

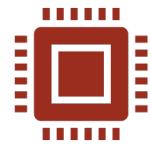
SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh



OUTLINE – WEEK 1



What is Software
design?



Design concepts



Design
considerations



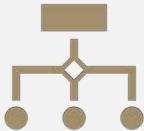
Software
Modeling



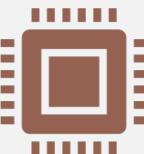
Software
development
challenges



A process of implementing software solutions to one or more set of problems.



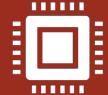
A process by which an agent creates a specification of a software artifact intended to accomplish goals using a set of primitive components and subject to constraints.



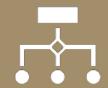
Software design can be considered as creating a solution to a problem in hand with available capabilities.

WHAT IS SOFTWARE DESIGN?

WHAT IS SOFTWARE DESIGN?



A process of converting the software requirements analysis (SRA) to list of specifications used for software development to solve problem/s.



Specifications could be as simple as

flow chart
UML diagram



The main difference between software analysis and design is that the output of a software analysis consist of smaller problems to solve.



Similar action but different output



Design is focused on solution to the problem as whole that may consist sub-problems.



Analysis consists of smaller (sub) problems to solve.



Analysis must be same to the multiple designs to the same problem.



Design may be platform-independent or platform-specific, depending on the availability of the technology used for the design.

SOFTWARE DESIGN VS. ANALYSIS

DESIGN CONCEPTS



The design concepts are a foundation from which more sophisticated methods can be applied.



A set of concepts has evolved.



Abstraction

A process or result of generalization by reducing the information content.

Retain only information which is relevant for a particular purpose.



Refinement

More specific to a certain process.

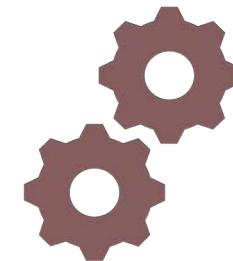
Abstraction and Refinement are complementary concepts.

DESIGN CONCEPTS



Modularity

Software architecture is divided into components called modules.

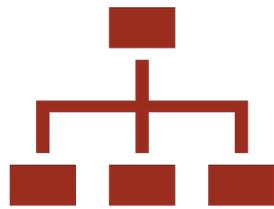


Software Architecture

Overall structure of the software.
A good architecture will yield to good quality product.

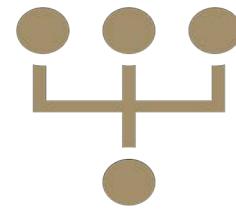
Effectiveness is measured in terms of performance, quality, schedule and cost.

DESIGN CONCEPTS



Hierarchy

A program structure that represents the organization of a program component.



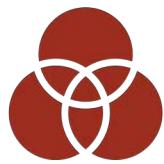
Structural Partitioning

The program structure can be divided both horizontally and vertically.

Horizontal partitions define separate branches of modular hierarchy.

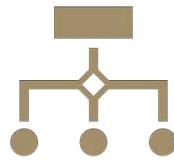
Vertical partitioning suggests that control and work should be distributed top down.

DESIGN CONCEPTS



Data Structure

It is a representation of the logical relationship among individual elements of data.



Software Procedure

It focuses on the processing of each modules individually



Information Hiding

Hide the implementation details.

DESIGN CONSIDERATIONS



There are many aspects to consider in the design of a piece of software based on the goals the software. Some of these aspects are:



Compatibility

The software is able to operate with other products.



Extensibility

New capabilities can be added to the software without major changes to the underlying architecture.



Fault-tolerance

The software is resistant to and able to recover from component failure.

DESIGN CONSIDERATIONS



Maintainability

A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.



Modularity

Smaller modules for each individual task.
Easy to test, debug, reuse, and maintain.



Reliability

The software is able to perform a required function under stated conditions for a specified period of time.

DESIGN CONSIDERATIONS



Reusability

It can be reused in other application and can be extended easily.



Robustness

The software is able to operate under stress or tolerate unpredictable or invalid input.



Security

The software is able to withstand hostile acts and influences.

DESIGN CONSIDERATIONS



Usability

User friendly and self explanatory.
Default values for the parameters.



Performance

The software performs its tasks within a user-acceptable time.



Scalability

The software adapts well to increasing data or number of users.

SOFTWARE MODELING LANGUAGE



Software models are used to represent software design.



Textual or graphical languages are used to express the software design.



For object-oriented software, an object modeling language such as UML is used to develop and express the software design.

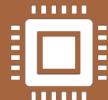


Defined by a consistent set of rules.

SOFTWARE MODELING LANGUAGE



Most commonly used software modeling language is UML.



Unified Modeling Language (UML) is a general modeling language to describe software both structurally and behaviorally.



A standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

SOFTWARE DEVELOPMENT CHALLENGES

Software is

- Used for a long time
- Updated and maintained by people who did not write it

Software requirements specification

- Initially may be incomplete
- Clarified through extensive interaction between user(s) and system analyst(s)
- Needed at the beginning of any software project
- Designers and users should both approve it

SOFTWARE DEVELOPMENT CHALLENGES

Requirements change

- Users needs and expectations change over the time
- Software use reveals limitations and flaws
- Desire for increased convenience, functionality
- Desire for increased performance

Environment change

- Hardware, OS, software packages (“software rot”)
- Need to interact with clients, parent org., etc.
- Law and regulations change
- Ways of doing business
- Style, “cool” factor, new trends

OTHER CHALLENGING FACTORS



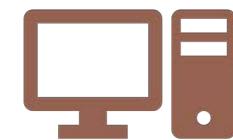
Resources

Time, budget, expertise



Organizational changes

People, goals, mergers and acquisitions



Technology constraints

Operating system, software language, framework, database, security, ever changing technology, innovations



You write code based on what you know



When was the last time you had to change the design?



What happened after you changed it?



Does your code turn into a loose cannon towards the



deadline?

RISKS IN DEVELOPMENT



Change is inevitable



No surprises



Feedback is critical



Frequent feedback is necessary



You want to know right away if you broke the code, isn't it?



Test to break it, break to test it.

EFFORTS TO MINIMIZE RISK



A good design leads to a good product.



In Engineering Construction is expensive, Design is relatively Cheap



In Software Development Construction is Cheap and design (which involves modeling and coding) is expensive



Can't we quickly test our design (since construction is cheap)?



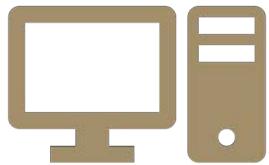
Testing is the Engineering Rigor in Software Development

MOTIVATION TO GOOD DESIGN

HOMEWORK



Review class notes.



Research software development methodologies and practices.

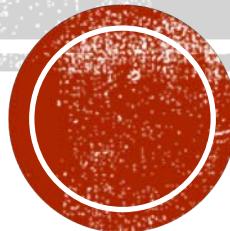


Start a discussion on Google Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh



OUTLINE



Software Development Process



Software Development Methodologies



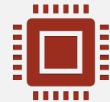
SDLC



Agile



SOFTWARE DEVELOPMENT PROCESS



A structure imposed on the development of a software product.

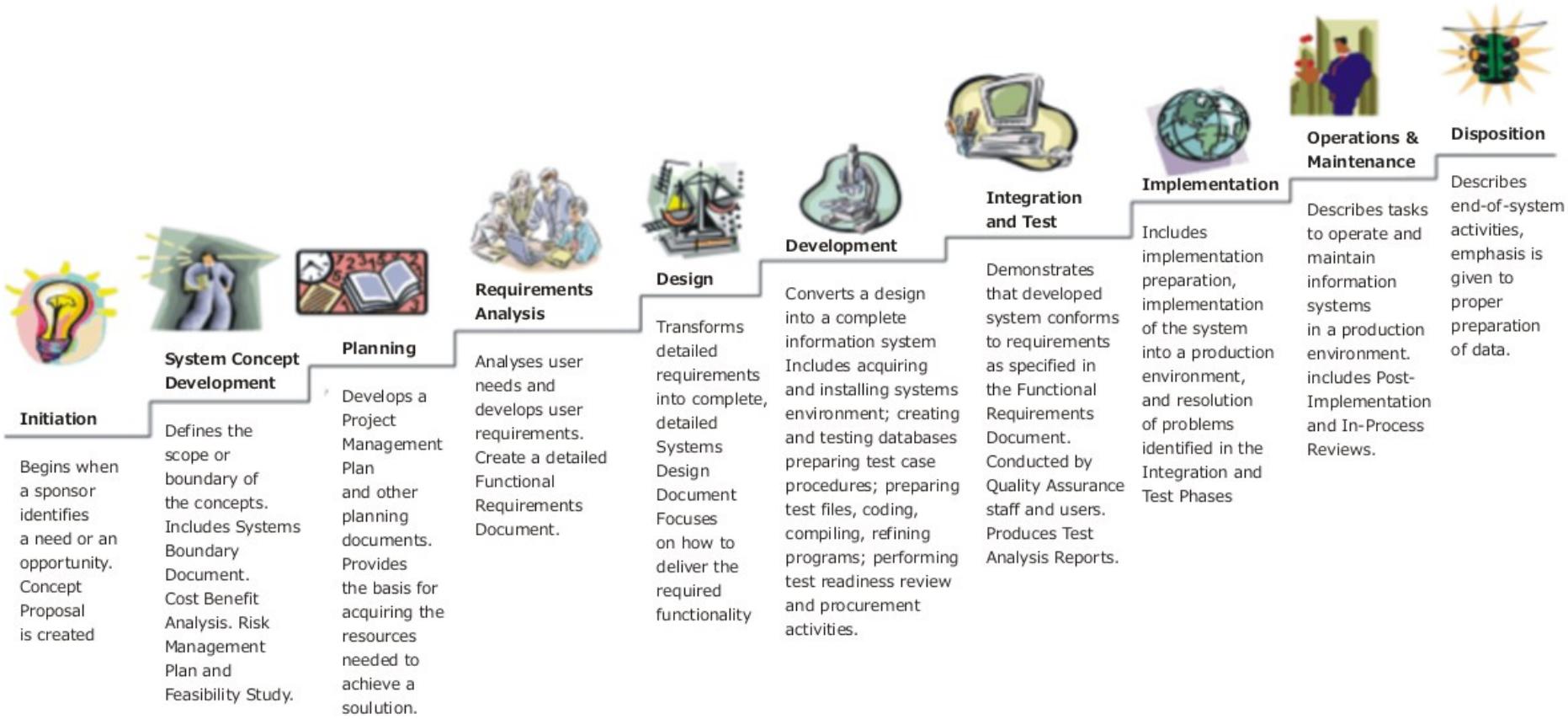


A framework that is used to structure, plan, and control the process of developing an information system.



Several software development approaches have been used since the origin of information technology.

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)



SOFTWARE DEVELOPMENT ACTIVITIES

Activity	Description
 Planning	An objective of each and every activity, where we want to discover things that belong to the project.
 Analysis & Design	Analysis of requirements and design of software is done throughout development
 Implementation	Implementation is the part of the process where software engineers actually program the code for the project.
 Testing	Software testing is the process to ensure that defects are recognized as soon as possible.
 Deployment	Deployment starts directly after the code is appropriately tested and approved for release to production environment.
 Support	Software training and support is important, as software is only effective if it is used correctly.
 Maintenance	Maintaining and enhancing software to new requirements can take substantial time and effort as missed requirements may force redesign of the software.

SOFTWARE DEVELOPMENT MODELS



Prescriptive Models

Traditional



Agile Models

Modern

PREScriptive MODELS

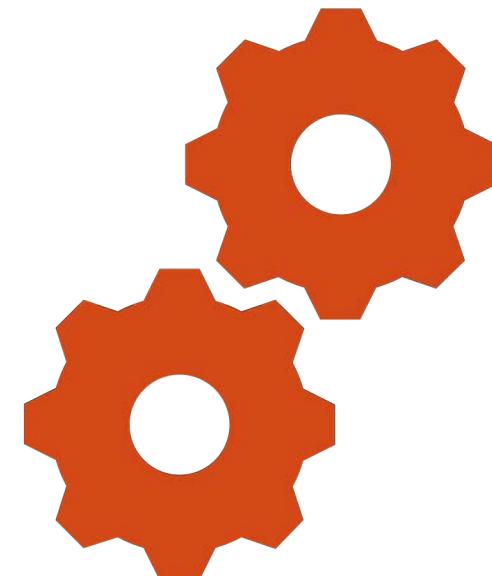
Prescriptive process models advocate an orderly approach to software engineering

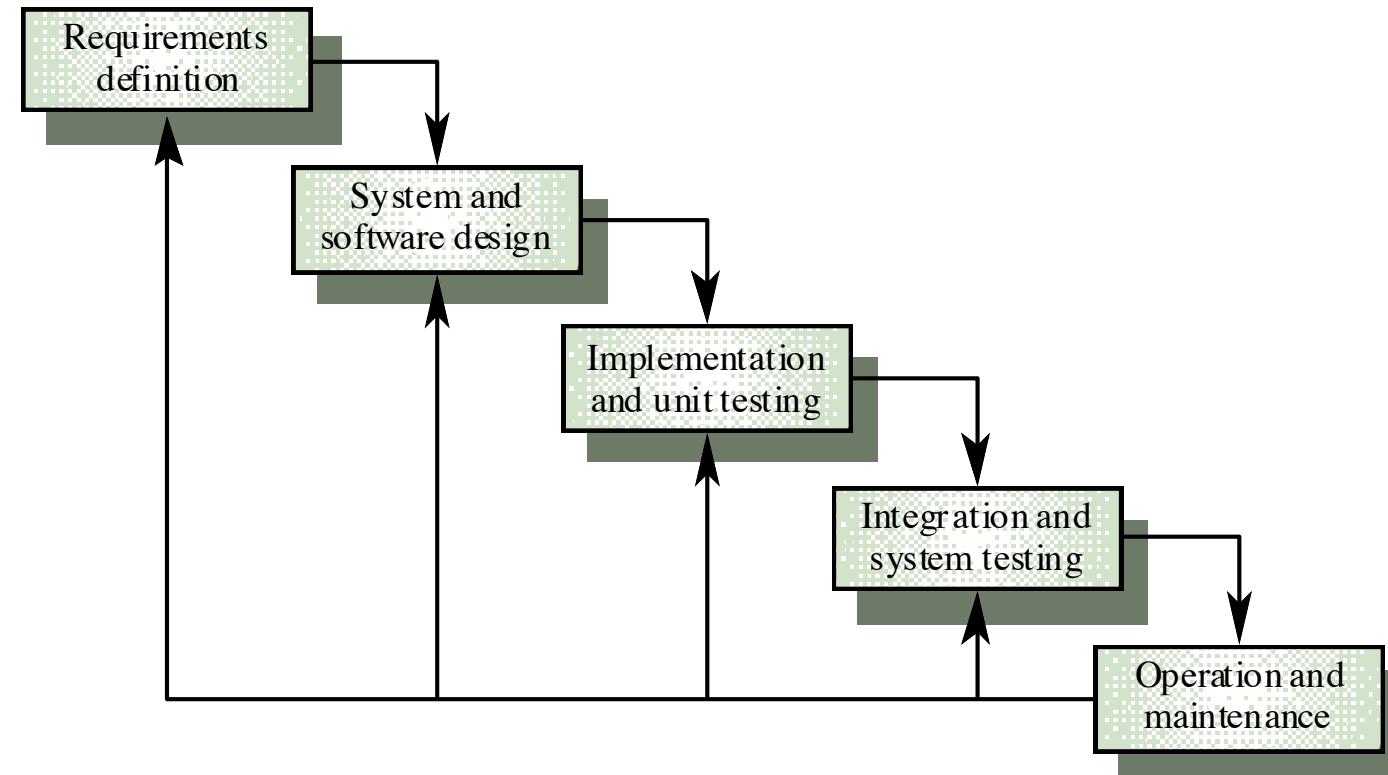
That leads to a few questions ...

- If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change?
- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

TRADITIONAL MODELS

- Waterfall
 - a linear framework
- Spiral
 - a combined linear-iterative framework
- Incremental
 - a combined linear-iterative framework or V Model
- Prototyping
 - an iterative framework
- Rapid application development (RAD)
 - an iterative framework





WATERFALL MODEL

WATERFALL MODEL

Software engineers are to follow these phases in order

- Requirements
- Software design
- Implementation
- Testing
- Deployment
- Maintenance

Each phase is dependent on previous step

Next phase starts only if previous step is finished

WATERFALL PROCESS CHARACTERISTICS

Figure out what needs to be done

Figure out how it will be done

Then do it

Verify its done right

Hand product to customer

What happens if requirements were not right?

WATERFALL MODEL ISSUES



Real projects rarely follow the sequential flow that the model proposes.



At the beginning of most projects requirements are not clear.



Requirements cannot be changed in the middle.

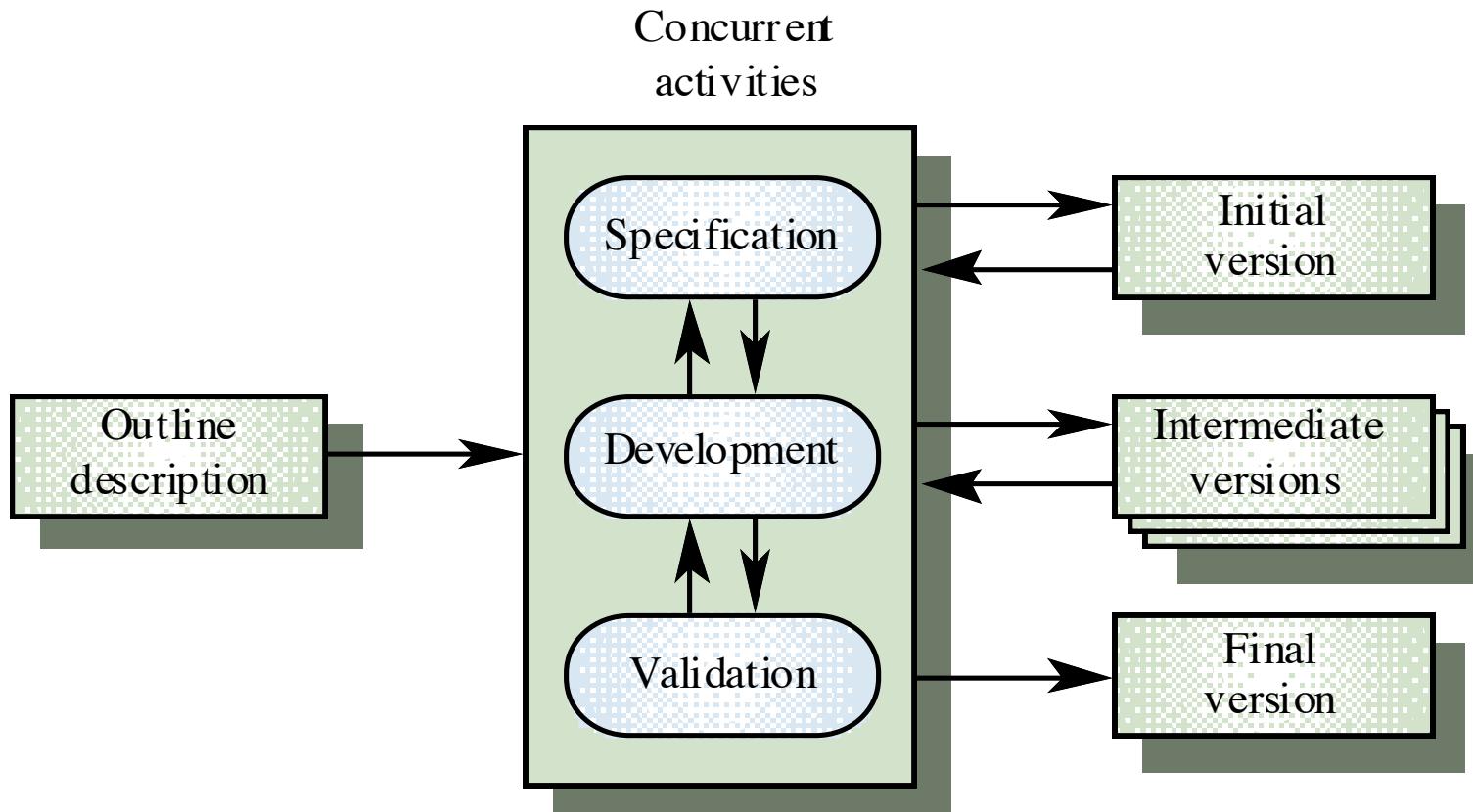


The model does not accommodate flexibility very well.



Development can take very long time and that does not yield a working version of the system until late in the process.

EVOLUTIONARY DEVELOPMENT



Modern development processes take evolution as fundamental, and try to provide ways of managing, rather than ignoring, the risk.

Requirements always evolve in the course of a project.

Specification is evolved in conjunction with the software

Not ideal for large systems.

Two (related) process models:

Incremental development

Spiral development

EVOLUTIONARY PROCESS CHARACTERISTICS

EVOLUTIONARY PROCESS CHARACTERISTICS

Modern development processes try to provide ways of managing, rather than ignoring, the risk.

Requirements always evolve in the course of a project.

Specification is evolved in conjunction with the software

Not ideal for large systems.

Two (related) process models:

- Incremental development
- Spiral development

INCREMENTAL DEVELOPMENT



Rather than delivering the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.

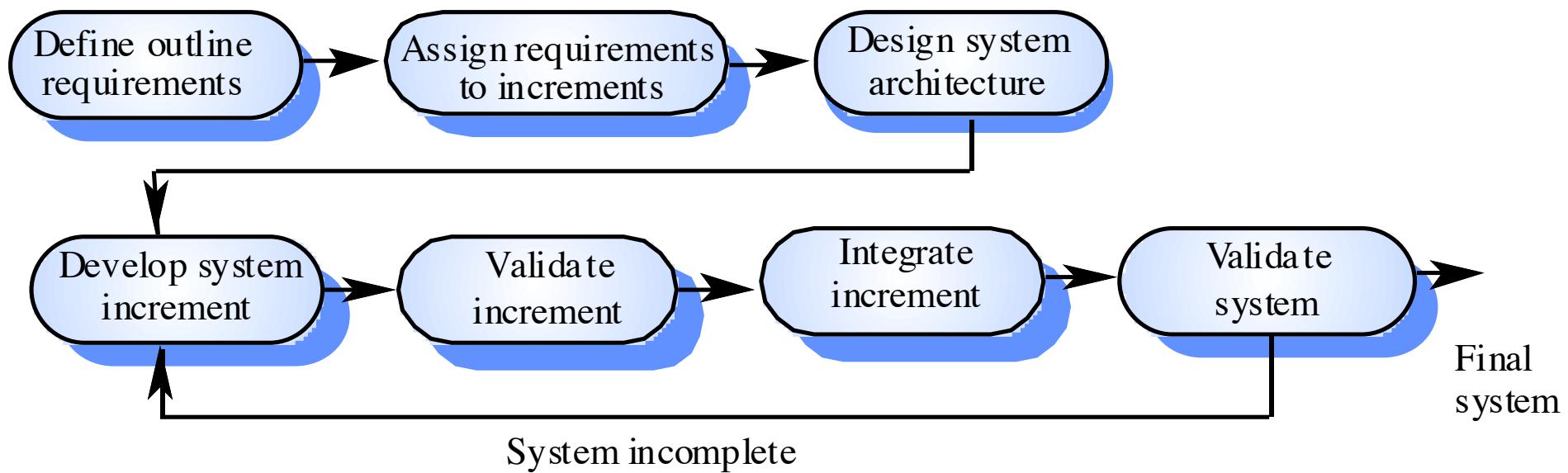


Requirements are prioritised and the highest priority requirements are included in early increments.



Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

INCREMENTAL DEVELOPMENT



INCREMENTAL DEVELOPMENT ADVANTAGES



Customer value can be delivered with each increment so system functionality is available earlier.



Early increments act as a prototype to help elicit requirements for later increments.

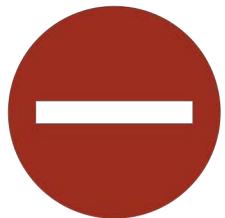


Lower risk of overall project failure.



The highest priority system services tend to receive the most testing.

INCREMENTAL DEVELOPMENT – PROBLEMS



Lack of process visibility.



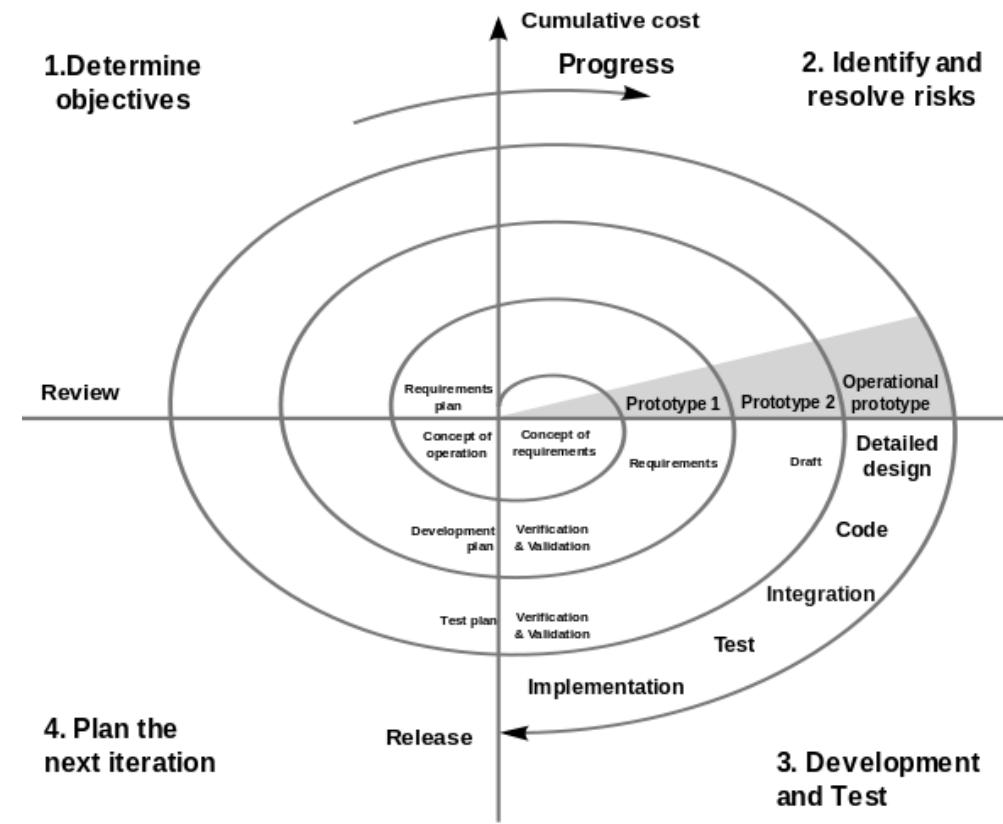
Systems are often poorly structured.



Not ideal for large systems.

SPIRAL

- The key characteristic of is risk management at regular stages in development cycle
- Combines key aspect of the waterfall model & rapid prototyping
- Good for complex systems.



SPIRAL



Process passing through some number of iterations.



More emphasis on risk analysis.



Requires to accept the analysis and act on it.



