Why Components?

„Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system" (Szyperski 1997)

The rationale behind component software:

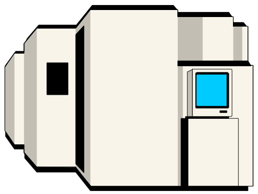
- Largely pushed by desktop – and Internet-based solutions.
- Complex technology to master – viable, component-based solutions will only evolve if benefits are clear.
- Benefits of traditional enterprise computing depend on enterprises willing to evolve substantially.

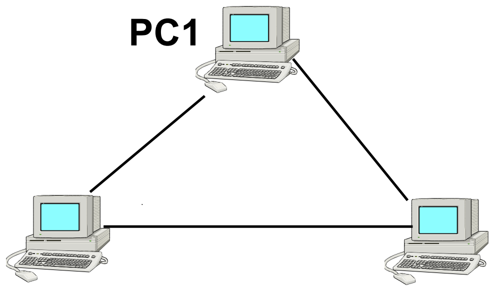
How to Create Standards

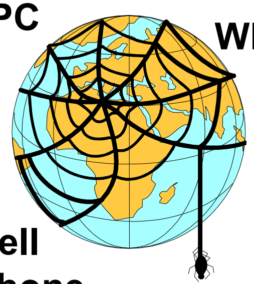
- Historically, closed solutions with proprietary interfaces addressed most customers' needs.
- Attempts to create low-level connection standards or wiring standards are either product or standard-driven.
  - Microsoft standards have always been product-driven.
  - COM-driven, incremental, evolutionary, legacy-laden by nature.
- Standard-driven approaches usually originate in industry consortia.
  - Prime example: Object Management Group (OMG)
    CORBA Beans as generalization of JavaSoft's Enterprise JavaBeans standards for components.
  - The EJB standard so far is not implementation language-neutral, bridging to existing services is non-trivial.

The Shifting Paradigm

Mainframes	*PC's*	*The Web*







HARDWARE	**SOFTWARE**	**MIDDLEWARE**
IBM	**MICROSOFT**	**???**
CLOSED PROPRIETARY	**CLOSED PROPRIETARY**	**OPEN STANDARDS**

The Internet World

- In the Internet world, the situation is different.
- Centralized control over what information is processed when and where is not an option.
- Content (web pages, documents) arrives at a user's machine and needs to be processed there and then.
- Monolithic applications have long reached their limit.
  - rapidly exploding variety of content types
  - open coding standards such as XML

- Flexibility of component software is its capability to dynamically grow to address changing needs.

Terms and Concepts

Components:

- are a unit of independent deployment;
- are a unit of third-party composition;
- have no persistent state.

Implications:

- A Component needs to be well-separated from its environment and from other components.
- A component encapsulates its constituent features.
- Components are never partially deployed.

Observations on Components

- Components need to come with clear specifications of what they provides and what they require.
  - Functional vs. non-functional properties
  - Well-defined interfaces and platform assumptions are essential.
  - Minimize hard-wired dependencies in favor of externally configurable providers.

- Components cannot be distinguished from copies of themselves.
- In any given process, there will be at most one copy of a particular component.
  - So, while it is useful to ask whether a particular component is available or not, it isn't useful to ask about the number of copies of that component.
- Many currently available components are heavyweights.
  - Database server, operating system services

Terms and Concepts (contd.)

Objects:

- are units of instantiation (Each object has a unique identity);
- have state that can be persistent;
- encapsulate their state and behavior.

Implications:

- Objects cannot be partially instantiated.
- Since an object has individual state, it also needs a unique identity to identify the object, despite state changes, for its lifetime.
- Nothing but an object‘s abstract identity remains stable over time.

Observations on Objects

- Objects need a construction plan that describes the new object's state space, initial state, and behavior before the object can exist.
  - Such a plan may be explicitly available and is then called a class.
  - Alternatively, it may be implicitly available in the form of an object that already exists, that is close to the object to be created, and can be cloned.
  - A preexisting object might be called a prototype object.
- The newly instantiated object needs to be set to an initial state.
  - The initial state needs to be a valid state of the constructed object, but it may also depend on parameters specified by the client asking for the new object.
  - The code that is required to control object creation and initialization could be a static procedure, usually called a **constructor**.
  - Alternatively, it can be an object of its own, usually called an **object factory**, or **factory** for short.

