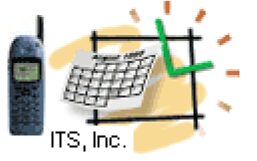
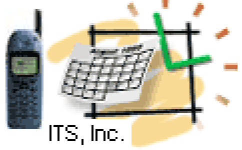


Design Patterns

- Design patterns are known solutions for common problems. Design patterns give us a system of names and ideas for common problems.

- **What are the major description parts?**



Design Patterns Descriptions

- Design Patterns consist of the following parts:
- - Problem Statement
- - Solution
- - Impact
- ---
- There are several Levels and Types of the Design Patterns.

What Levels and Types do you know?

Design Patterns Levels and Types

- There are different types and levels of design patterns. For example, the MVC is the **architectural level** of design pattern while the rest of the patterns from the list above are **component level** design patterns.

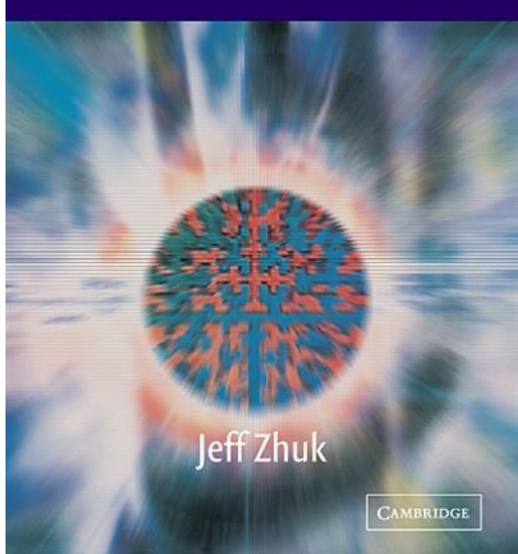
- The basic **types are Behavior, Creational, Structural, and System** design patterns. Names are extremely important in design patterns; they should be clear and descriptive.

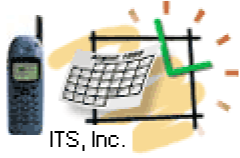
- **<u>More types: Enterprise and SOA Design Patterns</u>**

Christopher Alexander – The first book on Design Patterns

Classics: "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (GOF)

Among other good books: "Integration-Ready Architecture and Design or Software … and Knowledge Engineering"

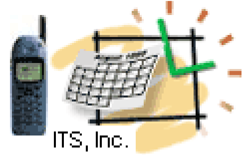




Here is an example of creating a new Design Pattern

- **What:** **Application development or even modification require longer and longer projects**
- **Why:** **Growing applications become more complex and rigid; *too firm and inflexible in spite of the name – Software***

Special efforts are needed



Industry Lessons Learned
Design Patterns
Business-Driven Architecture

- *How can technology be designed to **remain in alignment** with changing business goals and requirements?*

Technology

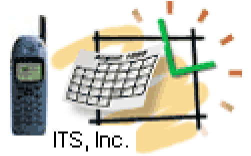
Technology

Business

Business

Good Alignment

Bad Alignment
Duplications
Code branches
Maintenance Cost



Business-Driven Architecture

- **Solution**
- Business and architecture analysis is conducted as collaborative efforts on a *regular basis*
- **Impact**
- To keep technology in alignment with the business that is changing over time, it will require a **commitment in time and cost to govern**

Design Pattern - MVC

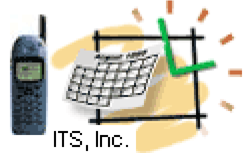
ITS, Inc.

