Birkbeck
UNIVERSITY OF LONDON

Enterprise Computing
Component-Based Software Development

Introduction to CBSD: Background, Motivation, Definitions, Concept, and Principles

Instructor: Keith Mannock
http://www.dcs.bbk.ac.uk/~keith

1st Lecture

Acknowledgements

- George T. Heineman
- William Councill
- Clemens Szyperski
- Dr. Bernd Bruegge
- Dr. Allen Dutoit
- …

Agenda

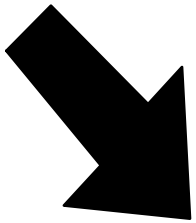
- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Overview

What is Software Engineering? (1)

- Systematic approach for developing software.
- "Methods and techniques to develop and maintain quality software to solve problems." [Pfleeger, 1990]
- "Study of the principles and methodologies for developing and maintaining software systems." [Zelkowitz, 1978]
- "Software engineering is an engineering discipline which is concerned with all aspects of software production." [Sommerville]

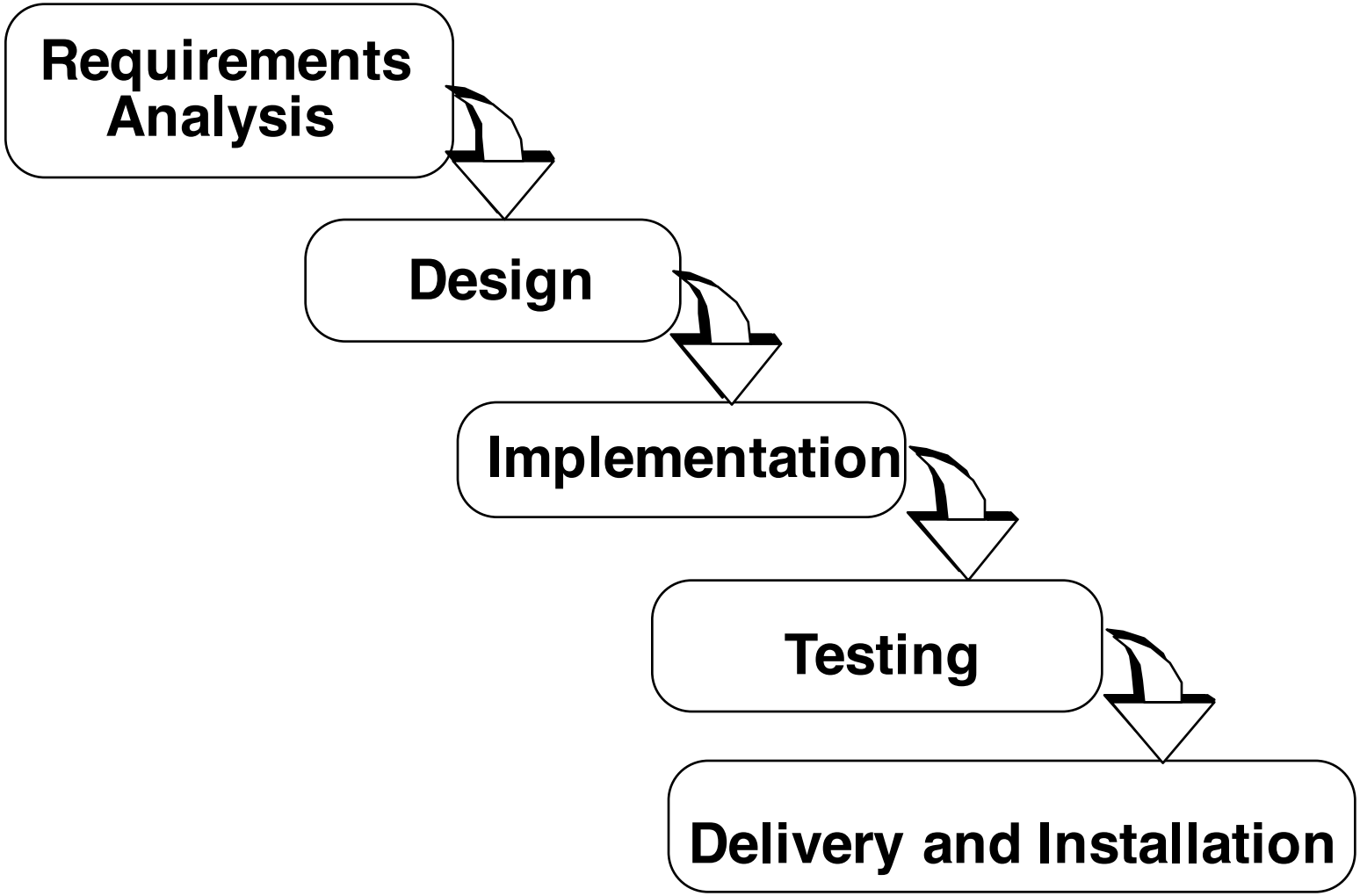
Our Intention

Requirements



Software

Our plan of attack (old style?)



How to Deal with Complexity? 1

- Three ways to deal with complexity:
  - Abstraction
  - Decomposition
    - Technique: Divide and conquer
  - Hierarchy
    - Technique: Layering
- Two ways to deal with decomposition:
  - Functional Decomposition
  - Object Orientation

How to Deal with Complexity? 2

- What is the right way?
  - Functional Decomposition
    - may lead to unmaintainable code.
  - Object Orientation
    - depending on the purpose of the system, different objects may be found.
- How to proceed?
  - Start with a description of the functionality
    - Use case model
  - Then find objects
    - Object model
- What activities and models are needed?
  - This leads us to the software lifecycle we use in this class.

Agenda

- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Review

Reuse Motivation

- Observation about Modeling of the Real World in [Gamma et al 94]
  - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.
  - There is a need for *reusable* and flexible designs.
  - Design knowledge complements application domain knowledge and solution domain knowledge.

Reuse Heuristics

- Look for existing classes in class libraries
  - JSAPI, JTAPI, ...
- Select data structures appropriate to the algorithms
  - Container classes
  - Arrays, lists, queues, stacks, sets, trees, ...
- Define new internal classes and operations only if necessary
  - Complex operations defined in terms of lower-level operations might need new classes and operations