Object References and Persistent Objects

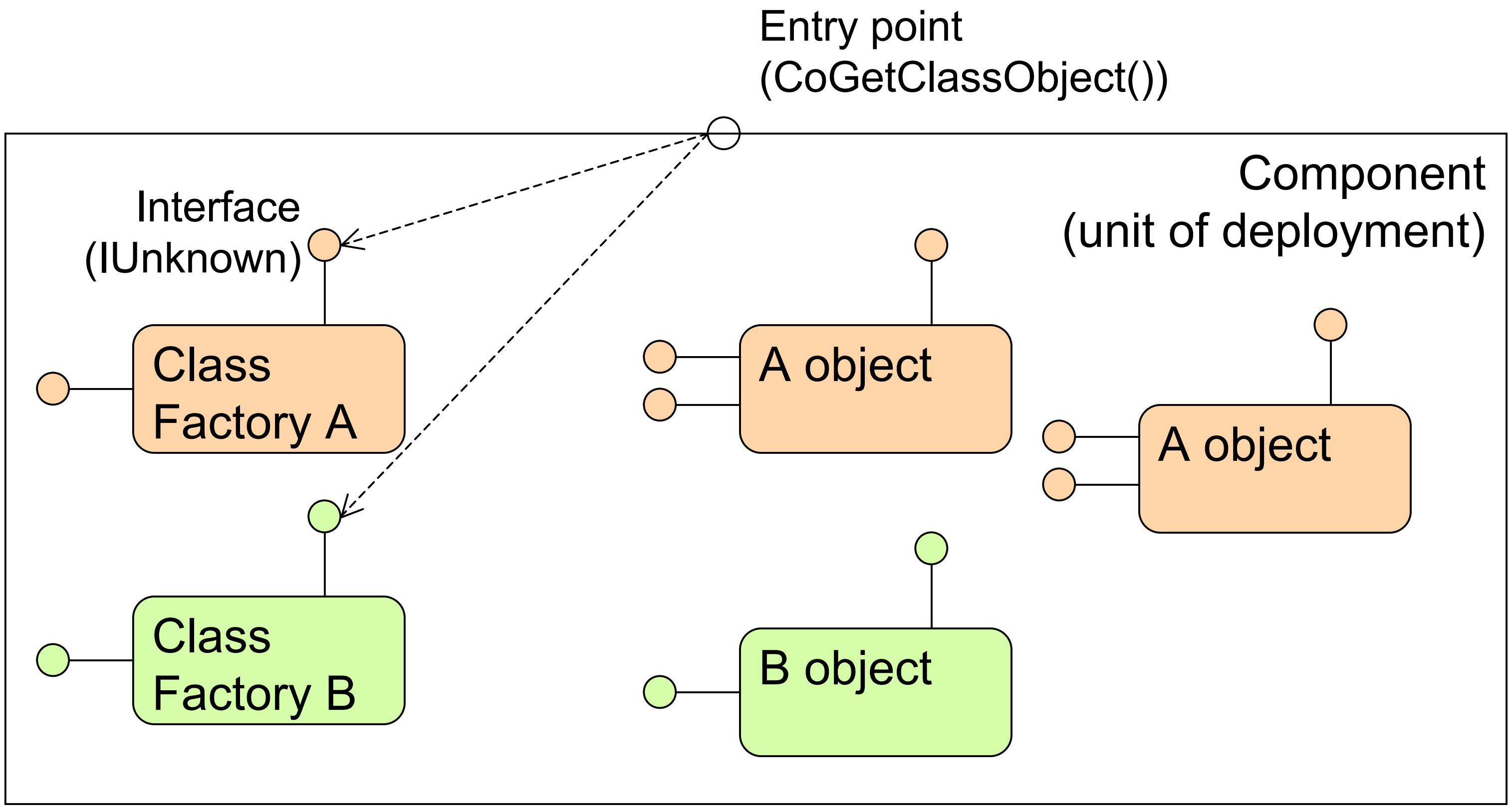
- The object's identity is usually captured by an object reference.
- Most programming languages do not explicitly support object references.
  - language-level references hold unique references of objects (usually their addresses in memory),
  - no direct high-level support to manipulate the reference as such.

