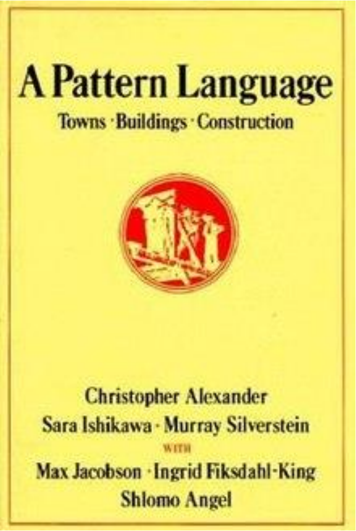
Information systems modeling

Tomasz Kubik

Design Pattern

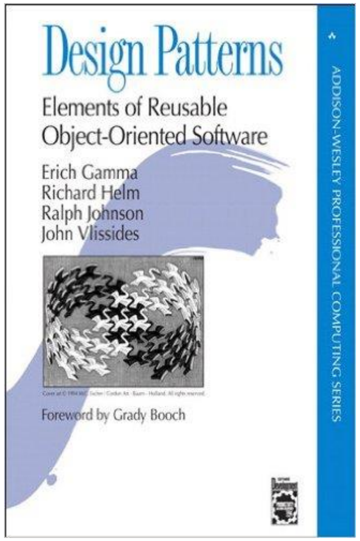
“Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice”

Ch.W. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: *A Pattern Language: Towns, Buildings, Construction*, 1977


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”

E. Gamma, R. Helm, R. Johnson, J. Vlissides (Gang of Four): Design Patterns: Elements of Reusable Object-Oriented Software, 1994


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

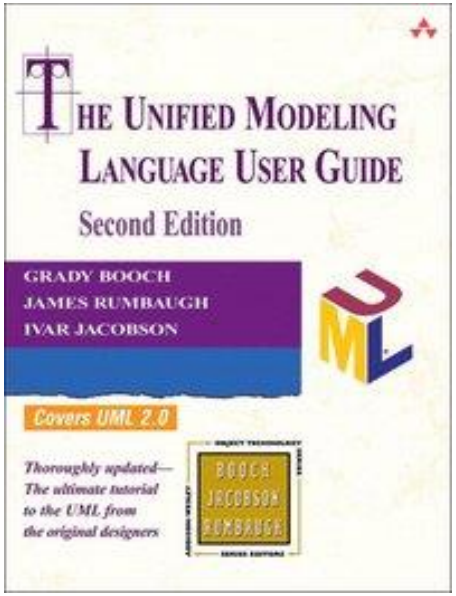
Design patterns can be expressed at various abstraction levels

- OO programming (like GoF patterns)
  - **Structure, Creational, and Behavioral**
- Design and implementation of the multi-tier software
  - **Presentation, Business and Integration**

Design Pattern

“All well-structured systems are full of patterns. **A pattern provides a good solution to a common problem in a given context.** A mechanism is a design pattern that applies to a society of classes; **a framework is typically an architectural pattern that provides an extensible template for applications within a domain**.

You use patterns to specify mechanisms and frameworks that shape the architecture of your system. You make a pattern approachable by clearly identifying the slots, tabs, knobs, and dials that a user of that pattern may adjust to apply the pattern in a particular context.”


THE UNIFIED MODELING LANGUAGE USER GUIDE
Second Edition
GRADY BOOCH
JAMES RUMBAUGH
IVAR JACOBSON
UML
Covers UML 2.0
Thoroughly updated—The ultimate tutorial to the UML from the original designers
BOOCH JACOBSON RUMBAUGH

G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*, 2005

Design Pattern

- re-usable solution to recurring problem, applied and tested
- template that can be adapted according to needs
- relies on the use of OO concepts (see also: *Object-oriented analysis*, *design*, *modeling*: OOA, OOD, OOM) :
  - aggregation
  - inheritance
  - encapsulation
  - interface
  - polymorphism
- can be defined with the use of:
  - class diagram to represent structure
  - sequence diagram to represent behavior

https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm

Essential elements of Design Pattern according to GoF

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The **problem** describes when to apply the pattern. It explains the problem and its context.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
4. The **consequences** are the results and trade-offs of applying the pattern, costs and benefits of applying the pattern.

Generally: **the use of design patterns** supports the creation of **high-quality software in an efficient manner**.

Main goals of GOF patterns

- Creational pattern
  - isolation of the rules for creating objects from the rules that determine how the use of these objects (separation of the code for creating objects from the code that uses them)
- Structural patterns
  - grouping classes and objects into larger structures
    - Class design pattern: use of inheritance and polymorphism to make structures of interfaces and their implementations
    - Object design pattern: describe a way how to combine objects in order to obtain a new functionality, even during program execution
- Behavioral patterns
  - allocating algorithms and obligations to objects, covering the patterns of objects and classes, and communication between objects

GoF Design Patterns classification

	Purpose reflects what a pattern does		
Scope domain over which it applies	**Creational** creation process of	**Structural** composition of	**Behavioral** interaction & responsibility of
Class - relationship between classes & subclasses; - statically defined at run-time	Factory Method	Adapter (class)	Interpreter Template Method
Object - object relationships (what type?) - manipulating at runtime (so what?)	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

GoF Design Patterns classification – examples with description

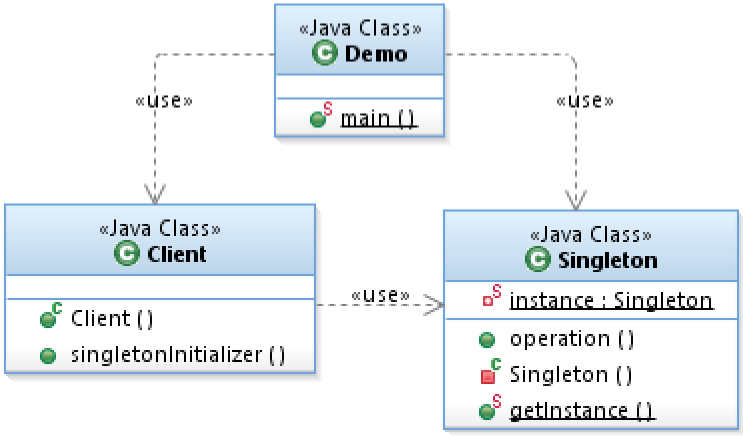
	Purpose		
Scope	**Creational**	**Structural**	**Behavioral**
Class	**Factory Method** defers object creation to sub-classes	**Adapter** applies inheritance to compose classes	**Template Method** uses inheritance to describe flow of control, algorithms
Object	**Abstract Factory** defers object creation to other objects	**Adapter** deals with object assembly	**Iterator** makes use of group of objects working together to carry out a task

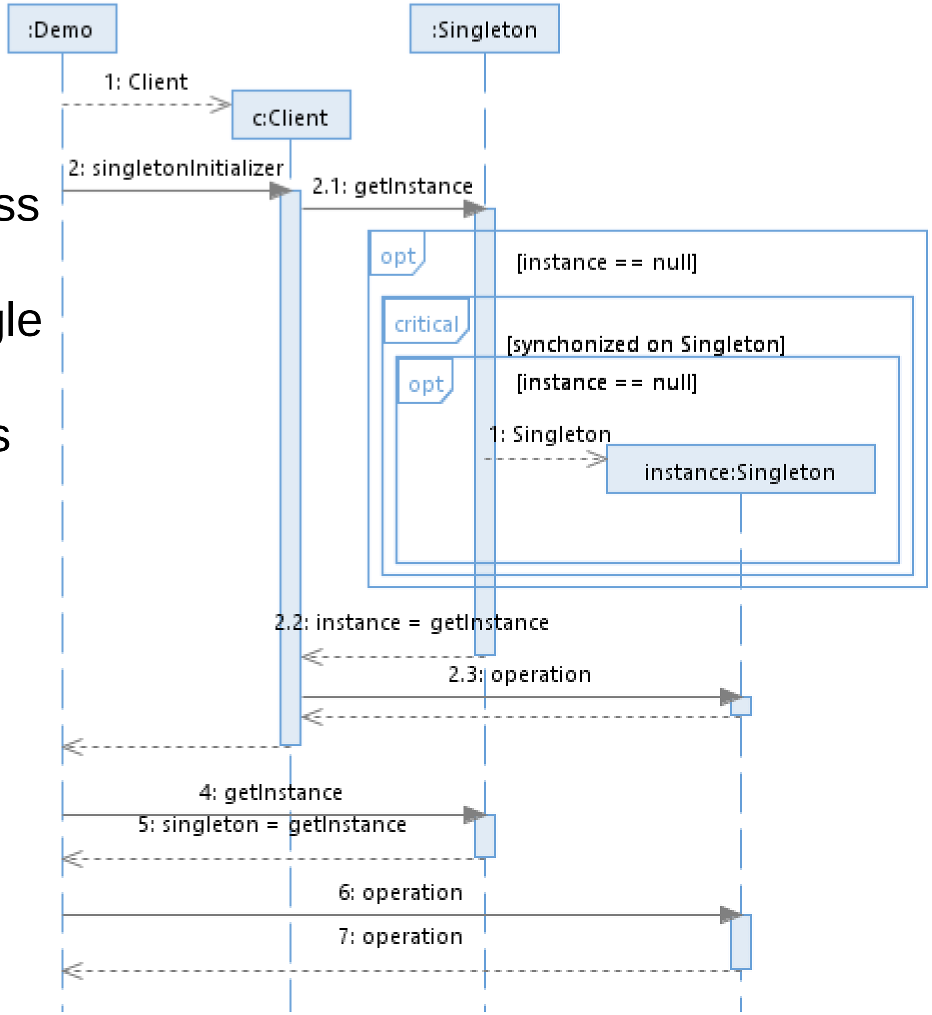
GoF Design Pattern Template

Subject	Explanation
Pattern Name and Classification	obvious
Intent	a brief overview on: purpose of the pattern, its rationale and intent, problems addressed
Also Known As	any other synonyms
Motivation	a scenario illustrating problem and the pattern’s use
Applicability	situations in which pattern can be applied
Structure	a graphical representation (in UML)
Participants	participating classes and/or objects with their responsibilities
Collaborations	how these participants collaborate to carry out their responsibilities
Consequences	the results of application, benefits, trade-offs, liabilities
Implementation	pitfalls, hints, techniques, language-specific issues
Sample Code	obvious
Known Uses	real life examples from at least two different domains.
Related Patterns	connections with other patterns

Singleton

- used when exactly one instance of a class is required
- applied when controlled access to a single object is necessary
- ensures that only one instance of a class is allowed within a system.