Controller

Model

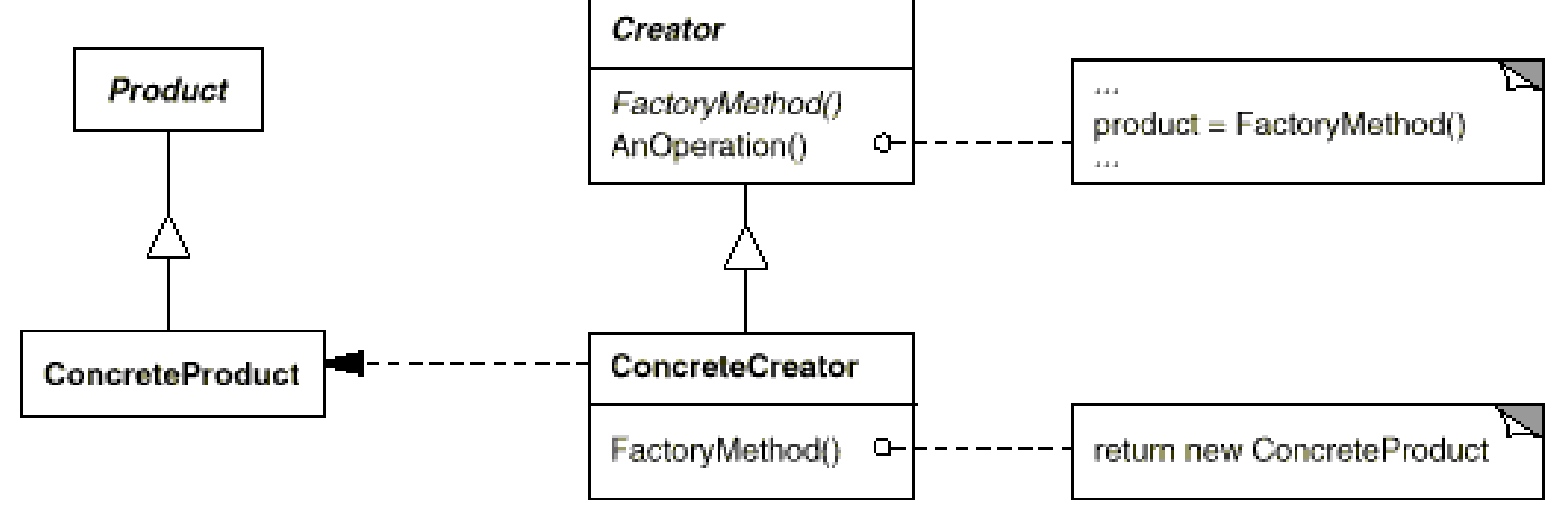
View

- MVC (Model – View – Controller) is well known pattern
- Name – MVC
- **Problem** – Complex object involves user interface and data. Need to simplify structure
- **Solution** – Data in one part (Model), user View in another part (View), interaction logic in a third part (Controller)
  - Model maintains state. Notifies view of changes in state.
  - Controller uses state information (in Model?) and user request to determine how to handle request, tells view what to display
  - View must correctly display the state of the Model
- **Consequences**
  - Allows "plug in" modules – eg. swap out Model to allow different ways of holding data
  - Requires separate engineering of the three parts, communication between them through interfaces



Factory Method

- **Problem** – Need to create a family of similar but different type objects that are used in standard ways.
- **Solution** – Creator class has a "getter" method which instantiate the correct subclass, i.e. ConcreteProduct, Subclass is used through generic interface, i.e. Product
- **Impact** – Extra time for analysis and modeling



Factory Method & Servlet Best Practices

ITS, Inc.

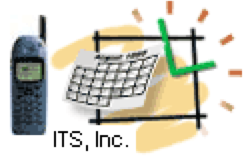
New services can be added run time as new JSPs/ASPs or Java™/.NET classes

```
//serviceName and serviceDetails are to be populated
// by servlet doPost() , doGet() or service() methods

String serviceName = request.getParameter("service");
Hashtable serviceDetails = getServiceDetails();

Service service =          // known or new service
(Service) Class.forName(serviceName).newInstance();

String content = service.run(serviceDetails);
response.setContentType("text/html");  // "application/xsl" and etc.
response.getWriter().println(content);
```

XML based Service API allows us to describe any existing and future service

```
<ServiceRequest service="Mail" action="get">
  <Param><paramName1=…></Param>
</ServiceRequest>
```

We can find both Dispatcher and Factory patterns in this example. This approach makes it possible to create a unified API for client – server communications. Any service (including new, unknown design time services) can be requested by a client without code change.



Design Pattern
Canonical Data Model

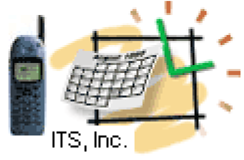
- *How can services be designed to avoid data model transformation?*

- **Problem**
- Services with disparate models for similar data impose transformation requirements that increase development effort, design complexity, and runtime performance overhead.

Canonical Data Model

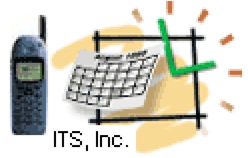
- **Solution**
- Data models for common information sets are standardized across service contracts within an inventory boundary.

- **Application**
- Design standards are applied to schemas used by service contracts as part of a formal design process.



