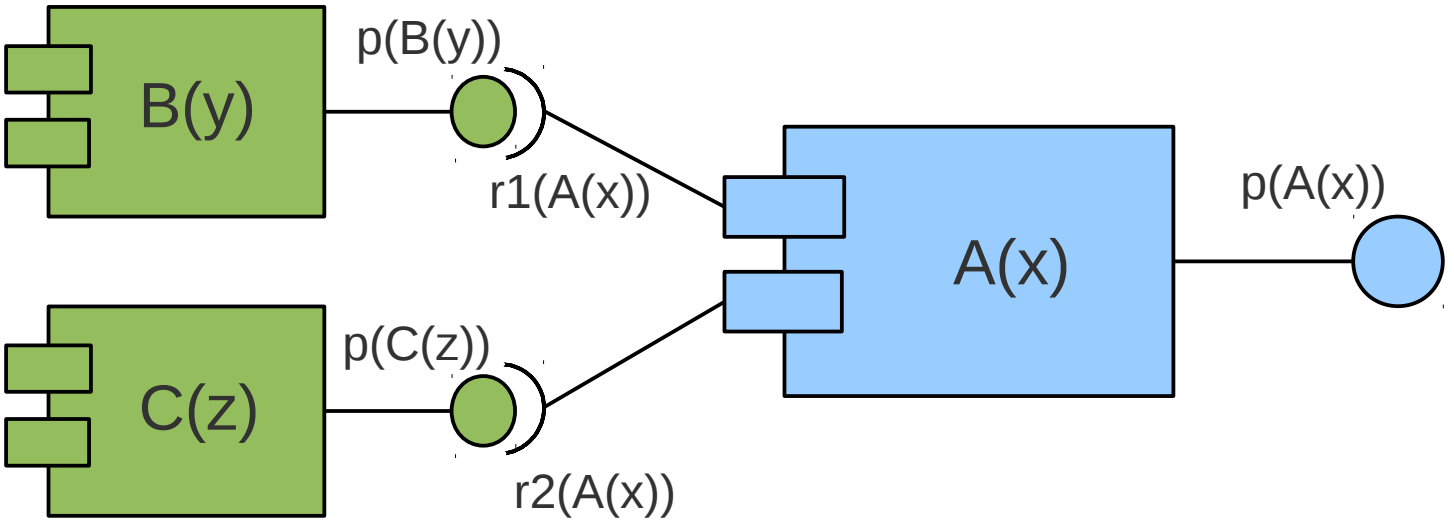
VII. Component identity (2/5)

- **Component identifier requirements:**
  - Unique
  - Human readable
  - Already existing solution
    - Naming (like Linux package naming for instance)
    - Versioning (i.e. APR's Version Numbering [5])
  - Primarily exposing compatibility information
    - We need to know in which way provided interface changed [5]
    - How required interfaces changed
  - A one-to-one relation between component identifier and component sources must be defined