Reuse

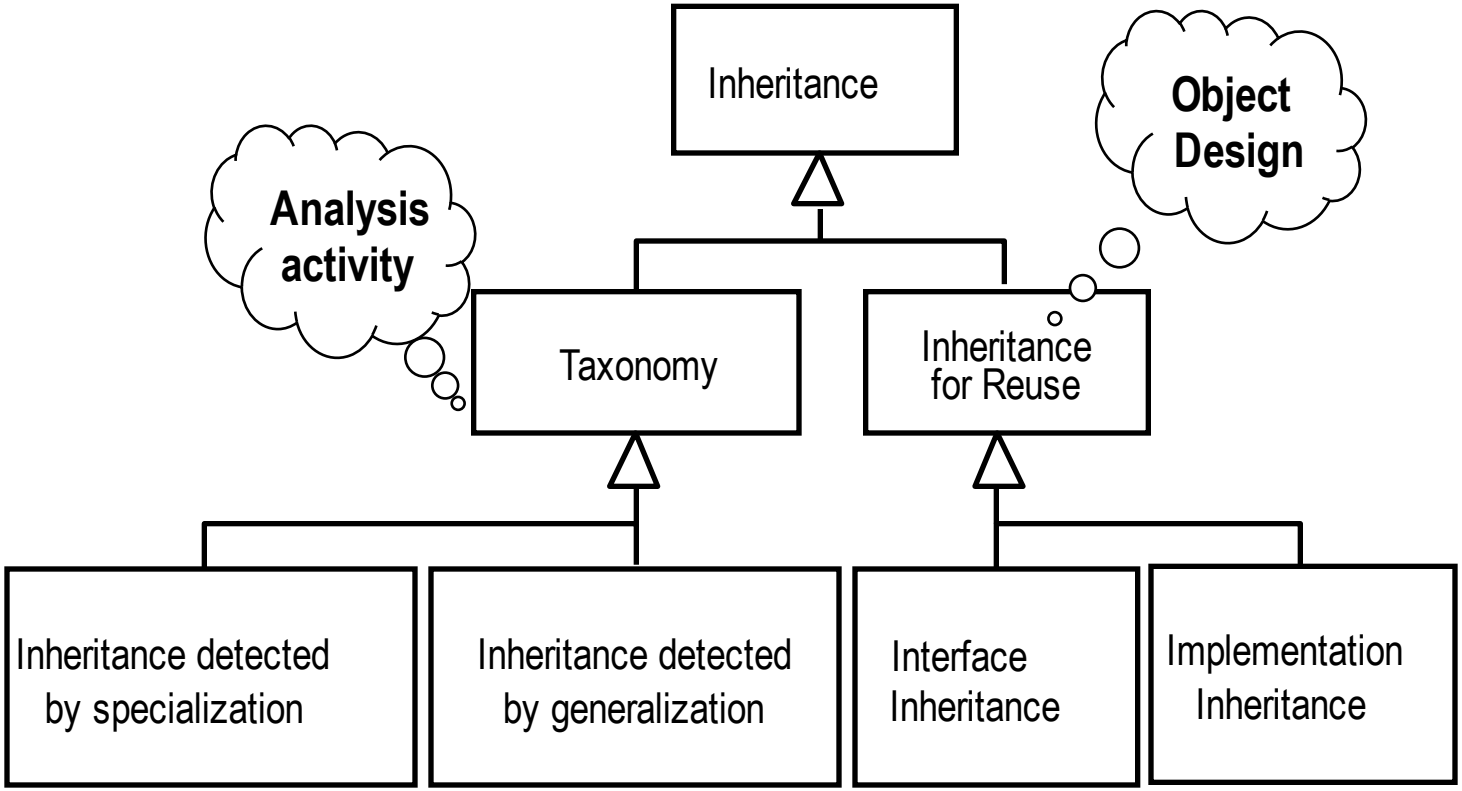
- Main goal:
  - Reuse ***knowledge*** and ***functionality*** from previous experience to current problem.
  - Approaches: Inheritance and Delegation
- Inheritance
  - Interface inheritance
    - Used for concept classifications ⇒ type hierarchies
  - Implementation inheritance
    - Used for code reuse.
- Delegation
  - An alternative to implementation inheritance.
    - Used for code reuse.

Agenda

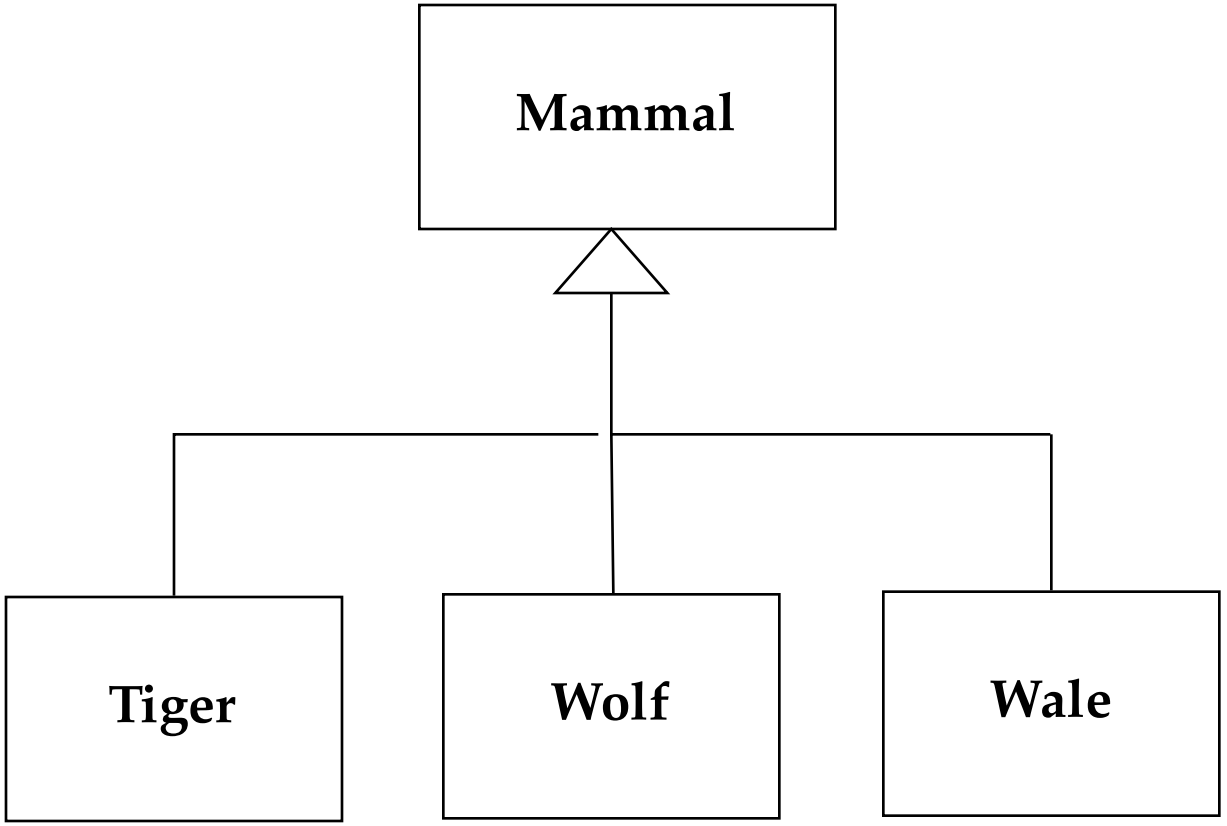
- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Review

Metamodel for Inheritance

- Inheritance is used to achieve two different goals: Taxonomies and Reuse.



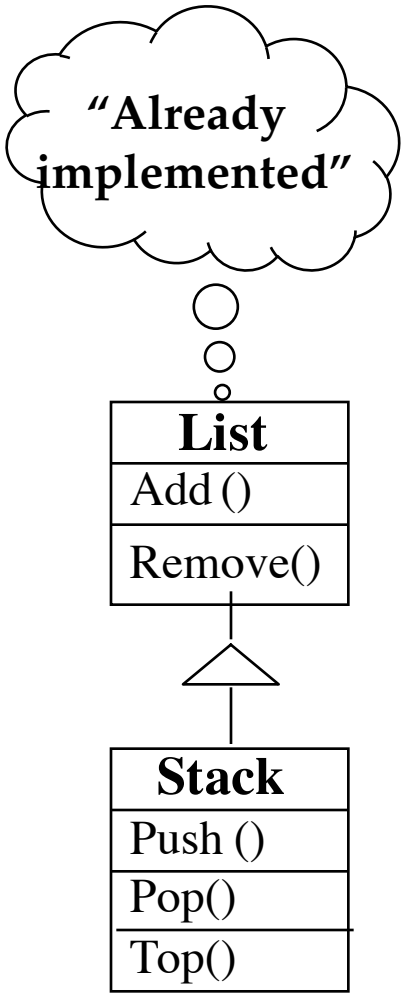
Taxonomy Example



Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.

- Example:
  - I have a **List** class, I need a **Stack** class.
  - How about subclassing the **Stack** class from the **List** class and providing three methods, Push() and Pop(), Top()?



Problems

- Problem with implementation inheritance:
  - Some of the inherited operations might exhibit unwanted behavior.
  - What happens if the Stack user calls Remove() instead of Pop()?
  - What happens if the superclass implementation is modified?
  - The subclass developer is exposed to internal information about the superclass and this may result in tight coupling of subclass to superclass.