Canonical Data Model

- **Principles**
- Standardized Service Contract
- **Architecture**
- Inventory, Service

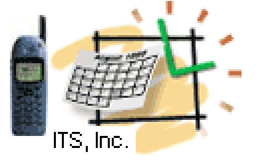


Design Pattern
Canonical Protocol

- *How can services be designed to avoid protocol bridging?*

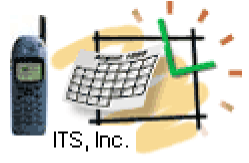
- **Problem**

- Services that support different communication technologies compromise interoperability, limit the quantity of potential consumers, and introduce the need for undesirable protocol bridging measures.



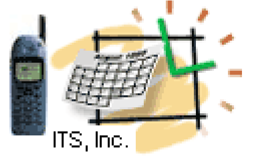
Canonical Protocol

- **Solution**
- The architecture establishes a single communications technology as the sole or primary medium by which services can interact.
- **Application**
- The communication protocols (including protocol versions) used within a service inventory boundary are standardized for all services.



Design Pattern Concurrent Contracts

- *How can a service facilitate multi-consumer coupling requirements and abstraction concerns at the same time?*
- **Problem**
- A service’s contract may not be suitable or applicable for all of the service’s potential consumers.



Concurrent Contracts

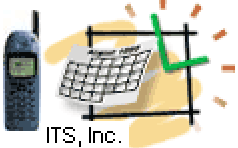
- **Solution**
- Multiple contracts can be created for a single service, each targeted at a specific type of consumer.

- **Application**
- This pattern is ideally applied together with the Service Façade pattern to support new contracts as required.