https://www.geeksforgeeks.org/java-singleton-design-pattern-practices-examples/
https://www.vainolo.com/2012/04/29/singleton-design-pattern-sequence-diagram/
https://www.javaworld.com/article/2073352/core-java/simply-singleton.html
https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples

Singleton

```java
public class Singleton {
private static Singleton instance;
  public void operation() { System.out.println("operation"); }
  private Singleton() { }
  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
        if (instance == null) instance = new Singleton();
    } }
  return instance;
 }
}
```

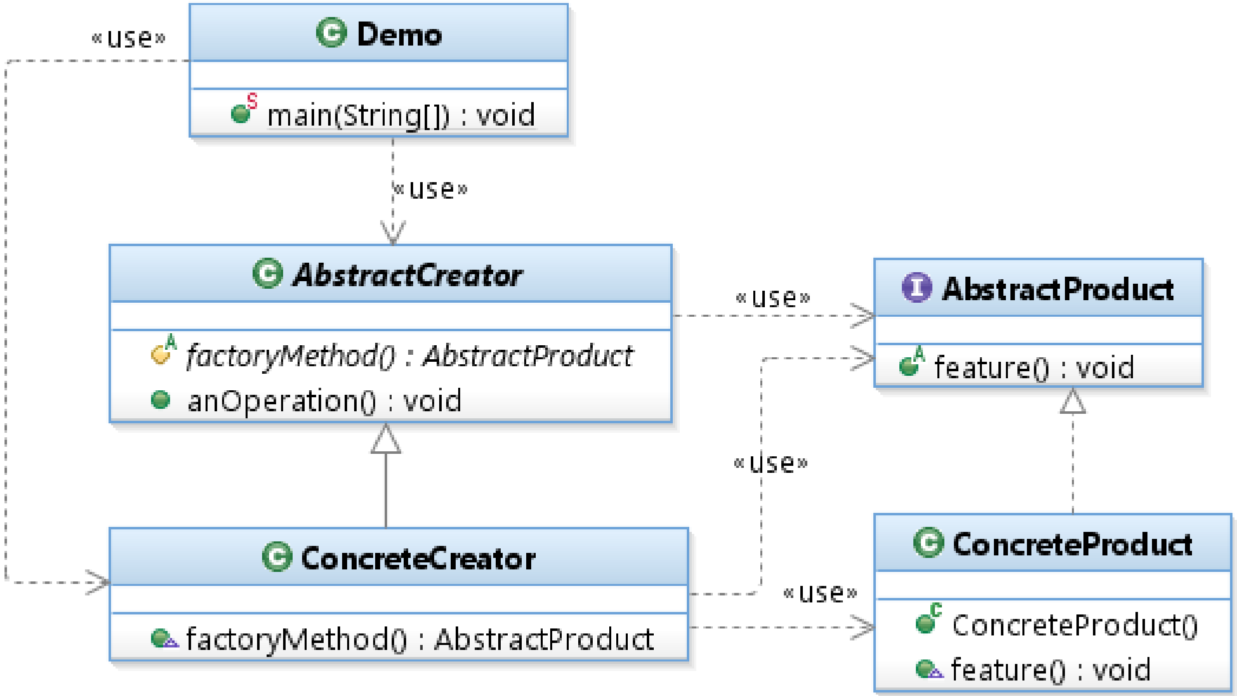
```java
public class Client {
  public Client(){}
  public void singletonInitializer() {
    Singleton s = Singleton.getInstance();
    s.operation();
  }
}
```

Singleton

```
public class Demo {
 public static void main(String[] args) {
    Client c = new Client();
    c.singletonInitializer();
    Singleton singleton = Singleton.getInstance();
    singleton.operation();
    Singleton anotherSingleton = Singleton.getInstance();
    anotherSingleton.operation();
    if (singleton == anotherSingleton) {
      System.out.println("Singleton and anotherSingleton are the 
same");
    } else {
        System.out.println("Singleton and anotherSingleton are 
different");
    }
 }
}
```

Factory method pattern

- used when a client doesn't know what concrete classes it will be required to create at runtime, but just wants to get a class that will do the job
- exposes a method to the client for creating the (single) object
- hides the construction of single object
- uses inheritance and relies on derived class or sub class to create object



Factory method pattern

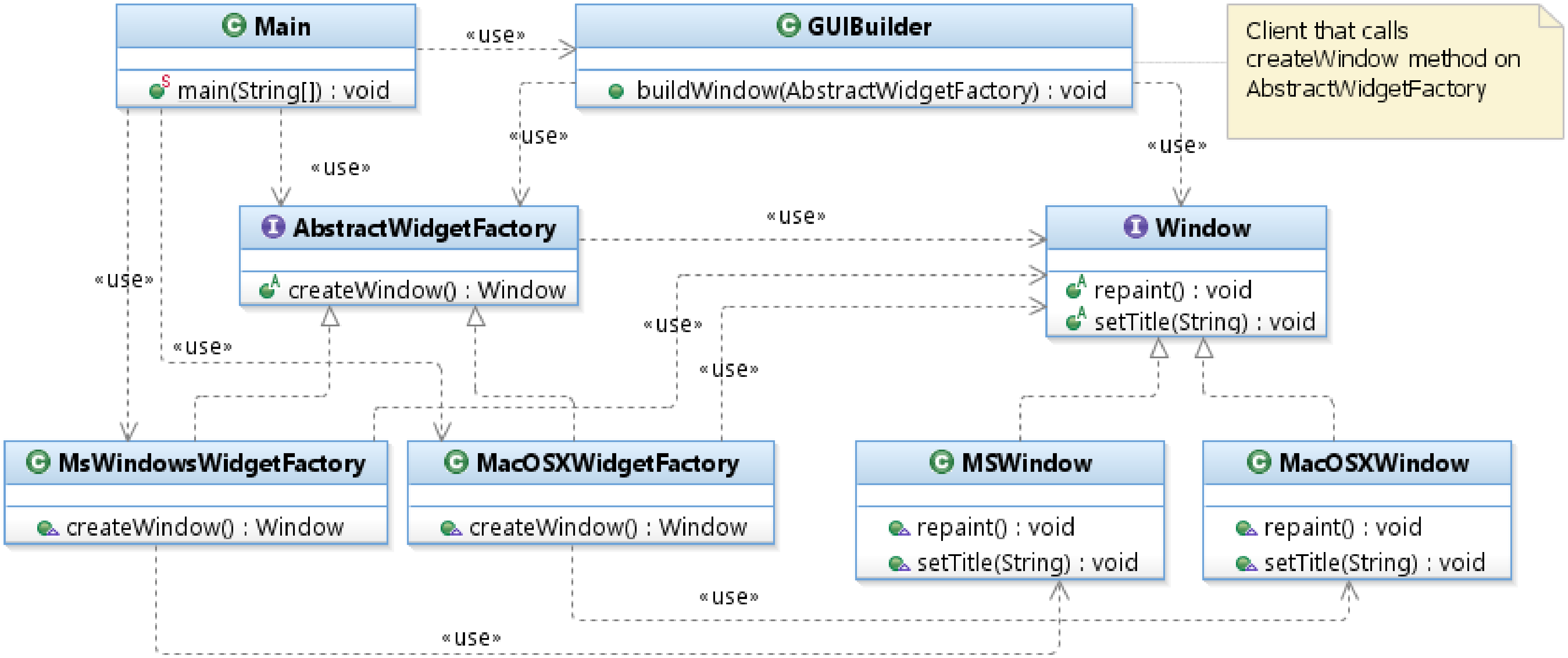
```java
public interface AbstractProduct {
  public void feature();
}

public class ConcreteProduct implements AbstractProduct {
  public ConcreteProduct(){
    System.out.println("Created: concrete product instance");
  }
  @Override
  public void feature() {
    System.out.println("Called: concrete product feature");
  }
}
```

Factory method pattern

```
public abstract class AbstractCreator {
 protected abstract AbstractProduct factoryMethod();
 public void anOperation(){
    AbstractProduct ap = factoryMethod();
    ap.feature();
 }
}
public class ConcreteCreator extends AbstractCreator {
  @Override
  public AbstractProduct factoryMethod() {
      System.out.println("Called: implementation of factory method; "
         + "will return: ConcreteProduct instance");
    return new ConcreteProduct();
  }
}


public class Demo {
  public static void main(String[] args) {
    AbstractCreator factory = new ConcreteCreator();
    factory.anOperation();
  }
}
```

Abstract factory pattern

- used when there is a need to create multiple families of products or to provide a library of products without exposing the implementation details
- uses composition to delegate responsibility of creating object to another class



Abstract factory pattern

```
public interface Window { // Abstract product
        public void repaint();
        public void setTitle(String text);
}
public class MSWindow implements Window { //Concrete product
        @Override
        public void repaint() {
                // MS Windows specific behaviour
        }
        @Override
        public void setTitle(String text) {
                // TODO Auto-generated method stub
        }
}
public class MacOSXWindow implements Window { //Concrete product
        @Override
        public void repaint() {
                // Mac OSX specific behaviour
        }
        @Override
        public void setTitle(String text) {
                // TODO Auto-generated method stub
        }
}
```

Abstract factory pattern

```java
// Abstract factory
public interface AbstractWidgetFactory {
        public Window createWindow();
}

// Concrete factory
public class MacOSXWidgetFactory implements AbstractWidgetFactory {
        @Override
        public Window createWindow() {
                MacOSXWindow window = new MacOSXWindow();
                return window;
        }
}

//Concrete factory
public class MsWindowsWidgetFactory implements AbstractWidgetFactory {
        @Override
        public Window createWindow() {
                MSWindow window = new MSWindow();
                return window;
        }
}
```

Abstract factory pattern

```
// Client
public class GUIBuilder {
        public void buildWindow(AbstractWidgetFactory widgetFactory) {
                Window window = widgetFactory.createWindow();
                window.setTitle("New Window");
        }
}

// Running example
public class Main {
        public static void main(String[] args) {
                GUIBuilder builder = new GUIBuilder();
                AbstractWidgetFactory widgetFactory = null;
                if (System.getProperty("os.name").contains("Windows"))
                        widgetFactory = new MsWindowsWidgetFactory();
                else
                        widgetFactory = new MacOSXWidgetFactory();
                builder.buildWindow(widgetFactory);
        }
}
```