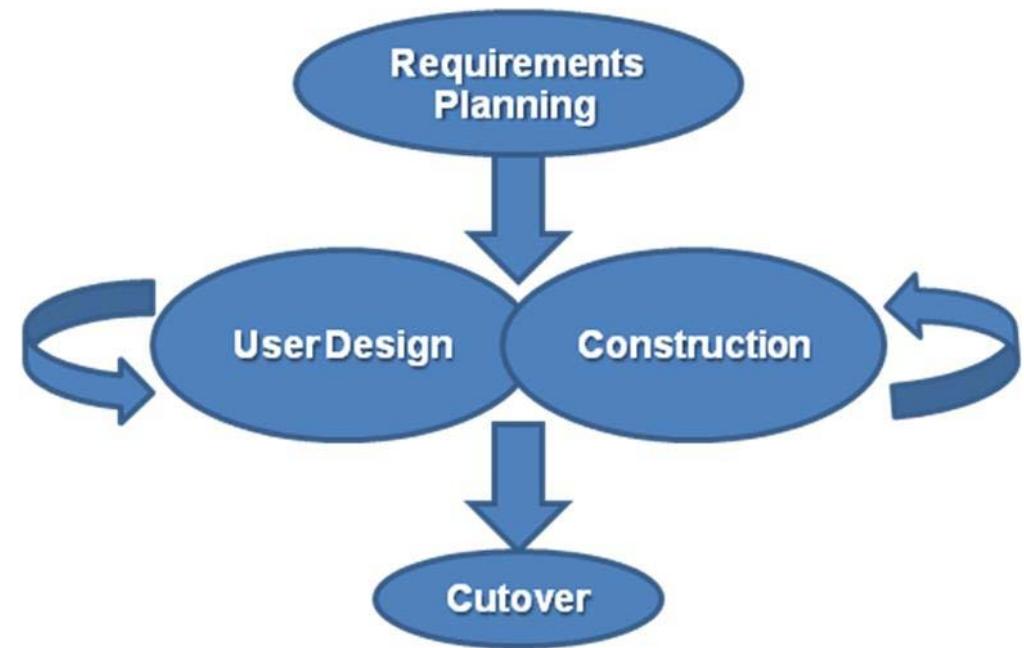
Willingness to spend more to fix the issues, which is the reason why this model is often used for large-scale internal software development.

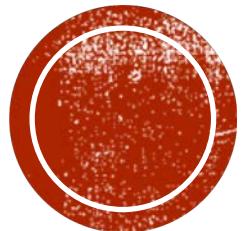


If the implementation of risk analysis will greatly affect the profits of the project, the spiral model should not be used.

RAPID APPLICATION DEVELOPMENT

- RAD requires minimal planning.
- Faster development.
- Easier to change requirements.
- Iterative & prototyping
- Starts with data models and business process modeling.
- Requirements are verified by prototyping, eventually to refine the data and process models.





AGILE DEVELOPMENT



PROJECT FAILURE – TRIGGER FOR AGILITY



ONE OF THE PRIMARY CAUSES OF
PROJECT FAILURE WAS THE
EXTENDED PERIOD OF TIME IT TOOK
TO DEVELOP A SYSTEM.



COSTS ESCALATED AND
REQUIREMENTS CHANGED.



AGILE METHODS INTEND TO
DEVELOP SYSTEMS MORE QUICKLY
WITH LIMITED TIME SPENT ON
ANALYSIS AND DESIGN.



Effective (rapid and adaptive) response to change



Effective communication among all stakeholders



Drawing the customer onto the team



Organizing a team so that it is in control of the work performed



Yielding ...

Rapid, incremental delivery of software

WHAT IS AGILITY?



Is driven by customer descriptions of what is required (scenarios)



Recognizes that plans are short-lived



Develops software iteratively with a heavy emphasis on construction activities



Delivers multiple 'software increments'



Adapts as changes occur

AN AGILE PROCESS

ACILE PROCESS



Agile methods are considered

Lightweight

People-based rather than Plan-based



Several agile methods

Extreme Programming (XP) most popular
SCRUM
TDD etc...

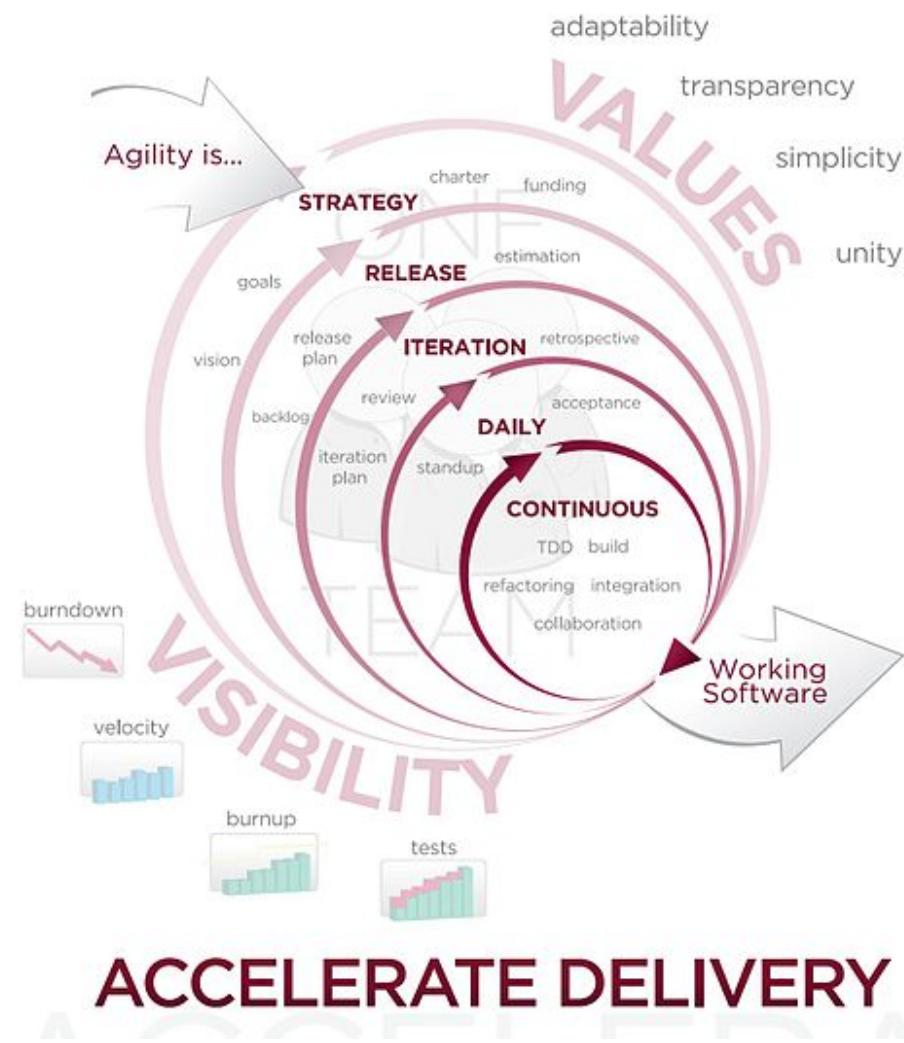


Agile Manifesto closest to a definition

Set of principles

Developed by Agile Alliance

AGILE DEVELOPMENT





Follows agile process



The phases are carried out in extremely small (or "continuous")



First write automated tests as concrete goal for development



Then coding. Complete only if all tests passed



Design and architecture emerge out of refactoring



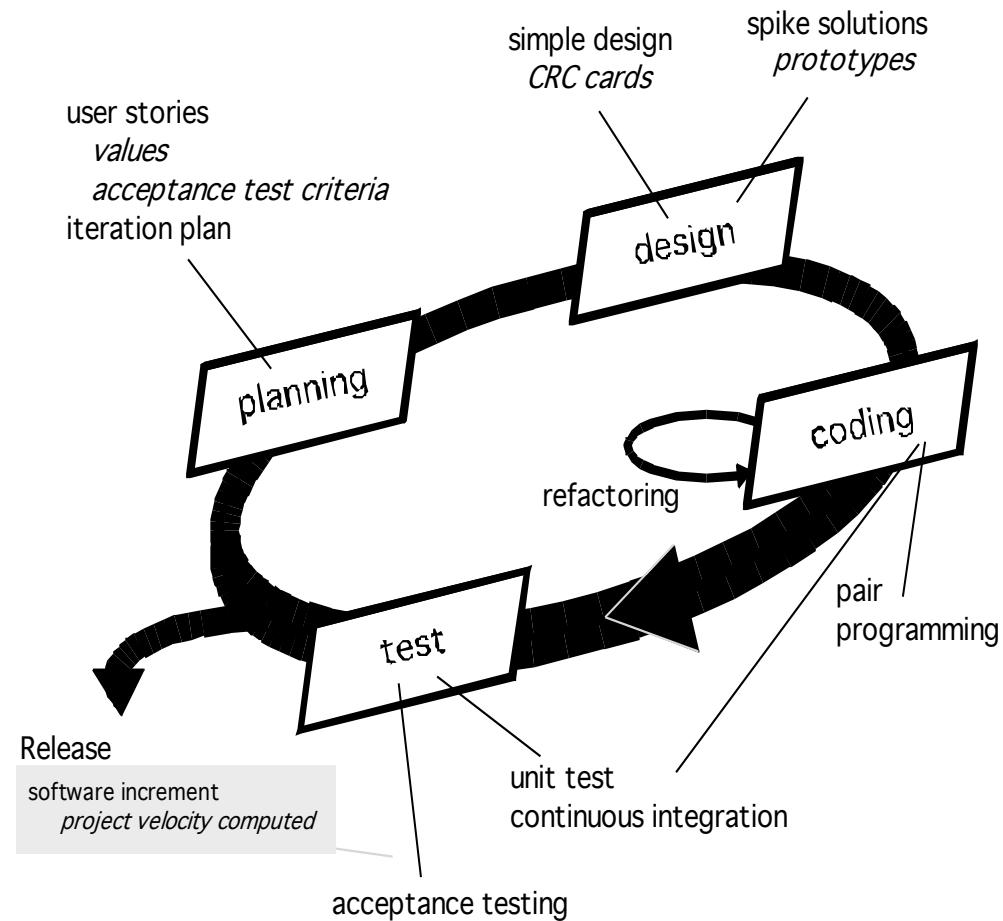
The incomplete but functional system is deployed or demonstrated



Move to next part of the system

EXTREME PROGRAMMING (XP)

EXTREME PROGRAMMING (XP)





Scrum is a framework for agile software development



Enables the creation of self-organizing teams by encouraging co-location of all team members



Testing and documentation are on-going as the product is constructed



Work occurs in “sprints” and is derived from a “backlog” of existing requirements



Meetings are very short and sometimes conducted without chairs



“demos” are delivered to the customer with the time-box allocated

SCRUM

	Scrum Team	product owner, development team, scrum master
	Sprint	Timeboxed iteration of a continuous development cycle
	Planning	Work and effort necessary to meet their sprint commitment
	Product Backlog	List of all things that needs to be done within the project
	Sprint Backlog	list of all things that needs to be done within a sprint
	Daily Meeting	15-minute meeting to provide status update
	Review	Review of the team's activities during the Sprint
	Retrospective	What went well and continue? What can be improved? Actions

SCRUM TERMS

SCRUM – PROCESS FLOW





A process that relies on the repetition of a very short development cycle



Based on test first programming concept of XP



First write an (initially failing) automated test case that defines a desired improvement or new function



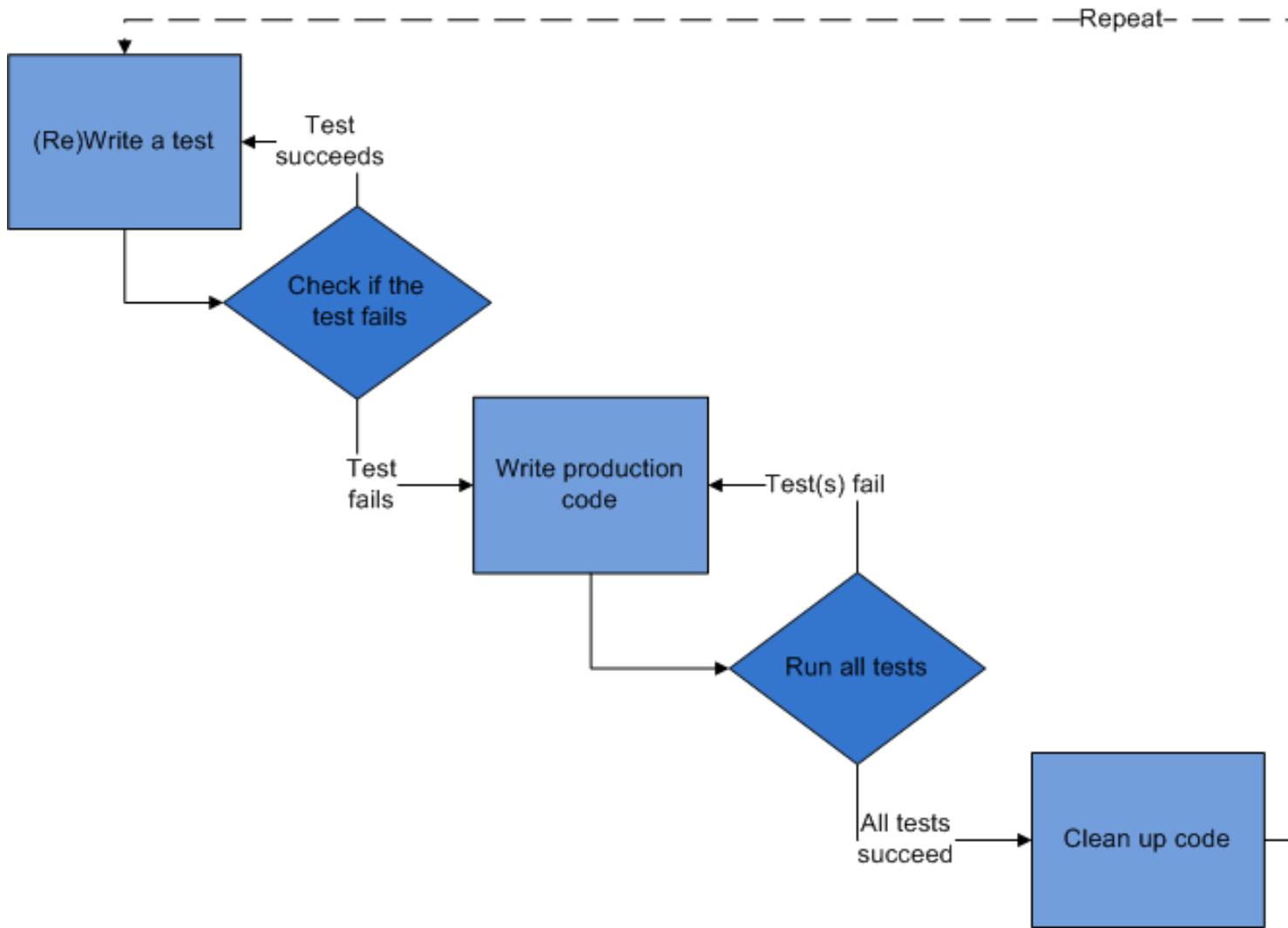
Write minimum amount of code to pass the test



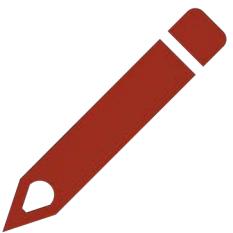
Finally re-factor the code to acceptable standards

TEST DRIVEN DEVELOPMENT

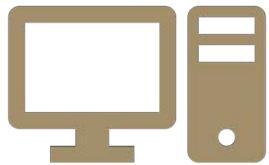
TDD



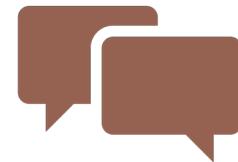
HOMEWORK



Review class notes.



Additional reading: latest
trends in software
development.

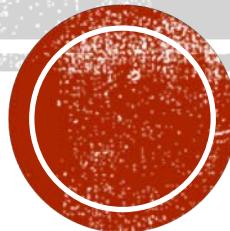


Start a discussion on Google
Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353 / 6353

Dr. Raj Singh



UML - HISTORY



The Unified Modeling Language (UML) is a general purpose modeling language designed to provide a standard way to visualize the design of a system.



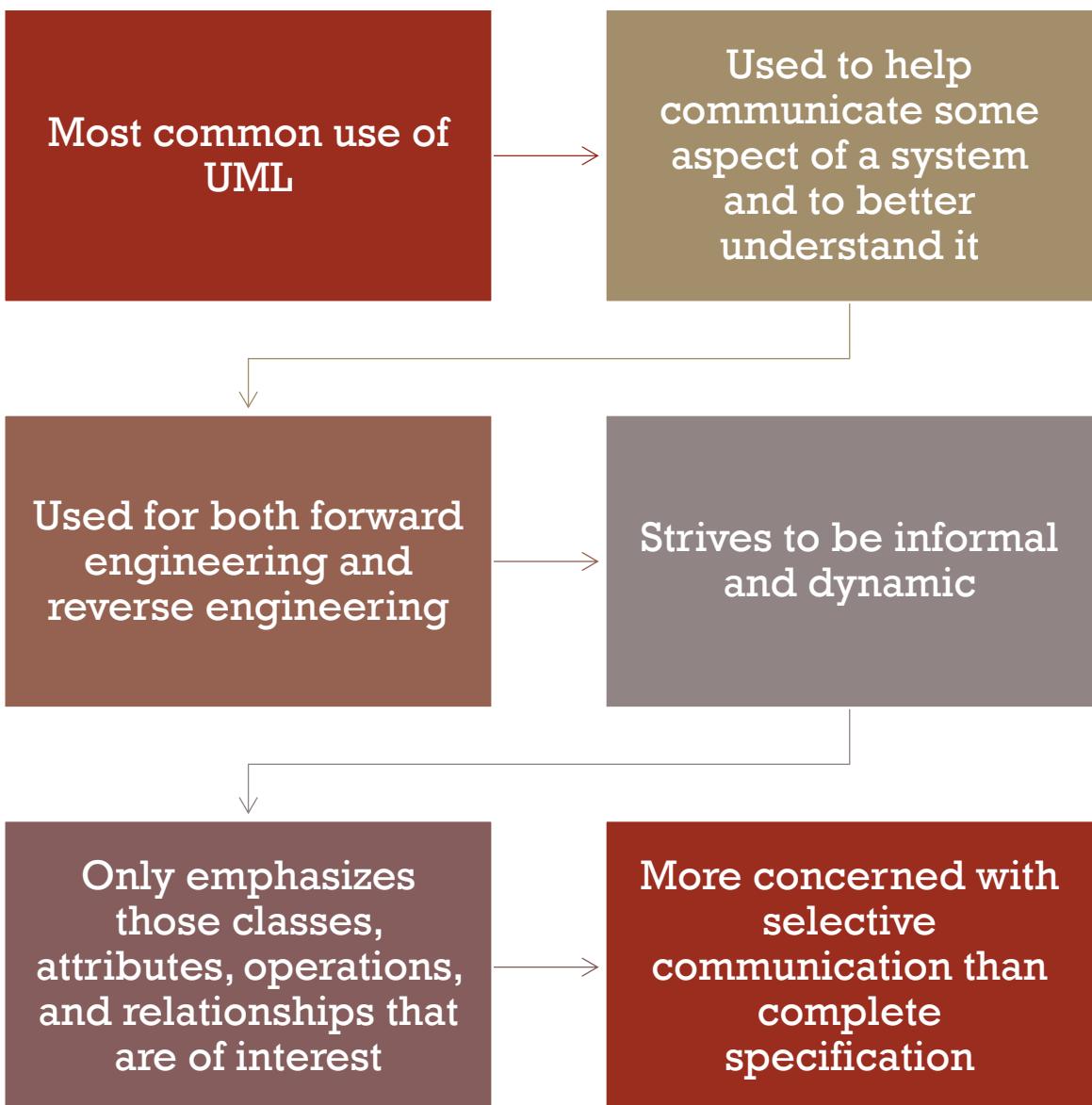
Created and developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software during 1994–95 with further development led by them through 1996.



In 1997 it was adopted as a standard by the Object Management Group (OMG)



In 2000 UML was also accepted by the International Organization for Standardization(ISO) as an approved ISO standard.



UML AS A SKETCH

UML AS A BLUEPRINT



Goal is completeness



It is more definitive, while the sketch approach is more explorative



Used to describe a detailed design for a programmer to follow in writing source code



Notation should be sufficiently complete so that a software engineer can follow it



It can be used by a designer to develop blueprint-level models that show interfaces of subsystems or classes



As a reversed engineered product, diagrams convey detailed information about the source code that is easier for software engineers to understand



Specifies the complete system in UML so that code can be automatically generated



Looks at UML from a software perspective rather than a conceptual perspective



Diagrams are compiled directly into executable code so that the UML becomes the source code



Challenge is making it more productive to use UML rather than some another programming language



Another concern is how to model behavioral logic

Done with interaction diagrams, state diagrams, and activity diagrams

UML AS A PROGRAMMING LANGUAGE



UML sketches are useful with both forward and reverse engineering and in both conceptual and software perspectives



Detailed forward engineering blueprints are difficult to do well and slow down the development effort

Actual implementation of interfaces will reveal the needs for changes



The value of reversed engineered blueprints depends on the CASE tool

A dynamic browser would be very helpful; a thick document wastes time and resources



UML as a programming language will probably never see significant usage

Graphical forms have not shown to be more productive in writing code than textual code for most programming tasks

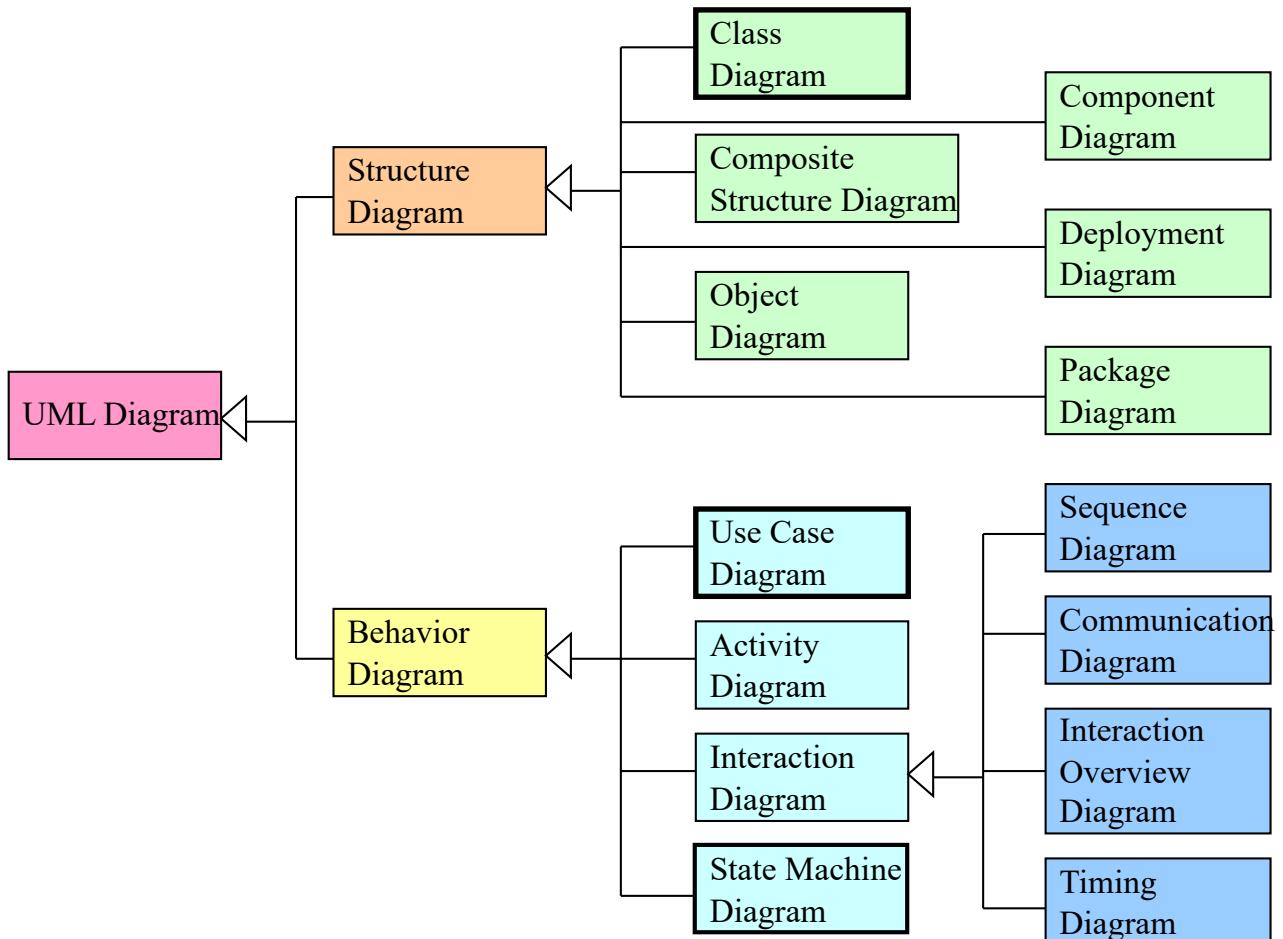
WAYS OF USING - COMPARISON

TYPES OF UML DIAGRAMS

DIAGRAM NAME	PURPOSE
Activity	Models procedural and parallel behavior
Class	Models classes, attributes, operations and relationships
Communication	Models interaction between objects
Component	Models structure and connection of components
Composite Structure	Models runtime decomposition of a class
Deployment	Models deployment of artifacts to nodes
Interaction overview	Mixes the sequence and activity diagram

TYPES OF UML DIAGRAMS (CONTINUED)

DIAGRAM NAME	PURPOSE
Object	Models example configurations of instances
Package	Models compile-time hierarchical structure
Sequence	Models sequence interaction between objects
State Machine	Models how events change an object over its life
Timing	Models timing interaction between objects
Use Case	Models how users interact with a system



CLASSIFICATION OF DIAGRAM TYPES



Use case / scenario defines how a user uses a system to accomplish a particular goal.



A modeling technique that defines the features to be implemented and the resolution of any errors that may be encountered.



A methodology used in system analysis to identify, clarify, and organize system requirements.



A set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.

SCENARIO-BASED MODELING



What are the main tasks or functions that are performed by the actor?



What system information will the the actor acquire, produce or change?



Will the actor have to inform the system about changes in the external environment?