Singleton Design Pattern

- **Problem** – need to be sure there is at most one object of a given class in the system at one time
- **Solution**
  - Hide the class constructor
  - Provide a method in the class to obtain the instance
  - Let class manage the single instance

```
public class Singleton{
    private static Singleton instance;
    private Singleton(){} // private constructor!
    public Singleton getInstance(){
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```



Provider Design Pattern

- **Context**

Separate implementations of the API from the API itself

Problem

We needed a flexible design and at the same time easily extensible

Solution

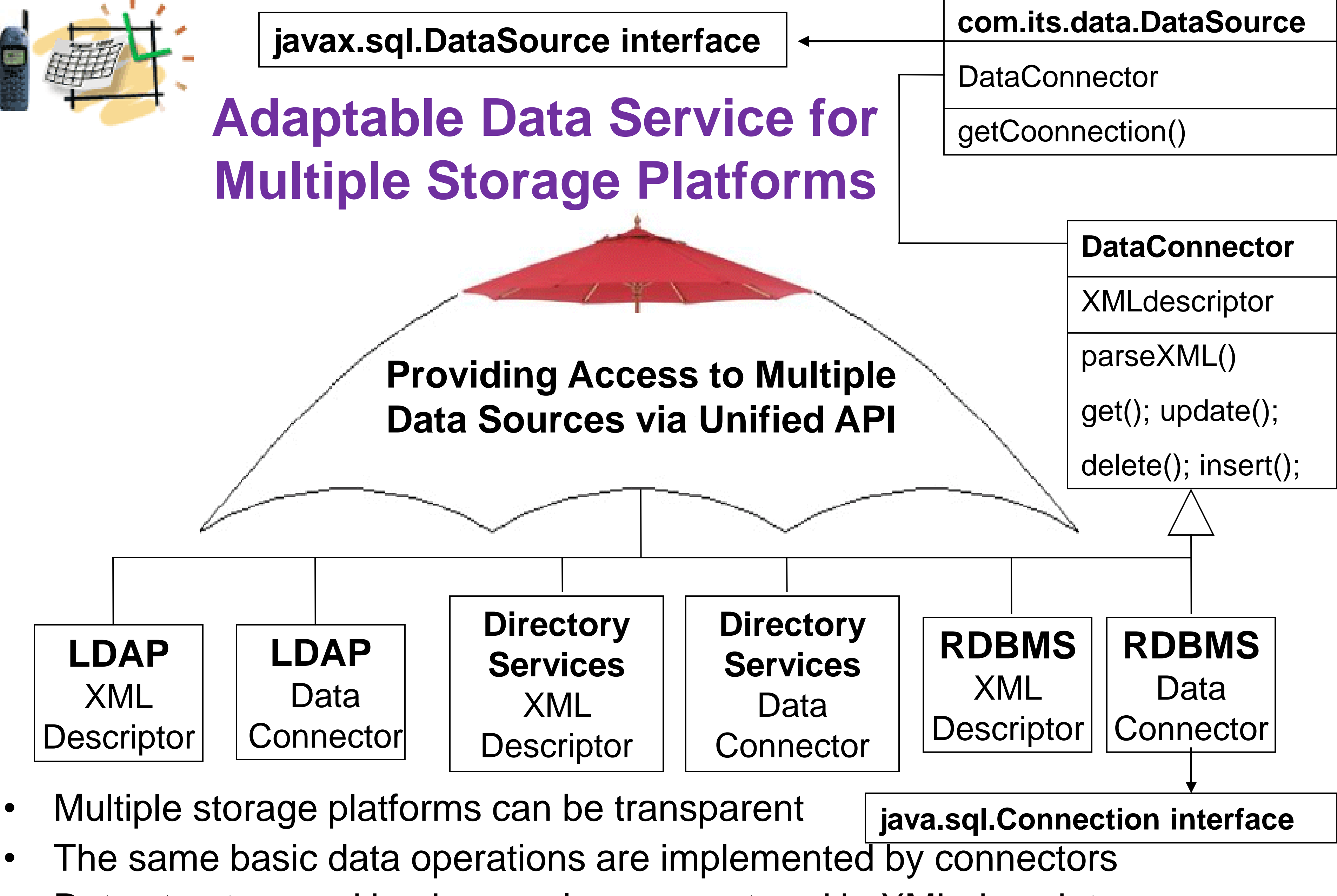
A provider implementation derives from an abstract base class, which is used to define a contract for a particular feature.

