Chapter 1

INTRODUCTION

Enterprise Application Architecture

The field of enterprise architecture essentially started in 1987, with the publication in the IBM Systems Journal of an article titled "A Framework for Information Systems Architecture," by J.A. Zachman. In that paper, Zachman laid out both the challenge and the vision of enterprise architectures that would guide the field for the next 20 years. The challenge was to manage the complexity of increasingly distributed systems. As Zachman said: "*The cost involved and the success of the business depending increasingly on its information systems require a disciplined approach to the management of those systems".* Zachman's vision was that business value and agility could best be realized by a holistic approach to systems architecture that explicitly looked at every important issue from every important perspective.

His multi perspective approach to architecting systems is what Zachman originally described as an information systems architectural framework and soon renamed to be an enterprise architecture framework. The evolution of enterprise architecture happened because at that time system development had to address two core problems:

- System complexity—Organizations were spending more and more money building IT systems.

- Poor business alignment—Organizations were finding it more and more difficult to keep those increasingly expensive IT systems aligned with business need.

Enterprise: An organizational unit – from a department to a whole corporation.

Architecture: A formal description of a system, or a detailed plan of the system at component level to guide its implementation. The structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time.

Enterprise Architecture: It is a formal description of an enterprise, a detailed map of the enterprise at component level to guide its changes. It provides details regarding the structure of an enterprise's components, their inter-relationships, and the principles and guidelines governing their design and evolution over time. Enterprise Architecture is about understanding all of the different components that go to make up the enterprise and how those components inter-relate.

Enterprise application (EA) (popularly known as enterprise software or enterprise application software (EAS)) is applications that a business uses to support the organization in order to solve enterprise problems. Normally, EA denotes a software platform that is quite complex and comparatively huge for small business use. However, they are designed to be user friendly and easy to use. They are database-centric and demand high requirements in terms of security, user-access and maintenance.

It is a large software system platform intended to work in a corporate atmosphere like business or government. They are component-based, complex, scalable, distributed and mission critical. EA software comprises of a collection of programs with common business applications and organizational modeling. EAs are developed with enterprise architecture.

The design of EA is accomplished in such a way that it can be successfully deployed across a security requirements and administration management. Contrasting from the small business applications, which target particular business (e.g. billing software), an EA defines the business logic and supports the entire organization (including its departments), so as to lower costs and increase productivity as well as efficiency in the enterprise.

"At its heart, enterprise computing is all about combining separate applications, services and processes into a unified system that is greater than the sum of its parts."
- Jim Farley et al. in Java Enterprise in a Nutshell

An Enterprise Application is a combination of other applications, services and processes that, when unified, form a application that is greater than the sum of its parts."

- Paul Grenyer

Java is widely used for developing EA.
*It is because it is named after (or illustrated by) coffee!! (**Are you sure??)***
Java has a vast array of excellent libraries for solving most of the common problems one needs to solve when developing enterprise applications. In many cases, there is more than one good choice for addressing a particular need, and oftentimes those libraries are free and open source under a business-friendly license.
Many companies choose Java for developing EA because:

Learning Curve
Java is a simple language with a huge support base. Enterprise projects tend to involve large numbers of developers and it is much easier to get a developer to a minimum level of competence in Java than other languages like C++. A lot of university graduates have been largely schooled in Java.

Choice & Reuse
Java has a vast array of libraries, frameworks, tools and IDEs, and server providers. To an enterprise it's good to have choice, even if that's just for use as a bargaining tool when negotiating price. The language lends itself to code quality tools that let implementation of corporate standards (and as mentioned there are a lot of those tools).

Platform Independence
Java is writing once, run everywhere. Sun has actively encouraged open standards that allow multiple vendors to implement their solutions. These standards give the customer the comfort that they can migrate from one vendor to another if a given vendor goes under or starts charging more. Of course the reality is that each vendor does their best to provide some "added value" features that tie the customer to them quite nicely.

Maturity
It has been around a long time, running a lot of servers. If your web application needs to be "6 sigma" or similar and you are the Mega Corp Chief Tech Officer,

you are not going to look for developer wanting to do it in relatively immature platforms like RoR.

Timing/Marketing

Java was introduced to the world when programming was moving towards the web. It was positioned smartly and got a solid position early in web development. Because of the open standards, there are some very big companies producing these platforms and they market
Java pretty hard to sell those platforms.

Inertia

Large corporations update/upgrade themselves very slow (many are still using Java 1.4), so once they have picked Java, it takes a huge investment to migrate to another platform. And convincing for change is to assure them to invest a lot of money for no immediate business benefit.
To recapitulate, Java ensures security, stability, robustness, scalability, platform independence, performance & resources usage, and Academic society wide acceptance.

Common Types of Enterprise Applications

Some of the more common enterprise applications include the following:

- Customer relationship management (CRM)

Wikipedia says - CRM is a model for managing a company's interactions with current and future customers. It involves using technology to organize, automate, and synchronize sales, marketing, customer service and technical support. It involves all kinds of interaction that a business corporation has with its client, whether it is sales or service-related. CRM enables the business by: Understand the customer, Retain customers through better customer experience, Attract new customer, Win new clients and contracts, and Increase profitably, decrease customer management costs.

Fig: CRM<*http://crmconsulting.net.au*>

- Automated billing systems
- Enterprise Resource Planning
- HR Management

Enterprise Application Architecture (EAA)

Enterprise Architecture describes how organizational, information and technology structures support the strategy and operations of organizations. The EAA architecture of a business is an explanation of the structure of the application & software usage through the organization. The systems are decomposed into subsystems and connected/related to each other based on their interaction. The relationships with external environment, guidelines, users, terminology are also specified as a part of enterprise architecture of Software Systems. During the

designing of EAA, an effective and thorough survey of existing systems should be performed by a team of high technical analysts in order to understand these systems and their interactions with other applications and databases.

Advantage/Need/Uses of enterprise application architecture

The purpose of enterprise architecture is to optimize across the enterprise the often fragmented legacy of processes (both manual and automated) into an integrated environment that is responsive to change and supportive of the delivery of the business strategy. Thus the primary reason for developing an EA is to get an overview (map) of the business' processes, systems, technology, structures and capabilities. We need an EA to provide a strategic context for the evolution of the IT system in response to the constantly changing needs of the business environment. We need an EA to achieve competitive advantage.

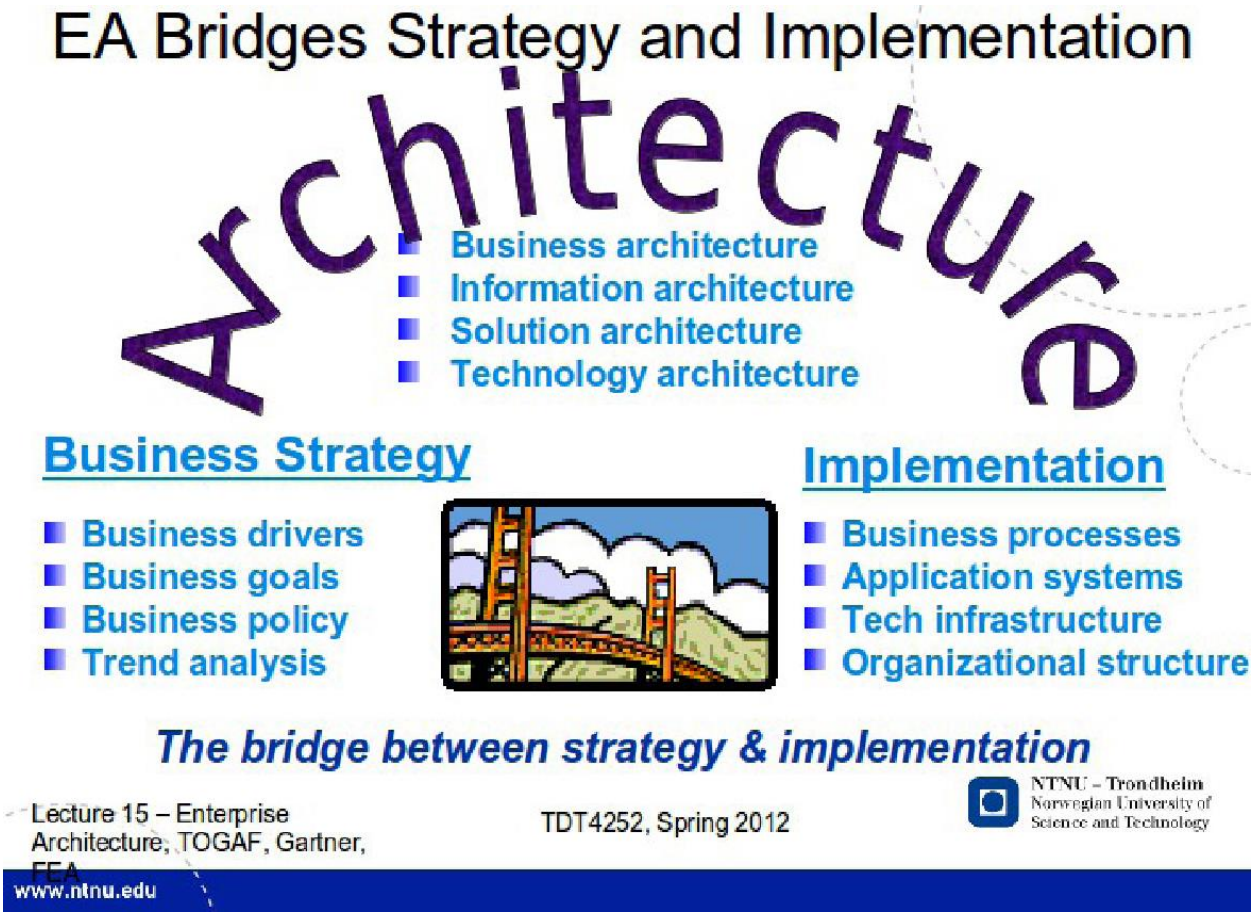
It acts for bridging the gap between Business and IT. It helps to enhance the relationships between IT and the business. It reinforces IT understanding of the business strategy. It creates a process for continuous IT/business alignment. Furthermore, it enhances IT agility to support business changes. And hence, create business value from IT.

Describing and characterizing the architecture of an enterprise application is beneficial as it acts like a central document. This central document specifies an unambiguous understanding of the complete system and how it works as a whole. This aids in finding problems and solving them in order to speed up existing systems and enhance them in terms of performance. At time of introduction of a new application software or improvement of existing applications, the application architecture acts a key document for studying the effects of the changes in the entire enterprise. This document should be referred as a guide during the improvement/replacement of the current systems. Whenever an enterprise application is introduced into an existing system, the application architecture acts as the foundation for developing this software.

The value of EA:

- Business - IT and business – business alignment
- Change enabler

- Improved agility to enable a real time enterprise
- Standardization, reuse and common principles, terms and work practices
- Integration and interoperability
- Structure, multiple perspectives and documentation



Desktop application runs on a single tier architecture. All the presentation logic, business logic and data storage and access logic resides in a single machine. Moreover, it can be a part of network. But the application and its database stay in the same machine. It does not support multiple users concurrently but only one user can access it at a time.

For beginners, web application and enterprise application seem to be similar although they are not. Enterprise application normally includes more than a single tier i.e. a multi-tier application. Multiple concurrent users can access the application.

For example Enterprise Application has at different tiers like Client, Presentation, and Business Logic and Data tier. Client Tier can be Java clients, browser-based clients and mobile clients.

Presentation Tier can be servlets, JavaBeans components and JSP components. Business Logic Tier can include servers, web services (SOAP, RESTful) ,etc. Data Tier can be of DBMS and LDAP. Web application can be implemented without such complex enterprise. For example we can use various technologies to implement a web application such as Servlets, JSP, Hibernate, Maven, Databases, JavaBeans, etc. However, web applications suffer shortcomings when special kind of infrastructure services (functional or non-functional) are required as an essential part of the application. There are various applications with such special service requirements which require a business layer running over an infrastructure offering special services like: timer services, distributed transactions, remote method invocation, messaging processing, etc. This kind of infrastructure services is not available in web servers. Hence, need arises for enterprise application that runs on a heavyweight application server (e.g. IBM WebSphere, Jboss and Geronimo).

Furthermore, Oracle website has put some clarifications regarding the standard edition of Java and Java Enterprise Edition. The major difference of Java EE from the standard edition is that Java EE requires an application server, instead of a web server like tomcat. The reason behind the need of application server is that, EJBs where the business logic resides needs a container. EJBs can be managed by application servers like Geronimo, IBM WebSphere and Jboss. These special application/containers can also handle web applications. Furthermore, an enterprise application holds enterprise beans (*an enterprise bean is a server-side component that encapsulates the business logic of an application*) and executes in a J2EE Container which caters security and transaction services to the beans.
The associated file is an .ear file. Whereas a web application runs in a web-container like tomcat, contains servlets/JSPs. The associated file is named with an extension of .war.

Enterprise Architecture Frameworks (EAF)

Frameworks help people organize and assess completeness of integrated models of their enterprises. An Architectural Framework gives a skeletal structure that

defines suggested architectural artifacts, describes how those artifacts are related to each other, and provides generic definitions for what those artifacts might look like. EAF says:

- How to create and use an enterprise architecture.
- Principles and practices for creating and using architectural description of the system.

Purpose of Framework

- Organize integrated models of an enterprise
- Assess completeness of the descriptive representation of an enterprise
- Understand an organization or a system
- Assist in identification and categorization
- Provide a communication mechanism
- Help manage complexity
- Identify the flow of money in the enterprise

Zachman Enterprise Architecture Framework

The **Zachman Framework** is an EAF which provides a formal and highly structured way of\ viewing and defining an enterprise. It consists of a two dimensional classification matrix based on the intersection of six communication questions (What, Where, When, Why, Who and
How) with five levels of reification, successively transforming the most abstract ideas (on the
Scope level) into more concrete ideas (at the Operations level). The Zachman Framework is a schema for organizing architectural artifacts (in other words, design documents, specifications, and models) that takes into account both whom the artifact targets (for example, business owner and builder) and what particular issue (for example, data and functionality) is being addressed. The Zachman Framework is not a methodology in that it does not imply any specific method or process for collecting, managing, or using the information that it describes

Fig:

	1. What (data)	2. How (function)	3. Where (network)	4. Who (people)	5. When (time)	6. Why (motivation)
1. Scope (context)						
2. Business model (concept)						
3. System model (logical)						
4. Technology model (physical)						
5. Detailed representation (component)						
6. Real system, i. e. executing the game of baseball						

Fig: An Empty Zachman Framework

The rows present:

- Different perspectives of the enterprise
- Different views of the enterprise
- Different roles in the enterprise

1. Scope describes the system's vision, mission, boundaries, architecture and constraints. The scope states what the system is to do. It is called a black box model, because we see the inputs and outputs, but the not the inner workings.

2. Business model shows goals, strategies and processes that are used to support the mission of the organization.

3. System model contains system requirements, objects, activities and functions that implement the business model. The system model states how the system is to perform its functions. It is called a white box model, because we see its inner workings.

4. Technology model considers the constraints of humans, tools, technology and materials.

5. Detailed representation presents individual, independent components that can be allocated to contractors for implementation.

6. Real system depicts the operational system under consideration.

Columns present the various aspects of the enterprise.

1. What (data) describes the entities involved in each perspective of the enterprise?
Examples include equipment, business objects and system data.

2. How (functions) shows the functions within each perspective.

3. Where (networks) shows locations and interconnections within the enterprise. This includes major business geographical locations, networks and the playing field.

4. Who (people) represents the people within the enterprise and metrics for assessing their capabilities and performance. The design of the enterprise organization has to do with the allocation of work and the structure of authority and responsibility.

5. When (time) represents time, or the event relationships that establish performance criteria. This is useful for designing schedules, the processing architecture, the control architecture and timing systems.

6. Why (motivation) describes the motivations of the enterprise. This reveals the enterprise goals, objectives, business plan, knowledge architecture, and reasons for thinking, doing things and making decisions.

Government Enterprise Architecture Frameworks (GEAFs)

In the context of government enterprises: a coordinated set of activity areas involving one or more public organizations and possibly third-party entities from

private organizations or civil society, an EA provides technical descriptions of the organizational goals, business and administrative processes, information requirements, supporting applications and technology infrastructure of the enterprise. These descriptions are typically captured in the form of models, diagrams, narratives, etc. A Government Enterprise Architecture (GEA) may be associated with a single agency or span functional areas transcending several organizational boundaries, e.g. health care, financial management and social welfare.

Reasons for developing enterprise architectures in government include:

1. Understanding, clarifying and optimizing the inter-dependencies and relationships among business operations, the underlying IT infrastructure and applications that support these operations in government agencies and in the context of specific government enterprises.

2. Establishing a basis for agencies to share information, knowledge and technology and other resources or jointly participate in the execution of business processes

3. Optimizing ICT investment and business cases across the whole of government by enabling the opportunities for collaboration and sharing of assets, thus reducing the tendency for duplicated and poorly integrated IT resources and capabilities. There is increasing awareness on the importance of EA as most of the leading countries in e-government have well established EA programs. Presently, there are EA maturity models with defined relations to well-known e-Government Maturity stages. The increasing popularity of EA practices by governments in both developed and developing countries is indicated by the different global surveys on EA.

Despite the popularity of the EA practice in the private sector and increasingly in the government, the EA discipline is relatively new, lacking foundational theories and models and characterized by multiplicity of frameworks and reference models; even lacking an agreement on the definition and scope of the subject matter. From the earlier orientation of EA as a technological optimization or standardization concern, EA has gradually evolved to a management practice with stronger emphasis placed on the organizational-IT alignment.

However, there are mixed results in terms of outcomes from EA initiatives. In general, demonstrating concrete benefits from EA program has been challenging for many organizations. This difficulty is attributed to lack of metrics for EA initiatives. Notwithstanding, a number of successful EA initiatives have been reported by some governments, particularly, in Canada and the US. Unfortunately, comparing and analyzing these EA initiatives and cases is difficult in the absence of assessment frameworks, techniques and tools.

1. Aim

The project aims to provide policy guidelines for the development of Government Enterprise
Architecture Frameworks (GEAFs), establish concrete requirements for such a framework in
Macao, and provide recommendations on how elements of a Macao GEAF (MGEAF) could be built from existing Government EA Frameworks, Reference Models, Methods, and
Modeling Framework. The project will also provide an example of agency-specific EA based on the recommended Macao framework.

2. Objectives

- Improving understanding and contributing to the body of knowledge of GEA through foundational research
- 2. Enhancing EA practice by providing policy guidelines and development of Government EA Frameworks based on results from (1) with the supporting toolkit
- 3. Understanding the factors that contribute to wide adoption of EA practice within a government
- 4. Building capacities of government agencies and their architects through development of courseware for educational and training purposes as well as the use of tools in (2)
- Dissemination of project output (1 through 4) through various channels including publications (books, journals, conference papers and technical reports), schools and courses, seminars, workshops and projects.

Chapter 2: Development Process Management

2.1 Source Code Management

Source Code Management/Version Control

The Problem:
In a typical software development environment, many developers will be engaged in work on one code base. If everyone was to be allowed to edit and modify any item whenever they felt, it ends into chaos. How do you handle the source code?

The Hard way:
You designate a guy to maintain the golden copy. As you finish code you give it to him, and he places it in a directory.

– Forces a guy to do this manually
– What if two people is working on the same file?
– What if you want to go back to an older version of a file?
– What if you want to maintain multiple "forks" of the program?

Instead of suffering under such an environment, most developers prefer to implement the *Source Code Management (SCM)* Systems or *Version Control* Tools.

Source Code Management (SCM):
These are the problems source code management is intended to solve. Effectively it is a database for source code that eases the problems of
– Multiple people in many places working on the same code
– Retrieving old versions of files
– Keeping logs about what changed in a file
– Integrating changed code
– Generating release builds

What it is NOT:
It can do a lot of things for you, but it does not try to be everything for everyone. It is not,
– A replacement for communication
– A replacement for management
– A substitute for other tools, such as bug tracking, mailing lists, and distribution
– It has no understanding of the semantics of your program; as far as it's concerned, it's all just text.

Version Control Tools:
There are many Version Control Tools.
1. Project Revision Control System (PRCS)
2. Source Code Control System (SCCS)
3. Revision Control System (RCS)
4. Concurrent Version System (CVS)

Project Revision Control System (PRCS):
The Project Revision Control System (PRCS) is the front end to a set of tools that (like CVS). It provides a way to deal with sets of files and directories as an entity, preserving coherent versions of the entire set. Its purpose is similar to that of SCCS, RCS, and CVS, but (according to its authors, at least) it is much simpler than any of those systems.

Source Code Control System (SCCS):
The Source Code Control System (SCCS) was developed at Bell Telephone Laboratories in 1972. Though SCCS lacks some of the amenities of RCS (a natural result of its early date of development), it is a generally equivalent system and has a few capabilities that RCS does not.

Revision Control System (RCS):
The Revision Control System (RCS) was designed at the Department of Computer Science at Purdue University in 1982. RCS is a software tool for UNIX systems which lets people working on the system control "multiple revisions of text ... that is revised frequently, such as programs or documentation." It can be applied to development situations of all sorts, including the creation of documents, drawings, forms, articles, and of course, source code.