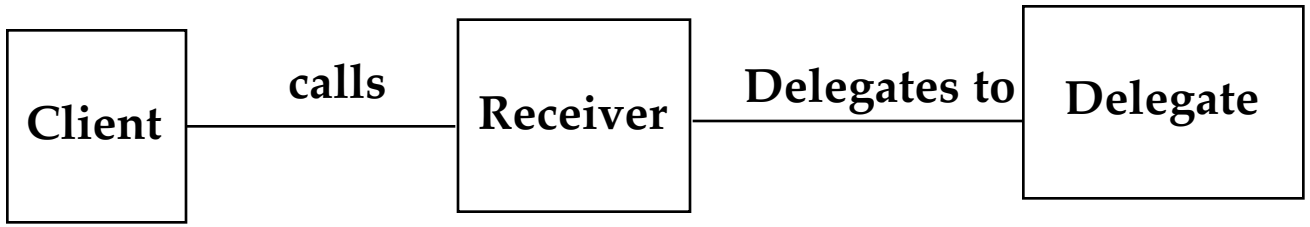
Impl. vs Interface Inheritance

- Implementation inheritance
  - Also called ***class inheritance***
  - Goal: Extend an application's functionality by reusing functionality in parent class
  - Inherit from an existing class with some or all operations already implemented
- Interface inheritance
  - Also called ***subtyping***
  - Inherit from an abstract class with all operations specified, but not yet implemented

Delegation

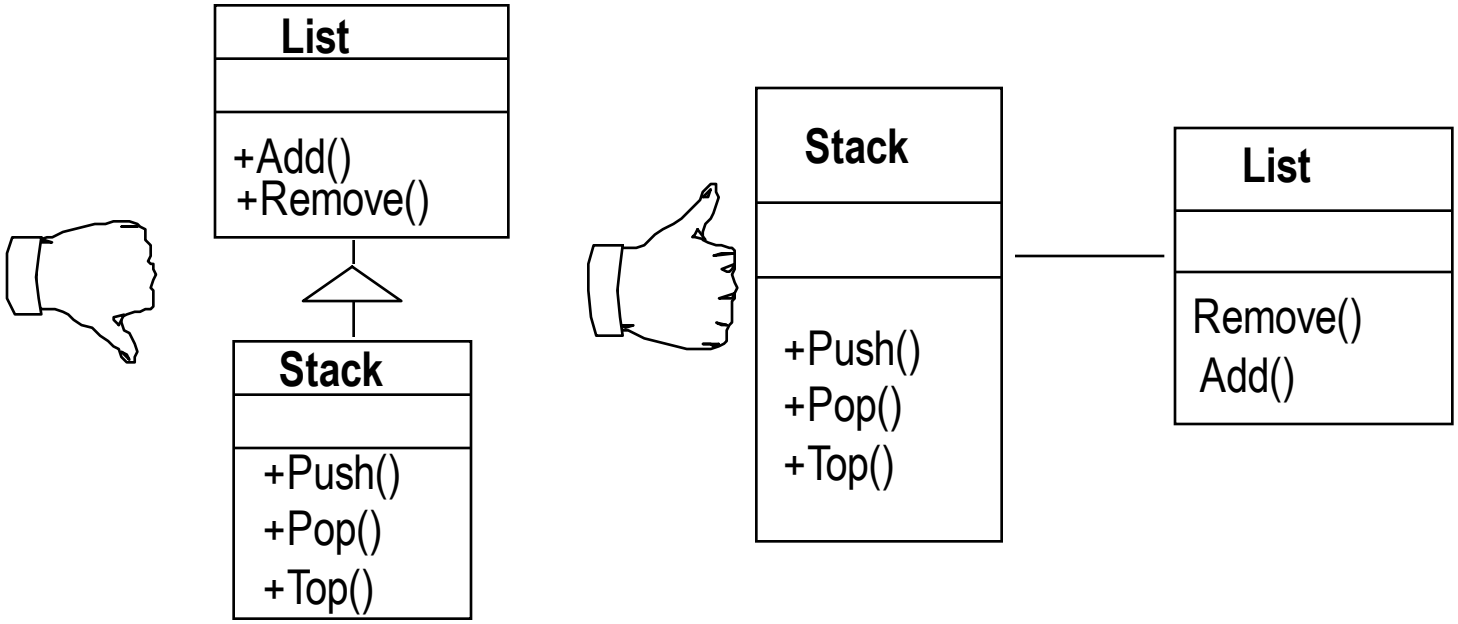
- Delegation
  - is an alternative to implementation inheritance.
  - is as powerful as implementation inheritance.

- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object.



Delegation vs. Impl. Inheritance

- **Inheritance**: Extending a base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?



Comparison

- Delegation
  - Pro:
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - Con:
    - Inefficiency: Objects are encapsulated.
- Inheritance
  - Pro:
    - Straightforward to use
    - Supported by many programming languages
    - Easy to implement new functionality
  - Con:
    - Inheritance exposes a subclass to the details of its parent class
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Agenda

- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Review

Reuse Code

- Select existing code
  - class libraries
  - frameworks
  - components
- Adjust the class libraries, framework or components
  - Change the API if you have the source code.
  - Use the adapter or bridge pattern if you don't have access
- What are the differences among class libraries, frameworks, and components?
  - They are all composed of a set of related classes.

Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.
- Frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or real-time avionics.

Key Benefits

- Reusability
  - leverages of the application domain knowledge and prior effort of experienced developers .
- Extensibility
  - is provided by *hook methods*, which are overwritten by the application to extend the framework.
  - Hook methods systematically decouple the interfaces and behaviors of an application domain from the variations required by an application in a particular context.

Classification of Frameworks

- **White-Box Frameworks**
  - Extensibility achieved through inheritance and dynamic binding.
  - Existing functionality is extended by subclassing framework base classes and overriding predefined hook methods
  - Often design patterns such as the template method pattern are used to override the hook methods.

- **Black-Box Frameworks**
  - Extensibility achieved by defining interfaces for components that can be plugged into the framework.
  - Existing functionality is reused by defining components that conform to a particular interface
  - These components are integrated with the framework via delegation.

Another Classification of Frameworks

- Infrastructure frameworks
  - aim to simplify the software development process
  - System infrastructure frameworks are used internally within a software project and are usually not delivered to a client.
- Middleware frameworks
  - are used to integrate existing distributed applications and components.
  - Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].
- Enterprise application frameworks
  - are application specific and focus on domains
  - Example domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.

Class libraries and Frameworks

- Class Libraries:
  - Less domain specific
  - Provide a smaller scope of reuse.
  - Class libraries are passive; no constraint on control flow.
- Framework:
  - Classes cooperate for a family of related applications.
  - Frameworks are active; affect the flow of control.
- In practice, developers often use both:
  - Frameworks often use class libraries internally to simplify the development of the framework.
  - Framework event handlers use class libraries to perform basic tasks (e.g. string processing, file management, numerical analysis….)

Components and Frameworks

- Components
  - Self-contained instances of classes.
  - Plugged together to form complete applications.
  - Defines a cohesive set of operations
  - Can be used based on the syntax and semantics of the interface.
  - Components may even be reused on the binary code level.
    - The advantage is that applications do not always have to be recompiled when components change.

- Frameworks:
  - Often used to develop components.
  - Components are often plugged into black-box frameworks.