Abstract Factory vs Factory Method

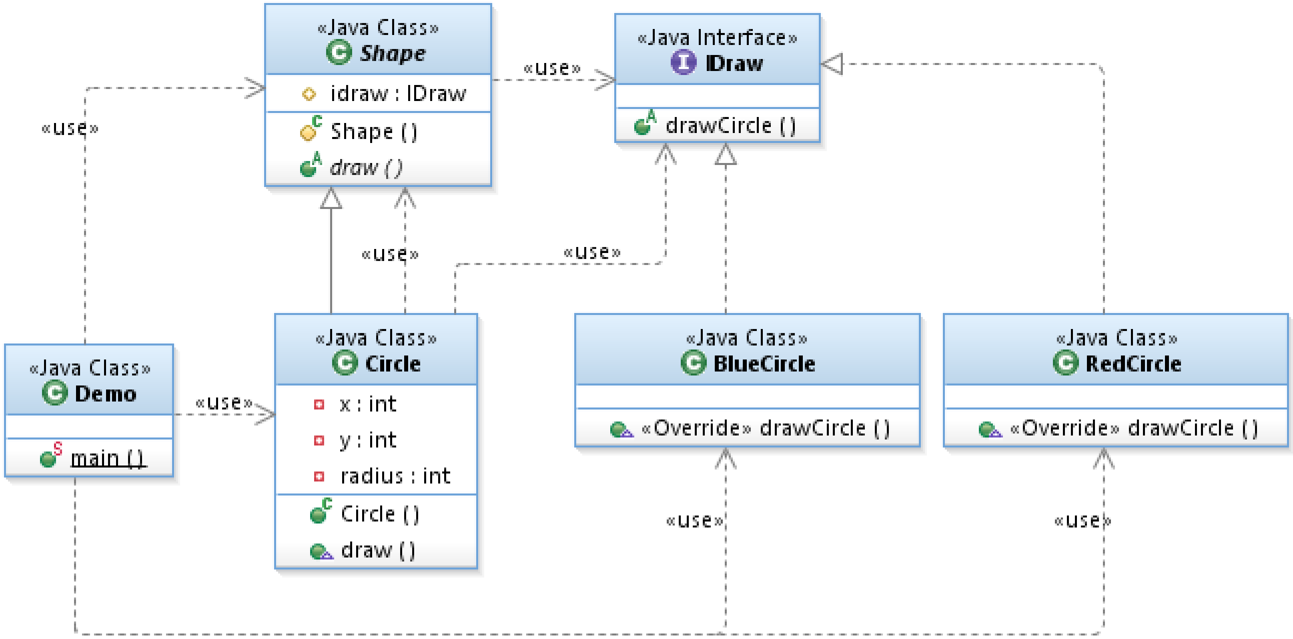
The main difference between Abstract Factory and Factory Method is that:

- with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition;
- Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

Abstract Factory is implemented by Composition;
but **Factory Method is implemented by Inheritance**.

Bridge pattern

- used when there is a need to decouple an abstraction from its implementation so that the two can vary independently
- comes under structural pattern as decouples implementation class and abstract class by providing a bridge structure between them
- involves an interface that acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes thus both types of classes can be altered structurally without affecting each other.



https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm

Bridge pattern

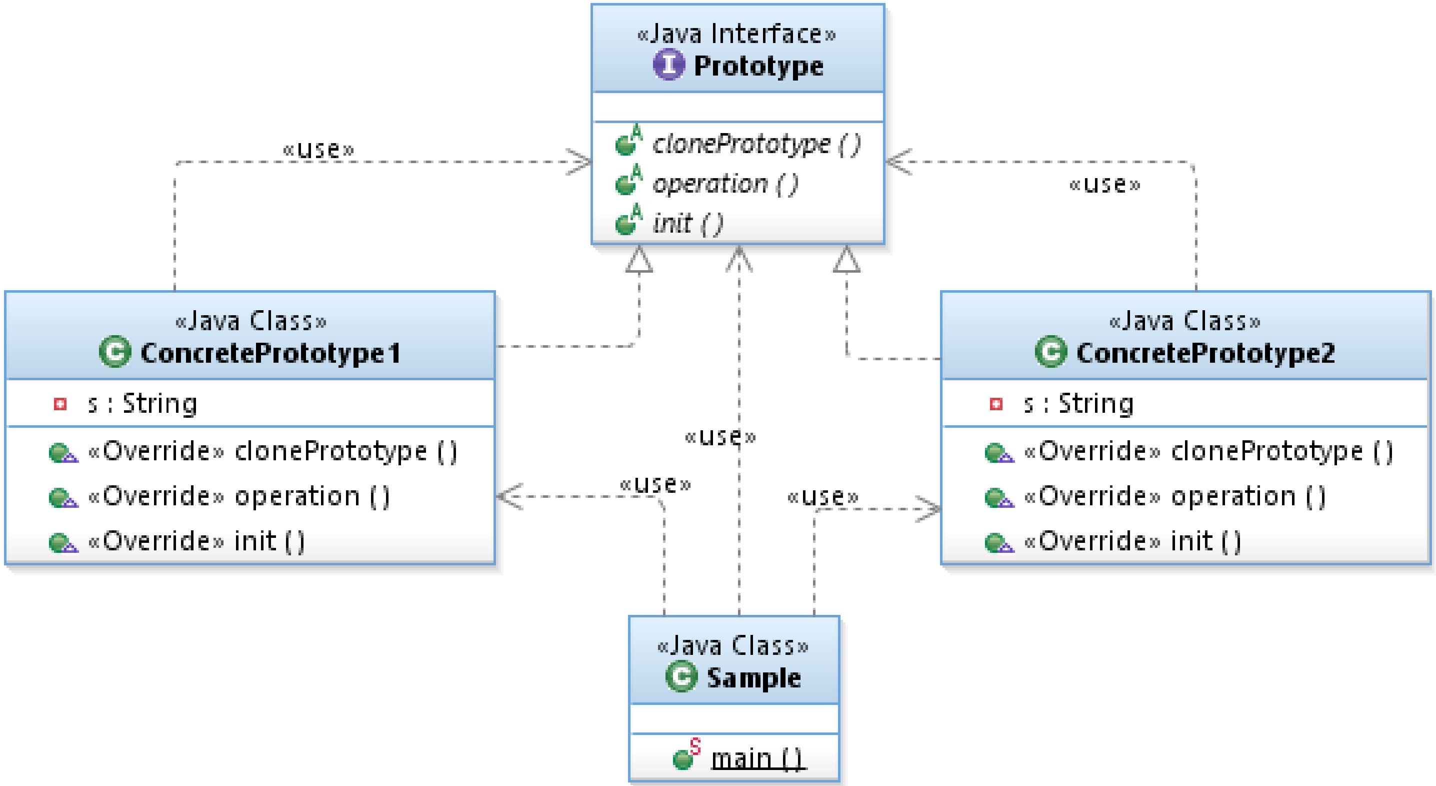
```
public interface IDraw {
  public void drawCircle(int radius, int x, int y);
}

public class RedCircle implements IDraw {
  @Override
  public void drawCircle(int radius, int x, int y) {
    System.out.println("Drawing Circle[ color: red, radius: " +
             radius + ", x: " + x + ", " + y + "]");
  }
}

public class BlueCircle implements IDraw {
  @Override
  public void drawCircle(int radius, int x, int y) {
    System.out.println("Drawing Circle[ color: blue, radius: " +
             radius + ", x: " + x + ", " + y + "]");
  }
}
```

Bridge pattern

```
public abstract class Shape {
  protected IDraw idraw;
  protected Shape(IDraw idraw) {
    this.idraw = idraw;   }
  public abstract void draw();
}
public class Circle extends Shape {
  private int x, y, radius;
  public Circle(int x, int y, int radius, IDraw idraw) {
    super(idraw);
    this.x = x; this.y = y;
    this.radius = radius;
  }
  public void draw() {
    idraw.drawCircle(radius, x, y);
  }
}
              public class Demo {
                public static void main(String[] args) {
                  Shape redCircle = new Circle(20, 20, 10, new RedCircle());
                  Shape greenCircle = new Circle(20, 20, 10, new BlueCircle());
                  redCircle.draw(); greenCircle.draw();  }
              }
```

Prototype pattern

- used when composition, creation, and representation of objects should be decoupled from a system
- relies on new objects creation through cloning and modifying of existing and initialized prototypes (what is especially productive in case of expensive initialization)
- helpful when classes to be created are specified at runtime



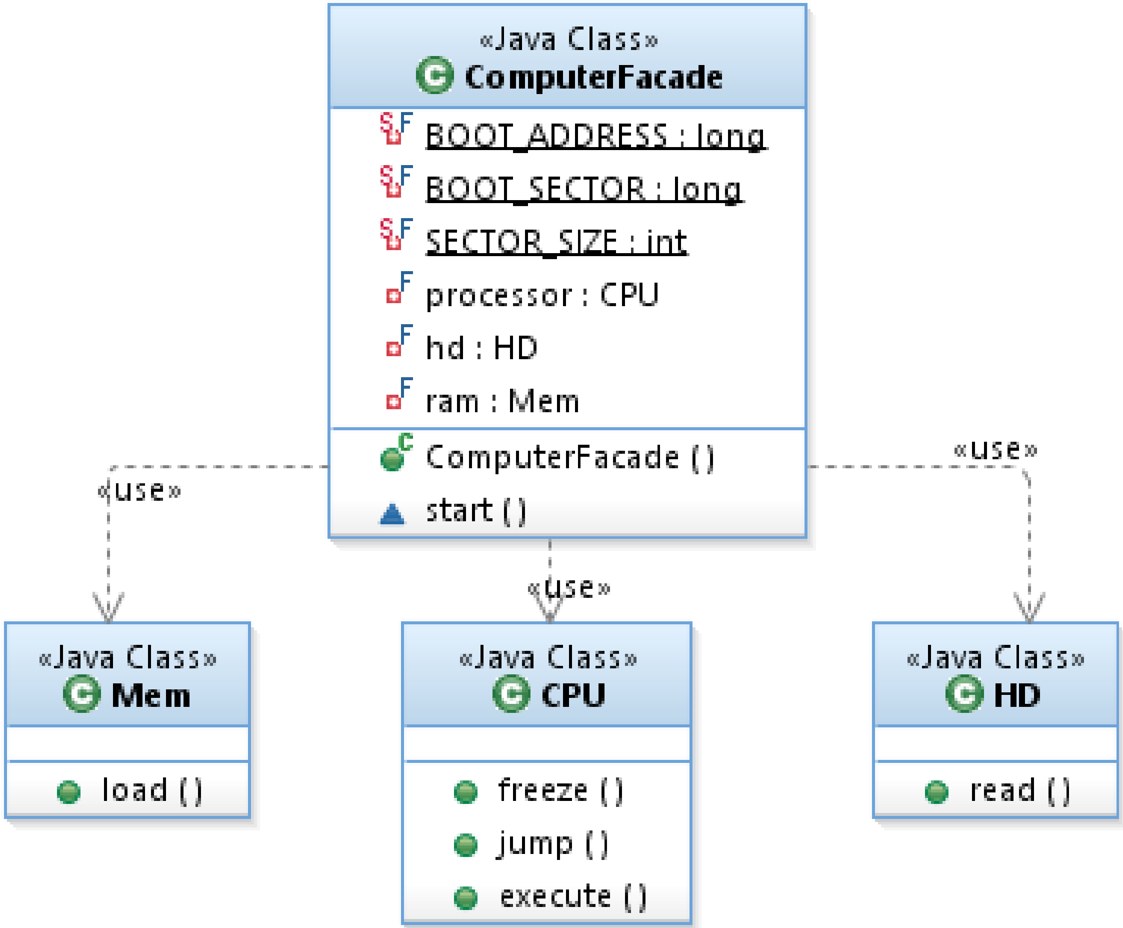
Prototype pattern

```
public interface Prototype extends Cloneable {
 public abstract Prototype clonePrototype() throws
CloneNotSupportedException;

 public abstract void operation();
 public abstract void init(String s);
}

public class ConcretePrototype1 implements Prototype
{
 private String s;

 @Override
 public Prototype clonePrototype() throws
CloneNotSupportedException {
  return (ConcretePrototype1) clone();
 }

 @Override
 public void operation() {
  System.out.println(s);
 }

 @Override
 public void init(String s) {
  this.s = s.toUpperCase();
 }
}
```

```
public class Sample {
 public static void main(String[] args) {
   try {
     Prototype p1 = new ConcretePrototype1();
     p1.init("test TEST");
     p1.clonePrototype().operation();
     p1.init("another ANOTHER");
     p1.clonePrototype().operation();

     Prototype p2 = new ConcretePrototype2();
     p2.init("test TEST");
     p2.clonePrototype().operation();
   } catch (CloneNotSupportedException e) {
        e.printStackTrace();
     }
  }
 }
```

