Algorithms

Advantages of Binary SearchBinary Search Trees over Hash Tables 125

Binary Search 122

Branch Coverage 6

Code Walkthrough 21

**Component reusability** 137

**Debugging** 5, 141

Debugging Patterns 141

**Decision Coverage Testing** 5

Example: 6

**Defects** 6

**Defect Life Cycle?** 8

Defect Logging and Tracking 7

Defect Tracking Parameters 7

**Definition of software engineering** 11

Delta Release 8

Dependability 72

Design principles 78

Mean Time Between Failures (MTBF) 75

Mean Time to Failure (MTTF) 74

Mean Time to Repair (MTTR) 74

N-version programming 76

self-monitoring architectures 76

Design Patterns

Dependency Inversion Principle 148

Open Close Principle 146

Design Principle

Liskov Substitution Principle 42

Design Principles

Examples of Good and Bad Abstraction 44

Diagrams

State Transition Diagram 11

**Documentation** 139

Extreme Programming (XP) 32

Framework 32

Feasbility

Technical feasibility 129

Feasibility 128

**Economic feasibility** 130

Feasibility Study Process 130

Operational feasibility 129

Formal Inspection 20

**Formal Methods**

Formal Specifications 139

**Statechart** 10

**Forward Engineering** 137

Legacy Systems 76

**Maintenance** 120, 131

Maintenance Activities 134

Types 121

Pair Programming 20

**project management**

danger signs 61

solutions 61

Project Management

Agile 28

Agile Pros and Cons 30

Components of Development Plan 65

Incremental Waterfall 27

Organizational Issues 66

PNR Time/Effort Curve 70

Project Planning 65

Prototype Model 118

Rayleigh Staffling Profile Curve 69

Spiral Model 117

V Model 112

Waterfall Model 110

Quality 119

**Refactoring** 3

Bad Smells 5

When to refactor 4

Requirements

Definition of requirements engineering 14

Good requirements 13

Need 13

phases of development 15

Safety 75

Types 15

REST

Design Principles 97

**Reverse Engineering** 136

Risk Management 64

Categories of risk 67

**Example of Risk Classification** 144

fault-tree analysis 75

Risk Assessment 66

**Safety** 142

Example of safety claim hierarchy 145

Example of safety requirements 145

Risk Triangle 144

Safety critical systems 142

**San Diego Union (reported 1994-04-27)** 12

**Security**

**Assumptions** 72

**Design guidelines** 77

Testing 78

Services

Definition of microservices 128

Difference Between Microservices and SOA 128

Microservices 126

Web Services 125

Shellshock Bug (reported 2014-09-24) 11

**Software**

Definition 9

**Software Engineering**

Alternatives 12

Definition 11

Goals 18

Hooker Principles 18

System modeling

Hatley-Pirbhai model 40

Test Driven Development 17

Cons 17

Pros 17

**Testing**

All-Pairs 24

Alpha / Beta 122

Blackbox 21

Boundary Value Testing 22

Domain Testing 23

Equivalence Class Testing 22

experience-based testing 71

Exploratory Testing 23

Objectives of Reliability Testing 75

operational profile 71

penetration testing 71

**Security Testing** 64

Statistical Testing 71

Stress/Performance Testing 64

User Interface 86

Command Line Interface (CLI) 87

**Common Usability Problems** 96

Design Activities 91

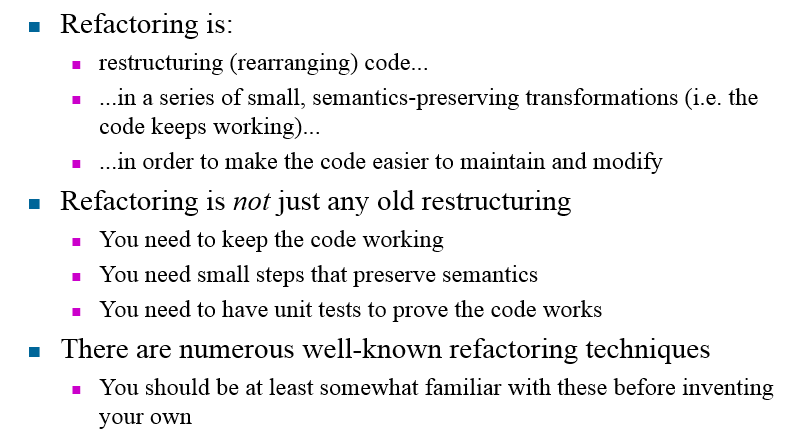
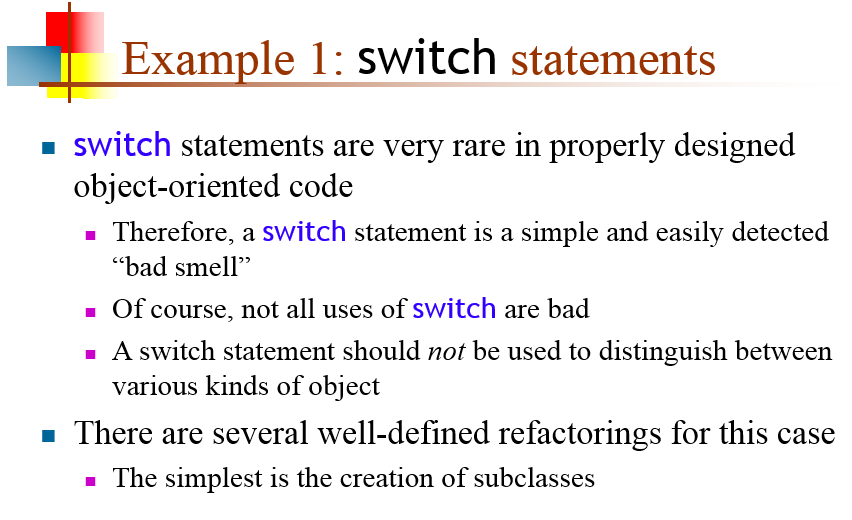
Design Issues 95

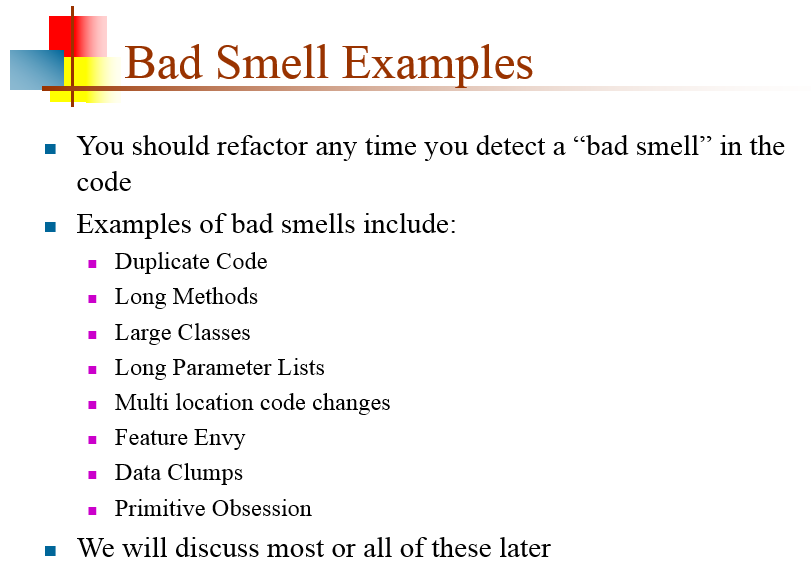
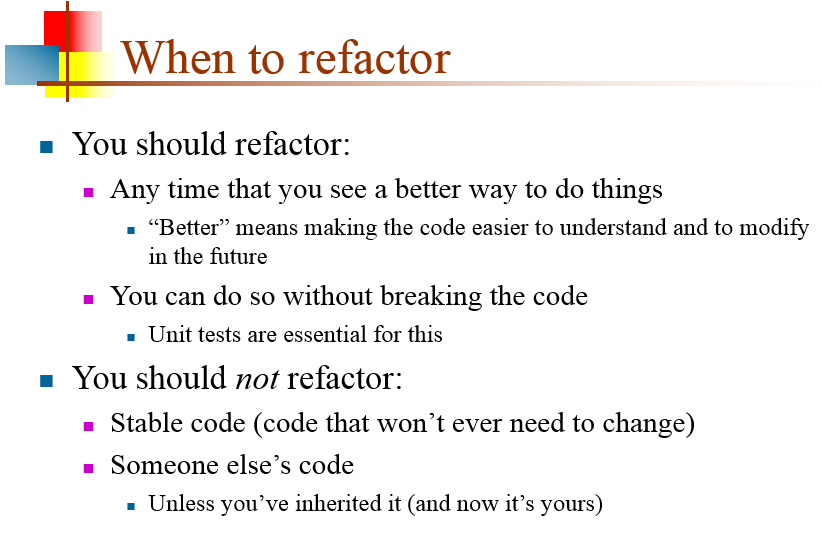
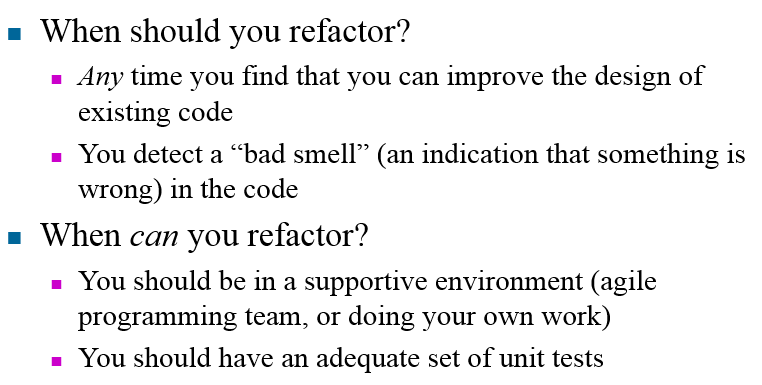
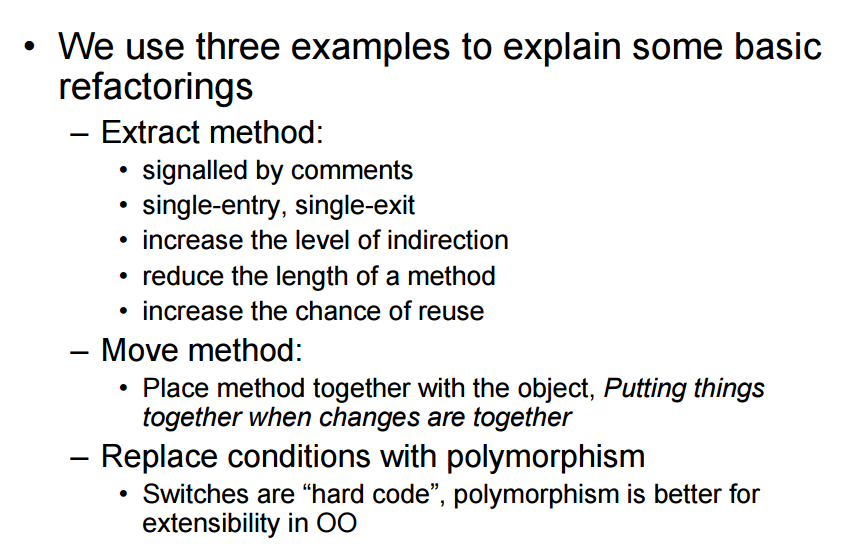
Design Principles 93

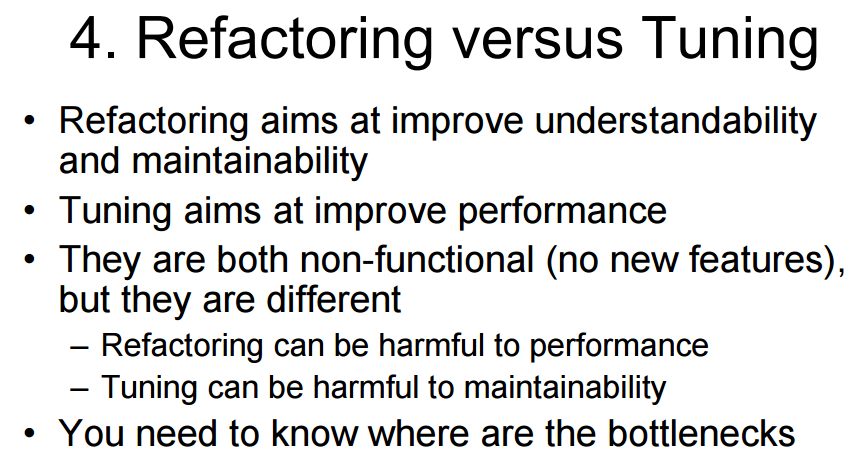
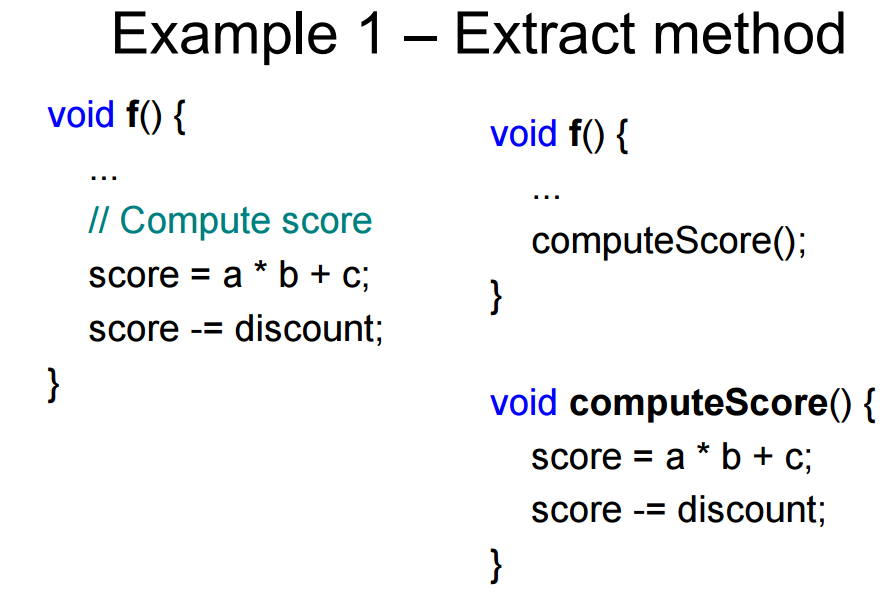
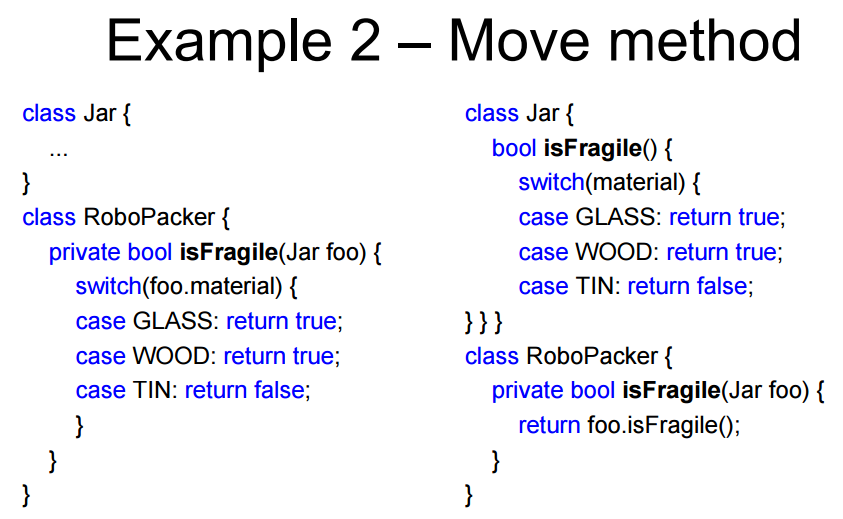
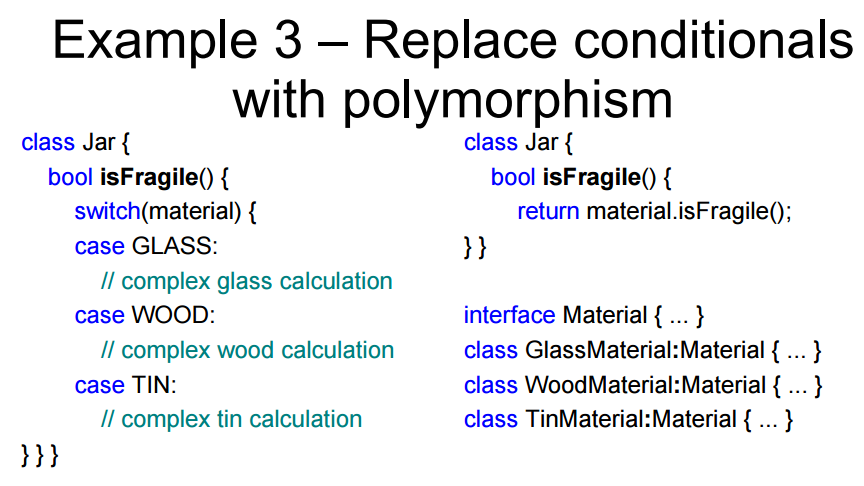
Graphical User Interface 88

Measuring Usability 96

**Refactoring**





****

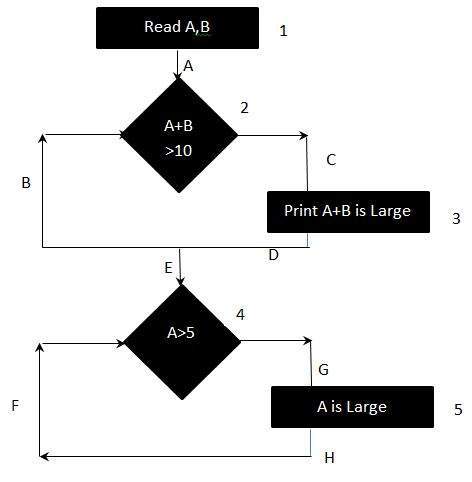
# **Debugging**

It is a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected. Debugging is harder for complex systems in particular when various subsystems are tightly coupled as changes in one system or interface may cause bugs to emerge in another.

Debugging is a developer activity and effective debugging is very important before testing begins to increase the quality of the system. Debugging will not give confidence that the system meets its requirements completely but testing gives confidence.

# **Decision Coverage Testing**

Decision coverage or Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

That is, every decision is taken each way, true and false. It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application.

**Example:**

Read A

Read B

IF A+B > 10 THEN

Print "A+B is Large"

ENDIF

If A > 5 THEN

Print "A Large"

ENDIF

The above logic can be represented by a flowchart as:

Result:

To calculate Branch Coverage, one has to find out the minimum number of paths which will ensure that all the edges are covered. In this case there is no single path which will ensure coverage of all the edges at once. The aim is to cover all possible true/false decisions.

(1) 1A-2C-3D-E-4G-5H

(2) 1A-2B-E-4F

Hence Decision or Branch Coverage is 2.

# **Defects**

A software bug arises when the expected result don't match with the actual results. It can also be error, flaw, failure, or fault in a computer program. Most bugs arise from mistakes and errors made by developers, architects.

Following are the methods for preventing programmers from introducing bugs during development:

* Programming Techniques adopted
* Software Development methodologies
* Peer Review
* Code Analysis

## Common Types of Defects

Following are the common types of defects that occur during development:

* Arithmetic Defects
* Logical Defects
* Syntax Defects
* Multithreading Defects
* Interface Defects
* Performance Defects

## Defect Logging and Tracking

Defect logging, a process of finding defects in the application under test or product by testing or recording feedback from customers and making new versions of the product that fix the defects or the clients feedback.

Defect tracking is an important process in software engineering as Complex and business critical systems have hundreds of defects. One of the challenging factors is Managing, evaluating and prioritizing these defects. The number of defects gets multiplied over a period of time and to effectively manage them, defect tracking system is used to make the job easier.

**Examples - Hp Quality Center, IBM Rational Quality Manager**

## Defect Tracking Parameters

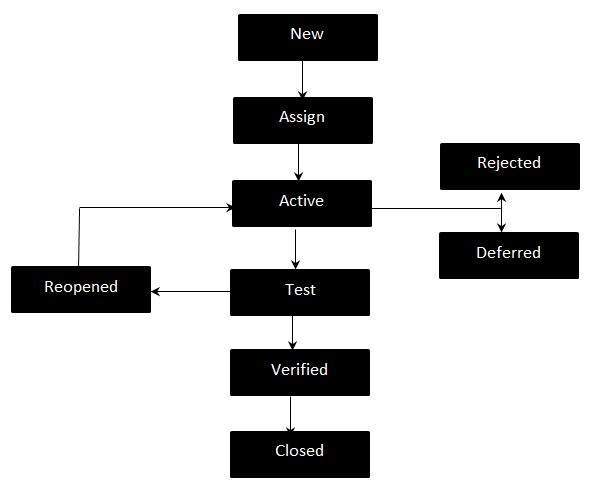
Defects are tracked based on various parameters such as:

* Defect Id
* Priority
* Severity
* Created by
* Created Date
* Assigned to
* Resolved Date
* Resolved By
* Status

**Defect Life Cycle?**

Defect life cycle, also known as Bug Life cycle is the journey of a defect cycle, which a defect goes through during its lifetime. It varies from organization to organization and also from project to project as it is governed by the software testing process and also depends upon the tools used.

**Defect Life Cycle - Workflow:**



**Defect Life Cycle States:**

* **New -**Potential defect that is raised and yet to be validated.
* **Assigned -**Assigned against a development team to address it but not yet resolved.
* **Active -**The Defect is being addressed by the developer and investigation is under progress. At this stage there are two possible outcomes; viz - Deferred or Rejected.
* **Test -**The Defect is fixed and ready for testing.
* **Verified -**The Defect that is retested and the test has been verified by QA.
* **Closed -**The final state of the defect that can be closed after the QA retesting or can be closed if the defect is duplicate or considered as NOT a defect.
* **Reopened -**When the defect is NOT fixed, QA reopens/reactivates the defect.
* **Deferred -**When a defect cannot be addressed in that particular cycle it is deferred to future release.
* **Rejected -**A defect can be rejected for any of the 3 reasons; viz - duplicate defect, NOT a Defect, Non Reproducible.