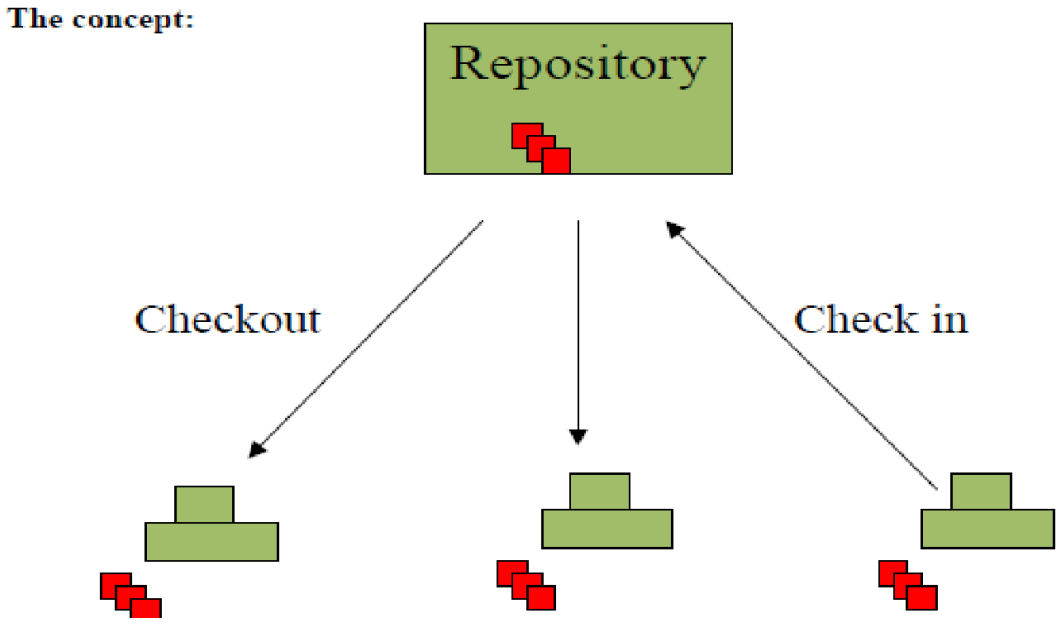
Concurrent Version System (CVS):
CVS is an open source version control system layered on top of RCS, designed to manage entire software projects. It is an important component of Source Configuration Management (SCM).

Basics description of the CVS system:
It works by holding a central `repository' of the most recent version of the files.
You may at any time create a personal copy of these files by `checking out' the files from the repository into one of your directories. If at a later date newer versions of the files are put in the repository, you can `update' your copy. You may edit your copy of the files freely. When you are satisfied with the changes you have made in your copy of the files, you can `commit' them into the central repository. When you are finally done with your personal copy of the files, you can
`Release' them and then remove them.

The concept:

You need two pieces of software to do CVS: a server and a client. The server handles the database end, the client the local side.

Server Side: Repository Directory; Client Side: Working/Local Directory



Some basic words and descriptions:

Versions, revisions and releases:

A file can have several versions; likewise, a software product can have several versions. A software product is often given a version number such as `2.1.1'.

Versions in the first sense are called *revisions* here, and versions in the second sense are called *releases*. To avoid confusion, the word *version* is almost avoided.

Repository:

The Repository is the directory, which stores the master copies of the files.

The main or master repository is a tree of directories.

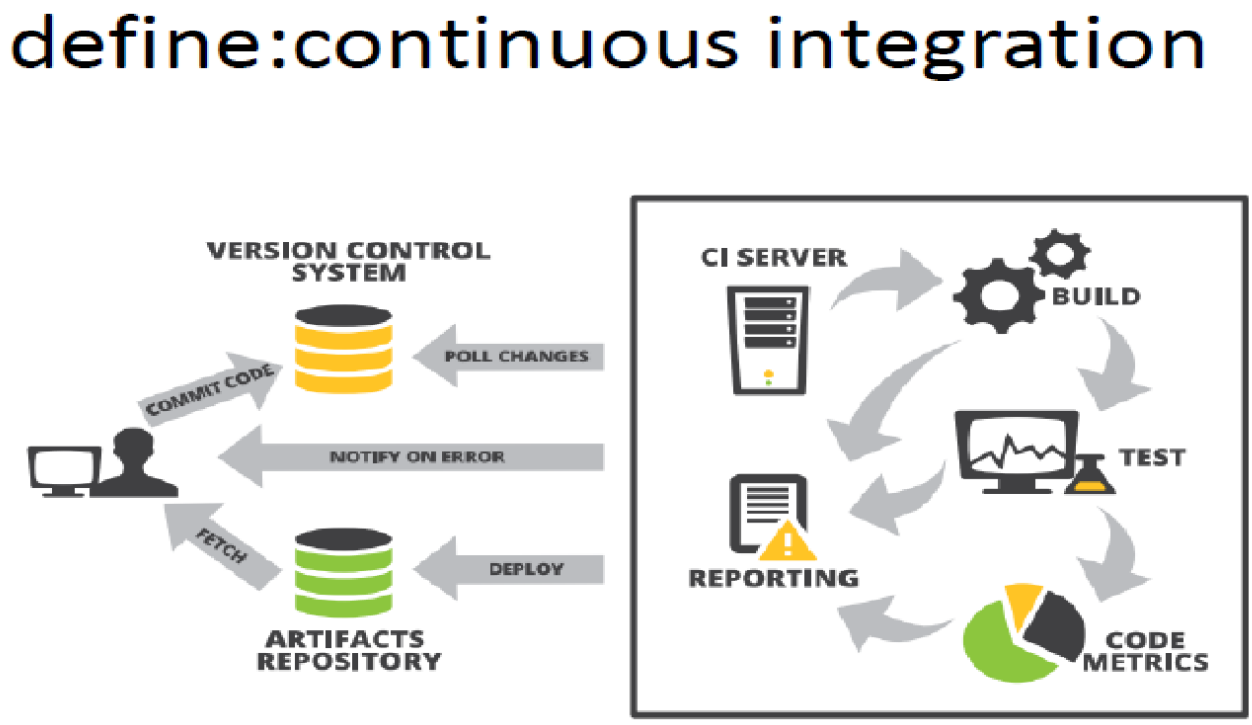
Module:

It is a specific directory (or mini-tree of directories) in the main repository.

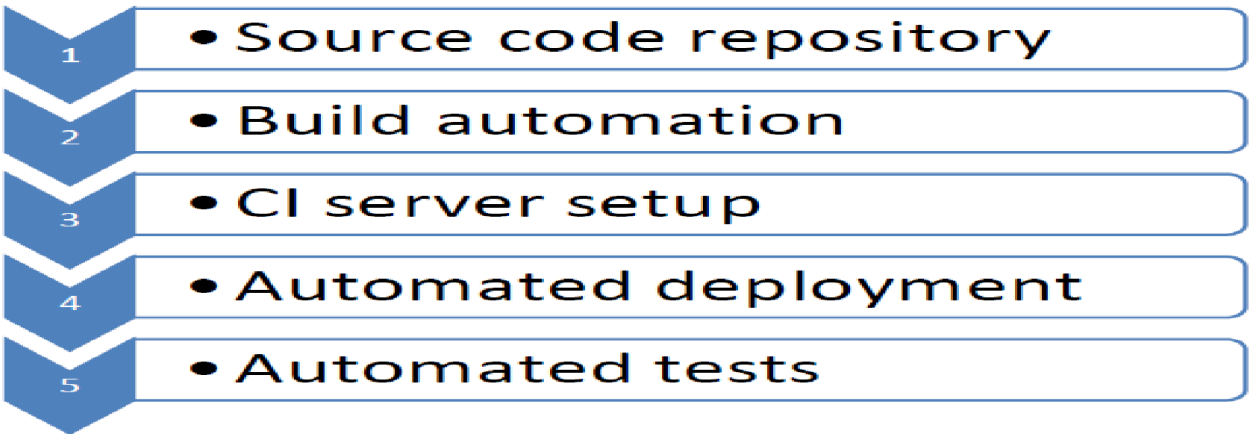
Modules are defined in the CVS modules file.

2.2 Continuous Integration

Continuous Integration is a software development practice of performing software integration frequently…several times a day, in fact. Ideally, your software application or system should be built automatically after each commit into a shared version control repository. Upon each successful build, the system integrity should be verified using automated tests that cover if not all, then at least most of the functionality. If some tests fail, the developer responsible is notified instantly and the problem can be identified and solved quickly. Using this approach, you can deliver working and reliable code to the customer much faster, while also mitigating the risk of releasing unstable, buggy software to your users.



5 steps to continuous integration



Step 1: Source code repository

•Keeps track of every change in your codebase
•Using homegrown solution can be challenging in a continuous integration environment
•Lots of tools on the market: CVS, Subversion, Mercurial, Perforce, BitKeeper, Roundtable…
•If you can generate something, don't store it in your SCM
–*It will be part of your build script*
• Separate requirements and dependencies
–*OpenEdge is a requirement, pdf_include is a dependency*
•Don't forget database versioning
•Commit as much as possible, using branches if necessary
•Associate a bug tracker to your SCM

Step 2: Build automation

•Architect / Developer Studio / AppBuilder **ARE NOT** build tools
•Just after checking out source code, a single command line should be enough to generate a standalone binary
•Common build tools are: Makefiles, shell scripts, Ant, Maven, Gradle…
•Looking for something stable, flexible and portable across platforms? Use Ant!

•Ant is an open-source product to deal with software builds
•XML based syntax, and provides lots of standard tasks
•PCT (Progress Compilation Toolkit) is an open-source extension to deal with the OpenEdge environment

Step 3: CI server setup

•Many products, but same functionalities
–Define and trigger jobs
–Store deliverables (and keep history)
–Make them easily accessible
–Keep users informed of build result

•Choosing a CI server:
–Free or not
–Integration with your tools
–Plugins

•Always use a clean server
•Use distributed jobs
•Define a job for every product and every environment (branches / clones)
•Keep deliverables only for production jobs. Keep only a dozen for integration
•Only send alerts for failures!

Step 4: Automate deployment

•Use only what has been generated during the build process
•Always deploy to a clean server
•Every deliverable generated by the CI server should be deployed
•Available in VMWare ESX
•Clones, snapshots and remote execution are your friends!
•Define clean VM for every target OS
•Define snapshots to be able to improve configuration
•New clone for a new job
•Remote execution to execute deployment

Step 5: tests

•Unit testing
•User interface testing
•API testing
•Regression testing
•Load testing
•Security testing

2.3 Software Testing

What is testing?

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. In simple words, testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.
According to ANSI/IEEE 1059 standard, Testing can be defined as - *A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item*.

When to Start Testing?

An early start to testing reduces the cost and time to rework and produce error-free software that is delivered to the client. However in Software Development Life Cycle (SDLC), testing can be started from the Requirements Gathering phase and continued till the deployment of the software. It also depends on the development model that is being used. For example, in the Waterfall model, formal testing is conducted in the testing phase; but in the incremental model, testing is performed at the end of every increment/iteration and the whole application is tested at the end.

Testing is done in different forms at every phase of SDLC:

- During the requirement gathering phase, the analysis and verification of requirements are also considered as testing.
- Reviewing the design in the design phase with the intent to improve the design is also considered as testing.

- Testing performed by a developer on completion of the code is also categorized as testing.

Verification & Validation

These two terms are very confusing for most people, who use them interchangeably. The following table highlights the differences between verification and validation.

S.N	Verification	Validation
1	Verification addresses the concern: "Are you building it right?"	Validation addresses the concern: "Are you building the right thing?"
2	Ensures that the software system meets all the functionality.	Ensures that the functionalities meet the intended behavior.
3	Verification takes place first and includes the checking for documentation, code, etc.	Validation occurs after verification and mainly involves the checking of the overall product.
4	Done by developers.	Done by testers.
5	It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify a software.	It has dynamic activities, as it includes executing the software against the requirements.
6	It is an objective process and no subjective decision should be needed to verify a software.	It is a subjective process and involves subjective decisions on how well a software works.

Testing and Debugging

Testing: It involves identifying bug/error/defect in a software without correcting it. Normally professionals with a quality assurance background are involved in bugs' identification. Testing is performed in the testing phase.

Debugging: It involves identifying, isolating, and fixing the problems/bugs. Developers who code the software conduct debugging upon encountering an error in the code. Debugging is a part of White Box Testing or Unit Testing. Debugging can be performed in the development phase while conducting Unit Testing or in phases while fixing the reported bugs.

Types of Testing:
There are different types of testing that may be used to test a software during SDLC.

Manual Testing

Manual testing includes testing a software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing.
Testers use test plans, test cases, or test scenarios to test a software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.

Automation Testing

Automation testing, which is also known as *Test Automation*, is when the tester writes scripts and uses another software to test the product. This process involves automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly. Apart from regression testing, automation testing is also used to test the application from load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money in comparison to manual testing.

Software Testing Tools

The following tools can be used for automation testing:

- HP Quick Test Professional, Selenium, IBM Rational Functional Tester, SilkTest , TestComplete, Testing Anywhere , WinRunner , LaodRunner , Visual Studio Test Professional , WATIR

Testing Methods:

There are different methods that can be used for software testing. This chapter briefly describes the methods available.

Black-Box Testing:

The technique of testing without having any knowledge of the interior workings of the application is called black-box testing. The tester is oblivious to the system architecture and does not have access to the source code. Typically, while performing a black-box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.

The following table lists the advantages and disadvantages of black-box testing.

Advantages	Disadvantages
● Well suited and efficient for large code segments. ● Code access is not required. ● Clearly separates user's perspective from the developer's perspective through visibly defined roles. ● Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.	● Limited coverage, since only a selected number of test scenarios is actually performed. ● Inefficient testing, due to the fact that the tester only has limited knowledge about an application. ● Blind coverage, since the tester cannot target specific code segments or error-prone areas. ● The test cases are difficult to design.

White-Box Testing:

White-box testing is the detailed investigation of internal logic and structure of the code. White-box testing is also called **glass testing** or **open-box testing**. In order to perform **white-box** testing on an application, a tester needs to know the internal workings of the code. The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

The following table lists the advantages and disadvantages of white-box testing.

Advantages	Disadvantages
● As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively. ● It helps in optimizing the code. ● Extra lines of code can be removed which can bring in hidden defects. ● Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	● Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased. ● Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested. ● It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.

Grey-Box Testing

Grey-box testing is a technique to test the application with having a limited knowledge of the internal workings of an application. In software testing, the

phrase *the more you know, the better* carries a lot of weight while testing an application. Mastering the domain of a system always gives the tester an edge over someone with limited domain knowledge. Unlike black-box testing, where the tester only tests the application's user interface; in grey-box testing, the tester has access to design documents and the database. Having this knowledge, a tester can prepare better test data and test scenarios while making a test plan.

Advantages	Disadvantages
● Offers combined benefits of black-box and white-box testing wherever possible. ● Grey box testers don't rely on the source code; instead they rely on interface definition and functional specifications. ● Based on the limited information available, a grey-box tester can design excellent test scenarios especially around communication protocols and data type handling. ● The test is done from the point of view of the user and not the designer.	● Since the access to source code is not available, the ability to go over the code and test coverage is limited. ● The tests can be redundant if the software designer has already run a test case. ● Testing every possible input stream is unrealistic because it would take an unreasonable amount of time; therefore, many program paths will go untested.

A Comparison of Testing Methods

The following table lists the points that differentiate black-box testing, grey-box testing, and
White-box testing.

Black-Box Testing	Grey-Box Testing	White-Box Testing
1. The internal workings of an application need not be known.	1. The tester has limited knowledge of the internal workings of the	1. Tester has full knowledge of the internal workings of the

2. Also known as closed-box testing, data-driven testing, or functional testing. 3. Performed by end-users and also by testers and developers. 4. Testing is based on external expectations - Internal behavior of the application is unknown. 5. It is exhaustive and the least time-consuming. 6. Not suited for algorithm testing. 7. This can only be done by trial-and-error method.	application. 2. Also known as translucent testing, as the tester has limited knowledge of the insides of the application. 3. Performed by end-users and also by testers and developers. 4. Testing is done on the basis of high-level database diagrams and data flow diagrams. 5. Partly time-consuming and exhaustive. 6. Not suited for algorithm testing. 7. Data domains and internal boundaries can be tested, if known.	application. 2. Also known as clear-box testing, structural testing, or code-based testing. 3. Normally done by testers and developers. 4. Internal workings are fully known and the tester can design test data accordingly. 5. The most exhaustive and time-consuming type of testing. 6. Suited for algorithm testing. 7. Data domains and internal boundaries can be better tested.

TESTING LEVELS:

There are different levels during the process of testing. Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are:

- Functional Testing
- Non-functional Testing

Functional Testing

This is a type of black-box testing that is based on the specifications of the software that is to be tested. The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for. Functional testing of a software is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. There are five steps that are involved while testing an application for functionality.

Steps	Description
i.	The determination of the functionality that the intended application is meant to perform.
ii.	The creation of test data based on the specifications of the application.
iii.	The output based on the test data and the specifications of the application.
iv.	The writing of test scenarios and the execution of test cases.
V.	The comparison of actual and expected results based on the executed test cases.

Unit Testing

This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team. The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

Limitations of Unit Testing

Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.
There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing

Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in two ways: Bottom-up integration testing and Top-down integration testing. In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

System Testing

System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. This type of testing is performed by a specialized testing team.
System testing is important because of the following reasons:

- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
- The application is tested thoroughly to verify that it meets the functional and technical specifications.
- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

Regression Testing

Whenever a change in a software application is made, it is quite possible that other areas within the application have been affected by this change. Regression testing is performed to verify that a fixed bug hasn't resulted in another functionality or business rule violation. The intent of regression testing is to ensure that a change, such as a bug fix should not result in another fault being uncovered in the application.

Regression testing is important because of the following reasons:

- Minimize the gaps in testing when an application with changes made has to be tested.
- Testing the new changes to verify that the changes made did not affect any other area of the application.
- Mitigates risks when regression testing is performed on the application.
- Test coverage is increased without compromising timelines.
- Increase speed to market the product.

Acceptance Testing

This is arguably the most important type of testing, as it is conducted by the Quality Assurance Team who will gauge whether the application meets the intended specifications and satisfies the client's requirement. The QA team will have a set of pre-written scenarios and test cases that will be used to test the application.

Acceptance tests are not only intended to point out simple spelling mistakes, cosmetic errors, or interface gaps, but also to point out any bugs in the application that will result in system crashes or major errors in the application. By performing acceptance tests on an application, the testing team will deduce how the application will perform in production. There are also legal and contractual requirements for acceptance of the system.

Alpha Testing

This test is the first stage of testing and will be performed amongst the teams (developer and QA teams). Unit testing, integration testing and system testing when combined together is known as alpha testing. During this phase, the following aspects will be tested in the application:

- Spelling Mistakes
- Broken Links
- Cloudy Directions
- The Application will be tested on machines with the lowest specification to test loading times and any latency problems.

Beta Testing

This test is performed after alpha testing has been successfully performed. In beta testing, a sample of the intended audience tests the application. Beta testing is also known as **pre-release testing**. Beta test versions of software are ideally distributed to a wide audience on the Web, partly to give the program a "real-world" test and partly to provide a preview of the next release.

In this phase, the audience will be testing the following:
- Users will install, run the application and send their feedback to the project team.
- Typographical errors, confusing application flow, and even crashes.
- Getting the feedback, the project team can fix the problems before releasing the software to the actual users.
- The more issues you fix that solve real user problems, the higher the quality of your application will be.
- Having a higher-quality application when you release it to the general public will increase customer satisfaction.

Non-Functional Testing

This section is based upon testing an application from its non-functional attributes. Non-functional testing involves testing a software from the requirements which are nonfunctional in nature but important such as performance, security, user interface, etc. Some of the important and commonly used non-functional testing types are discussed below.

Performance Testing

It is mostly used to identify any bottlenecks or performance issues rather than finding bugs in a software. There are different causes that contribute in lowering the performance of a software:

- Network delay
- Client-side processing
- Database transaction processing
- Load balancing between servers
- Data rendering

Performance testing is considered as one of the important and mandatory testing type in terms of the following aspects:

- Speed (i.e. Response Time, data rendering and accessing)
- Capacity
- Stability
- Scalability

Performance testing can be either qualitative or quantitative and can be divided into different sub-types such as **Load testing** and **Stress testing**.

Load Testing

It is a process of testing the behavior of a software by applying maximum load in terms of software accessing and manipulating large input data. It can be done at both normal and peak load conditions. This type of testing identifies the maximum capacity of software and its behavior at peak time.

Stress Testing

Stress testing includes testing the behavior of a software under abnormal conditions. For example, it may include taking away some resources or applying a load beyond the actual load limit. The aim of stress testing is to test the software by applying the load to the system and taking over the resources used by the software to identify the breaking point. This testing can be performed by testing different scenarios such as:

- Shutdown or restart of network ports randomly
- Turning the database on or off

- Running different processes that consume resources such as CPU, memory, server, etc.

Usability Testing

Usability testing is a black-box technique and is used to identify any error(s) and improvements in the software by observing the users through their usage and operation.
Molich in 2000 stated that a user-friendly system should fulfill the following five goals, i.e., easy to Learn, easy to remember, efficient to use, satisfactory to use, and easy to understand.
In addition to the different definitions of usability, there are some standards and quality models and methods that define usability in the form of attributes and sub-attributes such as ISO-9126, ISO-9241-11, ISO-13407, and IEEE std.610.12, etc.

UI vs Usability Testing

UI testing involves testing the Graphical User Interface of the Software. UI testing ensures that the GUI functions according to the requirements and tested in terms of color, alignment, size, and other properties.
On the other hand, usability testing ensures a good and user-friendly GUI that can be easily handled. UI testing can be considered as a sub-part of usability testing.

