

**HANDOUTS | MATERIAL
HIGHLIGHTED AND PROVIDED BY :**

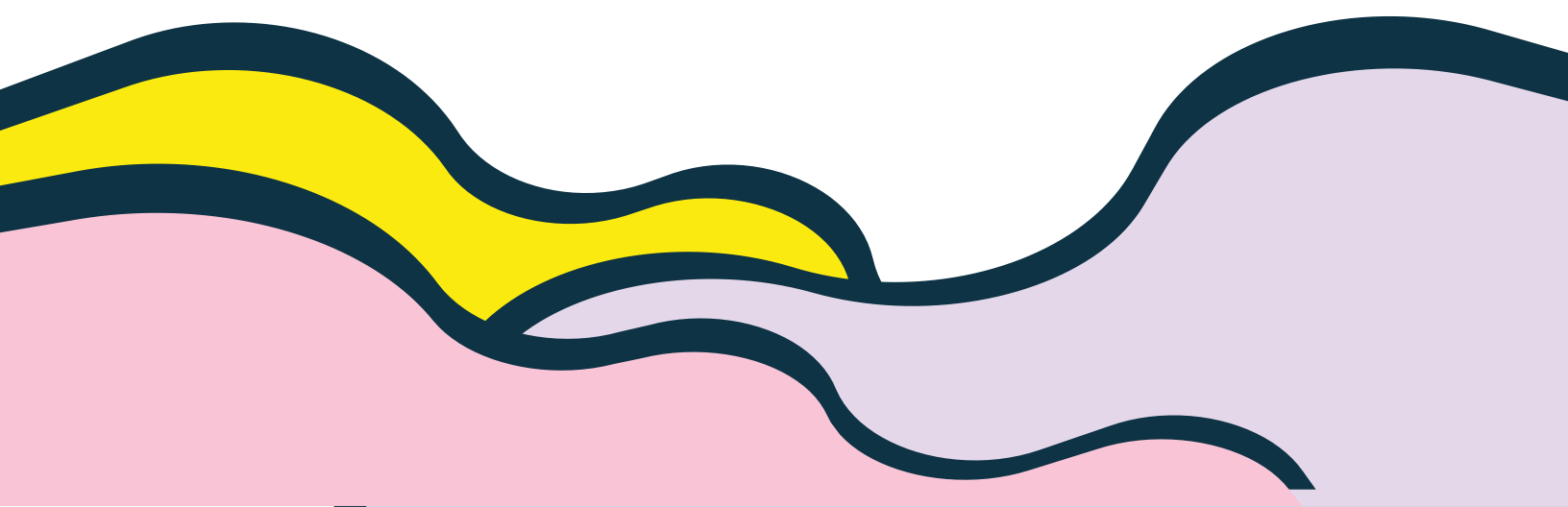
VU TALK

CLICK TO JOIN OUR WHATSAPP GROUP : [click me](#)

VU TALK YouTube CHANNEL : [click me](#)

For Any Help Contact Me On Whatsapp : [click me](#)

Follow VU TALK on Whatsapp Channel : [click me](#)



SE602 Handouts

Week 1

A Fairy Tale

Once upon a time there was a Good Software Engineer whose Customers knew exactly what they wanted. The Good Software Engineer worked very hard to design the Perfect System that would solve all the Customers' problems now and for decades. When the Perfect System was designed, implemented and finally deployed, the Customers were very happy indeed. The Maintainer of the System had very little to do to keep the Perfect System up and running, and the Customers and the Maintainer lived happily every after.

Manny Lehman and Les Belady Laws

The Law of Continuing Change:

A program that is used in a real world environment must change, or become progressively less useful in that environment.

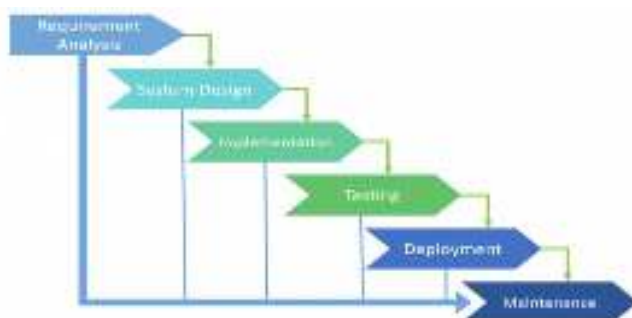
The Law of Increasing Complexity :

As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

Software Engineering

Software Engineering is the application of principles used in the field of engineering, which usually deals with physical systems, to the design, development, testing, deployment and management of software systems.

Traditional SDLC



Definitions of Re-engineering

- **Re-organising** and **modifying** existing software systems to make them more **maintainable**
- The **examination** of a subject system to re-constitute it in a new form and the subsequent implementation of the new form.
- Re-engineering is the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer.
- **Software Re-engineering** is a process of software development which is done to improve the maintainability of a software system. Re-engineering is the examination and alteration of a system to reconstitute it in a new form. This process encompasses a combination of sub-processes like reverse engineering, forward engineering, reconstructing etc.

Goals of Re-engineering are:

To understand artifacts of the existing software system, namely specification, design, implementation, and documentation.

To improve the functionality and quality attributes of the system.

Risks involved in Re-engineering are:

- The target system may not have the same functionality as the existing system
- The target system may be of lower quality than the original system
- The benefits are not realized in the required time frame

Objectives of Re-engineering

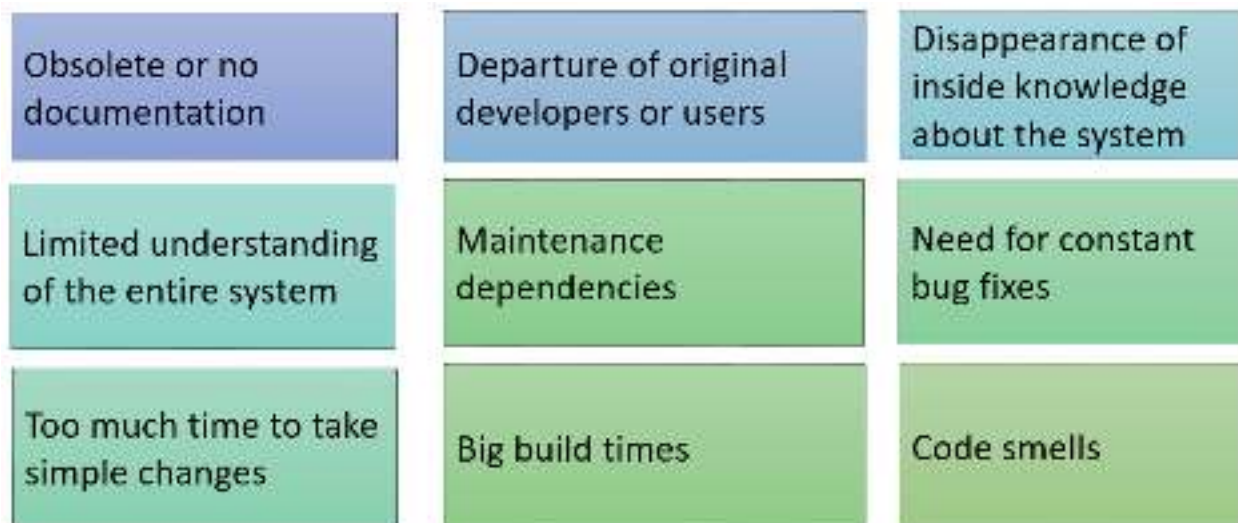
Software systems are re-engineered by keeping one or more of the following four general objectives in mind:



Motivation for Re-engineering

- Software system is an integral part of the organization
- System must be regularly maintained and maintenance is becoming a large cost factor.
- Less risk and cost than re-development.

When do we Re-engineer?



Why do we Re-engineer?

- We might want to unbundle a **monolithic system**, so that the individual parts can be more easily marketed separately or combined in different ways.
- Performance improvement.
- We might want to port the system to a new platform.
- We might want to extract the design as a first step to a new implementation.
- We might want to exploit new technology, such as emerging standards or libraries, as a step towards cutting maintenance costs.

We might want to reduce human dependencies by documenting knowledge about the system and making it easier to maintain.

Week 2

Software Evolution

A continual change from a lesser, simpler, or worse state to a higher or better state.

Software Development Life Cycle

- Requirements
- Analysis
- Design
- Implementation
- Testing
- Deployment
- Maintenance and Evolution

History of Software Evolution

In 1965, Mark Halpern introduced the concept of software evolution.

History of Software Evolution

The concept further attracted attention

after IBM researchers Lehman (right) and Belady (left) published the first effort in 1976 ("A Model of Large Program Development" in IBM Systems Journal) to formulate hypotheses on how and why large software systems evolve.

Evolution Vs Maintenance

- ⑩ **Evolution** means a continual change from a lesser, simpler, or worse state to a higher or better state.
- ⑩ **Maintenance** means preventing software from failing to deliver the intended functionalities by means of bug fixing.

Software Change

Software change is a fundamental ingredient of software evolution and maintenance.

Software change is a process that may

- introduce new requirements to the existing system.

(Evolution)

- fix incorrectly implemented requirements.

(Maintenance)

Change is inevitable because:

- New requirements emerge when the software is used
- The business environment changes
- Errors must be repaired
- New computers and equipment are added to the system
- The performance or reliability of the system may have to be improved

Importance of Evolution

- Organizations have huge investments in their software systems - they are **critical business assets**.
- To maintain the value of these assets to the business, they must be **changed** and **updated**.
- The software system moves to the evolution stage after the initial development is successful.
- **In this stage:**
- Quick patches and new releases are dispatched to the customers.
- Feedback from the customers are received for additional enhancement to the software system.

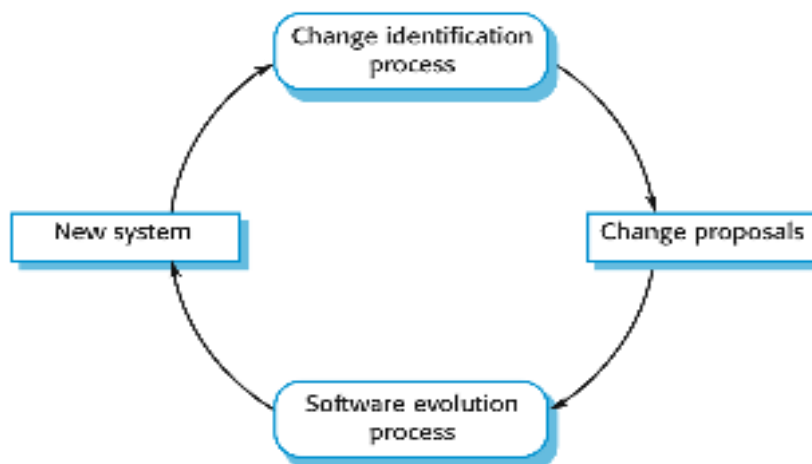
Why Software Systems Evolve?

- **Evolution** of the system may occur due to **changes** in the operating environment and the business practice.
- Customer demands for additional functionalities and competitive products from other vendors cause the system to **evolve**.

Change Identification and Evolution Process

- Change identification and evolution continues throughout the **system lifetime**.
- **Proposals for change** are the driver for system evolution.
- These should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.

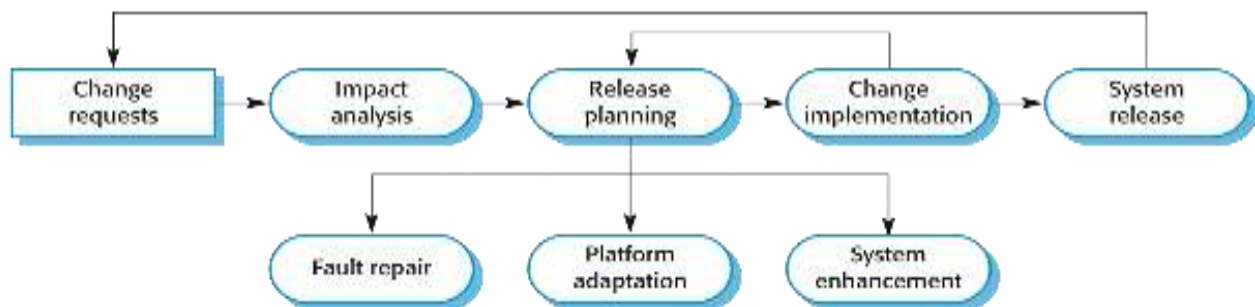
Change Identification and Evolution Process



Software Evolution Process depend on:

- The type of software being maintained
- The development processes used
- The skills and experience of the people involved

The Software Evolution Process



Change Implementation

Change implementation can be viewed as an iteration of the development process where the revisions to the system are designed, implemented and tested.

Change Implementation



Change Implementation

The first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.

Change Implementation

During the program understanding phase, you have to understand:

- How the program is structured?
- How it delivers functionality?
- How the proposed change might affect the program?

Week 3

Software Maintenance

To prevent a software from failing to deliver its intended functionalities by means of bug fixing.

Maintenance of software systems comprises all activities associated with the process of changing software for the purposes of:

- Fixing bugs
- Improving the design of the system so that future changes to the system are less expensive.

Software Maintenance

- It does not involve making major changes to the architecture of the system.
- It means keeping an installed system running with no change to its design.
- Maintenance tasks are very much planned. For example, bug fixing must be done and it is a planned activity.

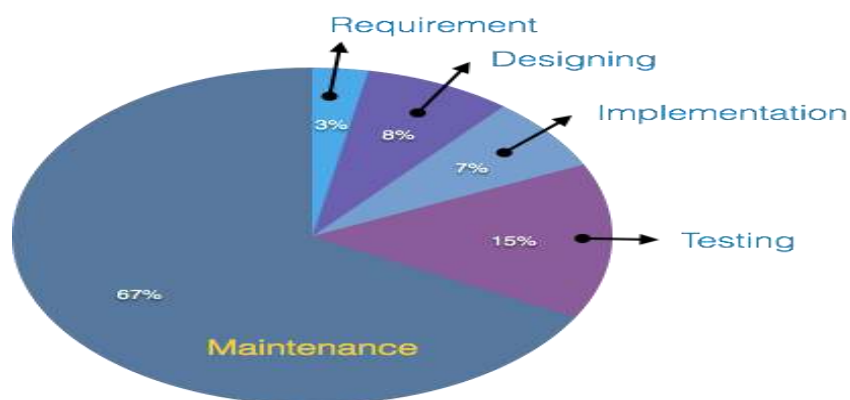
Maintenance of Hardware/Physical systems

- In hardware maintenance, a system or a component is returned to its original good state.
- Maintenance of physical systems often requires replacing broken and worn-out parts.

For example:

- An owner replaces his worn-out tires and broken lamps of his car.
- A malfunctioning memory card is replaced with a good one.