Facade pattern

- used to wrap a complicated subsystem with a simpler interface
- defines a higher-level interface that makes the subsystem easier to use



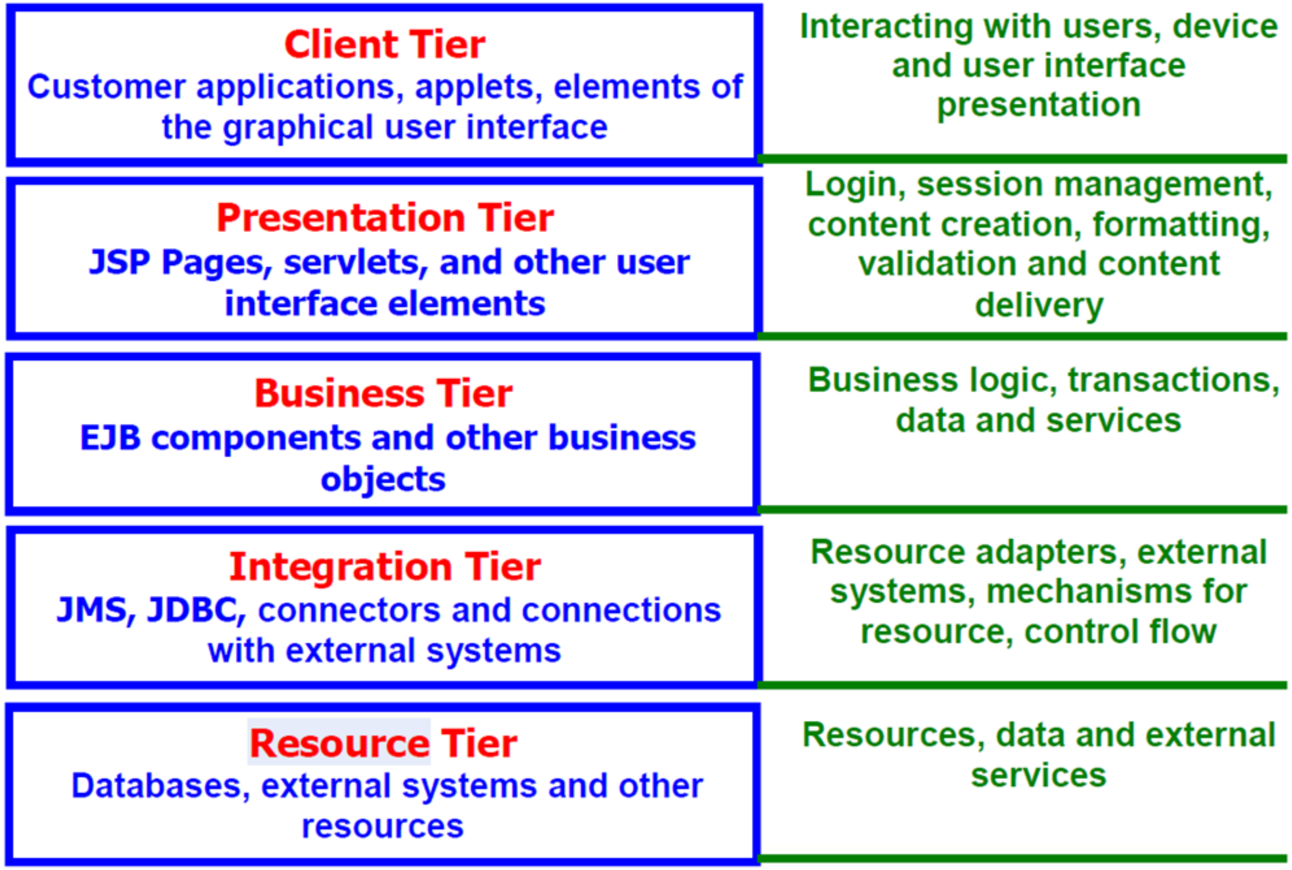
Facade pattern

```
public class ComputerFacade {
    private static final long BOOT_ADDRESS = 0;
    private static final long BOOT_SECTOR = 0;
    private static final int SECTOR_SIZE = 0;
    private final CPU processor;
    private final HD hd;
    private final Mem ram;

    public ComputerFacade() {
        processor = new CPU();
        hd = new HD();
        ram = new Mem();
    }

    void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR,
SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}
```

```
public class CPU {
    public void freeze() { }
    public void jump(long position) { }
    public void execute() { }
}

public class HD {
    public char[] read(long lba, int size) {
     return null; }
}

public class Mem {
    public void load(long position, char[]
data) {}
}
```

Multitiered Information System

by D.Alur, J.Crupi, D. Malks, Core J2EE. Desin Patterns

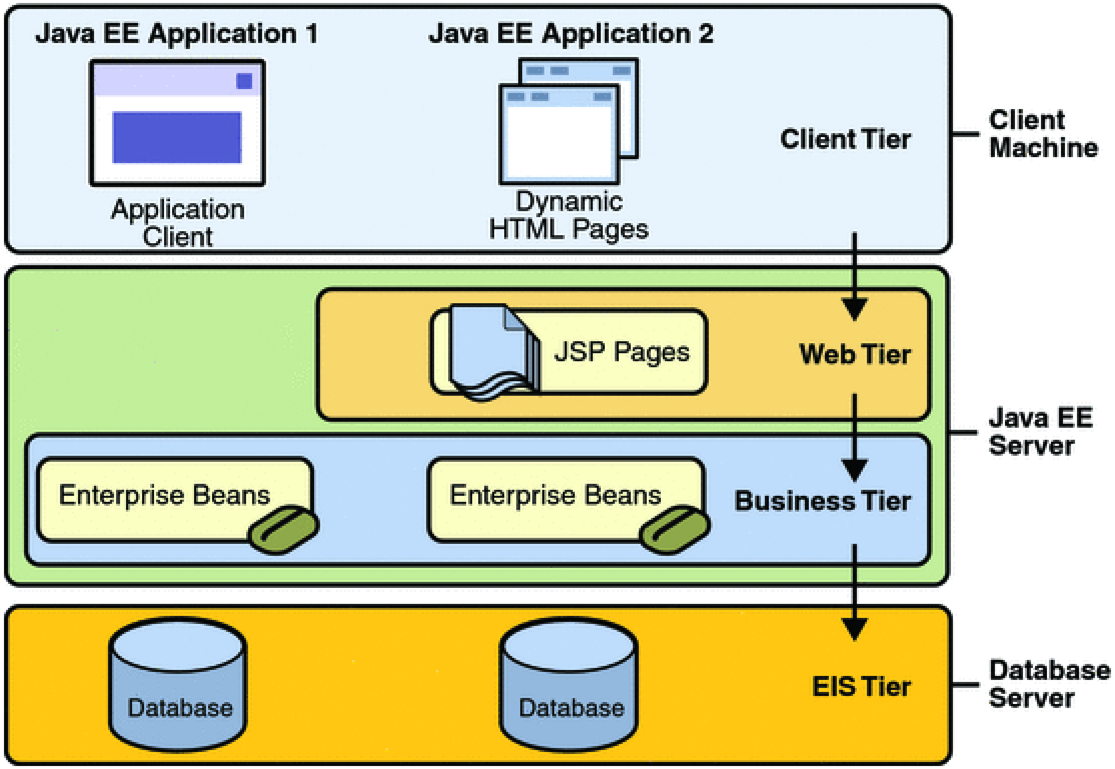


Client Tier Customer applications, applets, elements of the graphical user interface	Interacting with users, device and user interface presentation
Presentation Tier JSP Pages, servlets, and other user interface elements	Login, session management, content creation, formatting, validation and content delivery
Business Tier EJB components and other business objects	Business logic, transactions, data and services
Integration Tier JMS, JDBC, connectors and connections with external systems	Resource adapters, external systems, mechanisms for resource, control flow
Resource Tier Databases, external systems and other resources	Resources, data and external services

Data Access Object, Service Activator, Domain Store, Web Service Broker

Design and implementation of the multi-tier software can be based on **Presentation, Business and Integration** Patterns.