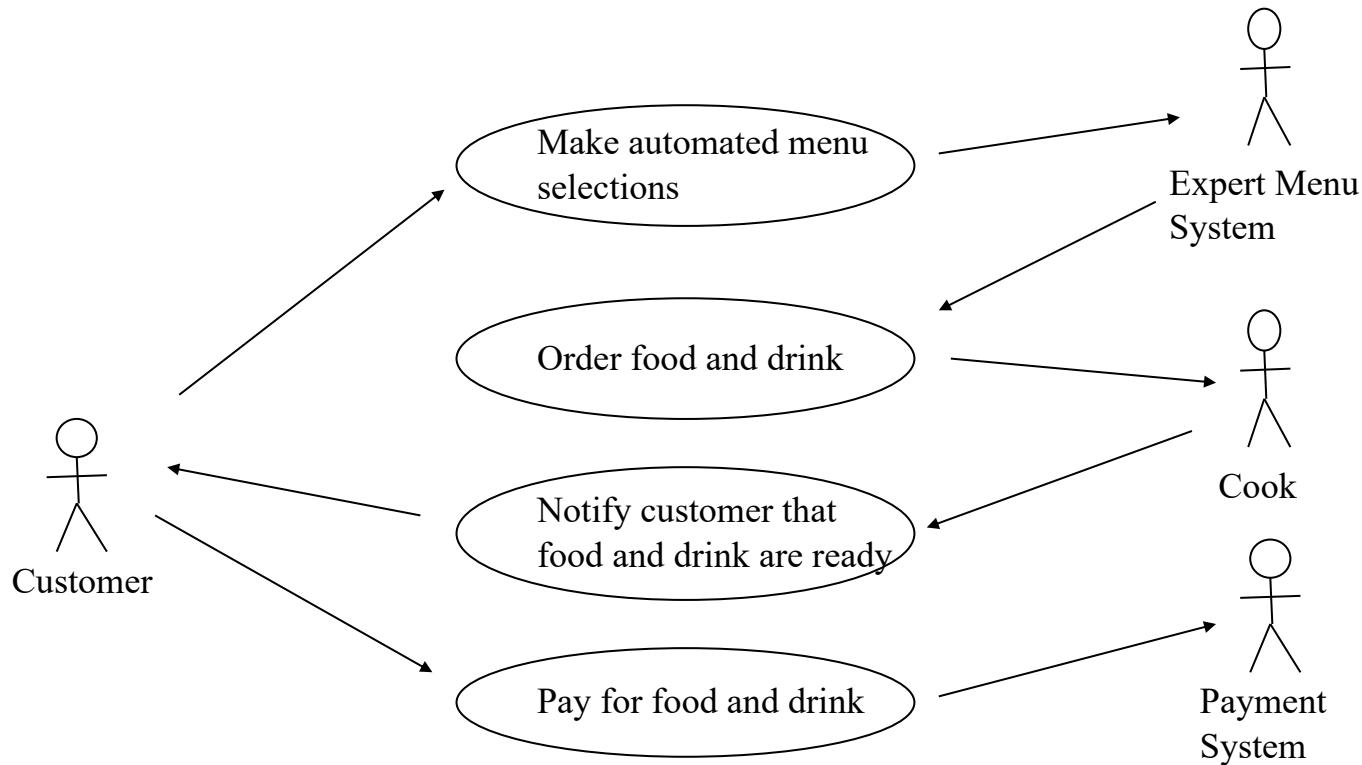


What information does the actor desire from the system?



Does the actor wish to be informed about unexpected changes?

DEVELOPING A USE-CASE



USE-CASE DIAGRAM EXAMPLE

EXCEPTIONS



Describe situations that may cause the system to exhibit unusual behavior



Brainstorm to derive a reasonably complete set of exceptions for each use case



Are there cases where a validation function occurs for the use case?



Handling exceptions may require the creation of additional use cases

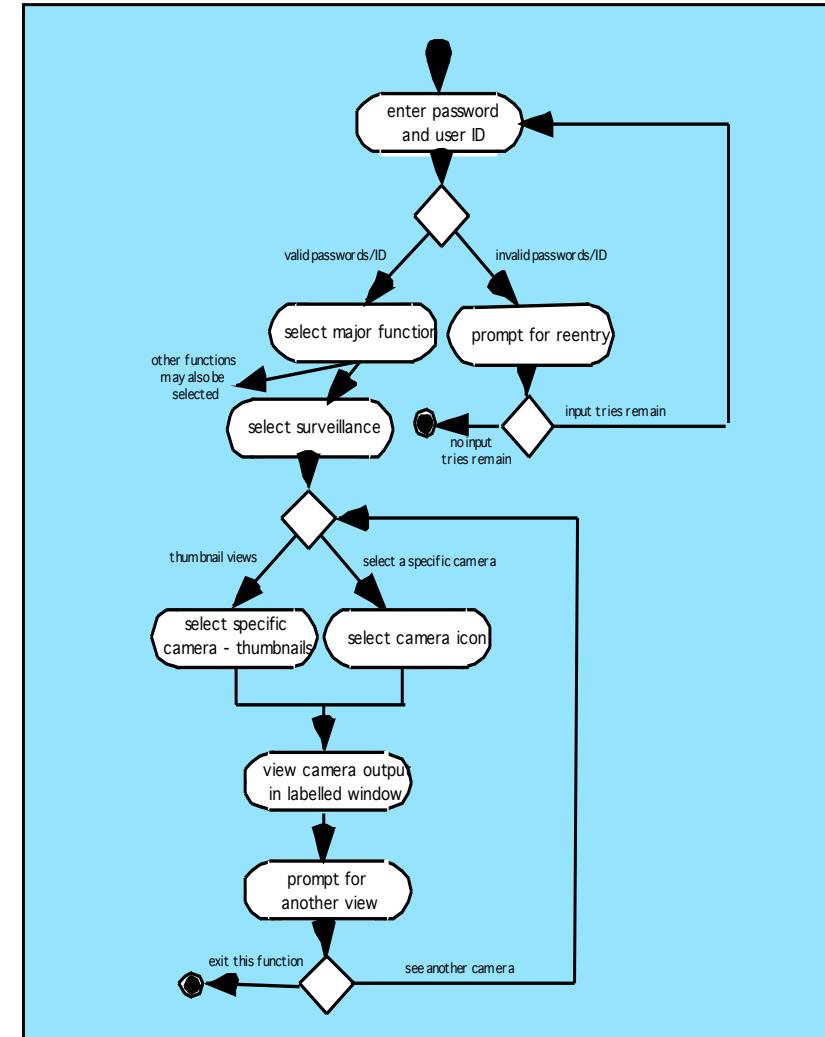
ACTIVITY DIAGRAM



Supplements the use case by providing a graphical representation



The flow of interaction between actor and the system within a specific scenario



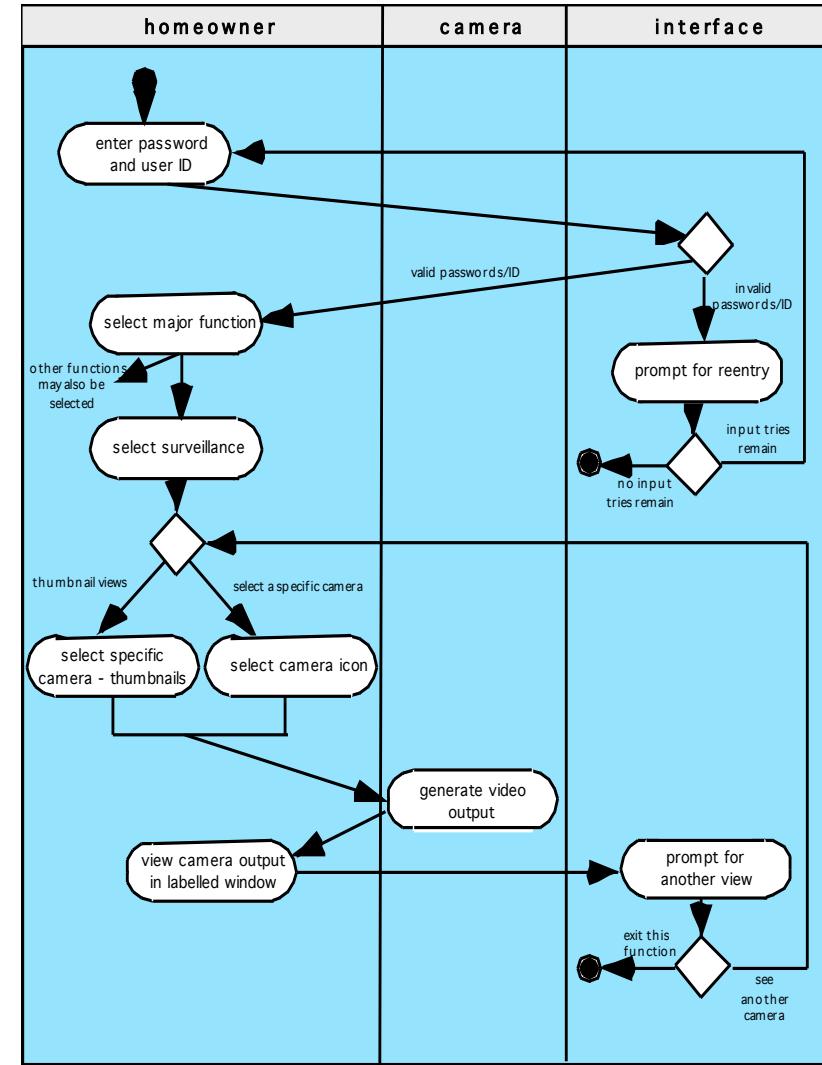
SWIMLANE DIAGRAM



Allows the modeler to represent the flow of activities described by the use-case



Indicates which actor or analysis class has responsibility for the action described by an activity rectangle

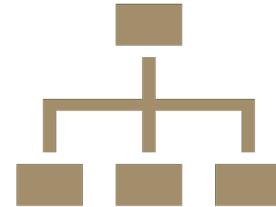


CLASS-BASED MODELING



Class-based modeling represents

objects that the system will manipulate
operations (also called methods or services)
relationships between the objects
collaborations that occur between the classes



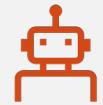
The elements of a class-based model include

classes and objects
attributes and operations
collaboration diagrams and packages

IDENTIFYING CLASSES



Classes are determined by underlining each noun or noun phrase and entering it into a simple table



If the class is required to implement a solution, then it is part of the solution space;



If a class is necessary only to describe a solution, it is part of the problem space.

External entities

- (e.g. other systems, devices, people) that produce or consume information

Things

- (e.g. reports, displays, letters, signals) that are part of the information domain for the problem

Occurrences or events

- (e.g. completion of actions) that occur within the context of system operation

Roles

- (e.g. manager, engineer, salesperson) played by people who interact with the system

Organizational units

- (e.g. division, group, team) that are relevant to an application

Places

- (e.g. manufacturing floor) that establish the context of the problem and the overall function

Structures

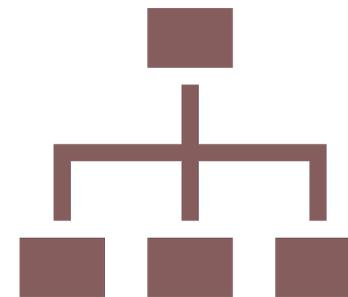
- (e.g. sensors, computers) that define a class of objects or related classes of objects

MANIFESTATIONS OF CLASSES

DEFINING ATTRIBUTES

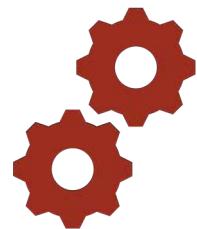


Attributes describe a class that has been selected for inclusion in the analysis model.



Attributes describe the structure and value of an instance of a class.

DEFINING OPERATIONS



An operation is a method or function that can be performed by a class.



Operations define the behavior of a class, what a class can do.



Operations can perform computation, take an action, call another method, etc.

CLASS TYPES



Entity classes

also called model or business classes, are extracted directly from the statement of the problem



Boundary classes

are used to create the interface that the user sees and interacts with the software



Controller classes

manage a “unit of work” from start to finish



System intelligence should be distributed across classes to best address the needs of the problem



Each responsibility should be stated as generally as possible



Information and the behavior related to it should reside within the same class



Information about one thing should be localized with a single class, not distributed across multiple classes.



Responsibilities should be shared among related classes, when appropriate.

RESPONSIBILITIES

COLLABORATIONS



Classes fulfill their responsibilities in one of two ways:

a class can use its own operations to fulfill a particular responsibility
a class can collaborate with other classes



Collaborations identify relationships between classes



Collaborations are identified by determining whether a class can fulfill each responsibility itself

ASSOCIATION, AGGREGATION AND COMPOSITION



Association is a (*a*) relationship between two classes, where one class use another

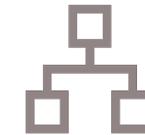


Aggregation, a special type of an association, is the (*the*) relationship between two classes.

Association is non-directional, aggregation insists a direction.



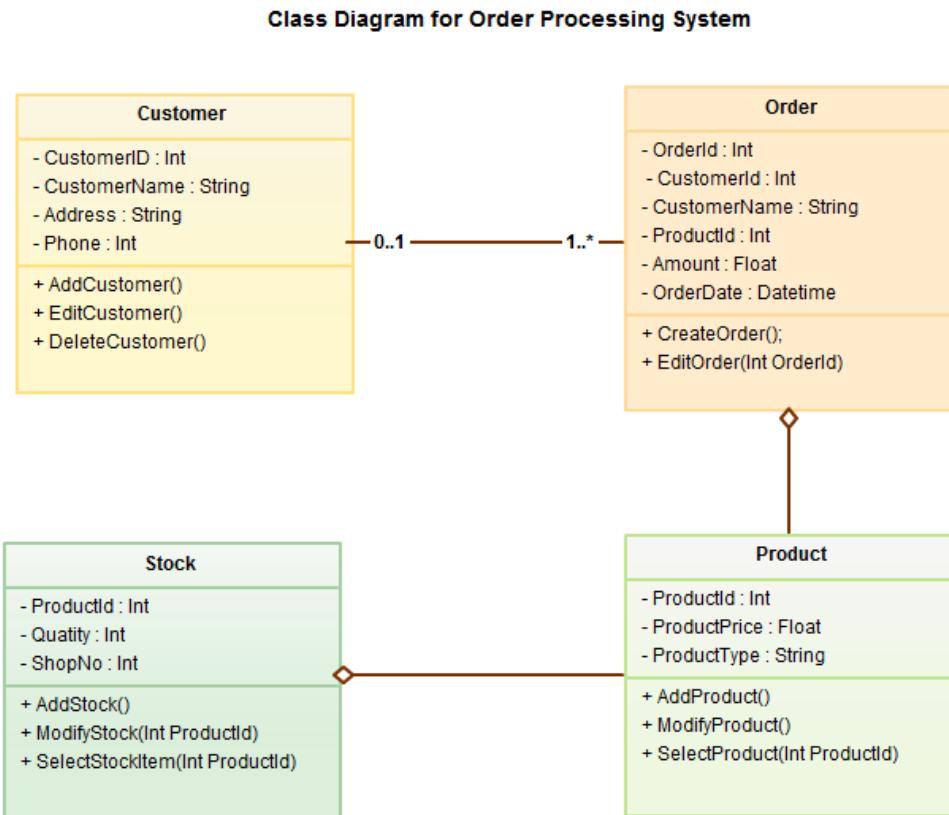
Composition can be recognized as a special type of an aggregation.

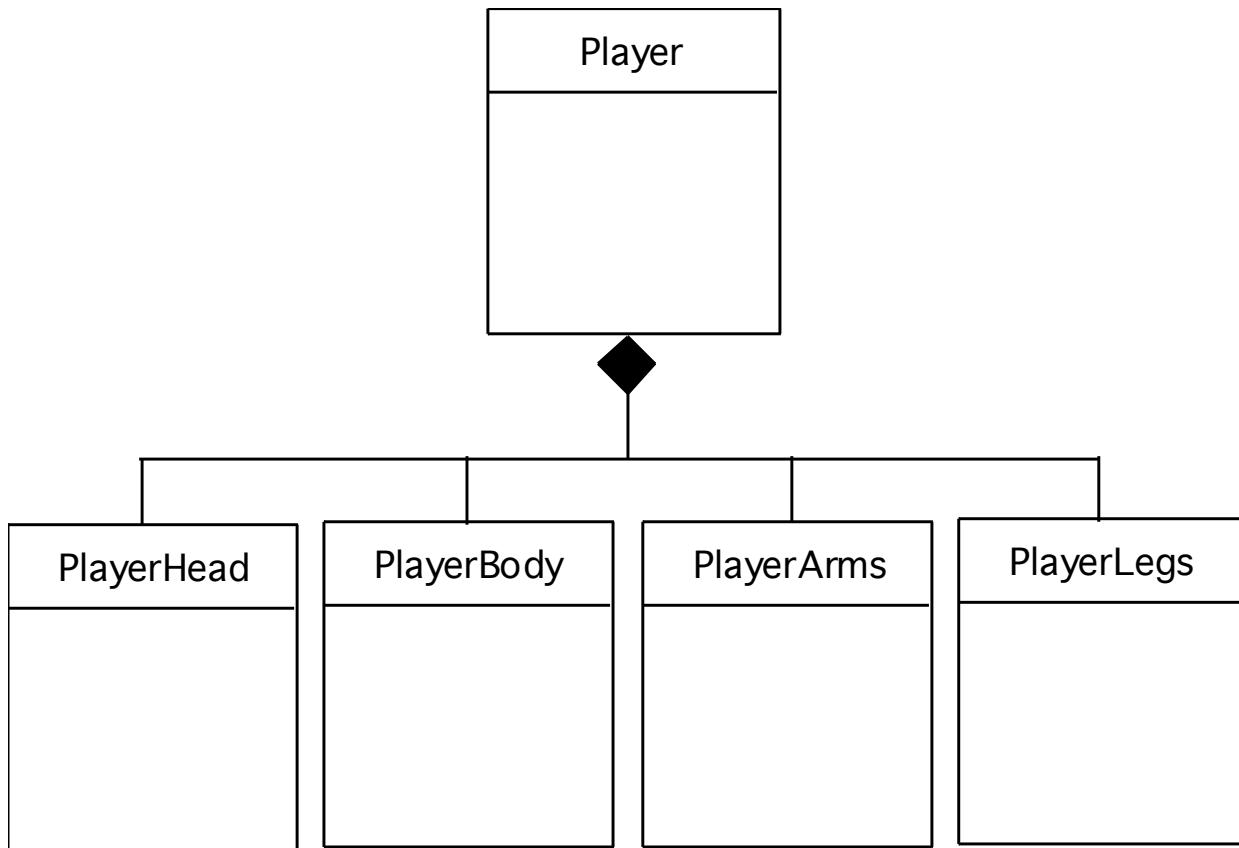


Aggregation is a special kind of an association and composition is a special kind of an aggregation.

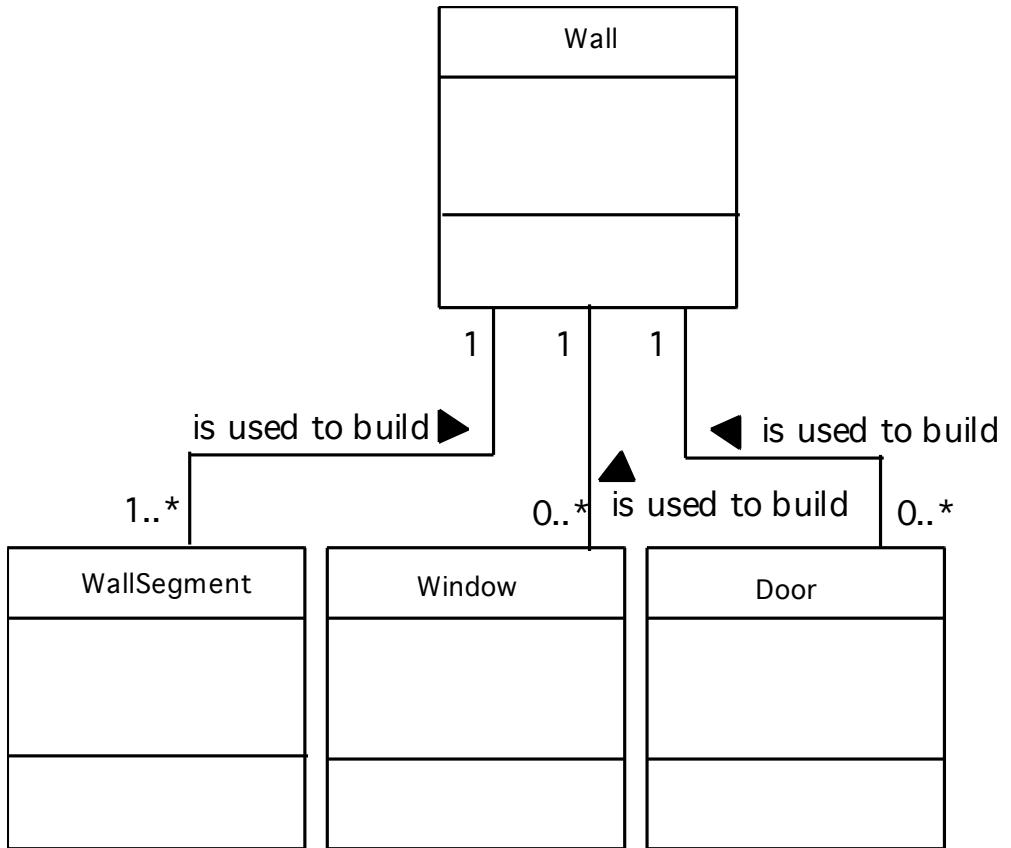
Association → Aggregation
→ Composition

EXAMPLE CLASS DIAGRAM

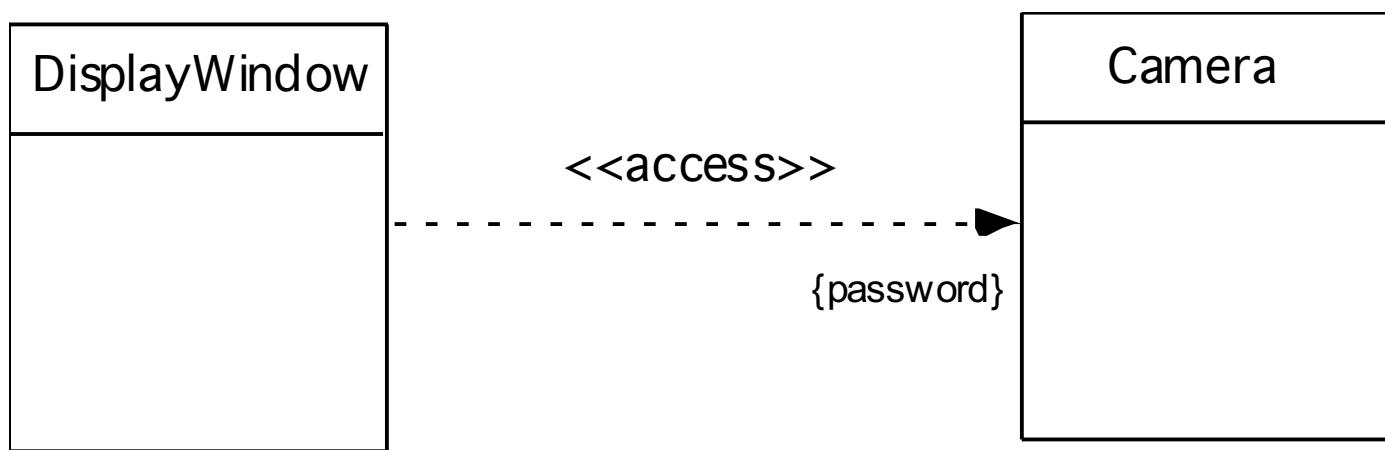




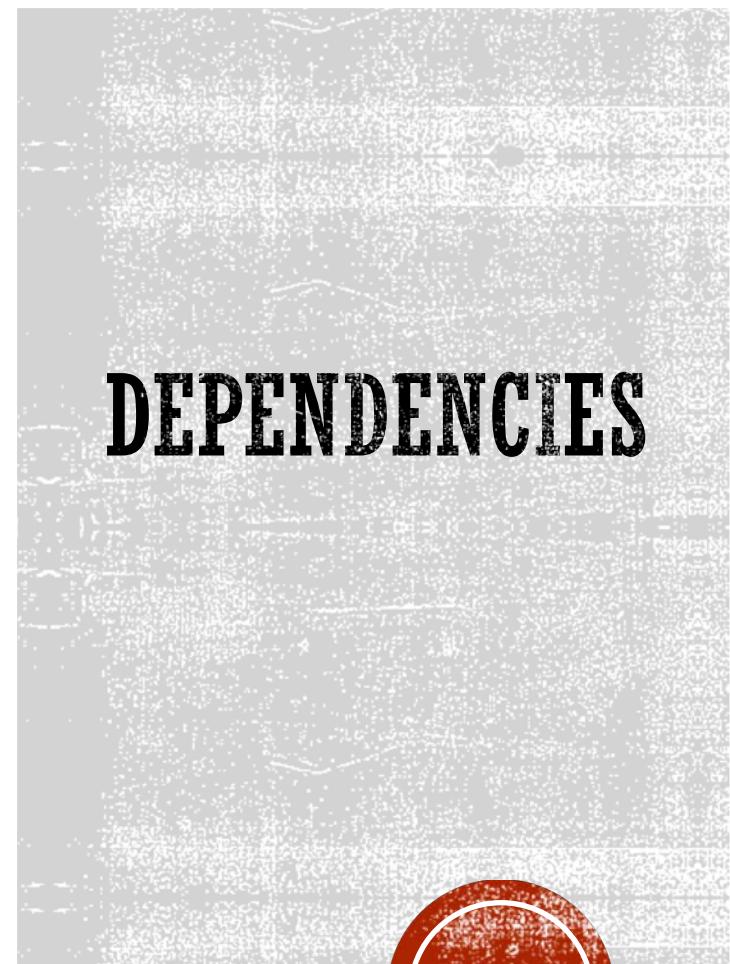
COMPOSITE AGGREGATE CLASS



MULTIPLICITY



A dependency exists between two elements if changes to the definition of one element (i.e., the source or supplier) may cause changes to the other element (i.e., the client)





The behavioral model indicates how software will respond to external events.



Evaluate all use-cases to fully understand the sequence of interaction within the system.



Identify events that drive the interaction sequence and understand how these events relate to specific objects.



Create a sequence for each use-case.



Build a state diagram for the system.



Review the behavioral model to verify accuracy and consistency.