Security Testing

Security testing involves testing a software in order to identify any flaws and gaps from security and vulnerability point of view. Listed below are the main aspects that security testing should ensure:

- Confidentiality
- Integrity
- Authentication
- Availability
- Authorization
- Non-repudiation
- Software is secure against known and unknown vulnerabilities
- Software data is secure

- Software is according to all security regulations
- Input checking and validation
- SQL insertion attacks
- Injection flaws
- Session management issues
- Cross-site scripting attacks
- Buffer overflows vulnerabilities
- Directory traversal attacks

Portability Testing

Portability testing includes testing a software with the aim to ensure its reusability and that it can be moved from another software as well. Following are the strategies that can be used for portability testing:

- Transferring an installed software from one computer to another.
- Building executable (.exe) to run the software on different platforms.

Portability testing can be considered as one of the sub-parts of system testing, as this testing type includes overall testing of a software with respect to its usage over different environments. Computer hardware, operating systems, and browsers are the major focus of portability testing.

Some of the pre-conditions for portability testing are as follows:

- Software should be designed and coded, keeping in mind the portability requirements.
- Unit testing has been performed on the associated components.
- Integration testing has been performed.
- Test environment has been established.

Software Documentation (Using UML)

UML: UML stands for Unified Modeling Language which is used in object oriented software engineering. Although typically used in software engineering, it is a rich language that can be used to model an application structure, behavior a even business processes.

The Complete Guide to UML Diagram Types with Examples

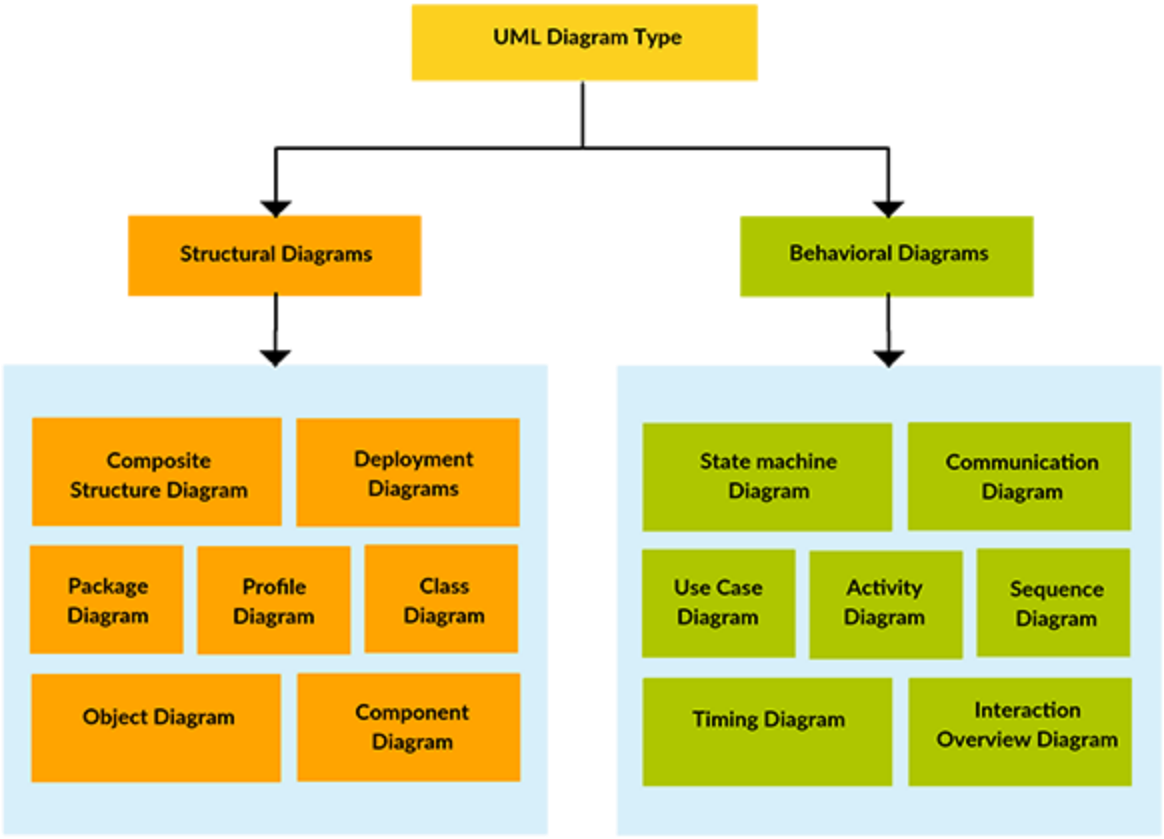
UML stands for Unified Modeling Language which is used in object oriented software engineering. Although typically used in software engineering it is a rich language that can be used to model an application structures, behavior and even business processes.

There are **14 UML diagram types** to help you model these behavior. They can be divided into two main categories; structure diagrams and behavioral diagrams. All 14 UML diagram types are listed below with examples and a brief introduction to them explaining how they are used when modeling applications.

List of UML Diagram Types

Types of UML diagrams with structure diagrams coming first and behavioral diagrams starting from position 8. Click on any diagram type to visit that specific diagram type's description.

1. Class Diagram
2. Component Diagram
3. Deployment Diagram
4. Object Diagram
5. Package Diagram
6. Profile Diagram
7. Composite Structure Diagram
8. Use Case Diagram
9. Activity Diagram
10. State Machine Diagram
11. Sequence Diagram
12. Communication Diagram
13. Interaction Overview Diagram
14. Timing Diagram



UML Diagram types

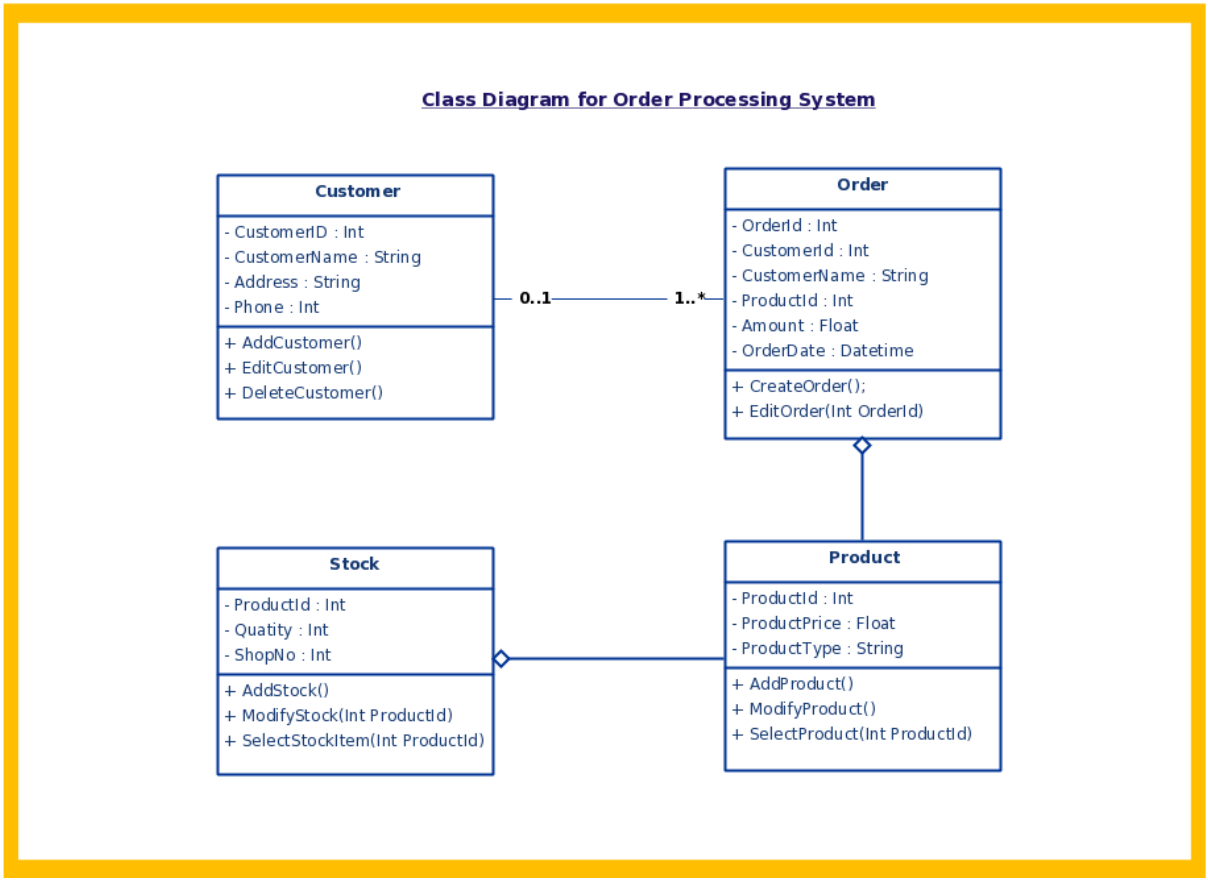
Structure diagrams show the things in a system being modeled. In a more technical term, they show different objects in a system. **Behavioral diagrams** shows what should happen in a system. They describe how the objects interact with each other to create a functioning system.

Class Diagram

Class diagrams are arguably the most used UML diagram type. It is the main building block of any object oriented solution. It shows the classes in a system, attributes and operations of each class and the relationship between each class.

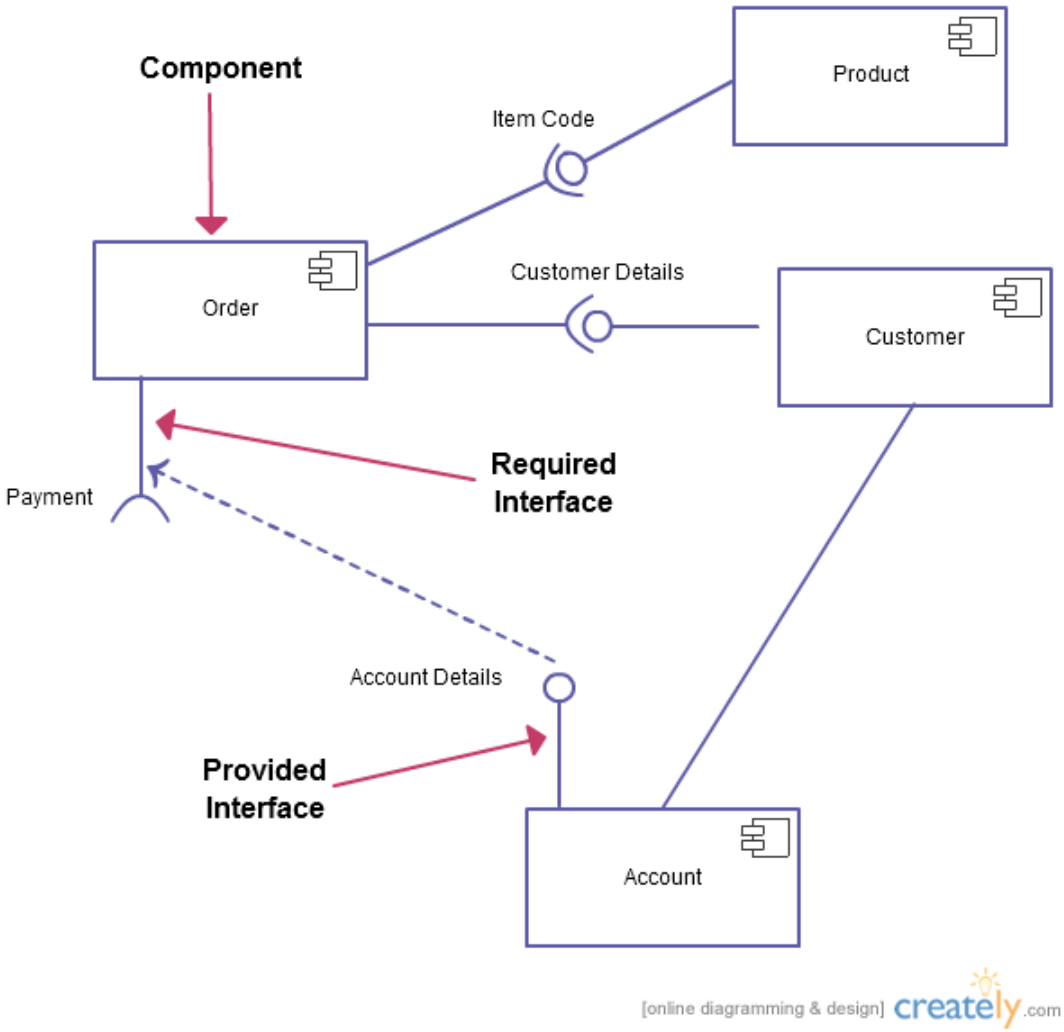
In most modeling tools, a class has three parts, name at the top, attributes in the middle and operations or methods at the bottom. In large systems with many related classes, classes are grouped together to create class diagrams. Different relationships between classes are shown by different types of arrows.

Below is an image of a class diagram.



Component Diagram

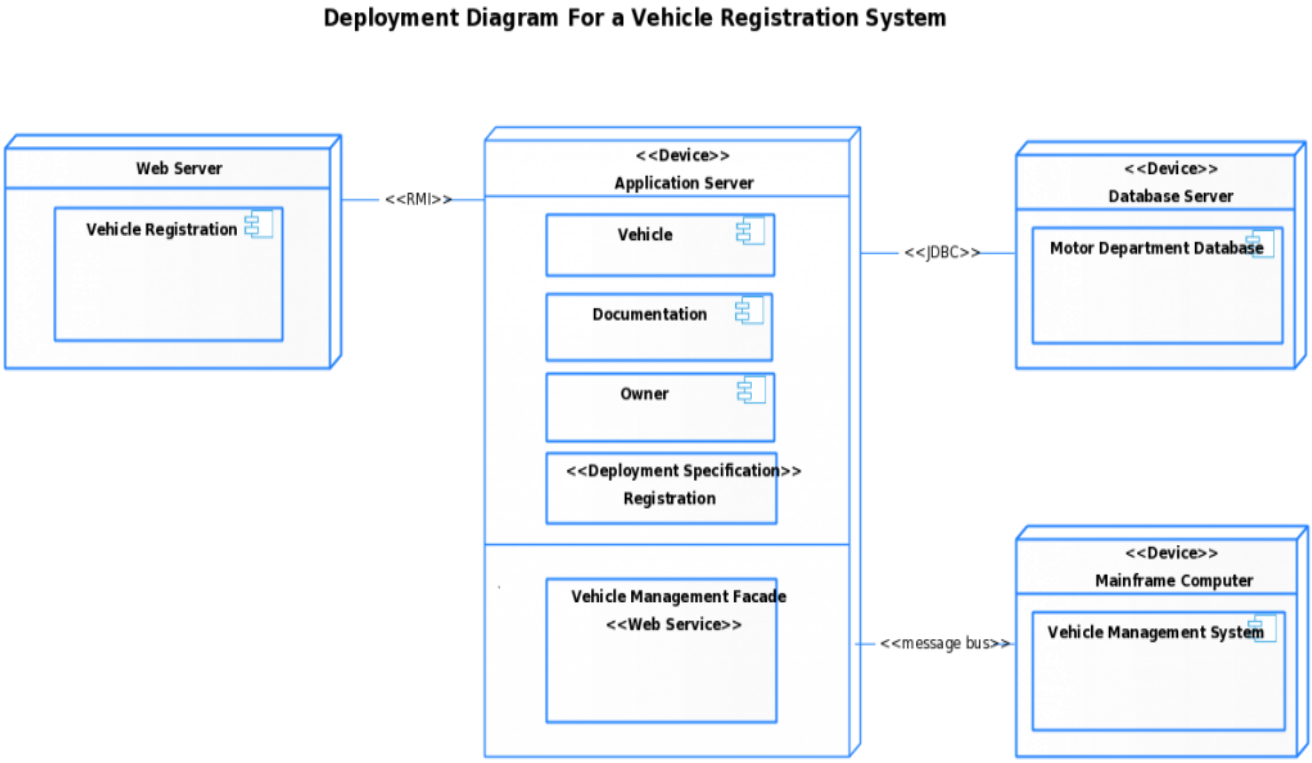
A component diagram displays the structural relationship of components of a software system. These are mostly used when working with complex systems that have many components. Components communicate with each other using **interfaces**. The interfaces are linked using connectors. The images below shows a component diagram.



Simple Component Diagram with Interfaces

Deployment Diagram

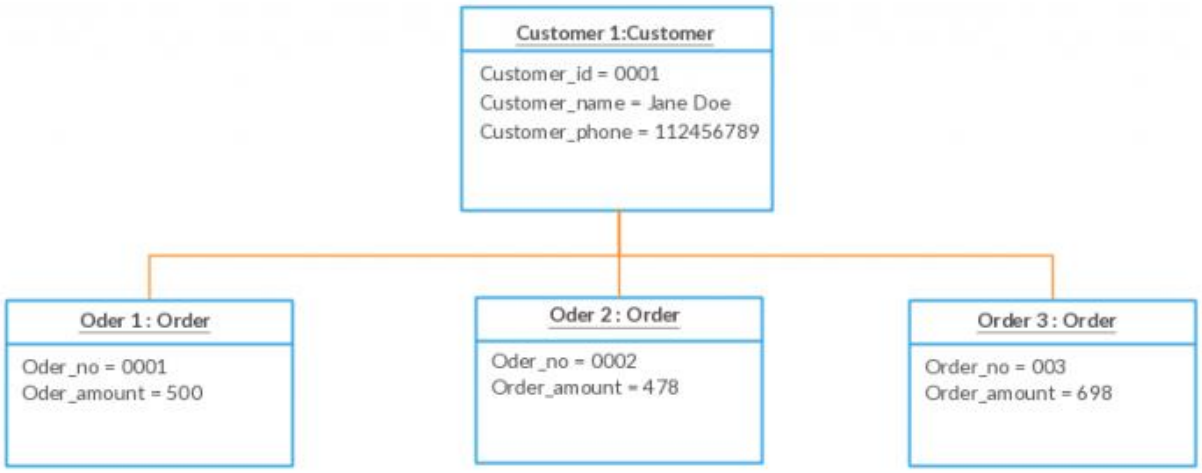
A deployment diagram shows the hardware of your system and the software in those hardware. Deployment diagrams are useful when your software solution is deployed across multiple machines with each having a unique configuration. Below is an example deployment diagram.



UML Deployment Diagram

Object Diagram

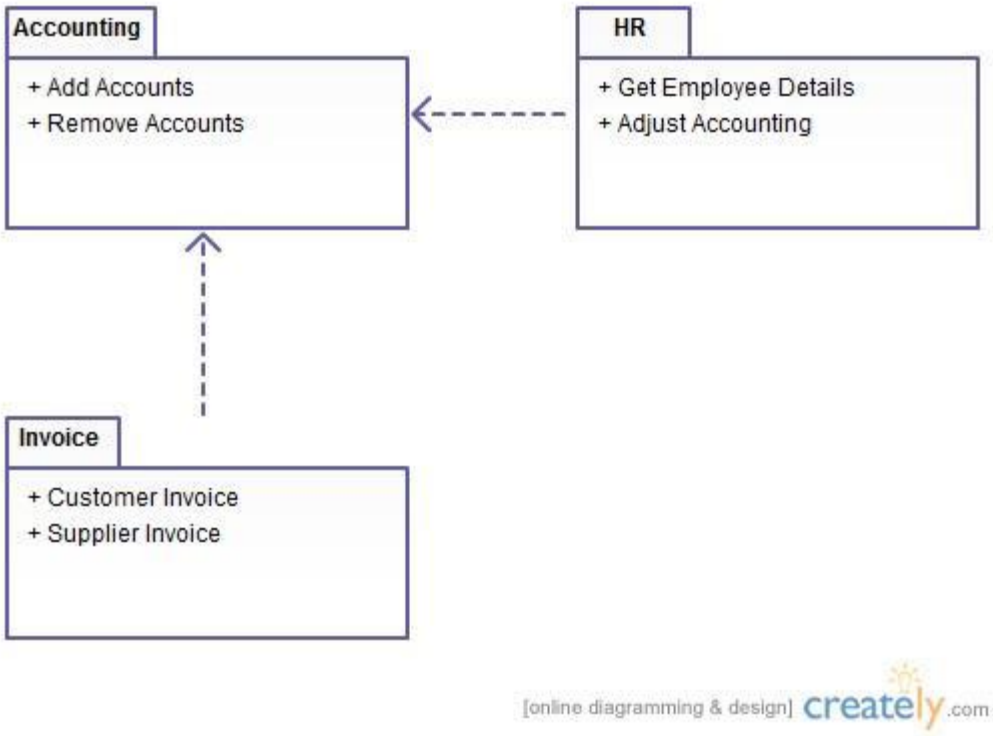
Object Diagrams, sometimes referred to as Instance diagrams are very similar to class diagrams. Like class diagrams, they also show the relationship between objects but they use real world examples. They are used to show how a system will look like at a given time. Because there is data available in the objects, they are often used to explain complex relationships between objects.