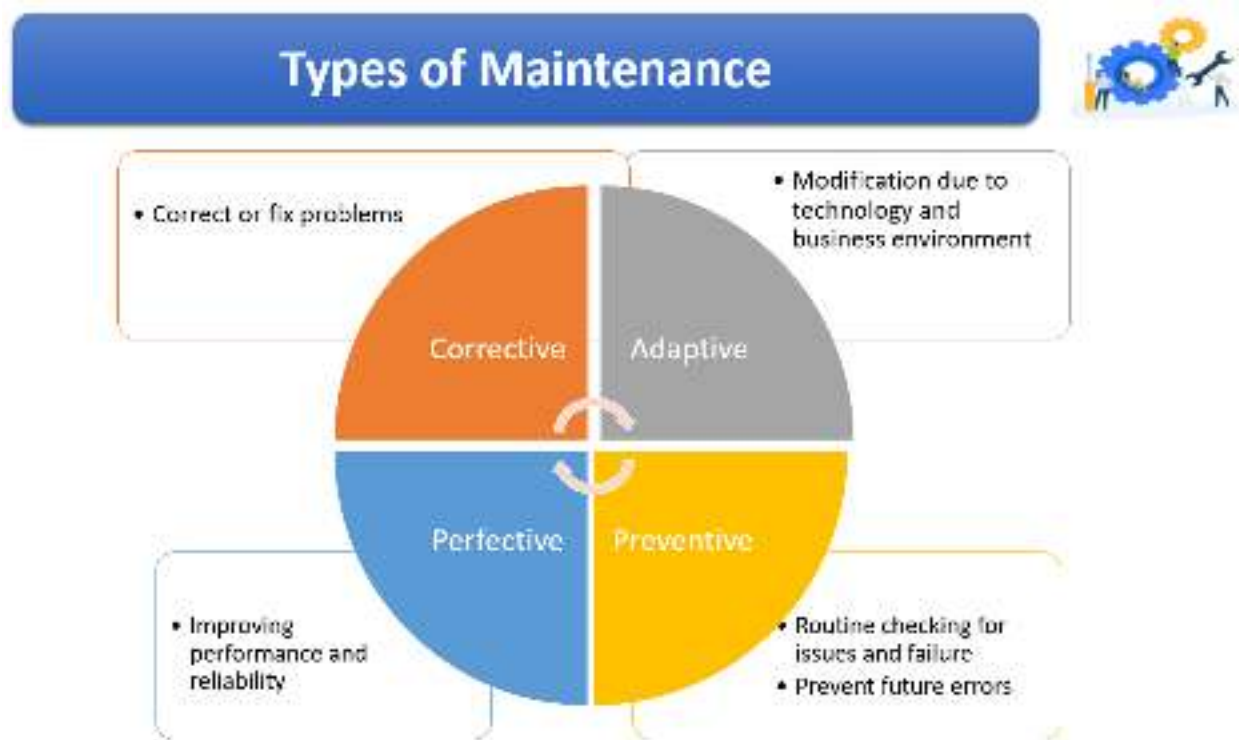
Cost of Maintenance



Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.

Types of Maintenance

- Maintenance
 - Corrective Maintenance
 - Adaptive Maintenance
 - Perfective Maintenance
 - Preventive Maintenance



Corrective Maintenance

This includes modifications and updates done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.

Adaptive Maintenance

This includes modifications and updates applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.

Perfective Maintenance

This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.

Preventive Maintenance

This includes modifications and updates to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

Maintenance Cost Factors

- Maintenance costs factors are usually greater than development costs (2x to 100x depending on the application).
- Costs are affected by both technical and non-technical factors.
- Aging software can have high support costs (e.g. old languages, compilers etc.).

Maintenance cost factors include:

- Team Stability
- Contractual Responsibility
- Staff Skills
- Program age and Structure

1. Team Stability:

Maintenance costs are reduced if the same staff are involved with them for some time.

2. Contractual Responsibility:

The developers of a system may have no contractual responsibility for maintenance, so there is no incentive to design for future change.

3. Staff Skills:

Maintenance staff are often inexperienced and have limited domain knowledge.

4. Program age and Structure:

As programs age, their structure is degraded and they become harder to understand and change.

Maintenance Prediction

- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs.
- Tightly coupled systems require changes whenever the environment is changed.

Maintenance Prediction

Predicting the number of changes requires an understanding of the relationships between a system and its environment.

Factors influencing this relationship are:

- Number and complexity of system interfaces.
- Number of inherently volatile system requirements.
- The business processes where the system is used.

Complexity depends on:

- Complexity of control structures
- Complexity of data structures
- Object, method (procedure) and module size

Evolution Vs. Maintenance

Another flaw in the human character is that everybody wants to build and nobody wants to do maintenance.

- The concept of software maintenance means preventing software from failing to deliver the intended functionalities by means of bug fixing.
- The concept of software evolution means a continual change from a lesser, simpler, or worse state to a higher or better state.

Week 4

Legacy

A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor.

Legacy System

A legacy system is a piece of software that:

- you have *inherited*, and
- is *valuable* to you.

Legacy Systems are a Reality:

- You must constantly be prepared to adapt to a changing business environment.
- The software that you use to keep your business running must therefore also be adaptable.
- Legacy system can neither be replaced nor upgraded except at a high cost.

Typical Problems with Legacy Systems

- Original developers **not available**
- **Outdated** development methods used
- Extensive patches and **modifications** have been made
- **Missing** or outdated documentation
- **Further** evolution and development may be **too expensive**

A legacy project not only focus on code but encompasses many other aspects, including:

- Build tools and scripts
- Dependencies on other systems
- The infrastructure on which the software runs
- Project documentation
- Methods of communication, such as between developers, or between developers & stakeholders

Characteristic of a Legacy Project

- Old
- Large
- Inherited
- Poorly Documented

Characteristic of A Legacy Project:

1. OLD

- Usually a project needs to exist for a few years before it gains enough entropy to become really difficult to maintain.
- In that time, it will also go through a number of generations of maintainers.
- With each of these handoffs, knowledge about the original design of the system and the intentions of the previous maintainer is also lost.

2. LARGE

- It goes without saying that the larger the project is, the more difficult it is to maintain.
- There is more code to understand, a larger number of existing bugs (if we assume a constant defect rate in software, more code = more bugs), and a higher probability of a new change causing a regression, because there is more existing code that it can potentially affect.
- The size of a project also affects decisions about how the project is maintained.
- Large projects are difficult and risky to replace, so they are more likely to live on and become legacy.

3. INHERITED

- As is implied by the common meaning of the word legacy, these projects are usually inherited from a previous developer or team.
- In other words, the people who originally wrote the code and those who now maintain it are not the same individuals, and they may even be separated by several intermediate generations of developers.
- This means that the current maintainers have no way of knowing why the code works the way it does, and they're often forced to guess the intentions and tacit design assumptions of the people who wrote it.

1. OLD

Old projects become harder to maintain
Developers keep changing over time
Original knowledge and design intent gets lost

2. LARGE

Bigger projects are harder to manage
More code means more bugs and more complexity
Small changes can affect many parts of the system
Large systems are hard to replace, so they become legacy

3. INHERITED

Code is passed from previous developers/teams
Current developers are not the original creators
It's hard to understand the original purpose of code

4. POORLY DOCUMENTED

Documentation is often missing or outdated
Developers don't regularly update docs
Old documentation cannot be fully trusted

4. POORLY DOCUMENTED

- Given that the project spans multiple generations of developers, it would seem that keeping accurate and thorough documentation is essential to its long-term survival.
- Unfortunately, if there's one thing that developers enjoy less than writing documentation, it's keeping that documentation up to date.
- So any technical documents that do exist must invariably be taken with a pinch of salt.

Legacy Projects.. an Example

- Linux Kernel is since 1991, so it is an old software
- Has more than 15 Million lines of code.

Legacy Code

- Most important part of the legacy project
- Untestable or untested code
- Implementing new features or changes to existing behavior is difficult (Inflexible code)
- Code encumbered by technical debt

Legacy Code..

Legacy code refers to existing software code that is old and difficult to maintain, understand, or modify, but is still in use because it works and is important for the system.

It is often more useful and appropriate to ship something that **works** than to spend excessive amounts of time striving for a paragon of algorithmic excellence.

Legacy Infrastructure

- Software depends on tools and infrastructure in order to run.
- Quality of these tools can also have a dramatic effect on a team's productivity.
- Setting your Development Environment.
- Checking out the code from version control system.
- Setup process will take time.

Setting Development Environment

Setting up a legacy project often involves a combination of these:

- Downloading, installing, and learning how to run whatever tool the project uses.
- Running the mysterious and un-maintained scripts you found in the project's /bin folder.
- Taking a large number of manual steps, listed on some "how to install" or "readme first" document.
- Might get some briefing from some outgoing or experienced developer.

Dealing with outdated Dependencies

What is a dependency in a project?

- Third party software

For example, a Java servlet application will require:

- Java installed
- Tomcat as servlet container
- Apache as web server
- Java libraries

Upgrades often provide performance improvements and bug fixes.

Dealing with Heterogenous Environment

Most software will run in a number of environments during its lifetime.

- Developers run the software on their local machines.
- They deploy it to a test environment for automatic & manual testing.
- It is deployed to a staging environment that mimics production as closely as possible.
- It is released and deployed to the production environment (or, in case of packaged software(e.g MS Office), it is shipped to the customer).

Dealing with Legacy Culture

- Nobody wants to think of themselves and their culture as legacy.
- Many legacy projects are so complex and poorly documented that even the teams charged with maintaining them don't understand everything about them.
- Which features are no longer in use and are thus safe to delete?

- Which *bugs* are safe to fix? (Some users of the software may be depending on the bug and treating it as a feature.)
- Which users need to be consulted before making any changes?
- Because of this lack of information, many teams come to respect the status quo as the safest option, and become fearful of making any unnecessary changes to the software.
- Any change is seen purely as a risk, and the potential benefits of changes are ignored.
- The biggest problem encountered by developers when writing and maintaining software is often a lack of knowledge.
- A benefit of working in a team is that if you're missing a particular piece of knowledge, a fellow team member might be able to provide it.

Other issues of legacy culture can be:

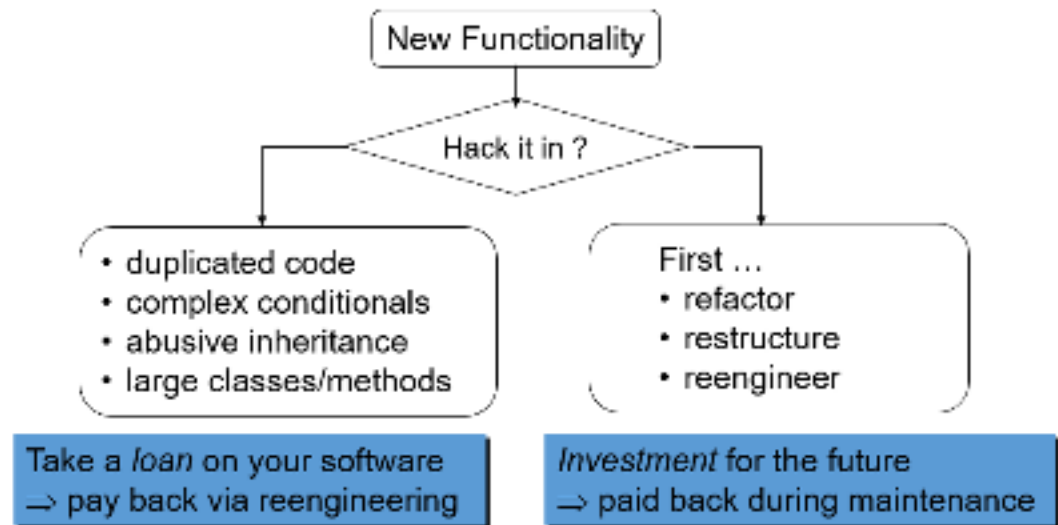
- Lack of face-to-face communication
- Code ego: If I let anybody figure out how my code works, they might criticize it

Key Points

- Legacy software is often large, old, inherited from somebody else, and poorly documented. But there are exceptions to the rule.
- Legacy software often lacks tests and is difficult to test.
- Legacy code is often inflexible, meaning that it takes a lot of work to make a **simple change**.
- **Refactoring can improve the situation.**
- Legacy software is encumbered by years' worth of accumulated technical debt.
- The infrastructure on which the code runs, all the way from the developer's machine through to production, deserves attention.
- The culture of the team that maintains the software can be a barrier to improvement.

How to deal with Legacy Systems ?

New or changing requirements will gradually degrade **original design**... unless **extra ordinary development effort** is spent to adapt the structure.



Common Symptoms

- Lack of Knowledge
- Process Symptoms
- Code Symptoms

1. Lack of Knowledge

- *Obsolete* or no documentation
- *Departure* of the original developers or users
- Lack of inside knowledge about the system
- *Limited understanding* of entire system
- Missing tests

2. Process Symptoms

- Simple changes take too long

- Need for constant bug fixes
- Maintenance dependencies

3. Code Symptoms

- *Duplicated code*
- *Code smells*
- *Big build times*

Architectural Problems

- *Insufficient documentation*
= non-existent or out-of-date
- *Improper layering*
= too few or too many layers
- *Lack of modularity*
= strong coupling
- *Duplicated code*
= copy, paste & edit code
- *Duplicated functionality*
= similar functionality
by separate teams

DLMCF
Docs, Layers, Modules,
Code, Function

Refactoring opportunities

- *misuse* of inheritance
= code reuse vs polymorphism
- *missing* inheritance
= duplication, case-statements
- *misplaced* operations
= operations outside classes
- *violation* of encapsulation
= type-casting; C++ "friends"

Changes	Benefits	Risks	Actions required
Removing an old feature	• Easier development • Better performance	• Somebody is still using the feature	• Check access logs • Ask users
Refactoring	• Easier development	• Accidental regression	• Code review • Testing
Upgrading a library	• Bug fixes • Performance improvements	• Regression due to a change in the library's behavior	• Read the changelog • Review library code • Manually test major features

Changes, Benefits and Risks for a Legacy Project

Week 5

Solutions for legacy systems:

- Freeze
- Outsource
- Carry on maintenance
- Discard and redevelop
- Wrap
- Migrate

1. Freeze

The organization decides to not carry out further work on a legacy system.

2. Outsource

- An organization may decide that supporting legacy (or any) software is not its core business.
- They may outsource it to a specialist organization offering this service.

3. Carry on Maintenance

Despite all the problems associated with supporting a software system, the organization **decides to carry on maintenance** for another period.

4. Discard and Re-develop

The organization throws all the software away and **re-develops** the application once again from scratch, using a new hardware platform and modern architecture tools and database.

This approach removes the legacy problem in one stroke, but also removes a useful source of information for defining the requirements of the new system.

The existing system is discarded in spite of it being stable, thoroughly debugged, and being a useful asset.

But, If the Legacy System cannot keep pace with changing business needs and if migration is not cost effective, then it is more appropriate to **re-develop** the system.

5. Wrap

It is a black-box-based modernization technique: Surround the Legacy System with a new layer of software to hide the complexity of the existing interfaces, applications, and data, with the new interfaces. **Wrapping** gives existing components a **modern** and **improved** appearance.

6. Migrate

A legacy system is ported to a modern platform, while **retaining** most of the system's functionality and introducing minimal disturbance in the business environment.

The target system runs in a different computing environment.

Migration **avoids** the expensive and long process that generally characterize new development.

Rather, reuse of portions of the Legacy system, namely, implementation, design, specification, and requirements, is **maximized**.

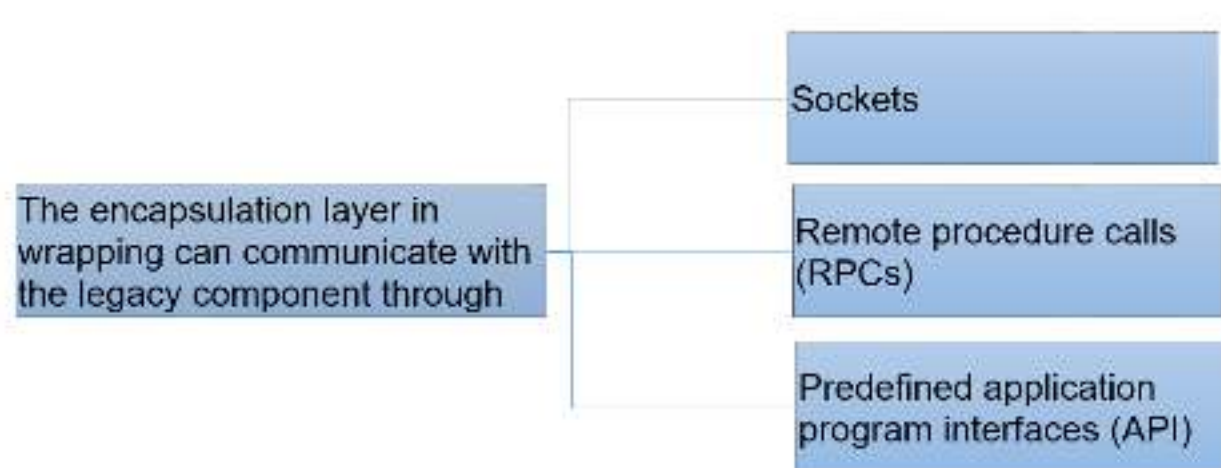
Wrapping

Wrapping is a straight forward way to modify and enhance a legacy component.