BEHAVIORAL MODELING

STATE DIAGRAMS

In the context of behavioral modeling, two different characterizations of states must be considered:

- the state of each class as the system performs its function and
- the state of the system as observed from the outside as the system performs its function

The state of a class takes on both passive and active characteristics:

- A passive state is simply the current status of all of an object's attributes.
- The active state of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

THE STATES OF A SYSTEM



State

a set of observable circum-stances that characterizes the behavior of a system at a given time



State transition

the movement from one state to another



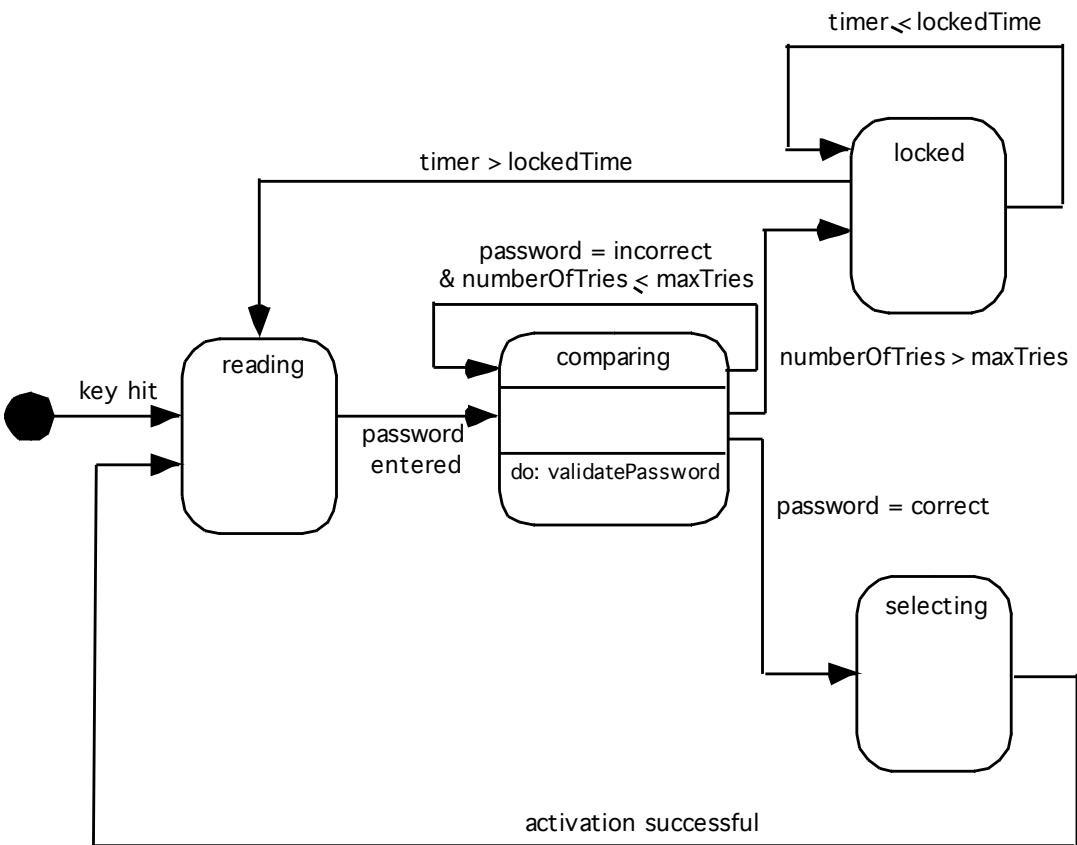
Event

an occurrence that causes the system to exhibit some predictable form of behavior

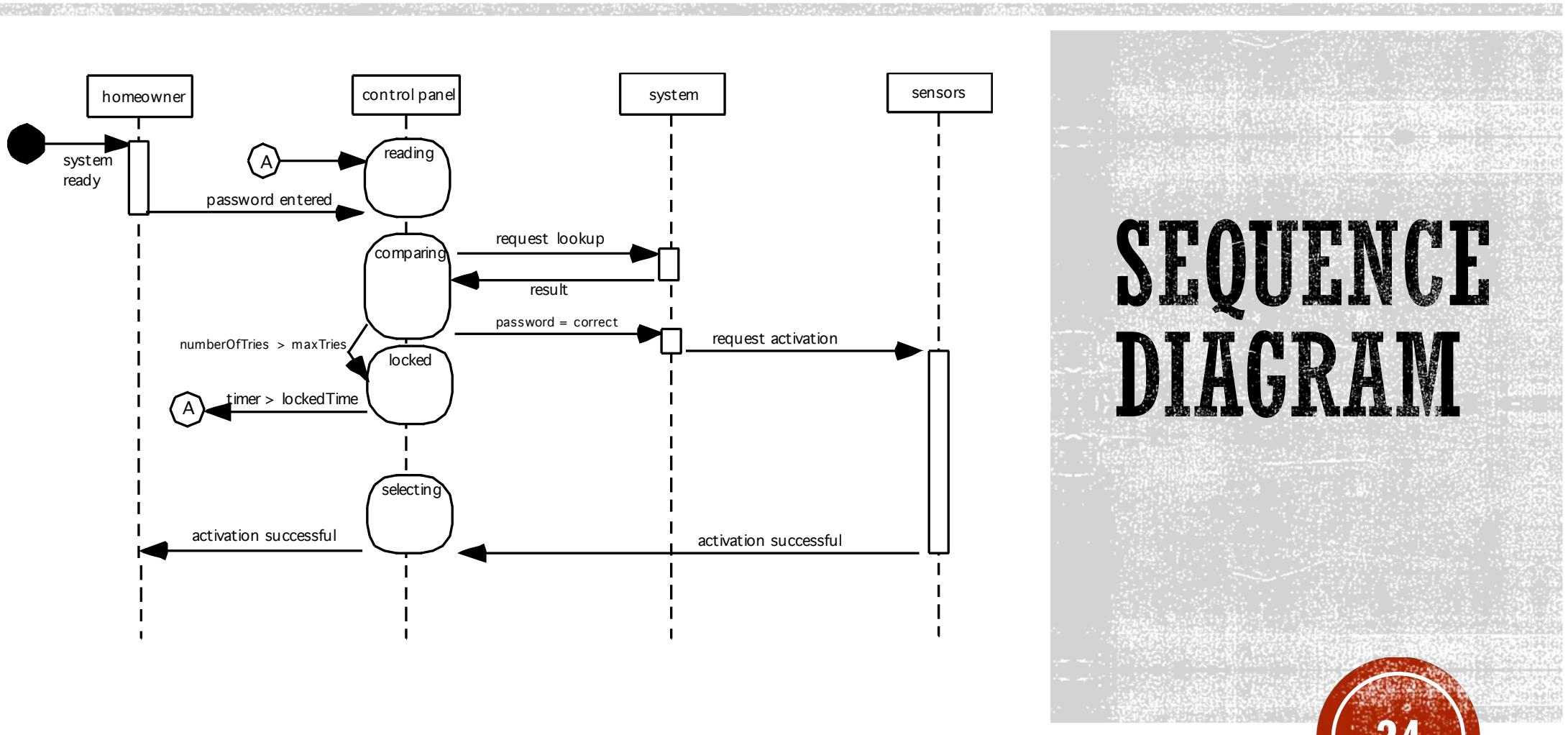


Action

process that occurs as a consequence of making a transition

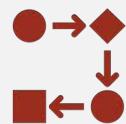


STATE DIAGRAM EXAMPLE



SEQUENCE DIAGRAM

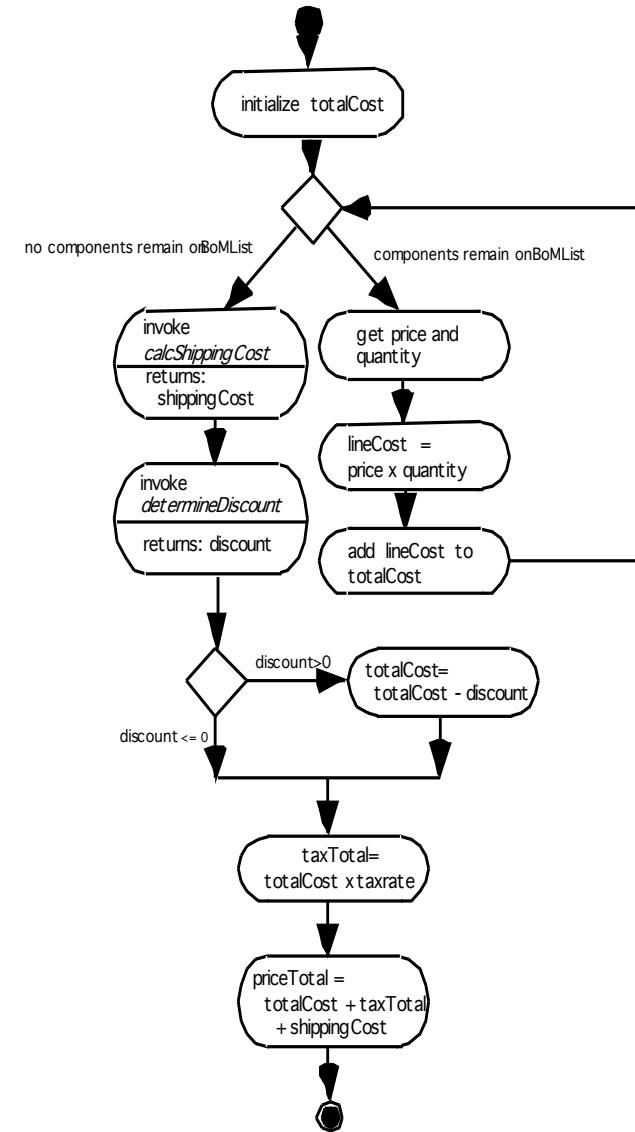
ACTIVITY DIAGRAM



A flowchart to represent the flow from one activity to another activity.



The activity can be described as an operation of the system.



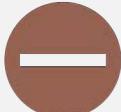
DATA FLOW DIAGRAM



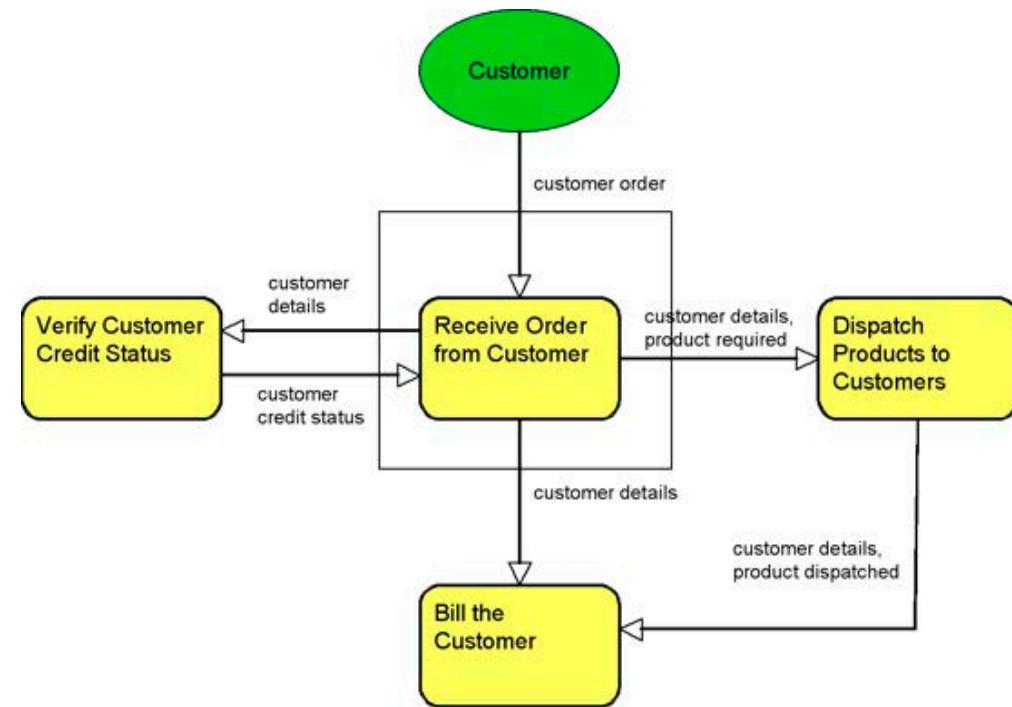
A data-flow diagram is a way of representing a flow of a data of a process or a system.



The DFD also provides information about the output and input of each entity and the process itself.



A data-flow diagram has no control flow, there are no decision rules and no loops.



OBJECT DIAGRAM



Represents a snapshot of the objects in a system at a point in time

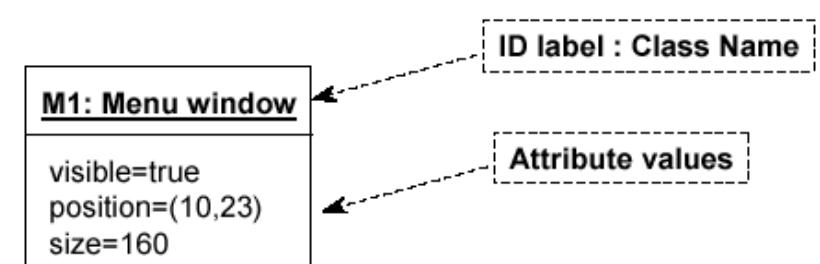


Shows instances rather than classes, therefore it is sometimes called an instance diagram



When to use object diagrams

To show examples of objects connected together based on a specific multiplicity number
To show instances with values for their attributes



PACKAGE DIAGRAM



Used to take any construct in UML and group its elements together into higher-level units



Used most often to group classes



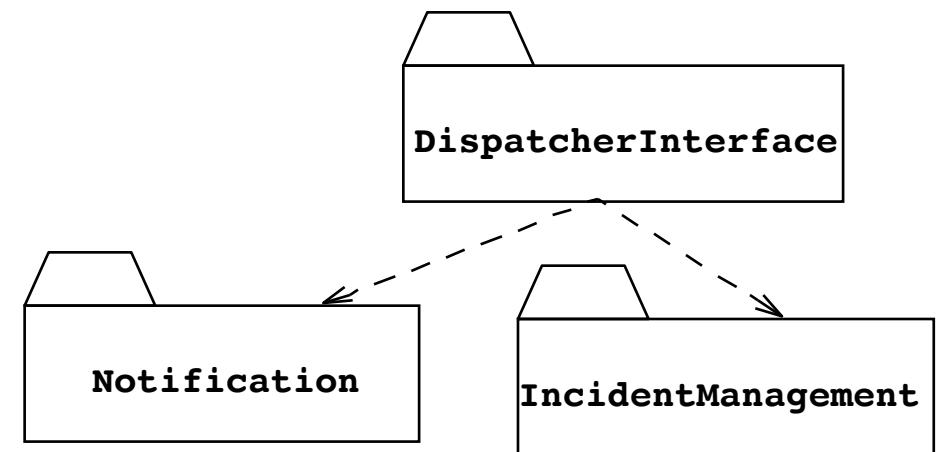
Corresponds to the package concept in Java



Represented by a tabbed folder, where the tab contains the package name



Can show dependencies between packages



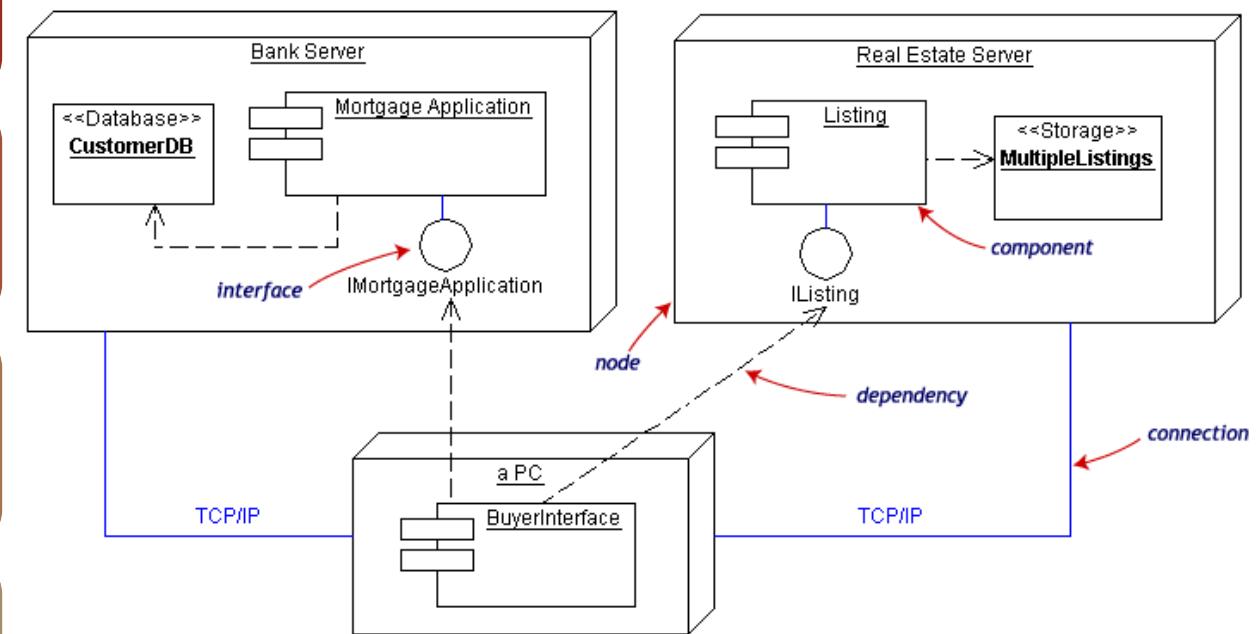
DEPLOYMENT DIAGRAM

Shows the physical architecture of the hardware and software of the deployed system

Typically contain components or packages

Physical relationships among software and hardware in a delivered systems

Explains how a system interacts with the external environment





A use case diagram helps describe how people interact with the system



An activity diagram shows the context for use cases and also the details of how a complicated use case works



A class diagram drawn from the conceptual perspective is a good way of building up a rigorous vocabulary of the domain

It also shows the attributes and operations of interest in domain classes and the relationships among the classes



A state diagram shows the various states of a domain class and events that change that state

FITTING UML INTO SOFTWARE REQUIREMENTS ANALYSIS



A class diagram drawn from the software perspective can show design classes, their attributes and operations, and their relationships with the domain classes



A sequence diagram helps to combine use cases in order to see what happens in the software



A package diagram shows the large-scale organization of the software



A state diagram shows the various states of a design object and events that change that state



A deployment diagram shows the physical layout of the software

FITTING UML INTO SOFTWARE DESIGN



Complements the written documentation and in some instances can replace it



Captures the outcome of the requirements analysis and design activities in a graphical format



Supplies a software maintainer with an overall understanding of a system



Provides a good logical roadmap of the system layout



Describes the various states in which a system may exist



Details complex algorithms in a more understandable form



Shows how multiple objects collaborate in the system

FITTING UML INTO SOFTWARE DOCUMENTATION

HOMEWORK



Review class notes.



Additional reading:
Examples of UML diagrams

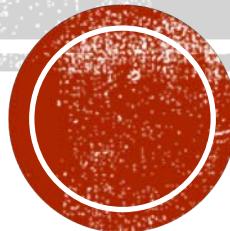


Start a discussion on Google
Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





Test Driven Development



Best practices



Advantages and Disadvantages



Fakes, mocks, and integration tests



Example

OUTLINE

INTRODUCTION

 A software development process.

 Repetition of a very short development cycle

 First write an initially failing automated test case that defines a desired improvement or new function

 then produce the minimum amount of code to pass that test

 and finally re-factor the new code to acceptable standards.

 Related to XP



TDD is a design (and testing) approach involving short, rapid iterations of

Unit Test
Code
Re-factor



Unit tests are automated



Forces to consider use of a method before implementation of the method

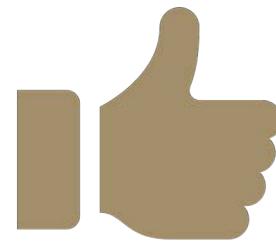
WHAT IS TDD?

WHY TDD?



Software development

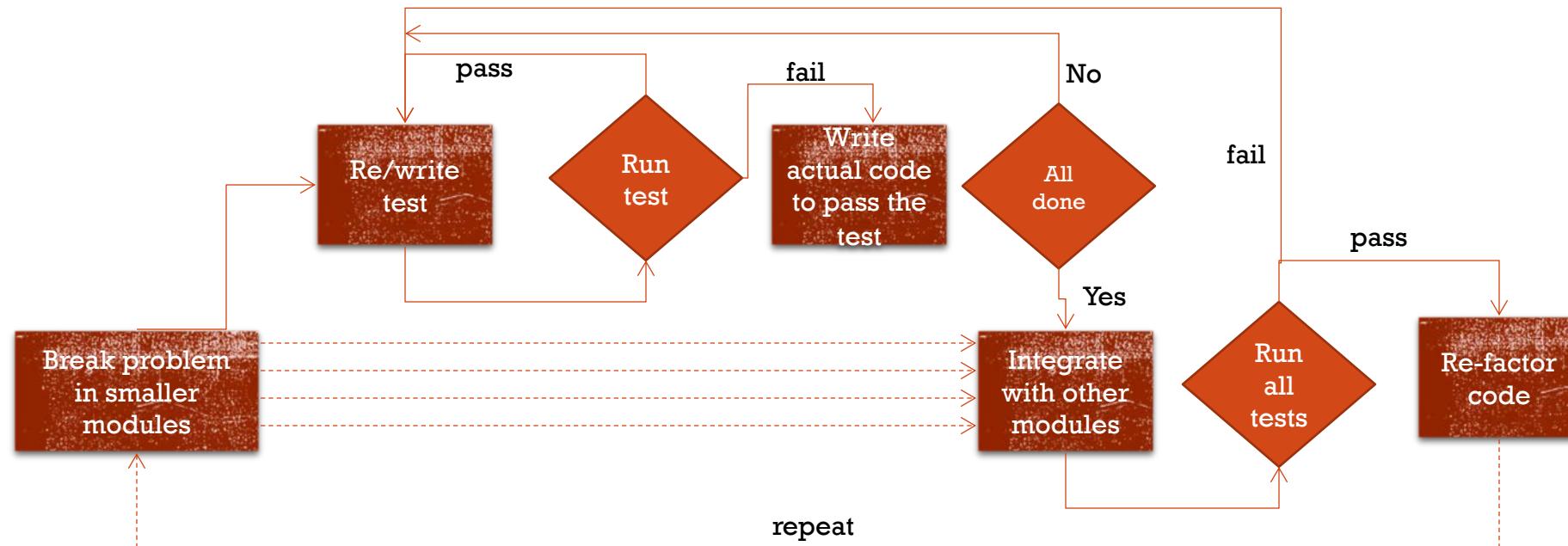
Should be faster
As economical as it could be
Good quality product



With TDD

Good programmers are more effective
Testing can close the gap
Quality improvement

TEST-DRIVEN DEVELOPMENT CYCLE





"keep it simple stupid" (KISS)



"you aren't gonna need it" (YAGNI)



By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer.



To achieve some advanced design concept, such as a design pattern, tests are written that generate that design.



The code may remain simpler than the target pattern, but still pass all required tests.

DEVELOPMENT STYLE



For TDD, a unit is most commonly defined as a class or group of related functions often called a module.



Keeping units relatively small is claimed to provide critical benefits.



Reduced Debugging Effort

When test failures are detected, having smaller units aids in tracking down errors.



Self-Documenting Tests

Small test cases have improved readability and facilitate rapid understandability.

DEVELOPMENT STYLE



Advanced practices of TDD can lead to Acceptance Test-driven development (ATDD) .



The criteria specified by the customer are automated into acceptance tests.



Give customer an automated mechanism to decide whether the software meets their requirements.



Focused development.

DEVELOPMENT STYLE



A commonly applied structure for test cases has:
setup, execution,
validation, and
cleanup



Treat your test code with the same respect as your production code.



It must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.



Review your tests and test practices with team to share effective techniques and catch bad habits.

BEST PRACTICES



Do not have test cases depend on system state manipulated from previously executed test cases.



Execution order has to be specifiable and/or constant.



Do not test precise execution behavior timing or performance.



Do not try to build “all-knowing oracles.”

PRACTICES TO AVOID, OR "ANTI-PATTERNS"

ADVANTAGES



A study shows that programmer using TDD are:

more productive
Rarely debug
Effective and efficient system design



Few failed systems



No unnecessary code



Focus on task



Reduced bugs in the code



More time needed initially



Requirements change



May not be flexible if only pass and fail is required



Code maintenance



Complex systems may require extended development

DISADVANTAGES



Unit tests focus on a unit



A complex module may have a thousand unit tests



Tests used for TDD should never cross process boundaries in a program.



Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite.



Introducing dependencies on external modules or data also turns unit tests into integration tests.

FAKES, MOCKS AND INTEGRATION TESTS



When code under development relies on a database, a web service, or any other external process

an interface should be defined that describes the access available



The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock.



Fake and mock objects:

methods return data to help the test process by always returning the same, realistic data that tests can rely upon



Integration tests:

any database should always be designed carefully with consideration of the initial and final state of the database, even if any test fails.

FAKES, MOCKS AND INTEGRATION TESTS

EXAMPLE – TEST-DRIVEN FIBONACCI

- The Fibonacci numbers or series or sequence are the numbers in the following integer sequence:

0, 1, 1, ,2 ,3, 5, 8, 13, 21, 34, ...

- The sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0, F_1 = 1$$

EXAMPLE – TEST-DRIVEN FIBONACCI

- The second test shows that $\text{fib}(1) = 1$.

```
public void testFibonacci() {  
    assertEquals(0, fib(0));  
    assertEquals(1, fib(1));  
}
```

```
int fib(int n) {  
    return 0;  
}
```

EXAMPLE – TEST-DRIVEN FIBONACCI

- The first test shows that $\text{fib}(0) = 0$. The implementation returns a constant.

```
public void testFibonacci() {  
    assertEquals(0, fib(0));  
}
```

```
int fib(int n) {  
    if (n == 0) return 0;  
    return 1;  
}
```

EXAMPLE – TEST-DRIVEN FIBONACCI

- We can factor out the common structure of the assertions by driving the test from a table of input and expected values.

```
public void testFibonacci() {  
    int cases[][]= {{0,0},{1,1}};  
    for (int i= 0; i < cases.length; i++)  
        assertEquals(cases[i][1], fib(cases[i][0]));  
}
```

EXAMPLE – TEST-DRIVEN FIBONACCI

- Now adding the next case requires 6 keystrokes and no additional lines:

```
public void testFibonacci() {  
    int cases[][]= {{0,0},{1,1},{2,1}};  
    for (int i= 0; i < cases.length; i++)  
        assertEquals(cases[i][1], fib(cases[i][0]));  
}
```

- The test works. It just so happens that our constant “1” is right for this case as well.

EXAMPLE – TEST-DRIVEN FIBONACCI

- On to the next test:

```
public void testFibonacci() {  
    int cases[][]= {{0,0},{1,1},{2,1},{3,2}};  
    for (int i= 0; i < cases.length; i++)  
        assertEquals(cases[i][1], fib(cases[i][0]));  
}
```

- Hooray, it fails. Applying the same strategy as before (treating smaller inputs as special cases), we write:

EXAMPLE – TEST-DRIVEN FIBONACCI

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return 2;  
}
```

- Now we are ready to generalize. We wrote “2”, but we don’t really mean “2”, we mean “ $l + l$ ”.

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return l + l;  
}
```

EXAMPLE – TEST-DRIVEN FIBONACCI

- That first “1” is an example of $\text{fib}(n-1)$:

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return fib(n-1) + 1;  
}
```

- The second “1” is an example of $\text{fib}(n-2)$:

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

EXAMPLE – TEST-DRIVEN FIBONACCI

- Cleaning up now, the same structure should work for `fib(2)`, so we can tighten up the second condition:

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

- And there we have Fibonacci, derived totally from the tests.

HOMEWORK



Review class notes.



Additional reading:
Examples of UML diagrams



Start a discussion on Google Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





Software Architecture



Object and Class



Object Oriented Principles

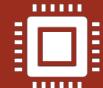


OOP Example



Terms to Remember

OUTLINE



A high level structure of a software system.



Rules, heuristics and patterns for system design.



Partitioning the system into discrete pieces



Techniques

used to create interfaces between the pieces
manage overall structure and flow
interface the system to its environment



Defines appropriate use of development and delivery techniques and tools.

SOFTWARE ARCHITECTURE



Controls complexity



Enforces best practices



Provides consistency and uniformity



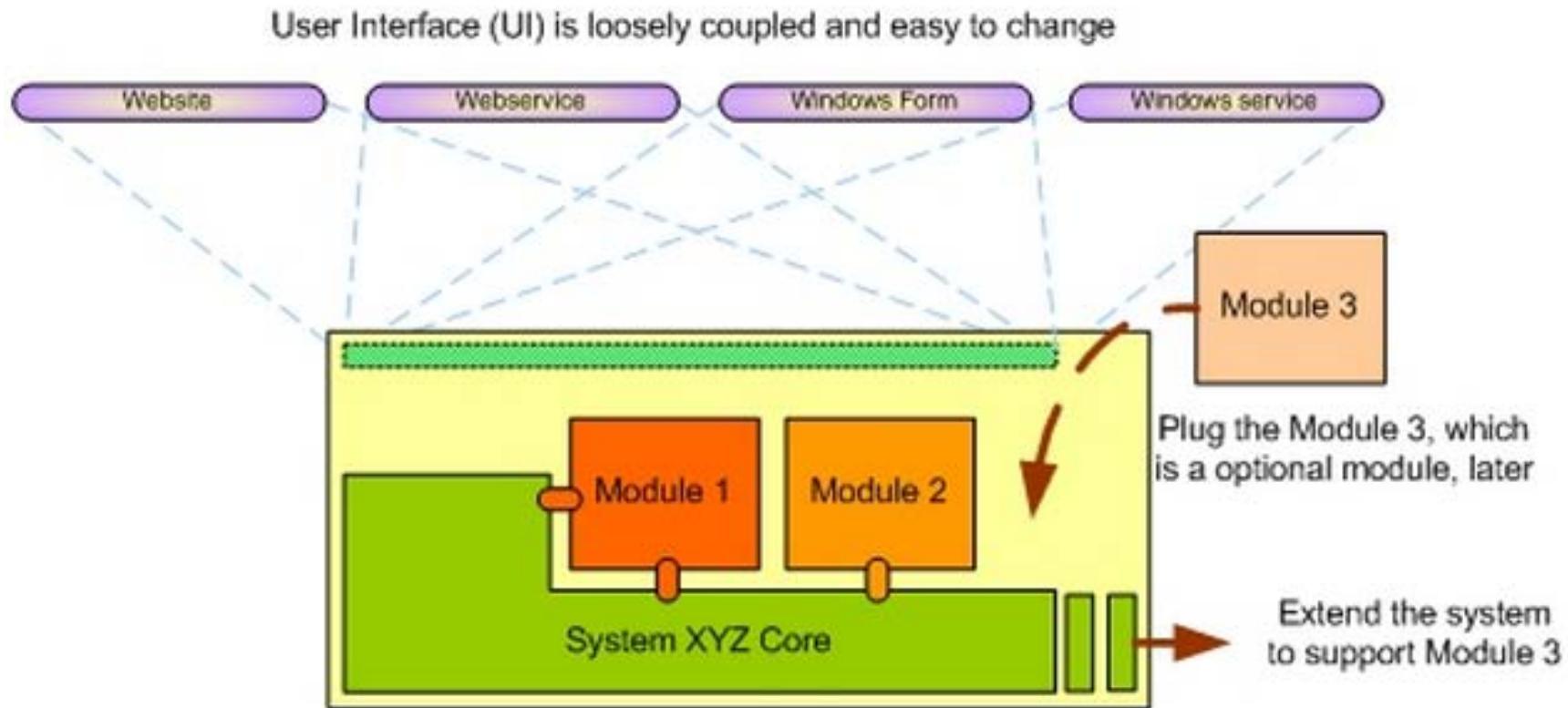
Increases predictability



Enables reusability

ARCHITECTURE IMPORTANCE

ARCHITECTURE EXAMPLE



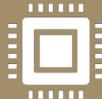
OBJECT-ORIENTED PROGRAMMING / DESIGN (OOP/OOD)

”

A paradigm that represents concepts as "objects" that have attributes that describe the object and associated procedures known as methods.



Objects, which are usually instances of classes, are used to interact with one another to design applications.



Objective-C, Smalltalk, Java and C# are examples of object-oriented programming languages.

OBJECT AND CLASS



An object can be considered as a "thing" that can perform a set of related activities.



The set of activities that the object performs defines the object's behavior.



In pure OOP terms an object is an instance of a class.

OBJECT AND CLASS

- A class is a representation of a type of object.
- It describes the details of an object.
- Class is composed of three things: name, attributes, and operations.