## Delta Release

A delta release also known as partial release is one that includes only those areas within the release unit that have changed or are new since the last full release or delta release.

If the release unit is the full program, a delta release contains only those modules that have changed since the last full release of the program.

# **Definition of Software**

---------------------------------------------

- set of rules a computer follows

microcode in commercial chips is not included as software

hardwired into the CPU

cannot be changed at all

firmware is not software either

halfway between hardware & software

code that is tightly bound with hardware

low-level code + hard to change (ROM)

we can write very low-level code

this is software but very low-level code

calculate a random number

// x86-64 machine, includes instruction

// return a random number 0 ~ 2^63 - 1

// this works only on x86-64 Hasweller machine

long randnum(void) { asm ("RDRAND"); }

**1. software is easy to change**

---------------------------

mutability of instructions

mutability brings forth engineering problems that make software hard

**2. it's not manufactured in the traditional sense**

-----------------------------------------------

don't have to worry about manufacturing

- type **'cp'** or **'scp'** to copy program and ship

**3. it doesn't wear out**

--------------------

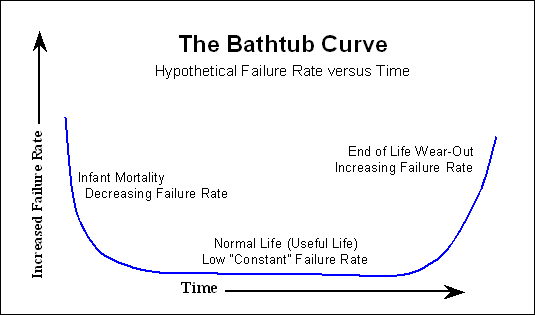
don't have to worry about parts wearing out

parts fail -> lubrication/replacement/repair

can run program any number of times you want

**hardware failure rate (bathtub curve)**

break-in period: not manufactured correctly

wear out: finally breaks down because of time

**software failure rate (spikey bathtub curve)**

increases failure rate with each new release

decreases failure rate as bugs get fixed

failure rate increases overall as software gets bigger

**4. there are no spare parts**

-------------------------

if program crashes, replacing with same new copy will not fix it

- maybe revert to previous version

gzip 1.7 on Solaris 10 with Oracle cc x86-64 crashes

configuration uses match.S <-- machine code does not run, dumps core

gzip 1.6 on this platform (can be considered spare part)

- maybe substitute a different implementation

use pigz instead

different implementation of gzip compression algorithm

runs in parallel

**Statechart**

A statechart is an alternative means of system specification. This specification method is particularly oriented to “reactive systems” — that is, systems that respond to a series of events rather than transforming an input into an output. Such systems may incorporate concurrent processing, and statecharts encompass this capability.

**Example 1: the cruise control system on an auto**

The events are increase/decrease of speed, application of the brake, etc., and actions include increasing/decreasing fuel supply, shifting the car into another gear, etc. Ordinarily examples of reactive systems have no natural stopping point.

**Example 2: a computer operating system**

An operating system is the agent that carries out physical input/output, but this is an action in response to a request and the operating system does not process the input or create the output (usually). There are numerous other event/action couplings among operating system reactions. Again, the normal expectation is that an operating system continues to run indefinitely.

• take the position that states and events are the natural medium of description for reactive systems

• focus on state transitions as basic fragments of such descriptions

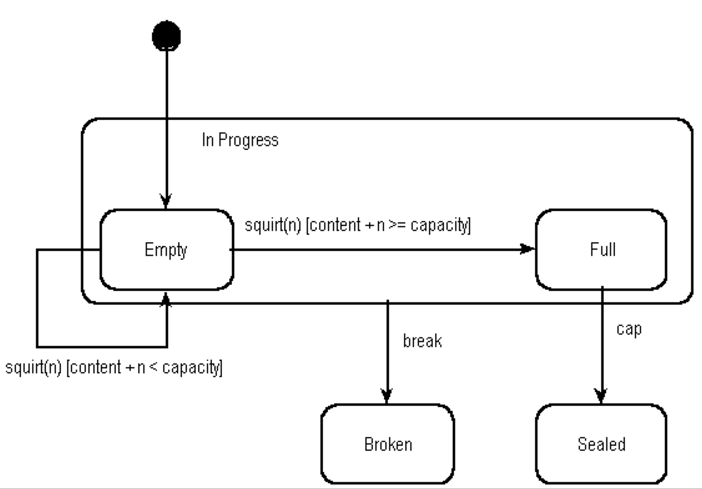
• emulate state-transition diagrams as the formal mechanism for collecting these fragments

• compensate for the state explosion in complex systems by replacing the “flat” unstratified view of states with a hierarchically structured state concept

• emphasize the visual character of descriptions to foster our intuitive grasp of the formalism

• are amenable to animation

# State Transition Diagram



State models are ideal for describing the behavior of a single object. They are also formal, so tools can be built which can execute them. Their biggest limitation is that they are not good at describing behavior that involved several objects, for these cases use an interaction diagram or an activity diagram.

People often do not find drawing state diagrams for several objects to be a natural way of describing a process. In these cases you can try either drawing a single state diagram for the process, or using an activity diagram. This defines the basic behavior, which you then need to refactor to split it across a number of objects.

# **Definition of software engineering**

------------------------------

1960s: software was out of control

"software crisis": too many bugs in software

need discipline for software development

**F.L. Baver (1969): the [establishment] and [use] of [sound] [engineering]**

**principles in order to obtain [economically] software that**

**is [reliable] and works [efficiently] on [real machines]**

this definition did not include teamwork

## Shellshock Bug (reported 2014-09-24)

-------------------------------------

bug in bash

$ cp = '(){ ... }'

$ export cp

this treated 'cp' as a function definition

then running 'cp a b' would run the function above

this bug had been in bash for 20 years

**San Diego Union (reported 1994-04-27)**

--------------------------------------

DMV spent $44 million on software that doesn't work and will never work

nobody was responsible

wanted to convert circa 1960 database to modern relational database

had bottleneck that could not handle the load that the DMV threw at it

\* Most software projects fail.

# Alternatives to software engineering

-------------------------------------

\* add people to a late project often not a good idea

**\* outsourcing**

outsource to another group

hire contractor, and let somebody else do the job

sometimes works, but 'distance' will be a problem

lose control of the software (intellectual rights access)

**\* vague objectives and requirements constantly changing**

"agile development"

sometimes works but often doesn't

**\* fire and forget**

get some code out the door by the stated deadline and never touching again

**\* code is all there is**

don't bother to write documentation because it is a hassle and probably

doesn't match the code either

**\* voluminous PPM (Policies & Procedures Manuals)**

software development process that documents everything

software development process takes much longer than it may take

good for developing reliable software

and proving that it is reliable

software engineering vs. computer science

----------------------------------------------------------------------

practical problems of theory & methods that

producing software underly programs

software engineering vs. system engineering

----------------------------------------------------------------------

just software plus software, hardware, firmware

human interface process design

policy

# Need for Requirements

-----------------------------

* developers != users
* legal, contract reasons
* reliability/safety is crucial
* security (tricky in practice)

How much work to put into requirements?

How long/big is the requirements document?

-------------------------------------------

it depends on the project

the more of the above required, the longer it will get

\* one of the most common requirements document problem is that customers would

not want to read the long and obvious document because they are specifying

what the customers already know.

## Requirement Engineering

------------------------

applying sound engineering principles to come up with requirements documents

requirements

stakeholders <---------+--------> design

|

- everybody who |

cares about |

the software |

- users/managers |

|

system model <---------+

- in developers' heads

### Good Requirements

- **are testable** (once system is implemented)

ideally would want to make it quantifiable

realistically turn it from something vague to something less vague

"build a system where the UI is userfriendly"

this is not testable

change to something testable

we could test them on users

**- are feasible** (in the indented environment)

make sure requirements are actually doable

do not make NP-complete requirements

**- don't conflict with each other**

conflict are not obvious

come from different parts of the requirements document

gathered from different parts of the customer organization

inconsistent requirements documents arise with conflicting user intentions

conflicts may not be obvious and may reflect conflicts among users

**- are attributed (to specific source)**

can go to any requirement in document and see who is responsible

each requirement should be attributed to specific source

should know who to ask if there is a problem in the requirements

**- are bounded**

do not want to have requirement that is infinitely hard to satisfy

should know when the requirements are satisfied

"as fast as possible"

do not want software engineers to develop "forever"

**- are unambiguous**

ambiguous: so few requirements or so poorly stated so that they can be

interpreted in many different ways

should avoid the ambiguity (English is ambiguous)

need something functional

**- are essential**

Aristotle: get at heart/core of the story

find out what really matters

this is called the "essence"

good requirements should focus on the essential part of the application

and not trivial details

**- are specified at the user level**

write in natural language that the user understands

do not write in code-like languages such as Java, C, Shell Script

**- match the system's vision**

when you try to change the world with your application, you should know what

the world looks like after your application

- are prioritized

an elaboration of "are essential"

some requirements are more important than others

in practice, some requirements conflict, but priority specifies direction

**- are validated**

requirements are checked

feasible: done a feasibility analysis

unambiguous: gone through whole document and checked for ambiguity

validated: we have to validate the validation process

# Definition of Requirements Engineering

Requirements Engineering (RE) is a set of activities concerned with identifying and communicating the purpose of a software-intensive system, and the contexts in which it will be used. Hence, RE acts as the bridge between the real-world needs of users, customers, and other constituencies affected by a software system, and the capabilities and opportunities afforded by software-intensive technologies.

## Types of Requirements

**(A) functional**

what the software does

behavior of software

get support from customers about functional requirements

**(B) nonfunctional**

other constraints on system that are less obvious because they do not

initially seem to have anything to do with what software does

security

reliability

performance

**(A) user**

imposed on system by end-users of application

will be able to look at user application and verify

**(B) system**

more detailed

more peopled are affected here

audience is people that want to make sure system work

developer

operation staff

finance

managers

## Phases of Requirements Development

**(1) inception**

some things may sound obvious but are easily done incorrectly or not done

**(a) identify stakeholders & their viewpoints**

stakeholders may not want to talk to you

e.g. prisoners in prison

have to indentify everyone who have something to do with the project

**(b) find agreements & disagreements**

get a good feeling on everybody who are using the system

may have completely different opinions between departments

(c) break the ice by asking "dumb questions"

context free questions

indicate that you don't understand the field

need humility

ask "dumb questions":

about goals and benefits (need to know why)

"how are you going to make money with this?"

about the problem

about communication activity itself

"did I ask all questions?"

"are there any questions I have left out?"

**(2) discovery/elicitation**

**(a) use a well-defined procedure**

- have meetings with agendas & prepare for the meetings

specialized training -> requirement facilitators (bridge gaps)

- define problems, pieces of solutions, in user-oriented way

- write everything down

- iterate -> multiple meetings

come up with draft document

**(b) produce**

**- scope of requirement**

specify boundary of what to do and what not to do

**- feasibility analysis**

show that requirement is feasible in document

**- justification of need**

why the requirement is needed

**- stakeholder list**

characterization of stakeholders & their viewpoints

**- environment characteristics**

what the system will operate in

where the system will be running

**- use cases**

little scenarios of where the system will be used

**- constraints**

any sort of extra high-level constraints that are not obvious

**- prototypes**

actually write some code as part of requirements discovery

may build end-to-end prototypes

tend to justify feasibility

"initial testing"

**(c) software requirements document (IEEE standard for requirements)**

contains

**- glossary**

standard nomenclature for problem

**- user requirements**

"normal" requirements

**- high-level system architecture**

document understanding of system model

**- system requirements**

stated in terms of high-level system architecture

**- system models**

- system evolution

potential changes to the requirements

**(3) negotiation**

come up with too many requirements and can't satisfy

ideally they are prioritized but practically not that easy

so we have to negotiate with clients

- want win-win situation

- key role of written requirements

**(4) validation**

list of things we want out of requirements

check consistency, completeness, etc.

via. reviews

prototypes (little programs to test)

test cases

# test driven development

buggy spec, if we explore all possible test cases, we can fill in the spec

test debug the spec before writing the code

it is simpler to write tests than writing code

now we can find the bugs in our spec faster

**do you use test driven development?**

we don't always practice what we preach. - Paul Eggert, Ph.D

if we have real-time constraints, this needs plan driven development

safety systems also requires plan driven development

the development team gets bigger and more parts of the software are not under

your control, and you can't continuously integrate.

#### **Pros of Test-Driven Development**

Proponents of TDD suggest that it leads to higher quality software in less time. Which is effectively saving money from a management point-of-view. If we drill down, there are various pros to TDD, such as:

* It can lead to simple, elegant, modular code.
* It can help developers find defects and bugs earlier than they otherwise would. It’s a common belief that a bug is cheaper to fix the earlier you find it.
* The tests can serve as a kind of live documentation and make the code much easier to understand.
* It’s easier to maintain and refactor code, your own and other programmer’s code.
* It can speed up development in the long-term.
* It can encourage developers to think from an end user point-of-view.

#### **Cons of Test-Driven Development**

In the absence of a lot of statistical evidence, it’s tough to say TDD definitely delivers. There’s no such thing as a one-size-fits-all solution in software development. Now, let’s take a look at some of the potential disadvantages:

* It necessitates a lot of time and effort up front, which can make development feel slow to begin with.
* Focusing on the simplest design now and not thinking ahead can mean major refactoring requirements.
* It’s difficult to write good tests that cover the essentials and avoid the superfluous.
* It takes time and effort to maintain the test suite – it must be reconfigured for maximum value.
* If the design is changing rapidly, you’ll need to keep changing your tests. You could end up wasting a lot of time writing tests for features that are quickly dropped.

## goals of software engineering

------------------------------

**(1) understand your problem**

much of the software engineering activity is devoted to finding the

problem that we are trying to solve, which is easiest to get wrong

**(2) design is crucial**

design better be there when we are done

we should know how the software was designed

**(3) quality**

software should always be high-quality

**(4) maintainability**

software has to be something that we can fix, improve, refactor

**(5) work across a lot of domains**

shouldn't be good for just one thing (web, realtime, embedded, system apps)

## software engineering principles (Hooker)

-----------------------------------------

**(1) provide value to users**

not always obvious

**(2) keep is simple stupid (KISS)**

when in doubt, use the simpler approach

keep code as simple as possible

**(3) have an architectural vision**

don't just look at little picture

**(4) plan to get hit by a bus**

do not assume that your software project will have you on it

other peoples may take over it

somebody else may be maintaining your project

if it's important, always write it down

be ready to be replaced

**(5) be ready to change**

designing and building software should not be like building the pyramid

it should be able to mutated

**(6) plan for reuse**

when you build your code, assume that it will be successful and you

or other people will reuse the current code

write code that can be reused in other systems

**(7) THINK before doing**

don't just code because it feels good to type keystrokes

think before you build the code, before it's too late

Sommerville likes 1,4,6, and

**(8) worry about dependability and performance**

obvious yet important

when you worry about dependability and performance, you are bringing to

the table software engineering strength

Developers Managers

------------------------------------------------------

I wanna code ensure it does what user wants

McConnell Sommerville

programming

textbooks

S.E. theory

software construction

----------------------

problem definition corrective maintenance

requirements definition

--------------------------------------------------------------

detailed design

construction planning coding integration

unit testing integration testing

--------------------------------------------------------------

software architecture system testing

(1) get your prerequisites right

**Plan to throw it away; you will anyhow - F. Brooks**

McConnell disagrees and thinks that we should make code work

software is not authored in the usual way

software is edited (like an encyclopedia)

don't plan to throw the whole thing away

**collaborative development**

**---------------------------**

(1) focus on cost-effective defect-detection

bugs will be the normal way of life

will spending more time fixing defects than writing code

reduce # of defects as many as possible

(2) collaborative practices do best on defects resistant to traditional tests

can you break up the project?

## pair programming

-----------------

pay 2 people's salaries to write one program

one programmer K has the keyboard and the house

one programmer J just kibitzes

find errors quickly, early when they are cheap to fix

if we wait to review, the cost goes up

reduces defect removal tests

have immediate feedback

back-and-forth is fast

requires 2 bus hits

guidelines

**(1) match pairs**

makes sure the two people are comfortable around each other

**(2) rotate**

switch roles

**(3) Keyboard = tactics**

Kibitzer = strategy

**(4) don't let the kibitzer relax**

make pair programming sessions short

**(5) don't use it for everything**

not all things are suited for pair programming

## formal inspection

------------------

gold standard for software review (IBM)

**(1) focus on defect detection not correction**

formal inspections are expensive

hard part is mostly finding bugs

so focus on hard part

**(2) use a checklist to focus reviewers' attention**

use different checklists and measure how well each checklist performs

checklist will depend on problem domain

**(3) reviewers prepare for meetings**

have multiple reviewers for reviewing system

give code ahead of time to read independently and come up with questions

in the meeting, combine the reviews, big merge of question list

**(4) all participants have roles**

moderator: requires most training, competent to organize reviewers

scribe: keep track of what is set

reviewers (2-5): reviews the code

author: usually doesn't participate, but also nice to have there

managers are not participants, inefficient

**(5) time and efficiency**

100-500 lines/hour

< 2 hours/meeting

code reviews are very expensive

code walk-through

------------------

code reading

demos (dog & pony)

"demo or die"

Code Walkthrough is a form of peer review in which a programmer leads the review process and the other team members ask questions and spot possible errors against development standards and other issues.

* The meeting is usually led by the author of the document under review and attended by other members of the team.
* Review sessions may be formal or informal.
* Before the walkthrough meeting, the preparation by reviewers and then a review report with a list of findings.
* The scribe, who is not the author, marks the minutes of meeting and note down all the defects/issues so that it can be tracked to closure.
* The main purpose of walkthrough is to enable learning about the content of the document under review to help team members gain an understanding of the content of the document and also to find defects.

## Black box Testing?

Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life cycle.

This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.

## Behavioural Testing Techniques:

There are different techniques involved in Black Box testing.

**Equivalence Class Testing**   
EC Testing is when you have a number of test items (e.g. values) that you want to test but because of cost (time/money) you do not have time to test them all. Therefore you group the test item into class where all items in each class are suppose to behave exactly the same. The theory is that you only need to test one of each item to make sure the system works.  
Example 1  
Children under 2 ride the buss for free. Young people pay $10, Adults $15 and Senior Citizen pay $5.  
Classes:  
Price:0 -> Age:0-1  
Price:10 -> Age:2-14   
Price:15 -> Age:15-64  
Price:5 -> Age:65-infinity

Example 2 (more than one parameter)  
Cellphones K80, J64 and J54 run Java 5. K90 and J99 run Java 6. But there are two possible browsers FireFox and Opera, J models run FF and K models run O.  
Classes:  
Browser:FF, Java:5 -> Phones:J64,J54  
Browser:FF, Java:6 -> Phones:J99  
Browser:O, Java:5 -> Phones:K80  
Browser:O, Java:6 -> Phones:K90

**Dangers Of Equivalence Class Testing**   
There is a danger of using EC Testing that is rarely mentioned in the testing books but is very important to remember.  
Just because two items/values are suppose to be in the same class and behave the same, does not mean they DO behave the same.  
That means that just because you test one value in the class that ALL values in the class behave the same. Real world example of mine is with cell phones that all had a certain Java Platform. They were suppose to all work the same but they didn't in reality. So testing just one value in a class is good, but not good enough. EC Testing is a good tool, but it's not fool proof and be careful with it. If test cases are cheap and fast (like automation), test more, or why not test them all!

**Boundary Value Testing**  
BV Testing is when you decide to test the values on the edge of each Class you have identified. The theory is that most defects is around the edges of a class. Example  
Classes:  
Price:0 -> Age:0-1 ( Boundary values 0, 1)  
Price:10 -> Age:2-14 ( Boundary values 2, 14)  
Price:15 -> Age:15-64 ( Boundary values 15, 64)  
Price:5 -> Age:65-infinity ( Boundary values 65)

**Critique of Boundary Value Testing**  
1) I, and other test professionals I have taken courses from, are not convinced that most defects are hidden around the edges of each class. And I have never seen any studies that proves this to be the case. 2) The fact that you need to use BV Testing proves that EC Testing is flawed since you test more than one value of each class. 3) It's easy to use when using values like integers. But what is a boundary value of class of phones models or browsers versions?

**Hidden Boundary Value Testing**  
The boundary values of a class is often based on the specification of how the system should work. This is all good and well but most systems contain boundaries that are not explained in any spec and you will need to look for yourself. E.g. 'How many characters can I put into the test field before the system fails and breaks.','How big can the data file become before it so slow to read it gets annoying'.  
Real world examples  
- Pasting one million characters into a text area in FireFox 3.5 on win 7 crashes it  
- ReCaptcha has a limit of 16003 characters, does your system handle the 413 that it passes back to it if somebody puts 16004+ characters in field. Or does it break

**Summary**  
EC Testing and BV Testing are great tools and you should use them but they are not perfect and don't expect to find all defects using them. Use your know-how about the system and your intelligence and intuition to try more items and looks for other ways it could fail. And look for the hidden boundaries!

* Domain Tests
* Domain testing is a software testing technique in which selecting a small number of test cases from a nearly infinite group of test cases. For testing few applications, Domain specific knowledge plays a very crucial role.
* Domain testing is a type of functional testing and tests the application by feeding interesting inputs and evaluating its outputs.
* Orthogonal Arrays
* Decision Tables
* State Models
* Exploratory Testing

During testing phase where there is severe time pressure, Exploratory testing technique is adopted that combines the experience of testers along with a structured approach to testing.