A wrapper does not directly modify the source code, but, instead, it **indirectly** modifies and changes the software functionality of the legacy component.

Wrapping means encapsulating the legacy component with a new software layer that provides a new interface and hides the complexity of the old component.

Encapsulation in Wrapping



Message Passing Mechanism in Wrapping

By means of a message passing mechanism, a wrapper connects to the clients.

On the input front: the wrapper

- accepts requests

- restructures requests
- invokes the target object with the restructured arguments

On the output front: the wrapper

- captures outputs from the wrapped entity
- restructures the outputs
- pushes them to the requesting entity

Types of Wrappers

- Wrappers
 - Database wrappers
 - Application wrappers
 - Function wrappers
 - System service wrappers

Migration of a Legacy System

It is the best alternative, When:

- Wrapping is unsuitable; and
- Re-development is not acceptable due to substantial risk.

However, it is a very complex process.

It offers better system understanding, easier maintenance, reduced cost, and more flexibility to meet future business requirements.

Migration involves changes, often including:

- Restructuring the system,
- Enhancing the functionality; or
- Modifying the attributes.

But, it retains the basic functionality of the existing system.

Legacy system migration involves:

- Creation of a modern database from the legacy database; and

- Adaptation of the application program components accordingly.

A database has two main components:

- Schema of the database; and
- Data stored in the database.

Therefore, in general, migration comprises three main steps:

- (i) Conversion of the existing schema to a target schema;
- (ii) Conversion of data; and
- (iii) Conversion of program.

Migration : Schema Conversion

It means translating the legacy database schema into an equivalent database structure expressed in the new technology.

The transformation of source schema to a target schema is made up of two processes:

1. The first one is called DBRE and it aims to recover the conceptual schema that express the semantics of the source data structure.
2. The second process is straightforward and it derives the target physical schema from this conceptual schema:
 - (i) an equivalent logical schema is obtained from the conceptual schema by means of transformations; and
 - (ii) a physical schema is obtained from the logical schema by means of transformations.

Migration : Data Conversion

Data conversion means moving of the data instances from the legacy database to the target database.

Data conversion requires three steps: Extract, Transform, and Load (ETL).

- First, **Extract** data from the legacy store.
- Second, **Transform** the extracted data so that their structures match the format (In addition, perform data cleansing / scrubbing to fix or discard data that do not fit the target database).
- Finally, **Load** the transformed data in the target database.

Migration : Program Conversion

- In this context, program conversion means modifying a program to access the migrated database instead of the legacy data.
- The conversion process leaves the functionalities of the program unchanged.
- Program conversion depends upon the rules that are used in transforming the legacy schema into the target schema.

Migration : Testing and Functionality

It is important to ensure that the outputs of the target system are consistent with those of the LS.

Therefore, new functionality is not to be introduced into the target system in a migration project.

If both the LS and the target system have the same functionality, it is easier to verify their outputs for **compliance**.

However, to justify the project expense and the risk, in practice, migration projects often add new functionalities.

Migration : Cut over or Roll over

The event of cutting over to the new system from the old one is required to cause minimal disruption to the business process.

There are three kinds of transition strategies:

- **Cut-and-Run**
 - **Phased Interoperability**
 - **Parallel Operation**
- **Cut-and-Run:** The simplest transition strategy is to switch off the legacy system and turn on the new system.
 - **Phased Interoperability:** To reduce risks, cut over is gradually performed in incremental steps.
 - **Parallel Operation:** The target system and the LS operate at the same time. Once the new system is considered to be reliable, the LS is taken off service.

Migration Methods

No single approach can be applied to all kinds of legacy systems, because they vary in their scale, complexity, and risks of failure while migrating.

Migration Methods: Seven Approaches are:

- Cold Turkey
- Database First
- Database Last
- Composite Database
- Chicken Little
- Butterfly
- Iterative

Week 6

Re-engineering Concepts

Abstraction and Refinement are key concepts used in software development, and they are equally useful in re-engineering.

It may be recalled that abstraction enables software maintenance personnel to reduce the complexity of understanding a system by:

(i) focusing on the more significant information about the system; and

(ii) Hiding the irrelevant details at the moment.

On the other hand, refinement is the reverse of abstraction.

1. Principle of Abstraction:

The level of abstraction of the representation of a system can be gradually increased by successively replacing the details with abstract information.

By means of abstraction one can produce a view that focuses on selected system characteristics by hiding information about other characteristics.

2. Principle of refinement:

The level of abstraction of the representation of the system is gradually decreased by successively replacing some aspects of the system with more details.

3. Principle of alteration(an optional principle):

The making of some changes to a **system representation** is known as **alteration**.

Alteration does not involve any change to the degree of abstraction, and it does not involve modification, deletion, and addition of information.

The concepts of abstraction and refinement are used to create models of software development as sequences of phases, where the phases map to specific levels of abstraction or refinement, as shown in the Figure:

The four levels are:

- Conceptual,
- Requirements,
- Design, and
- Implementation.

The refinement process:

why? ! what? ! what & how? ! how?

The abstraction process:

how? ! what & how? ! what? ! why?

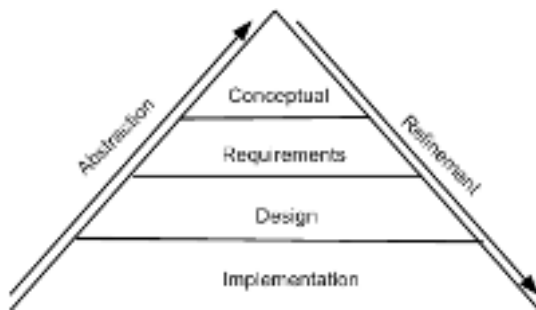


Figure: levels of abstraction and refinement © IEEE, 1992

Re-engineering principles are represented by means of **arrows**. **Abstraction** is represented by an up-arrow, **alteration** is represented by a horizontal arrow, and **refinement** by a down-arrow.

It may be noted that alteration is non-essential for re-engineering.



A General Model For Software Re-engineering

The re-engineering process **accepts as input** the existing code of a system and produces the code of the renovated system.

The re-engineering process may be as straight forward as translating (with a tool) the source code from the given language to source code in another language.

For example, a program written in **BASIC** can be translated into a new program in **C Language**.

The re-engineering process may be very complex as below:

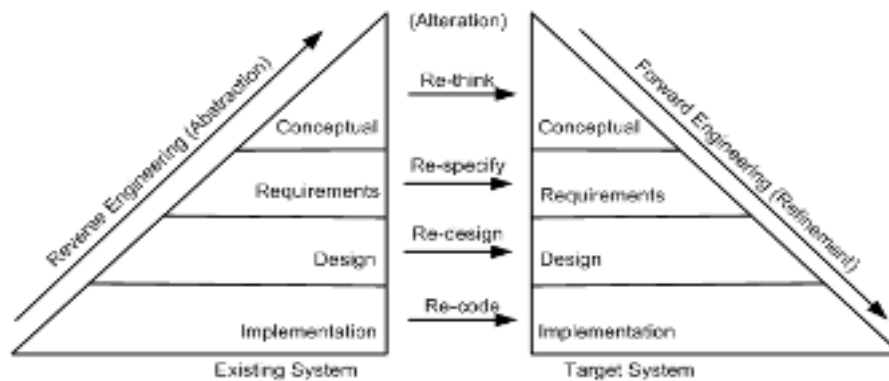
- Recreate a design from the existing source code
- Find the requirements of the system being re-engineered
- Compare the existing requirements with the new ones
- Remove those requirements that are not needed in the renovated system

- Make a new design of the desired system
- Code the new system

The model (shown in the Figure) proposed by **Eric J. Byrne** suggests that re-engineering is a sequence of three activities:

- Reverse engineering, Re-design, and Forward engineering
- strongly founded in three principles, namely, abstraction, alteration, and refinement.

A General Model For Software Re-engineering



Definitions of Software Re-engineering

Now, we are in a position to **re-visit** three definitions of re-engineering.

The definition by Chikofsky and Cross:

Software re-engineering is the analysis and alteration of an operational system to represent it in a new form and to obtain a new implementation from the new form.

Here, a new form means a representation at a higher level of abstraction.

The definition by Byrne:

Re-engineering of a software system is a process for creating a new software from an existing software so that the new system is better than the original system in some ways.

The definition by Arnold:

Re-engineering of a software system is an activity that:

- improves the comprehension of the software system.
- raises the quality levels of the software, i.e performance, reusability, and maintainability.

A General Model for Software Re-engineering

In summary, it is evident that software re-engineering entails:

- the creation of a more abstract view of the system by means of some reverse engineering activities,
- the restructuring of the abstract view, **and**
- implementation of the system in a new form by means of forward engineering activities.

This process is formally captured by Jacobson and Lindstorm with the following expression:

Re-engineering = Reverse engineering + Δ + Forward engineering

- The element " Δ " captures alterations made to the original system.
- Two major dimensions of alteration are: change in functionality and change in implementation technique.
-
- A change in functionality comes from a change in the business rules.
- Next, concerning a change of implementation technique, an end-user of a system never knows if the system is implemented in an object-oriented language or a procedural language.

Rehosting

Another common term used by practitioners of re-engineering.

- **Rehosting** means re-engineering of source code without addition or reduction of features in the transformed targeted source code.
- **Rehosting** is most effective when the user is satisfied with the system's functionality, but looks for better qualities of the system.
- **Examples** of better qualities are: improved efficiency of execution and reduced maintenance costs.

Types of Change

Based on the type of changes required, system characteristics are divided into groups:

- **Re-think**
- **Re-specify**
- **Re-design**
- **Re-code**

Re-code:

- Implementation characteristics of the source program are changed by re-coding it.
- Source-code level changes are performed by means of re-phrasing and program translation.
- **Program translation**
 - A program is transformed into a program in a different language.
 - Examples are compilation, decompilation, and migration.
- **Re-phrasing**
 - It keeps the program in the same language.
 - Examples are normalization, optimization, refactoring, and renovation.

Re-design:

The design characteristics of the software are altered by re- designing

the system. Common changes to the software design include:

- Restructuring the architecture
- Modifying the data model of the system
- Replacing a procedure or an algorithm with a more efficient one

Re-specify:

This means changing the requirement characteristics of the

system in two ways:

- Change the form of the requirements
- Change the scope of the requirements

Re-think:

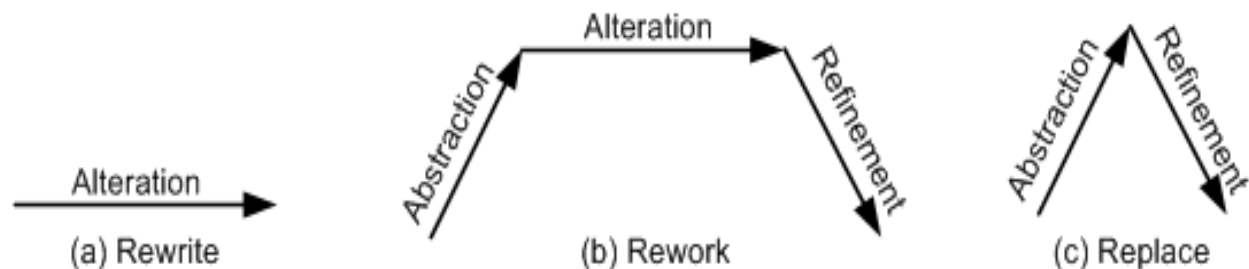
- Re-thinking a system means manipulating the concepts embodied in an existing system to create a system that operates in a different problem domain.
- It involves changing the conceptual characteristics of the system, and it can lead to the system being changed in a fundamental way.
- Moving from the development of an ordinary cellular phone to the development of smartphone system is an example of **Re-think**.

Week 7

Re-engineering Strategies

Three strategies that specify the basic steps of re-engineering are:

- Rewrite strategy
- Rework strategy
- Replace strategy



1. Rewrite Strategy:

This strategy reflects the principle of alteration. By means of alteration, an operational system is transformed into a new system, while preserving the abstraction level of the original system.

For example, C Language code of a system can be rewritten in Python.

1. Rewrite Strategy: Example of printing table of an integer number in C Language.

```
#include <stdio.h>
int main()
{
    int n, i;
    printf("Enter an integer: ");
    scanf("%d", &n);
    for (i = 1; i <= 10; ++i) {
        printf("%d * %d = %d \n", n, i, n * i);
    }
    return 0;
}
```

Output:

Enter an integer: 5

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

1. Rewrite Strategy: Example of printing table of an integer number in Python.

```
num = int(input("Display multiplication table of? "))
for i in range(1, 11):
    print(num, 'x', i, '=', num*i)
```

Output:

Display multiplication table of? 5

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50

2. Rework strategy:

The rework strategy applies all the three principles.

Let the goal of a reengineering project is to replace the unstructured control flow constructs, namely GOTOs, with more commonly used structured constructs, say, a “for” loop.

Rework strategy ka matlab hai purane ya messy code ko improve karna aur usay clean, structured or easy banana.