```
public class Student {  
    private String name;  
    ...  
    private void method1(){  
        ...  
    }  
    ...  
}  
Student objectStudent = new Student();  
• student object, named objectStudent, is created out of the Student class.
```

CLASS DESIGN PRINCIPLE



SRP - The Single Responsibility Principle

A class should have one, and only one, reason to change.



OCP - The Open Closed Principle

You should be able to extend a classes behavior, without modifying it.



LSP - The Liskov Substitution Principle

Derived classes must be substitutable for their base classes.



DIP - The Dependency Inversion Principle

Depend on abstractions, not on concretions.



ISP - The Interface Segregation Principle

Make fine grained interfaces that are client specific.



Encapsulation

information hiding



Abstraction

define, don't
implement



Inheritance

extensibility



Polymorphism

one object many
shapes

MAIN OOP/OOD CONCEPTS

ASSOCIATION, AGGREGATION AND COMPOSITION



Association is a (*a*) relationship between two classes, where one class use another

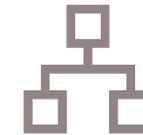


Aggregation, a special type of an association, is the (*the*) relationship between two classes.

Association is non-directional, aggregation insists a direction.



Composition can be recognized as a special type of an aggregation.



Aggregation is a special kind of an association and composition is a special kind of an aggregation.

Association → Aggregation
→ Composition

EXAMPLE



University aggregate Chancellor. University can exist without a Chancellor.



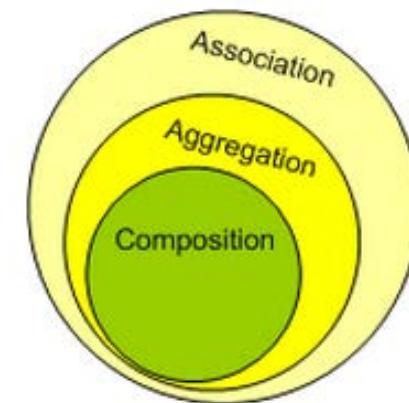
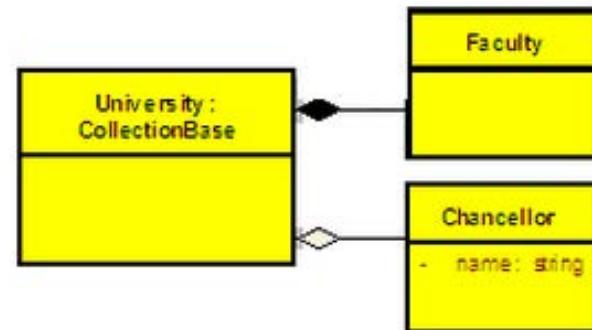
Faculties cannot exist without the University.



The life time of a Faculty is attached with the life time of the University .



University is composed of Faculties.



ABSTRACTION AND GENERALIZATION

“ , ”

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars.



“What” rather than “How”



Generalization is the broadening of application to encompass a larger domain of objects of the same or different type.



Abstraction and generalization are often used together.



Software reusability

Reuse an existing class and its behavior



Create new class from an existing class

Absorb existing class's data and behaviors
Enhance with new capabilities



Subclass extends superclass

More specialized group of objects
Behaviors inherited from superclass

INHERITANCE



Classes that are too general to create real objects



Used only as abstract superclasses for concrete subclasses and to declare reference variables



Many inheritance hierarchies have abstract superclasses occupying the top few levels



Keyword abstract

Use to declare a class abstract
Also use to declare a method abstract



Abstract classes normally contain one or more abstract methods



All concrete subclasses must override all inherited abstract methods

ABSTRACT CLASSES AND METHODS



Interfaces are used to separate design from coding as class method headers are specified but not their bodies.



Interfaces are similar to abstract classes but all methods are abstract and all properties are static final.



Interfaces can be inherited (i.e.. you can have a sub-interface).



An interface is used to tie elements of several classes together.

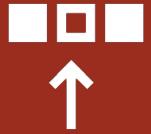


This allows compilation and parameter consistency testing prior to the coding phase.



Interfaces are also used to set up unit testing frameworks.

INTERFACES



Facilitates adding new classes to a system with minimal modifications



When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable



The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked

POLYMORPHISM



Overloading

More than one method in a class with same name different signature.

Does not depend on return type.



Overriding

Method in a subclass with same name and return type.



Dynamic Binding

Also known as late binding

Calls to overridden methods are resolved at execution time, based on the type of object referenced

TYPES OF POLYMORPHISM

POLYMORPHISM EXAMPLE: EMPLOYEE HIERARCHY CLASSES

	earnings	toString
Employee	abstract	<i>firstName lastName social security number: SSN</i>
Salaried-Employee	weeklySalary	salaried employee: <i>firstName lastName social security number: SSN weekly salary: weeklySalary</i>
Hourly-Employee	<i>If hours <= 40 wage * hours If hours > 40 40 * wage + (hours - 40) * wage * 1.5</i>	hourly employee: <i>firstName lastName social security number: SSN hourly wage: wage; hours worked: hours</i>
Commission-Employee	commissionRate * grossSales	commission employee: <i>firstName lastName social security number: SSN gross sales: grossSales; commission rate: commissionRate</i>
BasePlus-Commission-Employee	<i>(commissionRate * grossSales) + baseSalary</i>	base salaried commission employee: <i>firstName lastName social security number: SSN gross sales: grossSales; commission rate: commissionRate; base salary: baseSalary</i>

EMPLOYEE CLASS

```
1 // Employee.java
2 // Employee abstract superclass.
3
4 public abstract class Employee ← Declare abstract class Employee
5 {
6     private String firstName; ← Attributes common to all employees
7     private String lastName;
8     private String socialSecurityNumber;
9     public Employee( String first, String last){
10         firstName = first;
11         lastName = last;
12     }
13     public Employee( String first, String last, String ssn){ ← Method Overloading
14         firstName = first;
15         lastName = last;
16         socialSecurityNumber = ssn;
17     } // end three-argument Employee constructor
```

EMPLOYEE CLASS ...

```
18 // set first name
19 public void setFirstName( String first )
20 {
21     firstName = first;
22 } // end method setFirstName
23
24 // return first name
25 public String getFirstName()
26 {
27     return firstName;
28 } // end method getFirstName
29
30 // set last name
31 public void setLastName( String last )
32 {
33     lastName = last;
34 } // end method setLastName
35
36 // return last name
37 public String getLastname()
38 {
39     return lastName;
40 } // end method getLastname
41
```

EMPLOYEE CLASS . . .

```
42 // set social security number
43 public void setSocialSecurityNumber( String ssn )
44 {
45     socialSecurityNumber = ssn; // should validate
46 } // end method setSocialSecurityNumber
47
48 // return social security number
49 public String getSocialSecurityNumber()
50 {
51     return socialSecurityNumber;
52 } // end method getSocialSecurityNumber
53
54 // return String representation of Employee object
55 public String toString()
56 {
57     return String.format( "%s %s\nsocial security number: %s",
58             getFirstName(), getLastName(), getSocialSecurityNumber() );
59 } // end method toString
60
61 // abstract method overridden by subclasses
62 public abstract double earnings(); // no implementation here
63 } // end abstract class Employee
```

abstract method **earnings**
has no implementation

SALARIED EMPLOYEE SUBCLASS

```
1 // SalariedEmployee.java
2 // SalariedEmployee class extends Employee.
3
4 public class SalariedEmployee extends Employee ← Class SalariedEmployee
5 { extends class Employee
6     private double weeklySalary; ← Call superclass constructor
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, String ssn,
10         double salary ) ← Call setWeeklySalary method
11     {
12         super( first, last, ssn ); // pass to Employee constructor
13         setWeeklySalary( salary ); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15
16     // set salary
17     public void setWeeklySalary( double salary )
18     {
19         weeklySalary = salary < 0.0 ? 0.0 : salary; ← Validate and set weekly salary value
20     } // end method setWeeklySalary
21
```

SALARIED EMPLOYEE SUBCLASS

```
22 // return salary
23 public double getWeeklySalary()
24 {
25     return weeklySalary;
26 } // end method getWeeklySalary
27
28 // calculate earnings; override abstract method earnings in Employee
29 public double earnings() ←
30 {
31     return getWeeklySalary();
32 } // end method earnings
33
34 // return String representation of SalariedEmployee object
35 public String toString() ←
36 {
37     return String.format("salaried employee: %s\n%s: $%,.2f",
38             super.toString(), "weekly salary", getWeeklySalary());
39 } // end method toString
40 } // end class SalariedEmployee
```

Override **earnings** method so
SalariedEmployee can be concrete

Override **toString** method

Call superclass's version of **toString**

```
1 // HourlyEmployee.java
2 // HourlyEmployee class extends Employee.
3
4 public class HourlyEmployee extends Employee ← Class HourlyEmployee extends class
5 { Employee
6     private double wage; // wage per hour
7     private double hours; // hours worked for week
8
9     // five-argument constructor
10    public HourlyEmployee( String first, String last, String ssn,
11                           double hourlyWage, double hoursWorked ) ← Call superclass constructor
12    {
13        super( first, last, ssn );
14        setWage( hourlyWage ); // validate hourly wage
15        setHours( hoursWorked ); // validate hours worked
16    } // end five-argument HourlyEmployee constructor
17
18    // set wage
19    public void setWage( double hourlyWage ) ← Validate and set hourly wage value
20    {
21        wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22    } // end method setWage
23
24    // return wage
25    public double getWage()
26    {
27        return wage;
28    } // end method getWage
29
```

HOURLY EMPLOYEE SUBCLASS

```

30 // set hours worked
31 public void setHours( double hoursworked )
32 {
33     hours = ( ( hoursworked >= 0.0 ) && ( hoursworked <= 168.0 ) ) ?
34         hoursworked : 0.0;
35 } // end method setHours
36
37 // return hours worked
38 public double getHours()
39 {
40     return hours;
41 } // end method getHours
42
43 // calculate earnings; override abstract method earnings in Employee
44 public double earnings() ←
45 {
46     if ( getHours() <= 40 ) // no overtime
47         return getWage() * getHours();
48     else
49         return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50 } // end method earnings
51
52 // return String representation of HourlyEmployee object
53 public String toString() ←
54 {
55     return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
56     super.toString() → "hourly wage", getWage(),
57     "hours worked", getHours() );
58 } // end method toString
59 } // end class HourlyEmployee

```

Validate and set hours worked value

Override **earnings** method so
HourlyEmployee can be concrete

Override **toString** method

Call superclass's **toString** method

HOURLY EMPLOYEE SUBCLASS

COMMISSION EMPLOYEE SUBCLASS

```
1 // CommissionEmployee.java
2 // CommissionEmployee class extends Employee.
3
4 public class CommissionEmployee extends Employee ← Class CommissionEmployee
5 {                                                 extends class Employee
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // five-argument constructor
10    public CommissionEmployee( String first, String last, String ssn,
11        double sales, double rate )
12    {
13        super( first, last, ssn ); ← Call superclass constructor
14        setGrossSales( sales );
15        setCommissionRate( rate );
16    } // end five-argument CommissionEmployee constructor
17
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21        commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22    } // end method setCommissionRate ← Validate and set commission rate value
23
```

```
24 // return commission rate
25 public double getCommissionRate()
26 {
27     return commissionRate;
28 } // end method getCommissionRate
29
30 // set gross sales amount
31 public void setGrossSales( double sales )
32 {
33     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34 } // end method setGrossSales
35
36 // return gross sales amount
37 public double getGrossSales()
38 {
39     return grossSales;
40 } // end method getGrossSales
41
```

Validate and set the gross sales value

COMMISSION EMPLOYEE SUBCLASS

COMMISSION EMPLOYEE SUBCLASS

```
42 // calculate earnings; override abstract method earnings in Employee
43 public double earnings()
44 {
45     return getCommissionRate() * getGrossSales();
46 } // end method earnings
47
48 // return String representation of CommissionEmployee object
49 public String toString()
50 {
51     return String.format("%s: %s\n%s: $%,.2f; %s: %.2f",
52         "commission employee", super.toString(),
53         "gross sales", getGrossSales(),
54         "commission rate", getCommissionRate());
55 } // end method toString
56 } // end class CommissionEmployee
```

Override **earnings** method so **CommissionEmployee** can be concrete

Override **toString** method

Call superclass's **toString** method

BASE PLUS COMMISSION EMPLOYEE SUBCLASS

```
1 // BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee class definition
3
4 public class BasePlusCommissionEmployee extends CommissionEmployee
5 {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10         String ssn, double sales, double rate, double salary )
11    {
12        super( first, last, ssn, sales, rate ); ← Call superclass constructor
13        setBaseSalary( salary ); // validate and store base salary
14    } // end six-argument BasePlusCommissionEmployee constructor
15
16    // set base salary
17    public void setBaseSalary( double salary )
18    {
19        baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
20    } // end method setBaseSalary
```

Validate and set base salary value

BASE PLUS COMMISSION EMPLOYEE SUBCLASS

```
22 // return base salary
23 public double getBaseSalary()
24 {
25     return baseSalary;
26 } // end method getBaseSalary
27
28 // calculate earnings; override method earnings in CommissionEmployee
29 public double earnings() ← Override earnings method
30 {
31     return getBaseSalary() + super.earnings();
32 } // end method earnings
33
34 // return String representation of BasePlusCommissionEmployee object
35 public String toString() ← Call superclass's toString method
36 {
37     return String.format("%s %s; %s: $%,.2f",
38         "base-salaried", super.toString(),
39         "base salary", getBaseSalary());
40 } // end method toString
41 } // end class BasePlusCommissionEmployee
```

Call superclass's **toString** method

```
1 // PayrollSystemTest.java
2 // Employee hierarchy test program.
3
4 public class PayrollSystemTest
5 {
6     public static void main( String args[] )
7     {
8         // create subclass objects
9         SalariedEmployee salariedEmployee =
10            new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11         HourlyEmployee hourlyEmployee =
12            new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13         CommissionEmployee commissionEmployee =
14            new CommissionEmployee(
15                "Sue", "Jones", "333-33-3333", 10000, .06 );
16         BasePlusCommissionEmployee basePlusCommissionEmployee =
17            new BasePlusCommissionEmployee(
18                "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20         System.out.println( "Employees processed individually:\n" );
21     }
```

TEST CLASS

```

22     System.out.printf( "%s\n%s: $%,.2f\n\n",
23         salariedEmployee, "earned", salariedEmployee.earnings() );
24     System.out.printf( "%s\n%s: $%,.2f\n\n",
25         hourlyEmployee, "earned", hourlyEmployee.earnings() );
26     System.out.printf( "%s\n%s: $%,.2f\n\n",
27         commissionEmployee, "earned", commissionEmployee.earnings() );
28     System.out.printf( "%s\n%s: $%,.2f\n\n",
29         basePlusCommissionEmployee,
30         "earned", basePlusCommissionEmployee.earnings() );
31
32 // create four-element Employee array
33 Employee employees[] = new Employee[ 4 ];
34
35 // initialize array with Employees
36 employees[ 0 ] = salariedEmployee;
37 employees[ 1 ] = hourlyEmployee; ←
38 employees[ 2 ] = commissionEmployee;
39 employees[ 3 ] = basePlusCommissionEmployee;
40
41 System.out.println( "Employees processed polymorphically:\n" );
42
43 // generically process each element in array employees
44 for ( Employee currentEmployee : employees )
45 {
46     System.out.println( currentEmployee ); // invokes toString
47

```

Assigning subclass objects
to supercalss variables

Implicitly and polymorphically call `toString`

TEST CLASS

TEST CLASS

```
48 // determine whether element is a BasePlusCommissionEmployee  
49 if ( currentEmployee instanceof BasePlusCommissionEmployee )  
{  
    // downcast Employee reference to  
    // BasePlusCommissionEmployee reference  
    BasePlusCommissionEmployee employee =  
        ( BasePlusCommissionEmployee ) currentEmployee;  
    double oldBaseSalary = employee.getBaseSalary();  
    employee.setBaseSalary( 1.10 * oldBaseSalary );  
    System.out.printf(  
        "new base salary with 10% increase is: $%,.2f\n",  
        employee.getBaseSalary() );  
} // end if  
  
System.out.printf(  
    "earned $%,.2f\n\n", currentEmployee.earnings() );  
} // end for  
  
// get type name of each object in employees array  
for ( int j = 0; j < employees.length; j++ )  
    System.out.printf( "Employee %d is a %s\n", j,  
        employees[ j ].getClass().getName() );  
} // end main  
} // end class PayrollSystemTest
```

If the **currentEmployee** variable points to a **BasePlusCommissionEmployee** object

Downcast **currentEmployee** to a **BasePlusCommissionEmployee** reference

Give **BasePlusCommissionEmployees** a 10% base salary bonus

Polymorphically call **earnings** method

Call **getClass** and **getName** methods to display each **Employee** subclass object's class name

Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: \$800.00
earned: \$800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: \$16.75; hours worked: 40.00
earned: \$670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: \$10,000.00; commission rate: 0.06
earned: \$600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: \$5,000.00; commission rate: 0.04; base salary: \$300.00
earned: \$500.00

OUTPUT

OUTPUT

Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: \$800.00
earned \$800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: \$16.75; hours worked: 40.00
earned \$670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: \$10,000.00; commission rate: 0.06
earned \$600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: \$5,000.00; commission rate: 0.04; base salary: \$300.00
new base salary with 10% increase is: \$330.00
earned \$530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee

Same results as when the employees
were processed individually

Base salary is increased by 10%

Each employee's type is displayed

VOCABULARY I

- class – a description of a set of objects
- object – a member of a class
- instance – same as “object”
- field – data belong to an object or a class
- variable – a name used to refer to a data object
 - instance variable – a variable belonging to an object
 - class variable, static variable – a variable belonging to the class as a whole
 - method variable – a temporary variable used in a method

VOCABULARY II

- method – a block of code that can be used by other parts of the program
 - instance method – a method belonging to an object
 - class method, static method – a method belonging to the class as a whole
- constructor – a block of code used to create an object
- parameter – a piece of information given to a method or to a constructor
 - actual parameter – the value that is passed to the method or constructor
 - formal parameter – the name used by the method or constructor to refer to that value
- return value – the value (if any) returned by a method

VOCABULARY III

- hierarchy – a treelike arrangement of classes
- root – the topmost thing in a tree
- Object – the root of the class hierarchy
- subclass – a class that is beneath another in the class hierarchy
- superclass – a class that is above another in the class hierarchy
- inherit – to have the same data and methods as a superclass

HOMEWORK



Review class notes.



Additional reading:
Examples of UML diagrams



Start a discussion on Google Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





What is Refactoring?



Code Smells



Why Refactoring?



Techniques



IDEs

OUTLINE

WHAT IS REFACTORING?



“Art of improving the design of existing code”



Disciplined technique for restructuring an existing body of code.



Altering internal structure without changing external behavior of code.



Agile teams maintain and extend code a lot from iteration to iteration.



Without continuous refactoring code tends to rot (bad code smell).



A popular metaphor for refactoring is cleaning the kitchen as you cook.



In any kitchen you will typically find that cleaning and reorganizing occur continuously.



Someone is responsible for keeping the dishes, the pots, the kitchen itself, the food.



The refrigerator is cleaned and organized from moment to moment.



Without this, continuous cooking would soon collapse.

CODE HYGIENE



Code smell is any symptom in the code that possibly indicates a deeper problem



Code smells are usually not bugs or broken code



They don't currently prevent the program from functioning



They indicate weaknesses in design



Disaster waiting to happen



Slowing down development or increasing the risk of bugs or failures in the future

WHAT IS CODE SMELL?

```
public class BadCodeExample {  
    public static void main(String[] args) {  
        String fileName = "C:/WorkSpace/numbers.txt";  
        try {  
            Scanner input = new Scanner(new File(fileName));  
            ArrayList<Double> numbers = new ArrayList<Double>();  
            while(input.hasNext()) {  
                numbers.add(input.nextDouble());  
            }  
            System.out.println("Total: "+numbers);  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

EXAMPLE

— ADD NUMBERS



Example deals with files



What could go wrong if file doesn't exist?



Bad filename



File is locked while code executes



Someone else needs access to the file



Wait a minute ... we didn't close the file ...



Only one method that performs all steps

**WHAT BAD
SMELL DO YOU
NOTICE?**



Repeating Code:

duplicated code



Coding Standards:

too many parameters, long method, large class, naming conventions



Feature envy

a class that uses methods of another class excessively



Dependency:

a class that has dependencies on implementation details of another class.



Lazy class / Freeloader:

a class that does too little.

COMMON CODE SMELLS



Contrived complexity:

forced usage of overly complicated design patterns where simpler design would suffice.



Excessively long or short identifiers:

not following coding standards



Excessive use of literals:

these should be coded as named constants



Ubercallback:

a callback that is trying to do everything



Complex conditionals

Complex if-else blocks

COMMON CODE SMELLS

COMMON CAUSES OF CODE SMELLS

The habit of postponing code fixes

Insisting on a one-liner solution

Ignoring the warnings

Not my code / it's my code

Excessive use of design patterns without knowing the usage

Hard coded values

No unit testing

Not following standards



Refactoring is usually motivated by noticing a code smell



Once recognized, such problems can be addressed by refactoring the source code



Or transform code into a new form that behaves the same as before but that no longer “smells”

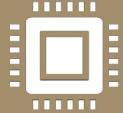


Failure to perform refactoring can result in accumulating technical debt

WHY REFACTORING?



Why fix what's not broken?



A software module

Should function its expected functionality
• It exists for this



It must be affordable to change

It will have to change over time, so it better be cost effective



Must be easier to understand

Developers unfamiliar with it must be able to read and understand it

BUT WHY?



Maintainability:

It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.



Extensibility:

It is easier to extend add new features



Reusability:

Reuse of some functionality without too much coding.



Quality:

Good quality code that doesn't smell.



Optimization

Maintain high standards and simplicity

BENEFITS



A solid set of automatic unit tests is needed



The tests are used to demonstrate that the behavior of the module is correct before the refactoring



If a test fails, then it's generally best to fix the test first



Understand the impact of refactoring, dependencies, and impact on other parts of the system

BEFORE WE REFACTOR



The tests are run again to verify the refactoring didn't break the tests



Of course, the tests can never prove that there are no bugs, but the important point is that this process can be cost-effective



Good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough



Do regression test of complete application to make sure other parts are still working.

AFTER REFACTORING



The process is an iterative cycle of making a small program transformation.



Testing it to ensure correctness, and making another small transformation.



If at any point a test fails, the last small change is undone and repeated in a different way.



Through many small steps the program moves from where it was to where you want it to be.



In order for this very iterative process to be practical, the tests have to run very fast.

SO HOW DO WE
DO IT?



Encapsulate Field

force code to access the field
with getter and setter methods



Generalize Type

create more general types to
allow for more code sharing



Replace type

check code with
State/Strategy



Simplify Conditions

replace conditional
with polymorphism

TECHNIQUES - MORE ABSTRACTION



Componentization

break code down into reusable semantic units



Extract Class

move parts of the code from an existing class into a new class



Extract Method

turn part of a larger method into a new method



Break down code

smaller code is more easily understandable

TECHNIQUES - MORE LOGICAL PIECES



Move Method or Move Field

move to a more appropriate class or method



Rename Method or Rename Field

changing the name into a new one that better reveals its purpose



Pull Up

in OOP, move to a superclass



Push Down

in OOP, move to a subclass

TECHNIQUES – MOVE CODE



Refactoring is much easier to do automatically than it is to do by hand



More Integrated Development Environments (IDEs) are building in automated refactoring support



Select the code you want to refactor, pull down the specific refactoring you need from a menu, and the IDE does the rest



You are prompted appropriately by dialog boxes for new names for things that need naming, and for similar input.



You can then immediately rerun your tests to make sure that the change didn't break anything.



If anything is broken, you can easily undo the refactoring and investigate

REFACTORING AUTOMATION IN IDES



Code review is systematic examination (often known as peer review) of source code.



It is intended to find and fix mistakes overlooked in the initial development phase.



It improves both the overall quality of software and the developers' skills.



Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.

CODE REVIEW

TYPES OF REVIEW



Pair programming

Two developers code together and review each others code and provide feedback



Formal code review

Involves a careful and detailed process with multiple participants and multiple phases



Lightweight code review

Conducted as part of the normal development process



Over-the-shoulder

One developer looks over the author's shoulder as the latter walks through the code.



Email pass-around

Email code to reviewers automatically after code is committed.



Tool-assisted code review

Authors and reviewers use specialized tools designed for peer code review.

