Software Reuse

Building software from reusable components
Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic reuse.
Benefits of Reuse

- **Increased reliability**
  - Components exercised in working systems

- **Reduced process risk**
  - Less uncertainty in development costs

- **Effective use of specialists**
  - Reuse components instead of people

- **Standards compliance**
  - Embed standards in reusable components

- **Accelerated development**
  - Avoid original development and hence speed-up production
Reusable Component Types

- Application system reuse
  - The whole of an application system may be reused on a different machine. Usually referred to as program portability.

- Sub-system reuse
  - Major sub-systems such as a pattern-matching system may be reused.

- Modules or object reuse
  - The reusable component is a collection of functions or procedures.

- Function reuse
  - The reusable component is a single function.
Reuse Practice

- **Application system reuse**
  - Widespread. It is common practice for developers of systems (e.g., Microsoft) to make their products available on several platforms.

- **Sub-system and module reuse**
  - Practiced informally in that individual engineers reuse previous work. Little systematic reuse but increasing reuse awareness.

- **Function reuse**
  - Common in some application domains (e.g., engineering) where domain-specific libraries of reusable functions have been established. Reuse is the principal reason why languages such as FORTRAN are still used.
Four Aspects of Reuse

- Software development with reuse
  - Developing software given a base of reusable components.

- Software development for reuse
  - How to design generic software components for reuse.

- Generator-based reuse
  - Domain-specific reuse through application generation.

- Application system reuse
  - How to write application systems so that they may be readily ported from one platform to another.
Software Development with Reuse

_attempts to maximize the use of existing components.

* These components may have to be adapted in a new application.

* Fewer components need be specified, designed and coded.

* Overall development costs should therefore be reduced.
Further Advantages

- System reliability is increased.
- Overall risk is reduced.
- Effective use can be made of specialists.
- Organizational standards can be embodied in reusable components.
- Software development time can be reduced.
Development with Reuse Process

Design system architecture → Specify components → Search for reusable components → Incorporate discovered components
Requirements for Reuse

- It must be possible to find appropriate reusable components in a component data base.
- Component re-users must be able to understand components and must have confidence that they will meet their needs.
- The components must have associated documentation discussing HOW they can be reused and the potential costs of reuse.
Reuse-driven Development

Rather than reuse being considered after the software has been specified, the specification takes into account the existence of reusable components.

This approach is commonplace in the design of electronic, electrical and mechanical systems.

If adopted for software, should significantly increase the proportion of components reused.
Re-use-driven Development

Outline system requirements → Search for reusable components → Modify requirements according to discovered components → Specify system components based on reusable components → Search for reusable components → Architectural design
Reuse Problems

- Difficult to quantify costs and benefits of development with reuse.
- CASE tool sets do not support development with reuse. They cannot be integrated with a component library systems.
- Some software engineers prefer to rewrite rather than reuse components.
- Current techniques for component classification, cataloging and retrieval are immature. The cost of finding suitable components is high.
Software Development for Reuse

- Software components are not automatically reusable. They must be modified to make them usable across a range of applications.
- Software development for reuse is a development process which takes existing components and aims to generalize and document them for reuse across a range of applications.
Development for Reuse

- The development cost of reusable components is higher than the cost of specific equivalents.
- This extra reusability enhancement cost should be an organization rather than a project cost.
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents.
Reusability Enhancement

- **Name generalization**
  - Names in a component may be modified so that they are not a direct reflection of a specific application entity.

- **Operation generalization**
  - Operations may be added to provide extra functionality and application specific operations may be removed.

- **Exception generalization**
  - Application specific exceptions are removed and exception management added to increase the robustness of the component.