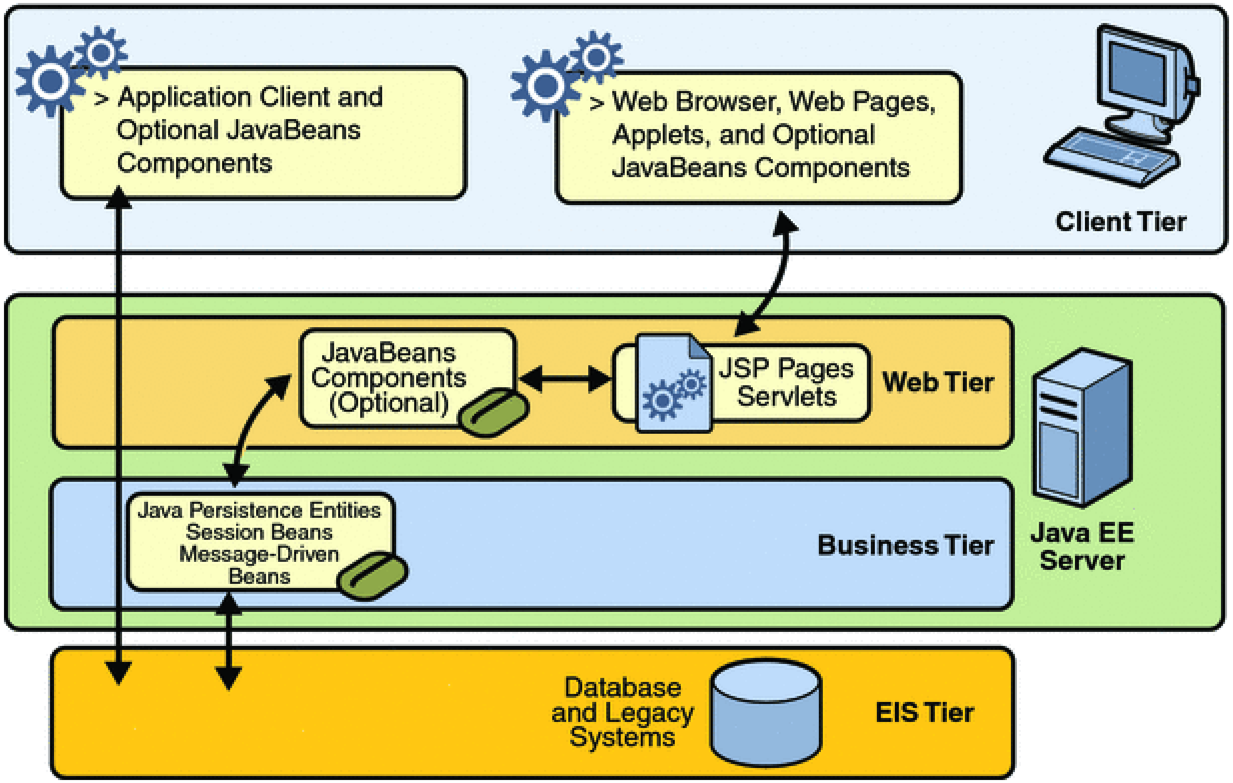
Distributed Multitiered Applications



* please note that technologies depicted are a bit outdated

http://docs.oracle.com/javaee/5/tutorial/doc/bnaay.html

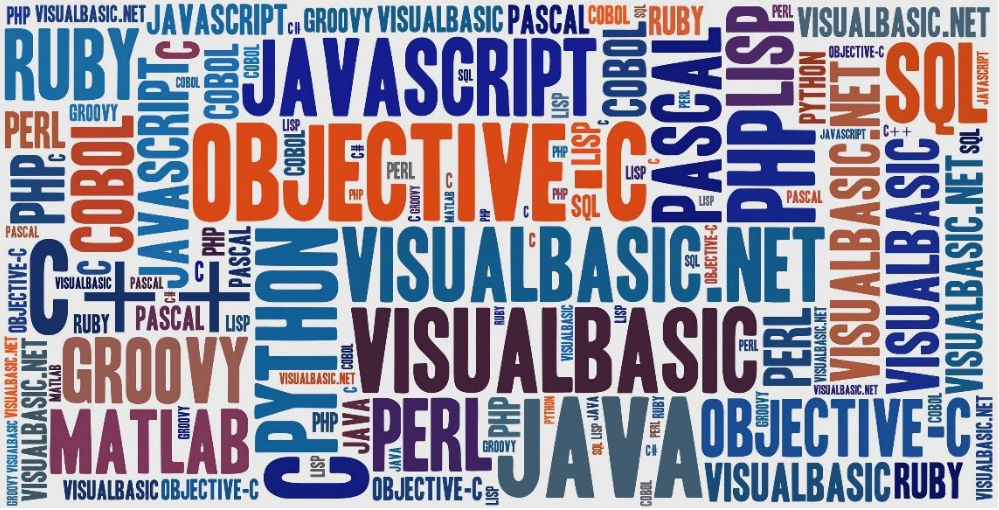
Distributed Multitiered Applications

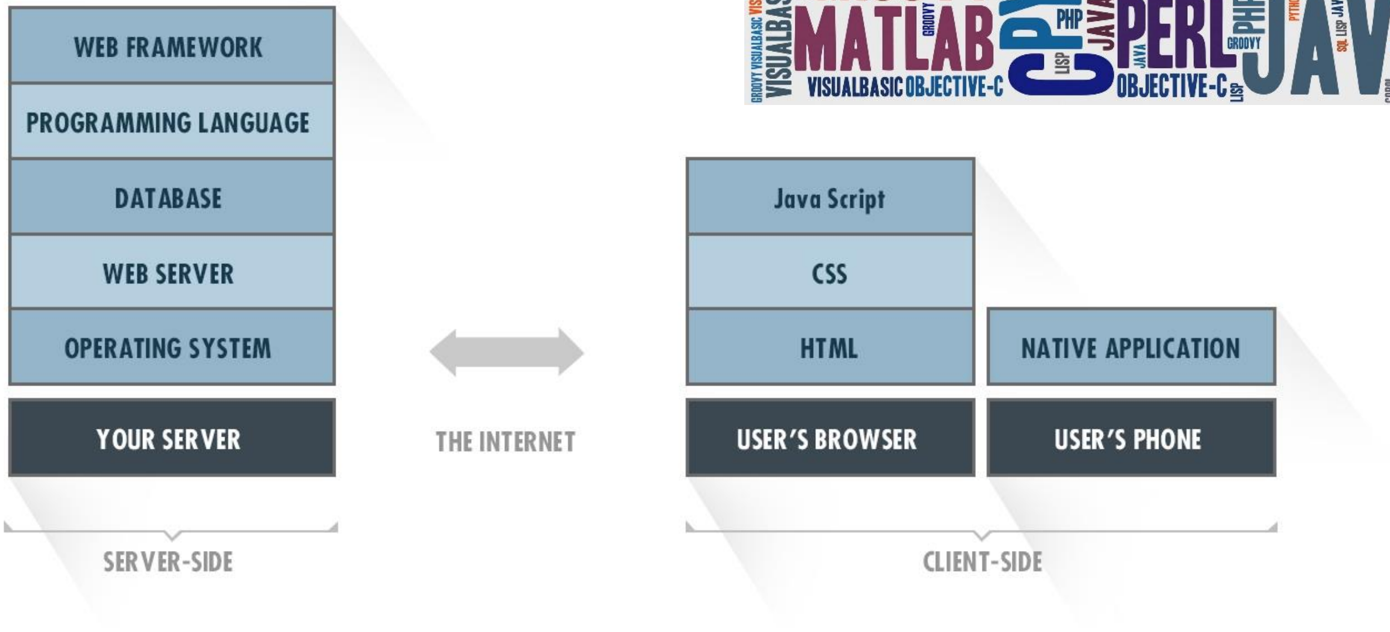

> Application Client and Optional JavaBeans Components
> Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
JavaBeans Components (Optional)
JSP Pages Servlets
Web Tier
Java Persistence Entities Session Beans Message-Driven Beans
Business Tier
Java EE Server
Database and Legacy Systems
EIS Tier

* please note that technologies depicted are a bit outdated

http://docs.oracle.com/javaee/5/tutorial/doc/bnaay.html

Web technology stack





WEB FRAMEWORK

PROGRAMMING LANGUAGE

DATABASE

WEB SERVER

OPERATING SYSTEM

YOUR SERVER

THE INTERNET

Java Script

CSS

HTML

NATIVE APPLICATION

USER'S BROWSER

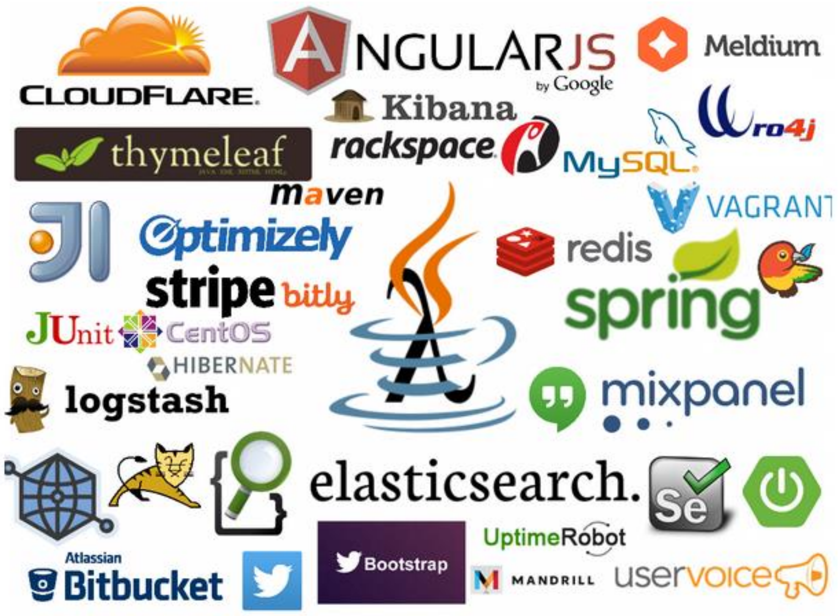
USER'S PHONE

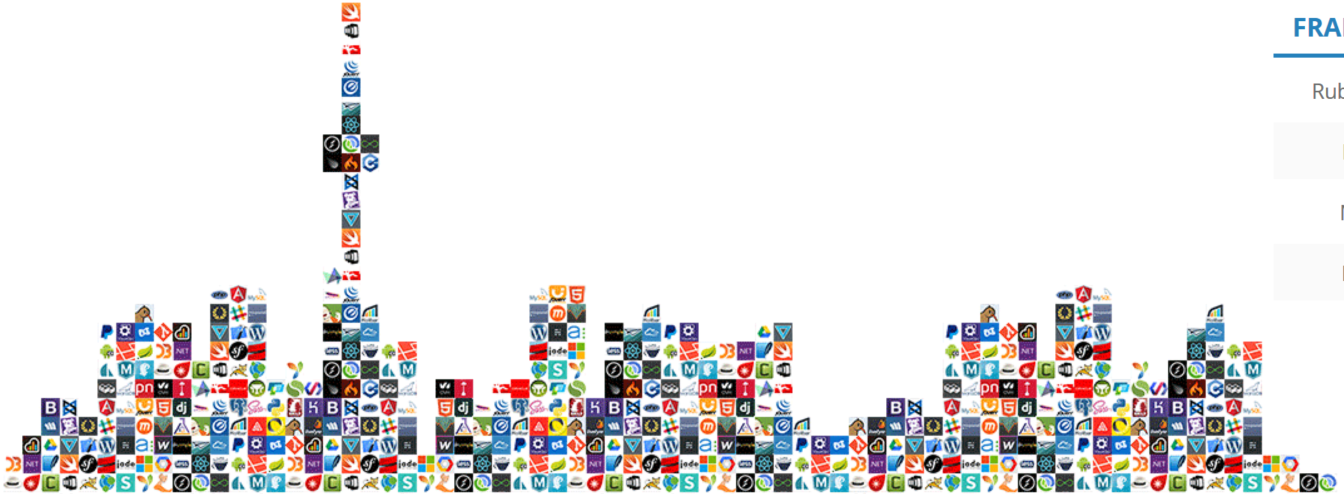
SERVER-SIDE

CLIENT-SIDE

http://svsg.co/how-to-choose-your-tech-stack/

Web technology stack





FRAMEWORK	LANGUAGE
Ruby on Rails	Ruby
Django	Python
Node.js	Javascript
Laravel	PHP
.NET	C#

http://www.slideshare.net/sagaroceanic11/understanding-c-for-java
http://dreamix.eu/blog/java/elasticsearch-spring-boot-and-angular-js-into-action-for-building-b2c-web-applications