LIGHTWEIGHT CODE REVIEW

```
public class CodeRefactoringExample {  
    String fileName = "C:/WorkSpace/numbers.txt";  
    Scanner input;  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        CodeRefactoringExample cre = new CodeRefactoringExample();  
        cre.openFile(cre.fileName);  
        try {  
            System.out.println("Total: "+cre.addNumbers());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }finally{  
            cre.closeFile();  
        }  
    }  
}
```

REVISITING THE EXAMPLE

```
public void openFile(String filename){  
    try {  
        input = new Scanner(new File(fileName));  
        System.out.println("File opened for processing!");  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
  
public double addNumbers(){  
    double total = 0d;  
    while(input.hasNext()){  
        total += input.nextDouble();  
    }  
    return total;  
}  
  
public void closeFile(){  
    if(input != null)  
        input.close();  
    System.out.println("File closed!");  
}
```

REVISITING THE EXAMPLE

To

- Anytime you can cleanup the code
- To make it readable, understandable, simpler
- You are convinced about the change
- Before adding a feature or fixing a bug
- After adding a feature or fixing a bug

Not to

- Not for the sake of refactoring
- When the change will affect too many things
- When change may render application unusable
- In the middle of adding a feature or fixing a bug
- You don't have unit tests to support your change

TO RE-FACTOR
OR NOT TO RE-
FACTOR?

HOMEWORK



Review class notes.



Additional reading:
Examples of UML diagrams



Start a discussion on Google Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





What are Design Patterns?



Why Design Patterns?



Example



Design Pattern Types

OUTLINE

TOOLKIT, FRAMEWORK, AND DESIGN PATTERN



A toolkit is a library of reusable classes designed to provide useful, general-purpose functionality

E.g., Java APIs (awt, util, io, net, etc)



A framework is a specific set of classes that cooperate closely with each other and together embody a reusable design for a category of problems

E.g., Struts, JSF, WCF, WPF, etc.



A design pattern describes a general recurring problem in different domains, a solution, when to apply the solution, and its consequences

E.g., Factory, Façade, Singleton etc.



A framework embodies a complete design of an application



A pattern is an outline of a solution to a class of problems



A framework dictates the architecture of an application and can be customized (e.g. Entity)



When one uses a framework, one reuses the main body of the framework and writes the code it calls.



When one uses a toolkit, one writes the main body of the application that calls the code in the toolkit.



Design patterns are integral parts of frameworks and toolkits

TOOLKIT, FRAMEWORK, AND DESIGN PATTERN

WHAT ARE DESIGN PATTERNS?

- 💡 A reusable solution for common occurring problems
- ✖ A description or template for how to solve a problem
- ✓ Formalized best practices to speed up the development process
- ⌚ Provide tested, proven development paradigms
- 👤 OOP/OOD compatible
- 📄 Documented in a platform independent format

WHY DESIGN PATTERNS?

Provides vocabulary to communicate, document, and explore design alternatives.

Captures the experience of an expert and codifies it in a form that is reusable.

Reusable solution to commonly recurring programming problems.

Represents the best programming practices adapted by experienced object-oriented software engineers.

WHY DESIGN PATTERNS?



Effective software design requires consideration of:

short term and long term issues
improved code readability
ease of implementation and reproducible results



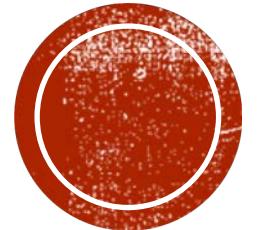
DP facilitates achieve reliable and flexible code



Patterns turn into components



Resolves known issues and can capture unknowns



EXAMPLES



THE INTERMEDIARY PATTERN



A client interacts with an intermediary



The requested services are carried out by the server/worker.

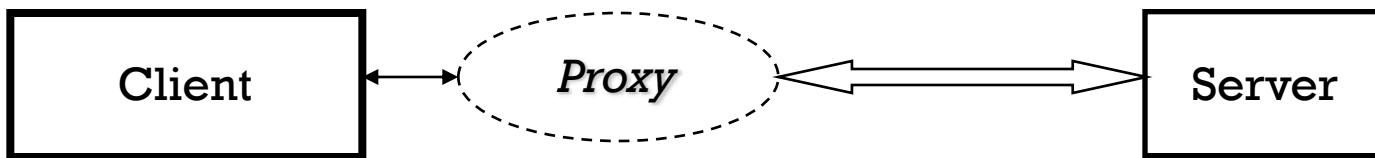
PROXY



Intermediary acts like a transmission agent



A *proxy*, in its most general form, is a class functioning as an interface to something else.



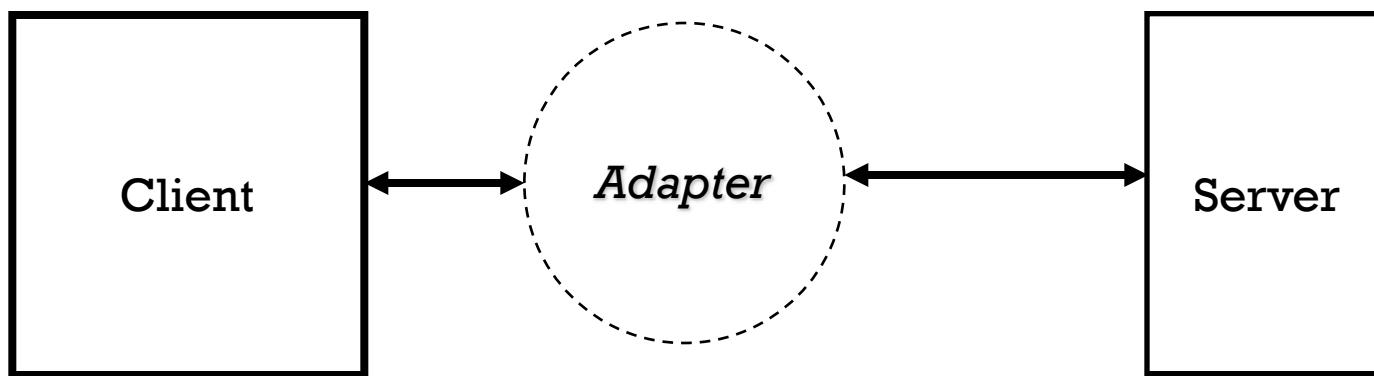
TRANSLATOR / ADAPTER



Intermediary acts like a translator between the client and the server.



E.g., Format/protocol conversions.



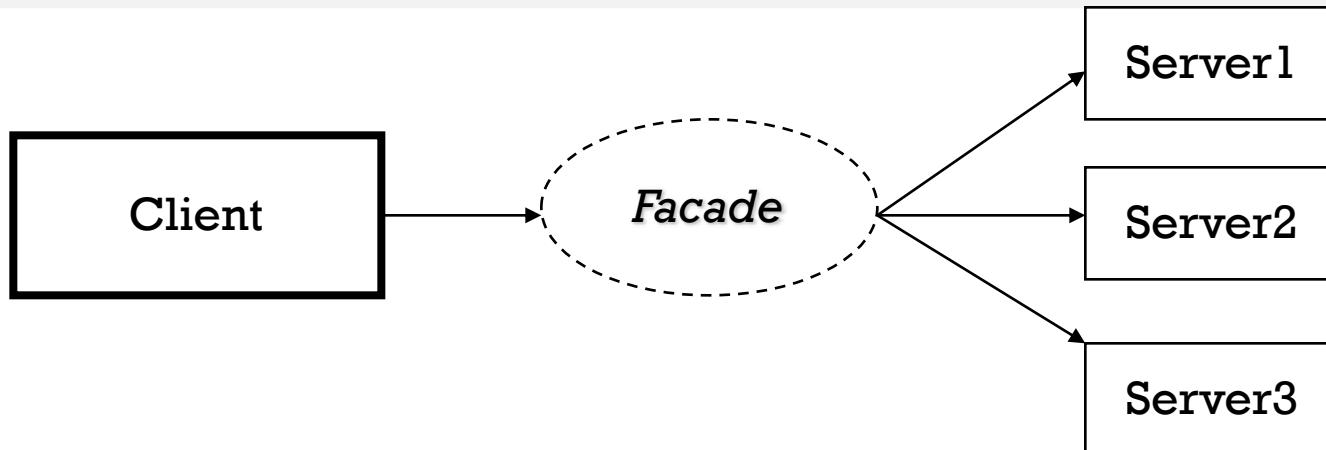
FACADE



Intermediary acts like a focal point distributing work to other agents.

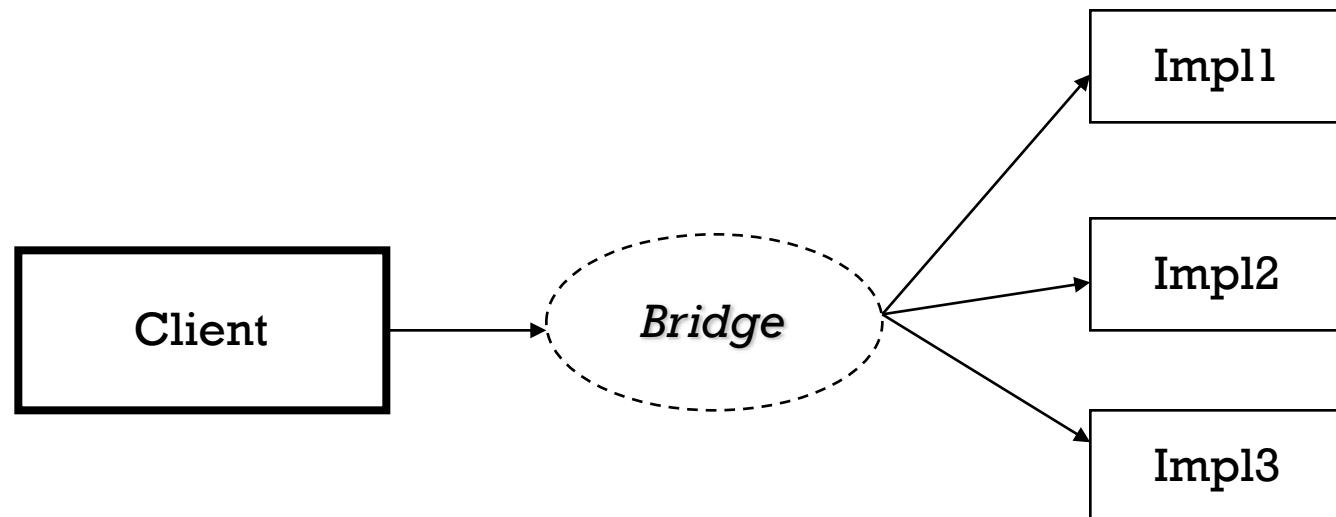


E.g. telnet, ftp, ... --> web-browser



BRIDGE/ABSTRACT FACTORY/HANDLE

- ❖ Intermediary defines the interface but not the implementation.
- ✓ E.g., Motif/Mac/Windows look and feel





Several sections defining:



a prototypical micro-architecture (classes and objects)



developers copy and adapt to their particular designs



solution to the recurrent problem described by the design pattern

DP STRUCTURE



Must explain why a particular situation causes problems



Why the proposed solution is considered a good one



Must define the boundaries and environments it is applicable in



Must be a general approach with options

PATTERNS – MUST HAVE



Based on the problem scope there are different types



Creational



Structural



Behavioral



Architectural

DP TYPES

CREATIONAL

Creates object for you, rather than having you instantiate objects directly.

More flexibility in deciding which objects need to be created for a given case.

Abstract Factory

- groups object factories that have a common theme.

Builder

- constructs complex objects by separating construction and representation.

Factory

- method creates objects without specifying the exact class to create.

Prototype

- creates objects by cloning an existing object.

Singleton

- restricts object creation for a class to only one instance.

STRUCTURAL

These concern class and object composition

Defines ways to compose objects to obtain new functionality

Adapter

- allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.

Bridge

- decouples an abstraction from its implementation so that the two can vary independently

Façade

- provides a simplified interface to a large body of code

Composite

- composes zero-or-more similar objects so that they can be manipulated as one object.

Flyweight

- reduces the cost of creating and manipulating similar objects

BEHAVIORAL

These concern how objects communicate with each other

Identifies common communication pattern

Chain of responsibility

- delegates commands to a chain of processing objects.

Command

- creates objects which encapsulate actions and parameters.

Interpreter

- implements a specialized language

Iterator

- accesses the elements of an object sequentially without exposing its underlying representation

State

- allows an object to alter its behavior when its internal state changes

ARCHITECTURAL

These address various issues in software engineering

Reusable solution to recurring problem in software architecture

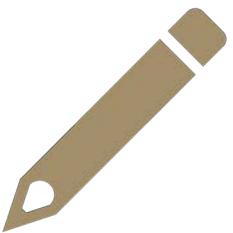
Application

- create the composite architecture scalable, reliable, available and manageable

Data

- rules or standards that govern which data is collected, and how it is stored, arranged

HOMEWORK



Review class notes.



Additional reading:
Examples of Design Patterns



Start a discussion on Google
Groups to clarify your doubts.

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





Most Used Design Patterns



Deep Dive



Examples

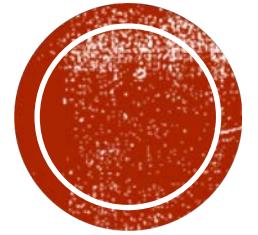


Where to go next?

OUTLINE

MOST IMPORTANT DESIGN PATTERNS

Pattern	Category	Notes
Singleton	Creational	limit creation of a class to only one object
Factory	Creational	objects are created by calling a factory method instead of a constructor
Builder	Creational	creating complex types can be simplified by using the builder pattern
Adapter	Structural	allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
Facade	Structural	create a simplified interface of an existing interface to ease usage for common tasks
Observer	Behavioral	when one object changes state, all its dependents are notified
Chain of Responsibility	Behavioral	delegates commands to a chain of processing objects



DEEP DIVE





This is the most used pattern



A lot of framework
already implement this
pattern, such as:

Spring (via `@ApplicationScoped`)
EJBs (using `@Singleton`)



Ensure a class has only one instance and provide a
global point to access it



The concept is sometimes generalized to systems
that operate more efficiently when only one object
exists



The term comes from the mathematical concept of
a singleton

SINGLETON

SINGLETON EXAMPLE

Problem: You need an object that needs to be instantiated once.

The class needs to declare a private constructor to prevent people to instantiate it from outside the class.

The method `getInstance()` assures that only one instance of this class is created at runtime.

```
public class SingletonExample {  
    private static SingletonExample instance = null;  
    private SingletonExample() {}  
    public static SingletonExample getInstance() {  
        if(instance == null) {  
            instance = new SingletonExample();  
        }  
        return instance;  
    }  
}
```

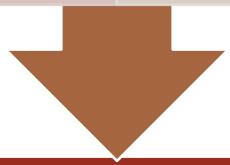
Problem: How can an object be created so that subclasses can redefine which class to instantiate?



The Factory design pattern describes how to solve such problems:

Define a separate operation (*factory method*) for creating an object.

Create an object by calling a *factory method*.



This enables writing of subclasses to change the way an object is created

FACTORY

FACTORY EXAMPLE

The MazeGame uses Rooms but it puts the responsibility of creating Rooms to its subclasses which create the concrete classes.

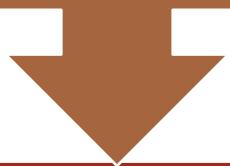
The regular game mode could use this template method.

```
public abstract class Room {  
    abstract void connect (Room room);  
}  
  
public class MagicRoom extends Room {  
    public void connect (Room room) {}  
}  
  
public class OrdinaryRoom extends Room {  
    public void connect (Room room) {}  
}  
  
public abstract class MazeGame {  
    private final List<Room> rooms = new ArrayList <> ();  
    Public MazeGame () {  
        Room room1 = makeRoom();  
        Room room2 = makeRoom();  
        room1.connect( room2 );  
        rooms.add( room1 );  
        rooms.add( room2 );  
    }  
    abstract protected Room makeRoom();  
}
```

Problem: Some objects require lots of parameters to be created



In this case, either using the constructor to create this object or using the setters will make code ugly and hard to understand



The builder pattern can help us in this case

The intent of the Builder design pattern is to separate the construction of a complex object from its representation

By doing so the same construction process can create different representations

BUILDER

BUILDER EXAMPLE

A product can have many types.

```
public class Product {  
    private String id;  
    private String name;  
    private String description;  
    private Double value;  
    private Product(Builder builder) {  
        setId(builder.id);  
        setName(builder.name);  
        setDescription(builder.description);  
        setValue(builder.value);  
    }  
    // Getter and Setter methods for all attributes  
    . . .  
}
```

BUILDER EXAMPLE

Builder class builds the product

```
public static final class Builder {  
    private String id;  
    private String name;  
    private String description;  
    private Double value;  
    private Builder() {}  
    public Builder id(String id) { this.id = id; return this; }  
    public Builder name(String name) { this.name = name; return this; }  
    public Builder description(String description) {  
        this.description = description; return this;  
    }  
    public Builder value(Double value) { this.value = value; return this; }  
    public Product build() { return new Product(this); }  
}
```

Problem: Often an existing class can't be reused because its interface doesn't conform to the interface clients require



The key idea is to work through a separate adapter that adapts the interface of an existing class without changing it



The adapter design pattern describes how to solve such problems:

Define a separate adapter class that converts the (incompatible) interface of a class (adaptee) into another interface (target) clients require.

Work through an adapter to work with (reuse) classes that do not have the required interface.

ADAPTER

ADAPTER EXAMPLE

Charging phones

```
interface LightningPhone {
    void recharge();
    void useLightning();
}
interface MicroUsbPhone {
    void recharge();
    void useMicroUsb();
}
class Iphone implements LightningPhone {
    private boolean connector;
    @Override
    public void useLightning() {
        connector = true;
    }
    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}
. . .
```

ADAPTER EXAMPLE . . .

Charging phones

```
class Android implements MicroUsbPhone {
    private boolean connector;

    @Override
    public void useMicroUsb() {
        connector = true;
        System.out.println("MicroUsb connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
        } else {
            System.out.println("Connect MicroUsb first");
        }
    }
}
. . .
```

ADAPTER EXAMPLE . . .

Charging phones

```
class LightningToMicroUsbAdapter implements MicroUsbPhone {
    private final LightningPhone lightningPhone;

    public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}
```

ADAPTER EXAMPLE . . .

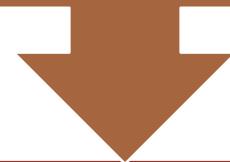
Charging phones

```
public class AdapterDemo {  
    static void rechargeMicroUsbPhone(MicroUsbPhone phone) {  
        phone.useMicroUsb();  
        phone.recharge();  
    }  
    static void rechargeLightningPhone(LightningPhone phone) {  
        phone.useLightning();  
        phone.recharge();  
    }  
    public static void main(String[] args) {  
        Android android = new Android();  
        Iphone iPhone = new Iphone();  
  
        System.out.println("Recharging android with MicroUsb");  
        rechargeMicroUsbPhone(android);  
        System.out.println("Recharging iPhone with Lightning");  
        rechargeLightningPhone(iPhone);  
        System.out.println("Recharging iPhone with MicroUsb");  
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iPhone));  
    }  
}
```

Problem: Clients that access a complex subsystem directly refer to many different objects having different interfaces, which makes the clients hard to implement, change, test, and reuse



Facade enables to work through an object to minimize the dependencies on a subsystem



The Facade design pattern describes how to solve such problems:

implements a simple interface in terms of (by delegating to) the interfaces in the subsystem

may perform additional functionality before/after forwarding a request

FACADE

FACADE EXAMPLE

How a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive)

```
/* Complex parts */

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

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}
. . .
```

FACADE EXAMPLE . . .

How a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive)

```
/* Facade */
class ComputerFacade {
    private final CPU processor;
    private final Memory ram;
    private final HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

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

/* Client */
class You {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

Problem: Tightly coupled objects are hard to implement, change, test, and reuse because they refer to many different objects with different interfaces



Observer solves following problems:

A one-to-many dependency between objects should be defined without making the objects tightly coupled.

It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.

It should be possible that one object can notify an open-ended number of other objects



The Observer design pattern describes how to solve such problems:

Define Subject and Observer objects

When a subject changes state, all registered observers are notified and updated automatically

OBSERVER

OBSERVER EXAMPLE

This example takes keyboard input and treats each input line as an event. When a string is supplied from System.in, the method notifyObservers is called, that notifies all observers of the event's occurrence, in the form of an invocation of their 'update' methods.

```
class EventSource {
    public interface Observer {
        void update(String event);
    }
    private final List<Observer> observers = new ArrayList<>();
    private void notifyObservers(String event) {
        observers.forEach(observer -> observer.update(event));
    }
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    public void scanSystemIn() {
        var scanner = new Scanner(System.in);
        while (scanner.hasNextLine()) {
            var line = scanner.nextLine();
            notifyObservers(line);
        }
    }
}
public class ObserverDemo {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        var eventSource = new EventSource();
        eventSource.addObserver(event -> {
            System.out.println("Received response: " + event);
        });
        eventSource.scanSystemIn();
    }
}
```

Problem: Implementing a request directly within the class that sends the request is inflexible because it couples the class to a particular receiver and makes it impossible to support multiple receivers.



Chain of Responsibility solves following problems:

Coupling the sender of a request to its receiver should be avoided.

In addition, it should be possible that more than one receiver can handle a request



The Chain of Responsibility design pattern describes how to solve such problems:

Enable to send a request to a chain of receivers without having to know which one handles the request

The request gets passed along the chain until a receiver handles the request. The sender of a request is no longer coupled to a particular receiver

CHAIN OF RESPONSIBILITY

CHAIN OF RESPONSIBILITY EXAMPLE

A logger is created using a chain of loggers, each one configured with different log levels.

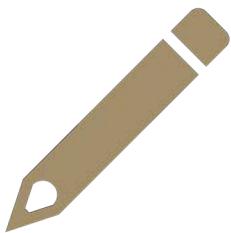
```
public interface Logger {  
    public enum LogLevel {  
        INFO, DEBUG, WARNING, ERROR, FUNCTIONAL_MESSAGE, FUNCTIONAL_ERROR;  
        public static LogLevel[] all() { return values(); }  
    }  
    abstract void message(String msg, LogLevel severity);  
    default Logger appendNext(Logger nextLogger) {  
        return (msg, severity) -> {  
            message(msg, severity);  
            nextLogger.message(msg, severity);  
        }  
    }  
    static Logger logger(LogLevel[] levels, Consumer<String> writeMessage) {  
        EnumSet<LogLevel> set = EnumSet.copyOf(Arrays.asList(levels));  
        return (msg, severity) -> {  
            if (set.contains(severity)) { writeMessage.accept(msg); }  
        };  
    }  
    static Logger consoleLogger(LogLevel... levels) {  
        return logger(levels, msg -> System.out.println("Writing to console: " + msg));  
    }  
    static Logger emailLogger(LogLevel... levels) {  
        return logger(levels, msg -> System.out.println("Sending via email: " + msg));  
    }  
    static Logger fileLogger(LogLevel... levels) {  
        return logger(levels, msg -> System.out.println("Writing to Log File: " + msg));  
    }  
}
```

CHAIN OF RESPONSIBILITY EXAMPLE

A logger is created using a chain of loggers, each one configured with different log levels.

```
public class ChainOfResponsibilityDemo {  
    public static void main(String[] args) {  
        // Build an immutable chain of responsibility  
        Logger logger = consoleLogger(LogLevel.all())  
            .appendNext(emailLogger(LogLevel.FUNCTIONAL_MESSAGE, LogLevel.FUNCTIONAL_ERROR))  
            .appendNext(fileLogger(LogLevel.WARNING, LogLevel.ERROR));  
  
        // Handled by consoleLogger since the console has a loglevel of all  
        logger.message("Entering function ProcessOrder()", LogLevel.DEBUG);  
        logger.message("Order record retrieved.", LogLevel.INFO);  
  
        // Handled by consoleLogger and fileLogger since filelogger implements Warning & Error  
        logger.message("Customer Address details missing in Branch DataBase.", LogLevel.WARNING);  
        logger.message("Customer Address details missing in Organization DataBase.", LogLevel.ERROR);  
  
        // Handled by consoleLogger and emailLogger as it implements functional error  
        logger.message("Unable to Process Order ORD1 Dated D1 For Customer C1.", LogLevel.FUNCTIONAL_ERROR);  
  