Exploratory testing often performed as a black box testing technique, the tester learns things that together with experience and creativity generate new good tests to run.

## Benefits:

Following are the benefits of Exploratory Testing:

* Exploratory testing takes less preparation.
* Critical defects are found very quickly.
* The testers can use reasoning based approach on the results of previous results to guide their future testing on the fly.

## Drawbacks:

Following are the Drawbacks of Exploratory Testing:

* Tests cannot be reviewed.
* It is difficult to keep track of what tests have been tested.
* It is unlikely to be performed in exactly the same manner and to repeat specific details of the earlier tests.
* **All-pairs testing**
* All-pairs also known as pairwise testing is a testing approach taken for testing the software using combinatorial method. It's a method to test all the possible discrete combinations of the parameters involved.
* Assume we have a piece of software to be tested which has got 10 input fields and 10 possible settings for each input field. Then, there are 10^10 possible inputs to be tested. In this case, exhaustive testing is impossible even if we wish to test all combinations.
* Let us also understand the concept of All-pairs testing by taking an example.
* Example
* An application with simple list box with 10 elements (Let's say 0,1,2,3,4,5,6,7,8,9) along with a checkbox, radio button, Text Box and OK Button. The Constraint for the Text box is it can accept values only between 1 and 100. Below are the values that each one of the GUI objects can take:
* List Box - 0,1,2,3,4,5,6,7,8,9
* Check Box - Checked or Unchecked
* Radio Button - ON or OFF
* Text Box - Any Value between 1 and 100
* Exhaustive combination of the application is calculated.
* List Box = 10
* Check Box = 2
* Radio Button = 2
* Text Box = 100
* Total Number of Test Cases using Cartesian Method : 10\*2\*2\*100 = 4000
* Total Number of Test Cases including Negative Cases will be > 4000
* Now, the idea is to bring down the number of test cases. We will first try to find out the number of cases using the conventional software testing technique. We can consider the list box values as 0 and others as 0 is neither positive nor negative. Radio button and check box values cannot be reduced. So each one of them will have 2 combinations (ON or OFF). The Text box values can be reduced into three inputs (Valid Integer, Invalid Integer, Alpha-Special Character).
* Now, we will calculate the number of cases using software testing technique is 2\*2\*2\*3 = 24 (including negative cases).
* Now, we can still reduce the combination further into All-pairs technique.
* **Step 1 :**Order the values such that one with most number of values is the first and the least is placed as the last variable.
* **Step 2 :**Now, start filling the table column by column. List box can take 2 values.
* **Step 3 :**The next column under discussion would be check box. Again, Check box can take 2 values.
* **Step 4 :**Now, we need to ensure that we cover all combinations between list box and Check box.
* **Step 5 :**Now, we will use the same strategy for checking the Radio Button. It can take 2 values.
* **Step 6 :**Verify if all the pair values are covered as shown in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Text Box** | **List Box** | **Check Box** | **Radio Button** |
| Valid Int | 0 | check | ON |
| Valid Int | others | uncheck | OFF |
| Invalid Int | 0 | check | ON |
| Invalid Int | others | uncheck | OFF |
| AlphaSpecialCharacter | 0 | check | ON |
| AlphaSpecialCharacter | others | uncheck | OFF |

* Result of Pair-Wise Testing
* Exhaustive Combination results in > 4000 Test Cases.
* Conventional Software Testing technique results in 24 Test Cases.
* Pair Wise Software Testing technique results in just 6 Test Cases.

## software process

-----------------

the set of heuristics

developers' heads (extreme approach)

(lisp (code))

lisp code in which developers are subroutines

this doesn't work

framework for what goes on in developers' heads

dynamic perspective - phases, in some sequence

communications (requirements)/planning/modeling/construction/deployment

practice perspective (umbrella activities)

- quality assurance

- reviews

- configuration management (how system/requirements are configured)

- project tracking (keep track of what's done/not done)

# plan driven vs. agile approach

different engineers have:

different terminology

different world views on how things work

software engineers want to work software engineers get more jobs

trading systems

nobody is in charge

if one system decides to change, others will have to deal with it

conceptual design --> procurement --> development

**software processes**

-------------------

(1) plan driven

**(a) waterfall model**

| requirements |

+--------------+

| architecture |

| design |

+--------------+

| component |

| design |

+-----------+

| coding |

+--------+

| unit testing |

+--------------+

| integration testing |

+---------------------+

| system testing |

+----------------+

| acceptance testing |

+--------------------+

finish each part before doing the next part

have a strict client who knows what they want

then the waterfall model is suitable

the requirements won't change

it is inflexible

going to spend a lot of time twiddling thumbs and waiting

**(b) incremental waterfall model**

repeated, parallel waterfalls

before first version completes we start gathering requirements on next

version to minimize the time to release version 3

**Advantages of Incremental Model**

* Generates working software quickly and early during the software life cycle.
* More flexible – less costly to change scope and requirements.
* Easier to test and debug during a smaller iteration.
* Easier to manage risk because risky pieces are identified and handled during its iteration.
* Each iteration is an easily managed milestone.

**Disadvantages of Incremental Model**

* Each phase of an iteration is rigid and do not overlap each other.
* Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

**When to use Incremental Model**

* Such models are used where requirements are clear and can implement by phase wise. From the figure it’s clear that the requirements ® is divided into R1, R2……….Rn and delivered accordingly.
* Mostly such model is used in web applications and product based companies.

**(c) spiral model**

increments "spirally"

goes in organic way

+---> plan -----> model ---------+

| |

+-> communication code <-+

| |

+-----------> deliver <---- test <----+

intent: each level of spiral is a major rethink such that the previous

versions doesn't matter too much

do risk analysis before next spiral

can revert if necessary

(d) concurrent development

break task into subtasks and resolve dependencies to concurrently

develop on the subtask with multiple waterfalls and merges

**(2) agile development**

Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds. These builds are provided in iterations. Each iteration typically lasts from about one to three weeks. Every iteration involves cross functional teams working simultaneously on various areas like −

* Planning
* Requirements Analysis
* Design
* Coding
* Unit Testing and
* Acceptance Testing.

At the end of the iteration, a working product is displayed to the customer and important stakeholders.

## What is Agile?

Agile model believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements. In Agile, the tasks are divided to time boxes (small time frames) to deliver specific features for a release.

Iterative approach is taken and working software build is delivered after each iteration. Each build is incremental in terms of features; the final build holds all the features required by the customer.

Here is a graphical illustration of the Agile Model −



The Agile thought process had started early in the software development and started becoming popular with time due to its flexibility and adaptability.

The most popular Agile methods include Rational Unified Process (1994), Scrum (1995), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995). These are now collectively referred to as **Agile Methodologies**, after the Agile Manifesto was published in 2001.

Following are the Agile Manifesto principles −

* **Individuals and interactions** − In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
* **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
* **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
* **Responding to change** − Agile Development is focused on quick responses to change and continuous development.

## Agile Vs Traditional SDLC Models

Agile is based on the **adaptive software development methods**, whereas the traditional SDLC models like the waterfall model is based on a predictive approach. Predictive teams in the traditional SDLC models usually work with detailed planning and have a complete forecast of the exact tasks and features to be delivered in the next few months or during the product life cycle.

Predictive methods entirely depend on the **requirement analysis and planning** done in the beginning of cycle. Any changes to be incorporated go through a strict change control management and prioritization.

Agile uses an **adaptive approach** where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed. There is feature driven development and the team adapts to the changing product requirements dynamically. The product is tested very frequently, through the release iterations, minimizing the risk of any major failures in future.

**Customer Interaction** is the backbone of this Agile methodology, and open communication with minimum documentation are the typical features of Agile development environment. The agile teams work in close collaboration with each other and are most often located in the same geographical location.

## Agile Model - Pros and Cons

Agile methods are being widely accepted in the software world recently. However, this method may not always be suitable for all products. Here are some pros and cons of the Agile model.

The advantages of the Agile Model are as follows −

* Is a very realistic approach to software development.
* Promotes teamwork and cross training.
* Functionality can be developed rapidly and demonstrated.
* Resource requirements are minimum.
* Suitable for fixed or changing requirements
* Delivers early partial working solutions.
* Good model for environments that change steadily.
* Minimal rules, documentation easily employed.
* Enables concurrent development and delivery within an overall planned context.
* Little or no planning required.
* Easy to manage.
* Gives flexibility to developers.

The disadvantages of the Agile Model are as follows −

* Not suitable for handling complex dependencies.
* More risk of sustainability, maintainability and extensibility.
* An overall plan, an agile leader and agile PM practice is a must without which it will not work.
* Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
* Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
* There is a very high individual dependency, since there is minimum documentation generated.
* Transfer of technology to new team members may be quite challenging due to lack of documentation.

rebellion against plans

promote: adaptability

self-organization (team will organize itself to get work done)

collaboration & communication

working software (new features) every two weeks

over: software processes

software tools

documentation

planning

developer teams need to select the work quantity

# **XP (eXtreme Programming) framework activities/phases**

-----------------------------------------------------

+---+ <- customer sets the value <-+

planning => set of stories -> | | |

+---+ <- dev team sets cost |

if > 3 weeks, split |

|

|

can do high-value first

can do high-risk first

can make commitment to a series

design => set of class designs -> CRC cards + spike solutions (running code)

|

(class responsibility collaborator)

spike solutions to test out a risky part of solution early

coding => code

unit tests

pair programming

refactoring

tinderboxes

testing => purely to acceptance testing

## Extreme Programming (XP)

----------

communication - informal, verbal

simplicity - don't over-engineer, code for today

discipline - "courage" to say no to client to keep things simple for goals

feedback - listen to clients, fellow developers, and software

respect - respect for clients, fellow developers, and software

**downside of XP (from a plan driven guy)**

----------------------------------------

lack of a formal design

lack of a formal requirements

volatile requirements (cause a lot of unnecessary work)

**downside of XP (from insider)**

------------------------------

1. inertia - fighting

2. refactoring /simplifying

3. prioritization is hard (especially in large organizations)

4. some developers aren't good collaborators

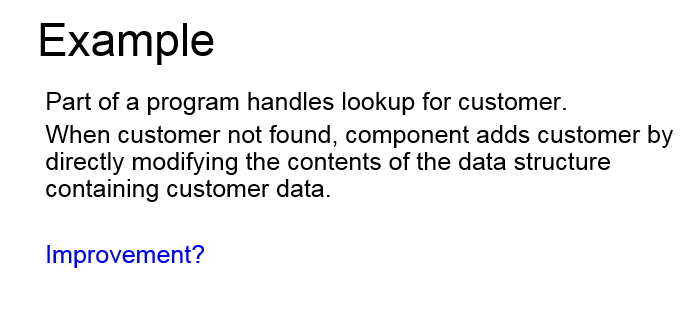
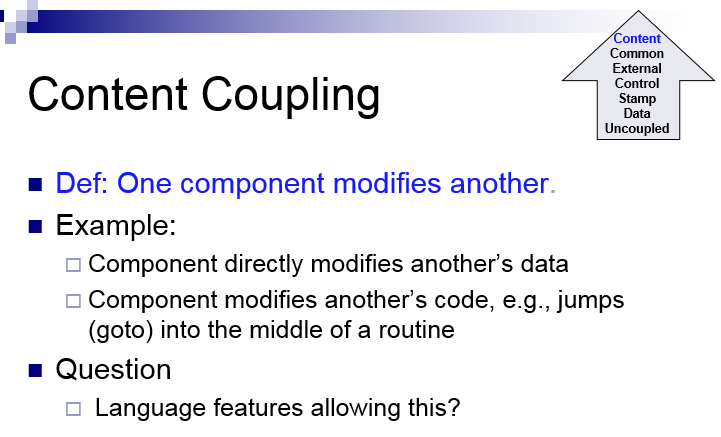
5. some clients aren't good collaborators

6. tinder boxes can't catch everything

|  |  |
| --- | --- |
| Planning [[User Stories](http://www.extremeprogramming.org/rules/userstories.html) User stories](http://www.extremeprogramming.org/rules/userstories.html) are written. [release plan](http://www.extremeprogramming.org/rules/planninggame.html) [Release planning](http://www.extremeprogramming.org/rules/planninggame.html) creates the [release schedule](http://www.extremeprogramming.org/rules/commit.html). [release often](http://www.extremeprogramming.org/rules/releaseoften.html) Make frequent [small releases](http://www.extremeprogramming.org/rules/releaseoften.html). [iterative](http://www.extremeprogramming.org/rules/iterative.html) The project is divided into [iterations](http://www.extremeprogramming.org/rules/iterative.html). [iteration planning](http://www.extremeprogramming.org/rules/iterationplanning.html) [Iteration planning](http://www.extremeprogramming.org/rules/iterationplanning.html) starts each iteration. Managing [optimize last](http://www.extremeprogramming.org/rules/space.html) Give the team a dedicated [open work space](http://www.extremeprogramming.org/rules/space.html). [steady pace](http://www.extremeprogramming.org/rules/overtime.html) Set a [sustainable pace](http://www.extremeprogramming.org/rules/overtime.html). [stand-up meeting](http://www.extremeprogramming.org/rules/standupmeeting.html) A [stand up meeting](http://www.extremeprogramming.org/rules/standupmeeting.html) starts each day. [project velocity](http://www.extremeprogramming.org/rules/velocity.html) The [Project Velocity](http://www.extremeprogramming.org/rules/velocity.html) is measured. [[move people around](http://www.extremeprogramming.org/rules/movepeople.html) Move people around](http://www.extremeprogramming.org/rules/movepeople.html). [[fix xp](http://www.extremeprogramming.org/rules/fixit.html) Fix XP](http://www.extremeprogramming.org/rules/fixit.html) when it breaks. Designing [[simplicity](http://www.extremeprogramming.org/rules/simple.html) Simplicity](http://www.extremeprogramming.org/rules/simple.html). [System Metaphor](http://www.extremeprogramming.org/rules/metaphor.html) Choose a [system metaphor](http://www.extremeprogramming.org/rules/metaphor.html). [CRC cards](http://www.extremeprogramming.org/rules/crccards.html) Use [CRC cards](http://www.extremeprogramming.org/rules/crccards.html) for design sessions. [Spike solution](http://www.extremeprogramming.org/rules/spike.html) Create [spike solution](http://www.extremeprogramming.org/rules/spike.html)s to reduce risk. [nothing early](http://www.extremeprogramming.org/rules/early.html) No functionality is [added early](http://www.extremeprogramming.org/rules/early.html). [[refactor](http://www.extremeprogramming.org/rules/refactor.html) Refactor](http://www.extremeprogramming.org/rules/refactor.html) whenever and wherever possible. | [Extreme Programming flow chart](http://www.extremeprogramming.org/map/project.html) Coding [customer on-site](http://www.extremeprogramming.org/rules/customer.html) The customer is [always available](http://www.extremeprogramming.org/rules/customer.html). [coding standard](http://www.extremeprogramming.org/rules/standards.html) Code must be written to agreed [standards](http://www.extremeprogramming.org/rules/standards.html). [Test Driven Development](http://www.extremeprogramming.org/rules/testfirst.html) Code the [unit test first](http://www.extremeprogramming.org/rules/testfirst.html). [pair programming](http://www.extremeprogramming.org/rules/pair.html) All production code is [pair programmed](http://www.extremeprogramming.org/rules/pair.html). [serial integration](http://www.extremeprogramming.org/rules/sequential.html) Only one pair [integrates code at a time](http://www.extremeprogramming.org/rules/sequential.html). [[continuous integration](http://www.extremeprogramming.org/rules/integrateoften.html) Integrate often](http://www.extremeprogramming.org/rules/integrateoften.html). [continuous integration](http://www.extremeprogramming.org/rules/dedicated.html) Set up a dedicated [integration computer](http://www.extremeprogramming.org/rules/dedicated.html). [collective ownership](http://www.extremeprogramming.org/rules/collective.html)Use [collective ownership](http://www.extremeprogramming.org/rules/collective.html). Testing [unit tests](http://www.extremeprogramming.org/rules/unittests.html) All code must have [unit tests](http://www.extremeprogramming.org/rules/unittests.html). [unit tests](http://www.extremeprogramming.org/rules/unittests.html) All code must pass all [unit tests](http://www.extremeprogramming.org/rules/unittests.html) before it  can http://www.extremeprogramming.org/images/pixel.gifbe released. [tests](http://www.extremeprogramming.org/rules/bugs.html) When [a bug is found](http://www.extremeprogramming.org/rules/bugs.html) tests are created. [[acceptance tests](http://www.extremeprogramming.org/rules/functionaltests.html) Acceptance tests](http://www.extremeprogramming.org/rules/functionaltests.html) are run often and the score http://www.extremeprogramming.org/images/pixel.gifis published. |
| User Stories User stories serve the same purpose as use cases but are not the same. They are used to create time estimates for the [release planning meeting](http://www.extremeprogramming.org/rules/planninggame.html). They are also used instead of a large requirements document. User Stories are written [by the customers](http://www.extremeprogramming.org/rules/customer.html) as things that the system needs to do for them. They are similar to usage scenarios, except that they are not limited to describing a user interface. They are in the format of about three sentences of text written by the customer in the customers terminology without techno-syntax.  User stories also drive the creation of the [acceptance tests.](http://www.extremeprogramming.org/rules/functionaltests.html) One or more automated acceptance tests must be created to verify the user story has been correctly implemented. One of the biggest misunderstandings with user stories is how they differ from traditional requirements specifications. The biggest difference is in the level of detail. User stories should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement. When the time comes to implement the story developers will go to the customer and receive a detailed description of the requirements face to face. | [Extreme Programming flow chart](http://www.extremeprogramming.org/map/project.html)  Developers estimate how long the stories might take to implement. Each story will get a 1, 2 or 3 week estimate in "ideal development time". This ideal development time is how long it would take to implement the story in code if there were no distractions, no other assignments, and you knew exactly what to do. Longer than 3 weeks means you need to break the story down further. Less than 1 week and you are at too detailed a level, combine some stories. About 80 user stories plus or minus 20 is a perfect number to create a [release plan](http://www.extremeprogramming.org/rules/commit.html) during release planning. Another difference between stories and a requirements document is a focus on user needs. You should try to avoid details of specific technology, data base layout, and algorithms. You should try to keep stories focused on user needs and benefits as opposed to specifying GUI layouts. [Extreme Programming](http://www.extremeprogramming.org/rules.html) |

# Coupling

* As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
* As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
* Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
* The objective is to keep coupling as low as possible
* The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
  + **Data coupling**
    - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
  + **Stamp coupling**
    - A whole data structure or class instantiation is passed as a parameter to an operation
  + **Control coupling**
    - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
    - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
  + **Common coupling**
    - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
  + **Content coupling**
    - One component secretly modifies data that is stored internally in another component
* Other kinds of coupling (unranked)
  + **Subroutine call coupling**
    - When one operation is invoked it invokes another operation within side of it
  + **Type use coupling**
    - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
    - If/when the type definition changes, every component that declares a variable of that data type must also change
  + **Inclusion or import coupling**
    - Component A imports or includes the contents of component B
  + **External coupling**
    - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)



# Component-level Design Principles

* **Open-closed principle**
  + A module or component should be open for extension but closed for modification
  + The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
* **Liskov substitution principle**
  + Subclasses should be substitutable for their base classes
  + A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
* **Dependency inversion principle**
  + Depend on abstractions (i.e., interfaces); do not depend on concretions
  + The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
* **Interface segregation principle**
  + Many client-specific interfaces are better than one general purpose interface
  + For a server class, specialized interfaces should be created to serve major categories of clients
  + Only those operations that are relevant to a particular category of clients should be specified in the interface

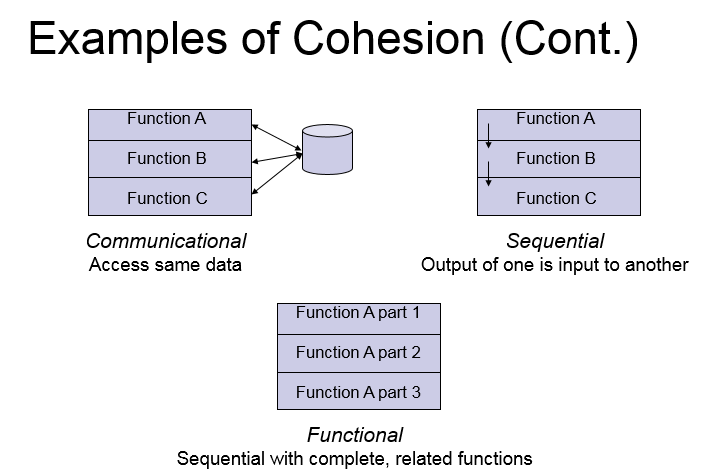
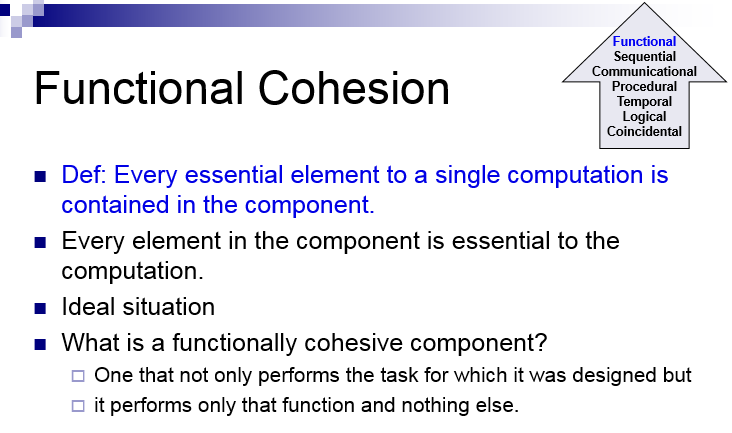
## Component Packaging Principles

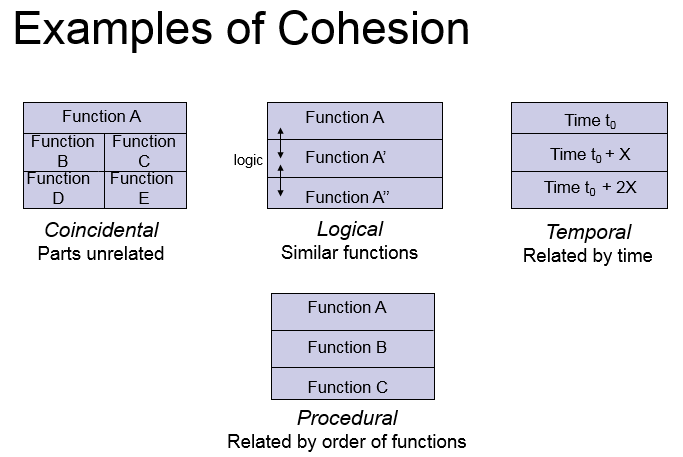
* **Release reuse equivalency principle**
  + The granularity of reuse is the granularity of release
  + Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
* **Common closure principle**
  + Classes that change together belong together
  + Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
* **Common reuse principle**
  + Classes that aren't reused together should not be grouped together
  + Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

## Cohesion

* Cohesion is the “single-mindedness’ of a component
* It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
* The objective is to keep cohesion as high as possible
* The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)

## Kinds of cohesion





* + **Sequential**
    - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
    - Def: The output of one part is the input to another. *Data flows* between parts (different from procedural cohesion). Occurs naturally in functional programming languages.