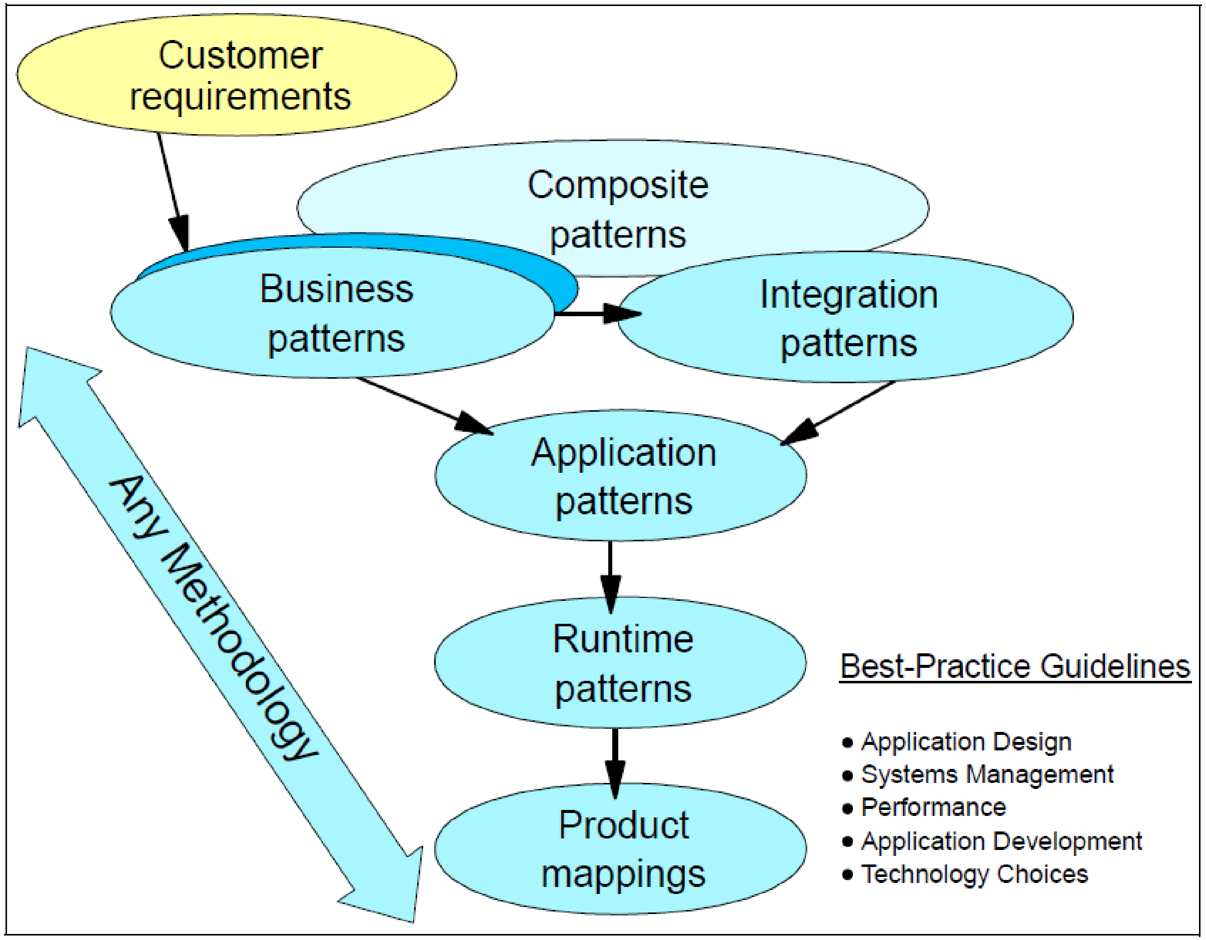
The Patterns for e-business layered asset model

“The Patterns approach is based on a set of layered assets that can be exploited by any existing development methodology. These layered assets are structured in a way that each level of detail builds on the last. These assets include:

- **Business patterns** that identify the interaction between users, businesses, and data.
- **Integration patterns** that tie multiple Business patterns together when a solution cannot be provided based on a single Business pattern.
- **Composite patterns** that represent commonly occurring combinations of Business patterns and Integration patterns.
- **Application patterns** that provide a conceptual layout describing how the application components and data within a Business pattern or Integration pattern interact.
- **Runtime patterns** that define the logical middleware structure supporting an Application pattern. Runtime patterns depict the major middleware nodes, their roles, and the interfaces between these nodes.
- **Product mappings** that identify proven and tested software implementations for each Runtime pattern.
- **Best-practice guidelines** for design, development, deployment, and management of e-business applications.”

M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, T. Newling: *Patterns: Service-Oriented Architecture and Web Services,* 2004

The Patterns for e-business layered asset model



Best-Practice Guidelines

- Application Design
- Systems Management
- Performance
- Application Development
- Technology Choices

WebSphere. software

IBM

Draft Document for Review March 31, 2004 3:33 pm

SG24-6303-00

Patterns: Service-Oriented Architecture and Web Services

Design service-oriented architectures using Web services

Explore service bus, directory, and gateway solutions

Learn by example with practical scenarios

Mark Endrei
Jenny Ang
Ali Arsanjani
Sook Chua
Philippe Comte
Pål Krogdahl
Min Luo
Tony Newling

Redbooks

ibm.com/redbooks

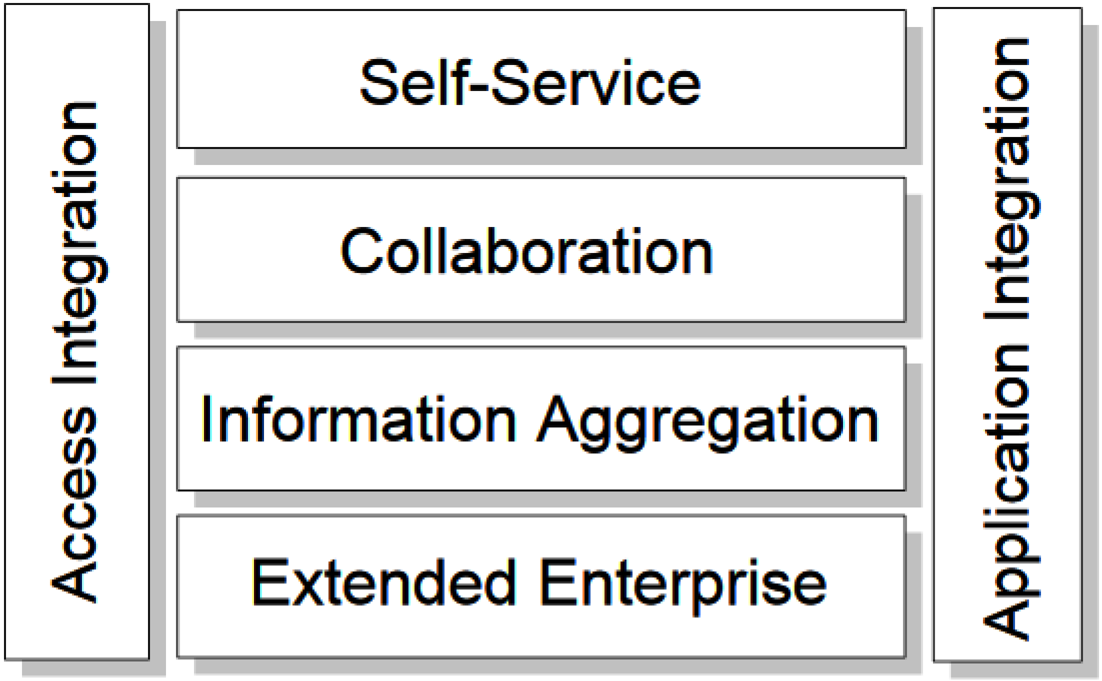
Business patterns

Business Patterns	Description	Examples
Self-Service (User-to-Business)	Applications where users interact with a business via the Internet or intranet	Simple Web site applications
Information Aggregation (User-to-Data)	Applications where users can extract useful information from large volumes of data, text, images, etc.	Business intelligence, knowledge management, Web crawlers
Collaboration (User-to-User)	Applications where the Internet supports collaborative work between users	E-mail, community, chat, video conferencing, etc.
Extended Enterprise (Business-to-Business)	Applications that link two or more business processes across separate enterprises	EDI, supply chain management, etc.