- **Component certification**
  - Component is certified as reusable.
Domain-specific Reuse

- Components can mostly be reused in the application domain for which they were originally developed as they reflect domain concepts and relationships.
- Domain analysis is concerned with studying domains to discover their elementary characteristics.
- With this knowledge, components can be generalized for reuse in that domain.
Domain-specific Reuse

- Reusable components should encapsulate a domain abstraction.
- The abstraction must be parameterized to allow for instantiation in different systems with specific requirements.
The Abstract Data Structures Domain

- Well-understood application domain.
- Important as a foundation for many types of software system.
- The requirements for reusable abstract data structures have been published by several authors (e.g., Booch).
- A classification scheme for such components has been invented.
ADS generalization

- Involves adding operations to a component to ensure domain coverage.

- Operations required include:
  - Access operations
  - Constructor operations
  - I/O operations
  - Comparison operations
  - Iterator operations, if the component is a collection of components
Model of a Reusable ADS

- Exported type names
- Access operations
- Constructor operations
- Abstract data structure
- I/O operations
- Iterator operations
- Comparison operations
Reuse Guidelines

- Implement data structures as generic packages.
- Provide operations to create and assign instances.
- Provide a mechanism to indicate whether or not operations have been successful.
Reuse Guidelines

- Implement operations which can fail as procedures and return an error indicator as an out parameter.
- Provide an equality operation to compare structures.
- Provide an iterator which allows each element in a collection to be visited efficiently without modification to that element.
Reusable Component Example

- Linked list of elements where each element maintains a pointer to the next element in the list.
- Commonly implemented in application systems but application-specific components are rarely generic as their operations reflect specific application needs.
- Linked list operations are usually independent of the type of element in the list.
template <class elem> class List
{
public:
    List();       // Automatic constructor
~List();       // Automatic destructor

    // Basic list operations
    elem Head (error_indic &Err) ;
    int Length ( ) ;
    List <elem> Tail (error_indic &Err) ;

    // Equality operations
    friend List <elem> operator == (List <elem> L1, List <elem> L2) ;
    friend List <elem> Equivalent (List <elem> L1, List <elem> L2) ;

    // Constructor operations for linked list
    void Append (elem E, error_indic &Err) ;
    void Add (elem E, error_indic &Err) ;
    void Add_before (elem E, error_indic &Err) ;
    void Add_after (elem E, error_indic &Err) ;
    void Replace (elem E, error_indic &Err) ;
    void Clear (error_indic &Err ) ;
    void Prune (error_indic &Err ) ;
    void Prune_to (elem E, error_indic &Err ) ;
    void Prune_from (elem E, error_indic &Err ) ;
    void Remove (elem E, error_indic &Err ) ;
    void Remove_before (elem E, error_indic &Err ) ;
    void Remove_after (elem E, error_indic &Err ) ;
C++

Linked List

// I/O functions
void Print(error_indic &Err) ;
void Write_list(char* filename, error_indic &Err) ;
void Read_list(char* filename, error_indic &Err) ;
private:
    typedef struct Linkedlist {
        elem     val;
        LinkedList*     next;
    } LinkedList;

    LinkedList* Listhead ; // (Internal) Pointer to start of list
};

template <class elem> class Iterator {
    friend class List <elem> ;
public:
    Iterator () ;
~Iterator () ;
    void Create (List <elem> L, error_indic &Err) ;
    void Go_next (error_indic &Err) ;
    elem Eval (error_indic &Err) ;
    boolean At_end () ;
private:
    LinkedList* iter ;
};
Language-dependent Reuse

- Reuse guidelines for domain abstractions are independent of the implementation language.
- Some reuse guidelines may be language independent.
  - In C++, always pass the array size as a parameter to reusable components which operate on arrays.
Component Adaptation

- Extra functionality may have to be added to a component. When this has been added, the new component may be made available for reuse.
- Unneeded functionality may be removed from a component to improve its performance or reduce its space requirements.
- The implementation of some component operations may have to be modified. This suggests that the original generalization decisions may be incorrect.
Reuse and Inheritance

- Objects are inherently reusable because they package state and associated operations. They can be self-contained with no external dependencies.
- Inheritance means that a class inherits attributes and operations from a super-class. Essentially, these are being reused.
- Multiple inheritance allows several objects to act as a base class so attributes and operations from several sources are reused.
A Class Lattice