For example, to create a provider for multiple storage platforms, you create the feature base class **RDBMSProvider** that derives from a common **StorageProvider** base class that forces the implementation of required methods and properties common to all providers.

Then you create the **DB2Provider**, **OracleProvider**, **MSSQLProvider**, etc. classes that derived from the **RDBMSProvider.**

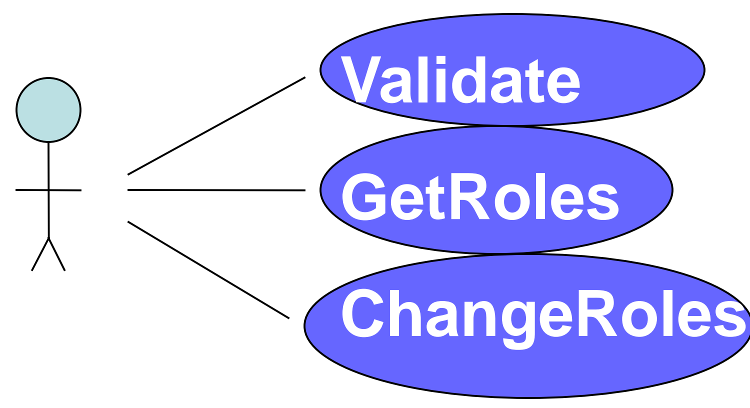
In a similar manner you create the **DirectoryStorageProvider** derived from the **StorageProvider** with its subclasses **ActiveDirectoryProvider, LDAPProvider,** and etc.

Adaptable Data Service for Multiple Storage Platforms



- Multiple storage platforms can be transparent