VII. Component identity (3/5)

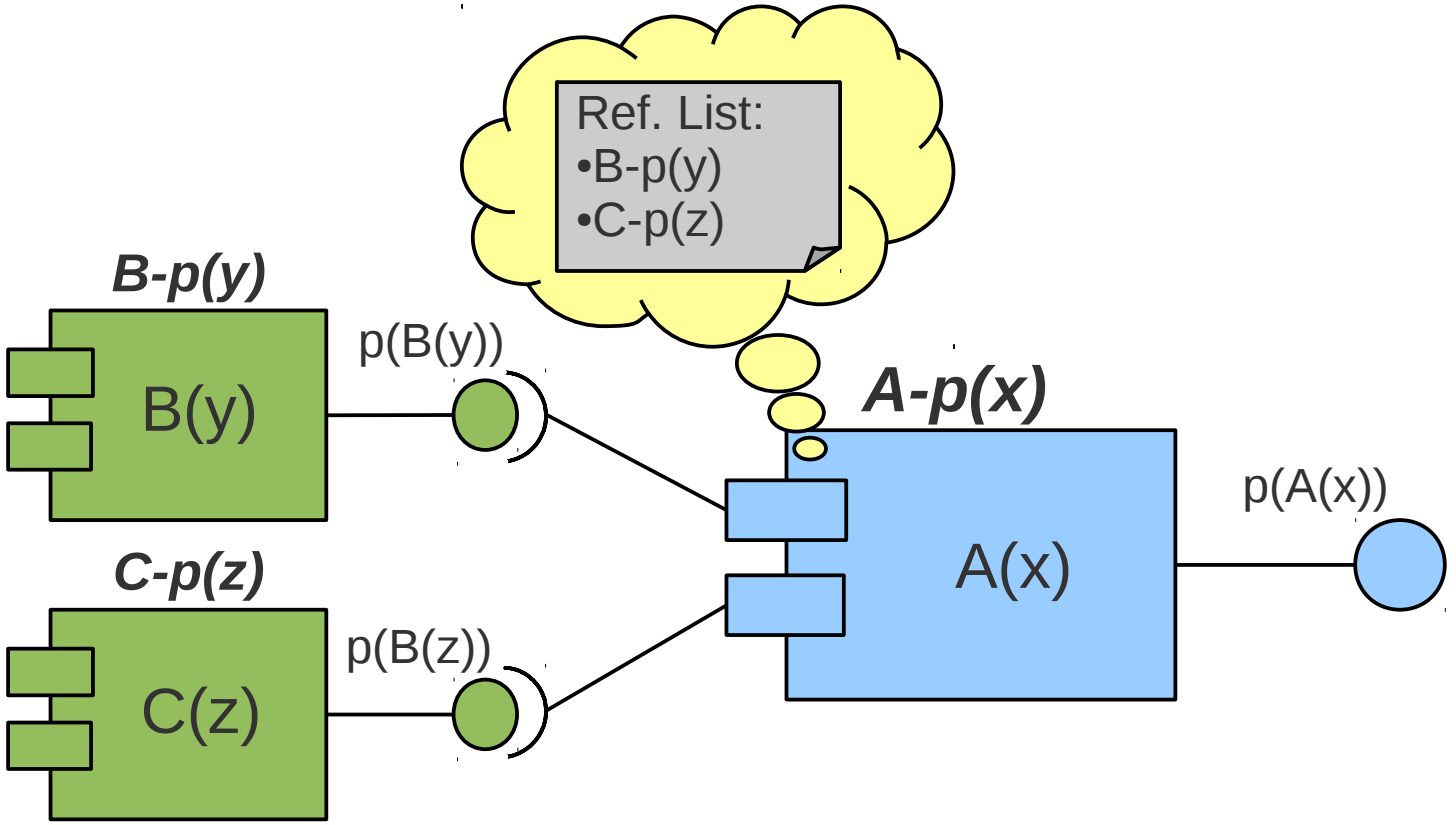
➔ **How to construct human readable component identifier**

- Generally **components** within a given SW subsystem need to be **uniquely named** (i.e. *A*) and each **implementation (revision) uniquely marked** (i.e. sequential count *x*)
- A component ***A(x)*** with provided interface ***p(A(x))*** and required interfaces ***r1(A(x)), r2(A(x))***
- A built component's package name may contain other attributes like package group indicators (i.e. lib, dev, debug, …) or target platform indicator (i.e. ppc, arm, i386, ...), … They aren't of our interest at the moment.