**Communicational:**

* Def: Functions performed on the same data or to produce the same data.
* Examples:
  + Update record in data base and send it to the printer
    - Update a record on a database
    - Print the record
  + Fetch unrelated data at the same time.
    - To minimize disk access
  + **Procedural**
    - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
    - Example:

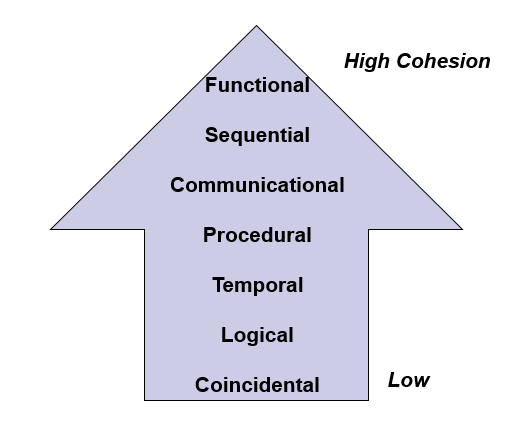
Write output record

Read new input record

Pad input with spaces

Return new record

* + **Temporal**
    - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
    - Def: Elements are related by timing involved. Elements are grouped by when they are processed.
    - Example: An exception handler that
      * Closes all open files
      * Creates an error log
      * Notifies user
      * Lots of different activities occur, all at same time
      * A **system initialization routine:** this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.
  + **Logical**
    - Def: Elements of component are related logically and not functionally.
    - Several logically related elements are in the same component and one of the elements is selected by the client component.
    - Ex:
      * A component reads inputs from tape, disk, and network.
      * All the code for these functions are in the same component.
      * Operations are related, but the functions are significantly different.
  + **Utility**
    - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other
    - Def: Parts of the component are unrelated (unrelated functions, processes, or data). Parts of the component are only related by their location in source code. Elements needed to achieve some functionality are scattered throughout the system. Accidental. Worst form



**Cohesion** refers to what the class (or module) will do. Low cohesion would mean that the class does a great variety of actions and is not focused on what it should do. High cohesion would then mean that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

Example of Low Cohesion:

-------------------

| Staff |

-------------------

| checkEmail() |

| sendEmail() |

| emailValidate() |

| PrintLetter() |

-------------------

Example of High Cohesion:

----------------------------

| Staff |

----------------------------

| -salary |

| -emailAddr |

----------------------------

| setSalary(newSalary) |

| getSalary() |

| setEmailAddr(newEmail) |

| getEmailAddr() |

----------------------------

As for **coupling**, it refers to how related are two classes / modules and how dependent they are on each other. Being low coupling would mean that changing something major in one class should not affect the other. High coupling would make your code difficult to make changes as well as to maintain it, as classes are coupled closely together, making a change could mean an entire system revamp.

All good software design will go for **high cohesion** and **low coupling**.

system engineering

-------------------

software engineering + everything else

hardware

networking

databases

documentation

people

procedures

\* systems nest

figure out how to meld together components and build systems out of systems

have multiple levels of nesting (at least 4 levels, usually 5 levels)

[WORLD VIEW]

enterprise strategy

overall goal

how to survive for the future twenty years

[Domain View]

business area design

interested in making sure students graduate on time

[Element view]

subsystem to help business to work

business system design

build tracking software and hardware to track students through degree

[detailed view]

correspond to traditional values of software engineering

construction and integration

### system engineering phases

--------------------------

conceptual design => system vision (feasibility + proposal)

procurement => regulations, competition, budget, buy vs build

development => e.g. waterfall

operation => system in use, bug reports, fix things while system operates

flexibility + adaptability are key

assume mistakes will happen & system will evolve

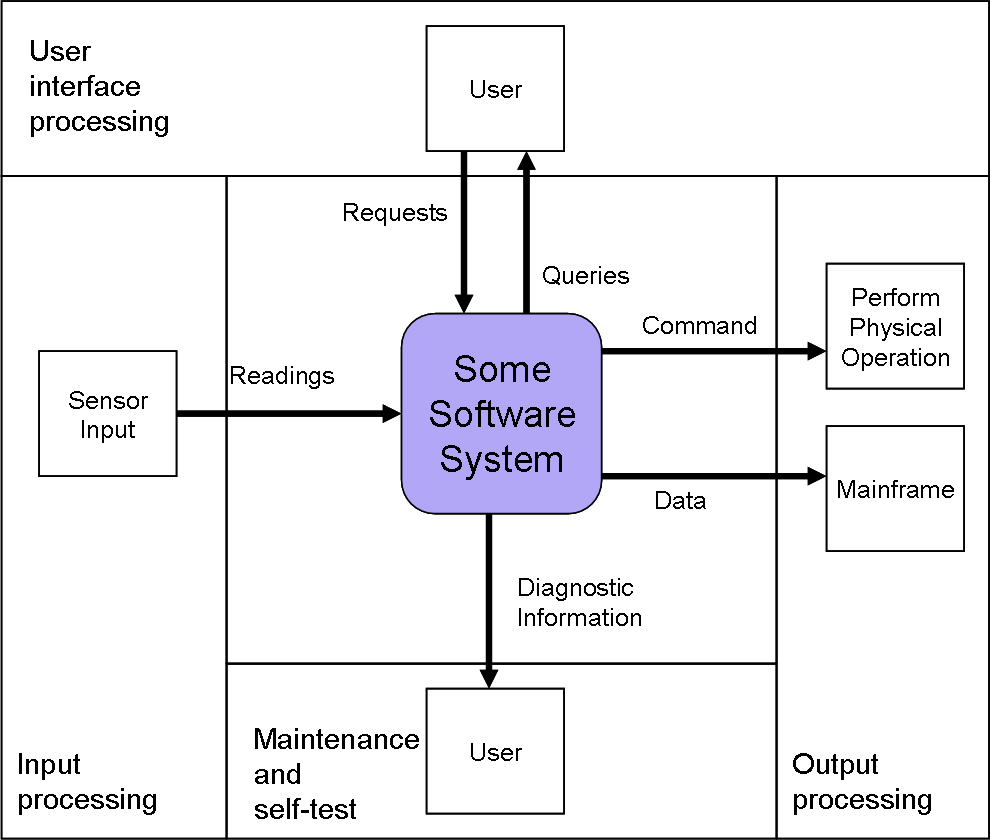
## system modeling

----------------

**Hatley-Pirbhai model**

Hatley–Pirbhai modeling is a system modeling technique based on the input–process–output model (IPO model), which extends the IPO model by adding user interface processing and maintenance and self-testing processing.[1]

The five components—inputs, outputs, user interface, maintenance, and processing—are added to a system model template to allow for modeling of the system which allows for proper assignment to the processing regions.[1] This modeling technique allows for creation of a hierarchy of detail of which the top level of this hierarchy should consist of a context diagram.[1] The context diagram serves the purpose of "establish[ing] the information boundary between the system being implemented and the environment in which the system is to operate."[1] Further refinement of the context diagram requires analysis of the system designated by the shaded rectangle through the development of a system functional flow block diagram.[1] The flows within the model represent material, energy, data, or information.[2]

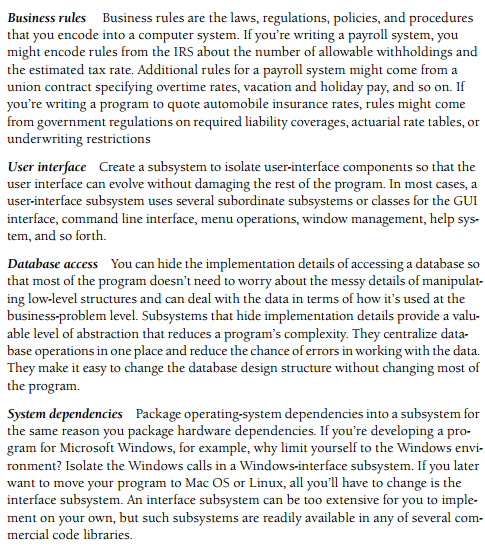


$ top

PID CPU CMD

... ... ...

## Common Subsystems



### describe a solution for the observer problem

---------------------------------------------

when a process changes, how do we notify the top program?

notification of change efficiently

want to decouble the observers from the doers

AbstractSubject AbstractObserver

addObserver() notify()

removeObserver()

notify() {

for o in observers:

o.notify()

}

ConcreteSubject ConcreteObserver

getState() State

can have several observers observing one subject

### Gang of Four of design problem

----------------------------------------

name

description of problem

description of solution

consequence (pros & cons) of the pattern

communicate with fellow software developers with short names

## Liskov Substitution Principle

------------------------------------------

have a parent class P and child classes C1 C2 C3

should be able to substitute C2 for C1 and the program should work the same

if (p instanceOf(C3))

print("got a C3!");

this violates the Liskov Substitution Principle

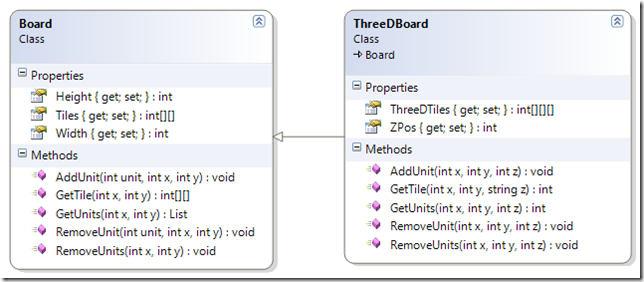
The Liskov Substitution Principle (LSP, [lsp](http://stackoverflow.com/questions/tagged/lsp)) is a concept in Object Oriented Programming that states:

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

At its heart LSP is about interfaces and contracts as well as how to decide when to extend a class vs. use another strategy such as composition to achieve your goal.

The most effective way I have seen to illustrate this point was in [Head First OOA&D](http://oreilly.com/catalog/9780596008673/index.html). They present a scenario where you are a developer on a project to build a framework for strategy games.

They present a class that represents a board that looks like this:



All of the methods take X and Y coordinates as parameters to locate the tile position in the two-dimensional array of Tiles. This will allow a game developer to manage units in the board during the course of the game.

The book goes on to change the requirements to say that the game frame work must also support 3D game boards to accommodate games that have flight. So a ThreeDBoard class is introduced that extends Board.

At first glance this seems like a good decision. Board provides both the Height and Widthproperties and ThreeDBoard provides the Z axis.

Where it breaks down is when you look at all the other members inherited from Board. The methods for AddUnit, GetTile, GetUnits and so on, all take both X and Y parameters in the Board class but the ThreeDBoard needs a Z parameter as well.

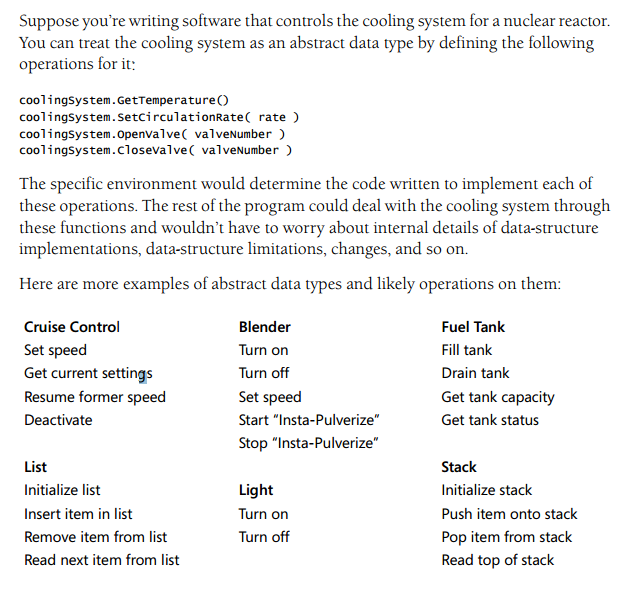
So you must implement those methods again with a Z parameter. The Z parameter has no context to the Board class and the inherited methods from the Board class lose their meaning. A unit of code attempting to use the ThreeDBoard class as its base class Board would be very out of luck.

Maybe we should find another approach. Instead of extending Board, ThreeDBoard should be composed of Board objects. One Board object per unit of the Z axis.

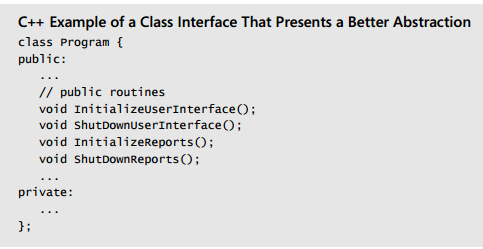
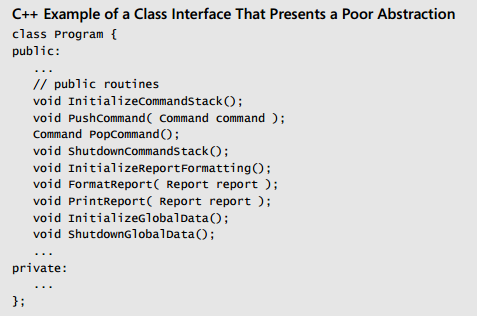
This allows us to use good object oriented principles like encapsulation and reuse and doesn’t violate LSP.

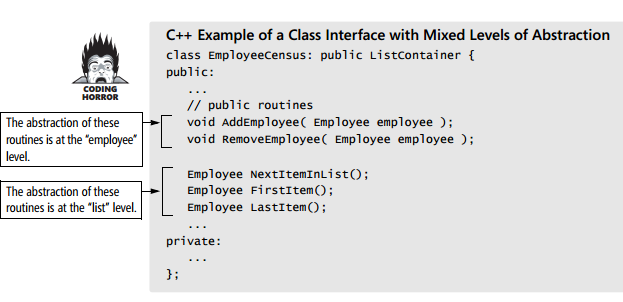
# Class Design

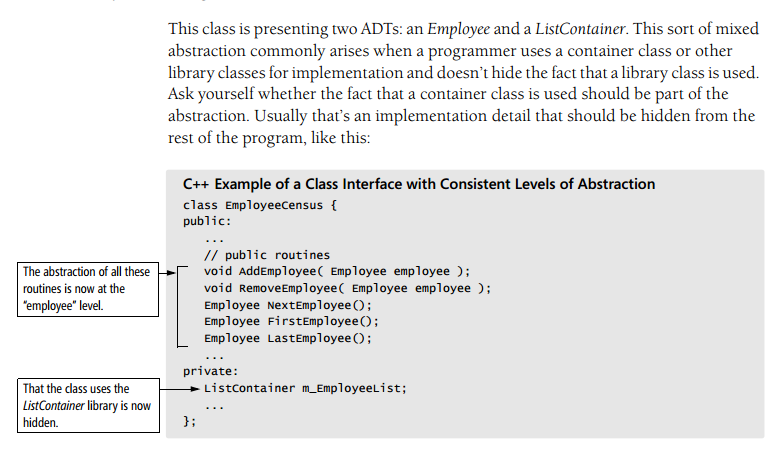
## Abstract Data Types

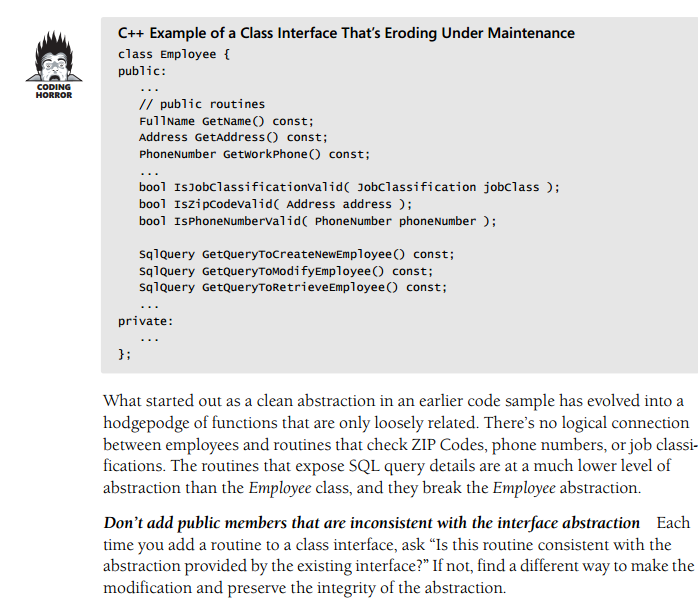


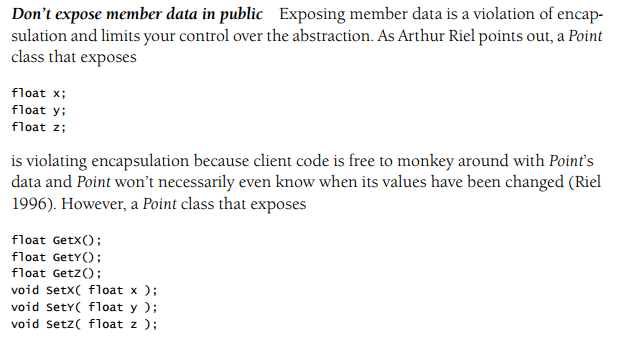
## Examples of Good and Bad Abstraction











## Law of Demeter

When applied to [object-oriented](https://en.wikipedia.org/wiki/Object-oriented) programs, the Law of Demeter can be more precisely called the “Law of Demeter for Functions/Methods” (LoD-F). In this case, an object A can request a service (call a method) of an object instance B, but object A should not "reach through" object B to access yet another object, C, to request its services. Doing so would mean that object A implicitly requires greater knowledge of object B's internal structure. Instead, B's interface should be modified if necessary so it can directly serve object A's request, propagating it to any relevant subcomponents. Alternatively, Amight have a direct reference to object C and make the request directly to that. If the law is followed, only object B knows its own internal structure.

More formally, the Law of Demeter for functions requires that a method m of an object O may only invoke the methods of the following kinds of objects:[[2]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-2)

1. O itself
2. m's parameters
3. Any objects created/instantiated within m
4. O's direct component objects
5. A global variable, accessible by O, in the scope of m

In particular, an object should avoid invoking methods of a member object returned by another method. For many modern object oriented languages that use a dot as field identifier, the law can be stated simply as "use only one dot". That is, the code a.b.Method() breaks the law where a.Method() does not. As an [analogy](https://en.wikipedia.org/wiki/Analogy), when one wants a dog to walk, one does not command the dog's legs to walk directly; instead one commands the dog which then commands its own legs.

**Advantages**

The advantage of following the Law of Demeter is that the resulting software tends to be more [maintainable](https://en.wikipedia.org/wiki/Maintainability) and [adaptable](https://en.wikipedia.org/wiki/Adaptive_reuse). Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers.

Basili et al.[[3]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-Basili-3) published experimental results in 1996 suggesting that a lower *Response For a Class* (RFC, the number of methods potentially invoked in response to calling a method of that class) can reduce the probability of [software bugs](https://en.wikipedia.org/wiki/Software_bug). Following the Law of Demeter can result in a lower RFC. However, the results also suggest that an increase in *Weighted Methods per Class* (WMC, the number of methods defined in each class) can increase the probability of software bugs. Following the Law of Demeter can also result in a higher WMC; see [Disadvantages](https://en.wikipedia.org/wiki/Law_of_Demeter#Disadvantages).

A [multilayered architecture](https://en.wikipedia.org/wiki/Multilayered_architecture) can be considered to be a systematic mechanism for implementing the Law of Demeter in a software system. In a layered architecture, code within each [layer](https://en.wikipedia.org/wiki/Layer_(object-oriented_design)) can only make calls to code within the layer and code within the next layer down. "Layer skipping" would violate the layered architecture.

**Disadvantages**

Although the LoD increases the adaptiveness of a software system, it may also result in having to write many wrapper methods to propagate calls to components; in some cases, this can add noticeable time and space overhead.[[3]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-Basili-3)[[4]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-BradApp-4)[[5]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-5)

At the method level, the LoD leads to narrow interfaces, giving access to only as much information as it needs to do its job, as each method needs to know about a small set of methods of closely related objects.[[6]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-6) On the other hand, at the class level, the LoD leads to wide (i.e. enlarged) interfaces, because the LoD requires introducing many auxiliary methods instead of digging directly into the object structures. One solution to the problem of enlarged class interfaces is the [aspect-oriented](https://en.wikipedia.org/wiki/Aspect-oriented_programming) approach,[[7]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-AOP-Programming-7) where the behavior of the method is specified as an aspect at a high level of abstraction. This is done by having an adaptive method that encapsulates the behaviour of an operation into a place, with which the scattering problem is solved. It also abstracts over the class structure that results in avoiding the tangling problem. The wide interfaces are managed through a language that specifies implementations. Both the traversal strategy and the adaptive visitor use only a minimal set of classes that participate in the operation, and the information about the connections between these classes is abstracted out.[[4]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-BradApp-4)[[7]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-AOP-Programming-7)

Since the LoD exemplifies a specific type of coupling, and does not specify a method of addressing this type of coupling, it is more suited as a metric for [code smell](https://en.wikipedia.org/wiki/Code_smell) as opposed to a methodology for building loosely coupled systems.