UML Object Diagram Example

Package Diagram

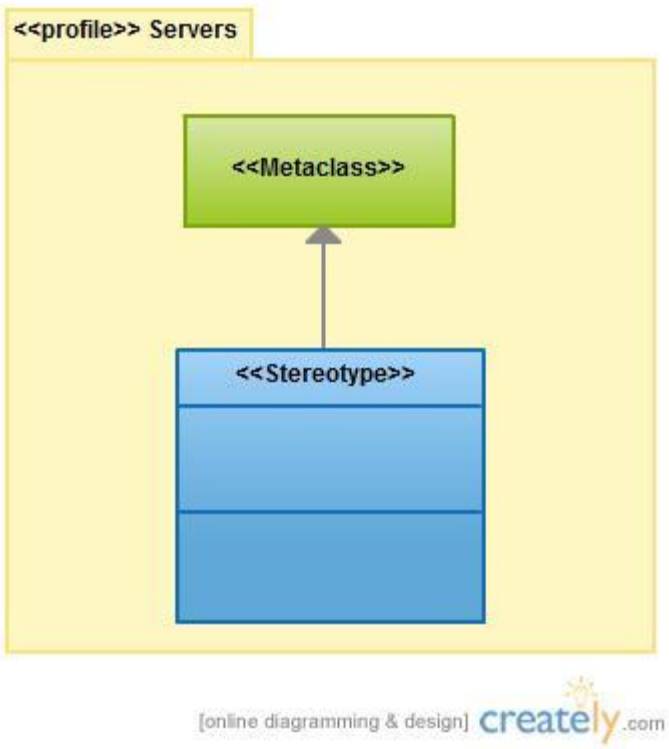
As the name suggests, a package diagram shows the dependencies between different packages in a system.



Package Diagram in UML

Profile Diagram

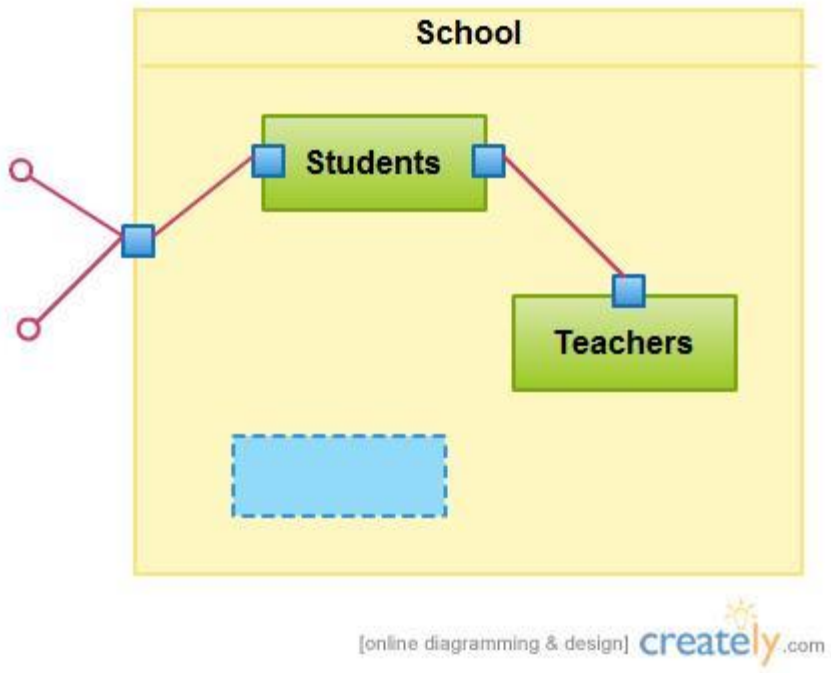
Profile diagram is a new diagram type introduced in UML 2. This is a diagram type that is very rarely used in any specification.



Basic UML Profile Diagram structure

Composite Structure Diagram

Composite structure diagrams are used to show the internal structure of a class.

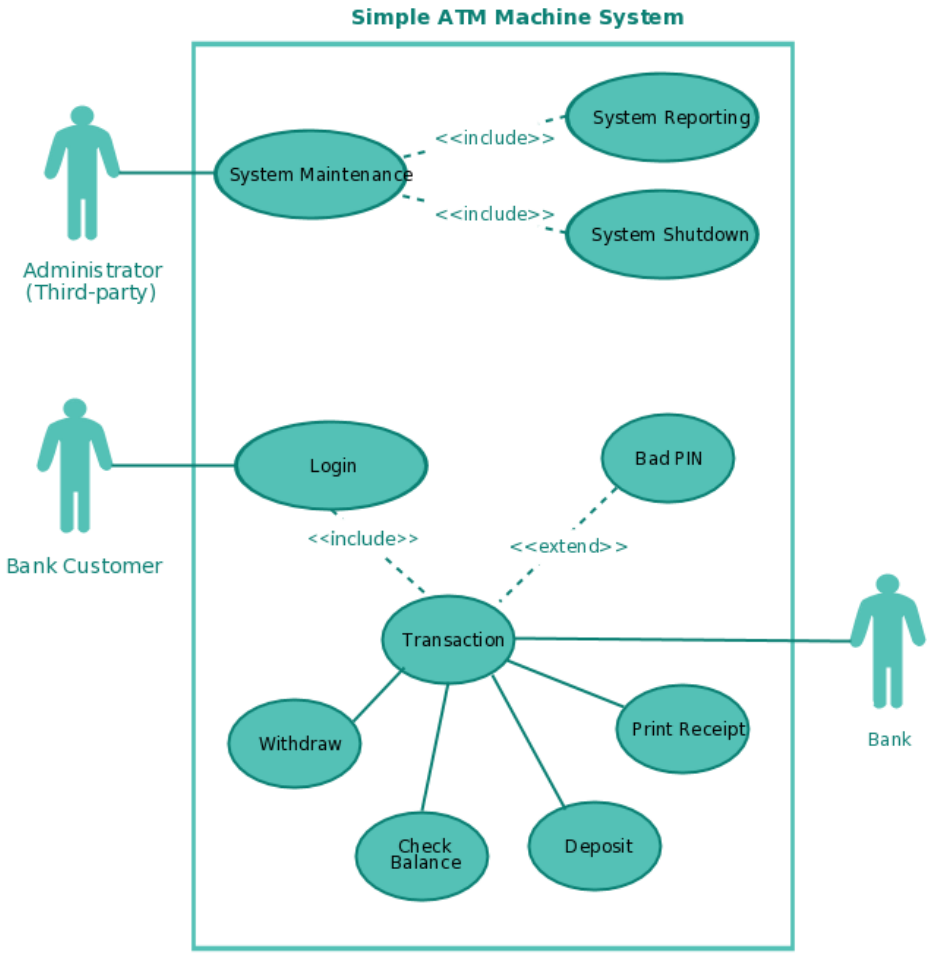


A simple Composite Structure Diagram

Use Case Diagram

As the most known diagram type of the behavioral UML diagrams, Use case diagrams give a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions are interacted.

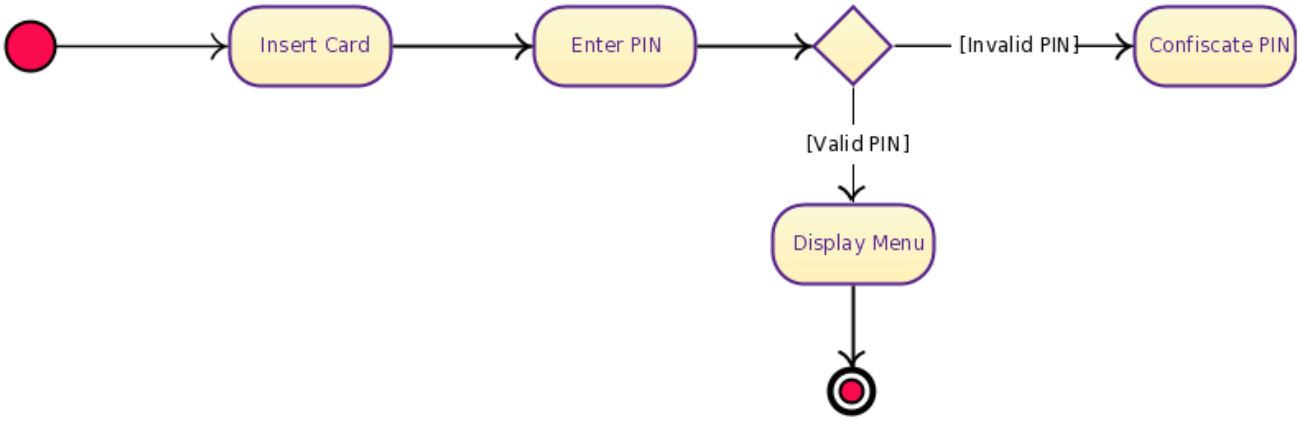
It's a great starting point for any project discussion, because you can easily identify the main actors involved and the main processes of the system.



Use Case diagram showing Actors and main processes

Activity Diagram

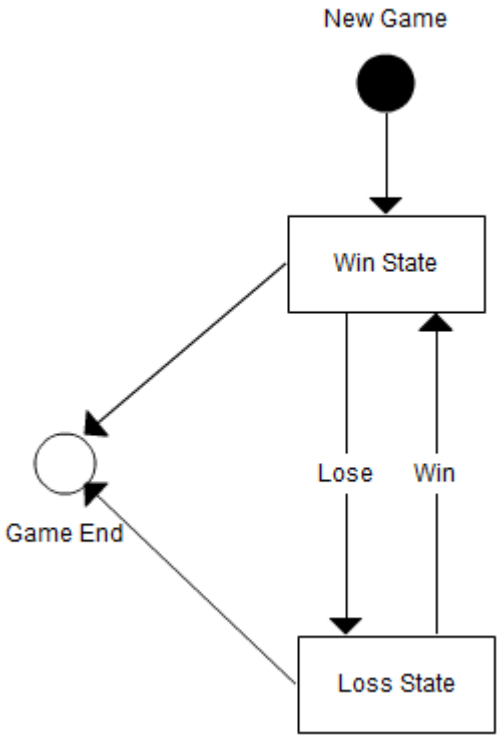
Activity diagrams represent workflows in a graphical way. They can be used to describe business workflow or the operational workflow of any component in a system. Sometimes activity diagrams are used as an alternative to State machine diagrams.



Activity Diagrams with start, end, processes and decision points

State Machine Diagram

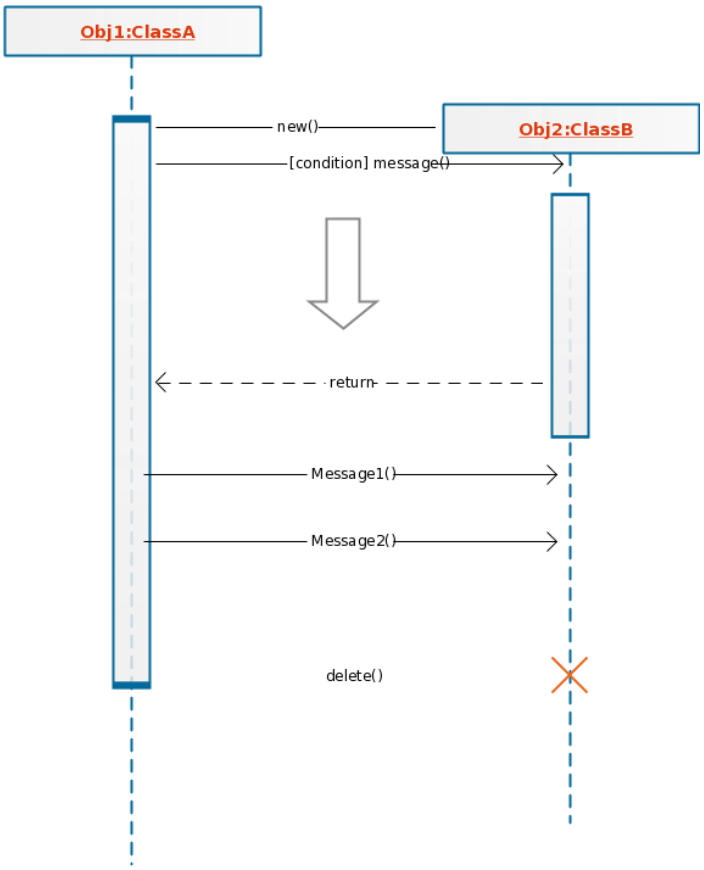
State machine diagrams are similar to activity diagrams, although notations and usage change a bit. They are sometime known as state diagrams or state chart diagrams as well. These are very useful to describe the behavior of objects that act differently according to the state they are in at the moment. The State machine diagram below shows the basic states and actions.



State Machine diagram in UML, sometime referred to as State or State chart diagram

Sequence Diagram

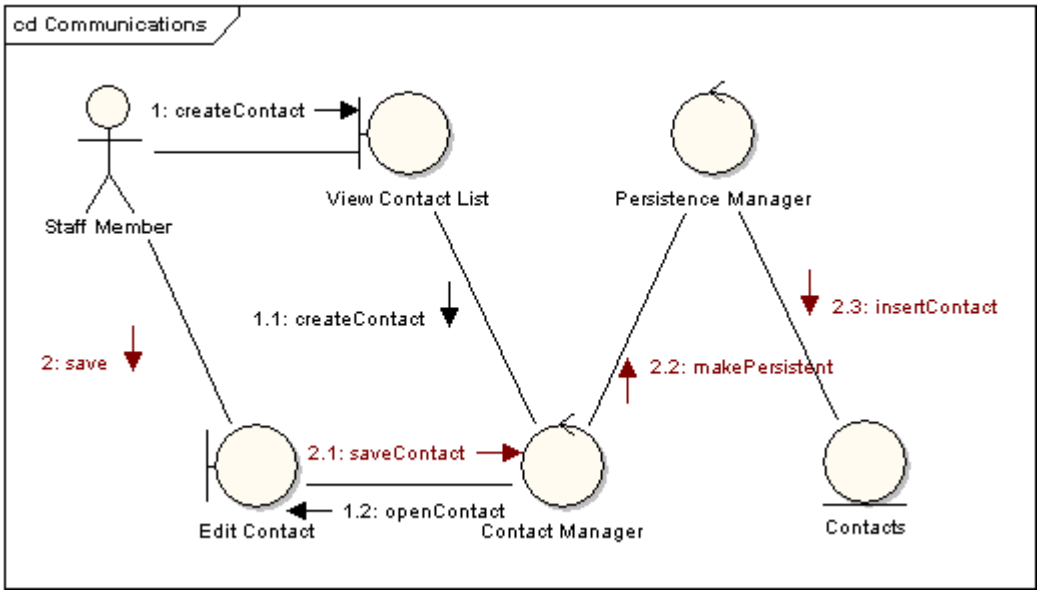
Sequence diagrams in UML show how objects interact with each other and the order those interactions occur. It's important to note that they show the interactions for a particular scenario. The processes are represented vertically and interactions are show as arrows. This article explains the **purpose and the basics of Sequence diagrams**.



Sequence Diagrams in UML shows the interaction between two processes

Communication Diagram

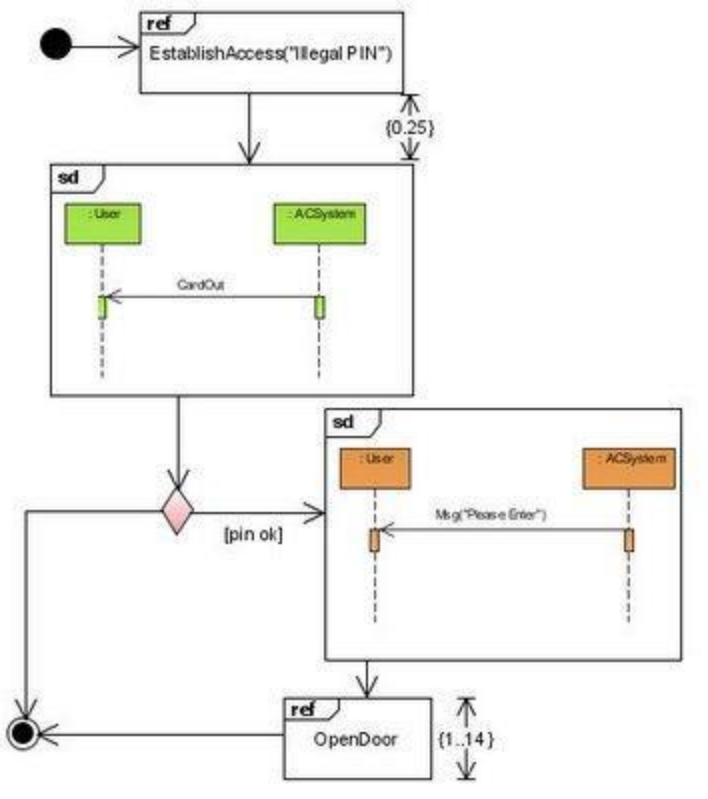
Communication diagram was called collaboration diagram in UML 1. It is similar to sequence diagrams, but the focus is on messages passed between objects. The same information can be represented using a sequence diagram and different objects.



Communication Diagram in UML

Interaction Overview Diagram

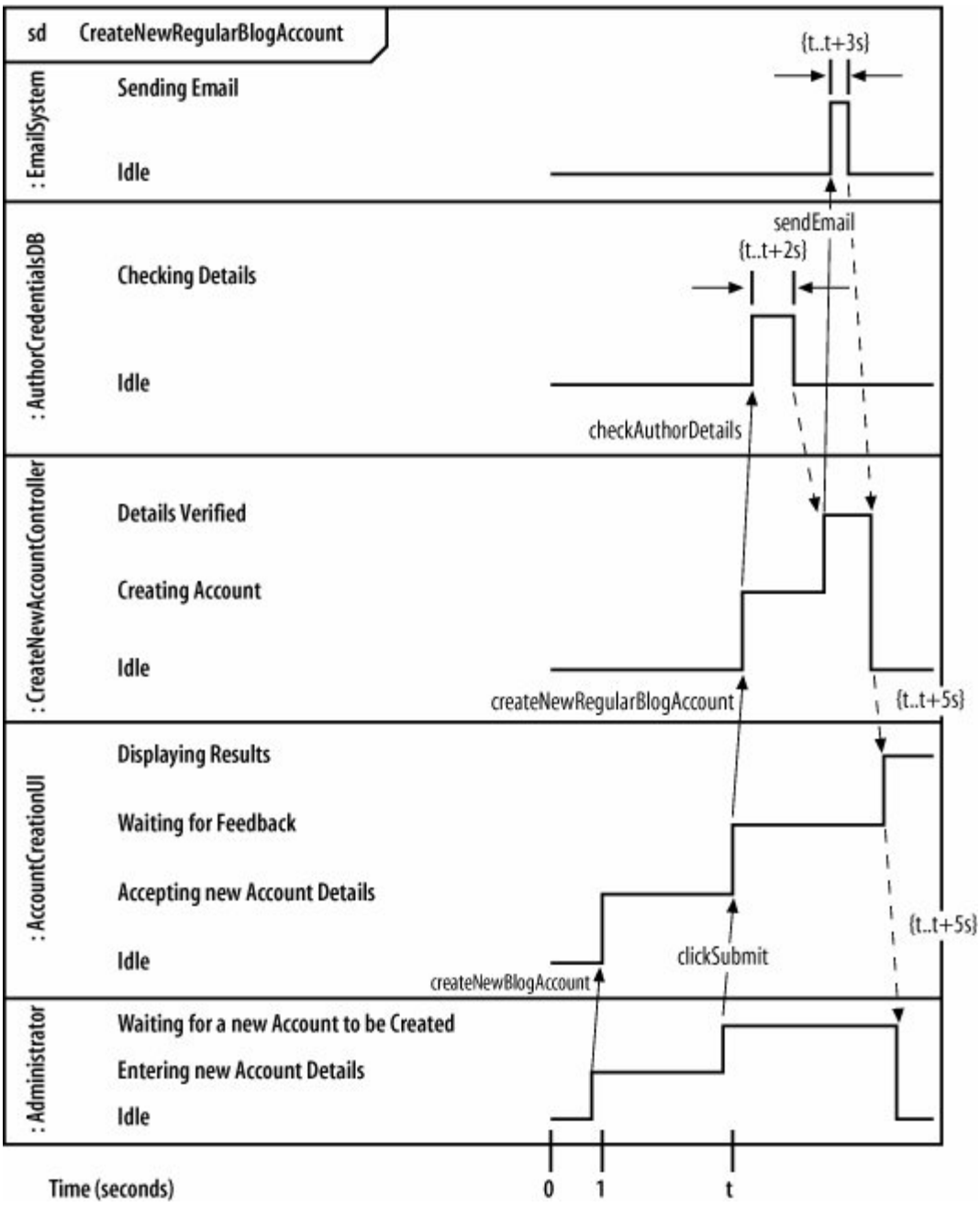
Interaction overview diagrams are very similar to activity diagrams. While activity diagrams show a sequence of processes, Interaction overview diagrams show a sequence of interaction diagrams. In simple terms, they can be called a collection of interaction diagrams and the order they happen. As mentioned before, there are seven types of interaction diagrams, so any one of them can be a node in an interaction overview diagram.



Interaction overview diagram in UML

Timing Diagram

Timing diagrams are very similar to sequence diagrams. They represent the behavior of objects in a given time frame. If it's only one object, the diagram is straight forward, but if there are more than one object involved, they can be used to show interactions of objects during that time frame as well.



Timing Diagram in UML

Mentioned above are all the UML diagram types. The links given in each section explain the diagrams in more detail and cover the usage, symbols etc. UML offers many diagram types, and sometimes two diagrams can explain the same thing using different notations.