- The same basic data operations are implemented by connectors
- Data structure and business rules are captured in XML descriptors
- **Design Patterns: Model, Adapter, Provider**

Authentication Service
Delegation, Façade and Provider Design Patterns



1. Delegation: application-specific rules are in a configuration file
2. Façade: a single interface for all applications regardless of data source
3. Provider: Works with multiple datasource providers

   Active Directory, LDAP and RDBMS

Layered: separated Utility and Data Access Layers

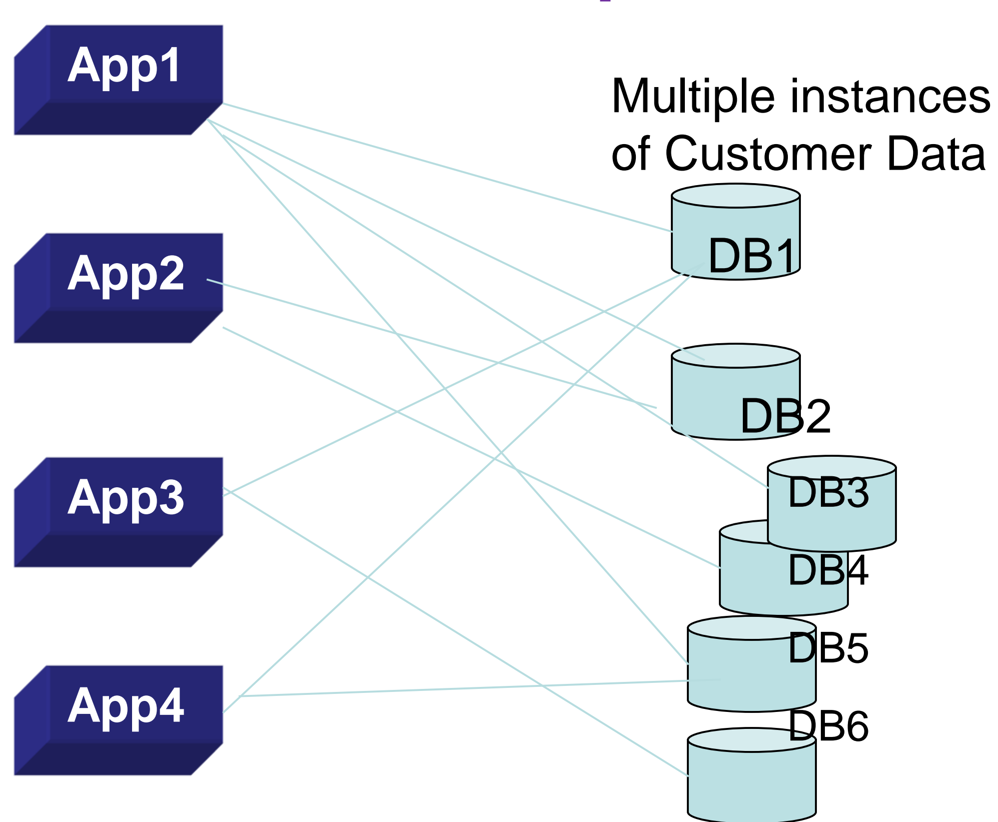
Standard-based: Web Service and Messaging Service Standard Interfaces

Secure: Protected by HTTPS and Valid Certificates

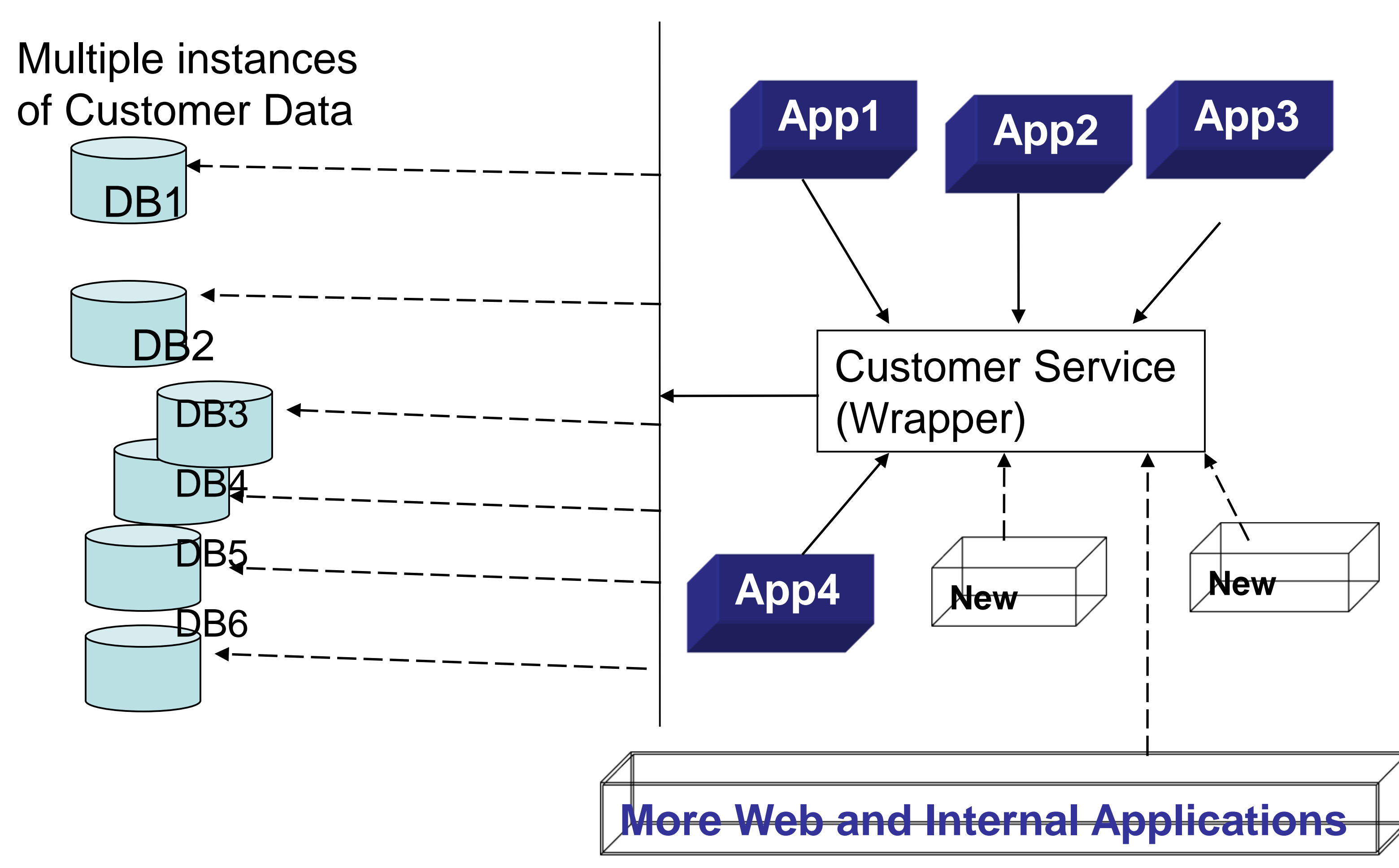
Authentication Service
Provider, Façade and Model Design Patterns