Agenda

- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Review

Other Engineering Fields

- Every other engineering profession
  - divides problems into sub-problems
    - Ready-made solutions
    - Sub-contracted to conform to specifications
  - conforms to standards
  - self-imposed industrial regulation
  - certification regiment for engineers

- But not Software Engineering…

Some Excuses

- Software is different from hardware
- My software needs are unique
- We don't trust software written elsewhere
- Users can live with defective software

- We want something better…

CBSE Methodology

- At its core, CBSE is a methodology that:
  - constructs applications from software units
  - supports vendors in producing reusable software units
  - develops core technologies that support the run-time execution of these units
  - defines agreed-upon standards
  - engenders trust among participants

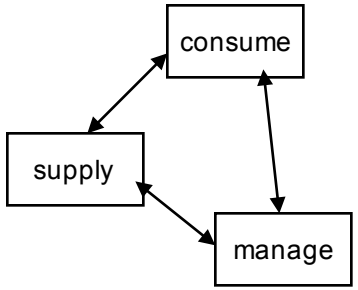
Software Component

A *Software Component* is

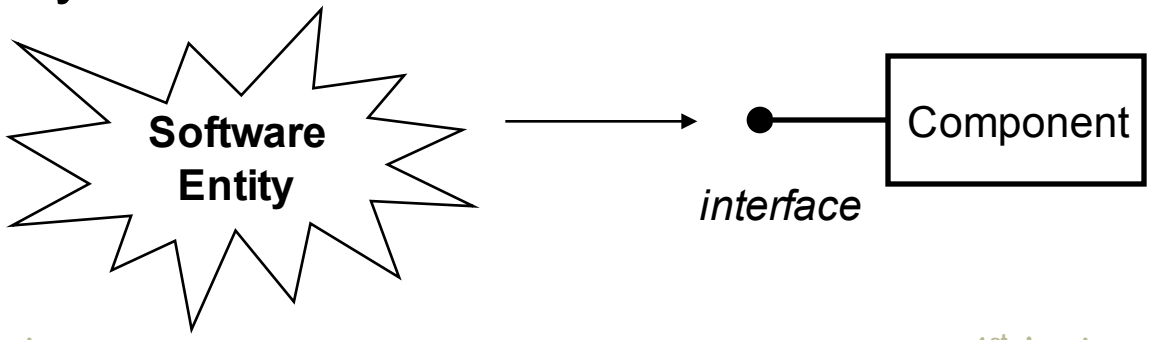
- a software element
- that conforms to a component model
- and can be independently deployed
- and can be composed without modification
- all above according to a composition standard

Component Model

A *Component Model* defines specific interaction and composition standards.

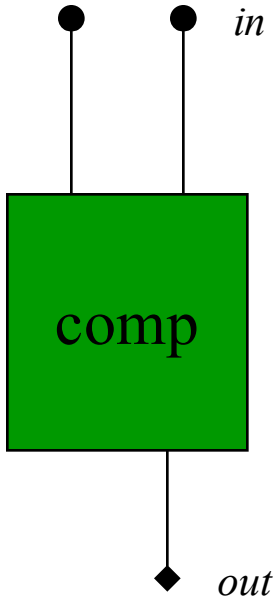


An interaction standard is the mandatory requirements employed and enforced to enable software elements to directly and indirectly interact with other software elements.



Interaction Standard

- Based on the concept of an interface
  - "An abstraction of the behavior of a component"
- Provided interface
  - component implements the interface
- Required interface
  - component interactions with other software elements are captured
- Context Dependency



Composition Standard

- Composition
  - the combination of two or more software components yielding a new component behavior

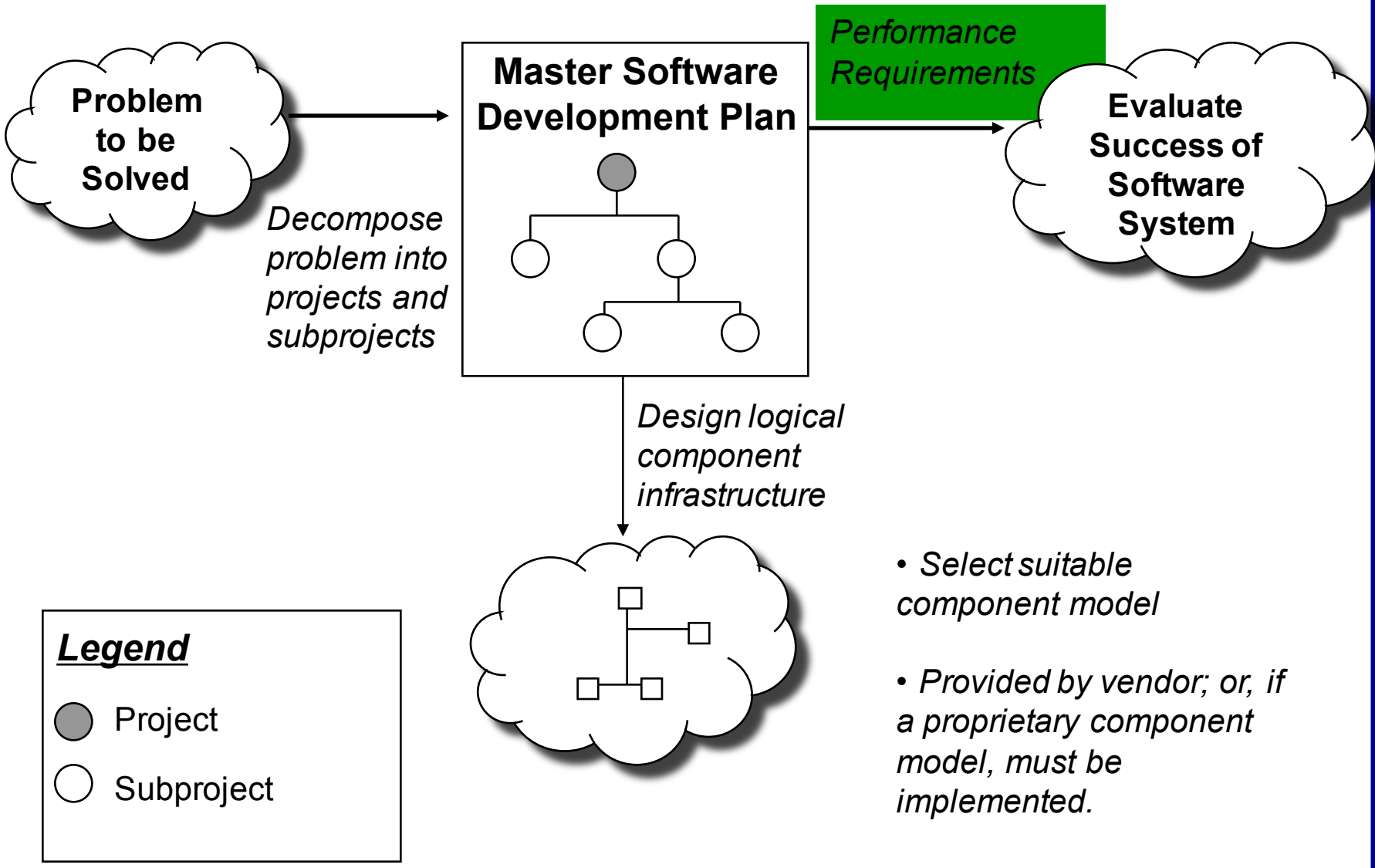
- Defines several important concepts
  - Packaging
  - Installing
  - Instantiating
  - Deploying
  - Configuring