2. Rework Strategy: Example : C Language code (to check an integer number is even or Odd?)

```
#include <stdio.h>
void checkEvenOrNot(int num) // function to check even or not
{
    if (num % 2 == 0)
        goto even;           // jump to even
    else
        goto odd;            // jump to odd
even:
    printf("%d is even", num);
    return;                  // return if even
odd:
    printf("%d is odd", num);
}

int main() {
    int num;
    printf("Enter an Integer: ");
    scanf("%d", &num);
    checkEvenOrNot(num);
    return 0;
}
```

Output:

Enter an integer: 5

5 is odd

2. Rework Strategy: Example : C Language code (to check an integer number is even or Odd?)

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);

    (num % 2 == 0) ? printf("%d is even", num) : printf("%d is odd", num);
    return 0;
}
```

Output:

Enter an integer: 5
5 is odd

Conditional / ternary operator which behaves like if/else.

Syntax: Condition ? True statement : False statement

3. Replace strategy:

The replace strategy applies two principles, namely, abstraction and refinement.

To change a certain characteristic of a system:

(i) the system is reconstructed at a higher level of abstraction by hiding the details of the characteristic; and

(ii) a suitable representation for the target system is generated at a lower level of abstraction by applying refinement.

Starting Abstraction Level	Type Change	Reengineering Strategy		
		Rewrite	Rework	Replace
Implementation Level	Re-code	Yes	Yes	Yes
	Re-design	Bad	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Design Level	Re-code	No	No	No
	Re-design	Yes	Yes	Yes
	Re-specify	Bad	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Requirement Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	Yes	Yes	Yes
	Re-think	Bad	Yes*	Yes*
Conceptual Level	Re-code	No	No	No
	Re-design	No	No	No
	Re-specify	No	No	No
	Re-think	Yes	Yes*	Yes*

Table 4.1: Reengineering process variations [5] (©[1992] IEEE).

Yes - One can produce a target system.

Yes* - Same as Yes, but the starting degree of abstraction is lower than the uppermost degree of abstraction within the conceptual abstraction level.

No - One cannot start at abstraction level *A*, make *B* type of changes by using strategy *C*, because the starting abstraction level is higher than the abstraction level required by the particular type of change.

Bad - A target system can be created, but the likelihood of achieving a good result is low.

Re-engineering Approaches

- There are five basic approaches to re-engineer software systems.
- Each approach advocates a different path to perform re-engineering.
- Several considerations are made while selecting a particular re-engineering approach:

Several considerations are made while selecting a particular re-engineering approach:

- a) Objectives of the project;
- b) Availability of resources;
- c) The present state of the system being re-engineered; and
- d) The risks in the re-engineering project.

Re-engineering Approaches

- Big bang

- Incremental
- Partial
- Iterative
- Evolutionary

Big Bang approach mein poora system ek hi dafa replace kiya jata hai.

Faida: New system ek hi time par ready ho jata hai.

Nuksan: Zyada time, cost aur resources lagte hain, aur project risky ho jata hai.

1. Big bang Approach

- The “Big Bang” approach replaces the whole system at once.
- Once a re-engineering effort is initiated, it is continued until all the objectives of the project are achieved and the target system is constructed.
- This approach is generally used if re-engineering cannot be done in parts.
- For example, if there is a need to move to a different system architecture, then all components affected by such a move must be changed at once.
- The consequent advantage is that the system is brought into its new environment all at once.
- The disadvantage of Big Bang is that the re-engineering project becomes a monolithic task, which may not be desirable in all situations.
- In addition, the Big Bang approach consumes too much resources at once for large systems, and takes a long stretch of time before the new system is visible.

Is approach mein system ko step-by-step improve kiya jata hai, ek time par thoda-thoda change karke.

2. Incremental Approach

Faida: Errors asaani se milte hain aur progress clearly nazar aati hai (har version ke sath).

- In this approach, a system is re-engineered gradually, one step closer to the target system at a time.
- For a large system, several new interim versions are produced and released.
- Successive interim versions satisfy increasingly more project goals than their preceding versions.
- The advantages of this approach are as follows:
 - (i) locating errors becomes easier, because one can clearly identify the newly added components, and
 - (ii) It becomes easy for the customer to notice progress, because interim versions are released.

Partial approach mein sirf system ka kuch hissa re-engineer kiya jata hai, phir usay baqi purane system ke sath jor diya jata hai.

Steps:

System ko 2 parts mein divide karo

Ek part ko improve karo (Big Bang ya Incremental se)

Dono parts ko wapas combine karo

- The disadvantages of the incremental approach are as follows:
- (i) with multiple interim versions and their careful version controls, the incremental approach takes much longer to complete, and
- (ii) even if there is a need, the entire architecture of the system cannot be changed.

3. Partial Approach

- In this approach, only a part of the system is re-engineered and then it is integrated with the non-engineered portion of the system.
- One must decide whether to use a “Big Bang” approach or an “Incremental” approach for the portion to be re-engineered.

The following three steps are followed in the partial approach:

- In the first step, the existing system is partitioned into two parts: one part is identified to be re-engineered and the remaining part not to be re-engineered.
 - In the second step, re-engineering work is performed using either the “Big Bang” or the “Incremental” approach.
 - In the third step, the two parts, namely, the not to be re-engineered part and the re-engineered part of the system, are integrated to make up the new system.

3. Partial Approach - Pros & Cons..

- The partial approach has the **advantage** of reducing the scope of re-engineering task (less time and cost).
- A **disadvantage** of the partial approach is that modifications are not performed to the interface between the portion modified and the portion not modified.

4. Iterative Approach

- The re-engineering process is applied on the source code of a few procedures at a time, with each re-engineering operation lasting for a short time.
- This process is repeatedly executed on different components in different stages.

During the execution of the process, it is ensured that the four types of components can co-exist:

- a) Old components not re-engineered,
- b) Components currently being re-engineered,
- c) Components already re-engineered, and

Iterative approach mein system ko chhote-chhote parts mein bar bar improve kiya jata hai, step-by-step.

Faida: System chalta rehta hai aur users ko aadat rehti hai.

Nuksan: Old aur new components ko sath manage karna mushkil hota hai.

d) New components added to the system.

There are two advantages of the iterative re-engineering process:

- a) It guarantees the continued operation of the system during the execution of the re-engineering process; and
- b) The maintainers' and the users' familiarities with the system are preserved.

The disadvantage of this approach is the need to keep track of the four types of components during the re-engineering process.

In addition, both the old and the newly re-engineered components need to be maintained.

5. Evolutionary Approach

In the Evolutionary approach, components of the original system are substituted with re-engineered components.

In this approach, the existing components are grouped by functions and re-engineered into new components.

Software engineers focus their re-engineering efforts on identifying functional objects irrespective of the locations of those components within the current system.

As a result, the new system is built with functionally cohesive components as needed.

There are two advantages of the "Evolutionary" approach:

- a) The resulting design is more cohesive; and
- b) The scope of individual components is reduced.

A major disadvantage:

- All the functions with much similarities must be first identified throughout the operational system.
- Next, those functions are refined as one unit in the new system.

Re-engineering Cost Factors

- The *quality* of the software to be re-engineered
- The *tool* support available for re-engineering
- The *extent* of the data conversion which is required

- The *availability* of expert staff for re-engineering

Advantages of Re-engineering

Re-engineering a software system has two key advantages over more radical approaches to system evolution:

- **Reduced risk**
- **Reduced cost**

1. Reduced Risk

Re-engineering ka faida yeh hai ke naya system banane ke muqable mein risk kam hota hai, kyunki existing system use hota hai aur errors ya problems kam hoti hain.

- There is a high risk in re-developing software that is essential for an organisation.
- Errors may be made in the system specification, there may be development problems, etc.

2. Reduced Cost

- The cost of re-engineering is significantly less than the costs of developing new software.
- Ulrich (1990) quotes an example of a commercial system where the re-implementation costs were estimated at \$50 million, which was successfully re-engineered for £12 million.
- If these figures are typical, it is about 4 times cheaper to re-engineer than to re-write.

Difficulties and Challenges

- Short life expectancy – not anticipated to last decades when first developed.
- Failure of process models and software engineering culture to treat evolution as a first class activity – future requirements ignored.

Week 8

Code Reverse Engineering

In the context of software engineering, Chikofsky and Cross defined reverse engineering as a process to:

- (i) identify the components of an operational software.
- (ii) identify the relationships among those components.
- (iii) represent the system at a higher level of abstraction or in another form.

Reverse engineering is performed to achieve two key objectives:

1. Redocumentation of artifacts
2. Design recovery

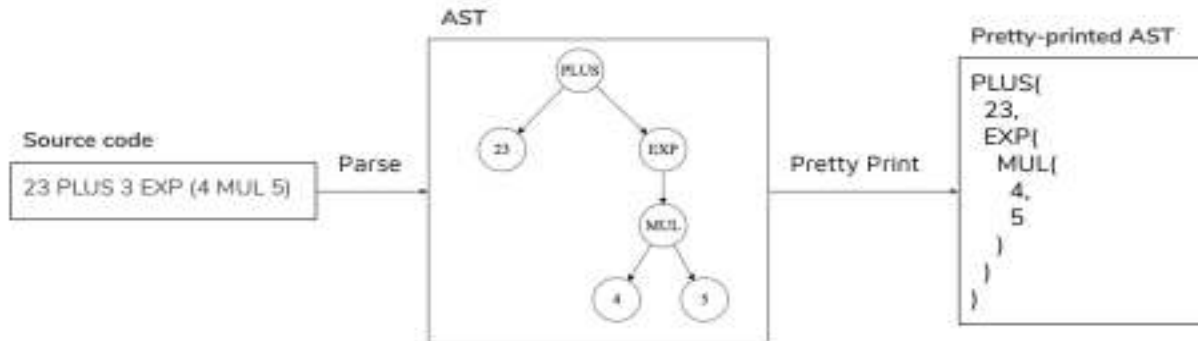
1. Redocumentation of artifacts

- It aims at revising the current description of components or generating alternative views at the same abstraction level.
- Examples of redocumentation are pretty printing and drawing CFGs.



Pretty Printing

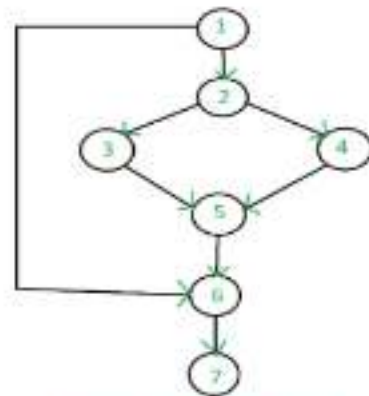
Goal: print out a human readable representation of an AST



Control Flow Graph

Example:

```
if A = 10 then  
  if B > C  
    A = B  
  else A = C  
  endif  
endif  
print A, B, C
```



Control Flow Graph

2. Design recovery

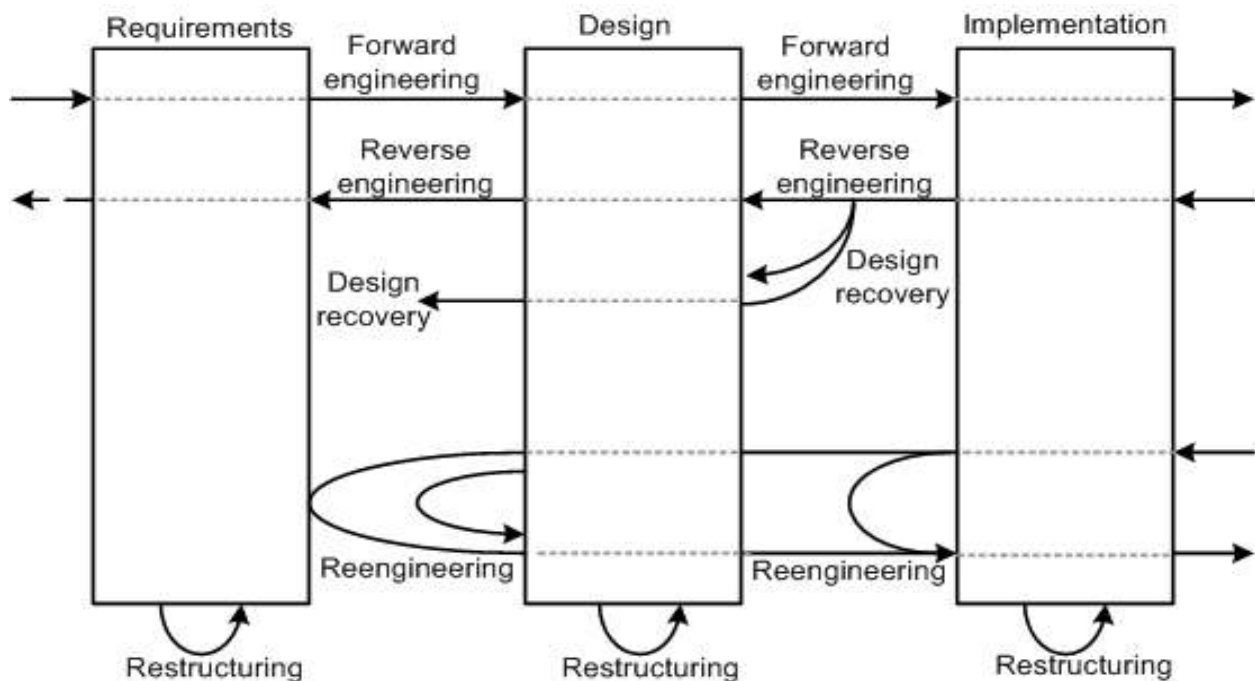
- It creates design abstractions from code, expert knowledge, and existing documentation.

Factors necessitating the need for Reverse Engineering:

- The original programmers have left the organization.
- The language of implementation has become obsolete, and the system needs to be migrated to a newer one.
- There is insufficient documentation of the system.

- The business relies on software, which many cannot understand.
- The company acquired the system as part of a larger acquisition and lacks access to all the source code.
- The system requires adaptations and/or enhancements.
- The software does not operate as expected.

Reverse Engineering & Re-Engineering



Objectives of Reverse Engineering

Six objectives of reverse engineering, as identified by Chikofsky and Cross:

- Generating alternative views
- Recovering lost information
- Synthesizing higher levels of abstractions
- Detecting side effects
- Facilitating reuse
- Coping with complexity

Applications of Reverse Engineering

Reverse engineering has been effectively applied in the following problem areas:

- Redocumenting programs
- Discovering design architectures
- Recovering design patterns
- Building traceability between code and documentation
- Deriving conceptual data models
- Detecting duplications and clones
- Cleaning up code smells
- Computing change impact
- Transforming binary code into source code
- Redesigning user interfaces
- Parallelizing largely sequential programs
- Translating a program to another language
- Migrating data
- Extracting business rules
- Wrapping legacy code
- Auditing security and vulnerability
- Extracting protocols of network applications

Reverse Engineering Process

- Typical Reverse engineering process recovers artifacts which exist at domain, functional, structural and implementation levels.
- A no. of Forward Engineering process models exists.
- Reverse Engineering process depends on the requirements of maintainer. There is still no standard reverse engineering process model.

The high level reverse engineering process is categorized into three phases:

- Extraction

- Abstraction
- Presentation

Presentation Phase

- It provides access to information produced by the extraction and abstraction processes as well as provides feedback to the user in a particular format.
- It helps the user to control and initiate the operations.
- It separates user presentation concerns from extraction and abstraction concerns.

Abstraction Phase

- Abstraction process communicates with the interface component and the extraction component to create and represent the abstract artifacts e.g. mapping or analysis scripts.

Extraction Phase

- The extraction Phase allows the user to extract the required artifacts of interest through the presentation phase.
- The user specify the patterns, actions and analysis depending on the available source code and documents.
- Documents are mapped to the source code.

User specification, Mapping, Inputs/Outputs

User specification

- User specifies the required artifacts specification for his requirement

Mapping

- Mapping is performed at all levels of abstraction to extract, abstract and present artifacts

Inputs/Outputs

- The inputs in the RE process are available source code, documentation and domain knowledge to extract, abstract and present artifacts in particular format

Reverse Engineering Tools

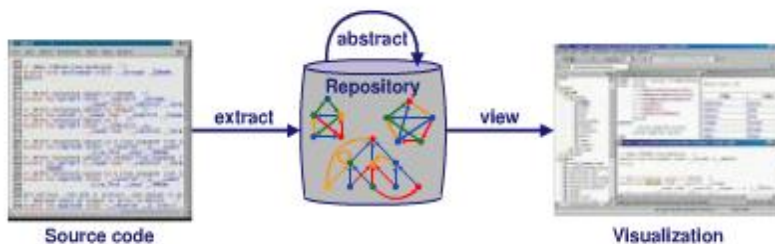
- Many reverse engineering tools concentrate on extracting the structure or architecture of a legacy system with the goal of transferring this information into the minds of the software engineers trying to maintain or reuse it.
- Traditional architecture of reverse engineering environment was made of extraction, repository and tools.
- Most tools are developed based on parse based and lexical based techniques.
- Tool architecture must provide functionality to extract, abstract and present the software system artifacts to support extraction, abstraction and presentation of the recovery of desired artifacts.

Reverse Engineering Tools Attributes

- RE tools are developed with number of features to support different methodologies according to customers' requirements.
- RE tools must have following attributes:
 - Flexibility
 - Extensibility
 - Scalability

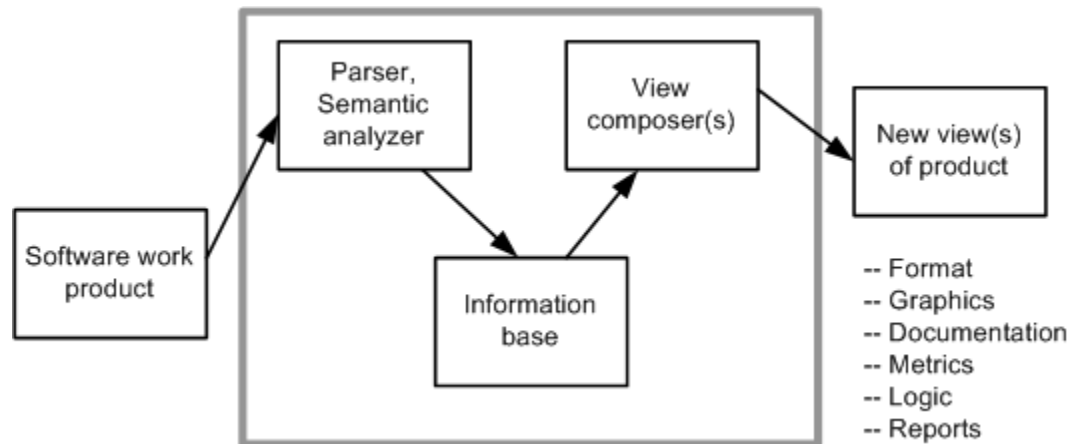
Reverse Engineering Tools

- A number of tools available.
- The reference architecture of RE tool is given as:



The basic structure of reverse engineering tools is as follows:

- The software system to be reverse engineered is analyzed.
- The results of the analysis are stored in an information base.
- View composers use the information base to produce alternative views of the system.



Enterprise Architect- a very useful tool

- UML based modeling tool with support of forward and reverse engineering
- Supports 10 plus programming languages
- Continuous improvement and progression
- Cost effective solution
- Over 300,000 licensed users
- Supports all types of UML diagrams
- www.sparxsystems.com

CASE Tools

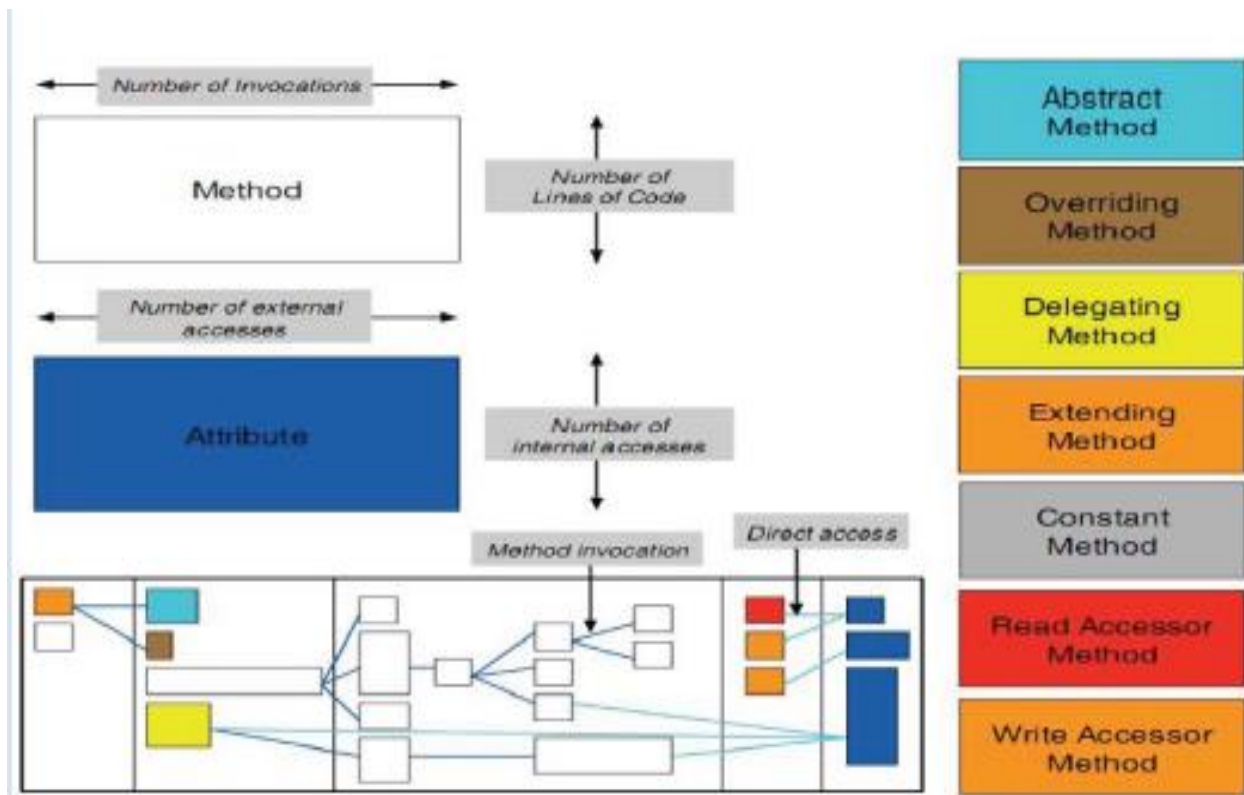
- Most CASE tools can reverse engineer class diagrams, but there is lack of tool support for extracting all type of diagrams. Popular CASE tools like Rational XDE, Borland Together, Eclipse GMT, and Fujaba support round-trip engineering.
- Borland Together is one of the leader in commercial round-trip engineering tools.
- Eclipse is an open source framework, which can be extended to be a complete integrated developing environment by external plugins.

- Several model transformation tools are implemented as Eclipse plugin, for example UML modeling tools including simple round-trip engineering support. Borland Together is available as an Eclipse plugin.
- Eclipse Generative Modeling Tools (GMT) is a set of tools in the area of model driven development.

Comparison of CASE Tools

Case Tools	Languages	Transformation	Modeling
<i>Borland Together</i>	Java, C++, VB, C#	source code - UML Class, database model-physical DB synchronization, source code to UML sequence, UML Class to any model with QVT	UML Class, database
<i>Rational XDE</i>	many, including Java, Delphi, VB, C#	source code - UML Class, database model-physical DB synchronization, source code to UML sequence	UML Class, database
<i>Eclipse GMT</i>	language independent	model-based transformations	models, metamodels
<i>MetaEdit+</i>	language independent	general model to source code, API for user written reverse engineering program	any domain-specific model
<i>FUJABA</i>	Java	Story-Diagrams + UML Class - executable source code round-trip	Story-Diagram, UML Class

Class Blueprint form



Week 9

What is Refactoring?

Refactoring is the process of **changing** a software system such that:

- The external behavior of the system does not change.
- The internal structure of the system is improved.

This is sometimes called “Improving the design after it has been written”.

Goals of Refactoring

Refactoring involves editing and cleaning up the design and previously written software code without changing the function of the code at all.

- The goal of refactoring is **not** to add new functionality.
- The goal of refactoring is to make code easier to maintain in the future.

Clean Code

Clean code is obvious for other programmers

Clean code doesn't contain duplication

Clean code contains a minimal number of classes and other moving parts

Clean code passes all tests

Clean code is easier and cheaper to maintain

Unclean Code

At what point does clean code become unclean?

- Requirements change over time, making it hard to update your code (leading to less optimal designs)
- Time and money cause you to take shortcuts

What is “Technical debt”?

Technical debt is the coding you must do tomorrow because you took a shortcut in order to deliver the software today.

"Is refactoring just cleaning up code?"

Refactoring goes further because it provides a technique for cleaning up code in a more efficient and controlled manner.

Why is Refactoring Useful?

The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time,

as a program's requirements change, the design may **need to change**

Refactoring provides techniques for evolving the design in small incremental steps.

Refactoring Benefits

- Refactoring improves the design of software.
- Without refactoring, a design will "decay", as people make changes to a software system.
- Refactoring makes software easier to understand because structure is improved, duplicated code is eliminated, etc.
- Refactoring helps you find bugs.
- Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs.
- Refactoring helps you program faster because a good design enables progress.

A Simple Example

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send ();  
}  
else {  
    total = price * 0.98;  
    send ();  
}
```

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send ();  
}  
else {  
    total = price * 0.98;  
    send ();  
}
```



```
if (isSpecialDeal())  
    total = price * 0.95;  
else  
    total = price * 0.98;  
send ();
```

A (very) Simple Example

Much better now

```
if (isSpecialDeal())  
    total = price * 0.95;  
else  
    total = price * 0.98;  
send ();
```



```
if (isSpecialDeal())  
    factor = 0.95;  
else  
    factor = 0.98;  
  