        // Handled by consoleLogger and emailLogger  
        logger.message("Order Dispatched.", LogLevel.FUNCTIONAL_MESSAGE);  
    }  
}
```

HOMEWORK



Review class notes.



Additional reading:
Examples of Design Patterns

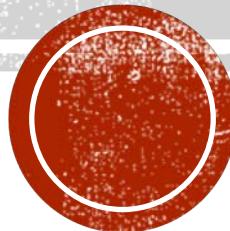


Start a discussion on Google
Groups to clarify your doubts.

SOFTWARE DESIGN

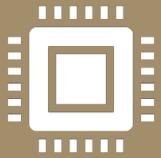
COSC 4353/6353

Dr. Raj Singh





An Architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.



Often documented as software design patterns.



The concept of an architectural pattern has a broader scope than the concept of design pattern.

ARCHITECTURAL PATTERNS

AP - FOCUS



SOFTWARE AND
HARDWARE
LIMITATIONS



PERFORMANCE



HIGH
AVAILABILITY



MINIMIZATION
OF A BUSINESS
RISK



SECURITY



DATA ACCESS
AND MODELING



DPs are usually associated with code level commonalities while APs are seen as commonality at higher level.



DPs provide various schemes for refining and building smaller subsystems. APs cover the fundamental organization of the system.



DPs are usually influenced by programming language whereas APs are influenced by the system.



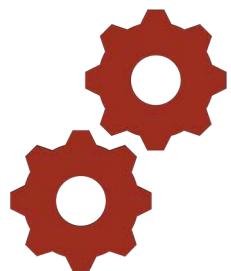
DPs are medium-scale tactics that flesh out some of the structure and behavior of entities and their relationships.



APs are high-level strategies that concerns large-scale components, the global properties and mechanisms of a system.

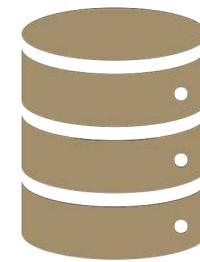
ARCHITECTURE PATTERNS VS. DESIGN PATTERNS

TYPES



Application Architecture Patterns

Science and art of ensuring the suite of applications being used by an organization to create the composite architecture is scalable, reliable, available and manageable.



Data Architecture Patterns

Composed of models, policies, rules or standards that govern which data is collected, and how it is stored, arranged, integrated, and put to use in data systems and in organizations



Multitier / n-tier Architecture



Model-View-Controller



Authentication Patterns



Authorization Patterns

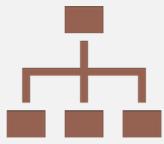
APPLICATION PATTERNS



N-tier application architecture provides a model by which developers can create flexible and reusable applications.



By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application.



A Three-tier architecture is typically composed of a presentation tier, a business or data access tier, and a data tier

MULTITIER/N-TIER ARCHITECTURE

THREE-TIER ARCHITECTURE

Presentation tier

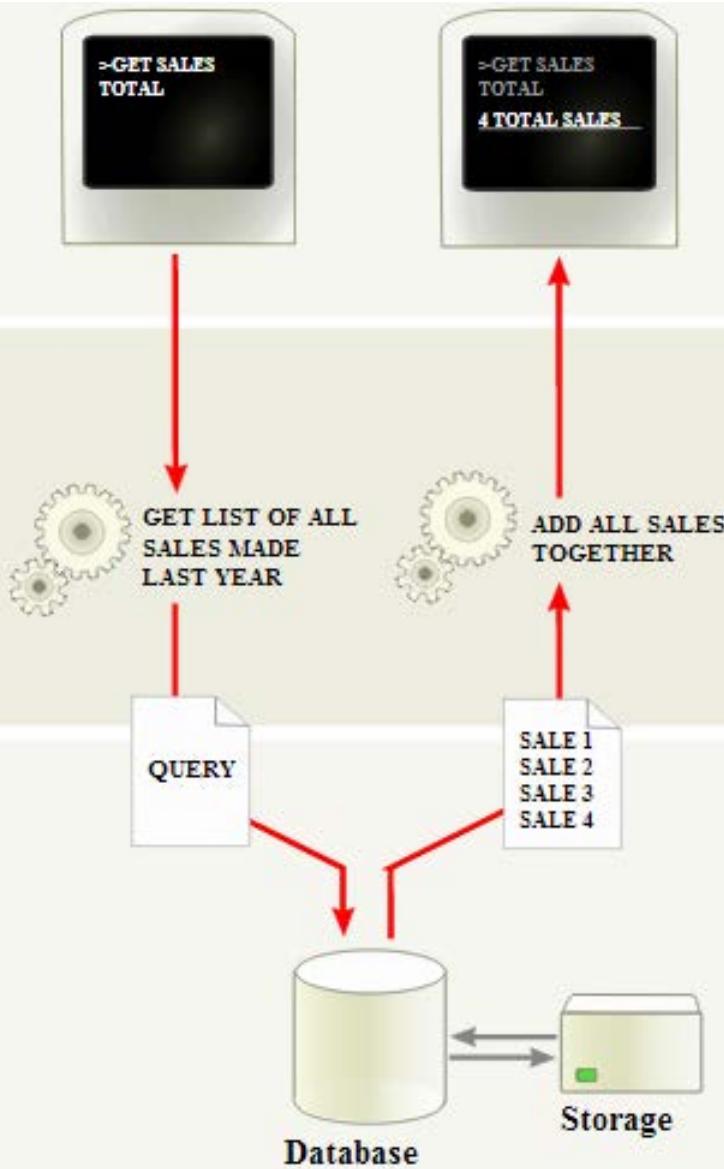
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

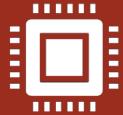
Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

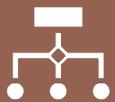




Model–view–controller (MVC) is a software architecture pattern which separates the representation of information from the user's interaction with it.



The model consists of application data, business rules, logic, and functions.



A view can be any output representation of data, such as a chart or a diagram. Multiple views of the same data are possible, such as a bar chart for management and a tabular view for accountants.



The controller mediates input, converting it to commands for the model or view

MVC

MVC



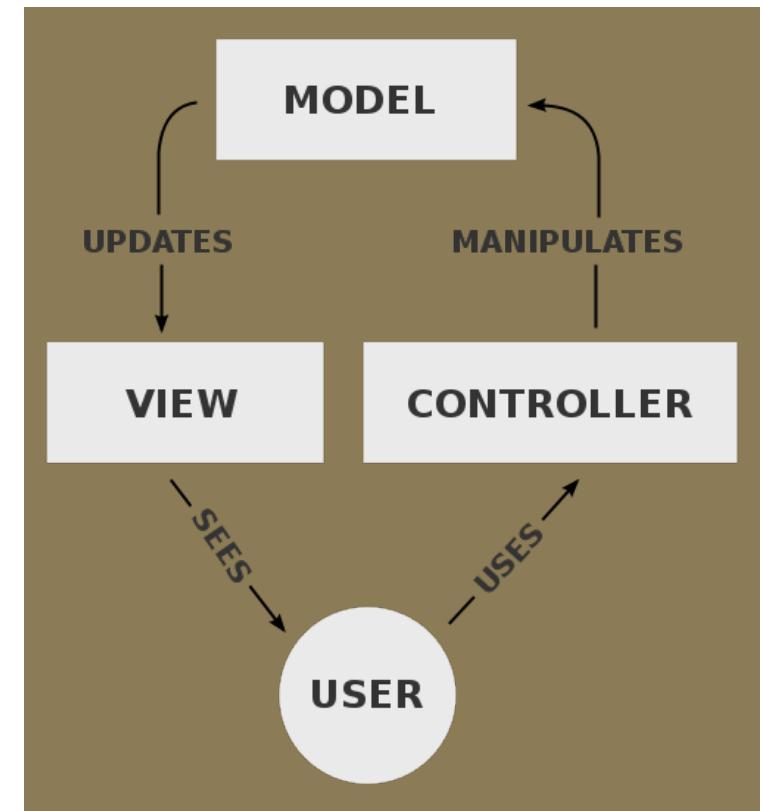
A **controller** can send commands to the model to update the model's state. It can also send commands to its associated view to change the view's presentation of the model



A **model** notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands.



A **view** requests information from the model that it needs for generating an output representation to the user.





At first glance, the three tiers may seem similar to MVC but fundamentally they are different.



The client tier never communicates directly with the data tier in a three-tier model.



All communication must pass through the middle tier. Conceptually the three-tier architecture is linear.



However, the MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

MULTITIER VS. MVC

AUTHENTICATION PATTERNS



Authentication is the process of identifying an individual using the credentials of that individual.



As computer systems have increased in complexity, the challenge of authenticating users has also increased.



There are a variety of models for authentication

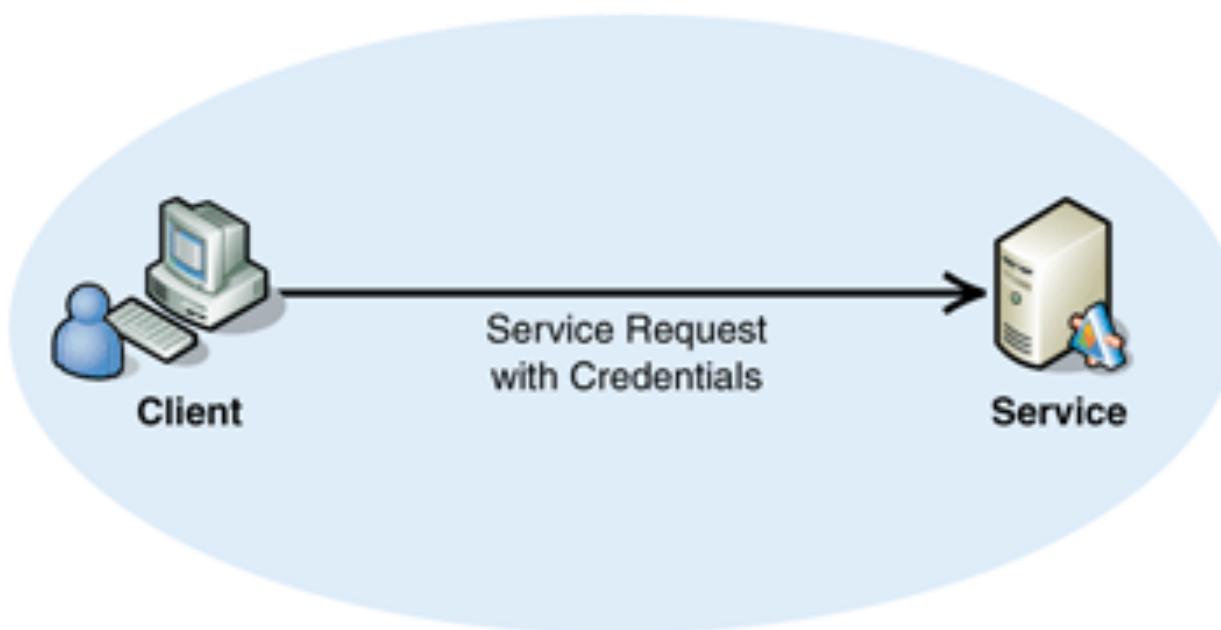
E.g., clients accessing a Web application may directly provide credentials for authentication. However, a third-party broker, such as a Kerberos domain controller, may be used to provide a security token for authentication.



Both the client and service participate in a trust relationship.



It allows them to exchange and validate credentials including passwords.



DIRECT AUTHENTICATION



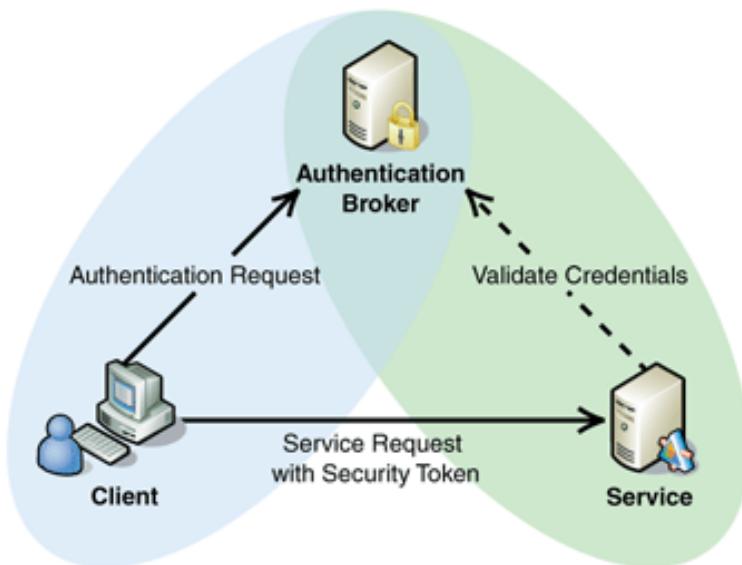
The broker authenticates the client and then issues a security token that the service can use to authenticate the client.



The security token is always verified, but the service does not need to interact with the broker to perform the verification.



This is because the token itself can contain proof of a relationship with the broker, which can be used by the service to verify the token



BROKERED AUTHENTICATION



Authorization is the process of determining whether an authenticated client is allowed to access a resource or perform a task within a security domain.



Authorization uses information about a client's identity and/or roles to determine the resources or tasks that a client can perform.

AUTHORIZATION PATTERNS

TYPES



Role-based Authorization

controlling which users have access to resources based on the role of the user



Resource-based Authorization

performed declaratively on a resource, depending on the type of the resource and the mechanism used to perform authorization



Activity-based Authorization

Who can do what, e.g., CRUD



Time-based Authorization

Time controlled access to a system or subsystem



Master Data Patterns



Business Intelligence

Data Mart
Data Warehouse



Data Integration

Extract Transform
Load (ETL)
Service Oriented
Architecture



Data Storage

Transactional Data
Stores
Operational Data Store

DATA ARCHITECTURE PATTERNS



The collective application of governance, business processes, policies, standards and tools facilitate consistency in master data



reference / basic business data used in a single application, system or process



a single source of reference data used across multiple systems, applications, and/or processes



is the single source of reference data used across all systems, applications, and processes



is the single source of basic business data for an entire marketplace

MASTER DATA PATTERNS

BUSINESS INTELLIGENCE

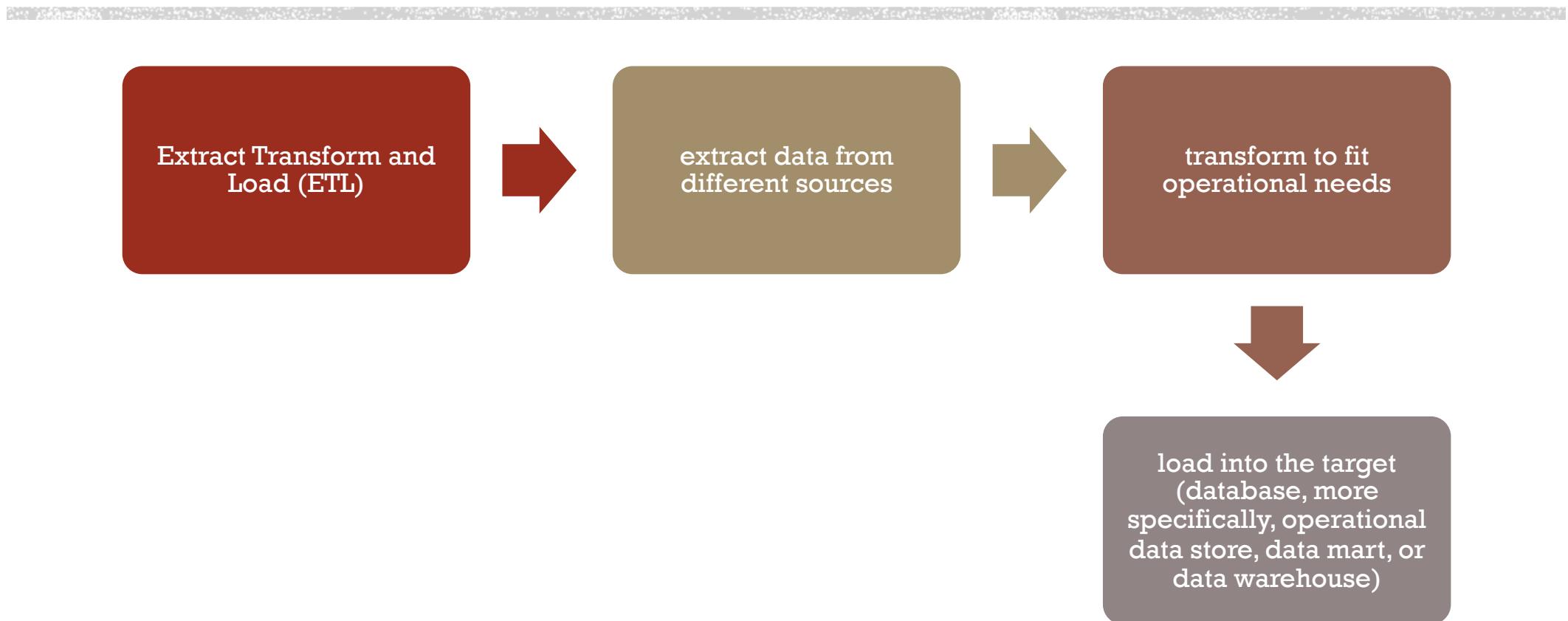
Data Warehouse

- is a database used for reporting and data analysis
- central repository of data which is created by integrating data from one or more disparate sources
- store current as well as historical data and are used for creating trending reports

Data Mart

- Access layer of the data warehouse environment that is used to get data out to the users.
- Easy access to frequently needed data
- Lower cost than implementing a full data warehouse
- Contains only business essential data and is less cluttered

DATA INTEGRATION



DATA STORAGE

Transactional Data Store

Transaction data are data describing an event (the change as a result of a transaction).

Transaction data always has a time dimension, a numerical value and refers to one or more objects (i.e. the reference data).



Database designed to integrate data from multiple sources for additional operations on the data.



The integration often involves cleaning, resolving redundancy and checking against business rules for integrity.

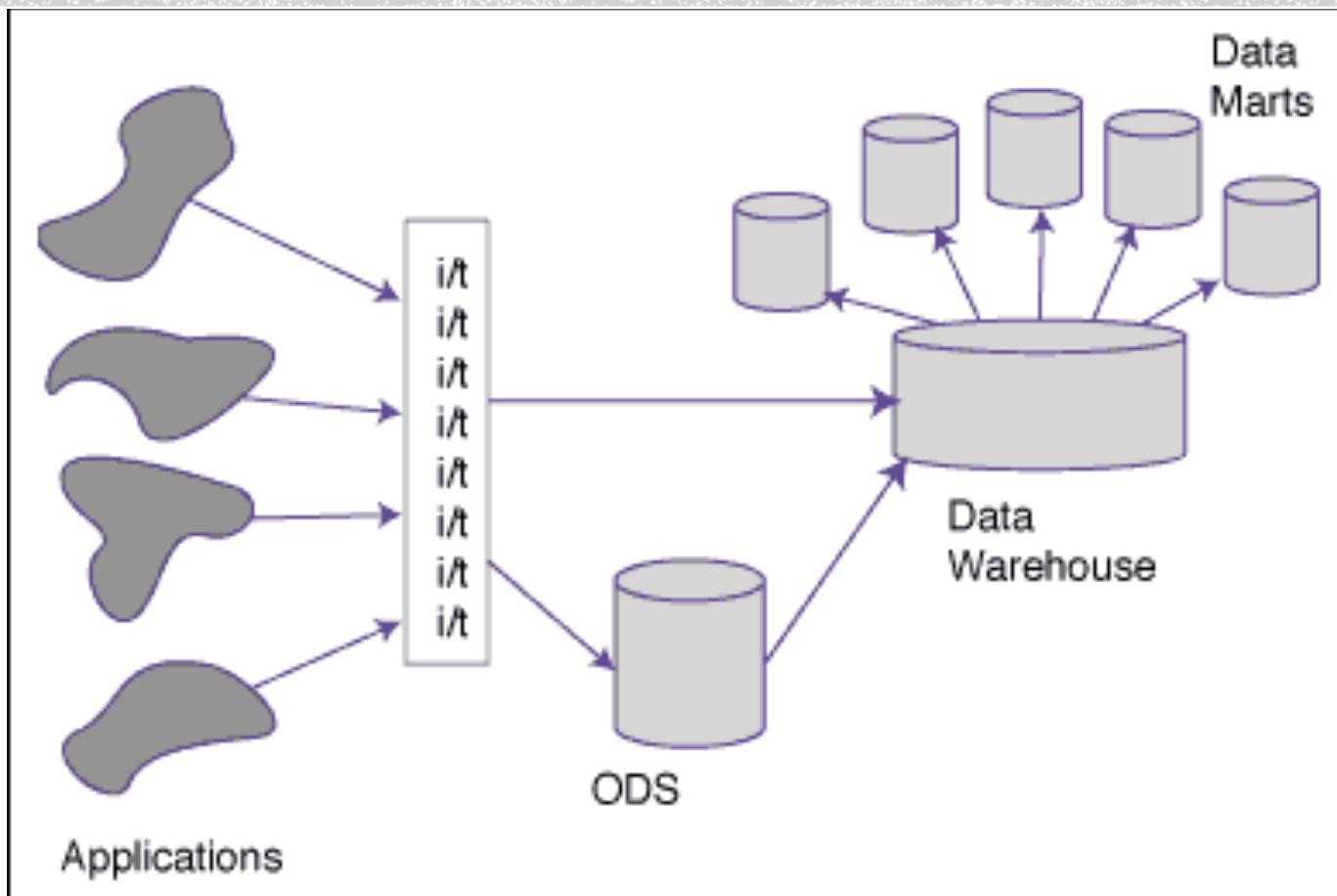


An ODS involves:

Extracting data from operational systems;
Moving it into ODS structures; and
Reorganizing and structuring the data for analysis purposes

OPERATIONAL DATA STORE (ODS)

ODS



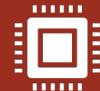
DW VS. ODS

Data Warehouse

- Is intended to support strategic planning and business intelligence decision support and should contain:
 - Integrated subject oriented data, e.g. sales data
 - Static data and historical data
 - Aggregated or summarized data

Operational Data Store :

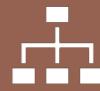
- Is intended to support operational management and monitoring and should contain:
 - Integrated subject oriented data (similar to data warehouses) e.g. sales data
 - Volatile data, will probably change frequently
 - Current detailed data



Discrete pieces of software providing application functionality as services to other applications.



Independent of any vendor, product or technology.



Services are unassociated, loosely coupled units of functionality that are self-contained.



Each service implements one action, such as

submitting an online order
retrieving an online bank statement
modifying an online order



Within a SOA, services use defined protocols that describe how services pass and parse messages using description metadata.

SERVICE-ORIENTED ARCHITECTURE (SOA)

SOA



Extensive use of XML in SOA to structure data



Web Services Description Language (WSDL) typically describes the services themselves



SOAP protocol describes the communication protocol



SOA depends on data and services that are described by metadata that should meet the following two criteria:

software systems can use metadata to configure dynamically by discovery and incorporation of defined services.
system designers can understand and manage with a reasonable expenditure of cost and effort.



Allows simultaneous use and easy mutual data exchange between programs of different vendors without additional programming



These services are also reusable, resulting in lower development and maintenance costs



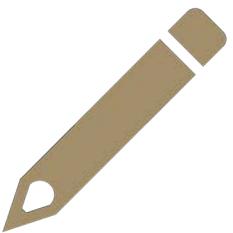
Providing more value once the service is developed and tested



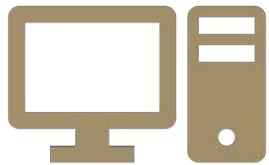
Having reusable services readily available also results in quicker time to market

SOA - BENEFITS

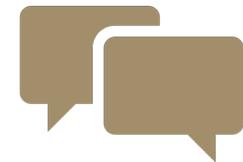
HOMEWORK



Review class notes and
previous recordings



Additional reading:
Examples of Design Patterns

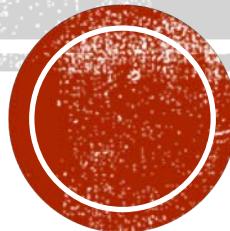


Start a discussion on Google
Groups to clarify your doubts

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh



SOFTWARE QUALITY

Functional Quality

- How well it complies with or conforms to a given design, based on functional requirements or specifications.

Structural Quality

- Refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

WHY CARE ABOUT CODE QUALITY?



You can't be Agile if your Code sucks



Customers prefer quality software products



A product of poor quality is very difficult to maintain



Its just like a ceramic plate that must not drop to the ground



Updates to bad code is a nightmare which if not handled carefully can break the whole product



Software quality drives predictability.



Do it once and do it right



There will be less re-work, less variation in productivity and better performance overall



Products get delivered on time, and they get built more productively



Poor quality is much more difficult to manage

QUALITY IMPACTS BUSINESS



Some companies have a reputation for building quality software.



A good, solid reputation is hard to establish and easy to lose, but when your company has it, it's a powerful business driver.



A few mistakes and that reputation can be gone, creating major obstacles to sales, and consequently, your bottom line.

QUALITY IMPACTS REPUTATION



The most productive and happy employees have pride in their work.



Enabling employees to build quality software will drive a much higher level of morale and productivity.



On the other hand, poor products, lots of re-work, unhappy customers and difficulty making deadlines have the opposite effect, leading to expensive turnover and a less productive workforce.

QUALITY IMPACTS EMPLOYEE



A quality product satisfies the customer.



A satisfied customer comes back for more and provides positive referrals.



Customer loyalty is heavily driven by the quality of the software you produce and service you provide.



Explosion of social media channels such as Twitter and Facebook, positive referrals can spread quickly.



Poor quality and dissatisfaction can also be communicated quickly, if not even quicker than the good ones.

QUALITY IMPACTS CUSTOMERS



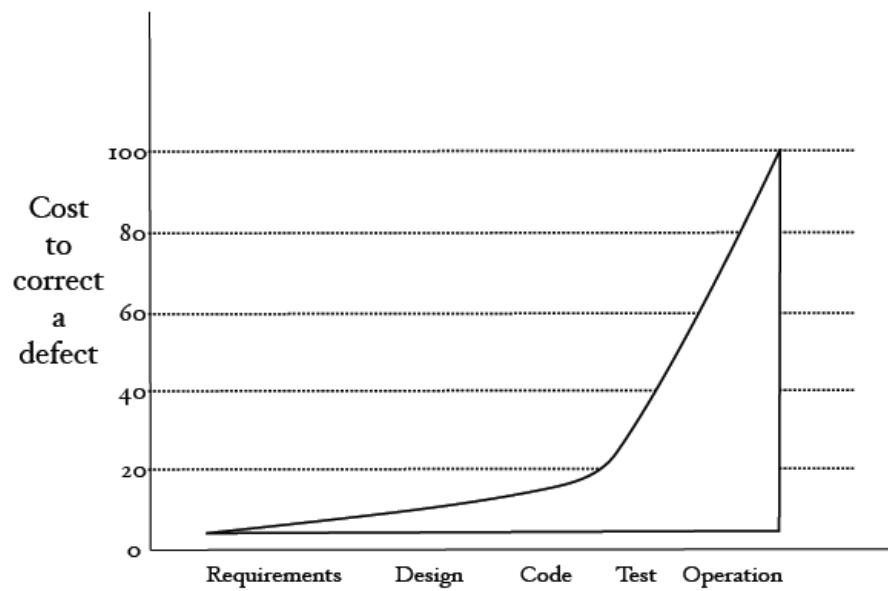
It's less costly to fix a defect if it's discovered early in the project



As project moves forward cost increases



The graph shows the cost impact of fixing defects



COST OF DEFECTS

SOFTWARE DEFECT REDUCTION TOP 10 LIST

- 👤 Finding, fixing problem in production is 100 times more expensive than during requirements/design phase.
- ✖ 40-50% of effort on projects is on avoidable rework.
- ➡ ~80% of avoidable rework comes from 20% of defects.
- 👉 ~80% of defects come from 20% of modules; about half the modules are defect free.
- ⌚ ~90% of downtime comes from at most 10% of defects.
- 🌐 Peer reviews catch 60% of defects.
- ❗ Perspective-based reviews catch 35% more defects than non-directed reviews.
- 💻 Disciplined personal practices can *reduce defect introduction rates* by up to 75%.
- ⌚ It costs 50% more per source instruction to develop high-dependability software product.
- 👤 ~40-50% of user programs have nontrivial defects.



Can't QA take care of quality, why should developers care?



QA shouldn't care about quality of design and implementation

”

They should care about acceptance, performance, usage, and relevance of the application



Give them a better quality software so they can really focus on that

WHY SHOULD I CARE, THERE'S QA?

PAY YOUR TECHNICAL DEBT



Technical debt are activities like

refactoring, upgrading a library, conforming to some UI or coding standard



These will hamper your progress if left undone for a longer time



You'll be more productive if quality is better



Some quality attributes are objective, and can be measured accordingly.



Some are subjective, and are therefore captured with more arbitrary measurements.

MEASURING QUALITY



Quality attributes can be external or internal.



External: Derived from the relationship between the environment and the system (or the process). (To derive, the system or process must run)