Attributes and operations reused by inheritance down the hierarchy
Problems with Inheritance

- As component classes are developed, the inheritance lattice becomes very complex with duplications across the lattice.
- To understand a component, many classes in the hierarchy may have to be examined and understood.
- In many cases, it may be impossible to avoid inheriting unneeded functionality.
Generator-based Reuse

- Program generators involve the reuse of standard patterns and algorithms.
- These are embedded in the generator and parameterized by user commands. A program is then automatically generated.
- Compilers are program generators where the reusable patterns are object code fragments corresponding to high-level language commands.
Reuse Through Program Generation

Application description → Program generator → Generated program

Application domain knowledge → Database
Types of Program Generator

- Types of program generator
  - Application generators for business data processing.
  - Parser and lexical analyzer generators for language processing.
  - Code generators in CASE tools.

- Generator-based reuse is very cost-effective but its applicability is limited to a relatively small number of application domains.
Application System Portability

- Portability is a special case of reuse where an entire application is reused on a different platform.
- The portability of a program is a measure of the amount of work required to make that program work in a new environment.
Application Program Interfaces (APIs)
Portability Dependencies

- Operating system dependencies
  - Dependencies on operating system characteristics.

- Run-time system problems
  - Dependencies on a particular run-time support system.

- Library problems
  - Dependencies on a specific set of libraries.
Development for Portability

- Isolate parts of the system which are dependent on the external program interfaces.
- Define a portability interface to hide operating system characteristics.
- To port the program, only the code behind the portability interface need be rewritten.
A Portability Interface

Application system

Portability interface

Data references

Operating system and I/O calls
Operating System Dependencies

- The program relies on the use of specific operating system calls such as facilities to support process management.
- The program depends on a specific file system organization supported by the operating system.
Portability Interface Implementation

Application

Abstract data type interface

OR

Unix filestore

Database system
Standards

- Standards are an agreement across the community which reduces the amount of variability in software systems.
- The development of standards in the 1980s means that program portability is now much simpler than before.
- In principle, as standards are further developed, heterogeneous systems may be developed where parts of a program may run on completely different machines.
Existing Standards

- **Programming language standards**
  - Ada, Pascal, C, C++, FORTRAN.

- **Operating system standards**
  - UNIX, MS Windows (de-facto standard).

- **Networking standards**
  - TCP/IP protocols, X400, X500, Sun NFS, OSI layered model, HTML, WWW.

- **Window system standards**
  - X-windows, Motif tool kit.
Component-based Development

- Component-based software engineering (CBSE) is an approach to software development that relies on reuse.
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.
- Components are more abstract than object classes and can be considered to be stand-alone service providers.
Components

Components provide a service without regard to where the component is executing or its programming language.

- A component is an independent executable entity that can be made up of one or more executable objects.
- The component interface is published and all interactions are through the published interface.

Components can range in size from simple functions to entire application systems.
Component Interfaces

Requires interface  
Component  
Provides interface
Component Interfaces

- Provides interface
  - Defines the services that are provided by the component to other components

- Requires interface
  - Defines the services that specifies what services must be made available for the component to execute as specified
Printing Services Component

Requires interface

PrintService

Provides interface

GetPDfile

PrinterInt

Print

GetQueue

Remove

Transfer

Register

Unregister
Component Abstractions

- **Functional abstraction**
  - The component implements a single function such as a mathematical function

- **Casual groupings**
  - The component is a collection of loosely related entities that might be data declarations, functions, etc.

- **Data abstractions**
  - The component represents a data abstraction or class in an object-oriented language

- **Cluster abstractions**
  - The component is a group of related classes that work together

- **System abstraction**
  - The component is an entire self-contained system
CBSE Processes

- Component-based development can be integrated into a standard software process by incorporating a reuse activity in the process.

- However, in reuse-driven development, the system requirements are modified to reflect the components that are available.