- Distinguishing between an object and an object reference is important when considering persistence.
  - almost all so-called persistence schemes just preserve an object's state and class, but not its absolute identity.
  - An exception is CORBA, which defines interoperable object references (IORs) as stable entities (which are really objects). Storing an IOR makes the pure object identity persist.

Components and Objects

- A component comes to life through objects.
- It would normally contain one or more classes or immutable prototype objects.
  - In addition, it might contain a set of immutable objects that capture default initial state and other component resources.
  - No need for a component to contain only classes or any classes at all.
  - A component could contain traditional procedures and even have global (static) variables; or it may be realized in its entirety using a functional programming approach, an assembly language, or any other approach.
  - Objects created in a component, or references to such objects, can become visible to the component's clients, usually other components.
  - If only objects become visible to clients, there is no way to tell whether or not a component is purely object-oriented inside.

Components and Objects illustrated



Components are rather on the level of classes than of objects

Components and Objects (contd.)

- A component may contain multiple classes, but a class is necessarily confined to a single component;
- partial deployment of a class wouldn't normally make sense.
  - Just as classes can depend on other classes (inheritance), components can depend on other components (import).
  - The *superclasses* of a class do not necessarily need to reside in the same component as the class. Where a class has a *superclass* in another component, the *inheritance* relation *crosses* component *boundaries*.
  - Not clear, whether cross-component inheritance is a good thing.

Modules and Components

- Components are rather close to modules (early 1980s).
  - The most popular modular languages are Modula-2 and Ada (packages).
  - Support of separate compilation,
  - Proper type-check across module boundaries.
- Eiffel: „a class is a better module“.
  - justified idea that modules would each implement one abstract data type (ADT).
  - However, modules can be used to package multiple entities, such as ADTs or classes, into one unit.
  - Modules do not have a concept of instantiation, while classes do.
- Recent language designs keep the modules and classes separate.
  - Oberon, Modula-3, and Component Pascal are examples
  - Where classes inherit from each other, they can do so across module boundaries.
  - Even modules that do not contain any classes can function as components.

Modules and Components (contd.)

- Modules are not configurable:
  - There are no persistent immutable resources that come with a module, beyond what has been hardwired as constants in the code.
  - Resources parameterize a component (and are modified in builder tools).
  - Resources allow for versioning a component without needing to recompile.
- Resources are different from mutable component state!
  - Components are neither supposed to modify their own resources nor their code!
- Component technology unavoidably leads to modular solutions.
  - The software engineering benefits can thus justify initial investment into component technology, even if you don't foresee component markets.

Component: A Definition

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

(Workshop on Component-Oriented Programming, ECOOP, 1996.)

Interfaces

- A component's interfaces define its access points.
  - These points let clients access the component's services.
  - Components may have multiple interfaces.
  - Each access point may provide a different service.
- Interface specifications have contractual nature.
  - Component and clients are developed in mutual ignorance.
  - The standardized contract forms ground for successful interaction.
- Economy of scale:
  - interfaces should be simple, extensible and fulfill a market need.
- Common media to advertise interfaces is required
  - Unique naming scheme (e.g., ISBN numbers).
  - Component identifier is not required to carry any meaning.