```
// read config & build application map on initiation
AppsArray[] apps = serviceConfig.getApplicationArray();
// apps maps each application to its data source(s)
-----------------------------------------------------

// getRoles(appName, userName);
AuthServiceDao dao = apps.getService(appName);

// dao is one of types: LdapDao, AdDao or DbDao
String roles = dao.getRoles(userName);
```

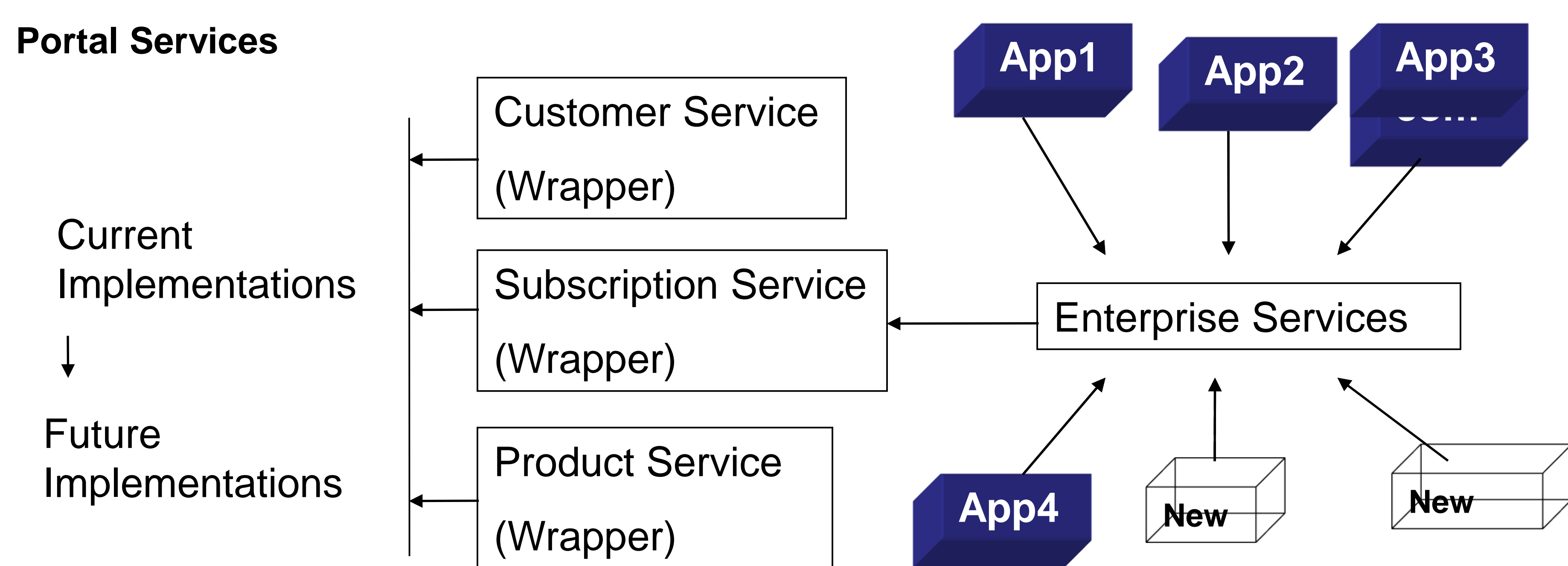
How Façade Design Pattern can help us to Improve Implementations of Internet Services, Increase Reuse and Remove Duplications



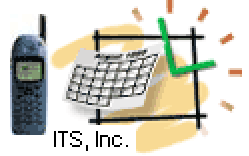
From Project-based code to Enterprise Services using Façade Design Pattern



Enterprise Services will Shield Applications and Enable Changes from current to better Implementations

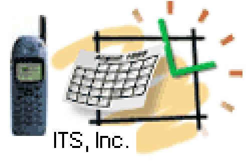


Publish and promote adaptation of Web Services



Design Pattern
<u>Delegate</u>

- **Problem**
- Business logics is often customized on client requests creating maintenance pain
- **Solution**
- Delegate changeable part of business logic to a special component, like a rules service, and simplify changing this logic.

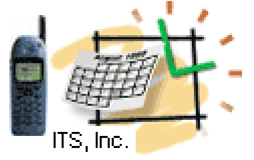


Design Pattern
Agnostic Context

- *How can multi-purpose service logic be positioned as an effective enterprise resource?*

- **Problem**

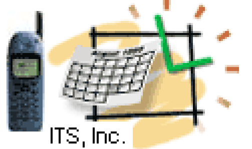
- Multi-purpose logic grouped together with single purpose logic results in programs with little or no reuse potential that introduce waste and redundancy into an enterprise.



Agnostic Context

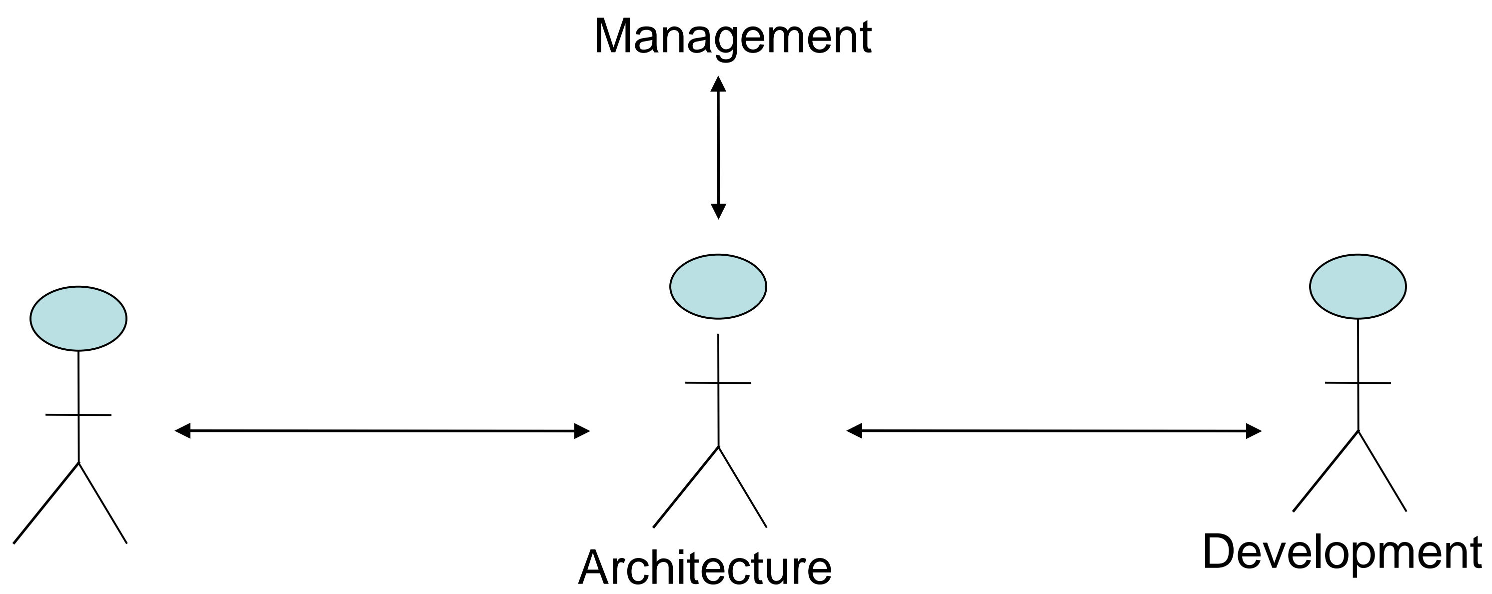
- **Solution**
- Isolate logic that is not specific to one purpose into separate services with distinct agnostic contexts.

- **Application**
- Agnostic service contexts are defined by ***carrying out service-oriented analysis*** and service modeling processes.

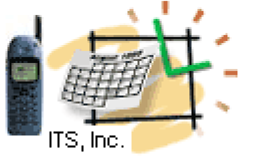


Governance

Connect System and Enterprise Architectures
Connect Business and Technology Architecture
Engage Teams in Collaborative Engineering