## accidents vs. essence

----------------------

core of design is essence

want to keep essence small as possible

want as few accidents as possible

call-and-return architectural style

subroutine calls

1 instruction pointer

more tightly coupled, fits in well with C++, Java

## message-passing architectural style

send message to server, eventually get a response

N instruction pointers

loosely coupled, fits in well with Smalltalk

usually more advanced and fancy

trend is going more towards this way

\* should write a programming language on this

Message passing is a technique for invoking behavior (i.e., running a program) on a computer. In contrast to the traditional technique of calling a program by name, message passing uses an [object model](https://en.wikipedia.org/wiki/Object_model) to distinguish the general function from the specific implementations. The invoking program sends a message and relies on the object to select and execute the appropriate code. The justifications for using an intermediate layer essentially falls into two categories: encapsulation and distribution.

[**Encapsulation**](https://en.wikipedia.org/wiki/Encapsulation_(computer_science)) is the idea that software objects should be able to invoke services on other objects without knowing or caring about how those services are implemented. Encapsulation can reduce the amount of coding logic and make systems more maintainable. E.g., rather than having IF-THEN statements that determine which subroutine or function to call a developer can just send a message to the object and the object will select the appropriate code based on its type.

One of the first examples of how this can be used was in the domain of computer graphics. There are all sorts of complexities involved in manipulating graphic objects. For example, simply using the right formula to compute the area of an enclosed shape will vary depending on if the shape is a triangle, rectangle, elipse, or circle. In traditional computer programming this would result in long IF-THEN statements testing what sort of object the shape was and calling the appropriate code. The object-oriented way to handle this is to define a class called Shape with subclasses such as Rectangle and Ellipse (which in turn have subclasses Square and Circle) and then to simply send a message to any Shape asking it to compute its area. Each Shape object will then invoke the way code with the formula appropriate for that kind of object.[[1]](https://en.wikipedia.org/wiki/Message_passing#cite_note-1)

Distributed message passing provides developers with a layer of the architecture that provides common services to build systems made up of sub-systems that run on disparate computers in different locations and at different times. When a distributed object is sending a message, the messaging layer can take care of issues such as:

* Finding the appropriate object, including objects running on different computers, using different operating systems and programming languages, at different locations from where the message originated.
* Saving the message on a queue if the appropriate object to handle the message is not currently running and then invoking the message when the object is available. Also, storing the result if needed until the sending object is ready to receive it.
* Controlling various transactional requirements for distributed transactions, e.g. ensuring [ACID](https://en.wikipedia.org/wiki/ACID) properties on data.[[2]](https://en.wikipedia.org/wiki/Message_passing#cite_note-2)

### Synchronous versus asynchronous message passing

One of the most important distinctions among message passing systems is whether they use synchronous or asynchronous message passing. Synchronous message passing occurs between objects that are running at the same time. With asynchronous message passing it is possible for the receiving object to be busy or not running when the requesting object sends the message.

Synchronous message passing is what typical object-oriented programming languages such as Java and Smalltalk use. Asynchronous message passing requires additional capabilities for storing and retransmitting data for systems that may not run concurrently.

The advantage to synchronous message passing is that it is **conceptually less complex.** Synchronous message passing is analogous to a function call in which the message sender is the function caller and the message receiver is the called function. Function calling is easy and familiar. Just as the function caller stops until the called function completes, the sending process stops until the receiving process completes. This alone makes synchronous message unworkable for some applications. For example, if synchronous message passing would be used exclusively, large, distributed systems generally would not perform well enough to be usable. Such large, distributed systems may need to continue to operate while some of their subsystems are down; subsystems may need to go offline for some kind of maintenance, or have times when subsystems are not open to receiving input from other systems.

Imagine a busy business office having 100 desktop computers that send emails to each other using synchronous message passing exclusively. Because the office system does not use asynchronous message passing, one worker turning off their computer can cause the other 99 computers to freeze until the worker turns their computer back on to process a single email.

Asynchronous message passing is generally implemented so that all the complexities that naturally occur when trying to synchronize systems and data are handled by an intermediary level of software. Commercial vendors who develop software products to support these intermediate levels usually call their software "[middleware](https://en.wikipedia.org/wiki/Middleware)". One of the most common types of middleware to support asynchronous messaging is called [Message-oriented middleware](https://en.wikipedia.org/wiki/Message-oriented_middleware) (MOM).

With asynchronous message passing, the sending system does not wait for a response. Continuing the function call analogy, asynchronous message passing would be a function call that returns immediately, without waiting for the called function to execute. Such an asynchronous function call would merely deliver the arguments, if any, to the called function, and tell the called function to execute, and then return to continue its own execution. Asynchronous message passing simply sends the message to the message bus. The bus stores the message until the receiving process requests messages sent to it. When the receiving process arrives at the result, it sends the result to the message bus, and the message bus holds the message until the original process (or some designated next process) picks up its messages from the message bus.[[3]](https://en.wikipedia.org/wiki/Message_passing#cite_note-3)

Synchronous communication can be built on top of asynchronous communication by using a [Synchronizer](https://en.wikipedia.org/wiki/Synchronizer_(algorithm)). For example, the α-Synchronizer works by ensuring that the sender always waits for an acknowledgement message from the receiver. The sender only sends the next message after the acknowledgement has been received. On the other hand, asynchronous communication can also be built on top of synchronous communication. For example, modern [microkernels](https://en.wikipedia.org/wiki/Microkernel) generally only provide a [synchronous messaging](https://en.wikipedia.org/wiki/Microkernel#Inter-process_communication) primitive[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]and asynchronous messaging can be implemented on top by using [helper threads](https://en.wikipedia.org/wiki/Thread_(computing)).

The buffer required in asynchronous communication can cause problems when it is full. A decision has to be made whether to block the sender or whether to discard future messages. **If the sender is blocked, it may lead to an unexpected**[**deadlock**](https://en.wikipedia.org/wiki/Deadlock)**.** If messages are dropped, then communication is no longer reliable. These are all examples of the kinds of problems that middleware vendors try to address.

## repository architecture (DB)

send data in to database and others can pickup whenever they want

## layered architecture

build applications in terms of layers on top of machine

each boundary gives abstraction level

+-----------------------+

| Python libraries |

+-----------------------+

| Python interpreter |

+-----------+-----------+

| OS Kernel | C library |

+-----------+-----------+

| machine instructions |

+-----------------------+

## client-server architecture

may be attached to DB

connected to network

have a client attached to display, keyboard, mouse

clients make requests to server and get response back

server is in charge, responsible

clients can only issue requests

event-processing architecture

events

+-+-+-+-+-+-+-+-+-+-+

outside world -> | | | |\*|\*|\*|\*|\*|\*|\*|

+-+-+-+-+-+-+-+-+-+-+

event queue

event processing

for (;;) {

e = remove\_queue();

handle(e); // this part must be fast

} // could do some work, then signal an event!

// to handle large events, partition and add to event queue

### distributed architecture

-------------------------

**+ increased availability & reliability & fault tolerance**

high availability means system is up at most times (0.999 availability)

failures of individual components don't cause performance degradation

if one server crashes, system can keep running with other servers

**+ scalability via concurrency**

this comes up even in a non-distribute system

always an issue in all systems

common impression is that distributed systems will work better with more

servers, but this is not the case as systems do not always scale

**+ openness**

achieved through protocols

key way to glue together distributed architecture

**+ resource sharing**

can support many different kinds of services on the same platform

machines support many different users simultaneously

this saves money & resources

key driver behind distributed architecture

BUT MUST CONSIDER THE FOLLOWING ISSUES

**- failure management**

keep track which part of system is not working

**- quality of service (QoS)**

how to specify quality of service

response time

requests/second

**- security**

safeness of system

**- transparency**

do users know it's distributed?

**- how to scale**

geography, where to put the servers?

manageability, a million servers -> configure by hand?

**- how open, really?**

specialized, proprietary protocols

# Distributed architecture examples

----------------------------------

master-slave

master

-------------------+-------------------

+-------+ +-------+ +-------+ +-------+

| slave | | slave | | slave | | slave |

+-------+ +-------+ +-------+ +-------+

### fat vs. thin client

fat clients contain lots of state, code, processing

less server load

thin clients have little state (stateless), code, processing (may have cache)

makes it easier to run on different platforms

makes app size smaller

security gets better since most of computation is done on server

In designing a client–server application, a decision is to be made as to which parts of the task should be executed on the client, and which on the server. This decision can crucially affect the cost of clients and servers, the robustness and security of the application as a whole, and the flexibility of the design to later modification or [porting](https://en.wikipedia.org/wiki/Porting).

The characteristics of the user interface often force the decision on a designer. For instance, a drawing package could choose to download an initial image from a server and allow all edits to be made locally, returning the revised drawing to the server upon completion. This would require a thick client and might be characterised by a long time to start and stop (while a whole complex drawing was transferred) but quick to edit.

Conversely, a thin client could download just the visible parts of the drawing at the beginning and send each change back to the server to update the drawing. This might be characterised by a short start-up time, but a tediously slow editing process.

### Centrally hosted thick client applications

Probably the thinnest clients (sometimes called "Ultra Thin") are [remote desktop](https://en.wikipedia.org/wiki/Remote_desktop) applications, for example the [X Window System](https://en.wikipedia.org/wiki/X_Window_System), [Citrix](https://en.wikipedia.org/wiki/Citrix) products and [Microsoft](https://en.wikipedia.org/wiki/Microsoft)'s [Terminal Services](https://en.wikipedia.org/wiki/Terminal_Services), which effectively allow applications to run on a centrally-hosted virtual PC and copy keystrokes and screen images between the local PC and the virtual PC. Ironically, these ultra-thin clients are often used to make available complex or data-hungry applications which have been implemented as thick clients but where the true client is hosted very near to the network server.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

**Advantages**

* **Lower server requirements**. A thick client server does not require as high a level of performance as a thin client server (since the thick clients themselves do much of the application processing). This results in drastically cheaper servers.
* **Working offline**. Thick clients have advantages in that a constant connection to the central server is often not required.
* **Better multimedia performance**. Thick clients have advantages in multimedia-rich applications that would be bandwidth intensive if fully served. For example, thick clients are well suited for [video gaming](https://en.wikipedia.org/wiki/Video_gaming).
* **More flexibility**. On some operating systems software products are designed for personal computers that have their own local resources. Running this software in a thin client environment can be difficult.
* **Using existing infrastructure**. As many people now have very fast local PCs, they already have the infrastructure to run thick clients at no extra cost.
* **Higher server capacity**. The more work that is carried out by the client, the less the server needs to do, increasing the number of users each server can support.

### multi-tiered-client server

layered architecture applied to client-server

browser

|

load balance

| |

cache cache

| |

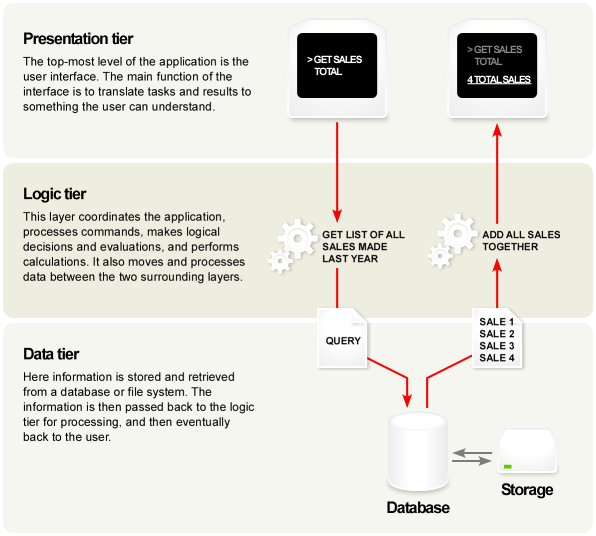
application server

|

database server

In software engineering, multi-tier architecture (often referred to as n-tier architecture) is a client-server architecture in which, the presentation, the application processing and the data management are logically separate processes. For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture. The most widespread use of "multi-tier architecture" refers to three-tier architecture.

It's debatable what counts as "tiers," but in my opinion it needs to at least cross the process boundary. Or else it's called layers. But, it does not need to be in physically different machines. Although I don't recommend it, you can host logical tier and database on the same box.



**Edit**: One implication is that presentation tier and the logic tier (sometimes called Business Logic Layer) needs to cross machine boundaries "across the wire" sometimes over unreliable, slow, and/or insecure network. This is very different from simple Desktop application where the data lives on the same machine as files or Web Application where you can hit the database directly.

For n-tier programming, you need to package up the data in some sort of transportable form called "dataset" and fly them over the wire. .NET's [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) class or Web Services protocol like [SOAP](http://www.w3.org/TR/soap/) are few of such attempts to fly objects over the wire.

### peer-to-peer (P2P)

internet originally designed that there wouldn't be client or server

"clients" only (peers)

run mostly because peers need to interchange information with each other

without the need for central server (control catalog) to manage them

can have helpers (semi-centralized architecture)

security is a problem, trust can be issues

### software as a service (SaaS)

applications on servers controlled by other organizations

how to glue together lost s of little services

outside your control

rely on other services to get your service to work, what if it goes down?

### management + architecture

--------------------------

- buy vs. build (any architecture)

- change management

- risk identification

- feasibility analysis

- quality requirements ("over-engineering")

### construction decisions

-----------------------

**- programming language**

should have at command many different languages

different languages are meant for different work

**- programming conventions**

hard to change after initial coding

$ foo --version # should print version and exit

can control this more easily than the language

**- programming tools**

IDE (Eclipse)

design

-------

history of software design

modular program 1950s

top-down development 1960s

structured programming ("goto-less programming")

sequencing: s;s;

choice: if, case

loop: while

Goto considered harmful, E.W. Dijkstra (1967)

first mentioned by D.ValSchorre (1962)

### design

-------

design is a heuristic process, not an algorithm

heuristic: often works but sometimes doesn't, never works first time

we should always have to iterate

"wicked problem"

you don't know statement of problem until you've solved most of it

managing complexity, both essential and accidental

keep both to a minimum to get it work

Emacs bug

----------

C-x 8 RET LATIN SMALL LETTER A WITH ACUTE RET

C-x 8 RET 201C RET

but what about BED, which is "bed" and also a valid hexadecimal number

had to find the solution first before Eggert figured out the problem

## important design concepts

--------------------------

**(1) aspects**

if you have something of a concern about the design scattered all over code

memory leaks, allocating memory all over the place

cache control

user authentication

internationalization

**(2) modularity**

redo your process or your software or both so that you can isolate these

aspects/concerns into a small set of modules

suppose we have 1 line of code/module, then code can;t be understandable

should pick appropriate size for a module

**(3) information hiding**

modules should not see each others' information

ensures functional independence

**(4) abstraction**

build higher-level layers with lower-level layers

**(5) refinement**

= anti-abstraction

**(6) patterns**

common refinement methods

**(7) refactoring**

you messed up the modularity, fix it

# design patterns

high level informal relationship among

context -> problem + solution <- design force

**e.g. factory method pattern**

context: lots of related classes need to create objects

not necessary the same classes

problem: standard framework to let each class decide on its own what

object to create (anytime)

solution: static method of a class C, returns C

but actual object may be a subclass of C

design force: creation via inheritance

relatively simple; often evolves into fancier pattern

perhaps evolve into protocol pattern (prototype for delegation)

"hash-consing"

(hash-cons a b)

Java, C++ doesn't support this

### categories of design patterns

------------------------------

**(1) creational**

factory, prototype, singleton, abstract factory

**(2) structural**

integrate existing into larger structures

container, adaptor, pipes & filters

**(3) behavioral**

communication & responsibility

visitor, event listener, interpreter, model view controller (MVC)

model: entity, "business classes", modeling what's in user's head

view: boundary, manages how to present business to users (UI components)

controller: glues together model + view, manages units of work

### class + interface issue

------------------------

\* beware erosion of abstraction

can often occur when cost of refactoring is too high

may be in environment where you have to talk to boss to add class

**\* beware if**

your class name is a verb

has only actions

your class has no methods (structs in C++)

no actions

your class does everything

not taking classes seriously

#### class design suspicions

\* be suspicious if you see:

repeated code in subclasses

base classes with just one subclass

base classes with > 10 subclasses

overriding methods that do less than what parent does

more than 7 levels of inheritance

multiple inheritance

validation tests whether code matches requirements

may be hard in practice because of communication problems

verification tests whether system works correctly

**configuration management**

-------------------------

baseline

there can be many baseline versions

may have many components, each comes in a version

have to pick version for each component

this is a baseline

codeline

line of development of individual components

make sure it is reliable so we can build future systems on it

codelines aren't actually 'lines',

but rather acyclic graphs because of branching

**version control**

----------------

not just code but also all aspects for program

requirements

tests

this is often broken by developers

pessimistic clashes will be bad if two people work on same module

involves locking to avoid race conditions

works better with few developers & many modules

optimistic let two people edit same module, it probably works

if collisions happen, we do merging

do not avoid races, but can merge changes to resolve races

works better with many developers & few modules

centralized one repository, must change this repository

change does not exist until you change central repository

database-oriented

used by CVS ...

**distributed**  no central repository

each developer has their copy of repository

copies are not necessary in sync with each other

used by git

problems in this approach

copies are not synched, merging is needed to synchronize

the patch becomes your programming element

what about patches for patches

Distributed revision control (DRCS) takes a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository.[[2]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control#cite_note-2) Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer. This results in some important differences from a centralized system:

* No canonical, reference copy of the codebase exists by default; only working copies.
* Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.[[3]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control#cite_note-OSullivan-3)

Rather, communication is only necessary when pushing or pulling changes to or from other peers.

* Each working copy effectively functions as a remote backup of the codebase and of its change-history, providing natural protection against data loss.[[3]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control#cite_note-OSullivan-3)

git log timestamps are not in time-date order

property of being distributed and de-centralized

can import patch that was dated earlier than when patch was applied

properties of version control

isolation

undo/revert

update (apply patch)

logs

backups - are key! test them!

VCS Terminology

**Baseline**

An approved revision of a document or source file from which subsequent changes can be made. See baselines, labels and tags.

**Branch**

A set of files under version control may be **branched** or **forked** at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.

**Change**

A **change** (or **diff**, or **delta**) represents a specific modification to a document under version control. The granularity of the modification considered a change varies between version control systems.

**Change list**

On many version control systems with atomic multi-change commits, a **changelist**, **change set**, or **patch** identifies the set of **changes** made in a single commit. This can also represent a sequential view of the source code, allowing the examination of source "as of" any particular changelist ID.

**Checkout**

A **check-out** (or **co**) is the act of creating a local working copy from the repository. A user may specify a specific revision or obtain the latest. The term 'checkout' can also used as a noun to describe the working copy.

**Commit**

A **commit** (**checkin**, **ci** or, more rarely, **install**, **submit** or **record**) is the action of writing or merging the changes made in the working copy back to the repository. The terms 'commit' and 'checkin' can also used in noun form to describe the new revision that is created as a result of committing.

**Conflict**

A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must **resolve** the conflict by combining the changes, or by selecting one change in favour of the other.

**Delta compression**

Most revision control software uses delta compression, which retains only the differences between successive versions of files. This allows for more efficient storage of many different versions of files.

**Dynamic stream**

A stream in which some or all file versions are mirrors of the parent stream's versions.

**Export**

**exporting** is the act of obtaining the files from the repository. It is similar to **checking-out** except that it creates a clean directory tree without the version-control metadata used in a working copy. This is often used prior to publishing the contents, for example.

**Head**

Also sometime called **tip**, this refers to the most recent commit.

**Import**

**importing** is the act of copying a local directory tree (that is not currently a working copy) into the repository for the first time.

**Label**

See **tag**.

**Mainline**

Similar to *trunk*, but there can be a mainline for each branch.

**Merge**

A **merge** or **integration** is an operation in which two sets of changes are applied to a file or set of files. Some sample scenarios are as follows:

* A user, working on a set of files, **updates** or **syncs** their working copy with changes made, and checked into the repository, by other users.[[6]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control#cite_note-svnbook-merge-6)
* A user tries to **check-in** files that have been updated by others since the files were **checked out**, and the **revision control software** automatically merges the files (typically, after prompting the user if it should proceed with the automatic merge, and in some cases only doing so if the merge can be clearly and reasonably resolved).
* A set of files is **branched**, a problem that existed before the branching is fixed in one branch, and the fix is then merged into the other branch.
* A **branch** is created, the code in the files is independently edited, and the updated branch is later incorporated into a single, unified **trunk**.

**Promote**

The act of copying file content from a less controlled location into a more controlled location. For example, from a user's workspace into a repository, or from a stream to its parent.[[7]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control#cite_note-7)

**Repository**

The **repository** is where files' current and historical data are stored, often on a server. Sometimes also called a **depot** (for example, by SVK, AccuRev and Perforce).

**Resolve**

The act of user intervention to address a conflict between different changes to the same document.

**Reverse integration**

The process of merging different team branches into the main trunk of the versioning system.

**Revision**

Also **version**: A version is any change in form. In SVK, a Revision is the state at a point in time of the entire tree in the repository.

**Ring**

**Share**

The act of making one file or folder available in multiple branches at the same time. When a shared file is changed in one branch, it is changed in other branches.

**Stream**

A container for branched files that has a known relationship to other such containers. Streams form a hierarchy; each stream can inherit various properties (like versions, namespace, workflow rules, subscribers, etc.) from its parent stream.