Classes and Interfaces

- Interfaces are used to express type-compatibility between multiple independent classes
  - Interfaces express what is common across classes
  - Interfaces allow classes to share a common design
  - Interfaces identify subsets of the set of all possible objects
  - Interfaces enable real polymorphism
- Interfaces are used to constrain the types of objects a variable/parameter/field can refer to
- Classes are used to manufacture objects in memory
- Components expose interfaces rather than classes

Explicit Context Dependencies

- Besides specifying provided interfaces, components are also required to specify their needs.
  - What does the deployment environment need to provide, so that the components can function (so-called context dependencies).
  - For example, a mail-merge component would specify that it needs a file system interface.
- Problems with today's components:
  - The list of required interfaces is not normally available.
  - Emphasis is usually just on provided interfaces.
- Non-functional component properties are not addressed
  - CPU/memory usage, timing behavior, fault-tolerance properties.

Context Dependencies – the Reality

- In reality, several component worlds coexist, compete, and conflict with each other.
  - OMG's CORBA, Microsoft's COM+, Sun's JavaBeans (EJB).
  - Component worlds are fragmented by the various computing platforms. (This is not likely to change anytime soon.)
  - A component's context dependencies specification must include its required interfaces and the component world (or worlds) for which it has been prepared.
- Markets for cross-component-world integration.
  - Bridging solutions (i.e., OMG's „COM and CORBA Interworking" spec).
  - .NET might develop towards a bridge among component worlds.

Component-Based Programming vs. Component Assembly

- Component technology == “visual assembly” ?
  - Wiring components is surprisingly productive for simple applications
  - plumbing instead of programming: JavaSoft’s BeanBox
- Look behind the scenes:
  - Visual assembly tools register event listeners with event sources
  - Not the graph of particular assembled objects that is saved but enough information to generate a new graph of same topology
  - The newly generated graph and the original graph will not share common objects: the object identities are all different.
- The stored graph represents persistent state
  - but not persistent objects
  - Tools could hard-code component assembly; but object graph might be easier to modify at runtime

Persistent Objects

- Only supported in two contexts:
  - object-oriented databases, still restricted to a small niche of the database market.
  - CORBA-based objects.
- Object identity is preserved when storing objects.
  - Cannot be used to save state and topology but not identity.
  - Expensive deep copy of the saved graph required to undo the effort of saving the universal identities of the involved objects.
- Persistent identity is a heavyweight concept.
  - can always be added where needed.

Persistent Objects (contd.)

- Neither COM nor JavaBeans support persistent objects.
  - Emphasis on saving the state and topology of a graph of objects.
  - Java terminology: “object serialization.”
    (object graph serialization would be more precise.)

  - COM says „persistence“ although object identity is not preserved.
  - COM’s persistence mechanisms is equivalent to a deep copy of the object graph.
- COM monikers are objects that resolve to other objects.
  - Monikers may carry a stable unique identifier (a surrogate) and the information needed to locate that particular instance.
  - Java does not yet offer a standard like COM monikers.

Component Objects

- Components carry instances that act at run time:
  - As prescribed by their generating component.
  - In the simplest case, a component is a class and the carried instances are objects of that class.
  - Most components will consist of many classes.
- JavaBeans are externally represented by a single class:
  - One kind of object representing all possible uses of that component.
- COM components are more flexible:
  - Arbitrary collection of objects; clients see sets of unrelated interfaces.
- JavaBeans and CORBA merge multiple interfaces:
  - One implementing class only.
  - Important cases not properly handled (i.e.; multiple versions of an interface).
  - The OMG's CORBA Components proposal fixes this problem.

The Ultimate Difference

- Components capture the static nature of a software fragment.
- Objects capture its dynamic nature.
  - Simply treating everything as dynamic can eliminate this distinction.

- Good software engineering practices strengthen the static description of systems as much as possible.
  - Dynamics can always be superimposed where needed.
  - Meta-programming and just-in-time compilation simplify this soft treatment of the boundary between static and dynamic.

The Ultimate Difference (Contd.)

- It is advisable to explicitly capture as many static properties of a design or architecture as possible.
- This is the role of components and architectures that assign components their place.
- The role of objects is to capture the dynamic nature of the arising systems built out of components.

- Component objects are objects carried by identified components.
  - Both components and objects together will enable the construction of next-generation software.

Back to overview