- CBSE usually involves a prototyping or an incremental development process with components being ‘glued together’ using a scripting language.
Component incompatibilities may mean that cost and schedule savings are less than expected.
Finding and understanding components.
Managing evolution as requirements change in situations where it may be impossible to change the system components.
Application Frameworks

Framework are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them.

The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.

Frameworks are moderately large entities that can be reused.
Framework Classes

- **System infrastructure frameworks**
  - Support the development of system infrastructures such as communications, user interfaces and compilers

- **Middleware integration frameworks**
  - Standards and classes that support component communication and information exchange

- **Enterprise application frameworks**
  - Support the development of specific types of application such as telecommunications or financial systems
Extending Frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system.

- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework
  - Adding methods that are called in response to events that are recognised by the framework

- Problem with frameworks is their complexity and the time it takes to use them effectively.
Model-view Controller

- System infrastructure framework for GUI design
- Allows for multiple presentations of an object and separate interactions with these presentations
- MVC framework involves the instantiation of a number of patterns (discussed later)
Model-view Controller

- Model state
- Model methods
- View state
- View methods
- Controller state
- Controller methods

User inputs → Controller methods → Model edits

Model queries and updates → Model state → Model edits

view modification messages → Controller state → View methods
COTS Product Reuse

- COTS - Commercial Off-The-Shelf systems.
- COTS systems are usually complete application systems that offer an API (Application Programming Interface).
- Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as E-commerce systems.
COTS System Integration Problems

- Lack of control over functionality and performance
  - COTS systems may be less effective than they appear

- Problems with COTS system inter-operability
  - Different COTS systems may make different assumptions that means integration is difficult

- No control over system evolution
  - COTS vendors not system users control evolution

- Support from COTS vendors
  - COTS vendors may not offer support over the lifetime of the product
OOD Patterns Topics

- Terminology and Motivation
- Reusable OO Design Patterns:
  - Adapter
  - Facade
  - Iterator
  - Composite
  - Template
  - Abstract Factory
  - Observer
  - Master-Slave
Design Patterns

- Good designers know not to solve every problem from first principles. They reuse solutions.
- Practitioners do not do a good job of recording experience in software design for others to use.
A Design Pattern systematically names, explains, and evaluates an important and recurring design.

We describe a set of well-engineered design patterns that practitioners can apply when crafting their applications.
Becoming a Master Designer

☞ **First, One Must Learn the Rules:**
  - Algorithms
  - Data Structures
  - Languages

☞ **Later, One Must Learn the Principles:**
  - Structured Programming
  - Modular Programming
  - OO Programming
Becoming a Master Designer (Cont’d)

Finally, One Must Study the Designs of Other Masters:

- Design patterns must be understood, memorized, and applied.
- There are thousands of existing design patterns.
The Adapter Pattern

**Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

**Motivation:** When we want to reuse classes in an application that expects classes with a different interface, we do not want (and often cannot) to change the reusable classes to suit our application.
Example of the Adapter Pattern

```
Editor -> Shape
  BoundingBox()
  CreateManipulator()

LineShape
  BoundingBox()
  CreateManipulator()

TextShape
  BoundingBox()
  CreateManipulator()

TextView
  GetExtent()

return text -> GetExtent()
return new Text Manipulator
```
Structure of the Adapter Pattern
Using Multiple Inheritance

Client

Target
Request()

Adaptee
SpecificRequest()

Adapter
Request()

SpecificRequest()

(implementation)
Structure of the Adapter Pattern
Using Object Composition

Client

Target

Request()

Adaptee

SpecificRequest()

Adapter

Request()

SpecificRequest()

adaptee
Participants of the Adapter Pattern

- **Target:** Defines the application-specific interface that clients use.
- **Client:** Collaborates with objects conforming to the target interface.
- **Adaptee:** Defines an existing interface that needs adapting.
- **Adapter:** Adapts the interface of the adaptee to the target interface.
The Facade Pattern (Intent)

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
The Facade Pattern (Motivation)

- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- Use a facade object to provide a single, simplified interface to the more general facilities of a subsystem.
Example of the Facade Pattern

- Compiler
  - Compile()
  - CodeGenerator
    - RISCG
    - StackMachineCG
  - Parser
  - ProgNodeBuilder
    - ProgNode
      - Statement Node
      - Expression Node
      - Variable Node
Structure of the Facade Pattern

Client Classes

Facade

Subsystem Classes
Participants of the Facade Pattern

**Facade:**
- Knows which subsystem classes are responsible for a request.
- Delegates client requests to appropriate subsystem objects.

