

Components

- *"Executable units of independent production, acquisition, and deployment that can be composed into a functioning subsystem"* -- Szyperski
 - Different from the use of the term in software architecture
- Configurable metaproduct
- Must adhere to a component model and a target platform

The Third Way

1. *Buy*: Outsourced from large vendors

- Co-evolves with standards, reduced risk, reduced competitive edge, must be adapted to, must be adopted

2. *Build*: In-house, custom development

- Expensive and risky, but tailorable

3. **Components**

- **Customized during assembly**
- **Note: objects, as a realization of components, have not succeeded in the marketplace**

Market Development

- 1995 - 24% of developers using components
 - e.g. Visual Basic
- 1996 - 38% of "major corporations" have component strategies
- ORBs, 4GL/GUI component builders, component business applications, component frameworks
- 1998 - \$400M revenue from components
- COM, ActiveX, EJB, VCL (Borland)

Implications

- Configuration \Rightarrow generality \Rightarrow extra cost
 - Configuration enables late binding \Rightarrow difficult to unit test \Rightarrow more need for safe languages and tools
- Critical-mass phenomenon
 - That is, it is not until enough users have bought in that the advantages of using the approach are available

Implications - 2

- Independent deployment
 - Someone different from the developer is configuring and using
- 3rd-party composition
 - Someone else is using the component to build products
- Versioning
 - Backward compatibility
- Extensions (features)
- Contracts
 - Interface guarantees
- Objects
- Scalability

Versioning

- Late binding; indirection
- Greater possibility of version incompatibility
 - Greater need for checking on version #
- Strategies
 - Immutable interface (COM)
 - *Ad hoc* compatibility rules (Java)
 - Sliding window of supported versions
 - Backward compatibility
 - Version id and dependencies (Microsoft CLI)

Extensibility: Features

- At domain-analysis time *features*: "represent reusable, configurable requirements"
-- Czarnecki and Eisenecker
- At feature-modeling time a *feature* is a "property of a domain concept which is relevant to some domain stakeholder and is used to distinguish between concept instances"
-- Czarnecki and Eisenecker
- At OO-design and implementation time *features* are: the set of attributes and operations associated with a given class of instances

Dimensions and Orthogonality

- *Dimensions*: feature with a single set of mandatory alternative subfeatures
 - Car's color must be red, green or blue
- *Orthogonality*: independence of features
 - Interior and exterior colors can be individually selected

Extensions

- Singleton
 - At most one component allowed
- Parallel extension of multiple components in the same dimension
 - May lead to resource contention
- Orthogonal extensions
 - Independent of each other
- Recursive extension
 - A component introduces a new extension framework

Contracts and Guarantees

(Beugnard)

1. Syntactic contracts
 - Components are link-compatible
2. Correctness contracts
 - Pre and post conditions
3. Collaboration contracts
 - E.g. Synchronization
4. Quality of service contracts
 - Availability, MTBF, MTTR, throughput, latency, data integrity, capacity, monitoring

Contract Issues

- Typical situation: client calls component
 - Well-specified interface; expected result
 - Client is only concerned with component state before and after call (invariants)
- Callback: component calls client
 - Issue: component state is revealed to client; client can make changes during the course of executing callback
 - That is, callbacks make it more difficult for components to maintain their invariants

Objects as Components

- Objects complicate component contracts
 - Presence of state together with callbacks
 - Self recursion; that is, methods can call other methods in class
- If client is responsible for representing the state of a component (such as a when the component is being visualized or accessed through a client interface), there is the possibility of client and server getting out of sync
 - Component gets event, makes change to model and notifies client before completely re-establishing invariant
 - Client queries component possibly assuming that the component is sane (all invariants hold)
 - May require time stamping and conditional code
- Situation is even worse with multithreading

Inheritance

- Subclassing
 - Inheritance of implementation
- Subtyping
 - Inheritance of interface
- Substitutability
 - No language-enforceable mechanism

Subtype Polymorphism

- Contract maintenance in the face of subtypes
 - Trying to maintain substitutability
- Contract specifies pre/post conditions on calls, including type information
- But what happens in the presence of subtypes? That is, when can you substitute one component for another?
With regard to a single method m with replacement method m' , the following rules must hold
 - Arguments of m' must be supertypes of arguments of m (*contravariance*). That is, m' must accept at least as much as m
 - Return value of m' must be a subtype of the return value of m (*covariance*). That is, m' can't produce anything more than m

Types of Polymorphism

- Inclusion (subtype)
- Overloading (*ad hoc*)
 - Multiple methods with the same names but different signatures
- Generics / templates (parametric)
 - Boundedness

Multiple Inheritance

- Name clashes
- Disjoint \Rightarrow catenation of states
- "Diamond" - shared ancestor
 - Duplicate or shared (virtual) state?
 - Independent method override