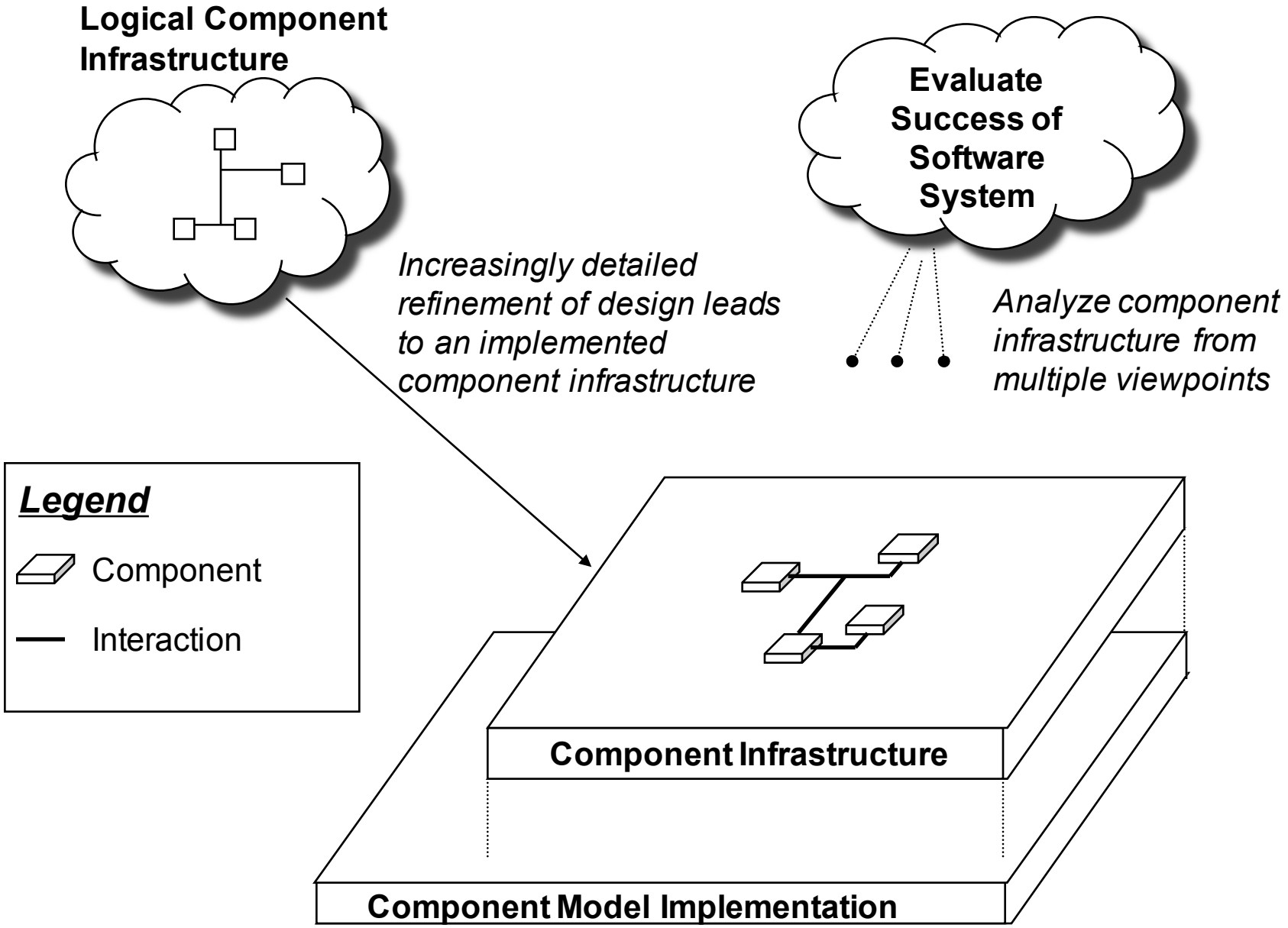
A *Component Model Implementation* is

- the dedicated set of executable software elements
- required to support the execution of components
- that conform to the model

- The "Plumbing" that enables components to connect and interact with each other

Big Picture


Problem to be Solved
Decompose problem into projects and subprojects
Master Software Development Plan
Performance Requirements
Evaluate Success of Software System
Design logical component infrastructure
Legend
Project
Subproject
• Select suitable component model
• Provided by vendor; or, if a proprietary component model, must be implemented.


Logical Component Infrastructure
Evaluate Success of Software System
Increasingly detailed refinement of design leads to an implemented component infrastructure
Analyze component infrastructure from multiple viewpoints
Legend
Component
Interaction
Component Infrastructure
Component Model Implementation

Agenda

- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Review

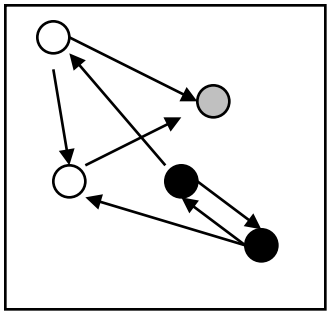
Component Infrastructure

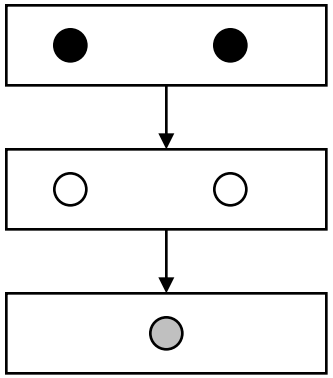
- A *software component infrastructure* is
  - a set of interacting software components

- The infrastructure is designed to ensure that a software system constructed using those components will satisfy clearly defined performance requirements

Designing a Component Infrastructure

- The Component Infrastructure is layered
- Layering is a strategy for decomposing complex structures





Four Layers

- User
- Workflow and Process Control
- Business Services
- Data and Operating System Services

User

- User components provide the external interface
  - graphical user interfaces (GUI), Web-based, or batch-oriented
- Understand user interactions
- Request services from other components in response to commands issued by the user

Workflow and Process Control

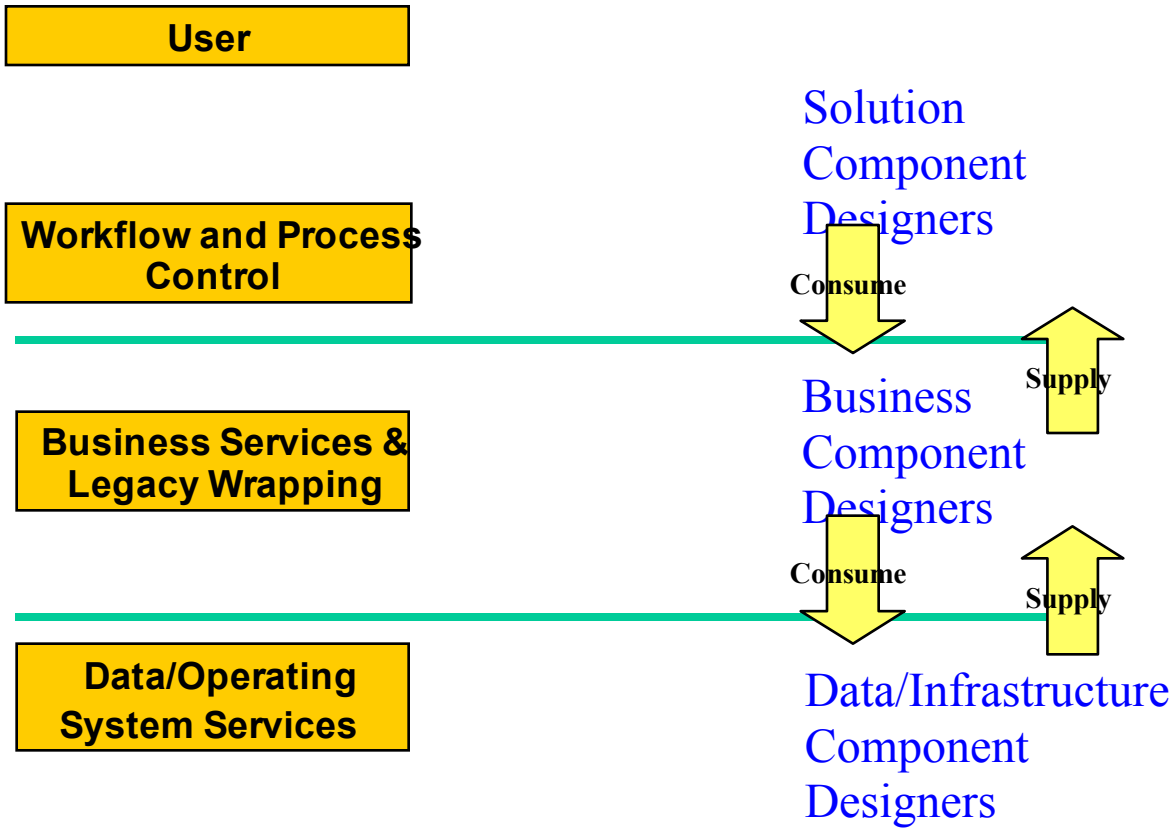
- Workflow and process control components manage complex, automated business processes that interact with (potentially) many business services