total = price * factor;  
send ();
```

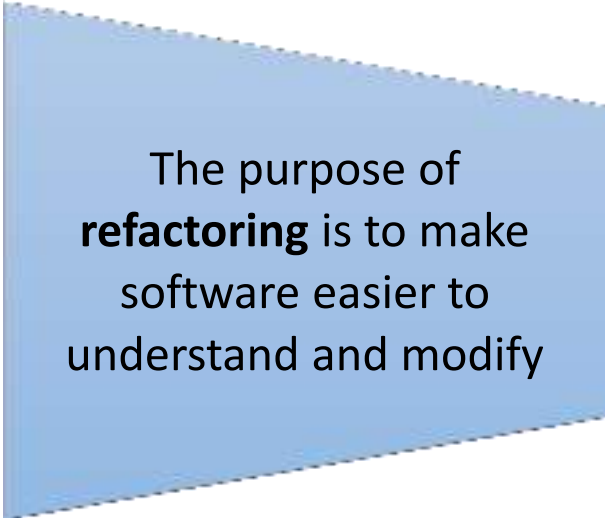
Refactoring versus Rewriting

Code has to work mostly correctly before you refactor,

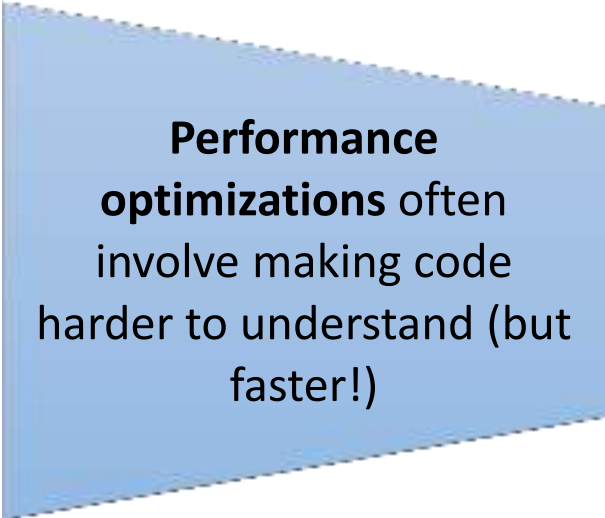
If the current code just does not work, do not refactor

Rewrite instead!

Optimization versus Refactoring

A blue trapezoidal box with a white border, containing text about refactoring.

The purpose of **refactoring** is to make software easier to understand and modify

A blue trapezoidal box with a white border, containing text about performance optimizations.

Performance optimizations often involve making code harder to understand (but faster!)

Week 10

How do you make refactoring safe?

(1) Use Refactoring “Patterns”

Fowler’s book assigns “names” to refactorings in the same way that the GoF’s book assigned names to patterns

(2) Test Constantly

This ties into the extreme programming paradigm, you write tests before you write code, after you refactor code, you run the tests and make sure they all still pass

If a test fails, the refactoring broke something, but you know about it right away and can fix the problem before you move on

TWO HATS

Kent's metaphor of the two hats. His notion is that when you're programming, you can operate in one of two modes: **refactoring** and **adding function**.

- When you're wearing the refactoring hat, every change you make preserves observable behavior, keeps the tests green, and allows you to make many small changes without going near a debugger.
- When you add function, however, things are more open ended as you will add tests and break existing tests. The adding function hat is more stressful and riskier, so it's nice to wear the refactoring hat as much as possible.

When Should You Refactor?

The Rule of Three

“Three strikes and you refactor”.

(i) **Refactor when you add function**

The most common time to refactor is when you want to add a new feature

- Refactoring helps in understanding the code that needs modification
- The other driver of refactoring here is a design that hinders adding a feature easily

PREPARATORY REFACTORING

It’s like I want to go 100 miles East but instead of just traipsing through the woods, I’m going to drive 20 miles north to the highway and then I’m going to go 100 miles East at three times the speed I could have if I just went straight there. When people are pushing you to just go straight

Refactoring Hat

Improve or clean the code.
Do not change how the program works.
Tests keep passing.
Safer and less risky.

Adding Function Hat

Add new features to the program.
Existing tests may fail, so new tests are needed.
More difficult and risky than refactoring.

there, sometimes you need to say, “Wait, I need to check the map and find the quickest route.” The preparatory refactoring does that for me.

(ii) *Refactor when you need to fix a bug*

- In fixing bugs much of the use of refactoring comes from making code more understandable
- if you do get a bug report, it's a sign you need refactoring
 - the code was not clear enough for you to see there was a bug

(iii) *Refactor as you do a code review*

- Some organizations do regular code reviews
- Refactoring helps in reviewing someone else's code
- Refactoring helps the code review have more concrete results

Week 11

Problems with Refactoring

- Databases
- Changing Interfaces
- Design Changes That Are Difficult to Refactor
- When Shouldn't You Refactor?

Databases

- A problem area for refactoring
- Most business applications are tightly coupled to the database schema that supports them
 - database is difficult to change
- data migration
 - changing the database schema forces you to migrate the data
- Some OO databases provide automatic migration

Changing Interfaces

- Objects allow you to change the implementation separately from the interface
- Many of the refactorings do change an interface
- If a refactoring changes a published interface
 - retain both the old interface and the new one
- Don't publish interfaces unless you really need to
- This is an area in which we have very incomplete data

When Shouldn't You Refactor?

- There are times when the existing code is such a mess that although you could refactor it, it would be easier to start from the beginning
 - This decision is not an easy one to make
 - A clear sign of the need to rewrite is when the current code just does not work

- code has to work mostly correctly before you refactor
- The other time you should avoid refactoring is when you are close to a deadline.

Refactoring is improving the code without changing its behavior or output.

Refactoring and Design

- Refactoring has a special role as a complement to design
- Refactoring can be an alternative to upfront design
 - Extreme Programming advocates this approach
- Mostly, you still do upfront design, but now you don't try to find *the solution*.
- Greater movement toward simplicity of design
 - Flexible solutions are more complex than simple ones
- Refactoring can lead to simpler designs without sacrificing flexibility

Refactoring and Performance

- A common concern with refactoring is the effect it has on the performance
- Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning

Refactoring and Performance....

Well-factored program helps with this style of optimization in two ways:

- gives you time to spend on performance tuning
- you have finer granularity for your performance analysis

Week 12

Code Smells

How do you identify code that needs to be refactored?

“Bad Smells” in Code



“if it stinks, change it”

Grandma Beck, discussing child-rearing philosophy

Code Smells

Smells aren't inherently bad on their own - they are often an **indicator of a problem** rather than the problem.

Code smells are **descriptions of signals or symptoms** that warn us of possible design problems.

It's crucial to detect these problems and remove them as soon as possible.

22 Bad smells listed in Fowler's book

Categories of Code Smells

1. Bloaters (5)
2. Object-orientation abusers (4)
3. Change preventers (3)
4. Dispensables (5)
5. Couplers (5)

Bloaters (5)

Bloaters are code, methods and classes that have increased to such enormous proportions that they are hard to work with.

- Long method
- Long class
- Primitive obsession
- Long parameter list
- Data Clumps

Object-orientation Abusers (4)

Incomplete or incorrect application of object-oriented programming principles.

- Switch statement
- Temporary field
- Refused bequest
- Alternative classes with different interfaces

Change Preventers (3)

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

- Divergent change
- Shotgun surgery
- Parallel inheritance hierarchies

Dispensables (5)

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

- Comments
- Lazy class
- Data class
- Duplicated code
- Speculative generality

Couplers (5)

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- Feature envy
- Message chain
- Middle man
- Inappropriate intimacy
- Incomplete library class

Solution to Bad Smells

How to fix a code smell ?

“A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality – simplicity, flexibility, understandability, performance”

(Kent Beck, Extreme Programming Explained, page 179)

Refactoring

- 68 Refactorings
- 4 Big Refactorings

Refactoring Catalog

Name. The name is important to building a vocabulary of refactorings.

Summary of the situation in which you need the refactoring

- **summary of what the refactoring does**

Motivation

Mechanics

Examples

Code Smell /Refactoring Example..

Duplicated Code

- Code repeated in multiple places

Refactorings

- Extract Method
- Extract Class
- Pull Up Method
- Form Template Method

Code Smell /Refactoring Examples...

Large Class

- A class with too many instance variables or too much code

Refactorings

- Extract Class
- Extract Subclass
- Extract Interface
- Replace Data Value with Object

Refactoring Examples.....

A few of the more common refactorings, include:

- Extract Method
- Replace Temp with Query
- Move Method
- Replace Conditional with Polymorphism
- Introduce Null Object

Week 13

Software Architecture (1/2)

In the context of software development, architecture refers to the overall structure and design of a software system.

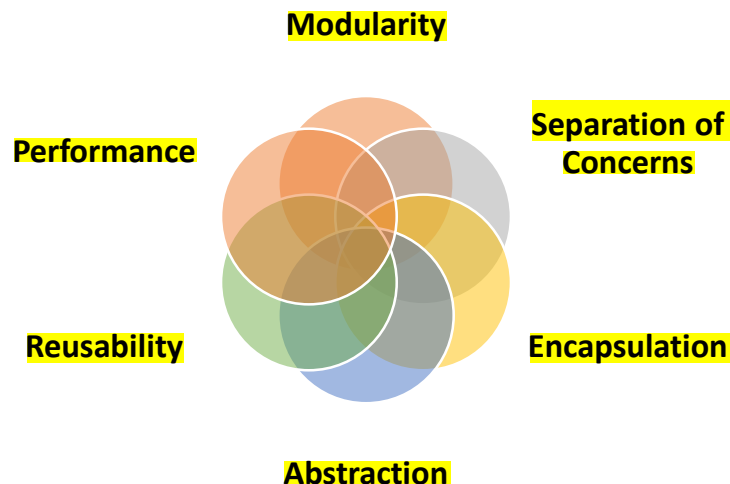
It defines the key components of the system, the relationships between these components, and how they interact with each other to achieve the desired functionality and performance.

Software Architecture (2/2)

There are many different ways to define software architecture, but one common approach is to use a set of principles and patterns that guide the design of the system.

These principles and patterns help ensure that the software is reliable, scalable, and maintainable over time, even as it evolves to meet changing user needs and technological requirements.

Some common principles and patterns used in software architecture include:



Software Re-architecting

Re-architecting is refactoring at a higher level than that of methods and classes.

You can think of it as refactoring in the large.

For example, when refactoring, you might move some classes into a separate package, whereas re-architecting might involve moving them out of the main codebase and into a separate library.

The main goals of splitting up an application, into either component modules or full-blown services, are as follows:

- Quality → modularity

- Maintainability → good design
- Autonomy → independence

Breaking up a Monolithic Application into Modules

If you've tried fine-grained refactoring on the level of methods or classes, but you find that the software still lacks clarity and structure at a higher level, you can try breaking it into modules.

Because each module has to provide an interface in order for other modules to interact with it, this partitioning process will force you to clarify the various parts of the application, as well as how these parts depend on and interact with each other.

Case study—a Log Management Application

A log management web application (developed in Java) targeted at medium to large enterprises. Some of the main features are:

Log collection—You could get log data into the system in various ways

Storage—Logs were stored in a custom-built database

Real-time alerting—Users could register alert conditions

Search—for logs of interest using conditions such as keyword, timestamp, etc.

Statistics—to display statistics using tables and graphs etc.

Reports—in a format such as HTML or PDF etc.

User interface—UI of the web application

Case study—a Log Management Application

The application was originally architected in a slightly unorthodox semi-monolithic fashion. The codebase was monolithic, but the application was deployed as two services communicating over Java RMI (Remote Method Invocation).

The source was split into three main packages: core, ui, and common. Subsets of this source code were then packaged as two services, the Core service (a command_line Java app) and the UI service (a Struts web app running on Tomcat). These services communicated using Java RMI.

The reason for splitting the code into two services was scalability. For large enterprises with a lot of log data, it was possible to deploy multiple instances of the Core service, making it possible to get more logs into the system at once. Because the Core service also contained the search engine, this made it possible to distribute searches across multiple machines in order to improve search performance. Having multiple instances of the UI didn't make much sense, so the system was designed to have one UI instance and one or more Cores.

Building and packaging was performed using a complicated Ant script, and dependency management consisted of a folder full of JAR files.

BACKGROUND: The log management application had originally been written as an inhouse utility, and from there had organically grown into a full-fledged business application, so some parts of the design were the result of historical accident rather than careful planning.