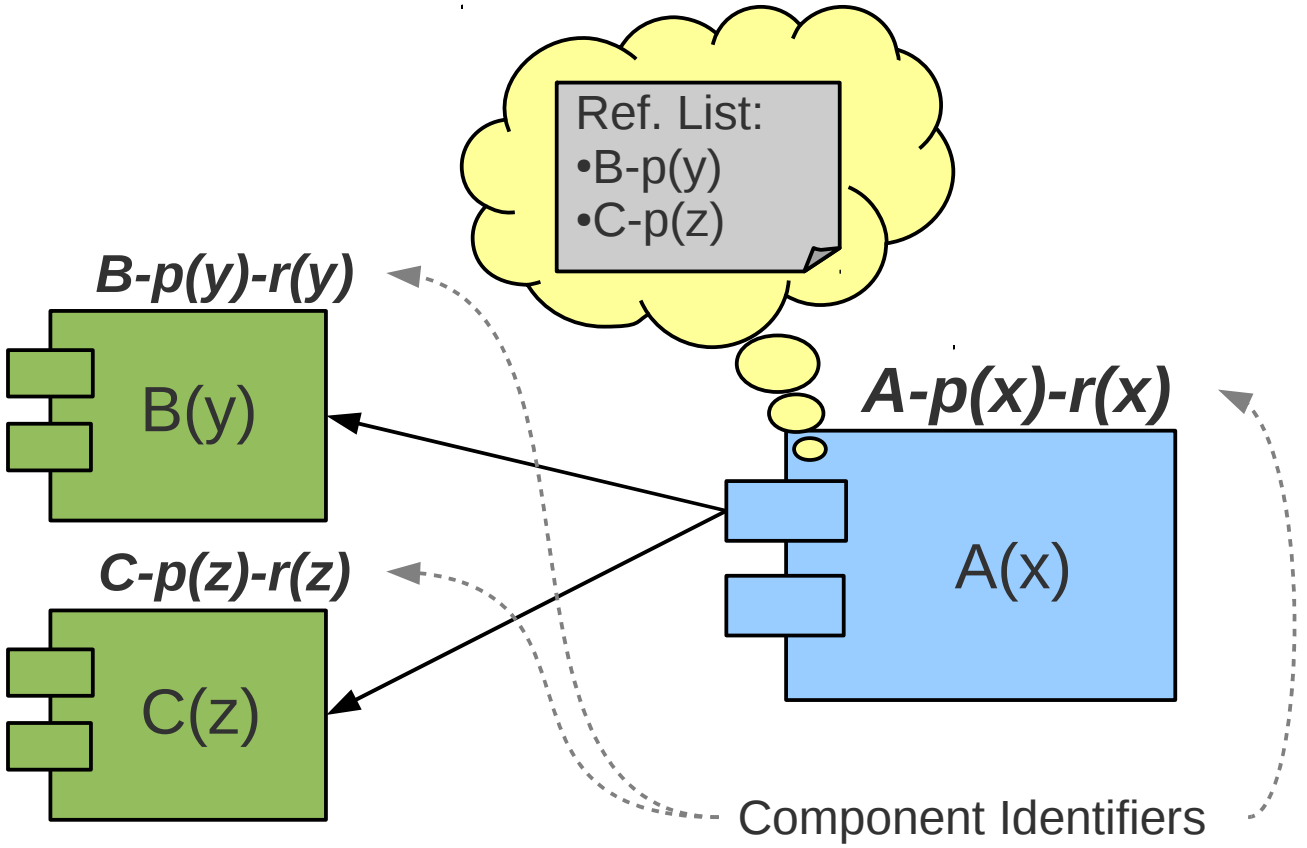
VII. Component identity (4/5)

- **How to construct component identifier**
  - **Identifying a component by its provided interface** (i.e. *p(x)*)allows that a component refers directly to other component versions. References are being defined by required interfaces.



VII. Component identity (5/5)

- **How to construct component identifier**
  - **To expose all compatibility information**, the required side compatibility information (i.e. *r(x)*) has to be added.



VIII. Name attributes (1/x)

- **A component build may produce several results (packages):**
  - Packages may contain build results, packed for different purposes.
  - Different packages may be used in different component or product build and installation phases.
  - Different packages may represent different static build configurations of a component.
  - Packages are being identified via component identifiers, but additional information is needed to distinguish between packages of the same component.

VIII. References (1/1)

[1] Debayan Bose, *Component Based Development CoRR abs/1011.2163, (2010)*

[2] HUIZING M., *Component Based Development, www.xootic.nl/magazine/jan-1999/huizing.pdf*

[3] X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programming. In F. Arbab and M. Sirjani, editors, *International Symposium on Fundamentals of Soft-ware Engineering (FSEN 2007)*, volume 4767 of *Lecture Notes in Computer Science*, pages 191–206. Springer, April 2007. UNU-IIST TR 350.

[4] A. Lanoix, J. Souquieres. *Component-based Development using the B method*, (Research report 2006) LORIA – Nancy-Universite, CNRS

[5] Apache Project, *APR's Version Numbering, http://apr.apache.org/versioning.html*

[6] ...