**Tag**

A **tag** or **label** refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number. See baselines, labels and tags.

**Trunk**

The unique line of development that is not a branch (sometimes also called Baseline or Mainline)

**Update**

An **update** (or **sync**) merges changes made in the repository (by other people, for example) into the local **working copy**.[[6]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control#cite_note-svnbook-merge-6)

**Working copy**

The **working copy** is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy, hence the name. Conceptually, it is a *sandbox*.

system building

----------------

development platform (record this in version control)

build

target

automation is key

shouldn't require many by-hand building

**automating builds**

------------------

sh - Lisp - chef

series of commands to execute

special case of 'make'

this is easier to debug though (conventional)

make [-j8]

dependencies

directed acyclic graph, explore the graph in parallel

different outputs every time we build

\* project specific

autoconf

configure.ac (100 lines) --> configure (50000 lines)

automake

Makefile.am (100 lines) --> Makefile (10000 lines)

testing

--------

testing is unnatural!

goal - success: find bugs!

you typically cannot use tests to prove absence of errors

exhaustive testing only works in small cases

bool not (bool b) { return !b; }

int add (int a, int b) { return a+b; }

testing itself does not improve software

it only exposes weaknesses

tempting to under-test

we should test everything!

code

specs

design

tests [!]

testing is a strategy that you can apply to everything

test-first programming

when to test

late

early

often/always

test driven development (TDD)

+ unit testing

+ converage

+ don't need debugger! (GDB) (some truth to this)

small changes -> unworking program (easy to find bug)

+ improves documentation

(1) the test case are the documentation!

(2) the act of writing down tests tells you what to write in documentation

- doesn't scale well

- doesn't work for graphical user interfaces (GUI)

- doesn't work well for non-deterministic programs

TDD originally was designed for single-core machine programs

coming from underlying system, multi-core races

GUI can be categorized as non-deterministic

**test tools**

-----------

**test data generators**

exist to provide test data for other programs

**project-specific scaffolding**

take down after building is built

supply tests

build the tester to generate tests

**coverage monitor**

$ gcov on SEASnet

run you program and will show how many times each line of code got executed

**platform monitors**

change platform in simple/complex way and see if it passes the same tests

$ gcc -m32 (generate code for x86 instead of x86-64)

**records + logs**

log all tests you run

keeps a test database that keeps track of all passed/unpassed test

along with all its parameters and return values

IEEE standard terminology (withdrawn)

--------------------------------------

error mistake that developers make in their heads

fault manifestation of error, mistake in the code due to error

failure fault gets triggered in production and user sees it

error -> fault -> failure

error <- fault <- failure (debugging process)

Should be only when project should be controlled

consistency

honesty

inclusion

respect

**project management danger signs**

--------------------------------

if you see such danger signs, it may be time for you to leave

\* managers and developers avoid best practices

you know you should do test-first development but people don't want to test

if there is a general tendency to avoid this, then there is a problem

\* sponsorship was lost or never present

should have backing for this project (inside or outside)

but sponsor ran out of money or person left the company

\* project lacks people who have right skills

database administrators are working on user interface redesign

\* users don't want the software

annoying software (ads, security software)

\* business needs change

needs changed completely

idea for funding is no longer supported

different division argue with each other

VA Linux (huge boost on first day)

\* unrealistic deadlines

promised that software has to work by deadline

\* chosen technology changes

Android -> iOS

\* changes are managed poorly

ship out new release but forgot to inform customers about changes

\* product scope isn't defined

trying to nail down requirements but haven't figured out whether a

particular feature belongs to this project or another

take fuzzy area and figure out sharp line for project

\* developers don't understand the customer needs

writing code, given spec, but don't get why it's there

think you understand customer need but actually don't

# solution elements for software project management

--------------------------------------------------

(1) people

good staff + good environment

big enough screens in work station

fit dev roles to personalities

- outgoing (love people!)

talk to customers, negotiate

manage a project

have the best correlation with software quality

people skills are important in building high quality software

- task oriented (love to program/building stuff!)

- self oriented (what's in it for me?)

- neuroticism (how well am I doing?)

- openness (open to new ideas)

- argreeableness (how like to agree with others)

-> job satisfaction

- conscientiousness (treat tasks seriously)

-> job satisfaction

- autonomy

- meterdependency

- conflict

provide motivation, coordination, organization into teams

toxic teams (weakest link dominates)

jelled teams (people working together, strong links fill in weak links)

people have different productivities

order of magnitude difference in productivity

5:1 ratio, even in large teams

25:1 ratio for individuals

Pareto principle (80-20 rule)

have 80% contribution from 20% developers

hire the 20% of the developers

find the motivation the other 80% need

Maslow motivation hierarchy

----------------------------

self-realization

esteem needs

social needs

safety needs

safety needs "do better or you're fired"

simple, crude, hopefully not needed

not that effective

get worried, apply funny tricks to seem to do better

social needs social networking, likes meetings

we have cities so people can meet with each other

esteem needs recognition by peers that you're doing good work

get the corner office, better pay, awards, nice parking

self-realization self-help, responsibility

realize that they can develop by themselves

(2) product

\* bound the scope

huge part of product management

draw line between product and non-product

\* decompose the problem into subproblems

draw line between subproducts

(3) process

organize and control by which your software is being developed

easier than people management

\* select appropriate process model

don't choose agile method for air-traffic control

\* adapt the generic model to specific environment + projects

\* WWWWWHH (Boehm), also W^5HH

who what when where why how how much

if these aren't answered then you don't know how process works

(4) product

\* understand the problem

get the right team

set realistic and clear objectives/goals

\* minimize turnovers

let the team do the right thing

focus on bottom-up process management

emphasize quality

start this from first day and not at the end

\* measure progress

need to know what you're doing and how well you're doing it

change management

metrics

\* keep it simple (KISS)

too easy to complicate things in software

avoid risk

simplicity to avoid risk

focus on complicated/risky areas

\* review what went wrong/right

don't just fire the project and forget!

make it better next time

scaling issues

---------------

O(N^2) communication overhead

as project scales, things working in small project starts failing

probability that a random chosen programmer in the US

will be working in a project of this size

1-3: 5%

4-10: 15%

11-25: 15%

26-50: 15%

50+: 50%

productivity | \

kLOC/year | `

| `

| `

| `

| -

| -

|

|

+-----------------------------

project size (kLOC) (on log scale)

# Risk Management

----------------

~= estimating variance in project cost

how to estimate variance?

--------------------------

risk table

risky event probability P impact I (qualitative)

------------------------------------------------------------------------

lead developer leaves 0.05

stakeholder vanishes

over AWS auota

impact on qualitative scale

1. catastrophic

2. critical

3. marginal

4. negligible

if impact is a cost then we can compute (P\*I)

system testing

---------------

requirement most planning

harder to change than simple software testing

recovery testing

for a fault-tolerant system or highly-available system

make random components fail

**security testing**

get penetration teams to try to break into it

social engineering is the best way to break into system

**stress/performance testing**

put it under large load to make sure it behaves correctly

best done at entire system level

need proper instrumentation

A \* SIZE^B \* M

---------------

B: on a scale of [0,5]

add these together, divide by 100, then add 1.1

risk analysis?

development flexibility

preceentedness

team cohesion

process maturity

M: on scale of [1,6]

personnel capability

personnel experience

reuse required

platform difficulty

schedule

support facilities

# Project Planning

-----------------

(1) resource allocation

have to give resources to a project to get it work

this is 'people' in a software project

large company: involve drafting people to get the job done

small company: use all you have

(2) scheduling

when to do what

as it scales, it gets complex

unrealistic schedules

are the project manager's faults

although tempting to come up with schedule to make customer happy

learn to resist the unrealistic schedules

avoid these schedules by negotiating with the customer

cost models are what we use in negotiations

learn to redo cost model as the project goes on

(3) cost estimation

resource allocation + scheduling = cost estimation

in practice, project planning is rarely disciplined

it is a interwined consideration

order of the 3 will not be the same

may have cost then determine resource + schedule

**typical components of a development plan**

-----------------------------------------

(1) team organization

who will work on project and how they will collaborate

(2) risk analysis

come up with idea on which parts of project are risky

(3) resource requirements

not just software but also hardware and networking, etc

(4) work breakdown

unless you have simple project, you have to split up the work

among developers to do it together

come up with tasks, milestones, deliverables

(5) schedule

(6) reporting mechanism

usually forgotten

crucial to keep track of what work has been done

what problems to worry about

hierarchical

probably the most efficient way to organize software development

not always needed for small projects

synchronous

have a lot of people working in parallel

glue it together in the end and it works

minimize effort coordinating

maximize effort getting work done

has problem with scaling

agile

starts up synchronous

then teams self-organize into small hierarchical trees

# Organizational Issues

-------------------------------

lines of communication

among whom?

email? IRC?

don't overuse communication

be efficient in communicating

selecting members

pick good members for team

have complementary personalities

technical competence

interviewing skills are essential

**risk assessment**

----------------

need to keep doing as project moves on

* preliminary

done during requirements gathering

haven't decided on implementation

have to do this to know whether to commit to project

most important to get right

* life-cycle

done during system development

know the implementation but don't know the usage fully

can predict how the software will be used

* operational

done during production

software is run with real users

goal is to resond to production issues

quick response to issues as they arise

**risk categories**

----------------

instead of looking at how the risks are going to hit you

look at where risks are coming from

\* business

may be driven out of business by large companies

Google competes!

\* product

something goes wrong inside software you've written

garbage collector doesn't scale

\* project

have nothing to do with actual product you're building

leading developer leaves

risks can overlap although we may think they're independent

sample resource issues

-----------------------

\* buy or build?

should you reuse software or simply build it yourself

major issue

\* inhouse or outsource?

write it in your own team or outsource it to other teams

\* different combinations of the two above

20 projects -> 2^20 combinations

how to prune intelligently

defining your tasks

-------------------

class-based

package-based

tasks nest

tasks have dependencies

dependency graph

we can generate a schedule with this graph

critical path analysis

scheduling time -> Gantt chart

when you're late

-----------------

\* add resources to project

\* add time (delay due date)

\* decrease scope

how to price a software project

--------------------------------

C costs (usually unknown)

P profit

-----------------------------

C+P done

we typically guess the cost or adjust the software to match our budget

adjust software to match budget (relies on trust)

general rules for project estimation

-------------------------------------

average large project is 1 year late and 100% over budget

(1) keep + use records of previous projects

(2) use several methods, and cross-check

(3) assume things will go wrong during development

(4) if possible, develop + estimate incrementally

factors affecting cost

-----------------------

(1) size

$ wc # number of lines

length of API

helpful to estimate size before writing the code

count statements instead of comments

;;;;;;;; is totally valid in C

(2) complexity

(3) requirements analyst capability

requirements analysis is among the hardest parts of software analysis

(4) programmer capability

(5) CPU time / storage constraints

typically bites during embedded application

(6) personnel turnover

the larger the project, the more likely this is going to happen

(7) platform variability

affect cost model quite a bit

(8) team experience

application area

language and tools

production platform

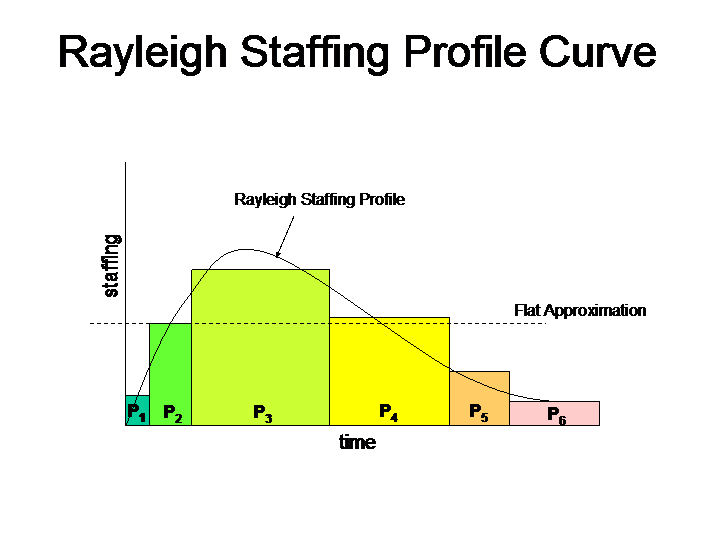
(9) use of software tools

do we have good static tools that will help find bugs before runtime

(10) location / communication

(11) motivation

# \* Putnam Norden Rayleigh (PNR) curve

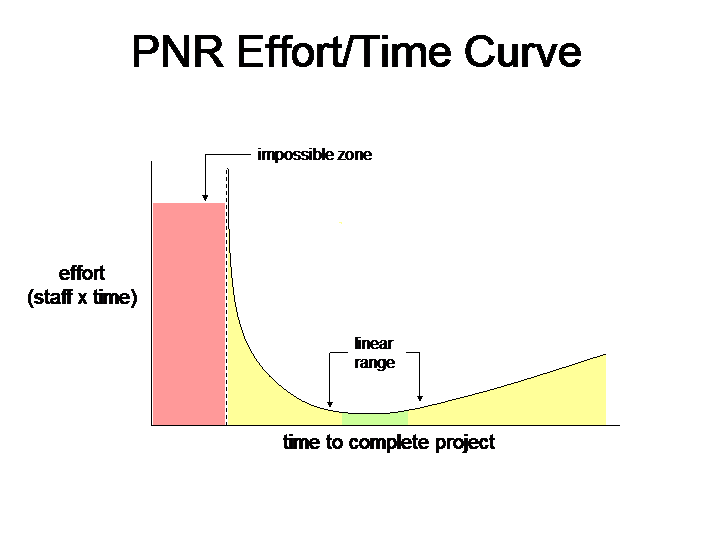


We can attempt to force a flat staffing profile on such a project, but it will result in considerable waste:

* In the early stages, there will be people standing around with nothing to do, because the requirements and architecture development process is more gated by time than by labor.
* In the mid stages, work may be delayed by not having enough people available to perform all of the defined tasks.
* In the late stages, over-staffing will not bring the project to completion any sooner, because the collection of feedback and shaking out of final problems is (again) more gated by time than available labor.

If we want to accomplish a project efficiently, we much understand the distinct phases, along with their respective skill and activity requirements. The notion that we can prepare an estimate in staff months and predict completion time by choosing a staffing level and dividing that into the total project size is a canard.

## Time/Effort and the PNR Curve



The clear message here is that every project has an optimal staffing level, and (correspondingly) an optimal time in which it will be completed. Going significantly above or below the optimal staffing level will reduce work efficiency ... and there may be a point beyond which adding people actually delays the project.

This curve breaks down into four zones:

1. **An impossible zone**  
   The project cannot be accomplished in less time than this, no matter how many people are applied to the problem. This situation is classically summarized as: "Nine women cannot have a baby in one month".
2. **An "Haste makes waste" zone**  
   Adding people does accelerate delivery, but not in proportion to the added effort. Each additional person added to the project lowers our productivity (they have to be trained, more time goes into communication and coordination, more misunderstandings). This is a very inefficient way to operate.
3. **A linear range**  
   This is the range of efficient staffing, and within this range it is possible to trade man-power for time, or vice versa.
4. **An under-staffed/over-staffed zone**  
   This curve does not yield completion time as a function of staffing, but merely shows the relationship between staffing level and completion times. What we can clearly see on the right of the curve is that productivity is dropping. Why might this be?
   * If the project is critically under-staffed, productivity will suffer because there aren't enough people to deal with the problems.
   * if the project is greatly over-staffed communications overhead will reduce efficiency and misunderstandings will create problems and result in wasted work.S

# Statistical Testing

**statistical testing**: testing randomly with undeterministic tests

figure out what real-world input would look like

and test these inputs

works best where we know what use cases are

sort program, we count out-of-order numbers and then

generate similar test cases to test that

**operational profile**: description of what inputs should look like

uncertain: haven't built up what input would look like

low quality of statistical testing

since we do not have the user input profile yet

**experience-based testing:** come up with test cases for unit, integration tests

use experience to come up with testing

penetration testing: have a team who is really good in breaking into systems

form of experience-based testing

special case of experience-based testing

4. give 3 basic methods for risk reduction in safety engineering

(a) avoid the risk, make it impossible for things to "catch fire"

(b) detect and remove, catch problem and fix it

(c) tolerate, ignore the risk and keep going

ORCA: election day application

Romney campaign

secret, didn't want to expose to opponent (imitate or counter)

data mining system designed to run in cloud-like system

clients running in election worker cellphones

focus limited resources on aspects that will do elections most good

apps on cellphones + web service + data mining

out-of-house development

reported to a different technique

rolled out at 0600 EST 2012-11-06

got Distributed Denial of Service Attack (DDoS) on release day

dependability

--------------

can rely on the system working on arbitrary real-world conditions

even under adverse real-world conditions

even if you have adversaries

assume universe is subtle

* security (for later)
* safety
* likelihood that software will not damage users either directly or indirectly
* reliability
* likelihood that software will match expectations
* availability
* likelihood that service will be up
* resilience
* likelihood that software will resist and recover from damage
* maintainability
* likelihood that software will stay working after change
* reparability
* how fixable when buggy
* user error tolerance

**dependability engineering**

--------------------------

**Security Assumptions**

---------

assume universe is malicious

paranoia is almost required here

security is a more difficult problem

often the most important

can trump functionality

bad P.R. if it fails

**cost ($) vs. dependability (0-1):** curve has asymptotic growth

choosing position on curve is a hard design decision

specifying dependability and security

--------------------------------------

often requirements are the hardest part

easy to think you're done because you filled out all forms, but you're not

over-specify dependability

techniques

-> identify risks

-> analyze & prioritize risks

-> decompose the risks into their root causes

-> reduce them

high failure rates are sometimes ok

failure rate may depend on other factors

may need to specify different failure rates for different causes

**safety terminology**

--------------------

accident unplanned event causing injury

hazard condition that might lead to an accident

damage measure of cost of accident

hazard severity what's the worst that can happen?

risk probability of an accident

**security terminology**

--------------------

exploit unwanted event causing loss of privacy, etc.

vulnerability condition that might lead to an exploit

exposure system is available to be attacked

attack attempt to exploit

threat vulnerability that might be attacked ('misuse case')

control mechanism to defend a vulnerability

**reliability terminology**

------------------------

MTBF mean time between failure

MTTF mean time to failure

MTTR mean time to repair

AVAIL = MTTF/MTBF >= 0.999 (3 9s)

POFOD = probability of failure on demand

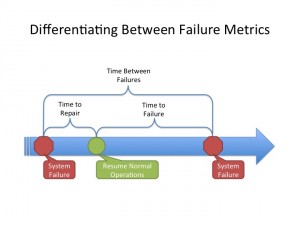
ROCOF = rate of occurrence of failure = 1/MTTF

### **Definition of a Failure**

I suppose it is wise to begin by considering what exactly qualifies as a “failure.” Clearly, if the system is down, it has failed. But what about the system running in degraded mode, such as a raid array that is rebuilding? And what about systems that are intentionally brought off-line?

Technically speaking, a failure is declared when the system does not meet its desired objectives. When comes to IT systems, including disk storage, this generally means an outage or down time. But I have experienced situations where the system was running so slowly that it should be considered failed even though it was technically still “up.” Therefore, I consider any system that cannot meet minimum performance or availability requirements to be “failed.”

Similarly, a return to normal operations signals the end of downtime or system failure. Perhaps the system is still in a degraded mode, with some nodes or data protection systems not yet online, but if it is available for normal use I would consider it to be “non-failed.”

[](http://static.fosketts.net/wp-content/uploads/2011/07/Slide2.jpg)MTBF is the sum of MTTR and MTTF

### **Mean Time to Failure (MTTF)**

The first metric that we should understand is the time that a system is not failed, or is available. Often referred to as “uptime” in the IT industry, the length of time that a system is online between outages or failures can be thought of as the “time to failure” for that system.

For example, if I bring my RAID array online on Monday at noon and the system functions normally until a disk failure Friday at noon, it was “available” for exactly 96 hours. If this happens every week, with repairs lasting from Friday noon until Monday noon, I could average these numbers to reach a “mean time to failure” or “MTTF” of 96 hours. I would probably also call my system vendor and demand that they replace this horribly unreliable device!

Most systems only occasionally fail, so it is important to think of reliability in statistical terms. Manufacturers often run controlled tests to see how reliable a device is expected to be, and sometimes report these results to buyers. This is a good indication of the reliability of a device, as long as these manufacturer tests are reasonably accurate. Unfortunately, many vendors refer to this metric as “mean time between failure” (MTBF), which is incorrect as we shall soon see.

Note too that “MTTF” often [exceeds the expected lifetime](http://www.faqs.org/faqs/arch-storage/part2/section-151.html) or usefulness of a device by a good margin. A typical hard disk drive might list an MTTF of 1,000,000 hours, or over 100 years. But no one should expect a given hard disk drive to last this long. In fact, [disk replacement rate is much higher than disk failure rate](http://db.usenix.org/events/fast07/tech/schroeder/schroeder_html/index.html)!

### **Mean Time to Repair (MTTR)**

Many vendors suppose that repairs are instantaneous or non-existent, but IT professionals know that this is not the case. In fact, I might still be a systems administrator if it wasn’t for the fact that I had to spend hours in freezing cold datacenters trying to repair failed systems! The amount of time required to repair a system and bring it back online is the “time to repair”, another critical metric.

In our example above, our flaky RAID array had an MTTF of 96 hours. This leaves three days, or 72 hours, to get things operational again. Over time, we would come to expect a “mean time to repair” or “MTTR” of 72 hours for any typical failure. Again, we would be justified in complaining to the vendor at this point.

Repairs can be excruciating, but they often do not take anywhere near as long as this. In fact, most computer systems and devices are wonderfully reliable, with MTTF measured in months or years. But when things do go wrong, it can often take quite a while to diagnose, replace, or repair the failure. Even so, MTTR in IT systems tends to be measured in hours rather than days.