Integration patterns

Integration Patterns	Description	Examples
Access Integration	Integration of a number of services through a common entry point	Portals
Application Integration	Integration of multiple applications and data sources without the user directly invoking them	Message brokers, workflow managers

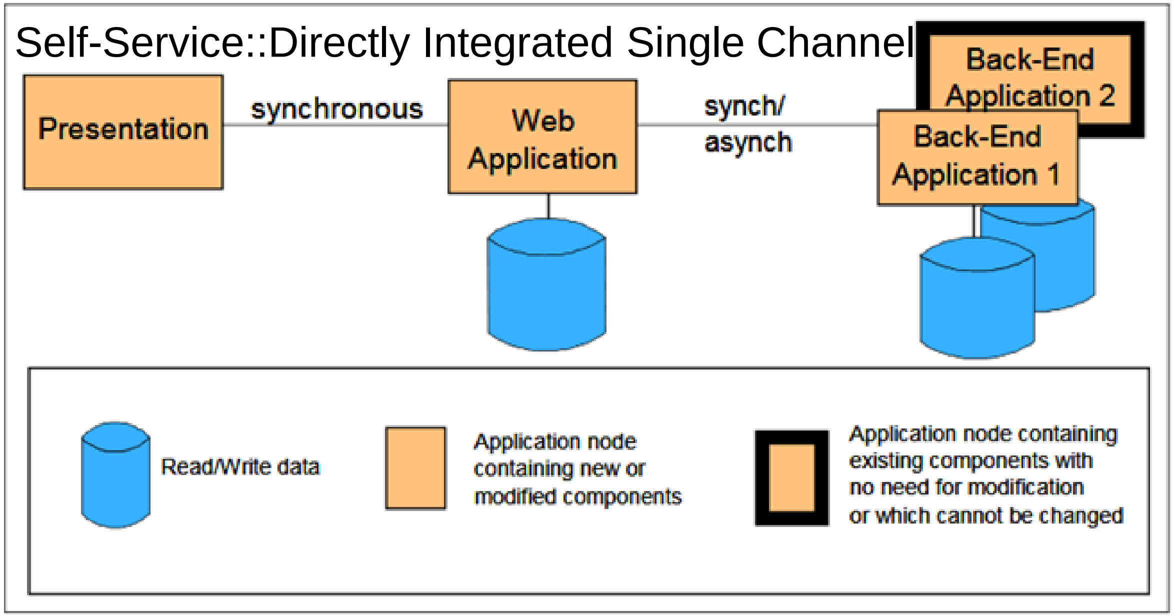
Custom design

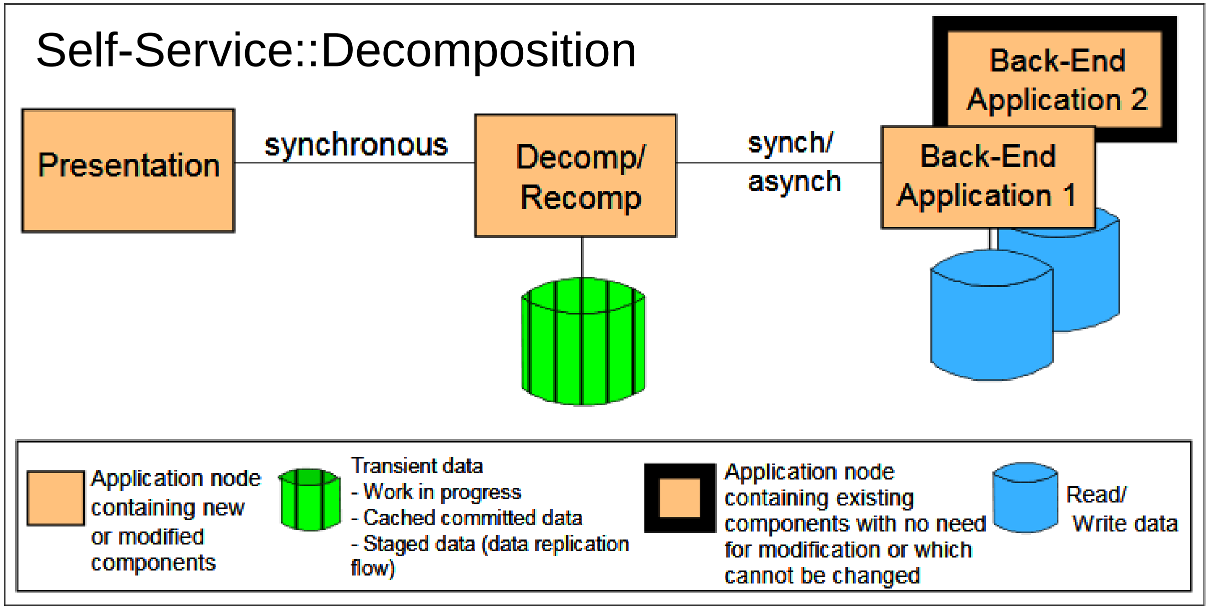


Composite patterns

Composite Patterns	Description	Examples
Electronic Commerce	User-to-Online-Buying	www.macys.com www.amazon.com
Portal	Typically designed to aggregate multiple information sources and applications to provide uniform, seamless, and personalized access for its users.	Enterprise Intranet portal providing self-service functions such as payroll, benefits, and travel expenses. Collaboration providers who provide services such as e-mail or instant messaging.
Account Access	Provide customers with around-the-clock account access to their account information.	Online brokerage trading apps. Telephone company account manager functions. Bank, credit card and insurance company online apps.
Trading Exchange	Allows buyers and sellers to trade goods and services on a public site.	Buyer's side – interaction between buyer's procurement system and commerce functions of e-Marketplace. Seller's side – interaction between the procurement functions of the e-Marketplace and its suppliers.
Sell-Side Hub (Supplier)	The seller owns the e-Marketplace and uses it as a vehicle to sell goods and services on the Web.	www.carmax.com (car purchase)
Buy-Side Hub (Purchaser)	The buyer of the goods owns the e-Marketplace and uses it as a vehicle to leverage the buying or procurement budget in soliciting the best deals for goods and services from prospective sellers across the Web.	www.wre.org (WorldWide Retail Exchange)

Application patterns


Self-Service::Directly Integrated Single Channel
Presentation
synchronous
Web Application
synch/ asynch
Back-End Application 2
Back-End Application 1
Read/Write data
Application node containing new or modified components
Application node containing existing components with no need for modification or which cannot be changed


Self-Service::Decomposition
Presentation
synchronous
Decomp/ Recomp
synch/ asynch
Back-End Application 2
Back-End Application 1
Application node containing new or modified components
Transient data
- Work in progress
- Cached committed data
- Staged data (data replication flow)
Application node containing existing components with no need for modification or which cannot be changed
Read/ Write data

URI

Uniform Resource Identifier is a global, rigid resource identifier of the form:

```
scheme ":" hier_part ["?" query ] [ # fragment ]
```

`scheme` – string (a letter followed by letters, digits, and `["+"|"."|"-"]`)

`hier_part` – has the following syntax:

```
[userInfo "@"] hostname [:port_number] [path]
```

`query` – optional information, commonly organized as a sequence of: `<key>=<value>` pairs, separated by `";"` or `"&"`

`fragment` – optional part (local reference)

HTTP URL conforms to the syntax of a generic URI:

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment
]
```

Example:

```
http://example.org/users?name=Adam
```

URI, URL, URIRef

- The syntax of URI has been defined in [RFC2396] and updated in [RFC2732]
- The current generic URI syntax specification is [RFC3986]
- Uniform Resource Locators were defined in [RFC1738]

- URI can be absolute or relative:
  - absolute : a resource is identified with full and context independent resource reference
  - eelative : a reference has not given full information to identify a resource and missing information must be derived from the context
- A URIRef is relative form of URI
  - consists of URI and optional `fragment` preceded by `#`
  - absolute URI of `#section2` from the document `http://www.example.org/index.html` is `http://www.example.org/index.html#section2`

IRI

- Internationalized Resource Identifier is a complement to URI
- provides wider repertoire of characters allowed
  - Unicode/ISO10646 characters beyond U+007F
  - private characters of that set can occur only in query parts
- Standardized in [RFC3987] that defines "internationalized" versions corresponding to other constructs from [RFC3986], such as URI references.
- In many cases URI and IRI are used interchangeably, but practical replacement of URIs (or URI references) by IRIs (or IRI references) depends on the application.

HTTP - Hypertext Transfer Protocol

- HTTP methods
  - HTTP/1.0
    - **GET**, HEAD, **POST**,
  - HTTP/1.1
    - **GET**, HEAD, **POST**, **PUT**, **DELETE**, TRACE, OPTIONS, CONNECT, PATCH

- GET - method used probably most often
  - URI with attributes: (`http://books.example.com/books/12345`)

```
GET /books/12345 HTTP/1.1
Host: books.example.com
```

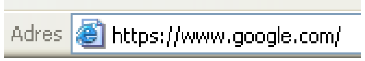
  - URI with method and attributes: `http://books.example.com/service?method=lookupBook&id=123345`

```
GET/service?method=lookupBook&id=12345 HTTP/1.1
Host: books.example.com
```

https://tools.ietf.org/html/rfc2068
https://tools.ietf.org/html/rfc2616

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

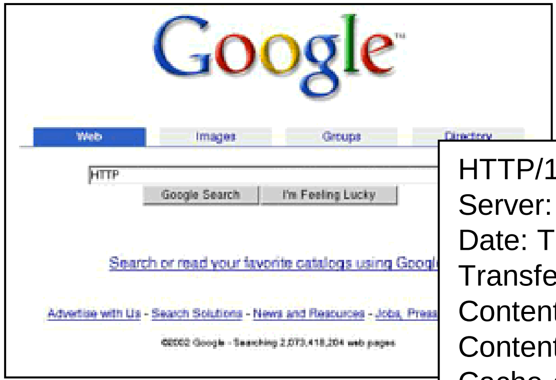
HTTP GET



request

```
GET /search?hl=en&q=HTTP&btnG=Google+Search HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 Galeon/1.2.0 (X11; Linux i686; U;) Gecko/20020326
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,
        text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,
        text/css,*/*;q=0.1
Accept-Language: en
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
```

Client



Search or read your favorite catalogs using Googl

Advertise with Us - Search Solutions - News and Resources - Jobs, Press

©2002 Google - Searching 2,073,418,204 web pages

response

```
HTTP/1.1 200 OK
Server: GWS/2.0
Date: Tue, 21 May 2002 12:34:56 GMT
Transfer-Encoding: chunked
Content-Encoding: gzip
Content-Type: text/html
Cache-control: private
Set-Cookie: PREF=ID=58c005a7065c0996:TM=1021283456:LM=1021283456:S=OLJcXi3RhSE;
        domain=.google.com; path=/; expires=Sun, 17-Jan-2038 19:14:07 GMT

(Web content compressed with gzip)
```

Server

HTTP protocol – GET request

- request method 🟥
- HTTP headers
  - general headers (appears in requests and responses) 🟧
  - request headers (specific to the request) 🟨
  - entity headers (present, if there is a content) 🟧
- content 🟥

```
GET /request=WMS HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shock
wave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application
/msword, */*
Accept-Language: pl
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1
.4322)
Host: localhost:3541
Connection: Keep-Alive
```

HTTP protocol – POST request

- request method 🟥
- HTTP headers
  - general headers (appears in requests and responses) 🟧
  - request headers (specific to the request) 🟨
  - entity headers (present, if there is a content) 🟫
- content 🟥