Chapter 3 Design Patterns

Convention vs Configuration:

Convention over configuration (also known as **coding by convention**) is a software design paradigm used by software frameworks that attempt to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility. The concept was introduced by David Heinemeier Hansson to describe the philosophy of the Ruby on Rails web framework, but is related to earlier ideas like the concept of "sensible defaults" and the principle of least astonishment in user interface design. Software frameworks that support the convention over configuration development approach include Ruby on Rails, JavaBeans and CakePHP.

The phrase essentially means a developer only needs to specify unconventional aspects of the application. For example, if there is a class Sales in the model, the corresponding table in the database is called "sales" by default. It is only if one deviates from this convention, such as calling the table "product sales" that one needs to write code regarding these names.

When the convention implemented by the tool matches the desired behavior, it behaves as expected without having to write configuration files. Only when the desired behavior deviates from the implemented convention is explicit configuration required.

Ruby on Rails' use of the phrase is particularly focused on its default project file and directory structure, which prevent developers from having to write XML configuration files to specify which modules the framework should load, which was common in many earlier frameworks.

Disadvantages of the convention over configuration approach can occur due to conflicts with other software design principles, like the Zen of Python's "explicit is better than implicit." A software framework based on convention over configuration often involves a domain-specific language with a limited set of constructs or an inversion of control in which the developer can only affect behavior using a limited set of hooks, both of which can make implementing behaviors not easily expressed by the provided conventions more difficult than when using a software library that does not try to decrease the number of decisions developers have to make or require inversion of control.

Other methods of decreasing the number of decisions a developer needs to make include programming idioms and configuration libraries with a multilayered architecture.

Intent

Design a framework so that it enforces standard naming conventions for mapping classes to resources or events. A programmer only needs to write the mapping configurations when the naming convention fails.

Motivation

General-purpose frameworks usually require one or more configuration files in order to set up the framework. A configuration file provides a mapping between a class and a resource (a database) or an event (a URL request). As the size and complexity of applications grow, so do the configuration files, making them harder to maintain.

Below is an example showing a typical configuration file.

```
<hibernate-mapping>
<class name="User" table="users">
 <id name="ID" column="id" type="string">
    <generator class="assigned"></generator>
 </id>
 <property name="password"  column="password"  type="string" />
</class>
</hibernate-mapping>
```

Figure 1: A Hibernate mapping definition

```
CREATE TABLE users (
 id VARCHAR(20) NOT NULL,
```

```
  password VARCHAR(20),
  PRIMARY KEY(id)
);
```

Figure 2: The Users table in the database

Figure 1 above shows a mapping file for Hibernate, an object/relational persistence and query service framework for Java. The Hibernate mapping definition in Figure 1 maps the User class to the Users table in the database. The Users table in the database is described in Figure 2 using SQL. The fields of class User are also mapped to the columns in the Users table.

Hibernate uses the configuration file in Figure 1 to map objects to the database. For instance, if we create an object Alice of type User, calling Alice.getId() will actually perform a look up to the relevant row in the Users database table and retrieve the information in the id column.

Modifying configuration files, usually in XML, is tedious and also error-prone. Most errors in configuration files are only detected at runtime in the form of a runtime error, usually a ClassNotFoundException. More importantly, a lot of the mapping information can be inferred easily from the structure of the database table without the need for any configuration.

For instance, we can set up the naming convention that:

1. Database table names should be the pluralized form of the class name.
2. The columns in a database table should have identical names with the fields in the class that it maps to.

These two naming conventions are *natural*. As can be seen above, the configuration file is indeed echoing the convention. In fact, most developers already adhere to certain naming conventions when they program. The Convention over Configuration pattern rewards developers for adhering to naming conventions and enforces this in a stricter manner by building it into the framework.

When the designer of a framework establishes a standard naming convention, there is little need for the configuration file. The framework has code that will invoke the relevant classes or methods based on their names. In this example, the

framework queries the table for the table name and the names of the fields when a method is invoked. And if the corresponding class does not have a corresponding field for a column name, when we access it for the first time we get a runtime error. This is no different from getting a runtime error in the event that we specified the configuration file wrongly.

The Convention over Configuration pattern reduces the amount of configuration by establishing a set of naming conventions that developers follow.

Applicability

As the designer of a framework, use the Convention over Configuration pattern when

- There are clear and practical naming conventions that can be established between parts of the framework.
- There is the opportunity to reduce the amount of configuration files that duplicate mapping information from other parts of the system.

The Convention over Configuration pattern does not preclude the need for configuration files. Configuration files are still important for the cases where convention fails. But for most cases, sticking to the conventions works and keeps things simple for the programmer and anyone reading the code.

Whenever a configuration property is set explicitly, that property overrides the underlying naming convention. Thus the framework is still fully configurable.

The configurations can be specified in a separate file (shown in Figure 1) or the configurations may be embodied in the code, as we shall in the *Sample Code* section.

Consequences

The Convention over Configuration pattern has the following benefits and liabilities:

1. ***Allows new developers to learn a system quickly***. Once developers understand the naming convention, they can quickly start developing without worrying about writing the configurations to make things work.

This gives developers the impression that the framework Works Out of the Box[15]with little or no configuration. Frameworks that work out of the box empowers developers to quickly create prototypes for testing. Compare this to frameworks that require multiple configuration files to get the system up and running even for simple tasks. After they have become more familiar with the framework, they can write configurations for the unconventional cases.

2. ***Promotes uniformity*.** Developers working on different projects but using the same framework can quickly grasp how different systems work since the same naming conventions are promoted throughout the framework. This helps in maintaining a ubiquitous language[3] for the development team.

3. ***Better dynamism***. Changing the name of the class or method in the source code does not require modifying a configuration file. Since the framework does not rely on static configuration files, but rather enforces the naming conventions during runtime, changes made are automatically propagated through the application.

"This is the problem with conventions – they have to be continually resold to each developer. If the developer has not learned the convention, or does not agree with it, then the convention will be violated. And one violation can compromise the whole structure." -Robert C. Martin[2]"

4. ***Requires familiarity***. The naming conventions become part of the implicit knowledge of the framework. Once a set of conventions has been established, it becomes hard to change them. In fact, not following those conventions makes the system harder to use. Naming conventions have to be included in the documentation and followed consistently in code samples to avoid confusion.

5. ***Larger framework***. By shifting the responsibility of configuration from the developer, the framework itself has to enforce those conventions; the set of conventions has to be baked into the framework. If there are a large number of conventions that need to be supported, the framework becomes larger. Thus, only enforce clear and practical naming conventions in the framework itself.

6. ***Hard to refactor existing frameworks to adopt a new naming convention*.** It might not be feasible to use Convention over Configuration when an existing framework has a large group of developers using it. There are currently no automated tools that can upgrade an application to use features in a newer version of the framework. So developers using a version of the framework that used an older convention cannot upgrade easily to a newer convention. The Convention over Configuration pattern is best used during the initial creation of the framework and maintained throughout updates to the framework.

Usage

The naming convention should be distilled from the ubiquitous language[3] if one exists. The naming convention can also be distilled from existing applications if the framework has been designed using the Three Examples pattern.

As the framework evolves, existing naming conventions may have to be modified or new naming conventions have to be added. This set of naming conventions has to be promoted to all the developers that are using the framework. Then, as the designers of the framework, enforce this set of conventions as part of the framework. Enforcing the naming convention is done using the reflection and Meta programming properties of the programming language.

Also, design the framework to accommodate for the ability for developers to configure if the conventions do not fit.

Lazy Initialization:

Lazy initialization of an object means that its creation is deferred until it is first used. (For this topic, the terms lazy initialization and lazy instantiation are synonymous.) Lazy initialization is primarily used to improve performance, avoid wasteful computation, and reduce program memory requirements. These are the most common scenarios:

When you have an object that is expensive to create, and the program might not use it. For example, assume that you have in memory a Customer object that has an Orders property that contains a large array of Order objects that, to be initialized, requires a database connection. If the user never asks to display the Orders or use the data in a computation, then there is no reason to use system memory or computing cycles to create it. By using Lazy to declare the Orders object for lazy initialization, you can avoid wasting system resources when the object is not used.

When you have an object that is expensive to create, and you want to defer its creation until after other expensive operations have been completed. For example, assume that your program loads several object instances when it starts, but only some of them are required immediately. You can improve the startup performance of the program by deferring initialization of the objects that are not required until the required objects have been created.

Dependency Injection and Inversion of Control:

Inversion of control is a design paradigm with the goal of reducing awareness of concrete implementations from application framework code and giving more control to the domain specific components of your application. In a traditional top down designed system, the logical flow of the application and dependency awareness flows from the top components, the ones designed first, to the ones designed last. As such, inversion of control is an almost literal reversal of the control and dependency awareness in an application.

Dependency injection is a pattern used to create instances of classes that other classes rely on without knowing at compile time which implementation will be used to provide that functionality.

There are several basic techniques to implement inversion of control. These are:

1. using a factory pattern
2. using a service locator pattern
3. using a **dependency injection** of any given below type:
    - a constructor injection
    - a setter injection
    - an interface injection

Working Together

Inversion of control can utilize dependency injection because a mechanism is needed in order to create the components providing the specific functionality. Other options exist and are used, e.g. activators, factory methods, etc., but frameworks don't need to reference those utility classes when framework classes can accept the dependency (ies) they need instead.

Examples

One example of these concepts at work is the plug-in framework in Reflector. The plug-ins have a great deal of control of the system even though the application didn't know anything about the plug-ins at compile time. A single method is called on each of those plug-ins, Initialize if memory serves, which passes control over to the plug-in. The framework doesn't know what they will do, it just lets them do it. Control has been taken from the main application and given to the component doing the specific work; inversion of control.

The application framework allows access to its functionality through a variety of service providers. A plug-in is given references to the service providers when it is created. These dependencies allow the plug-in to add its own menu items, change how files are displayed, display its own information in the appropriate panels, etc. Since the dependencies are passed by interface, the implementations can change and the changes will not break the code as long as the contract remains intact.

At the time, a factory method was used to create the plug-ins using configuration information, reflection and the Activator object (in .NET at least). Today, there are tools, MEF for one, that allow for a wider range of options when injecting dependencies including the ability for an application framework to accept a list of plugins as a dependency.

Design Patterns

The Originator of Patterns

• "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 the same way twice." -- Christopher Alexander 1977

What are Patterns?

• Current use comes from the work of the architect
• Alexander studied ways to improve the process of designing buildings and urban areas
• “Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.”
• Hence, the common definition of a pattern: “A solution to a problem in a context.”
• Patterns can be applied to many different areas of human endeavor, including software development.

Patterns in Software

• "Designing object-oriented software is hard and designing reusable object-oriented software is even harder." - Erich Gamma
• Experienced designers reuse solutions that have worked in the past
• Well-structured object-oriented systems have recurring patterns of classes and objects
• Knowledge of the patterns that have worked in the past allows a designer to be more productive and the resulting designs to be more flexible and reusable

The Gang of Four

• The authors of the Design Patterns Book (1994) came to be known as the "Gang of Four." The name of the book ("Design Patterns: Elements of
Reusable Object-Oriented Software") is too long for e-mail, so "book by the gang of four" became a shorthand name for it. After all, it isn't the ONLY book on patterns. That got shortened to "GOF book", which is pretty cryptic the first time you hear it.

• The GOF are :- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides

Design Patterns

• Design patterns describe the relations and interactions of different class or objects or types.
• They do not specify the final class or types that will be used in any software code, but give an abstract view of the solution.
• Patterns show us how to build systems with good object oriented design qualities by reusing successful designs and architectures.
• Expressing proven techniques speed up the development process and make the design patterns, more accessible to developers of new system.

Classification of Design Patterns

• Design patterns were originally classified into three types
– Creational patterns
– Structural patterns
– Behavioral patterns.

• A fourth has since been added
– Concurrency patterns

Creational Patterns

• Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
• The basic form of object creation could result in design problems or added complexity to the design.
• Creational design patterns solve this problem by somehow controlling this object creation

Structural Patterns

• **Structural design patterns** are design patterns that ease the design by identifying a simple way to realize relationships between entities.
• These describe how objects and classes combine themselves to form a large structure

Behavioral Patterns

• Design patterns that identify common communication patterns between objects and realize these patterns.
• These patterns increase flexibility in carrying out this communication.

Structure of a Design Pattern

• Design pattern documentation is highly structured.
• The patterns are documented from a template that identifies the information needed to understand the software problem and the solution in terms of the relationships between the classes and objects necessary to implement the solution.
• There is no uniform agreement within the design pattern community on how to describe a pattern template.

Pattern Documentation

• **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
• **Intent:** A description of the goal behind the pattern and the reason for using it.
• **Also Known As:** Other names for the pattern.
• **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
• **Applicability:** Situations in which this pattern is usable; the context for the pattern.
• **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
• **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
• **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
• **Consequences:** A description of the results, side effects, and tradeoffs caused by using the pattern.
• **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
• **Sample Code:** An illustration of how the pattern can be used in a programming language
• **Known Uses:** Examples of real usages of the pattern.
• **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Creational Patterns

- Abstract Factory - Creates an instance of several families of classes
- Builder - Separates object construction from its representation
- Factory Method - Creates an instance of several derived classes
- Prototype - A fully initialized instance to be copied or cloned
- Singleton - A class of which only a single instance can exist

Structural Patterns

- Adapter - Match interfaces of different classes
- Bridge - Separates an object’s interface from its implementation
- Composite - A tree structure of simple and composite objects
- Decorator - Add responsibilities to objects dynamically
- Facade - A single class that represents an entire subsystem
- Flyweight - A fine-grained instance used for efficient sharing
- Proxy - An object representing another object

Behavioral Patterns

- Chain of Resp. - A way of passing a request between a chain of objects
- Command - Encapsulate a command request as an object

Interpreter A way to include language elements in a program

- Iterator - Sequentially access the elements of a collection
- Mediator - Defines simplified communication between classes
- Memento - Capture and restore an object's internal state
- Observer - A way of notifying change to a number of classes
- State - Alter an object's behavior when its state changes
- Strategy - Encapsulates an algorithm inside a class
- Template Method - Defer the exact steps of an algorithm to a subclass
- Visitor - Defines a new operation to a class without change

Factory Pattern

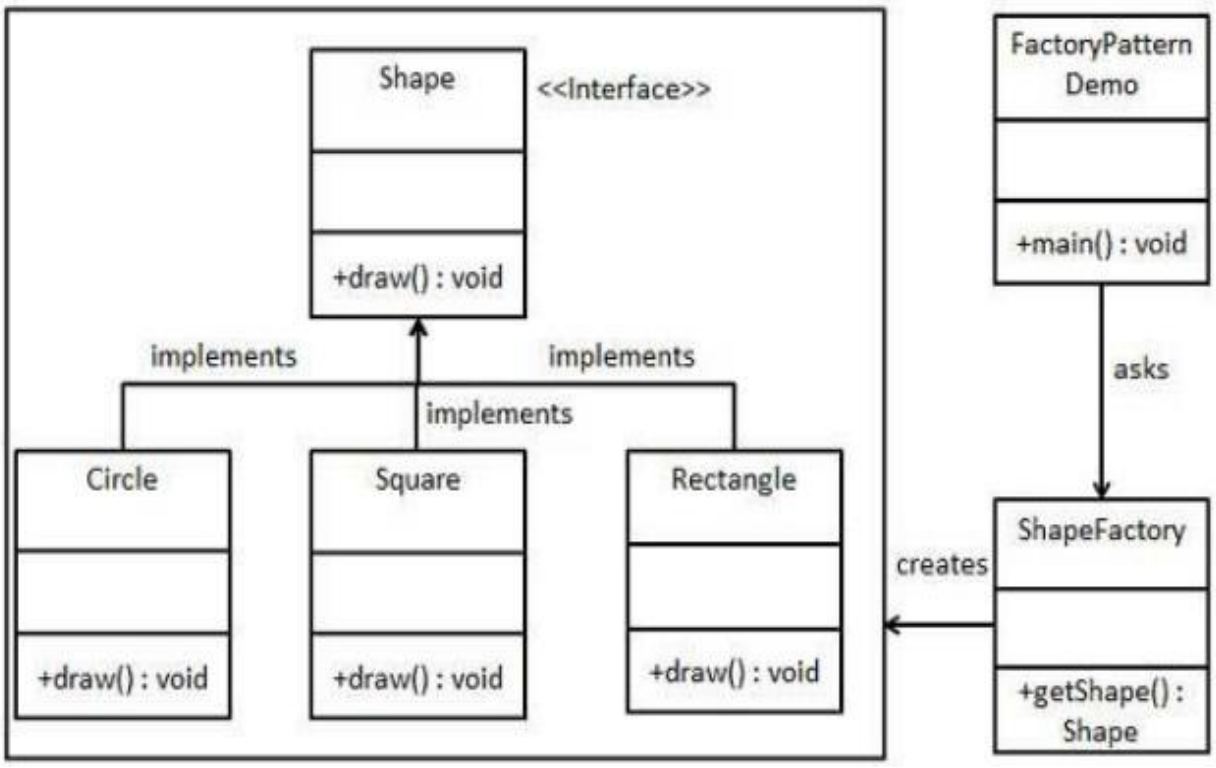
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

• We're going to create a ***Shape interface*** and concrete classes implementing the Shape interface. A factory class ***ShapeFactory*** is defined as a next step.

• ***FactoryPatternDemo***, our demo class will use ***ShapeFactory*** to get a Shape object.
• It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

Class Diagram



Shape.java

• Step 1

– Create an interface (Shape.java)

```
 Public interface Shape {
Void draw ();
};
```

Rectangle.java

• Step 2

– Create concrete classes implementing the same interface

– //Rectangle.java

```
public class Rectangle implements Shape
{
@Override
public void draw()
{
System.out.println("Inside Rectangle::draw() method.");
}
};
```

Circle.java

```
public class Circle implements Shape {
@Override
public void draw()
{
System.out.println("draw method in circle class");
}
};
```

Square.java

```
public class Square implements Shape{
@Override
public void draw()
{
System.out.println("Draw method in
```

Square class");
}
};

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Singleton Design Pattern

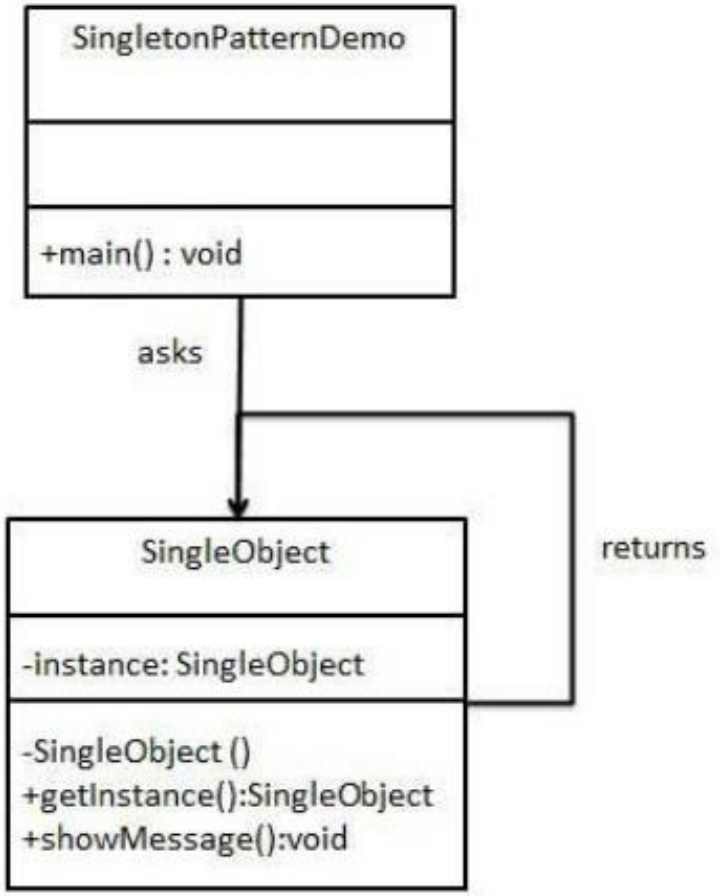
• Singleton pattern is one of the simplest design patterns in Java.
• This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.
• This pattern involves a single class which is responsible to creates own object while making sure that only single object get created.
• This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

• We're going to create a ***SingleObject*** class. ***SingleObject*** class have its constructor as private and have a static instance of itself.
• ***SingleObject*** class provides a static method to get its static instance to outside world.

• ***SingletonPatternDemo***, our demo class will use SingleObject class to get a SingleObject object.

Class Diagram



Step 1:

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not
visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Step 3

Verify the output.

```
Hello World!
```

Chapter 4: Web Application Architecture

Presentation Layer:

Overview

The presentation layer contains the components that implement and display the user interface, and manage user interaction. This layer includes controls for user input and display, in addition to components that organize user interaction.

Figure 1. Shows how the presentation layer fits into a common application architecture.