### **Mean Time Between Failures (MTBF)**

The most common failure related metric is also mostly used incorrectly. “Mean time between failures” or “MTBF” refers to the amount of time that elapses between one failure and the next. Mathematically, this is the sum of MTTF and MTTR, the total time required for a device to fail and that failure to be repaired.

For example, our faulty disk array with an MTTF of 96 hours and and MTTR of 72 hours would have an MTBF of one week, or 168 hours. But many disk drives only fail once in their life, and most never fail. So manufacturers don’t bother to talk about MTTR and instead use MTBF as a shorthand for average failure rate over time. In other words, “MTBF” often reflects the number of drives that fail rather than the rate at which they fail!

## Objectives Reliability Testing

The main objective of the reliability testing is to test software performance under given conditions without any type of corrective measure using known fixed procedures considering its specifications.

The secondary objectives of reliability testing is:

1. To find perceptual structure of repeating failures.
2. To find the number of failures occurring in a specified amount of time.
3. To find the mean life of the software.
4. To discover the main cause of failure.
5. Checking the performance of different units of software after taking preventive actions.

**safety requirements**

--------------------

primary (e.g. brakes?)

vs.

secondary (e.g. both brake lights?)

analysis of risks from requirements

------------------------------------

**fault-tree analysis**

try to categorize all bad things that can happen

tree where leaves represent accident or hazard

nodes represent events that leads to accident

tree node will be AND/OR

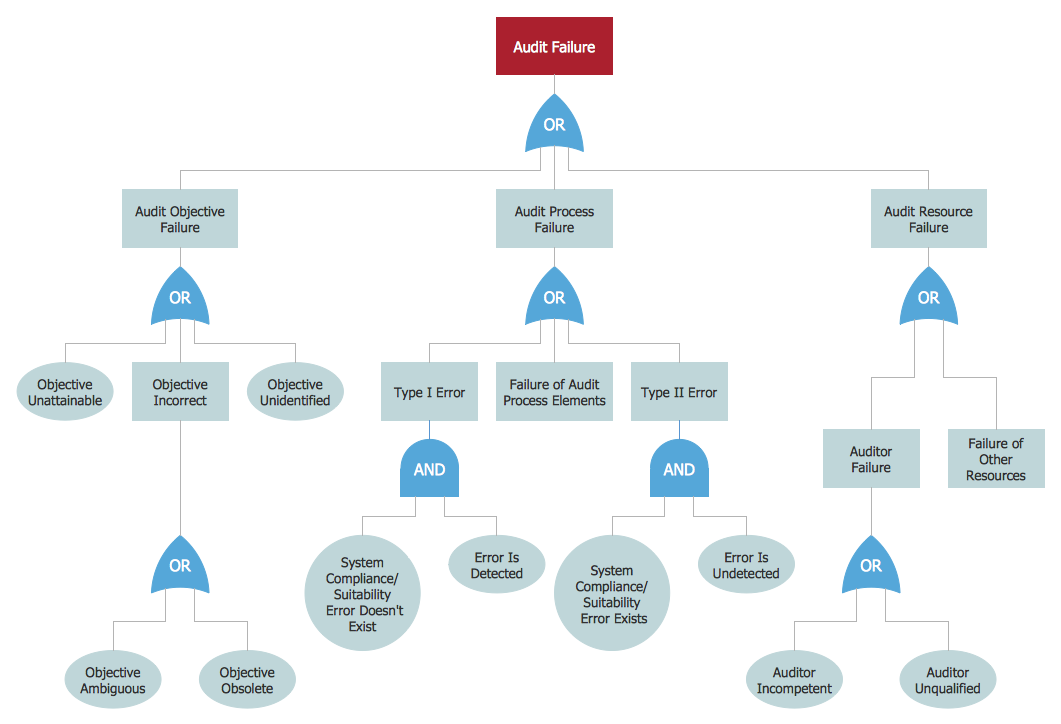
organize the tree to capture dependencies of the accidents

if the analysis works well, you can assign probabilities to branches of tree

not always clear (correlation between events)

can calculate probability for hazard analysis table

this makes hazards/accidents easy to explain



dependability & architecture

-----------------------------

security kernel + protection systems

minimalize amount of software that have to be secure and put in kernel

architecture that makes it easier to build dependable systems

self-monitoring architectures

log the stuff that try to cross the walls

e.g. Firefox Health

N-version programming

create N versions for the same program

vote for the best and the majority rules

paying N times (or more) as much for the same program

have discrepancies between engineers in different versions

# Legacy Systems

legacy systems have both code and people

code is mysterious to programmers

replacing legacy system doesn't mean duplicating what code does

but also have to figure out what people do

bad smells for refactoring

data clumping

duplicate code

speculartive generality

security & dependability

-------------------------

hard to retrofit security

security risk assessment (how worried about security)

identify assests (things to protect)

financial, phyiscal

prioritize assets (give value to asset)

identify exposures

identify threats

attacks

estimate cost of each attack

specify controls + feasibility

**security design guidelines**

---------------------------

have an explicit security policy, and use it

providing scope for security, which is a problem in software development

there's always more you can do with security

you should know where to stop

defense in depth

used in football, military

don't rely on one technique to defend system

have multiple levels of defense so that attackers have a hard time

fail securely

robust, dependable system

deal with component failure

a good chunk of code deals with failures, exception handling

fail in such a way that you still have security policy

exception handlers are big problems in security

balance security and usability

a brick is very secure

need software that actually works

security is not end goal that dominates everything else

people are willing to give up some security for features

log actions

log actions coming from the outside world (users)

this is brute force debugging applied to security

`printf' statements sent to logs

doing this under assumption that program will fail, mess up

reduce risk via redundancy and diveristy

have multiple identical copies of the password server

does not necessary reduce risk

specify imput format

people don't do this all the time

fairly mundane but is often forgotten

compartmentalize assets

related to defense in depth but different

put several layers around valuable assets

break up assets into pieces

attacker will not get all information with breaking in at once

have more database systems, although harder to manage

design for deployment

when a new version of program comes out, want this program secure

no configuration, no simple default root password (`admin')

design for recovery

when system gets breached, need a way to recover into secure system

without posting root password `on the wall'

**Security Testing**

-------------------------

how to make sure that system is secure?

tiger teams, test cases

**procedures for developers**

don't think security as something built into product

security is also a property of development process

what to do if a developer goes bad?

log developer actions

reuse security guidelines above and apply to development process

**dependable software processes**

you not only build dependable software but also using dependable processes

convince customer / developer / regulator that software is dependable

can involve the following

test plan + management

need to document these

static analysis

describe tools and run the analysis, then list the results

formal inspections

formal code reviews, etc.

formal specifications

write down specifications for code in formal language, based in logic

system modeling

provide model / simulation of system in report

review requirements + management

management => version control

go back and look how requirements evolved with time

goal of the process

auditable

redundant

provide multiple ways on making sure system is dependable & secure

robust

assume development steps will not be done correctly

want system to recover well from failure

**dependable programming**

information hiding

private class members not to make it hard on users

provide interior security walls

keep part of implementation private so not visible to other parts of code

can isolate walls into the code, fewer avenues of attack into system

input validity checking

specify format

constantly checking validity of input data

exception handling

commonly used for dependability

resume execution after some part of computation fails

promotes redundancy

omit tricky stuff

floating point

rounding errors

computing with values that are close to what you want but not exactly

descrepancy can lead to unreliability

goto's

generally a bad idea to use them

pointers

null pointers

dangling pointers (freed memory)

imposter pointers (point to memory not supposed to point at)

subscripts

arrays and indexing into an array

can have dangling index in array

all problems with pointers can exist with subscripts

new

creating object is trouble, similar to pointers

recursion

dangerous

understood by only 25% of programmers

structure hard to understand

parallelism (threads)

source of enormous class of bugs

race conditions / deadlock

signals / traps / interrupts

asynchronous function calls

can break a lot of things with signals

aliasing

can get aliasing without pointers

it is helplessly confusing

recipe for bugs

inheritance

breaks incapsulation

don't have walls between parent and children

another recipe for bugs

polymorphism

also very comfusing

debugging (M.W. Wilkes)

------------------------

can take half the development time

20:1 ratio of performance between experienced & inexperienced developers

connotation of dread, get little reward in publishing papers on debugging

this is the least understood part of software development/construction

how not to debug (classic mistakes)

(1) assume the error is somewhere else in the system

`printf is broken!'

printf("%d\n", i) // prints out garbage where i = 2

actually defined: long i;

(2) fix the problem without completely understanding it

you're in a hurry

you despair in ever understanding the problem

how to debug

(1) make the bug reproducable

make the bug happen whenever you want it to happen

it may be difficult, especially as we gain parallelism

and undeterministic programs

bug doesn't always happen in the program

get rid of the nondeterministic

$ gcc -fdeterministic foo.c # doesn't exist

[!] except that there is no `deterministic' flag

how to make deterministic

turn off multi-threading/multi-processing

$ make -j # builds in parallel, remove it

specify the seed for the random generator

can be really helpful to run in controlled environment

clear all variables before startup

reset the clock

use a clock permanently set to 1970

or just set the time

(2) locate the fault

doesn't have an algorithm

use the scientific method

form a hypothesis about the fault

test the hypothesis

painstaking process

(3) fix the defect

typically a relatively easy step compared to the above

(4) test the fix

common for debugging to involve adding new test cases

this prevents the bug from going undetected in the future

(5) look for similar errors

bugs travel in clumps

there's probably more than one floating around

useful heuristics

------------------

(1) look at code that is changed recently

$ git bisect

|-----------------------|-----------------------|

ok ok ng

do binary search for the defect

(2) prefer O(logN) techniques to search for defects

(3) budget your time

try another technique

rewrite the program! (only if have decent budget)

(4) one fix at a time

general advice in software development

can test small pieces

apply multiple patches where each patch fixes one bug at a time

(5) keep out of ruts

be flexible, jump around a lot

try looking at things a different way

`part-time debugger', come back to bug with a fresh mind

typically more efficient

software evolution

-------------------

deliver the software (people are using)

apply a change (delta) to the software

maybe your business changed

maybe your technology changed

bugs

planned vs. agile

all plans for development applies to evolution as well

new phase: understanding existing software

evolving pre-engineered old code

can be significant part of your work

gain maximum amount of understanding for minimum amount of work

how to efficienty grok existing software

look at its documentation

not always available

source code translator

translate to other language and run the translated code (C -> Python)

this may not work well

==============

May. 23, 2016

==============

How is Z Schema defined?

what are state variables?

what are their constraints?

what were the state transitions?

Two types of code beautifiers?

(1) changes the code, into a certain type of style

(2) doesn't change the code, gives highlighting

why haven't formal methods taken over?

---------------------------------------

first proposed in 1960s and lots of work was done in this

(1) scalability

tends to work best when spec is small

has problems scaling up to large systems

large systems can't use them & small systems don't need them

(2) limited scope (e.g. no UIs)

(3) expectations for reliability have relaxed

don't care if software is reliable anymore

in old times, software had to be reliable to process important things

(4) there are other cheaper ways of improving reliability

these other ways are cheaper than formal methods

formal inspections

hardware support (more checking, bad pointers, subscript)

object oriented information hiding

formal methods downsides

-------------------------

(1) they're hard (need an expert)

complicated

(2) notation is crucial

coming up with good notation is more important than programming

(3) easy to overformalize

once you've captured the idea to formalize

it becomes tempting to formalize out spec completely

cost goes up and may cost more than it's worth

try to formalize the important part

retain the ability to leave other parts informal to keep costs down

(4) not a substitute for competence

will not make you competent in the first place

(5) they still need comments

will still require informal comments

(6) don't guarantee code to be bug-free

formally verify code satisfies the spec, you know your code matches spec

but this doesn't guarantee that you code is bugfree, you spec may have bugs

(7) they need same software engineering effort as everything else

put formal specs under version control

have to review specs so to make sure they are the right specs

And yet -

Coverity (practical)

uses formal methods

proof-carrying code

gives answers + proof that answer is correct

checker can detect malicious behavior, check proof, proof is wrong

informal specifications - problems

-----------------------------------

(1) get contradictions

result from several meetings

write down multiple specs

discover that specs are contradictory

(2) ambiguities, vagueness, incompleteness

forget to include some aspects

all the same thing in formal specs

nailing down set of possible worlds for implementation

contradiction: empty

ambiguity: too large

spend most of effort in finding incompleteness

standard formal methods ideas

------------------------------

(1) invariant

bool expression involving all state variables

should always give true

(2) precondition

bool expression involving all state variables

should be true just before a particular state transition

(3) postcondition

bool expression involving all state variables

should be true just after a particular state transition

specification languages

------------------------

standard programming languages

tends not to work very well

don't have to be executed

they can be much higher level

sets, for all, exists

they are often tailored for specific domains

they encourage you to nail down all spec details

formal methods details

-----------------------

algebraic

typically for functional code

no state

just recursive data structures

vs

model based

typically for imperative code

state machine via state variables

invariants that have to be always true

state transitions

initial state predicate

invariant, trans

==> invariant

initial state, trans1, trans2, ... , transN

==> invariant

examples

---------

signatures

List(Elem)

sort List

constructors

Create -> List

Cons(Elem, List) -> List

inspectors

Head(List) -> Elem

Tail(List) -> List

axioms

Head(Cons(V,L)) = V

Tail(Cons(V,L)) = L

behavior driven development

----------------------------

poor person approximation to 'Z'

write down a scenario in a text file

given: <precondition>

bank balance is positive

when: <state name + arguments>

withdraw N dollars

then: <postcondition>

new bank balance is N less than old one

domain specific language

read text file and generate some python code

/balance is

/{check(balance > N);}

construction tools

-------------------

(1) editing source code

Eggert uses Emacs

others like Vim, nano, Eclipse, VS, Xcode

(2) refactoring source code

add new argument to function

should be easy to change function calls to it

(3) version control

diff: compare two files

diff X Y => XdeltaY

patch: apply diff and get new file

patch XdeltaY X => Y

merge: merge two different files

diff X Y => XdeltaY diff X Z => XdeltaZ

patch XdeltaY Z => M patch XdeltaZ Y => M'

is M = M' ?

not always, there might be incompatible merge conflicts

$ (emacs) M-x vc-pull -> returned gibberish

+----+----+

source code in UTF-16 | | |

+----+----+

16 bits

git diff -> "Binary files A and B diff" (in ASCII)

emacs converted this into UTF-16, and returned gibberish

(4) code beautifiers

indent (Linux)

can lead to many problems

fight against other changes to source code, since it changes globally

javadoc (Java)

turns comments into beautiful documentation

xref

cross-referencing tool

where a name is used and defined

does auto-completion when you start typing identifier

diagram generators

traditionally, simple tools to generate flow chart for code flow

state machine diagram

lint

cc but no code compiling

instead style/correctness checking

==============

May. 25, 2016

==============

tools for debugging

--------------------

(1) google search

very good choice

(2) print statements

logging out actions

add stuff to code to output useful information through execution

highly recommended, even in production code, usually turned off

want log analysis tools for this (grep)

Apache log format

well documented

with ecosystem of tools to analyze

(3) memory analysis tools

valgrind, etc.

suspicious accesses, memory leaks

run different version (valgrind version) of the program

sometimes static analysis

(4) debuggers

GDB, step through program

avoid these if possible

breakpoints

jump ahead and stop at certain points

watchpoints

wait for program to hit a piece of data (read/write)

(5) ask a debugging expert

some are really good at this

(6) profilers

gprof program

performance instrumenting mode

generates extra code to increment counter each time code is executed

gprof looks at execution output and analyzes

$ gcc -pg

(7) project-specific tools

know that it's going to work for your particular application

Ken Thompson: chess machine

relied on scanned games, got many scanned errors

wrote special debugging tool

looked at raw data, returned possibilities with probabilties

then fed possibilities into validator to check if move was possible

build vs. buy

buy from someone else of build it yourself

sometimes it will be easier to just build your debugging tool

user interface design

----------------------

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

* Attractive
* Simple to use
* Responsive in short time
* Clear to understand
* Consistent on all interfacing screens

UI is broadly divided into two categories:

* Command Line Interface
* Graphical User Interface

## Command Line Interface (CLI)

CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

### Command Line Interface (CLI)**CLI Elements**

A text-based command line interface can have the following elements:

* **Command Prompt** - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
* **Cursor** - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
* **Command** - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

## Graphical User Interface

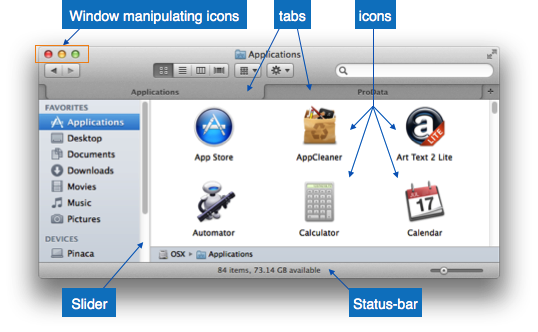
Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

### **GUI Elements**

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:

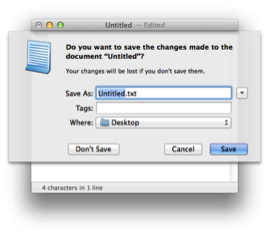


* **Window** - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.
* **Tabs** - If an application allows executing multiple instances of itself, they appear on the screen as separate windows.**Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.
* **Menu** - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.
* **Icon** - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.
* **Cursor** - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

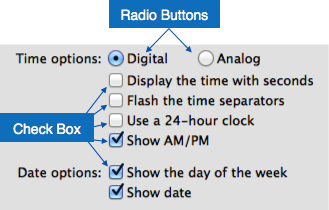
### **Application specific GUI components**

A GUI of an application contains one or more of the listed GUI elements:

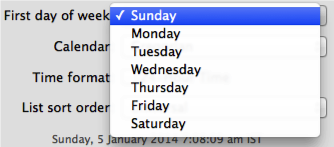
* **Application Window** - Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.
* **Dialogue Box**- It is a child window that contains message for the user and request for some action to be taken. For Example: Application generate a dialogue to get confirmation from user to delete a file.



* **Text-Box** - Provides an area for user to type and enter text-based data.
* **Buttons** - They imitate real life buttons and are used to submit inputs to the software.



* **Radio-button** - Displays available options for selection. Only one can be selected among all offered.
* **Check-box** - Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.
* **List-box**- Provides list of available items for selection. More than one item can be selected.



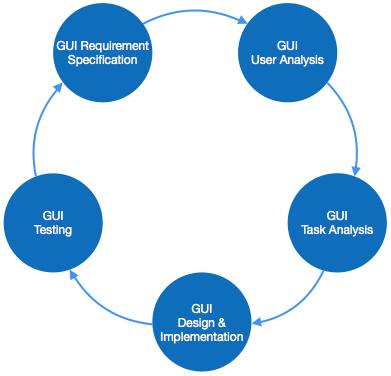
Other impressive GUI components are:

* Sliders
* Combo-box
* Data-grid
* Drop-down list

## User Interface Design Activities

There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.

A model used for GUI design and development should fulfill these GUI specific steps.



* **GUI Requirement Gathering** - The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
* **User Analysis** - The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
* **Task Analysis** - Designers have to analyze what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
* **GUI Design & implementation** - Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.
* **Testing** - GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

## GUI Implementation Tools

There are several tools available using which the designers can create entire GUI on a mouse click. Some tools can be embedded into the software environment (IDE).

GUI implementation tools provide powerful array of GUI controls. For software customization, designers can change the code accordingly.

There are different segments of GUI tools according to their different use and platform.

### **Example**

Mobile GUI, Computer GUI, Touch-Screen GUI etc. Here is a list of few tools which come handy to build GUI:

* FLUID
* AppInventor (Android)
* LucidChart
* Wavemaker
* Visual Studio

## User Interface Golden rules

The following rules are mentioned to be the golden rules for GUI design, described by Shneiderman and Plaisant in their book (Designing the User Interface).

* **Strive for consistency** - Consistent sequences of actions should be required in similar situations. Identical terminology should be used in prompts, menus, and help screens. Consistent commands should be employed throughout.
* **Enable frequent users to use short-cuts** - The user’s desire to reduce the number of interactions increases with the frequency of use. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.
* **Offer informative feedback** - For every operator action, there should be some system feedback. For frequent and minor actions, the response must be modest, while for infrequent and major actions, the response must be more substantial.
* **Design dialog to yield closure** - Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and this indicates that the way ahead is clear to prepare for the next group of actions.
* **Offer simple error handling** - As much as possible, design the system so the user will not make a serious error. If an error is made, the system should be able to detect it and offer simple, comprehensible mechanisms for handling the error.
* **Permit easy reversal of actions** - This feature relieves anxiety, since the user knows that errors can be undone. Easy reversal of actions encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.
* **Support internal locus of control** - Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
* **Reduce short-term memory load** - The limitation of human information processing in short-term memory requires the displays to be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions

special case of design

all points on how to do design all apply to user interface design

how is it different from general software design

crucial to learn the user characteristics before doing the design

learn to think like the users

some general principles for UI focus neeed

(1) reduce user memory load

assume users will forget, because they do not focus on application

\* have consistent user interface

don't have differnent ways for deleting things!

usually different designers & developers make different things

\* be compatible with existing practice

comply to the 'two-finger pinch'

can collide with above

either be consistent with system or consistent with world

\* use real-world metaphors

use from real world so they have less to learn

\* disclose information progressively

don't have to give user at first glance

if they may not want the information provided

file browser

list names

hover mouse over it and show more

open it to show even more

avoid information overload

\* make context explicit

if the user has interacted with system

they should be able to see, just by looking at the page

the context of the page, such as 'how I got here'

show 'breadcrumbs', path to how they got here

\* defaults

supply good defaults for things that users may not change

choosing defaults becomes delicate and important exercise

sometimes increase memory load

\* shortcuts

can be user-programmable

different users have different habits

so they can have their own shortcuts for doing things

sometimes increase memory load

(2) have to put the user in control

example where user is not in control

+----------------------+

| fatal error: exit? |

| +----+ |

| | ok | |

| +----+ |

+----------------------+

\* no dead ends

user interface should not hang

\* flexible

lots of options

\* undo/redo

should have undo/redo key

can try things out and go back to previous state

use redo to avoid chain of undo

A1 A2 A3 <-> A4

A1 <- A2 <- A3 <- A4

**common user interface design issues**

------------------------------------

**(1) response time**

should not have LONG response time

**(2) help**

how can user discover how to use application as they use it

usually added at last minute, and does not work well

**(3) error handling**

when user makes a mistake/system has a problem, how to let user know

what will the user see?

decide which problem are worth notifying the user about

**(4) shortcuts/commands vs. menu/mouse**

commonly known as CLI vs. GUI

prompt the user or wait for commands?