Business Services

- Business service and legacy wrapping components provide the implementation of the business rules and the operational activity

- This is accomplished using internal business object implementations, or by wrapping legacy systems behind the component interface

Design Layers and Designers

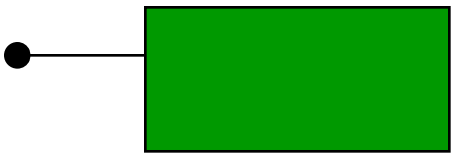


Component Model

- Interfaces
  Naming
  Meta data
  Composition
  Customization
  Interoperability
  Evolution Support
  Packaging & Deployment

Specification of component behavior and properties

Definition of Interface Description Language (IDL)



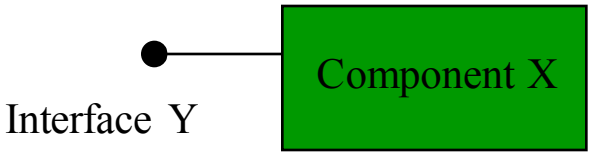
Agenda

- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Overview

Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

Global unique names for interfaces and components

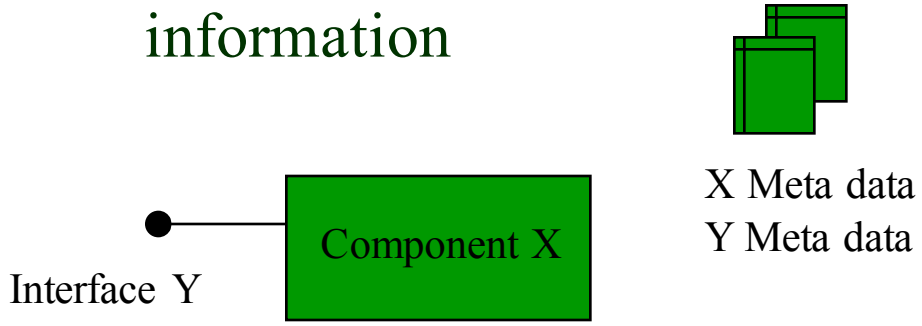


Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

Information about components, interfaces, and their relationships

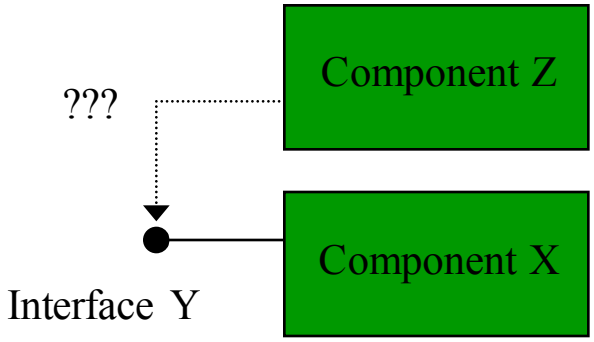
Interfaces to services providing such information



Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

Interfaces and rules for combining components to create larger structures and for substituting and adding components to existing structures

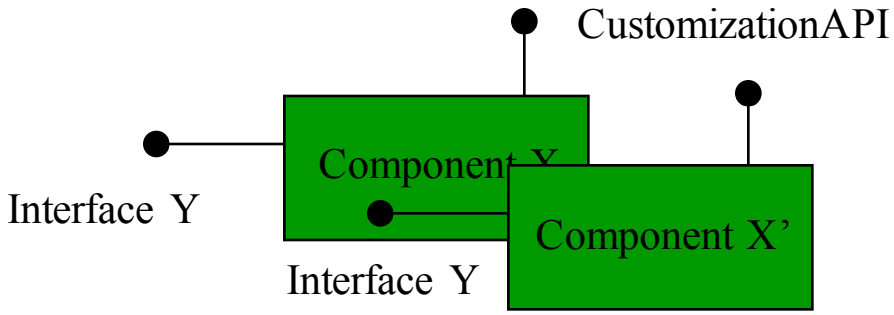


Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

Interfaces for customizing components

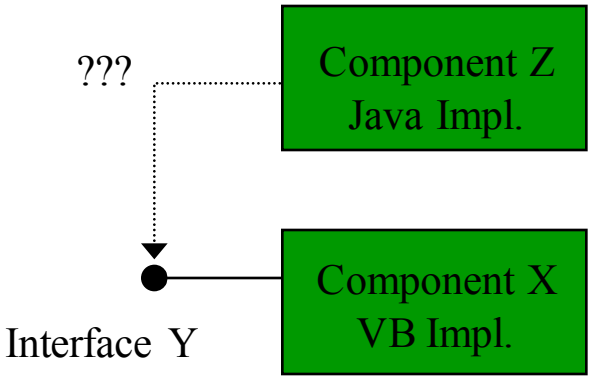
User-friendly customization tools will use these interfaces



Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

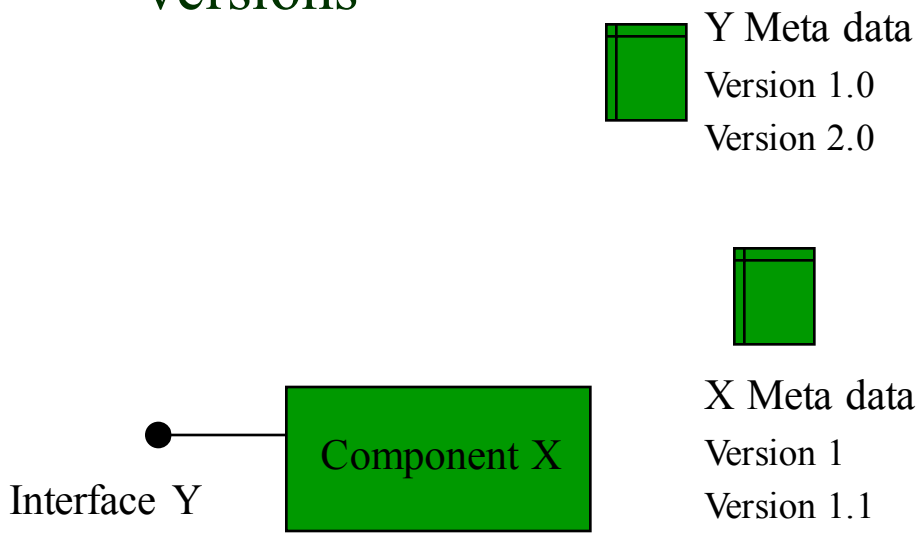
Communication and data exchange among components from different vendors, implemented in different languages



Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

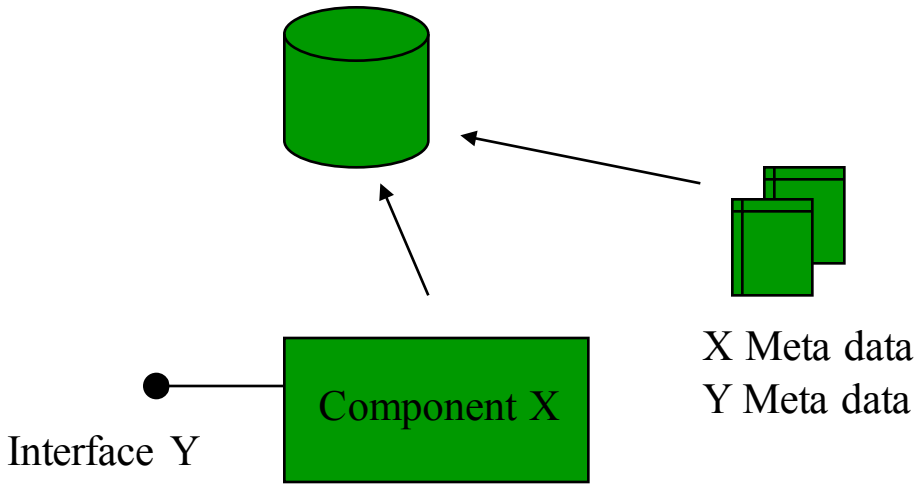
Rules and services for replacing components or interfaces by newer versions



Component Model

- Interfaces
- Naming
- Meta data
- Composition
- Customization
- Interoperability
- Evolution Support
- Packaging & Deployment

Packaging implementation and resources needed for installing and configuring a component



Agenda

- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Overview

Terminology (1)

- participants – all persons involved in a project.

  e.g., developers, project manager, client, end users.