**Subsystem Classes:**
- Implement subsystem functionality.
- Handle work assigned by the facade object.
- Have no knowledge of the facade; that is, they keep no references to it.
The Iterator Pattern (Intent)

☞ Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

☞ Move the responsibility for access and traversal from the aggregate object to the iterator object.
The Iterator Pattern (Motivation)

- One might want to traverse an aggregate object in different ways.
- One might want to have more than one traversal pending on the same aggregate object.
- Not all types of traversals can be anticipated a priori.
- One should not bloat the interface of the aggregate object with all these traversals.
Example of the Iterator Pattern

```
List
Count()
Append(Element)
Remove(Element)
...

ListIterator
First()
Next()
IsDone()
CurrentItem()

index
```

list
Structure of the Iterator Pattern

Aggregate
CreateIterator()

ConcreteAggregate
CreateIterator()

Iterator
First()
Next()
IsDone()
CurrentItem()

ConcreteIterator
return new ConcreteIterator(this)
Participants of the Iterator Pattern

- **Iterator:** Defines an interface for accessing and traversing elements.

- **Concrete Iterator:** Implements an iterator interface and keeps track of the current position in the traversal of the aggregate.

- **Aggregate:** Defines an interface for creating an iterator object.

- **Concrete Aggregate:** Implements the iterator creation interface to return an instance of the proper concrete iterator.
The Composite Pattern (Intent)

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
The Composite Pattern
(Motivation)

If the composite pattern is not used, client code must treat primitive and container classes differently, making the application more complex than is necessary.
Example of the Composite Pattern

```
Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)

forall g in graphics
  g.Draw()
```

Line
Draw()

Rect.
Draw()

Text
Draw()

Picture
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)

forall g in graphics
  g.Draw()
Structure of the Composite Pattern

Client

Component

- Operation()
- Add(Component)
- Remove(Component)
- GetChild(int)

Leaf

Operation()

Composite

- Operation()
- Add(Component)
- Remove(Component)
- GetChild(int)

forall g in children
  g.Operation()
Participants of Composite Pattern

Component:
- Declares the interface for objects in the composition.
- Implements default behavior for the interface common to all classes.
- Declares an interface for accessing and managing its child components.
- Defines an interface for accessing a component’s parent in the recursive structure (optional).
Participants of Composite Pattern (Cont’d)

- **Leaf:**
  - Represents leaf objects in the composition. A leaf has no children.
  - Defines behavior for primitive objects in the composition.

- **Composite:**
  - Defines behavior for components having children.
  - Stores child components.
  - Implements child-related operations in the component interface.
Participants of Composite Pattern (Cont’d)

🌿 Client:

- Manipulates objects in the composition through the component interface.
The Template Pattern (Intent)

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

The Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.
The Template Pattern
(Motivation)

By defining some of the steps of an algorithm, using abstract operations, the template method fixes their ordering.
Structure of the Template Pattern

```
AbstractClass
  TemplateMethod()
  PrimitiveOp1()
  PrimitiveOp2()

ConcreteClass
  PrimitiveOp1()
  PrimitiveOp2()

  ... PrimitiveOp1()
  ... PrimitiveOp2()
```

...
Structure of the Template Pattern

☞ Abstract Class:

- Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in Abstract Class or those of other objects.
Structure of the Template Pattern (Cont’d)

Concrete Class: Implements the primitive operations to carry out subclass-specific steps to the algorithm.
The Abstract Factory Pattern
(Intent)

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
The Abstract Factory Pattern (Behavior)

Sometimes we have systems that support different representations depending on external factors.