Although the application included a reporting engine for generating reports about logs, there was no single Java interface corresponding to this engine. Instead, the code for report generation was spread across a number of tightly coupled classes, and these classes were coupled to other parts of the application.

Some parts of the application were also very difficult to test, owing to excessive coupling, and test coverage was low. The development team had made efforts to improve test coverage, but had hit an impasse when faced with untestable code. A lot of code also depended on a few large God classes containing utility methods.

Despite extensive refactoring efforts, development velocity had been steadily decreasing and code quality had reached a plateau. Developers were becoming increasingly frustrated with both the source code (particularly its lack of structure and testability) and the rather antiquated tool chain.

The goals of the modularization project were as follows:

- **Introduce explicit interfaces**—Java interfaces corresponding to each of the application's main features
- **Split the source code into modules**— dependencies between modules become explicit
- **Improve dependency management**— a collection of modules with dependencies between them
- **Clean up and simplify the build script**—Splitting up the codebase into modules would result in more build scripts being necessary

Equally importantly, we made clear a few things that were outside the scope of the project:

Changes to system architecture—The separation into Core and UI services worked reasonably well, allowing the core to be scaled separately from the UI, so we decided to leave the architecture alone. However, when designing the modules, we kept in mind the possibility that the architecture might change in the future.

Changes in functionality—Trying to add any new features or changes at the same time as a major reorganization of the codebase would be asking for trouble.

Defining modules and interfaces:

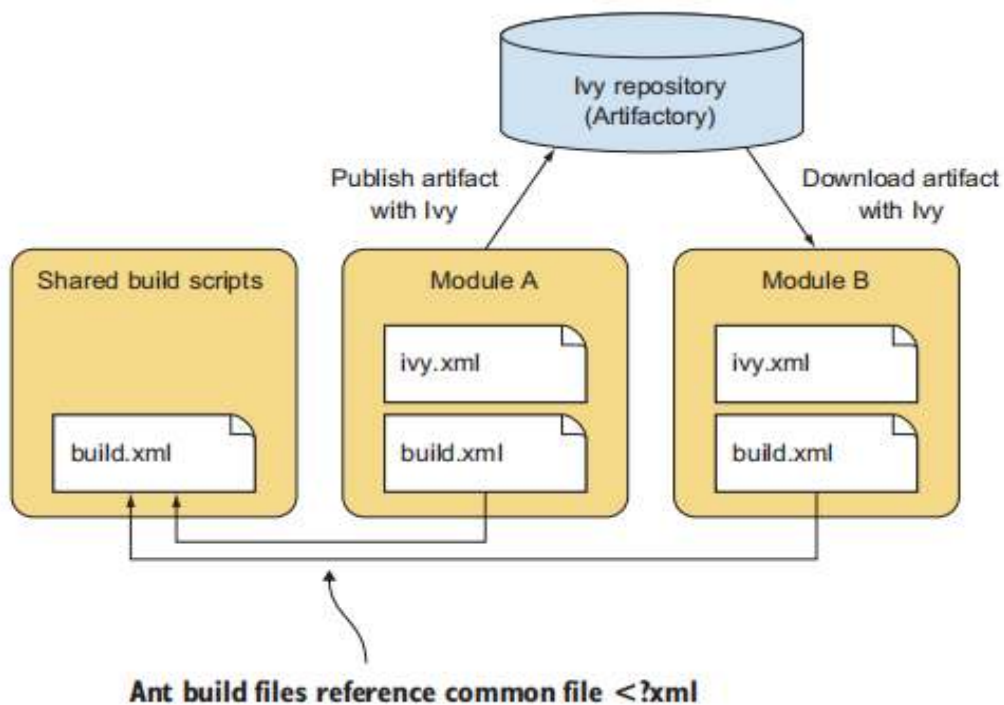
The module structure is one module for each major feature of the application.

- No component has too much responsibility, as its purpose can be explained in only one sentence
- Every component does one useful thing, so components are not too fine grained

The Core and UI services were also implemented as modules. Each service module would depend on all the modules in that service.

A number of utility classes that were heavily depended upon by large parts of the codebase. Teasing apart these dependencies, while not impossible, would have taken a huge refactoring effort. Instead we decided to simply make a **Common** module, and let all other modules depend on **Common** where necessary.

Build scripts and dependency management



Week 14

Another Case study

Distributing a Web Application into Services

We'll look at the pros and cons of re-architecting a monolithic web application into a Service Oriented Architecture (SOA).

Orinoco.com a popular e-commerce site.

The main features of the site include product listings, search, checkout, My Page, recommendations, and user authentication.

It is implemented as a monolithic Java servlet application.

To handle heavy traffic, and for redundancy purposes, there are several instances of the application running, fronted by a load balancer. The site is backed by an Oracle SQL database.

The site is maintained by a couple of dozen full-stack developers, who take care of everything from the HTML, CSS, and JavaScript in the UI to the machine-learning algorithm used for product recommendations.

There's also an operations team that keeps the site running smoothly, a quality assurance team, and a team of visual designers.

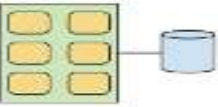
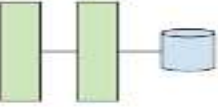
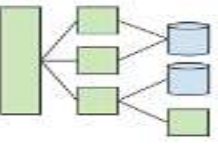
With such a large legacy codebase, and so many people committing changes to it, releases have to be performed with care. Currently the developers make a release every two weeks, but the testers are complaining that they don't have enough time to complete their manual tests, so they're considering a switch to a three-week release cycle. Development has gotten noticeably slower in the last couple of years as the site has grown more complex. And even with all the manual testing, critical bugs are sometimes introduced, so there are occasional emergency releases or rollbacks.

The product managers are desperate to develop and release a smartphone application for Orinoco.com, but that would require adding a REST API to the site, and the company can't spare the development resources to build it.

Choosing an Architecture

The Orinoco.com developers are looking to re-architect the application, mostly to improve scalability in order to handle the heavy load on Thanksgiving weekends, and to increase development velocity.

Table (next slide) compares some of the possible architectures they could choose.

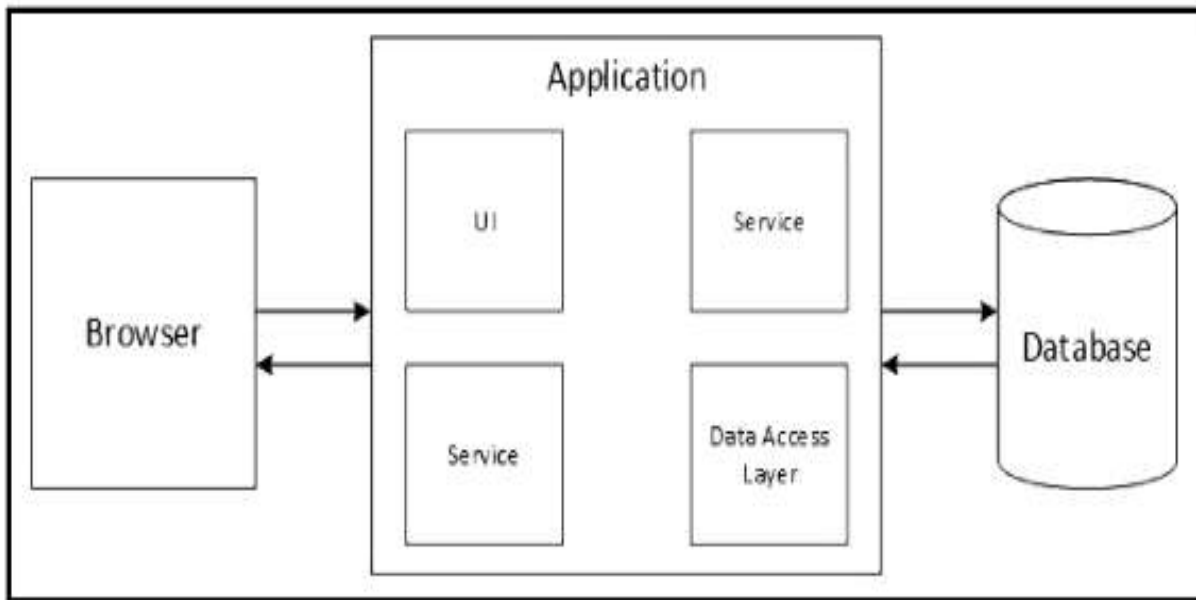
Architecture	Technical benefits	Technical challenges	Organizational benefits	Organizational challenges
Monolithic 	• Low latency • Simple development • No duplication of models/validation	• Scaling • Complexity due to large codebase • Danger of unexpected interactions	• Low intra-feature comms overhead	• Fear of failure • High inter-feature comms overhead
Front end = back end 	• Can scale FE and BE separately • Separate presentation from business logic • Can reuse BE and build more FEs	• Complexity due to network calls	• Specialization • Can iterate FE more quickly • Stepping stone toward SOA	• Communication overhead • Siloing • FE/BE devs blocking on each other
SOA 	• Fine-grained scaling • Isolation • Encapsulation	• Ops overhead • Latency • Service discovery • Tracing/debugging/logging • Hotspot services • API docs, clients • Integration testing • Data fragmentation	• Autonomy	• Dilemma over extent of autonomy • Risk of duplicated work
Microservices 	• Same as SOA, only more so!	• Same as SOA, only more so! • Risk of implicit coupling	• Even more autonomy due to bounded contexts	• Implies DevOps • Requires a platform team • Demands major shift in thinking

1. Sticking with a Monolithic Architecture

A **monolithic architecture** is one in which a software application is designed to work as a single, self-contained unit.

Applications that have this type of architecture are common.

The components within a monolithic architecture are interconnected and interdependent, resulting in tightly coupled code.



Sticking with a Monolithic Architecture

The different concerns of an application, such as user interface, business logic, authorization, logging, and database access, are not kept separate in a monolithic architecture.

These different pieces of functionality are intertwined in a monolithic application.

Advantages of Monolithic Architecture

Applications with a monolithic architecture typically have better performance.

Small applications that have this type of architecture are easier to deploy because of the simplicity of the high-level architecture.

In spite of the tightly coupled logic, monolithic applications can be easier to test and debug because they are simpler, with fewer separate components to consider.

Advantages of Monolithic Architecture

Monolithic applications are typically easy to scale because all it takes is to run multiple instances of the same application.

- However, different application components have different scaling needs and we cannot scale the components independently with a monolithic architecture.
- We are limited to adding more instances of the entire application in order to scale.

Disadvantages of Monolithic Architecture

Monolithic applications greatly inhibit the agility of the organization as it becomes difficult to make changes to the software.

Continuous deployment is difficult to achieve.

- Even if a change is made to only one component of a monolithic application, the entire software system will need to be deployed.

Organizations are required to devote more resources, such as time and testers, to deploy a new version of a monolithic application.

Disadvantages of Monolithic Architecture

Low maintainability

- Tightly coupled components make it more difficult to make changes because a change in one part of the application is more likely to affect other parts of the application
- Difficult to understand

It also takes longer for such applications to start up, lowering the productivity of the team during development.

Disadvantages of Monolithic Architecture

Monolithic applications require a commitment to a particular programming language and technology stack.

- If a migration to a different technology is needed, it requires the organization to commit to rewriting the entire application.

2. Separating Front-end and Back-end

Another common architecture for web applications is to run the front-end and back-end as separate services.

2. Separating Front-end and Back-end

Disadvantages of Monolithic Architecture (Easy & Short)

1. Hard to Change
Small changes are difficult because all parts are connected.
2. Difficult Deployment
Even a small update requires deploying the whole application.
3. More Time and Resources
Needs more developers, testers, and time for every release.
4. Low Maintainability
Tightly coupled components make maintenance difficult.
5. Hard to Understand
Large codebase is difficult to read and manage.
6. Slow Startup
The application takes longer to start.
7. Technology Lock-in
Uses one programming language/technology.
Changing technology often means rewriting the entire application.

The back-end implements the business logic of the application and usually includes a relational DB or other datastore.

The front-end should be as thin as possible and contain no business logic, and its job is to present the application to the user.

The front end might be a traditional web application, generating HTML pages on the server side, or a client-side front end running in JavaScript in the user's browser using a framework such as AngularJS or Backbone JS.

Advantages of Separating Front-end and Back-end

Improved Performance, Scalability, Maintainability

- **Separation of concerns (SoC)** in a layered designs (e.g., presentation layer, business logic layer, data access layer)
- **Modularity**

Advantages of Separating Front-end and Back-end

Another benefit of having a back-end that exposes an API is that it can be used by more than one front-end.

For example, the Orinoco.com developers want to build an API for use by smartphone applications. They could implement this as a separate front-end service, talking to the same back-end as the website does.

Disdvantages of Separating Front-end and Back-end

1. The communication and code management with the APIs increases an overhead burden of documentation on the teams. Along with this, there is no clear way to address the API changes to the front-end.
2. Efforts increases with the division of the front-end and back-end. In order to commit any changes, two synchronous commits are made rather than one.
3. The integration of front-end and back-end results in bugs, progress delays and in worst cases leads to development failure. Additionally, communication is inefficient between the teams.

3. Service-oriented Architecture

Many large-scale web applications

use an SOA in which the application's functionality is distributed across many services.

Service-oriented Architecture

Service-oriented architecture (SOA) is an architectural pattern for developing software systems by creating loosely coupled, interoperable services that work together to automate business processes.

A service is a part of a software application that performs a specific task, providing functionality to other parts of the same software application or to other software applications.

Some examples of service consumers include web applications, mobile applications, desktop applications, and other services.

Service-oriented Architecture

