Components ... What the heck?

Clemens Szyperski
Microsoft Research
Redmond, WA 98052
http://www.research.microsoft.com/~cszypers/

This is a brief transcript of a talk presented at the 4th Workshop on Component-Based Software Engineering (CBSE4) held at ICSE 2001 in Toronto, Canada. No attempt was made to add references; for further reading and proper references, consult the following publications by the author:

- Szyperski C. (1998) *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley.
- Szyperski C. (2000) Modules and components – Rivals or partners. In Böszörményi L., Gutknecht J. and Pomberger G. (eds.) *The School of Niklaus Wirth – The Art of Simplicity*, dpunkt Verlag, Heidelberg and Morgan Kaufmann Publishers, San Francisco.

Why components?

Three tiers of arguments can used to argue for components.

Cost: Make and Buy

The main driver is cost—it can be cheaper to use acquired components instead of developing from scratch. For most organizations it will be important to maintain their own edge on top of acquired baseline products, hence asking for a mix of making and buying. For larger organizations, such 'acquisition' of components may be from an in-house source.

Static: Product Lines

Beyond the somewhat ad-hoc reuse of software fragments, a more disciplined and principled approach can be adopted by aiming for reusable assets in the context of a specific architecture. Typically aiming for products that form a production line controlled by a single organization, Product Line Architecture aims at just this. The composition scenarios are static: members of the product line are assembled ahead of time and then deployed as a whole.

Dynamic: Upgrade, Extension, Integration, Evolution, …

Components can be used as units of change, upgrade, extension (and shrinkage). Such components need to be units of deployment themselves: they need to be sufficiently self-contained, with all dependencies made explicit, that automatic installation in a running system is feasible. This is the main driving criterion that leads to the definition of software components (further below).

The notion of adding to an installed system poses many challenges; adding to a running system is even harder; the hardest is removal from a running system. A

combination of runtime removal and addition is often called hot swapping in analogy to similar operations on hardware systems. It is unlikely that any system can accommodate such forms of advanced dynamic component integration unless based on a principled architecture.

Component Maturity Model

A separate dimension of component integration is the degree of openness towards externally acquired (and thus less controllable) components. As illustrated in Fig. 1, an organization can attempt component use at four levels—that need to be matched by adequate capabilities of that organization, reflecting degrees of maturity.

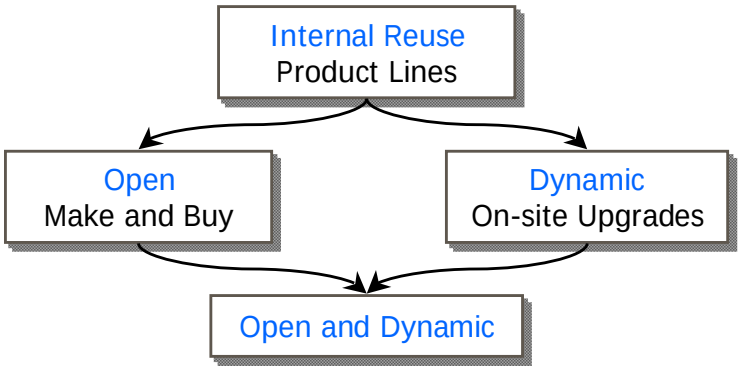

Fig. 1 – The Component Maturity Model

As indicated in the figure, product line approaches are only the beginning—an important one though, since they introduce principled architecture that forms a foundation for the incorporation of components.

Picking the right level of maturity to aim for is a business tradeoff. Required system qualities obviously need to be met: if it is a requirement to support open and dynamic incorporation of components (example: browser that has to accept plug-ins), then there is no escape. Beyond explicit requirements, the business tradeoff involves estimates of need to evolve the solution at customer sites, the need to integrate with other and foreign products, and the expected amortization and time to market effects of building up and applying the chosen component maturity level. Longer-term gains need to be traded against short-term costs, which is often a difficult call for organizations to make.

What's a Component anyway?

Instead of just throwing out a definition of components, the above brief observations help to arrive at necessary properties of *software* components. (There is any number of other concepts, such as source components, specification components, architectural components, etc., all of which are not meant to be distributed and deployed as operational software.)

A software component is a unit of deployment and independent composition. It is a deliverable that is executable by a (virtual) machine. Requiring 'executability' by a machine (an automaton) guarantees that no operator intervention is required. (Not that most other component concepts, including those listed above, do not meet this simple requirement.) Not meeting this requirement leads to components that cannot function at the highest level of component maturity (dynamic and open integration).

To be a unit of delivery and deployment, a component cannot be an object (as objects exist in exactly one copy at a particular locus at anyone point in time), nor can it be a distributed object (which doesn't change the object nature). Software components cannot be concepts such as template since these do not deploy without human interference.

A component could be a single class, but that is unlikely: most classes do not deploy in isolation and do not expose their dependencies sufficiently well. It is more likely that a component will contain many classes.

Technically, a component can thus be seen as a collection of modules and resources. A module is a container for immutable code (perhaps sets of classes, perhaps non-OO constructs as well) and metadata. A resource is a container for immutable data (immutable persisted prototype objects). The metadata typically includes deployment information.

Components / Composites, Contexts, Platforms

A component (leaf or composite) provides services presented via interfaces. There are two kinds of interfaces: traditional 'incoming' or provides interfaces and 'outgoing' or requires interfaces. The latter present the parametric dependencies of a component, since the connection of outgoing interfaces to another component's incoming interfaces enables configuration of requirements against available services.

A component also has static dependencies that are not parametric and thus need to be met as such. Examples are per-instance deployment context dependencies (such as requiring a particular container), dependencies on target component platforms, and assumed architectural embedding (such as assuming use of the component within the context of a particular product line architecture).

Components vs. Modules: Dependencies & Coupling

Decoupling of components by design is hard. Structured, modular, and OO programming are not enough. Here is a probably incomplete list of possible dependencies, from easiest to hardest in terms of tying a component and limiting evolution and flexibility.

A component can take a dependency on an external: -

- *Type* (interface, fully abstract class, etc.)
- *Shared specification model* (coupled type specs; architectural context)
- *Function* (specific implementation but no side effects)
- *Non-sealed class* (polymorphism; same with higher-order function parameters)
- *Abstract data type* (sealed class: monomorphic, no shared state)
- *Non-sealed class with overriding* (implementation inheritance)
- *Abstract data structure* (encapsulated global state – side effects)
- *Global variables* (… pain!)

Component & Architecture

Component-oriented architecture and architecture-aligned components are really flip sides of the same coin! Solutions to many quality problems of composed systems flow from this observation.

This calls for architectural reference models and for architecture-aware component engineering. Neither top-down nor bottom-up processes are adequate here!

Beyond Architectural Styles

Component-oriented reference architectures provide: -

- Concrete technology grounding (it's a von Neumann / Turing machine, alright) – technology neutrality is an illusion;
- Semantic domain grounding (horizontal or vertical [or diagonal?]) – domain independence is an illusion;
- Structural and system property grounding (tiered component frameworks) – structure neutrality is an illusion.

Now drop your illusions and get back to work :-)