There is an *Abstract Factory* that provides an interface for the client. In this way the client can obtain a specific object through this abstract interface.
Example of the Abstract Factory Pattern

WidgetFactory
- CreateScrollBar()
- Create Window()

MotifWidgetFactory
- CreateScrollBar()
- Create Window()

PMWidgetFactory
- CreateScrollBar()
- Create Window()

Client

Window

ScrollBar

PMScrollBar

MotifScrollBar

PMWindow

MotifWindow
Structure of the Abstract Factory Pattern

AbstractFactory

CreateProductA()
CreateProductB()

ConcreteFactory1
CreateProductA()
CreateProductB()

ConcreteFactory2
CreateProductA()
CreateProductB()

AbstractProductA

ProductA1

ProductA2

AbstractProductB

ProductB1

ProductB2

Client
Participants of the Abstract Factory Pattern

- **Abstract Factory:**
  - Declares an interface for operations that create abstract product objects.

- **Concrete Factory:**
  - Implements the operations to create concrete product objects.
Participants of the Abstract Factory Pattern (Cont’d)

☞ **Abstract Product:**
  - Declares an interface for a type of product object.

☞ **Concrete Product:**
  - Defines a product object to be declared by the corresponding concrete factory. (Implements the Abstract Product interface).

☞ **Client:**
  - Uses only interfaces declared by Abstract Factory and Abstract Product classes.
The Observer Pattern (Intent)

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
The Observer Pattern
(Motivation)

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
Example of the Observer Pattern

requests, modifications

change notification

\[a = 50\%\]
\[b = 30\%\]
\[c = 20\%\]
Structure of the Observer Pattern

Subject
- Attach(Observer)
- Detach(Observer)
- Notify()

Observer
- Update()

ConcreteSubject
- GetState()
- SetState()
- subjectState

ConcreteObserver
- observerState
- Update()
Structure of the Observer Pattern

- The key objects in this pattern are **subject** and **observer**.
  - A subject may have any number of dependent observers.
  - All observers are notified whenever the subject undergoes a change in state.
Participants of the Observer Pattern

☞ **Subject:**
  - Knows its numerous observers.
  - Provides an interface for attaching and detaching observer objects.
  - Sends a notification to its observers when its state changes.

☞ **Observer:**
  - Defines an updating interface for concrete observers.
Participants of the Observer Pattern (Cont’d)

Concrete Subject:
- Stores state of interest to concrete observers.

Concrete Observer:
- Maintains a reference to a concrete subject object.
- Stores state that should stay consistent with the subject's.
- Implements the updating interface.
The Master-Slave Pattern (Intent)

- Handles the computation of replicated services within a software system to achieve fault tolerance and robustness.
- Independent components providing the same service (slaves) are separated from a component (master) responsible for invoking them and for selecting a particular result from the results returned by the slaves.
The Master-Slave Pattern
(Motivation)

- Fault tolerance is a critical factor in many systems.
- Replication of services and delegation of the same task to several independent suppliers is a common strategy to handle such cases.
Example of the M/S Pattern

NuclearPP
acceptableRL()

Voter
RadLevel()

return max(
slave1->RadLevel(),
slave2->RadLevel(),
slave3->RadLevel())

Slave1
RadLevel()

Slave2
RadLevel()

Slave3
RadLevel()
Structure of the M/S Pattern

Client
  ------------------> Master
    Compute()        service()        forward request
                        request service
                          
  forward request
                          
Slave1
  ------------------> ServiceImp1()

Slave2
  ------------------> ServiceImp1()

Slave3
  ------------------> ServiceImp1()
Participants of the M/S Pattern

**Slave:**
- Implements a service.

**Master:**
- Organizes the invocation of replicated services.
- Decides which of the results returned by its slaves is to be passed to its clients.

**Client:**
- Requires a certain service in order to solve its own task.