SOA achieves a SoC, which is a design principle that separates a software system into parts, with each part addressing a distinct concern.

A key aspect of SOA is that it decomposes application logic into smaller units that can be reused and distributed.

By decomposing a large problem into smaller, more manageable concerns satisfied by services, complexity is reduced and the quality of the software is improved.

Service-oriented Architecture

Each service in a SOA encapsulates a certain piece of logic.

This logic may be responsible for a very specific task, a business process, or a sub-process.

Services can vary in size and one service can be composed of multiple other services to accomplish its task.

4. Micro-service Architecture

Micro-service architecture emerged as the result of the shortcomings and drawbacks of the traditional service-oriented architecture (SOA) and monolithic architecture.

Micro-service Architecture

The **micro-service architecture (MSA)** pattern builds software applications using small, autonomous, independently versioned, self-contained services.

These services use well-defined interfaces and communicate with each other over standard, lightweight protocols.

Incoming requests are commonly handled by an API gateway, which serves as the entry point to the system.

It is an HTTP server that takes requests from clients and routes them to the appropriate micro-service through its routing configuration.

Micro-service Architecture

A micro-service should be a **black box** to the consumers of the service, hiding its implementation and complexity.

Each micro-service focuses on doing one thing well and they can work together with other micro-services in order to accomplish tasks that are more complex.

Micro-service Architecture

A micro-service architecture is particularly well-suited for large and/or complex software systems.

In contrast with the monolithic architecture, applications built using a micro-service architecture handle complexity by splitting the application into smaller services that are easier to manage.

Characteristics of Micro-service Architecture

- Small, focused services
- Well-defined service interfaces
- Autonomous and independently deployable services
- Independent data storage
- Communicating with lightweight protocols
- Better fault isolation

Advantages of Micro-service Architecture

One of the many advantages of using a micro-service architecture is that it affords you the option of using multiple programming languages, runtimes, frameworks, and data storage technologies.

Week 15

Do we need an Architecture?

In simple endeavors, for example construction of a mud hut, a formal architecture blueprint may not be needed to ensure successful construction and subsequent maintenance.

Context and Need for Architecture

Greater the **complexity** of the endeavor, greater is the necessity of a guiding blueprint.

Blueprint is also essential to undertake further development or make modifications.

For significantly large endeavors, not only is a blueprint essential, a phased plan/roadmap is a must.

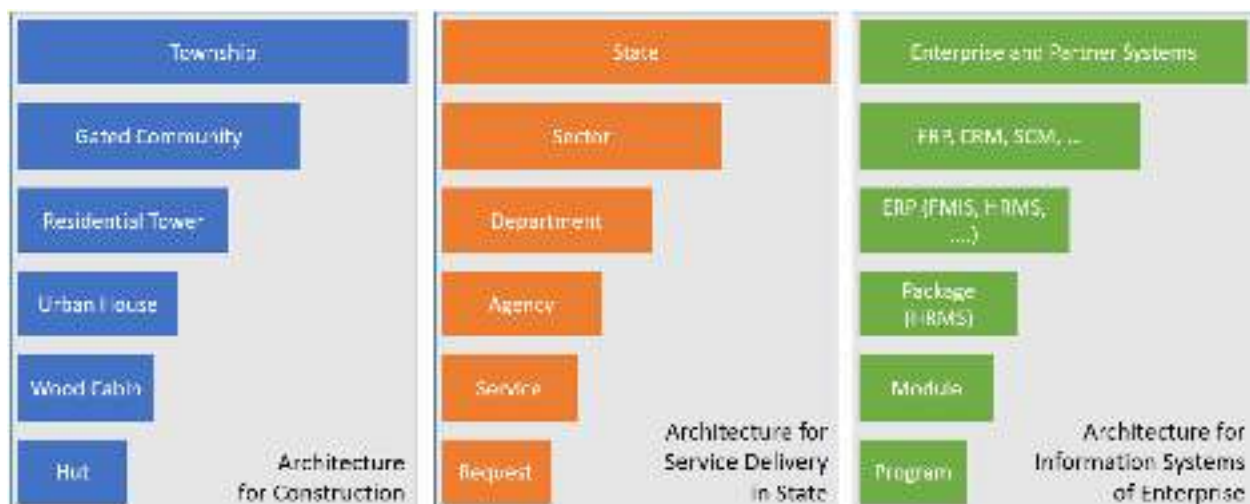
"You can use an eraser on the drafting table or a sledgehammer on the construction site."

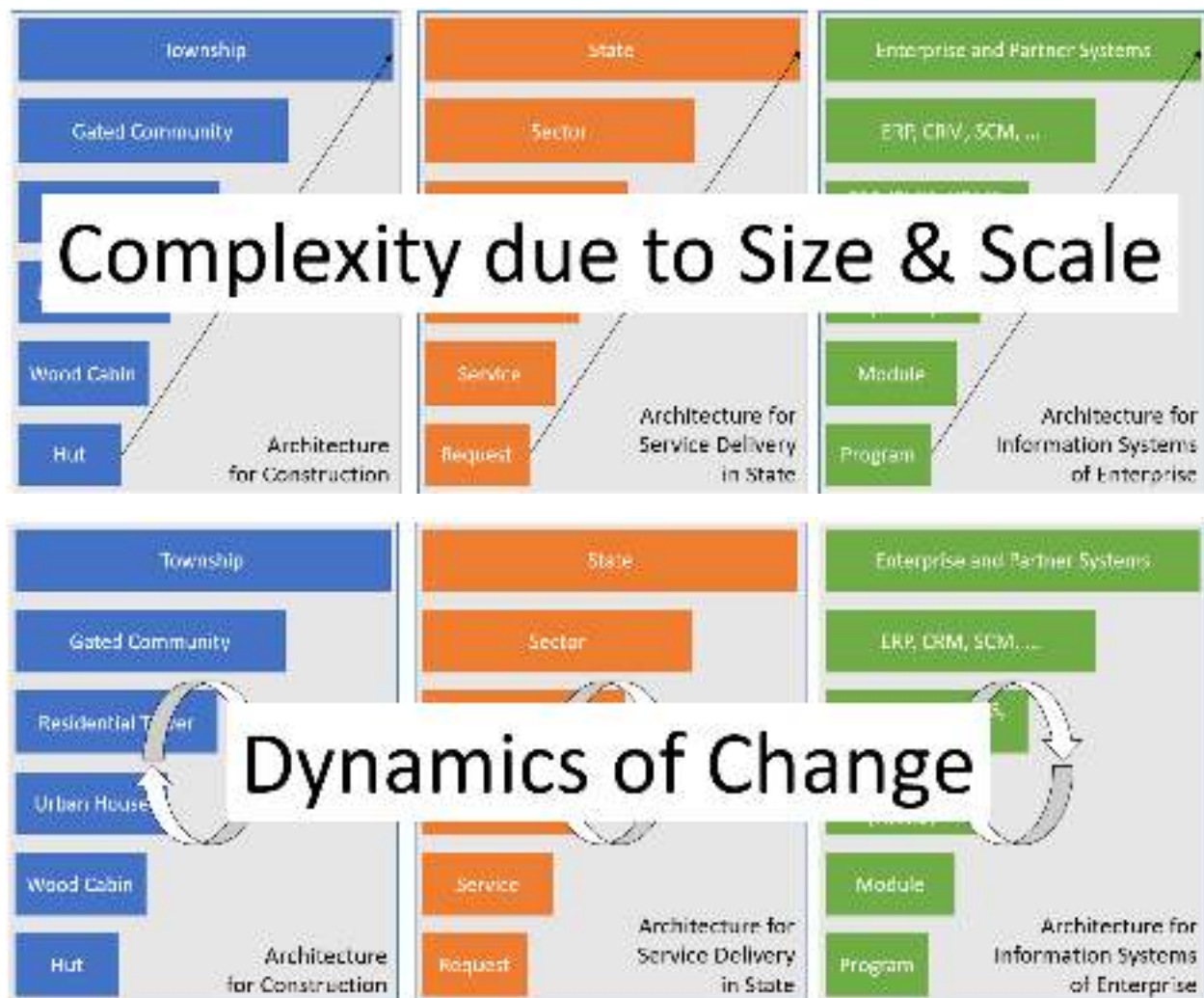
--Frank Lloyd Wright

Building large, complex, enterprise-wide **Information Systems** without an enterprise architecture is like trying to build a city without a city plan.

Can you build a city without a city plan? Probably **Yes**.

Would you want to live in such a city? Probably **not**.





Do we need Enterprise Architects?

There are many enterprises that exist without the intervention of an Enterprise Architect. However, there are many situations when it is better to architect an enterprise than to leave the creation and evolution of its architecture to chance.

Every enterprise will always have an architecture, its not optional. But we do have a choice. whether we manage its evolution or not, and how well we manage it.

Value of Architectural Thinking



- Public transit, schools and parks are afterthoughts
- No common services
- Inefficient and fragmented processes
- Aesthetically displeasing
- Limited scope for growth



- Future-oriented, planned and sustainable
- Common services
- Principles and standards (fire, safety, master plan)
- Higher productivity and effectiveness
- Aesthetically pleasing

Benefits of Enterprise Architecture

Increased efficiency of Service delivery

Improved effectiveness of scheme implementation

Seamless interoperability between systems

Increased innovation enabled by a defined architecture and its governance

Increased agility through quicker technology changes in response to dynamic business changes

Higher RoI from investments in technology, through common, re-usable components

Decisions in Enterprise Architecture (1/2)

- Should we use a common set of applications across different programs / services OR develop independent applications?
- Which systems must we develop / enhance / retire?
- Which data area / elements must we standardize across the enterprise?

Decisions in Enterprise Architecture (2/2)

- What enterprise-wide interfaces should our systems provide?
- Should we standardize the technology we use?

- What kind and which current or upcoming technologies should we invest in?

Key Takeaways (1/3)

Enterprise architecture:

is to understand the enterprise on a holistic basis (business, technology and operating environment)

helps effectively direct the enterprise to deliver strategic goals by aligning business and technology operations to strategic intent

Key Takeaways (2/3)

We need Enterprise Architecture to:

Improve service delivery efficiency and scheme implementation effectiveness by improving alignment of business & technology to strategic intent

Increase agility by making faster changes in response to dynamic business

Increase RoI from technology investments through optimized landscape enabled by common & re-usable components

Key Takeaways (3/3)

EA allows for better management of increasingly complex IT systems, while ensuring that those systems address an organization's business goals and create value.

- Better services to citizens
- Faster launch of new initiatives
- Lower risk and better RoI on IT

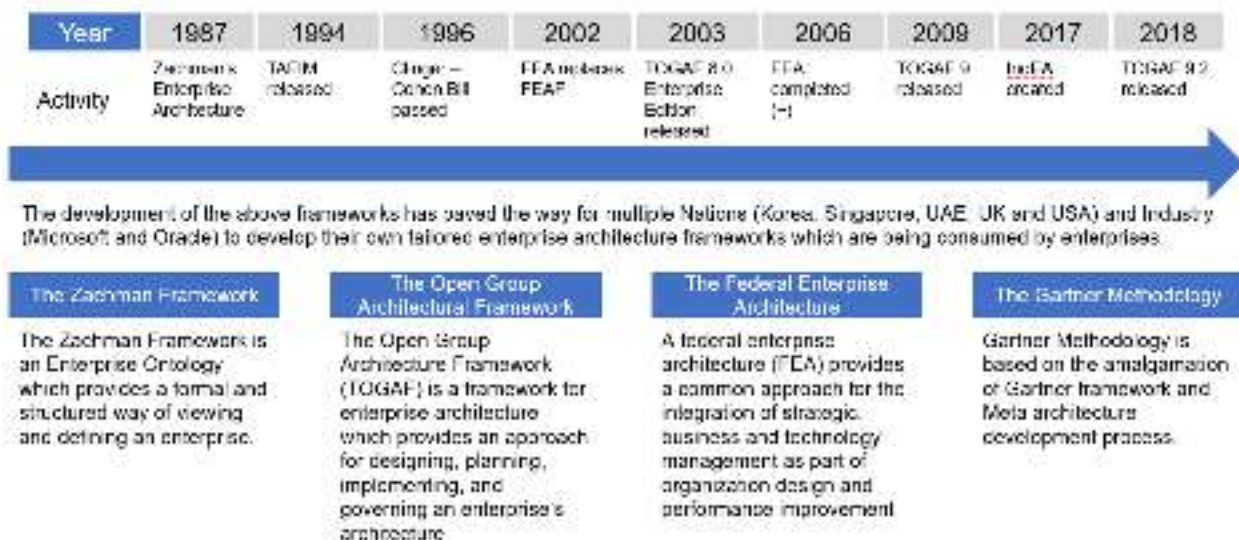
Week 16

EA Frameworks Overview

EA Frameworks typically include:

- Processes & methods
- Catalog of architecture deliverables and artifacts
- Content model and repository structure
- Reference models
- Best practices, guidelines & techniques
- Architecture capability development
- Architecture governance & compliance

EA Frameworks Evolution

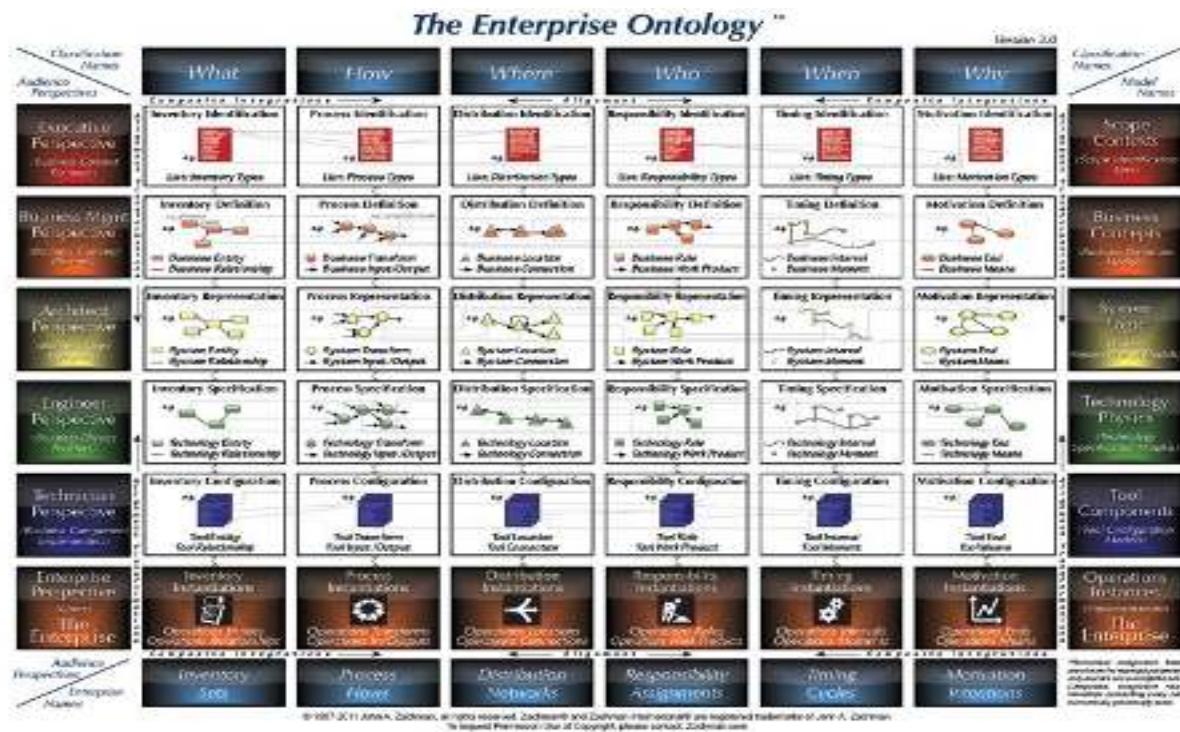


Zachman Framework (1/2)

The Zachman Framework™ is a schema - the intersection between two historical classifications that have been in use for literally thousands of years.

- **First** is the fundamentals of communication found in the primitive interrogatives: What, How, When, Who, Where, and Why. It is the integration of answers to these questions that enables the comprehensive, composite description of complex ideas.
- **Second** is derived from the transformation of an abstract idea into an instantiation – Identification, Definition, Representation, Specification, Configuration and Instantiation.

Zachman Framework (2/2)



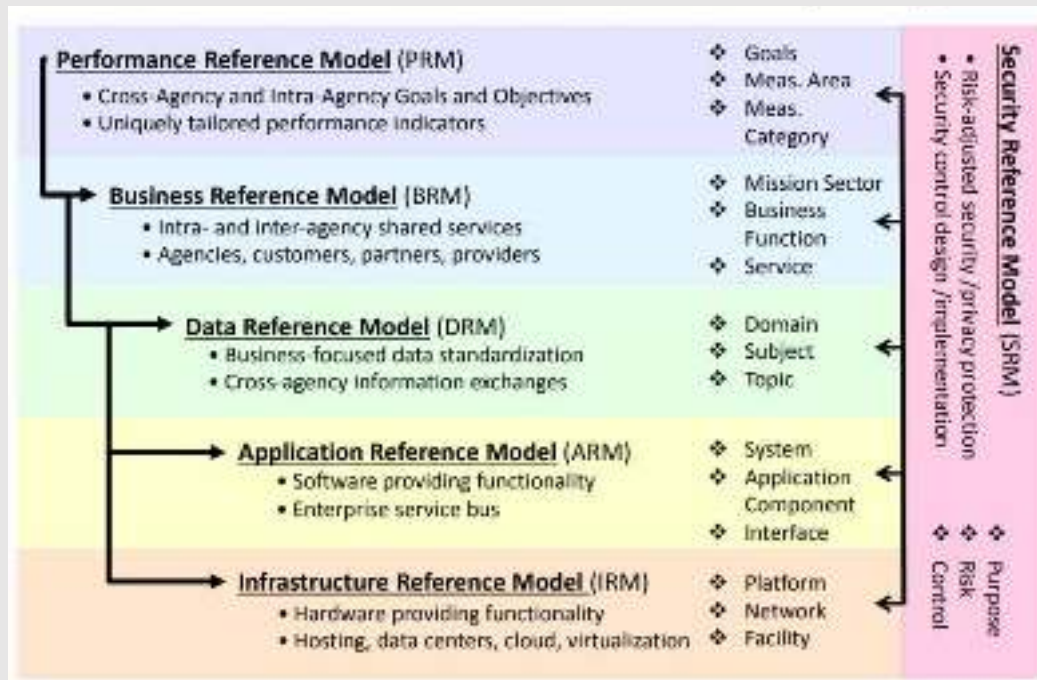
Federal Enterprise Architecture (FEA) (1/2)

Supports planning and decision-making in US Federal Govt. through documentation and information providing an abstracted view of enterprise at various levels of scope and detail.

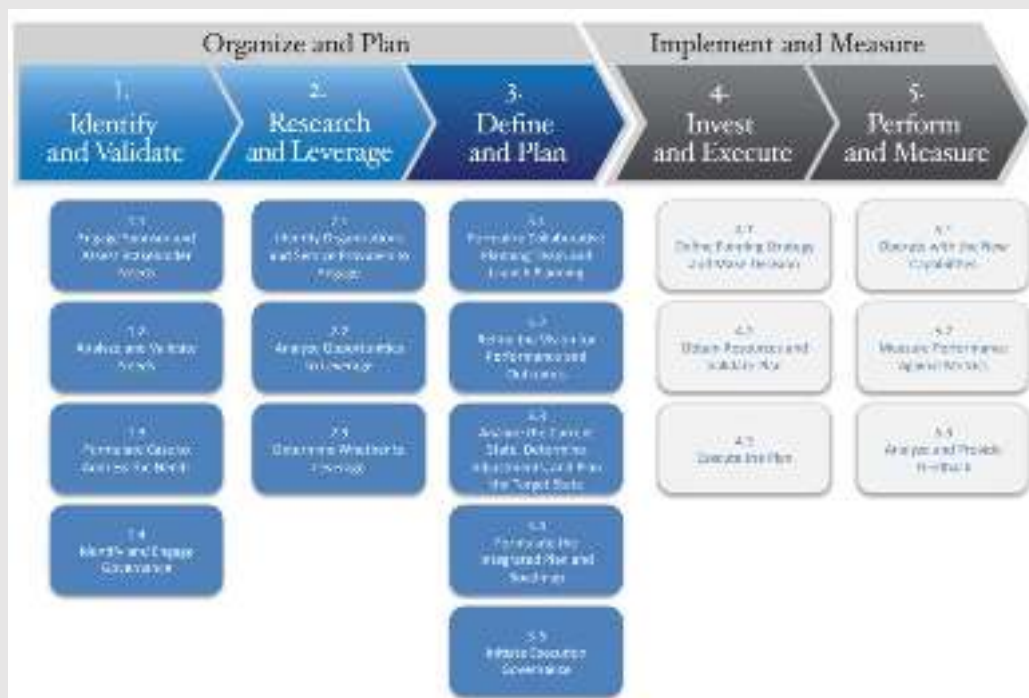
- Core is Consolidated Reference Model (CRM) – set of 5 interrelated reference models.
- Also provides Collaborative Planning Methodology – a simple, repeatable process to aid planning. It includes the master steps and detailed guidance for planners to use throughout the planning process.

Federal Enterprise Architecture (FEA) (2/2)

Consolidated Reference Model



Collaborative Planning Methodology



Consolidated Reference Model:

- PRM (Performance Reference Model): Measures how well an organization is achieving its goals.
- BRM (Business Reference Model): Defines business functions and services of an organization.
- DRM (Data Reference Model): Organizes and standardizes data for easy sharing.
- ARM (Application Reference Model): Defines software applications and their functions.
- IRM (Infrastructure Reference Model): Defines the hardware, network, and platform that run the system.
- SRM (Security Reference Model): Protects the system by managing security, privacy, and risk.

Gartner Methodology (1/2)

- Gartner methodology believes that EA is about bringing together three constituents:
 - Business owners
 - Information specialists
 - Technology implementers
- According to Gartner, EA project must be started with understanding enterprise direction on business, not with finding its current position.
- In order to obtain pure and concise information about enterprise, Gartner tries to achieve them in simple words, without concerning about recommended standard documents, or technical jargons. The result of this method is providing common understanding about enterprise situation and strategic plan.

Gartner Methodology (2/2)

The sequence of future-state EA development activities include:

Develop requirements - identification of the business-driven requirements

Develop principles - qualities that must be exhibited by an enterprise to realize its goals

Develop models - detailed drilling down of architecture content

These activities are meant to convey a logical sequence of development-based on relationships and dependencies, rather than a rigidly linear sequence of events.

The Open Group Architecture Framework (TOGAF)

- TOGAF is an Architecture Framework which has been developed by the Open Group “to provide the methods and tools for assisting in the acceptance, production, use and maintenance of an Enterprise Architecture”.
- It is based on an iterative process model called Architecture Development Method (ADM) supported by best practices and a re-usable set of existing architecture assets.

Who is The Open Group?

The Open Group is a vendor- and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability.

TOGAF EA

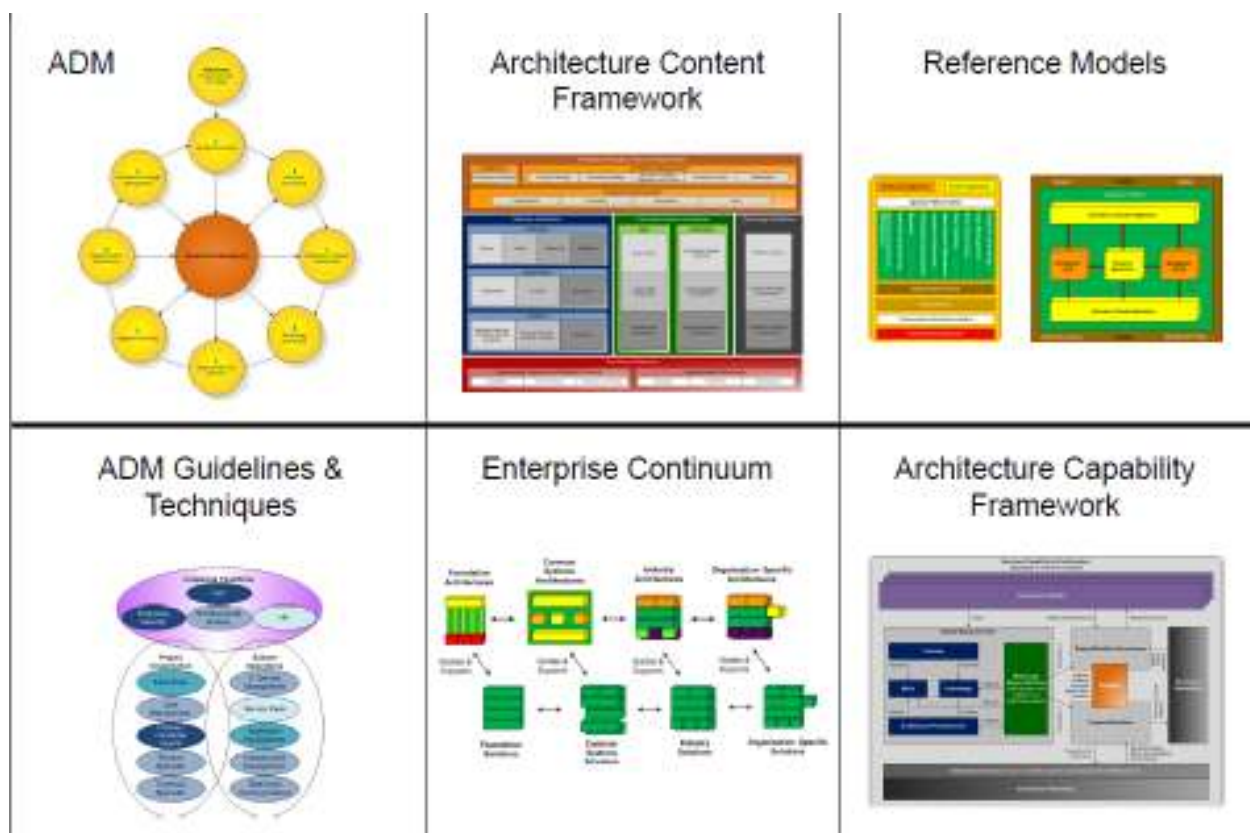
Business architecture—Describes the processes the business uses to meet its goals

Application architecture—Describes how specific applications are designed and how they interact with each other

Data architecture—Describes how the enterprise datastores are organized and accessed

Technical architecture—Describes the hardware and software infrastructure that supports applications and their interactions

TOGAF Structure



Frameworks Comparison (1/2)

- The Zachman Framework for Enterprise Architectures—Although self-described as a *framework*, is actually more accurately defined as a *taxonomy*
- The Open Group Architectural Framework (TOGAF)—Although called a *framework*, is actually more accurately defined as a *process*
- The Federal Enterprise Architecture—Can be viewed as either an *implemented enterprise architecture* or a *proscriptive methodology* for creating an enterprise architecture

- Based on Roger Sessions, ObjectWatch Inc.

Frameworks Comparison (2/2)

Criteria	Zachman	TOGAF	FEA	Gartner
Taxonomy Completeness	4	2	2	1
Process Completeness	1	4	2	3
Reference Model Guidance	1	3	4	1
Practice Guidance	1	2	2	4
Maturity Model	1	1	3	2
Business Focus	1	2	1	4
Governance Guidance	1	2	3	3
Partitioning Guidance	1	2	4	3
Prescriptive Catalog	1	2	4	2
Vendor Neutrality	2	4	3	1
Information Availability	2	4	2	1
Time to Value	1	3	1	4

Rating Scale:
1. Very Poor
2. Inadequate
3. Acceptable
4. Very Good

Source:
Comparison of
the Top Four
Enterprise
Architecture
Methodologies
by Roger
Sessions, CTO
of ObjectWatch

-By Broken404 ♥

Whatsapp: [03446811405](https://www.whatsapp.com/channel/0029VagHXNX1t90dYXOVP03h)

For Current Papers Follow This Channel:

<https://www.whatsapp.com/channel/0029VagHXNX1t90dYXOVP03h>

Best of luck for your exams...☺