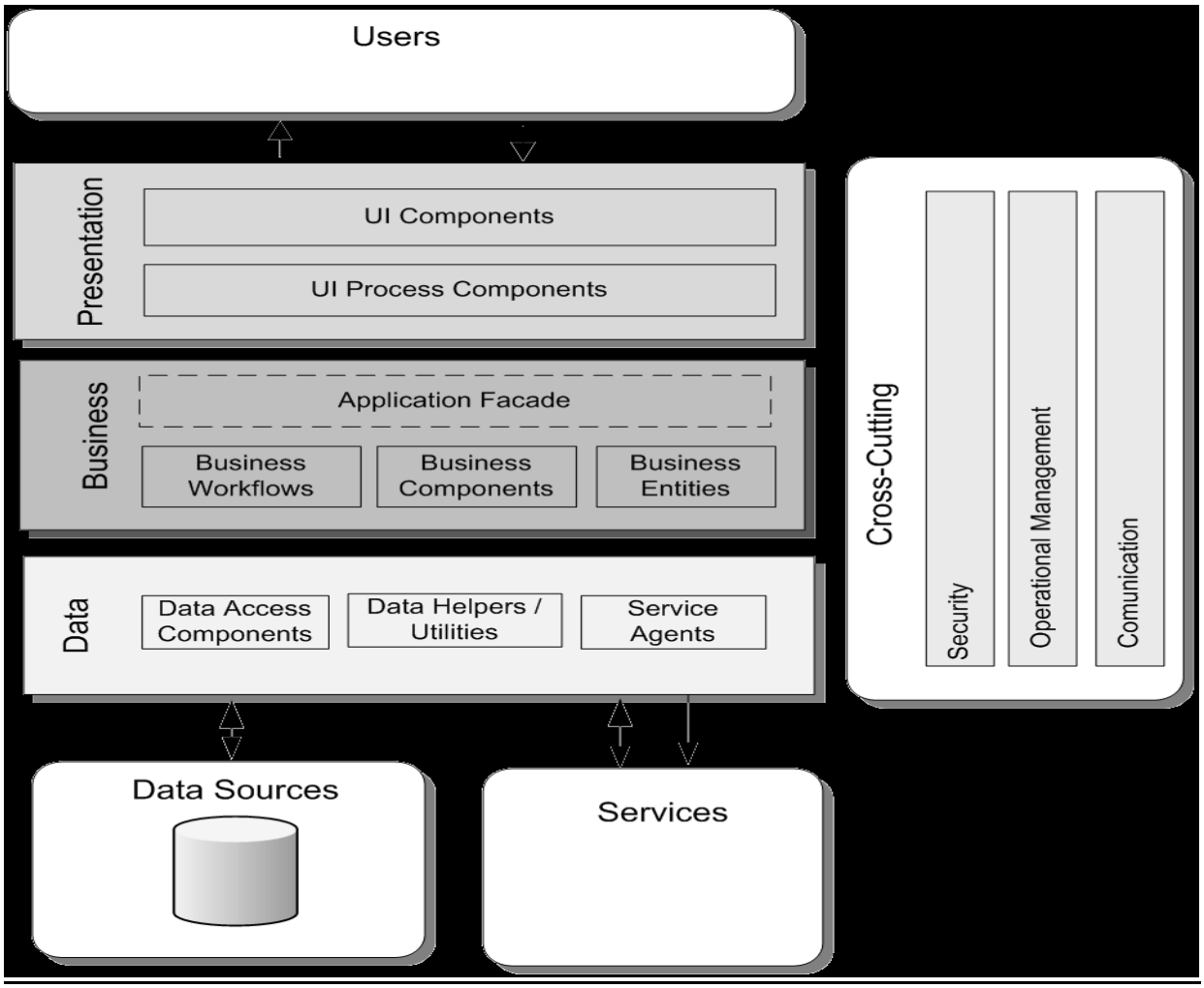


Figure 1 - A typical application showing the presentation layer and the components it may contain.

Presentation Layer Components

• **User interface (UI) components**. User interface components provide a way for users to interact with the application. They render and format data for users. They also acquire and validate data input by the user.

• **User process components**. User process components synchronize and orchestrate user interactions. Separate user process components may be useful if you have a complicated user interface. Implementing common user interaction patterns as separate user process components allows you to reuse them in multiple user interfaces.

Approach

The following steps describe the process you should adopt when designing the presentation layer for your Web application. This approach will ensure that you consider all of the relevant factors as you develop your architecture:

1. Determine how you will present data. Choose the data format for your presentation layer and decide how you will present the data in your User Interface (UI).
2. Determine your data validation strategy. Use data validation techniques to protect your system from un-trusted input.
3. Determine your business logic strategy. Factor out your business logic to decouple it from your presentation layer code.
4. Determine your strategy for communication with other layers. If your application has multiple layers, such as a data access layer and a business layer, determine a strategy for communication between your presentation layer and other layers.

Design Considerations

There are several key factors that you should consider when designing your Web presentation layer. Use the following principles to ensure that your design meets the requirements for your application, and follows best practices:

• **Use the relevant patterns.** Review the presentation layer patterns for proven solutions to common presentation problems.
• **Design for separation of concerns.** Use dedicated UI components that focus on rendering and display. Use dedicated presentation entities to manage the data required to present your views. Use dedicated UI process components to manage the processing of user interaction.

• **Consider human interface guidelines.** Review your organization's guidelines for user interface design. Review established user interface guidelines based upon the client type and technologies that you have chosen.

• **Adhere to user-driven design principles.** Before designing your presentation layer, understand your customer. Use surveys, usability studies, and interviews to determine the best presentation design to meet your customer’s requirements.

Presentation Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Category	Common Issues
Caching	• Caching volatile data. • Caching unencrypted sensitive data. • Incorrect choice of caching store. • Failing to choose a suitable caching mechanism for use in a Web farm. • Assuming that data will still be available in the cache – it may have expired and been removed.
Composition	• Failing to consider use of patterns and libraries that support dynamic layout and injection of views and presentation at runtime. • Using presentation components that have dependencies on support classes and services instead of considering patterns that support run-time dependency injection. • Failing to use the Publish/Subscribe pattern to support events between components. • Failing to properly decouple the application as separate modules that can be added easily.
Exception Management	• Failing to catch unhandled exceptions. • Failing to clean up resources and state after an exception occurs. • Revealing sensitive information to the end user. • Using exceptions to implement application logic. • Catching exceptions you do not handle. • Using custom exceptions when not necessary.
Input	• Failing to design for intuitive use, or implementing over-complex interfaces. • Failing to design for accessibility. • Failing to design for different screen sizes and resolutions. • Failing to design for different device and input types, such as mobile devices, touch-screen, and pen and ink enabled devices.
Layout	• Using an inappropriate layout style for Web pages. • Implementing an overly-complex layout. • Failing to choose appropriate layout components and technologies. • Failing to adhere to accessibility and usability guidelines and standards. • Implementing an inappropriate workflow interface. • Failing to support localization and globalization.

Navigation	• Inconsistent navigation. • Duplication of logic to handle navigation events. • Using hard-coded navigation. • Failing to manage state with wizard navigation.
Presentation Entities	• Defining entities that are not necessary. • Failing to implement serialization when necessary.
Request Processing	• Blocking the user interface during long-running requests. • Mixing processing and rendering logic. • Choosing an inappropriate request-handling pattern.
User Experience	• Displaying unhelpful error messages. • Lack of responsiveness. • Over-complex user interfaces. • Lack of user personalization. • Lack of user empowerment. • Designing inefficient user interfaces.
UI Components	• Creating custom components that are not necessary. • Failing to maintain state in the MVC pattern. • Choosing inappropriate UI components.
UI Process Components	• Implementing UI process components when not necessary. • Implementing the wrong design patterns. • Mixing business logic with UI process logic. • Mixing rendering logic with UI process logic.
Validation	• Failing to validate all input. • Relying only on client-side input validation. You must always validate input on the server or in the business layer as well. • Failing to correctly handle validation errors. • Not identifying business rules that are appropriate for validation. • Failing to log validation failures.

Business Layer:

Business Components

The following list explains the roles and responsibilities of the main components within the business layer:

• **Application Facade**. (Optional). An application façade combines multiple business operations into single message-based operation. You might access the application façade from the presentation layer using different communication technologies.

• **Business components**. After a user process collects the data it requires, the data can be operated on using business rules. The rules will describe how the data should be manipulated and transformed as dictated by the business itself. The rules may be simple or complex, depending on the business itself. The rules can be updated as the business requirements evolve.

• **Business entity components**. Business entities are used to pass data between components. The data represents real-world business entities, such as products and orders. The business entities that the application uses internally are usually data structures such as Datasets, Extensible Markup Language (XML) streams. Alternatively, they can be implemented using custom object-oriented classes that represent the real-world entities your application has to work with, such as a product or an order.

• **Business workflow**. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. Business workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.

Approach

When designing a business layer, you must also take into account the design requirements for the main constituents of the layer, such as business components, business entities and business workflow components. This section briefly explains the main activities involved in designing each of the components and the business layer itself. Perform the following key activities in each of these areas when designing your data layer:

1. **Create an overall design for your business layer:**
- Identify the consumers of your business layer.
- Determine how you will expose your business layer.
- Determine the security requirements for your business layer.
- Determine the validation requirements and strategy for your business layer.
- Determine the caching strategy for your business layer.
- Determine the exception management strategy for your business layer.

2. **Design your business components**:
- Identify business components your application will use.
- Make key decisions about location, coupling and interactions for business components.
- Choose appropriate transaction support.
- Identify how your business rules are handled.
- Identify patterns that fit the requirements

3. **Design your business entity components**:

- Identify common data formats for the business entities.
- Choose the data format.
- Optionally, choose a design for your custom objects.
- Optionally, determine what serialization support you will need.

4. **Design your workflow components:**

- Identify workflow style using scenarios.
- Choose an authoring mode.
- Determine how rules will be handled.
- Choose a workflow solution.
- Design business components to support workflow.

Design Considerations

When designing a business layer, the goal of a software architect is to minimize the complexity by separating tasks into different areas of concern. For example, business processing, business workflow, and business entities all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not include code related to other areas of concern. When designing the business layer, consider following guidelines:

• **Decide if you need a separate business layer**. It is always a good idea to use a separate business layer where possible to improve the maintainability of your application

• **Identify the responsibilities of your business layer.** Use a business layer for processing complex business rules, transforming data, applying policies, and for validation.

• **Do not mix different types of components in your business layer.** Use a business layer to decouple business logic from presentation and data access code, and to simplify the testing of business logic.

• **Reuse common business logic.** Use a business layer to centralize common business logic functions and promote reuse.

• **Identify the consumers of your business layer.** This will help to determine how you expose you business layer. For example, if your business layer will be used by your presentation layer and by an external application, you may choose to expose your business layer through a service.

• **Reduce round trips when accessing a remote business layer**. If you are using a message based interface, consider using coarse-grained packages for data, such

as Data transfer Objects. In addition, consider implementing a remote façade for the business layer interface.

• **Avoid tight coupling between layers**. Use abstraction when creating an interface for the business layer. The abstraction can be implemented using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation layer and the business layer.

Data Access layer:

Data Layer Components

• **Data access logic components**. Data access components abstract the logic necessary to access your underlying data stores. Doing so centralizes the data access functionality, which makes the application easier to configure and maintain.

• **Data Helpers / Utilities**. Helper functions and utilities assist in data manipulation, data transformation, and data access within the layer. They consist of specialized libraries and/or custom routines especially designed to maximize data access performance and reduce the development requirements of the logic components and the service agent parts of the layer.

• **Service agents**. When a business component must use functionality exposed by an external service, you may need to create code that manages the semantics of communicating with that service. Service agents isolate your application from the idiosyncrasies of calling diverse services, and can provide additional services such as basic mapping between the format of the data exposed by the service and the format your application requires.

Approach

A correct approach to designing the data layer will reduce development time and assist in maintenance of the data layer after the application is deployed. This section briefly outlines an effective design approach for the data layer. Perform the following key activities in each of these areas when designing your data layer:

1. **Create an overall design for your data access layer:**

- Identify your data source requirements

- Determine your data access approach
- Choose how to map data structures to the data source
- Determine how to connect to the data source
- Determine strategies for handling data source errors.

2. **Design your data access components**:
- Enumerate the data sources that you will access
- Decide on the method of access for each data source
- Determine whether helper components are required or desirable to simplify data access component development and maintenance
- Determine relevant design patterns. For example, consider using the Table Data Gateway,
- Query Object, Repository, and other patterns.

3. **Design your data helper components:**
- Identify functionality that could be moved out of the data access components and centralized for reuse
- Research available helper component libraries
- Consider custom helper components for common problems such as connection strings, data source authentication, monitoring, and exception processing
- Consider implementing routines for data access monitoring and testing in your helper components
- Consider the setup and implementation of logging for your helper components.

4. **Design your service agents:**
- Use the appropriate tool to add a service reference. This will generate a proxy and the data classes that represent the data contract from the service
- Determine how the service will be used in your application. For most applications, you should use an abstraction layer between the business layer and the data access layer, which will provide a consistent interface regardless of the data source. For smaller applications, the business layer, or even the presentation layer, may access the service agent directly.

Design Guidelines

The following design guidelines provide information about different aspects of the data access layer that you should consider. Follow these guidelines to ensure that

your data access layer meets the requirements of your application, performs efficiently and securely, and is easy to maintain and extend as business requirements change.

• **Choose the data access technology.** The choice of an appropriate data access Technology will depend on the type of data you are dealing with, and how you want to manipulate the data within the application. Certain technologies are better suited for specific scenarios. The following sections of this guide discuss these options and enumerate the benefits and drawbacks of each data access technology.

• **Use abstraction to implement a loosely coupled interface to the data access layer**. This can be accomplished by defining interface components, such as a gateway with well-known inputs and outputs, which translate requests into a format understood by components within the layer. In addition, you can use interface types or abstract base classes to define a shared abstraction that must be implemented by interface components.

• **Consider consolidating data structures.** If you are dealing with table-based entities in your data access layer, consider using Data Transfer Objects (DTOs) to help you organize the data into unified structures. In addition, DTOs encourage coarse-grained operations while providing a structure that is designed to move data across different boundary layers.

• **Encapsulate data access functionality within the data access layer**. The data access layer hides the details of data source access. It is responsible for managing connections, generating queries, and mapping application entities to data source structures. Consumers of the data access layer interact through abstract interfaces using application entities such as custom objects, Datasets, Data Readers, and XML documents. Other application layers that access the data access layer will manipulate this data in more complex ways to implement the functionality of the application. Separating concerns in this way assists in application development and maintenance.

• **Decide how to map application entities to data source structures.** The type of entity you use in your application is the main factor in deciding how to map those entities to data source structures.

• **Decide how you will manage connections.** As a rule, the data access layer should create and manage all connections to all data sources required by the application. You must choose an appropriate method for storing and protecting connection information that conforms to application and security requirements.

• **Determine how you will handle data exceptions.** The data access layer should catch and (at least initially) handle all exceptions associated with data sources and CRUD operations.
Exceptions concerning the data itself, and data source access and timeout errors, should be handled in this layer and passed to other layers only if the failures affect application responsiveness or functionality.

• **Consider security risks**. The data access layer should protect against attacks that try to steal or corrupt data, and protect the mechanisms used to gain access to the data source. It should also use the "least privilege" design approach to restrict privileges to only those needed to perform the operations required by the application. If the data source itself has the ability to limit privileges, security should be considered and implemented in the data access layer as well as in the source.

• **Reduce round trips**. Consider batching commands into a single database operation.

• **Consider performance and scalability objectives**. Scalability and performance objectives for the data access layer should be taken into account during design. For example, when designing an Internet-based merchant application, data layer performance is likely to be a bottleneck for the application. When data layer performance is critical, use profiling to understand and then limit expensive data operations.

Chapter 5: SOA and RESTFUL Web services

Service-oriented architecture (SOA) and development is a paradigm where software components are created with concise interfaces, and each component performs a discrete set of related functions. With its well-defined interface and contract for usage, each component, provides a service to other software components. This is analogous to an accountant who provides a service to a

business, even though that service consists of many related functions—bookkeeping, tax filing, investment management, and so on.

There are no technology requirements or restrictions with SOA. You can build a service in any language with standards such as CORBA, remote procedure calls (RPC), or XML. Although SOA has been around as a concept for years, its vague definition makes it difficult to identify. The client/server development model of the early '90s was a simple example of an SOA-based approach to software development.

A web service is an example of an SOA with a well-defined set of implementation choices. In general, the technology choices are SOAP and the Web Service Definition Language (WSDL); both XML-based. WSDL describes the interface (the "contract"), while SOAP describes the data that is transferred. Because of the platform-neutral nature of XML, SOAP, and WSDL, Java is a popular choice for web-service implementation due to its OS-neutrality.

Web-service systems are an improvement of client/server systems, and proprietary object models such as CORBA or COM, because they're standardized and free of many platform constraints. Additionally, the standards, languages, and protocols typically used to implement web services helps systems built around them to scale better.

Representational State Transfer (REST)

However, there exists an even less restrictive form of SOA than a web service—representational state transfer (REST). Described by Roy Fielding in his doctoral dissertation, REST is a collection of principals that are technology independent, except for the requirement that it be based on HTTP.

A system that conforms to the following set of principals is said to be "RESTful":

- All components of the system communicate through interfaces with clearly defined methods and dynamic code.
- Each component is uniquely identified through a hypermedia link (URL).
- A client/server architecture is followed (web browser and web server).
- All communication is stateless.
- The architecture is tiered, and data can be cached at any layer.

These principals map directly to those used in the development of the Web, and according to Fielding, account for much of the Web's success. HTTP protocol, its interface of methods (GET, POST, HEAD, and so on), the use of URLs, HTML, and JavaScript, as well as the clear distinction between what is a web server and web browser, all map directly to the first four principals. The final principal (involving tiers) allows for the common network technology found in most website implementations: load balancers, in-memory caches, firewalls, routers, and so on. These devices are acceptable because they don't affect the interfaces between the components; they merely enhance their performance and communication.

The Web is the premier example of a RESTful system, which makes sense since much of the Web's architecture preceded the definition of REST. What the Web makes clear, however, is that complex remote procedure call protocols are not needed to create successful, scalable, understandable, and reliable distributed software systems. Instead, the principals of REST are all you need.

Overall, REST can be described as a technology and platform-independent architecture where loosely coupled components communicate via interfaces over standard web protocols. Software, hardware, and data-centric designs maximize system efficiency, scalability, and network throughput. The underlying principal is simplicity.

Figure 1 illustrates the REST architecture, combining both logical software architecture and physical network elements. Communication is performed over HTTP, clients contain optional server caches for efficiency, services employ caches to backend databases, there are no restrictions on maximum clients, or maximum services per client, services can call services, load-balancing hardware is used for scalability, and firewalls can be used for security.

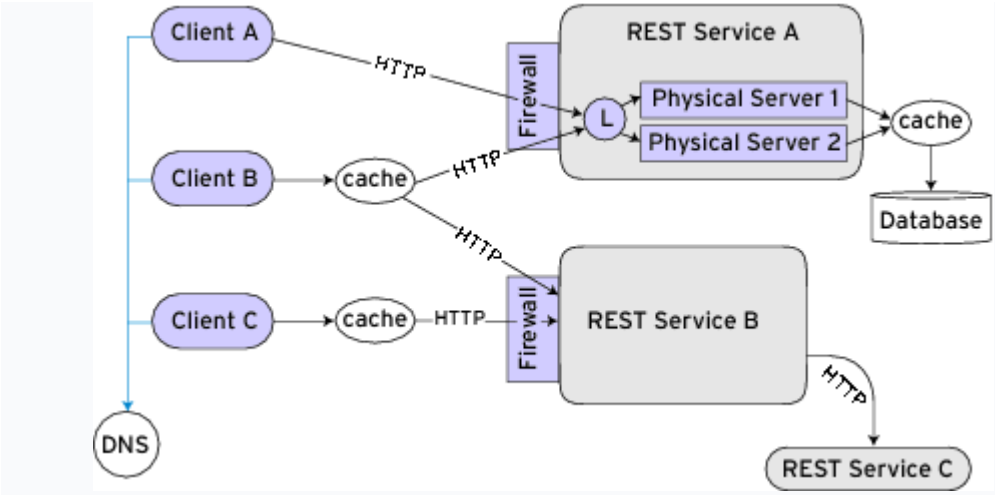


Figure 1: This architectural diagram provides a visual overview of the REST principals.

There are some interesting points on data caching that need to be made. First, data must be marked, either implicitly or explicitly, as cacheable, or non-cacheable. Second, although specialized caches may be used (custom, in-memory data structures), general-purpose caches, such as web browser caches or third-party web caches (such as Akamai), can also be used.

Resource Oriented Architectures

Main Concepts

Resource Oriented Architectures (ROA) [5] are based upon the concept of resource; each resource is a directly accessible distributed component that is handled through a standard, common interface making possible resources handling. RESTFul platforms based on REST development technology enable the creation of ROA. The main ROA concepts are the following:

• Resource – anything that's important enough to be referenced as a thing itself

• Resource name – unique identification of the resource

• Resource representation – useful information about the current state of a resource

• Resource link – link to another representation of the same or another resource

• Resource interface – uniform interface for accessing the resource and manipulating its state.
The resource interface semantics is based on the one of HTTP operations. The following table summarizes the resource methods and how they could be implemented1 using the HTTP protocol:

Resource method	*Description*	*HTTP operation*
createResource	Create a new resource (and the corresponding unique identifier)	PUT
getResourceRepresentation	Retrieve the representation of the resource	GET
deleteResource	Delete the resource (optionally including linked resources)	DELETE (referred resource only), POST (can be used if the delete is including linked resources)
modifyResource	Modify the resource	POST
getMetaInformation	Obtain meta information about the resource	HEAD

Table 1

In terms of platform implementation specification, each resource must be associated to a unique identifier that usually consists of the URL exhibiting the resource interface.