- role – associated with a set of tasks assigned to a participant.

- system – underlying reality.

- model – abstraction of the reality.

- work product – an artifact produced during development.

Terminology (2)

- deliverable – work product for client.
- activity – a set of tasks performed toward a specific purpose.
- milestone – end-point of a software process activity.
- task – an atomic unit of work that can be managed and that consumes resources.
- goal – high-level principle used to guide the project.

Terminology (3)

- functional requirement – describe the interaction between the system and its actors (*e.g.,* end users and other external systems) independent of its implementation.
- nonfunctional requirement – describe aspects of the system that are not directly related to the functional requirements of the system (*e.g.,* QoS, security, scalability, performance, and fault-tolerance).
- notation – is a graphical or textual set of rules representing a model (*e.g.,* UML*)*
- method – a repeatable technique for solving a specific problem e.g. *sorting algorithm*
- methodology – a collection of methods for solving a class of problems (*e.g.,* Unified Software Development Process).

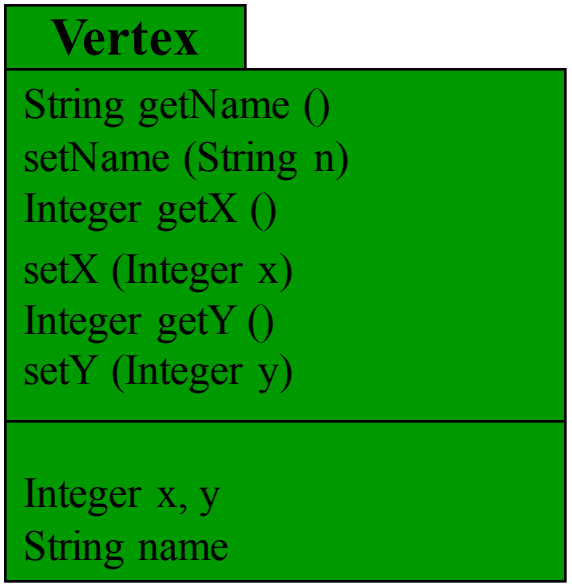
Agenda

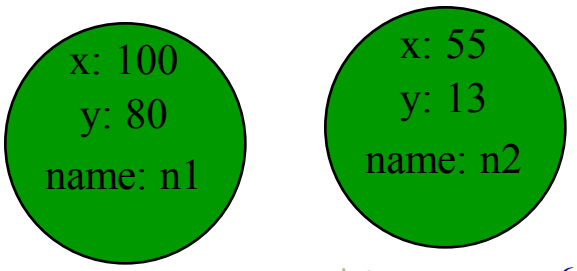
- Background
  - Software Engineering
  - Software Complexity
- Why components?
  - Reuse Concepts
  - Inheritance and Delegation
  - Libraries, Frameworks, and Components
- Component Based Software Engineering
  - Definitions
  - Component Infrastructure
  - Component Model
- Recap
  - Terminology
  - Java Quick Review

Object-oriented Class

- Class
  - name
  - set of attributes
    - information of interest
  - set of methods
    - functionality of the class
  - public declarations

- Object
  - has all the attributes of a class
    - (“belongs” to the class)
  - exhibits the specified functionality
  - private implementation

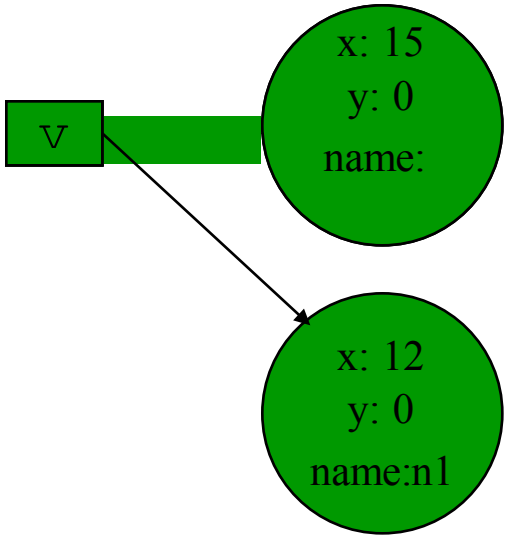




Objects are instantiated

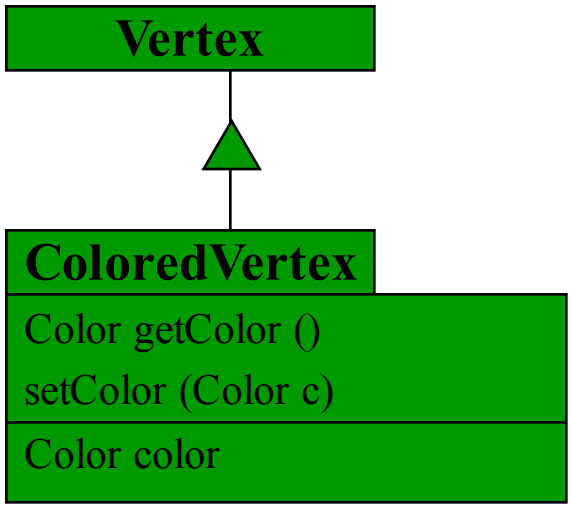
```
Vertex v;
```

```
v = new Vertex ();
v.setX(15);
v = null;
v = new Vertex ();
v.setX(12);
v.setName("n1");
```



Inheritance

- Mechanism to create new classes
  - `ColoredVertex` IS A `Vertex`
  - `Vertex` is the superclass
  - `ColoredVertex` is a subclass
- Multiple inheritance
  - Supported by C++ not Java
- Enables polymorphism
- Supports overriding



Polymorphism

- Concept that a number of different (yet related) operations have the same name
  - improvement over C
  - increased readability of code
- Consider ability of Vertex to draw()

```
forall vertices v in graph
    v.draw()
```

Overriding

- When subclass provides alternate implementation to an inherited method
- Also useful for realizing concrete subclasses for abstract base classes

Access rights

- Designer can restrict access to
  - classes, methods, attributes

- Public – all classes can access
- Private – only defining class can access
- Protected
- (java) default package