Mixins

- Parent class is a parameter to generic class definition
 - That is, class being defined could be mixed into any other class mentioned as a parameter
- Multiple mixin can be used to implement (non-functional) *policies*

The *Fragile Base Class* Problem

- What happens when new versions change base classes? Are existing derived classes broken?
 - Syntactic (binary compatibility) issue
 - Can recompilation be avoided?
 - Initialize dispatch tables at load time
 - Semantic
 - Overriding; calls to overridden code; all the problems of callbacks and more
 - Need contracts between base and derived classes

Issues in Scaling Up Components

- Abstraction
 - What level of abstraction should components be?
 - Low coupling implies reduced object complexity
 - High coupling, as in OO frameworks, can reduce client complexity
- Accounting
 - How should component use be charged for?
 - Overhead vs. accountability
- Analysis
 - What checks can be made when composing / configuring an assembly of components?
 - E.g. type checking across component boundaries
- Compilation
 - What should be the unit of compilation for independent components?
 - Responsiveness vs. optimization

Scaling Issues - 2

- Delivery
 - What is the appropriate unit of marketing and delivery?
 - Classes are too tightly coupled
- Deployment
 - *"Readying a unit for operation in an environment"*
 - XML manifests plus configuration preprocessing
- Disputes
 - Multiple vendors
- Extensions

Scaling Issues - 3

- Fault containment
- Instantiation (objects, plugins)
- Installation
- Loading (Java class loading)
 - Side-by-side loading of multiple versions
- Locality (distribution)
 - Minimize cross-component requests
- Maintenance
- System management (availability, load)

Component Frameworks

- CORBA (OMG)
 - OMA (Object Management Architecture), IDL (Interface Definition Language), CCM (CORBA Component Model)
- Microsoft
 - COM (Component Object Model), DCOM (Distributed COM), OLE (Object Linking and Embedding), ActiveX (web controls), COM+ (transactions)
 - .NET, CLI (Common Language Infrastructure), CLR (Runtime), ASP.NET
- JAVA (SUN)
 - Beans, EJB (Enterprise Java Beans), J(2)EE, JSP (Java Server Pages)

Connecting Components

- Indirections inherent in late binding
- Push vs. pull
- Independent connection mechanisms
 - Event and message services
 - Connections as independent entities
- Independent connection notations
 - XML, SOAP
- Integrity
 - Compiler, run-time, sandbox

Comparison: Shared Attributes

- Late binding, encapsulation, subtyping
- Some form of component transfer packaging
 - Java jar files, COM cab files, CLI assemblies
- Uniform data transfer mechanism
- Events and channels
- Meta-information and reflection
- Persistence
- Deployment descriptions
- Component model
 - Application server models (EJB, COM+, CCM)
 - Web server models (JSP, ASP.NET)

Comparison: Differences

- Bridging between component technologies
 - CORBA-COM; JavaBeans-ActiveX
- Memory management
 - Garbage collection (Java, CLR), reference counting (COM), nothing for CORBA
- Container managed persistence
 - EJB, CCM (CORBA) but nothing for CLR or COM+
- Versioning
 - Frozen, version #s, compatibility rules, side-by-side execution

Differences - 2

- Industrial-strength implementations and applications
 - COM (client & desktop), J2EE/COM (server solutions), CORBA (legacy applications)
- Development environments
- Services (transactions, messaging, etc.)
 - CORBA limited, COM+ and Java rich, CLR there but not CLI compliant
- Extent of support for web service (JSP & ASP.NET)
- Protocols
 - Java, CORBA (IIOP & XML), Java (RMI), COM, CLR (DCOM), CLR (XML, SOAP)

Comparison: Supported Variability

- Java, CLI
 - Single virtual machine for all platforms
 - Many languages can generate byte codes
- COM
 - Multiple languages
 - But Microsoft platforms only
- CORBA
 - Each language must have an IDL binding
 - Each platform must have an ORB

Ongoing Concerns

- Domain standards
 - Tension between proprietary solutions and domain standards
 - Long time for approval and penetration
- Rethinking software engineering
 - Independent extensibility
 - Reduced ability to do integration testing
 - Feature orientation

Concerns - Object Orientation

- CORBA, COM, CLR are language neutral which makes imposing an object model problematic
- Transmittal of object references across machines
 - Java RMI
 - COM interface references
 - CORBA - distinction between local and distributed objects, "stringified" persistence
 - CLR - unified type system; persistence provided by a service
- Mobile objects (not references)

Future Directions

- Liability / accountability
- Quality guarantees (execution time, resources)
- Contract persistence over versions
- Semantic inter-component compatibility (e.g. reentrancy)