Designing Resource Oriented Architectures

One of the most important decisions that have to be taken in the design of a Resource Oriented Architecture is what must be considered a resource (by definition, each component deserving to be directly represented and accessed). This is one of the main differences between ROA and SOA where in the latter one the single, directly accessible distributed component, represents one or more business functionalities that often process different potential resources. Such resources are credited candidates for being considered resources in a ROA, thus deserving to be represented as distributed components (e.g. features offered by WFS, registers or items of registries/catalogues, WFS and Registry/Catalogue services functionalities).

Once having defined the granularity of the resources composing the ROA it is necessary to define, for each resource type, the content of the messages for invoking the methods as well as the corresponding responses. More in detail, beside the definition of resource types (and sub types), an addressing schema for accessing instances of those resource types, a response schema (response is not binary) and a mapping of logical functions to the HTTP operations. All resources in a ROA are accessed via the same common interface which is plain HTTP. It is worth noting that the usage of a common interface does not necessarily mean that all the necessary information enabling interoperability and collaboration among resources are available. The necessity of integrating a standard common interface description with some specific service instance aspects has been already addressed in other existing solutions such as, for instance, for the OGC WPS execute operation [7] where the specific processing detailed information are offered through the describe Process operation. In a ROA, this information completes the description of the resources and their interface thus enabling: i) system integration and interoperability through tools for the creation of resource clients, ii) message validation processes, and iii) model driven development (top-down approach) for which a typed description of the interface and its operation is necessary. In the REST technology such a description is based on WADL documents that play the same role of WSDL in the W3C Web Services platform; both languages use XML schema for expressing the structure of exchanged messages.

Designing Read-Only Resource-Oriented Services:

Once you have an idea of what you want your program to do. It produces a set of resources that respond to a read-only subset of HTTP's uniform interface: GET and possibly HEAD. Once you get to the end of this procedure, you should be ready to implement your resources in whatever language and framework you like. The following steps are to be done to design read-only resource-oriented services:

1. Figure out the data set
2. Split the data set into resources

For each kind of resource:

3. Name the resources with URIs
4. Expose a subset of the uniform interface
5. Design the representation(s) accepted from the client

6. Design the representation(s) served to the client
7. Integrate this resource into existing resources, using hypermedia links and forms
8. Consider the typical course of events: what's supposed to happen?
9. Consider error conditions: what might go wrong?

In the sections to come, I'll show, step by step, how following this procedure results in a RESTful web service that works like the Web.

Figure Out the Data Set:

A web service starts with a data set, or at least an idea for one. This is the data set you'll be exposing and/or getting your users to build.

Split the Data Set into Resources:

Once you have a data set in mind, the next step is to decide how to expose the data as HTTP resources. Remember that a resource is *anything interesting enough to be the target of a hypertext link*. Anything that might be referred to by name ought to have a name. Web services commonly expose three kinds of resources:

Predefined one-off resources for especially important aspects of the application.

This includes top-level directories of other available resources. Most services expose few or no one-off resources.

Example: A web site's homepage. It's a one-of-a-kind resource, at a well-known URI, which acts as a portal to other resources.

A resource for every object exposed through the service.

One service may expose many kinds of objects, each with its own resource set. Most services expose a large or infinite number of these resources.

Resources representing the results of algorithms applied to the data set.

This includes collection resources, which are usually the results of queries. Most services either expose an infinite number of algorithmic resources, or they don't expose any.

Example: A search engine exposes an infinite number of algorithmic resources. There's one for every search request you might possibly make. The Google search engine exposes one resource at *http://google.com/search?q=jellyfish* (that'd be "a directory of resources about jellyfish") and another at *http://google.com/search?q=chocolate* ("a directory of

resources about chocolate"). Neither of these resources were explicitly defined ahead of time: Google translates *any* URI of the form *http://google.com/search?q={query}* into an algorithmic resource "a directory of resources about *{query}*."

Name the Resources:

Now the resource need names. Resources are named with URIs, so let's pick some. Remember, in a resource-oriented service the URI contains all the scoping information. Our URIs need to answer questions like: "Why should the server operate on this map instead of that map?" and "Why should the server operate on this place instead of that place?"

Now let's consider the resources. The most basic resource is the list of planets. It makes sense to put this at the root URI, *http://maps.example.com/*. Since the list of planets encompasses the entire service, there's no scoping information at all for this resource (unless you count the service version as scoping information).

For the other resources I'd like to pick URIs that organize the scoping information in a natural way. There are three basic rules for URI design, born of collective experience:

1. Use path variables to encode hierarchy: /parent/child
2. Put punctuation characters in path variables to avoid implying hierarchy where none exists:
   /parent/child1; child2
3. Use query variables to imply inputs into an algorithm, for example:
   /search?q=jellyfish&start=20

Map URIs:

Now that I've designed the URI to a geographic point on a planet, what about the corresponding point on a road map or satellite map? After all, the main point of this service is to serve maps.

Earlier I said I'd expose a resource for every point on a map. For simplicity's sake, I'm not exposing maps of named places, only points of latitude and longitude. In addition to a set of coordinates or the name of a place, I need the name of the planet and the type of map (satellite map, road map, or whatever). Here are some URIs to maps of planets, places, and points:

- *http://maps.example.com/radar/Venus*
- *http://maps.example.com/radar/Venus/65.9,7.00*

- *http://maps.example.com/geologic/Earth/43.9,-103.46*

Design Your Representations:

I've decided which resources I'm exposing, and what their URIs will look like. Now I need to decide what data to send when a client requests a resource, and what data format to use.

Integrate this resource in to existing resources, using hypermedia links and forms (Link the Resources to Each Other):

Since I designed all my resources in parallel, they're already full of links to each other (see Figure 5-3). A client can get the service's "home page" (the planet list), follow a link to a specific planet, follow another link to a specific map, and then follow navigation and zoom links to jump around the map. A client can do a search for places that meet certain criteria, click one of the search results to find out more about the place, then follow another link to locate the place on a map.
One thing is still missing, though. How is the client supposed to get to a list of search results? I've set up rules for what the URI to a set of search results looks like, but if clients have to follow rules to generate URIs, my service isn't well connected. HTML solves this problem with forms. By sending an appropriate form in a representation, I can tell the client how to plug variables into a query string. The form represents infinitely many URIs, all of which follow a certain pattern. I'm going to extend my representations of places by including this HTML form.

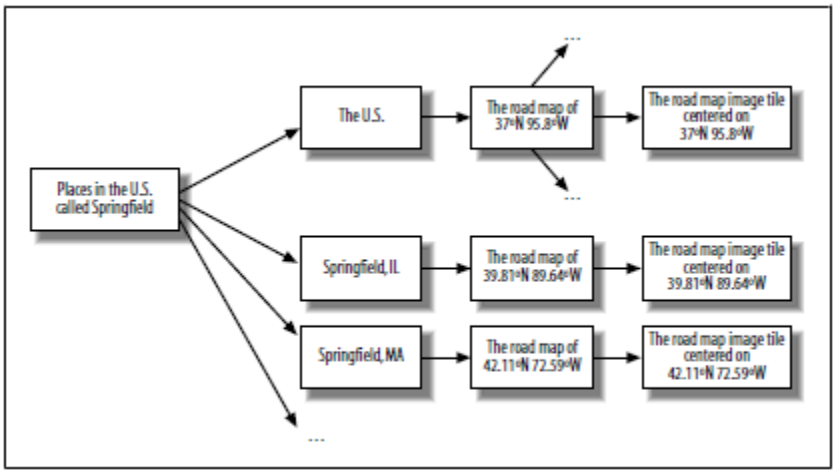


Figure 5-3. Where can you go from the list of search results?

What's supposed to happen?

This step of the design is conceptually simple, but it's the gateway to where you're going to spend much of your implementation time: making sure that client requests are correctly turned into responses. Most read-only resources have a pretty simple typical course of events. The user sends a GET request to a URI, and the server sends back a happy response code like 200
("OK"), some HTTP headers, and a representation. A HEAD request works the same way, but the server omits the representation. The only main question is which HTTP headers the client should send in the request, and which ones the server should send in the response.

What Might Go Wrong?

I also need to plan for requests I can't fulfill. When I hit an error condition I'll send a response code in the 3xx, 4xx, or 5xx range, and I may provide supplementary data in HTTP headers. If they provide an entity-body, it'll be a document describing an error condition, not a representation of the requested resource (which, after all, couldn't be served).

Here are some likely error conditions for the map application:

• The client may try to access a map that doesn't exist, like /road/Saturn. I understand what the client is asking for, but I don't have the data. The proper response code in this situation is 404 ("Not Found"). I don't need to send an entity-body along with this response code, though it's helpful for debugging.

Designing Read/Write Resource-Oriented Services:

• Same process, but now we examine full range of uniform interface operations
• Build matrix with resource types as rows, and operations as columns
• Indicate what operations apply to which types
• provide example URIs and discussion of what will happen
• Especially in the case of POST and PUT
    • PUT: create or modify resource
    • POST: append content to existing resource OR append child resource to parent resource (blog entries)
• Two questions to help
• Will clients be creating new resources of this type?
• Who's in charge of determining the new resource's URI? Client or Server? If the former, then PUT. If the latter, then POST.

New Issues: Authentication and Authorization

• Now that we are allowing a client to change stuff on our server, we need
    • Authentication: problem of tying a request to a user
    • Authorization: problem of determining which requests to let through for a given user
• HTTP provides mechanisms to enable this (HTTP Basic/Digest) and other web services roll their own (Amazon's public/private key on subset of request)
• Another Issue: Privacy
• Can't transmit "private information" in the clear; need to use HTTPS
• Another Issue: Trust
• How do you trust your client software to do the right thing?
• Especially in today's environment with malware becoming harder and harder to discern

Chapter 6:

Software Security:

Software security is an idea implemented to protect software against malicious attack and other hacker risks so that the software continues to function correctly under such potential risks. Security is necessary to provide integrity, authentication and availability.

Any compromise to integrity, authentication and availability makes a software unsecure. Software systems can be attacked to steal information, monitor content, introduce vulnerabilities and damage the behavior of software. Malware can cause DoS (denial of service) or crash the system itself.

Basic Attacks: Buffer overflow, stack overflow, command injection and SQL injections are the most common attacks on the software. Buffer and stack overflow attacks overwrite the contents of the heap or stack respectively by writing extra bytes.

Command injection can be achieved on the software code when system commands are used predominantly. New system commands are appended to existing commands by the malicious attack. Sometimes system command may stop services and cause DoS.

SQL injections use malicious SQL code to retrieve or modify important information from database servers. SQL injections can be used to bypass login credentials. Sometimes SQL injections fetch important information from a database or delete all important data from a database. The only way to avoid such attacks is to practice good programming techniques. System-level security can be provided using better firewalls. Using intrusion detection and prevention can also aid in stopping attackers from easy access to the system.

Cross-site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Stored and Reflected XSS Attacks

XSS attacks can generally be categorized into two categories: stored and reflected.

Stored XSS Attacks

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

Reflected XSS Attacks

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user’s browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

Dos and Don'ts of client authentication:

Client authentication is a common requirement for modern Web sites as more and more personalized and access-controlled services move online. Unfortunately, many sites use authentication schemes that are extremely weak and vulnerable to attack. These problems are most often due to careless use of authenticators stored on the client. We observed this in an informal survey of authentication mechanisms used by various popular Web sites. Of the twenty-seven sites we investigated, we weakened the client authentication of two systems, gained unauthorized access on eight, and extracted the secret key used to mint authenticators from one.

This is perhaps surprising given the existing client authentication mechanisms within HTTP and SSL/TLS, two well-studied mechanisms for providing authentication secure against a range of adversaries. However, there are many reasons that these mechanisms are not suitable for use on the Web at large. Lack of a central infrastructure such as a public-key infrastructure or a uniform Kerberos contributes to the proliferation of weak schemes. We also found that many Web sites would design their own authentication mechanism to provide a better user experience. Unfortunately, designers and implementers often do not have a background in security and, as a result, do not have a good understanding of the tools at their disposal. Because of this lack of control over user interfaces and unavailability of a client authentication infrastructure, Web sites continue to reinvent weak home-brew client authentication schemes.

Clients want to ensure that only authorized people can access and modify personal information that they share with Web sites. Similarly, Web sites want to ensure that only authorized users have access to the services and content it provides. Client authentication addresses the needs of both parties.

Client authentication involves proving the identity of a *client* (or user) to a *server* on the Web. We will use the term ``authentication'' to refer to this problem. Server authentication, the task of authenticating the server to the client, is also important but is not the focus this paper.

Practical limitations

Deploy ability:

- Technology must be widely deployed
- HTTP is stateless and session less.
- Client must provide authentication token
- Useful but high overhead
- JavaScript, Flash, Shockwave…

User Acceptability: Web sites must also consider user acceptability. Because sites want to attract many users, the client authentication must be as non-confrontational as possible. Users will be discouraged by schemes requiring work such as installing a plug-in or clicking away dialog boxes.

Performances: Stronger security protocols generally cost more in performance. Service providers naturally want to respond to as many requests as possible. Cryptographic solutions will usually degrade server performance. Authentication should not needlessly consume valuable server resources such as memory and clock cycles. With current technology, SSL becomes unattractive because of the computational cost of its initial handshaking.

Hints for Web client authentication

- Use Cryptography Appropriately
- Protect Passwords
- Handle Authenticators Carefully
- Use cryptography appropriately
- Appropriate amount of security

Keep It Simple, Stupid

- Do not be inventive

Designers should be security experts

- Do not rely on the secrecy of a protocol

Vulnerable to exposure

- Understand the properties of cryptographic tools

Example: Crypt ()

- Do not compose security schemes

Hard to foresee the effects

Protect Passwords

Limit exposure

Don't send it back to the user (much less in the clear)

Authenticate using SSL vs. HTTP

Prohibit guessable passwords

No dictionary passwords

Re-authenticate before changing passwords

Avoid replay attack

Handle authenticators carefully

Make authenticators unforgeable

highschoolalumni.com

If using keys as session identifier: should be cryptographically random

Protect from tampering (MAC)

Protect authenticators that must be secret

Authenticator as cookie

Sent by SSL

Don't forget the flag! (SprintPCS)

Authenticator as part of URL

Avoid using persistent cookies

Persistent vs. ephemeral cookies

Cookie files on the web

Limit the lifetime of authenticators

Encrypt the timestamp

Secure binding limits the damage from stolen authenticators

Bind authenticators to specific network addresses

Increases the difficulty of a replay attack

SQL INJECTION:

SQL injection is a type of web application security vulnerability in which an attacker is able to submit a database SQL command that is executed by a web application, exposing the back-end database. A SQL injection attack can occur when a web application utilizes user-supplied data without proper validation or encoding as part of a command or query. The specially crafted user data tricks the application into executing unintended commands or changing data. SQL injection allows an attacker to create, read, update, alter or delete data stored in the back-end database. In its most common form, a SQL injection attack gives access to sensitive information such as social security numbers, credit card numbers or other financial data.

Key Concepts of a SQL Injection Attack

- SQL injection is a software vulnerability that occurs when data entered by users is sent to the SQL interpreter as a part of a SQL query.
- Attackers provide specially crafted input data to the SQL interpreter and trick the interpreter to execute unintended commands.
- Attackers utilize this vulnerability by providing specially crafted input data to the SQL interpreter in such a manner that the interpreter is not able to distinguish between the intended commands and the attacker's specially crafted data. The interpreter is tricked into executing unintended commands.

- A SQL injection attack exploits security vulnerabilities at the database layer. By exploiting the SQL injection flaw, attackers can create, read, modify or delete sensitive data.

How SQL Injection works

In order to run malicious SQL queries against a database server, an attacker must first find an input within the web application that is included inside of an SQL query.

In order for an SQL injection attack to take place, the vulnerable website needs to directly include user input within an SQL statement. An attacker can then insert a payload that will be included as part of the SQL query and run against the database server.

Let's look an example code:

```
String SQLQuery ="SELECT Username, Password
 FROM users WHERE Username='" + Username +
    "' AND Password='" + Password +"'";

Statement stmt = connection.oreateStaement();
ResultSet rs = stmt.executeQuery(SQLQuery);
while (rs.next()) { … }
```

You will notice that user input is required to run this query. The interpreter will execute the command based on the inputs received for the username and password fields.


```
String SQLQuery ="SELECT Username, Password
 FROM users WHERE Username='" + Username +
    "' AND Password='" + Password +"'";
```

If an attacker provides ‘or 0=0’ as the username and password, then the query will be constructed as:

String SQLQuery =”SELECT Username, Password FROM users WHERE Username=” or 0=0” AND Password=” or 0=0”;

```
String SQLQuery ="SELECT Username, Password
    FROM users WHERE Username=" or 0=0"
        "' AND Password=" or 0=0" "'";
```

Since under all circumstances, zero will be equal to zero, the query will return all records in the database. In this way, an unauthorized user will be able to view sensitive information.

Preventing SQL Injection

- You can prevent SQL injection if you adopt an input validation technique in which user input is authenticated against a set of defined rules for length, type and syntax and also against business rules.
- You should ensure that users with the permission to access the database have the least privileges. Additionally, do not use system administrator accounts like “sa” for web applications. Also, you should always make sure that a database user is created only for a specific application and this user is not able to access other applications. Another method for preventing SQL injection attacks is to remove all stored procedures that are not in use.
- Use strongly typed parameterized query APIs with placeholder substitution markers, even when calling stored procedures.
- Show care when using stored procedures since they are generally safe from injection. However, be careful as they can be injectable (such as via the use of exec () or concatenating arguments within the stored procedure).

State-Based attacks:

The concept of **state**, or the ability to remember information as a user travels from page to page within a site, is an important one for Web testers. The Web is stateless in the sense that it does not remember which page a user is viewing or the order in which pages may be viewed. A user is always free to click the Back button or to force a page to reload. Thus, developers of Web applications must take it upon themselves to code state information so they can enforce rules about page access and session management.

The first option is using **forms** and **CGI parameters**, which allow the transfer of small amounts of data and information to be passed from page to page, essentially allowing the developer to bind together pairs of pages. More sophisticated state requirements mean that data needs to be stored, either on the client or the server, and then made available to various pages that have to check these values when they are loaded.

Attack: Hidden Fields

- Something that is part of page, but not shown
- Defined parameters & assigned values
- Sent back to server to communicate state
- Part of a form

Hidden Field Example

```
<html>
    <head>
        <title>My Page</title>
```

```
        </head>
    <body>
    <form name="myform” action="http://www.foo.com/form.php” method="POST">
    <div align="center">
    <input type="text" size="25" value="Enter your name here!">
    <input type="hidden" name="Language” value="English">
    <br><br>
    </div>
    </form>
    </body>
    </html>
```

Hidden Field Attack

- Look for them (scan source)
- Change their values

Protection - Hidden Fields

- Use misleading names for hidden fields
- Use hashed/encrypted values
- Don’t use them!
- Use them for innocuous input only
- Treat as user input!
- Check their values (content filtering again)

CGI Parameters

- Part of the GET request with URL
- Name-Value pairs
- At end of URL (after?)
- Pairs separated by &
  http://www.foo.com/script.php?user=mike&passwd=guessWho
- Data is sent to server for processing

Parameter Attack

- View source
- Observe URL targets
- In status bar
- In tool tip
- Edit request URL by hand
  http://www.foo.com/script.php?user=mike&passwd=bigDummy
- Try changing values
  Record = notMine

  Item = 6789
- Adding parameters
  Debug = on

  Debug = true

  Debug = 1

Protection – Parameters

- Treat as user input!
- Check values (content filtering again)

GET - POST, Who-What?

- Both send info to server
- GET
  - Per W3C, for idempotent form processing

- No side-effect (e.g., database search)

- POST
  - For transactions that change state on server
  - Have a side-effect (e.g., database update/insertion)
  - For sending large requests
  - Reload/refresh a GET vs a POST form – different behaviors

Cookie Poisoning: How to Off the Cookie Monster

➢ Cookies revisited
- Local file to store data
- Data about you, session, etc...
- Plain text

➢ Change information in cookie file
- Expiration date
- Authentication data
- Shopping cart

Protection - Cookies

- Don’t use cookies for any authentication data
- Encrypt critical data - not fool proof
- Treat cookies as user input
- Be careful what you trust!

Chapter 7:

Software Development Methodologies

A software development methodology is a way of managing a software development project. This typically address issues like selecting features for inclusion in the current version, when software will be released, who works on what, and what testing is done.

No one methodology is best for all situations. Even the much maligned waterfall method is appropriate for some organizations. In practice, every organization implements their software development project management in a different way, which is often slightly different from one project to the next. None the less, nearly all are using some subset or combination of the ones discussed here.

Choosing an appropriate management structure can make a big difference in achieving a successful end result when measured in terms of cost, meeting deadlines, client happiness, robustness of software, or minimizing expenditures on failed projects. As such, it is worth your time to learn about a number of these and make your best effort to choose wisely.

Agile family - Agile methods are meant to adapt to changing requirements, minimize development costs, and still give reasonable quality software. Agile projects are characterized by many incremental releases each generated in a very short period of time. Typically all members of the team are involved in all aspects of planning, implementation, and testing. This is typically used by small teams, perhaps nine or fewer, who can have daily face-to-face interaction. Teams may include a client representative. There is a strong emphasis on testing as software is written. The disadvantages of the Agile methods are that they work poorly for projects with hundreds of developers, or lasting decades, or where the requirements emphasize rigorous documentation and well documented design and testing.