e.g. Reliability,
Robustness



Internal: Derived immediately from the product or process description (To derive, it is sufficient to have the description)

Underlying assumption: internal quality leads to external quality (cfr. metaphor manufacturing lines)
e.g. Efficiency

QUALITY ATTRIBUTES

CORRECTNESS, RELIABILITY, ROBUSTNESS

Correctness

A system is correct if it behaves according to its specification

An absolute property (i.e., a system cannot be “almost correct”)

Reliability

The user may rely on the system behaving properly

Reliability is the probability that the system will operate as expected over a specified interval

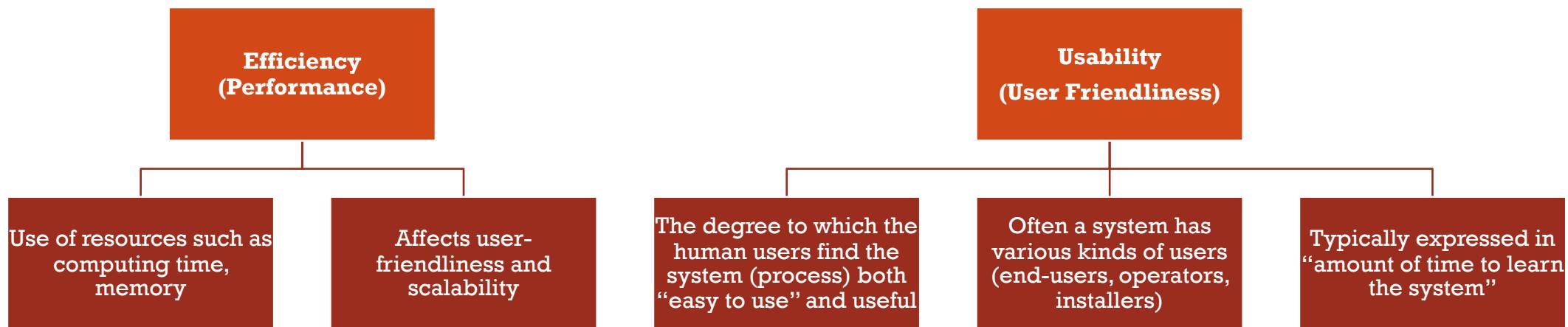
A relative property (a system has a mean time between failure of 3 weeks)

Robustness

A system is robust if it behaves reasonably even in circumstances that were not specified

A vague property (once you specify the abnormal circumstances they become part of the requirements)

EFFICIENCY, USABILITY





How easy it is to change a system after its initial release

software entropy ⇒ maintainability gradually decreases over time



Repairability

How much work is needed to correct a defect



Evolvability (Adaptability)

How much work is needed to adapt to changing requirements (both system and process)



Portability

How much work is needed to port to new environment or platforms

MAINTAINABILITY

Verifiability:

How easy it is to verify whether desired attributes are there?

- internally: verify requirements, code inspections
- externally: testing, efficiency

Understandability:

How easy it is to understand the system?

- internally: contributes to maintainability
- externally: contributes to usability

**VERIFIABILITY,
UNDERSTANDABILITY**

Productivity

- Amount of product produced by a process for a given number of resources
- productivity among individuals varies a lot

Timeliness

- Ability to deliver the product on time
- often a reason to sacrifice other quality attributes

Visibility (Transparency)

- Current process steps and project status are accessible
- important for management

**PRODUCTIVITY,
TIMELINESS,
VISIBILITY**



Start early



Don't Compromise



Schedule time to lower your technical debt



Make it work; make it right)right away*



Requires monitoring and changing behavior



Be willing to help and be helped



Devise lightweight non-bureaucratic measures

WAYS TO IMPROVE QUALITY

INDIVIDUAL EFFORTS



What can you do?



Care about design of your code



Keep it Simple



Write tests with high coverage



Run all your tests before checkin



Learn your language



Court feedback and criticism



Avoid shortcuts



Take collective ownership
team should own the code



Promote positive interaction



Provide constructive feedback



Constant code review

TEAM EFFORTS



Code review is by far the proven way to reduce code defects and improve code quality



Code review does not work if it's not done right.



Do you get together as a team, project code, and review?



Don't make it an emotionally draining

CODE REVIEW

 We've used code review effectively

 Code reviewed by one developer right after task is complete
(or anytime before)

 Rotate reviewer for each review

 Say positive things, what you really like

 Constructively propose changes

 Instead of “that’s lousy long method” say, “why don’t you split that method...”

 Review not only code, but also tests

 Do not get picky on style, instead focus on correctness, readability, and design.

SEEKING AND RECEIVING FEEDBACK



Rigorous inspection can remove up to 90 percent of errors before the first test case is run.



Reviews are both technical and sociological, and both factors must be accommodated.



Code review makes me smarter.



I learn ways to improve my code by looking at somebody's code.

VALUE OF REVIEW



Cohesion – single responsibility principle



Extensibility and Flexibility – OOP



Triangulation – generalization



Cost of Change – code that does many things is hard to maintain



Code Coverage – how much code needs testing



Complexity – large classes and methods are hard to maintain



Code Size – too much or too little



Code Duplication – why are you repeating?

CODE QUALITY FACTORS



Analyzing code to find bugs



Look for logic errors, coding guidelines violations, synchronization problems, data flow analysis, ...



IDEs

Automated tools



Other tools:

[Java] PMD, FindBugs, JLint
[.NET] VS, FxCop, ...
[C++] VS, Lint, ...

CODE ANALYSIS



It's a feeling or sense that something is not right in the code



You can't understand it



Hard to explain



Does some magic

CODE SMELL

✓ Duplication

✗ Unnecessary complexity

⌚ Useless/misleading comments

⚠ Long classes, methods, poor naming

||||| Code that's not used

🚫 Improper use of OOP

🔒 Tight coupling

🧠 Design Pattern overuse

COMMON CODE SMELLS



Deal with code smells

Refactor frequently



Don't rush

take time to write tests



Commenting and self documenting code

Just enough comments. Don't write stories.



Capture errors

Surround code that might be troublesome with try catch blocks

THINGS TO REMEMBER FOR QUALITY CODE



Practice tactical peer code review



Consider untested code is unfinished code



Make your code coverage and metrics visible



Use tools to check code quality



Treat warnings as errors



Keep it small and simple

THINGS TO REMEMBER FOR QUALITY CODE



Review class notes.



Additional reading:
Why quality matters?



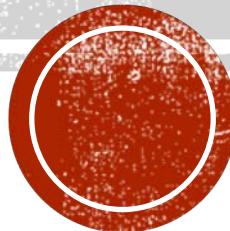
Start a discussion on Google
Groups to clarify your doubts.

HOMEWORK

SOFTWARE DESIGN

COSC 4353/6353

Dr. Raj Singh





What is SOA?



Why SOA?



SOA and Java



Different layers of SOA



REST



Microservices

OUTLINE

WHAT IS SOA?



Service Oriented Architecture (SOA)



An architectural style of building software applications that promotes loose coupling between components for reusability.



Services are software components that have published contracts/interfaces; these contracts are platform-, language-, and operating-system-independent.



XML and the Simple Object Access Protocol (SOAP) are the enabling technologies for SOA, since they're platform-independent standards.

Consumers can dynamically discover services.
Services are interoperable



The basic building block of SOA is the service.



A service is a self-contained software module that performs a predetermined task.



Services are platform and technology independent.



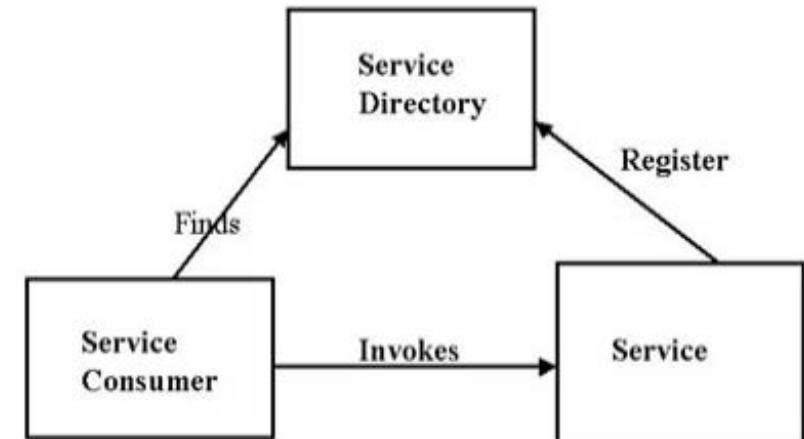
As developers, we tend to focus on reusing code; thus, we tend to tightly integrate the logic of objects or components within an application.

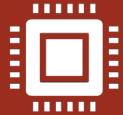


SOA promotes application assembly because services can be reused by numerous consumers.

For example: USPS address verification service

BUILDING BLOCKS





IT organizations invariably employ disparate systems and technologies.



J2EE and .NET will continue to coexist in most organizations and the trend of having heterogeneous technologies in IT shops will continue.



SOA provides a clear solution to these application integration issues.



Allowing systems to expose their functionality via standardized, interoperable interfaces without rewriting the application.

WHY SOA?



Reusable components



Platform independent



Adapt applications to changing technologies.



Easily integrate applications with other systems.



Leverage existing investments in legacy applications.



Quickly and easily create a business process from existing services.

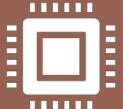
ADVANTAGES



Web services and SOA are not synonymous.



SOA is a design principle, whereas web services is an implementation technology.



You can build a service-oriented application without using web services--for example, by using other traditional technologies such as Java RMI.



There are two main API's defined by Java for developing web service applications since JavaEE 6.

JAX-WS: for SOAP web services.

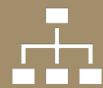
JAX-RS: for RESTful web services. There are mainly 2 implementation currently in use: Jersey and RESTeasy.

SOA AND JAVA



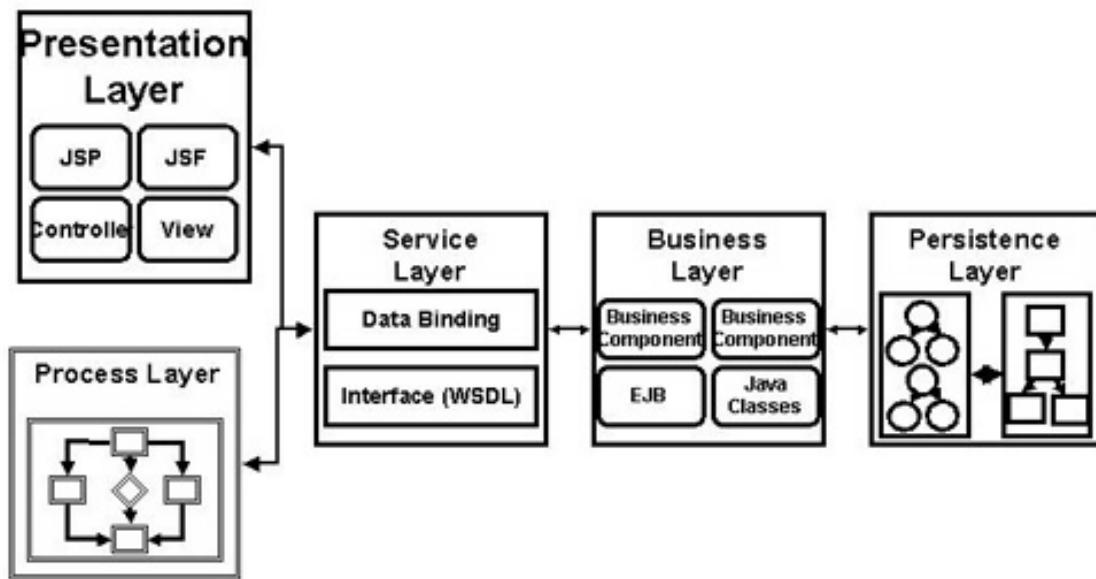
Service-oriented applications are multi-tier applications and have:

presentation, business logic, and persistence layers



The two key tiers in SOA are the

services layer and the business process layer



SOA LAYERS

SERVICE LAYER

Services are the building blocks of service-oriented applications.

Services are somewhat analogous to Java objects and components such as EJBs.

Unlike objects, however, services are self-contained, maintain their own state, and provide a loosely coupled interface.

Java provides a comprehensive platform for building the service layer of service-oriented applications.

Java APIs	Description
JAXP	Java API for XML Parsing
JAXB	Java API for XML Binding
JAX-RPC (JSR 101)	Java API for XML-Remote Procedure Call
SAAJ	SOAP API for Attachments in Java
JAXR	Java API for XML Registries
JSR 109	Web Services Deployment Model
EJB 2.1	Stateless Session EJB Endpoint Model

BUSINESS PROCESS LAYER

With SOA you can build a new application from existing services.

SOA has standardized business process modeling, often referred to as service orchestration.

You can build a web-service-based layer of abstraction over legacy systems and subsequently leverage them to assemble business processes.

Business Process Execution Language (BPEL) is the standard programming language for defining business processes represented in XML.



Partner links for the services with which the process interacts.



Variables for the data to be manipulated.



Correlations to correlate messages between asynchronous invocations.



Faults for message definitions for problems.



Compensation handlers to execute in the case of problems.



Event handlers that let the process deal with anticipated events in a graceful fashion.

BPEL



The presentation layer is used for user interaction.



Several Model-View-Controller (MVC) frameworks allow loose coupling between presentation layer and the model that supplies the data and business logic.



The main problem is that there's no standard way of binding data between different kinds of clients.



Clients have to know the exact underlying implementation of the service layer.

PRESENTATION LAYER



Representational state transfer (REST) or RESTful web services are a way of providing interoperability between computer systems on the Internet.



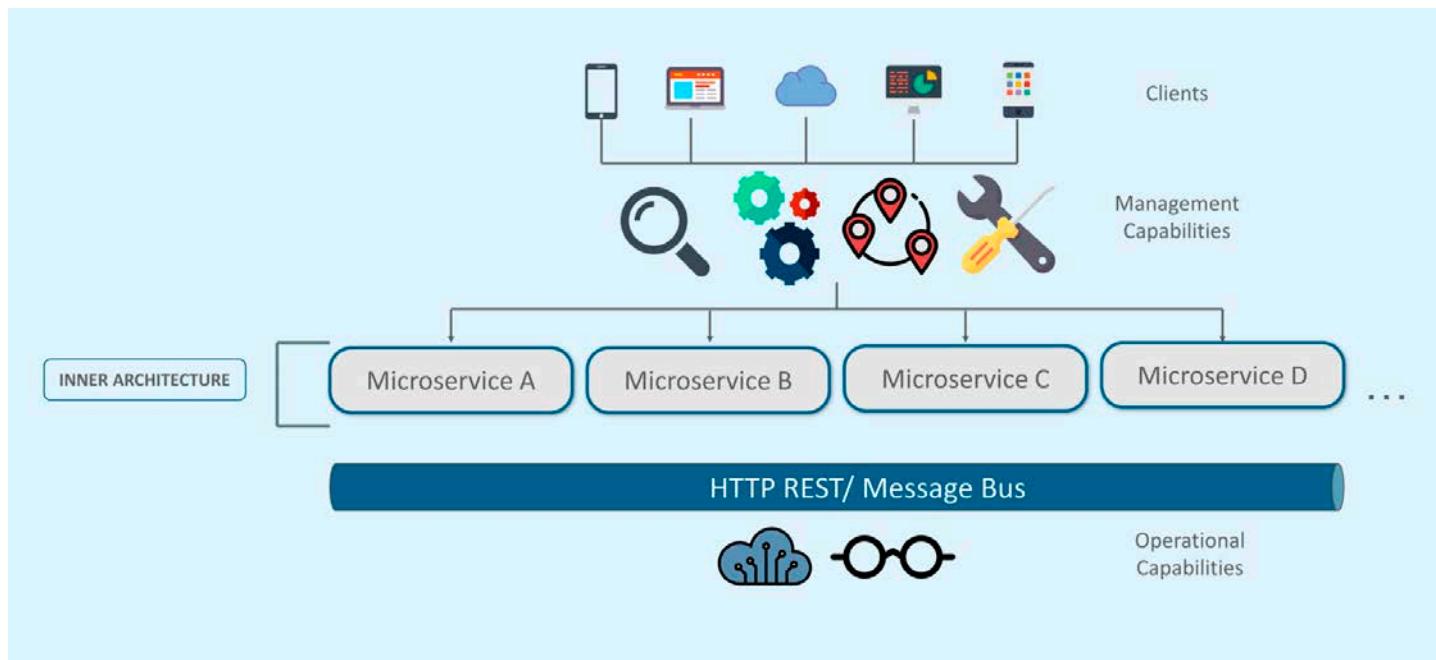
In a RESTful Web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format.



Using HTTP, as is most common, the kind of operations available include those predefined by the CRUD HTTP methods GET, POST, PUT, DELETE and so on.

REST SERVICES

MICROSERVICES



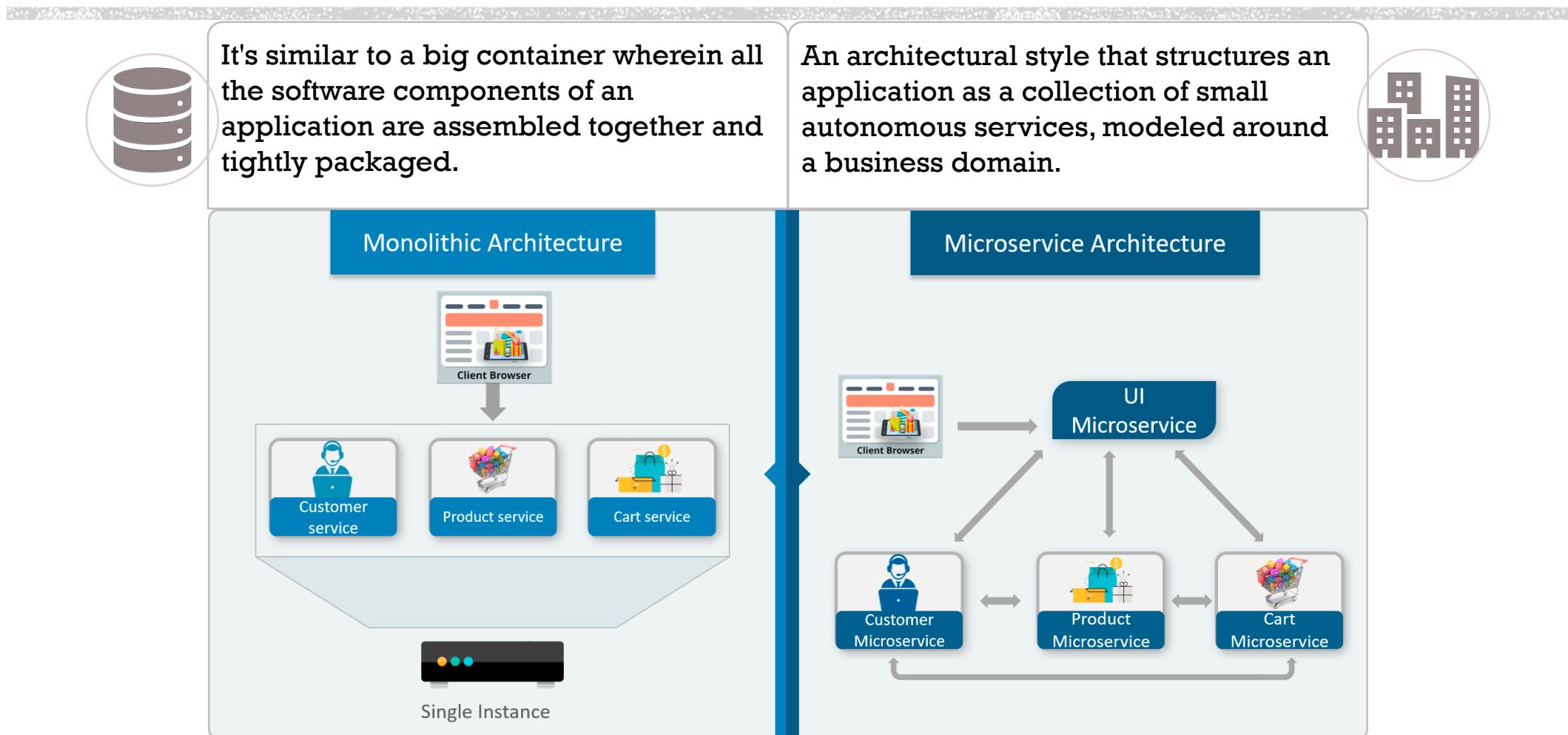
Microservices are a modern interpretation of SOA used to build distributed software systems.

Services are processes that communicate with each other over the network.

These services use technology agnostic protocols, which aid in encapsulating choice of language and frameworks.

Microservices have become popular since 2014 (and after the introduction of DevOps), and which also emphasize continuous deployment and other agile practices.

MONOLITHIC VS MICROSERVICES





Inflexible - Monolithic applications cannot be built using different technologies.



Unreliable - If even one feature of the system does not work, then the entire system does not work.



Unscalable - Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt.



Blocks Continuous Development - Many features of an application cannot be built and deployed at the same time.



Slow Development - Development in monolithic applications takes a lot of time to be built since each and every feature has to be built one after the other.



Not Fit for Complex Applications - Features of complex applications have tightly coupled dependencies.

CHALLENGES OF MONOLITHIC ARCHITECTURE

Decoupling

- Services within a system are largely decoupled, so the application as a whole can be easily built, altered, and scaled.

Componentization

- Microservices are treated as independent components that can be easily replaced and upgraded.

Business Capabilities

- Microservices are very simple and focus on a single capability.

Autonomy

- Developers and teams can work independently of each other, thus increasing speed.

Continuous Delivery

- Allows frequent releases of software through systematic automation of software creation, testing, and approval.

Responsibility

- Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible.

Decentralized Governance

- The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems.

Agility

- Microservices support agile development. Any new feature can be quickly developed and discarded again.

MICROSERVICES FEATURES





Independent Development

All microservices can be easily developed based on their individual functionality.



Independent Deployment

Based on their services, they can be individually deployed in any application.



Fault Isolation

Even if one service of the application does not work, the system still continues to function.



Mixed Technology Stack

Different languages and technologies can be used to build different services of the same application.



Granular Scaling

Individual components can scale as per need, there is no need to scale all components together.

MICROSERVICES ADVANTAGES



When you open a shopping cart application, all you see is just a website.



Behind the scenes, the shopping cart application has a service for accepting payments, a service for customer services and so on.

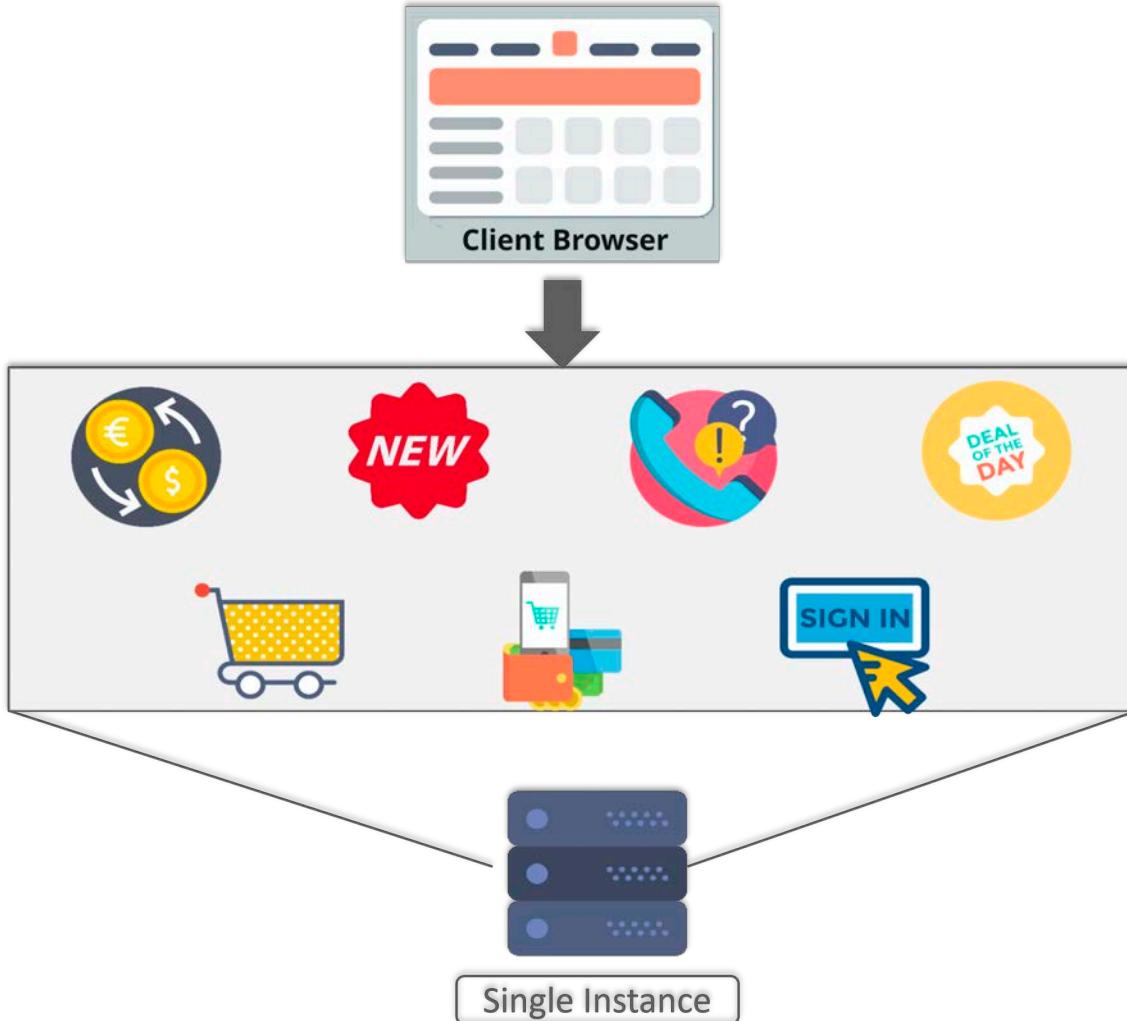


Solution can be implemented as a monolithic service or microservices.

EXAMPLE – SHOPPING CART



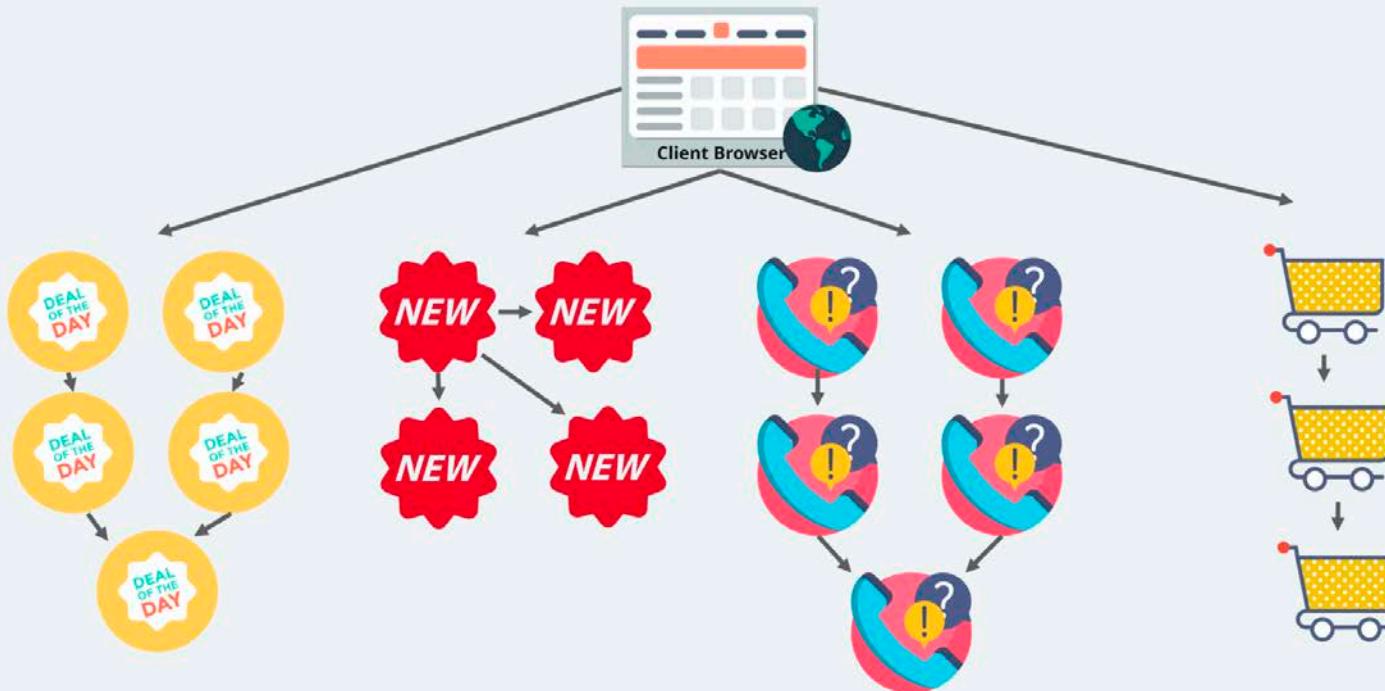
All the features are put together in a single code base and are under a single underlying database.



EXAMPLE — MONOLITHIC ARCHITECTURE



Create separate microservices for search, recommendations, customer services and so on. This helps the shopping cart application to be built, deployed, and scale up easily.



EXAMPLE – MICROSERVICES ARCHITECTURE



<https://www.mkyong.com/webservices/jax-rs/restfull-java-client-with-java-net-url/>



Get the code working.



Fix any errors and understand how rest services work.

PRACTICE LAB