Conduct service-oriented analysis to re-think Enterprise Architecture

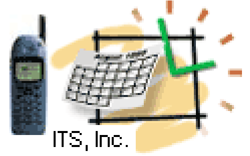


SOA with TOGAF

Learn:

TOGAF Intro

TOGAF ADM Features to Support SOA



Why TOGAF & SOA?

- The Open Group Architecture Framework (TOGAF)
- TOGAF is a mature EA framework
- SOA is an architecture style
- Enterprises struggle to move to SOA
- TOGAF helps to describe EA and steps for SOA

Preliminary phase

Define principles,
Adapt framework

A
Architecture vision

A: Define scope; create vision; obtain approvals

B
Business Architecture

B: Develop a business architecture

C
Information Systems Architecture

C: Develop data and application architecture

D
Technology Architecture

D: Develop a technology architecture

E
Opportunities and Solutions

E: Check point suitability for implementation

F
Migration Planning

F: Prioritize, select major work packages, develop migration plan

G
Implementation Governance

G: Provide architectural oversight of the implementation

H
Architecture Change Management

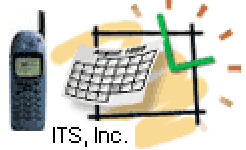
H: Establish for procedures for managing change to new architecture

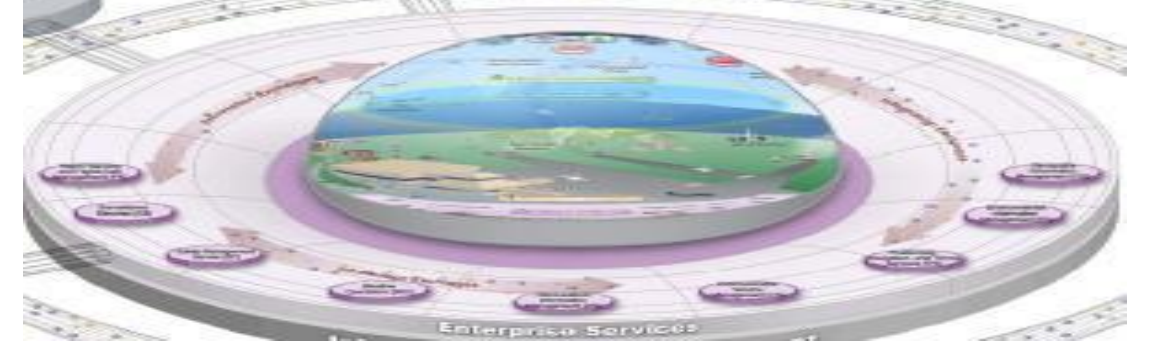
Requirement Managements

ITS, Inc.

Phase A: TOGAF General Views

- Business Architecture views
- Data Architecture views
- Applications Architecture views
- Technology Architecture views





Mapping Business and Technology Views

Business
Data
Service
Infrastructure

Business Architecture/Process View: Workflows & Scenarios

Business Architecture/Product View:
Product Lines, Products, Features
Descriptions and order terms

Service Views:
Business/Utility/Data Services
Descriptions and execution terms

Data Architecture:
Standards, Repositories
Descriptions and Models

Technology Architecture:
Platforms/Servers/Net/Security

Questions?

Please feel free to email or call Jeff:

720-299-4701

Looking for your feedback: what was especially helpful and what else you would like to know, and what are better ways to work together in a collaborative fashion