- SCRUM - is currently the most popular implementation of the agile ideals. Features are added in short sprints (usually 7-30 days), and short frequent meetings keep people focused. Tasks are usually tracked on a scrum board. The group is self-organizing and collaboratively managed, although there is a scrum master tasked with enforcing the rules and buffering the team from outside distractions.

- Dynamic Systems Development Model (DSDM) - is an agile method that sets time, quality, and cost at the beginning of the project. This is accomplished by prioritizing features into musts, shoulds, coulds, and won't haves. Client involvement is critical to setting these priorities. There is a pre-project planning phase to give the project a well-considered initial direction. This works well if time, cost, and quality are more important than the completeness of the feature set.

- Rapid Application Development (RAD) - is a minimalist agile method with an emphasis on minimizing planning, and a focus on prototyping and using reusable components. This can be the best choice when a good prototype is good enough to serve as the final product. RAD has been criticized because the lack of structure leads to failed projects or poor quality products if there is not a team of good developers that feel personally committed to the project.

- Extreme Programming (XP) - is a frequent release development methodology in which developers work in pairs for continuous code review. This gives very robust, high quality software, at the expense of twice the development cost. There is a strong emphasis on test driven development.

- Feature-Driven Development (FDD) - is an iterative development process with more emphasis on planning out the overall architecture, followed by implementing features in a logical order.

- Internet-Speed Development - is an iterative format that emphasizes daily builds. It is tailored to the needs of open source projects where volunteer developers are geographically distributed, and working around the clock. The project is built from a vision and scope statement, but there are no feature

freezes. Development is separated into many small pieces that can be developed in parallel. The down side of this process is that the code is constantly in flux, so there are not necessarily stable release points where the code is particularly well tested and robust.

Agile: Strengths and Weaknesses

Strengths

- … Iterative-incremental process
- … Based on modeling the problem domain and the system
- … Requirements are allowed to evolve over time.
- … Traceability to requirements through the Product Backlog
- … Architecture of the system drafted before the development engine is started.
- … Iterative development engine governed by careful planning and reviewing planning and
- … Active user involvement
- … Simple and straightforward process
- … Simple and straightforward process
- … Early and frequent releases, demonstrating functionality at the end of each iteration

(Sprint) of the development cycle.

Weaknesses

- … Integration is done after all increments are built
- … Lack of scalability
- … Lack of scalability
- … Based on the assumption that human communication is sufficient for running projects of

Any size and keeping them focused

- … Not necessarily seamless (details of tasks are not prescribed)
- … No clear-cut design effort
- … Model-phobic
- … Models are not prescribed, leaving it to the developer to decide what

model can be useful
- … Lack of formalism

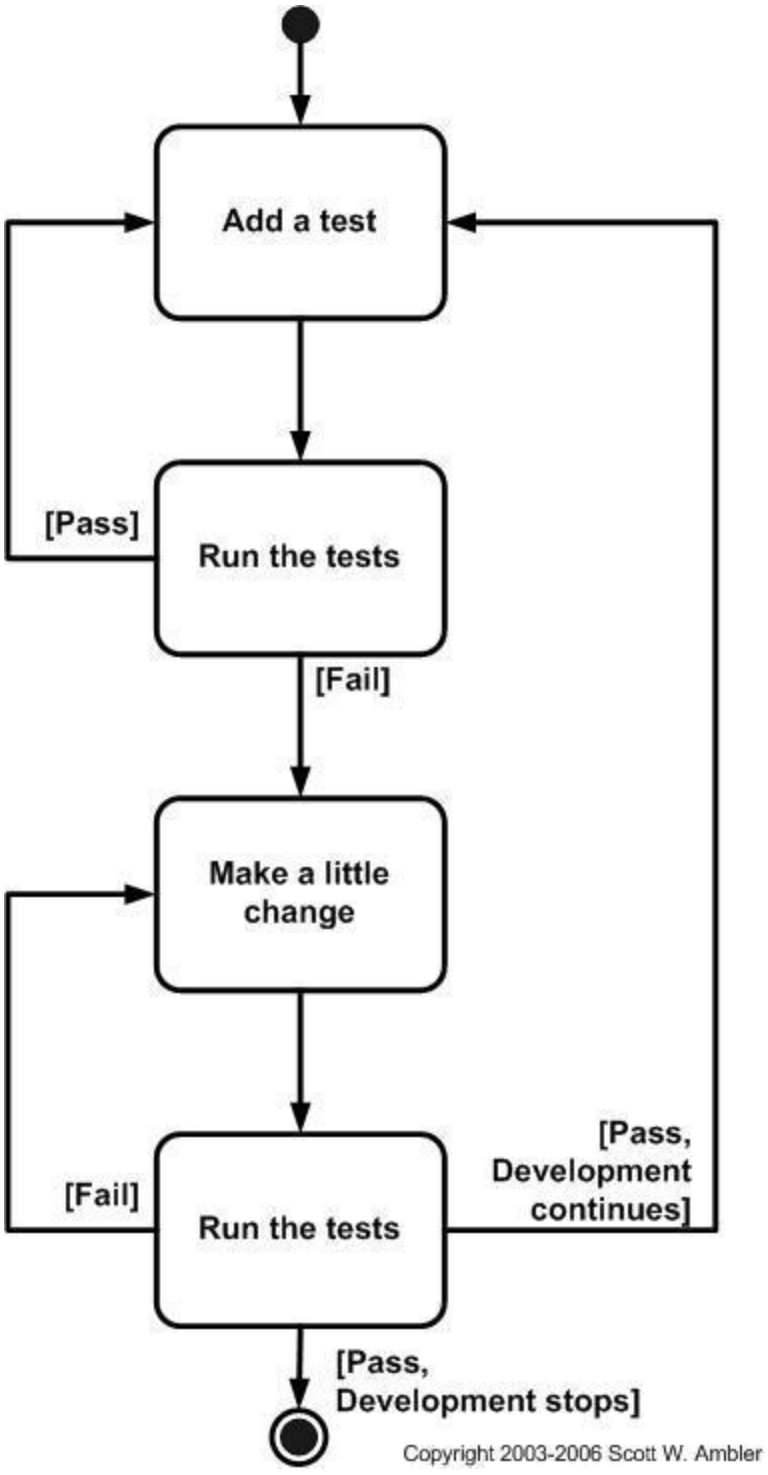
Test Driven Development

Test-driven development (TDD) (Beck 2003; Astels 2003), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. What is the primary goal of TDD? One view is the goal of TDD is specification and not validation (Martin, Newkirk, and Kess 2003). In other words, it's one way to think through your requirements or design before your write your functional code (implying that TDD is both an important agile requirements and agile design technique). Another view is that TDD is a programming technique.

What is TDD?

The steps of test first development (TFD) are overviewed in the UML activity diagram of Figure 1. The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).

Figure 1. The Steps of test-first development (TFD).



TDD can be describe with this simple formula:

TDD = Refactoring + TFD.

TDD completely turns traditional development around. When you first go to implement a new feature, the first question that you ask is whether the existing design is the best design possible that enables you to implement that functionality. If so, you proceed via a TFD approach. If not, you refactor it locally to change the portion of the design affected by the new feature, enabling you to add that feature as easy as possible. As a result you will always be improving the quality of your design, thereby making it easier to work with in the future.

Instead of writing functional code first and then your testing code as an afterthought, if you write it at all, you instead write your test code before your functional code. Furthermore, you do so in very small steps – one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn't present. In fact, they refuse to add even a single line of code until a test exists for it. Once the test is in place they then do the work required to ensure that the test suite now passes (your new code may break several existing tests as well as the new one). This sounds simple in principle, but when you are first learning to take a TDD approach it proves require great discipline because it is easy to "slip" and write functional code without first writing a new test. One of the advantages of pair programming is that your pair helps you to stay on track.

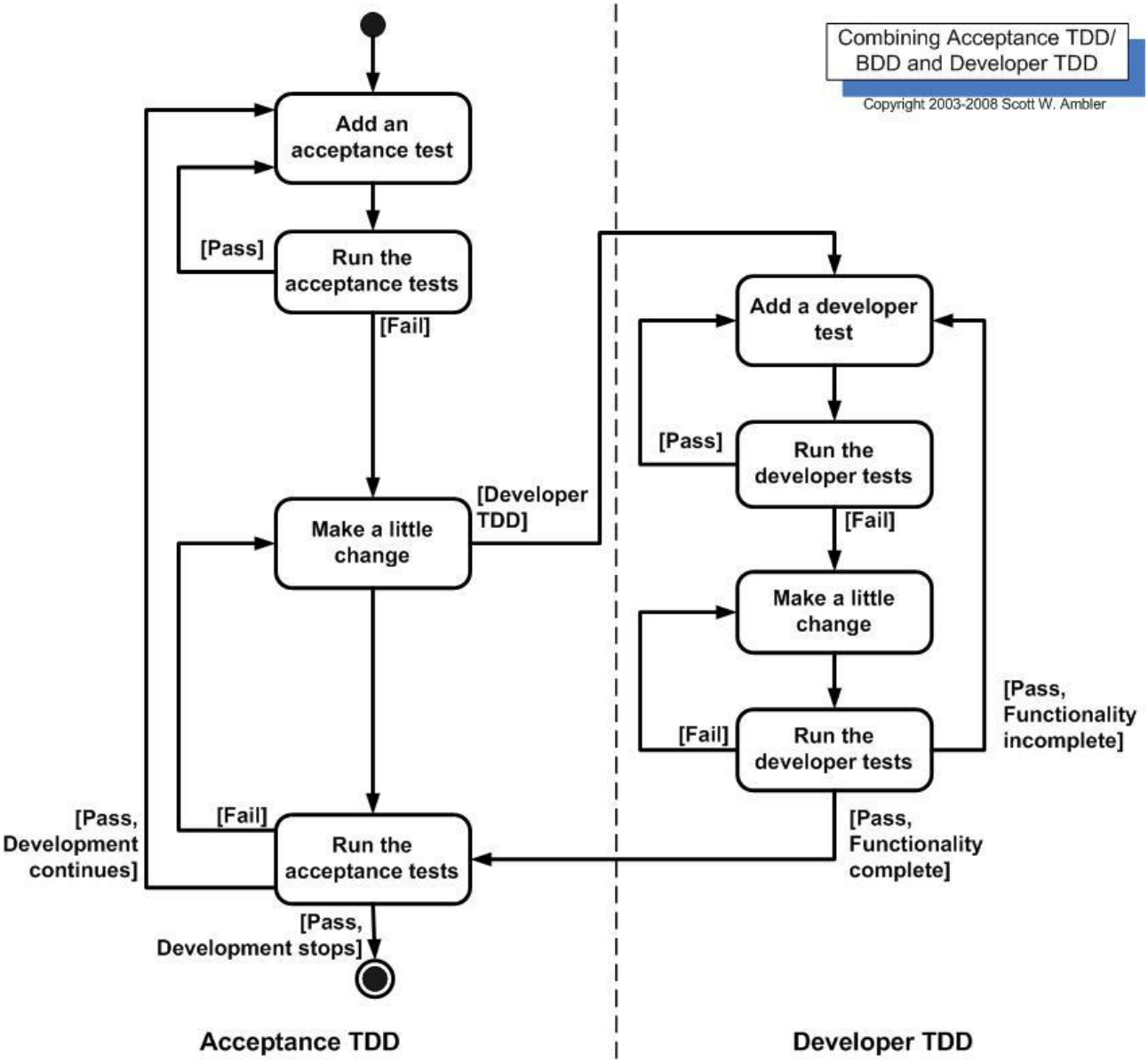
There are two levels of TDD:

1. **Acceptance TDD (ATDD)**. With ATDD you write a single acceptance test, or behavioral specification depending on your preferred terminology, and then just enough production functionality/code to fulfill that test. The goal of ATDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis. ATDD is also called Behavior Driven Development (BDD).
2. **Developer TDD**. With developer TDD you write a single developer test, sometimes inaccurately referred to as a unit test, and then just enough production code to fulfill that test. The goal of developer TDD is to specify a

detailed, executable design for your solution on a JIT basis. Developer TDD is often simply called TDD.

Figure 2 depicts a UML activity diagram showing how ATDD and developer TDD fit together. Ideally, you'll write a single acceptance test, then to implement the production code required to fulfill that test you'll take a developer TDD approach. This in turn requires you to iterate several times through the write a test, write production code, get it working cycle at the developer TDD level. -

Figure 2. How acceptance TDD and developer TDD work together.

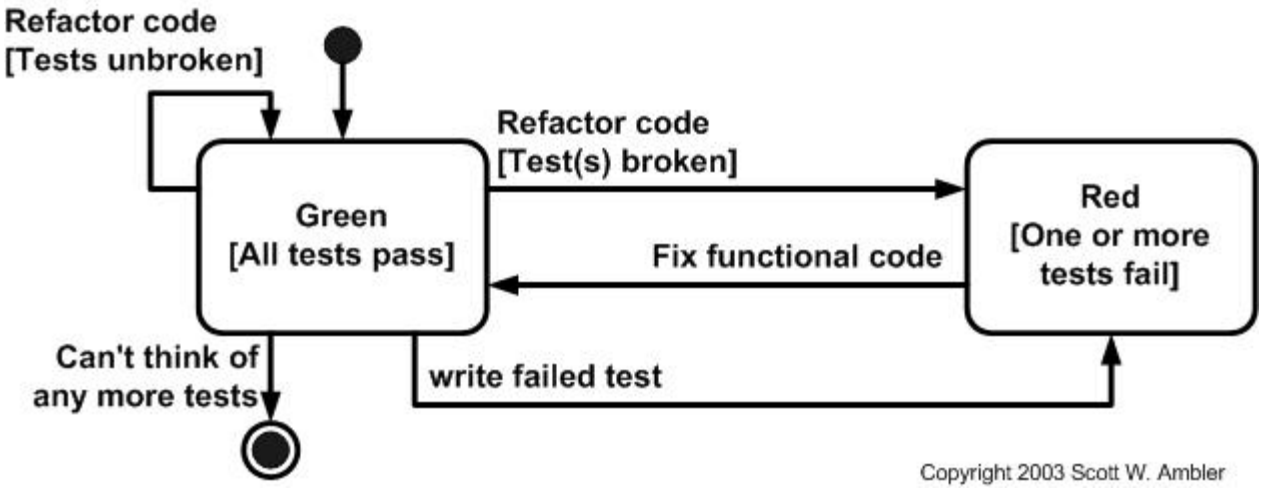


Note that Figure 2 assumes that you're doing both, although it is possible to do either one without the other. In fact, some teams will do developer TDD without doing ATDD, see survey results below, although if you're doing ATDD then it's pretty much certain you're also doing developer TDD. The challenge is that both forms of TDD require practitioners to have technical testing skills, skills that many requirement professionals often don't have (yet another reason why generalizing specialists are preferable to specialists).

An underlying assumption of TDD is that you have a testing framework available to you. For acceptance TDD people will use tools such as Fitnesse or RSpec and for

developer TDD agile software developers often use the xUnit family of open source tools, such as JUnit or VBUnit, although commercial tools are also viable options. Without such tools TDD is virtually impossible. Figure 3 presents a UML state chart diagram for how people typically work with such tools. This diagram was suggested by Keith Ray. -

Figure 3. Testing via the xUnit Framework.



Kent Beck, who popularized TDD in eXtreme Programming (XP) (Beck 2000), defines two simple

TDD (Beck 2003). First, you should write new business code only when an automated test has f

Second, you should eliminate any duplication that you find. Beck explains how these two simpl

generate complex individual and group behavior:

- You develop organically, with the running code providing feedback between decisions.
- You write your own tests because you can't wait 20 times per day for someone else to write them for you.
- Your development environment must provide rapid response to small changes (e.g you need a fast compiler and regression test suite).

- Your designs must consist of highly cohesive, loosely coupled components (e.g. your design is highly normalized) to make testing easier (this also makes evolution and maintenance of your system easier too).

For developers, the implication is that they need to learn how to write effective unit tests. Beck's experience is that good unit tests:

- Run fast (they have short setups, run times, and break downs).
- Run in isolation (you should be able to reorder them).
- Use data that makes them easy to read and to understand.
- Use real data (e.g. copies of production data) when they need to.
- Represent one step towards your overall goal.

TDD and Traditional Testing

TDD is primarily a specification technique with a side effect of ensuring that your source code is thoroughly tested at a confirmatory level. However, there is more to testing than this. Particularly at scale you'll still need to consider other agile testing techniques such as pre-production integration testing and investigative testing. Much of this testing can also be done early in your project if you choose to do so (and you should). With traditional testing a successful test finds one or more defects.

It is the same with TDD; when a test fails you have made progress because you now know that you need to resolve the problem. More importantly, you have a clear measure of success when the test no longer fails. TDD increases your confidence that your system actually meets the requirements defined for it, that your system actually works and therefore you can proceed with confidence.

As with traditional testing, the greater the risk profile of the system the more thorough your tests need to be. With both traditional testing and TDD you aren't striving for perfection, instead you are testing to the importance of the system. To paraphrase Agile Modeling (AM), you should "test with a purpose" and know why you are testing something and to what level it needs to be tested. An interesting side effect of TDD is that you achieve 100% coverage test – every single line of code is tested – something that traditional testing doesn't guarantee (although it does recommend it). In general I think it's fairly safe to say that although TDD is a specification technique, a valuable side effect is that it results in significantly better code testing than do traditional techniques.

Comparing TDD and AMDD:

- TDD shortens the programming feedback loop whereas AMDD shortens the modeling feedback loop.
- TDD provides detailed specification (tests) whereas AMDD is better for thinking through bigger issues.
- TDD promotes the development of high-quality code whereas AMDD promotes high-quality communication with your stakeholders and other developers.
- TDD provides concrete evidence that your software works whereas AMDD supports your team, including stakeholders, in working toward a common understanding.
- TDD "speaks" to programmers whereas AMDD speaks to business analysts,

stakeholders, and data professionals.

- TDD is provides very finely grained concrete feedback on the order of minutes whereas AMDD enables verbal feedback on the order minutes (concrete feedback requires developers to follow the practice Prove It with Code and thus becomes dependent on non-AM techniques).
- TDD helps to ensure that your design is clean by focusing on creation of operations that are callable and testable whereas AMDD provides an opportunity to think through larger design/architectural issues before you code.
- TDD is non-visually oriented whereas AMDD is visually oriented.
- Both techniques are new to traditional developers and therefore may be threatening to them.
- Both techniques support evolutionary development.

Which approach should you take? The answer depends on your, and your teammates, cognitive preferences. Some people are primarily "visual thinkers", also called spatial thinkers, and they may prefer to think things through via drawing. Other people are primarily text oriented, non-visual or non-spatial thinkers, who don't work well with drawings and therefore they may prefer a TDD approach. Of course most people land somewhere in the middle of these two extremes and as a result they prefer to use each technique when it makes the most sense. In short, the answer is to use the two techniques together so as to gain the advantages of both.

How do you combine the two approaches? AMDD should be used to create models with your project stakeholders to help explore their requirements and then to explore those requirements sufficiently in architectural and design models (often simple sketches). TDD should be used as a critical part of your build efforts to ensure that you develop clean, working code. The end result is that you will have a high-quality, working system that meets the actual needs of your project stakeholders.

Why TDD?

A significant advantage of TDD is that it enables you to take small steps when writing software. This is a practice that have promoted for years because it is far more productive than attempting to code in large steps. For example, assume you add some new functional code, compile, and test it. Chances are pretty good that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps. I generally prefer to add a few new lines of functional code, typically less than ten, before I recompile and rerun my tests.

Behavior Driven:

Behavior-driven development (BDD) is a software development methodology in which an application is specified and designed by describing how its behavior should appear to an outside observer.

A typical business application project would begin by having stakeholders offer concrete examples of the behavior they expect to see from the system. All coding efforts are geared toward delivering these desired behaviors. The real-life examples gleaned from stakeholders are converted into acceptance criteria with validation tests that are often automated. The results of these tests provide confidence to stakeholders that their desired business objectives for the software are being achieved. Ideally, the reports are generated in such a way that the average stakeholder can understand the business logic of the application. Living documentation is used throughout the system to ensure that all documentation is up to date and accurate.

In practice, behavior-driven development may be similar to test-driven development when all stakeholders have programming knowledge and skills.

However, in many organizations, BDD offers the ability to enlarge the pool of input and feedback to include business stakeholders and end users who may have little software development knowledge. Because of this expanded feedback loop, BDD may more readily be used in continuous integration and continuous delivery environments.

BDD practices

The practices of BDD include:

- Establishing the goals of different stakeholders required for a vision to be implemented
- Drawing out features which will achieve those goals using feature injection
- Involving stakeholders in the implementation process through outside–in software development
- Using examples to describe the behavior of the application, or of units of code
- Automating those examples to provide quick feedback and regression testing
- Using 'should' when describing the behavior of software to help clarify responsibility and allow the software's functionality to be questioned
- Using 'ensure' when describing responsibilities of software to differentiate outcomes in the scope of the code in question from side-effects of other elements of code.
- Using mocks to stand-in for collaborating modules of code which have not yet been written .

BDD vs TDD

• It is important to keep BDD distinct from TDD. These two practices are equally important but address different concerns and should be complementary in best development practices.

• BDD is concerned primarily with the specification of the behavior of the system under test as a whole, thus is particularly suited for acceptance and regression testing. TDD is concerned primarily with the testing of a component as a unit, in isolation from other dependencies, which are

typically mocked or stubbed.

• BDD should talk the language of the business domain and not the language of the development technology, which on the other hand is “spoken” by TDD.