```
POST /search HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 Galeon/1.2.5 (X11; Linux i686; U;) Gecko/20020606
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,
        text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,
        text/css,*/*;q=0.1
Accept-Language: en
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 31

hl=en&q=HTTP&btnG=Google+Search
```

HTTP protocol – response

- status line (HTTP version, status code, short description) 🟥
- HTTP headers
  - general headers (appears in requests and responses) 🟧
  - response headers (specific to the response) 🟨
  - entity headers (present, if there is a content) 🟫
- content 🟥

```
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.1
Date: Thu, 23 Aug 2007 17:48:48 GMT
X-Powered-By: ASP.NET
Location: localstart.asp
Content-Length: 121
Content-Type: text/html
Set-Cookie: ASPSESSIONIDACQBQQQS=CEGLAHJCPAEBAIINILNPHKAF; path=/
Cache-control: private

<head><title>Object moved</title></head>
 <body><h1>Object Moved</h1>This object may be found <a
HREF="">here</a>.</body>
```

HTTP status codes

- `1xx` (*Informational*): The request was received, continuing process
- `2xx` (*Successful*): The request was successfully received, understood, and accepted
- `3xx` (*Redirection*): Further action needs to be taken in order to complete the request
- `4xx` (*Client Error*): The request contains bad syntax or cannot be fulfilled
- `5xx` (*Server Error*): The server failed to fulfill an apparently valid request

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Introduction to REST

REST (Representational State Transfer)

- an architectural ***pattern*** for developing web services
- REST web services communicate over the HTTP specification, using HTTP vocabulary:
  - HTTP methods (GET, PUT, POST, DELETE …)
  - **HTTP URI syntax** (paths, parameters, etc.)
  - Media types (xml, json, html, plain text, etc)
  - HTTP Response codes

Author

- **Roy Fielding:** *Architectural Styles and the Design of Network-based Software Architectures*, PhD Thesis, 2000.

Introduction to REST

- Main characteristics
  - Representational
    - Clients possess the information necessary to identify, modify, and/or delete a web resource.
  - State
    - All resource state information is stored on the client.
  - Transfer
    - Client state is passed from the client to the service through HTTP.
- Services that do not conform constraints mentioned below are not strictly RESTful web services.
  - Uniform interface
  - Decoupled client-server interaction
  - Stateless
  - Cacheable
  - Layered
  - Extensible through code on demand (optional)

Swagger

- Swagger
  - the world’s largest framework of API developer tools for the OpenAPI Specification(OAS), enabling development across the entire API lifecycle, from design and documentation, to test and deployment
- Swagger editor:
  - an open source editor fully dedicated to Swagger-based APIs
  - may be used to design, describe, and document your API
  - great for quickly getting started with the Swagger specification
  - clean, efficient, and armed with a number of features to help you design and document your RESTful interfaces, straight out of the box

https://swagger.io/
https://swagger.io/swagger-editor/

Swagger

- Swagger codegen:
  - an open source code-generator to build server stubs and client SDKs directly from a Swagger defined RESTful API with source code available from:
    - `https://github.com/swagger-api/swagger-codegen`
  - can be downloaded

    -
      ```
      git clone https://github.com/swagger-api/swagger-codegen
      ```
  - and run as the executable .jar to generate code

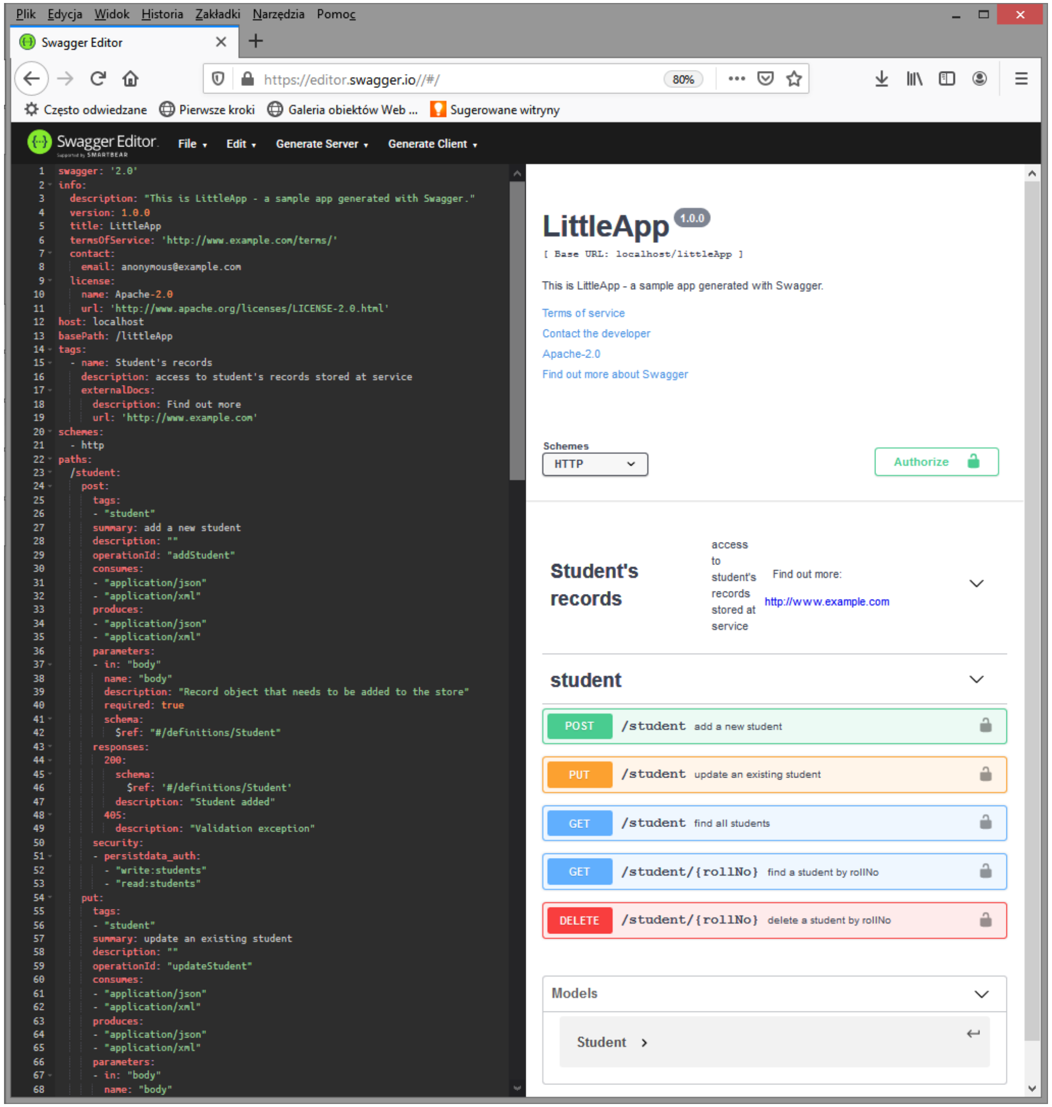
    -
      ```
      cd swagger-codegen
      ```

    -
      ```
      java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar generate -i <path of your Swagger specification> -l <language>
      ```
  - in particular suitable for spring

    -
      ```
      java -jar swagger-codegen-cli-2.2.1.jar generate -i http://petstore.swagger.io/v2/swagger.json -l spring -o samples/server/petstore/springboot
      ```

https://swagger.io/docs/open-source-tools/swagger-codegen/
https://github.com/swagger-api/swagger-codegen/wiki/Server-stub-generator-HOWTO#java-springboot

LittleApp example



```yaml
swagger: '2.0'
info:
  description: "This is LittleApp - a sample app generated with Swagger."
  version: 1.0.0
  title: LittleApp
  termsOfService: 'http://www.example.com/terms/'
  contact:
    email: anonymous@example.com
  license:
    name: Apache-2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
host: localhost
basePath: /littleApp
tags:
  - name: Student's records
    description: access to student's records stored at service
    externalDocs:
      description: Find out more
      url: 'http://www.example.com'
schemes:
  - http
paths:
  /student:
    post:
      tags:
      - "student"
      summary: add a new student
      description: ""
      operationId: "addStudent"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/json"
      - "application/xml"
      parameters:
      - in: "body"
        name: "body"
        description: "Record object that needs to be added to the store"
        required: true
        schema:
          $ref: "#/definitions/Student"
      responses:
        200:
          schema:
            $ref: '#/definitions/Student'
          description: "Student added"
        405:
          description: "Validation exception"
      security:
      - persistdata_auth:
        - "write:students"
        - "read:students"
    put:
      tags:
      - "student"
      summary: update an existing student
      description: ""
      operationId: "updateStudent"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/json"
      - "application/xml"
      parameters:
      - in: "body"
        name: "body"
```

LittleApp 1.0.0

[Base URL: localhost/littleApp]

This is LittleApp - a sample app generated with Swagger.

Terms of service

Contact the developer

Apache-2.0

Find out more about Swagger

Schemes: HTTP

Authorize

Student's records — access to student's records stored at service — Find out more: http://www.example.com

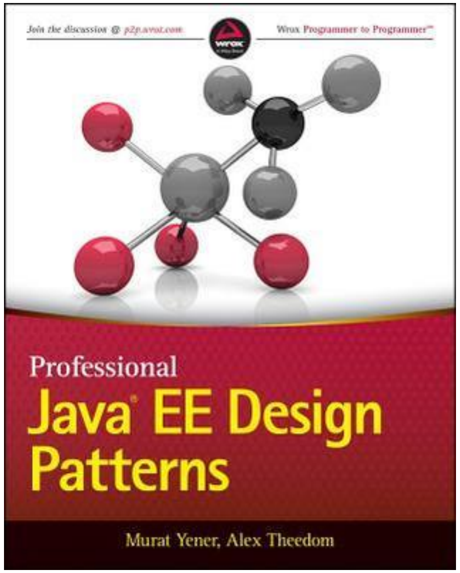
student

- POST /student add a new student
- PUT /student update an existing student
- GET /student find all students
- GET /student/{rollNo} find a student by rollNo
- DELETE /student/{rollNo} delete a student by rollNo

Models

Student ›

Resources


Join the discussion @ p2p.wrox.com
wrox
Wrox Programmer to Programmer™
Professional
Java® EE Design Patterns
Murat Yener, Alex Theedom

THE EXPERT'S VOICE® IN JAVA
Java Design Patterns
A Tour with 23 Gang of Four Design Patterns in Java
Vaskaran Sarcar
Apress®

https://www.journaldev.com/1827/java-design-patterns-example-tutorial
https://dzone.com/articles/gof-design-patterns-using-java-part-1
http://www.fluffycat.com/Java-Design-Patterns/
https://sourcemaking.com/design_patterns
https://www.avajava.com/tutorials/categories/design-patterns
https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
http://www.blackwasp.co.uk/GofPatterns.aspx