Java – Interfaces

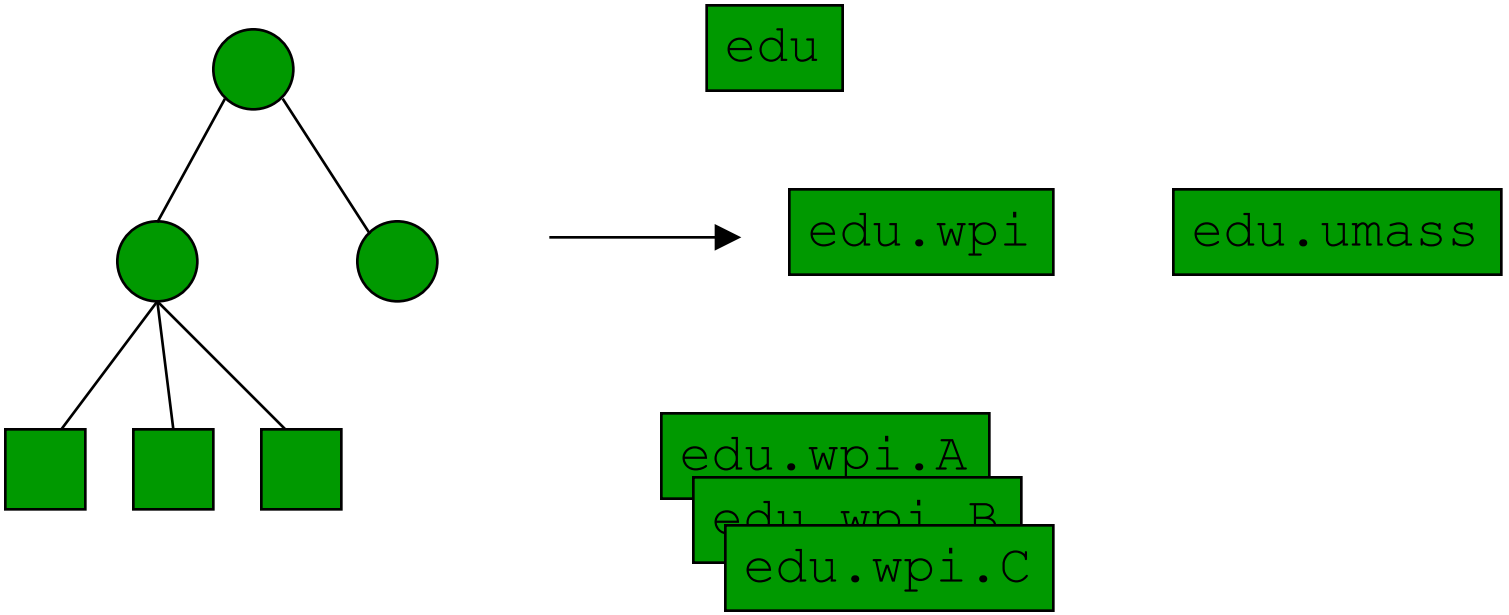
- Contractual obligation on a class
- A class *implements* an interface

```
public interface ICalc {
        /** return the multiplication of a*b. */
        int multiply (int a, int b);
}
```

```
public class Calculator implements IBlock, ICalc {
    …
    public int multiply (int x, int y) { … }
}
```

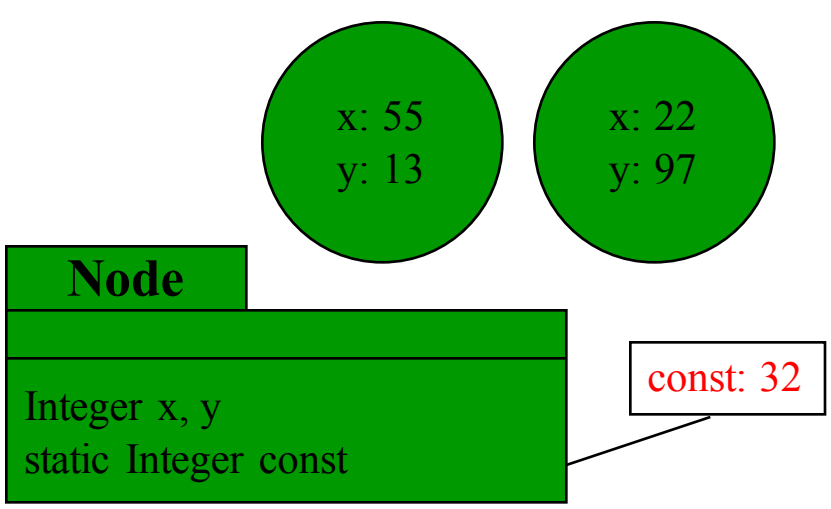
Java – Packages

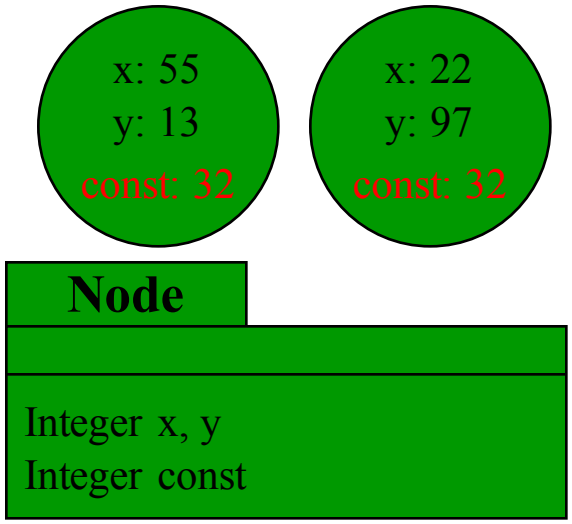
- Grouping of classes
- One-to-one mapping to directory structure



Java – Static data

- In Java, all data must be defined in a class
- What about data common to all objects within a class?





Accessing Static Data/Methods

```
Node n = new Node ();
n.setX (10);
n.setY (20);

// Associated with the class
int x = Node.const;

// Can also work on objects (simply delegated to class)
int y = n.const;
```

- Similar access works for static methods

Java – Coupled Communication

```
public class Agent {
  private Manager mgr;
  public Agent (Manager m)
{
    mgr = m;
  }

  public void
requestWork() {
    mgr.showHow();
  }
}
```

```
public class Manager {
 public Manager () {
 }

  public void showHow() {
  }
}
```

- `Agent` can't be used without `Manager`

Java – Decoupled Communication

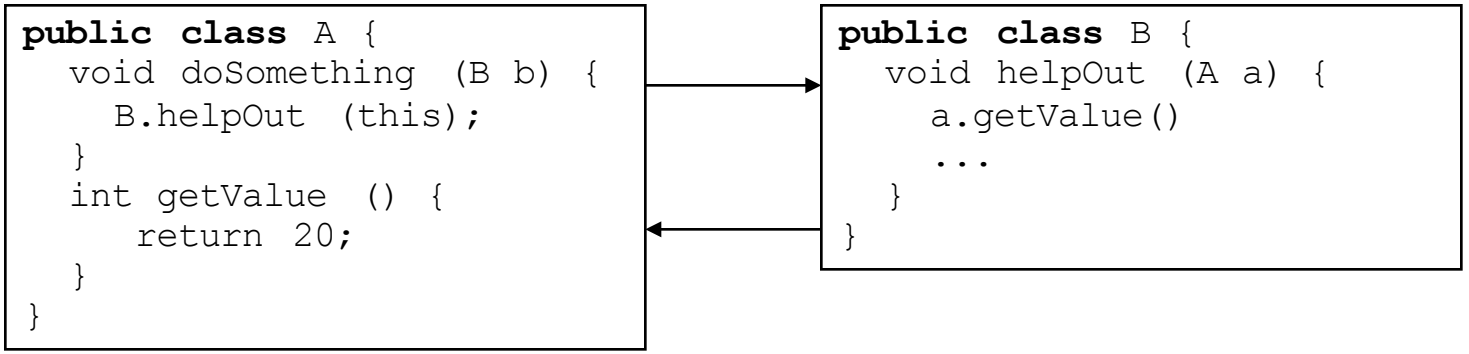
```
public class Agent {
   private Vector helpers = new Vector();

   public void addHelper (WorkInterface s) {
     helpers.addElement (s);
   }
   public void removeHelper (WorkInterface s) {
     helpers.removeElement (s);
   }

   public void requestWork () {
     for (Enumeration en = helpers.elements();
en.hasMoreElements(); ) {
       WorkInterface wi = (WorkInterface)
en.nextElement();
       wi.showHow();
     }
   }
}
```

```
public interface WorkInterface {
  void showHow();
}
```

```
public class Manager implements WorkInterface {
  public void showHow() {
     // Individual work gets done...
  }
}
```

Decoupled Principles

- Avoid "death-embrace" coupling



```
public class A {
  void doSomething (B b) {
    B.helpOut (this);
  }
  int getValue () {
     return 20;
  }
}
```

```
public class B {
  void helpOut (A a) {
    a.getValue()
    ...
  }
}
```

- Instead use interfaces

```
public class A implements IVal {
  void doSomething (IHelp h) {
    h.helpOut (this);
  }
  int getValue () {
     return 20;
  }
}
```

```
public class B implements IHelp {
  void helpOut (IVal v) {
    v.getValue()
    ...
  }
}
```

Questions?