**(5) internationalization (i18n)**

error message text

horizontally & vertically for different languages

make user interface easily portable to such languages

**(6) accessibility**

make application useful who have disabilities

might have a blind user

**\* provide text alternatives for any useful non-text content**

picture on page is how user will select where to go to

<img href="foo.jpg" alt="agree to arbitration?">

if there is no alt text, screen reader will read 'foo.jpg'

**\* synchronize multimedia alternatives**

make sure all finish at the same time

\* main image/video should be clear to people with partial disabilities

**make sure text is big enough**

what about people who are red/green color blind?

**\* make all functionality doable from the keyboard**

can also talk to keyboard

**\* give users enough time**

for the speech-to-text interfaces

**\* don't create context known to cause seizures**

> 3 flashes/second

**common usability problems**

--------------------------

in intranets

(1) search

need to find out small part of larger user interface

can use search engine (Google/Bing)

sometimes cannot use search engine on intranet

can use internal search engine

some things will not be easily searchable

should have good organization for your system

(2) login

authentication/authorization

getting access to stuff you need access to

how to define usability?

-------------------------

(1) more effective

does what you want

(2) more efficient to use

can tell it what you want to do quicker than other UIs

(3) easier to learn and remember

minimizing the memory load for the user interface itself, not data

(4) more satisfying to use

hardest to nail down

more 'user-friendly', comfortable, natural

typically need market research to establish

(5) fewer and less important errors

how to measure usability?

--------------------------

sometimes it will be straightforward

e.g. errors in UI

questions to answer before measuring

(1) know what user does and what the implementation does

people want to do something at high level

decompose into smaller commands and there are different commands

(2) what is the user's context?

why would the user want to use your application

(3) what is the user's background

who's the audience for your application

(4) what is the user's need

what they want to do

questions while measuring usability

(1) how much training to use system?

training is part of usability testing

(2) does help suffice?

can users figure stuff out on their own

do you have to step in to help them (this can be considered as a bug)

(3) what kind of and how many error?

can count errors

each time user makes mistake is a success in usability testing

(4) how do users recover from errors?

what users say is less important than what they say

measure what the DO, not what they SAY!

don't partition usability testing by function

how do you know when to stop?

you don't know, you stop when the money runs out!

no this is actually not correct

need to specify the scope

suppose catch users in Los Angeles, New Orleans, New York

not useful unless application differs in regional diversity

probably useful to test different ages

# RESTful Design Principles

Here, we will outline the set of RESTful design principles that should be adhered to when creating a ‘proper’ RESTful service.

Let’s start with the basics. **What is REST**?

**REST** = REpresentational State Transfer. REST is an architectural style for network based software that requires stateless, cacheable, client-server communication via a uniform interface between components.

The primary focus of this blog post is to introduce REST along with REST terminology, REST concepts, and some simple examples describing what REST looks like in practice. As a secondary focus, I will address a topic that often confuses folks. Many folks often ask to compare REST vs SOAP. This comparison *does not* make sense. REST is an architectural style, while SOAP, like HTTP, is communication protocol. What does this mean? Well, REST and SOAP are not mutually exclusive. In theory, they can be used together. I would highly recommend against this usage, but their is nothing about the REST style that prohibits this case. During this post we’ll touch on this. We’ll also see why REST is so widely used over HTTP. All in all, the primary focus will remain an introduction to REST!

Let’s get started. First off, I introduced a number of loaded terms in my one line description of REST. These terms were *stateless*, *cacheable*, *client-server* communication, and*uniform interface*. These represent the basic principles of REST. Let’s briefly introduce these principles and their meaning within the context of REST.

## Basic Principles

**Client-Server Communication**

Client-server architectures have a very distinct separation of concerns. All applications built in the RESTful style must also be client-server in princple.

**Stateless**

Each each client request to the server requires that its state be fully represented. The server must be able to completely understand the client request without using any server context or server session state. It follows that all state must be kept on the client. We will discuss stateless representation in more detail later.

**Cacheable**

Cache constraints may be used, thus enabling response data to to be marked as cacheable or not-cachable. Any data marked as cacheable may be reused as the response to the same subsequent request.

**Uniform Interface**

All components must interact through a single uniform interface. Because all component interaction occurs via this interface, interaction with different services is very simple. The interface is the same! This also means that implementation changes can be made in isolation. Such changes, will not affect fundamental component interaction because the uniform interface is always unchanged. One disadvantage is that you are stuck with the interface. If an optimization could be provided to a specific service by changing the interface, you are out of luck as REST prohibits this. On the bright side, however, REST is optimized for the web, hence incredible popularity of REST over HTTP!

The above concepts represent defining characteristics of REST and differentiate the REST architecture from other architectures like web services. It is useful to note that a REST service is a web service, but a web service is not necessarily a REST service.

Let’s now dive into a bit more detail and discuss a variety of elements used to compose a RESTful system.

## Resource and Resource Identifier:

A key abstraction of REST is the *resource*. A *resource* can be just about anything. It can be a document or an image, an object, a collection of other *resources*, and more.  A *resource* is identify by its *resource identifier*. The *resource identifier* is often used when multiple components communicate with one another. They are able to reference specific *resources* using the *resource identifier*.

In practice, resources are *nouns*. Resources and identified by URIs  
e.g. This Car *resource* is identified by the *resource identifier*, http://www.automart.com/cars/12345

**Representation:**

Components perform actions a *resource* by applying operation provided by the component’s uniform interface. A resource is represented by its current state or its intended state (assuming the action will modify the resource in some way). This representation includes a sequence of bytes and some description of those bytes. The format of a representation is defined as its media type.

In practice, *resources* are often represented as XML, JSON, RDF, and more

**Basic Principles in Practice**

Let’s harken back to the Basic Principles section and describe how those principles can be applied in practice.

**Client-server**

HTTP is a client-server protocol. Why not use it with REST. Check!

**Uniform Interface**

REST is optimized for the web, thus HTTP is typically used. HTTP defines GET,POST, PUT, DELETE. Woah! That meets REST’s requirement to provide a *uniform interface*for components*.*

**Cacheable**

HTTP provides a cache control mechanism. See [here](http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html). Dang! HTTP just filled another REST requirement

**Stateless**

Hmm. Not quite so easy. Let’s apply some rules to the uniform interface provided by HTTP

GET – Safe, Cacheable, Idempotent  
PUT – Idempotent  
DELETE – Idempotent  
HEAD – Safe, Idempotent  
POST – n/a

Cool! But what does that mean?

– Safe – the operation must not have side effects  
– Cacheable – the result may be cached e.g. by a proxy server  
– Idempotent – The operation must always return the same result

Check!

**Rest Practical Usage**

Let’s now provide some example of in-practice usage:

**Resource and Resource Identifiers**

Example of resources are Car, Engine, Part. Each resource is identified by its *resource identifier*. For example:

Car: http://www.automart.com/cars/12345

Part: http://www.automart.com/part/12345

Part: http://www.automart.com/engine/12345

**Representation**  
Our Car with *resource identifier,* http://www.automart.com/cars/12343 can be manipulated by a component via a uniform interface of GET, POST, PUT, DELETE. Here are some examples:

GET returns a representation of a resource’s state, For example, http://www.automart.com/cars/12343.  
An XML representation of that state might be:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <Car>    <Make>Audi</Make>    <Model>A5</Model>    <Year>2013</Year>  </Car> |

or in JSON

|  |  |
| --- | --- |
| 1  2  3  4  5 | {  "Make" : "Audi",  "Model" : "A5",  "Year" : 2013  } |

**Representation with Linked Resources**  
Resource representations may contain links to other resources  
e.g.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <Car>    ...    <Engine uri="<http://www.automart.com/engine/1242>"/>    ...  </Car> |

or in JSON

|  |  |
| --- | --- |
| 1  2  3  4  5 | {  ...  "engine" : "<http://www.automart.com/engine/1242>"  ...  } |

That’s it!!

# Layered Architecture

The most common architecture pattern is the layered architecture pattern, otherwise known as the n-tier architecture pattern. This pattern is the de facto standard for most Java EE applications and therefore is widely known by most architects, designers, and developers. The layered architecture pattern closely matches the traditional IT communication and organizational structures found in most companies, making it a natural choice for most business application development efforts.

# **Pattern Description**

Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database ([Figure 1-1](https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html#sapr_0101_img)). In some cases, the business layer and persistence layer are combined into a single business layer, particularly when the persistence logic (e.g., SQL or HSQL) is embedded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers.

Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request. Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn’t need to know or worry about how to get customer data; it only needs to display that information on a screen in particular format. Similarly, the business layer doesn’t need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.



###### *Figure 1-1.****Layered architecture pattern***

One of the powerful features of the layered architecture pattern is the separation of concerns among components. Components within a specific layer deal only with logic that pertains to that layer. For example, components in the presentation layer deal only with presentation logic, whereas components residing in the business layer deal only with business logic. This type of component classification makes it easy to build effective roles and responsibility models into your architecture, and also makes it easy to develop, test, govern, and maintain applications using this architecture pattern due to well-defined component interfaces and limited component scope.

# **Key Concepts**

Notice in [Figure 1-2](https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html#sapr_0102_img) that each of the layers in the architecture is marked as being closed. This is a very important concept in the layered architecture pattern. A closed layer means that as a request moves from layer to layer, it must go through the layer right below it to get to the next layer below that one. For example, a request originating from the presentation layer must first go through the business layer and then to the persistence layer before finally hitting the database layer.



###### *Figure 1-2.****Closed layers and request access***

So why not allow the presentation layer direct access to either the persistence layer or database layer? After all, direct database access from the presentation layer is much faster than going through a bunch of unnecessary layers just to retrieve or save database information. The answer to this question lies in a key concept known as layers of isolation.

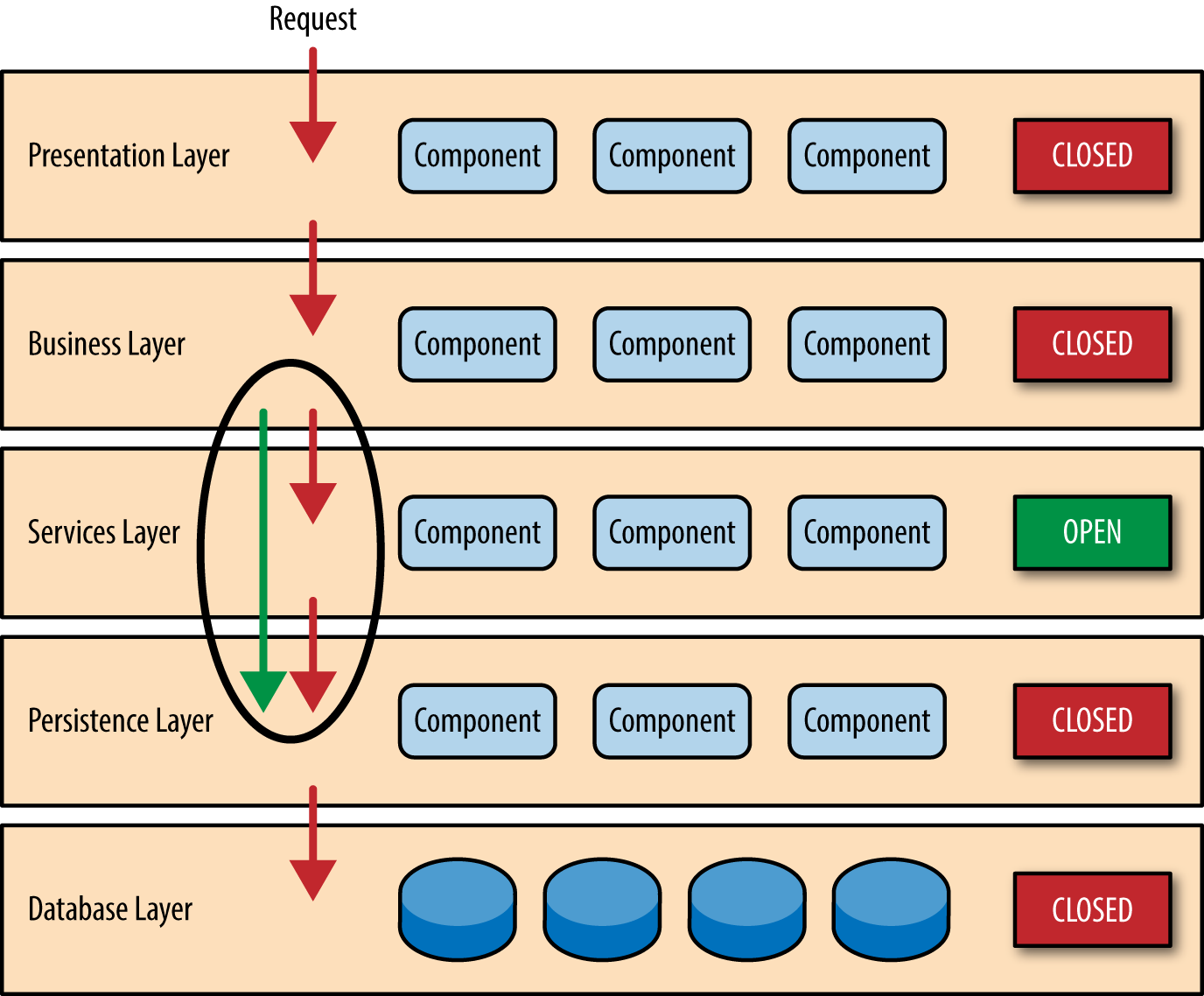
The layers of isolation concept means that changes made in one layer of the architecture generally don’t impact or affect components in other layers: the change is isolated to the components within that layer, and possibly another associated layer (such as a persistence layer containing SQL). If you allow the presentation layer direct access to the persistence layer, then changes made to SQL within the persistence layer would impact both the business layer and the presentation layer, thereby producing a very tightly coupled application with lots of interdependencies between components. This type of architecture then becomes very hard and expensive to change.

The layers of isolation concept also means that each layer is independent of the other layers, thereby having little or no knowledge of the inner workings of other layers in the architecture. To understand the power and importance of this concept, consider a large refactoring effort to convert the presentation framework from JSP (Java Server Pages) to JSF (Java Server Faces). Assuming that the contracts (e.g., model) used between the presentation layer and the business layer remain the same, the business layer is not affected by the refactoring and remains completely independent of the type of user-interface framework used by the presentation layer.

While closed layers facilitate layers of isolation and therefore help isolate change within the architecture, there are times when it makes sense for certain layers to be open. For example, suppose you want to add a shared-services layer to an architecture containing common service components accessed by components within the business layer (e.g., data and string utility classes or auditing and logging classes). Creating a services layer is usually a good idea in this case because architecturally it restricts access to the shared services to the business layer (and not the presentation layer). Without a separate layer, there is nothing architecturally that restricts the presentation layer from accessing these common services, making it difficult to govern this access restriction.

In this example, the new services layer would likely reside below the business layer to indicate that components in this services layer are not accessible from the presentation layer. However, this presents a problem in that the business layer is now required to go through the services layer to get to the persistence layer, which makes no sense at all. This is an age-old problem with the layered architecture, and is solved by creating open layers within the architecture.

As illustrated in [Figure 1-3](https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html#sapr_0103_img), the services layer in this case is marked as open,  meaning requests are allowed to bypass this open layer and go directly to the layer below it. In the following example, since the services layer is open, the business layer is now allowed to bypass it and go directly to the persistence layer, which makes perfect sense.



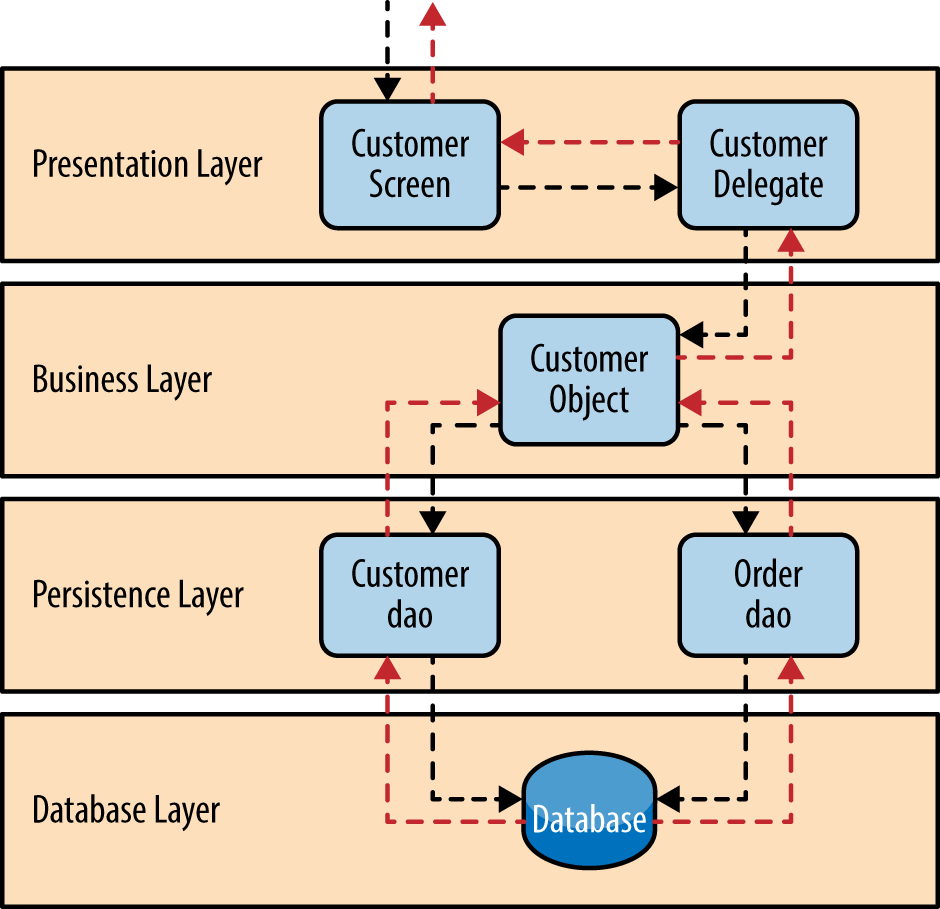
###### *Figure 1-3.****Open layers and request flow***

Leveraging the concept of open and closed layers helps define the relationship between architecture layers and request flows and also provides designers and developers with the necessary information to understand the various layer access restrictions within the architecture. Failure to document or properly communicate which layers in the architecture are open and closed (and why) usually results in tightly coupled and brittle architectures that are very difficult to test, maintain, and deploy.

# **Pattern Example**

To illustrate how the layered architecture works, consider a request from a business user to retrieve customer information for a particular individual as illustrated in [Figure 1-4](https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html#sapr_0104_img). The black arrows show the request flowing down to the database to retrieve the customer data, and the red arrows show the response flowing back up to the screen to display the data. In this example, the customer information consists of both customer data and order data (orders placed by the customer).

The customer screen is responsible for accepting the request and displaying the customer information. It does not know where the data is, how it is retrieved, or how many database tables must be queries to get the data. Once the customer screen receives a request to get customer information for a particular individual, it then forwards that request onto the customer delegate module. This module is responsible for knowing which modules in the business layer can process that request and also how to get to that module and what data it needs (the contract). The customer object in the business layer is responsible for aggregating all of the information needed by the business request (in this case to get customer information). This module calls out to the customer dao (data access object) module in the persistence layer to get customer data, and also the order dao module to get order information. These modules in turn execute SQL statements to retrieve the corresponding data and pass it back up to the customer object in the business layer. Once the customer object receives the data, it aggregates the data and passes that information back up to the customer delegate, which then passes that data to the customer screen to be presented to the user.



###### *Figure 1-4.****Layered architecture example***

From a technology perspective, there are literally dozens of ways these modules can be implemented. For example, in the Java platform, the customer screen can be a (JSF) Java Server Faces screen coupled with the customer delegate as the managed bean component. The customer object in the business layer can be a local Spring bean or a remote EJB3 bean. The data access objects illustrated in the previous example can be implemented as simple POJO’s (Plain Old Java Objects), MyBatis XML Mapper files, or even objects encapsulating raw JDBC calls or Hibernate queries. From a Microsoft platform perspective, the customer screen can be an ASP (active server pages) module using the .NET framework to access C# modules in the business layer, with the customer and order data access modules implemented as ADO (ActiveX Data Objects).

# **Considerations**

The layered architecture pattern is a solid general-purpose pattern, making it a good starting point for most applications, particularly when you are not sure what architecture pattern is best suited for your application. However, there are a couple of things to consider from an architecture standpoint when choosing this pattern.

The first thing to watch out for is what is known as the architecture sinkhole anti-pattern. This anti-pattern describes the situation where requests flow through multiple layers of the architecture as simple pass-through processing with little or no logic performed within each layer. For example, assume the presentation layer responds to a request from the user to retrieve customer data. The presentation layer passes the request to the business layer, which simply passes the request to the persistence layer, which then makes a simple SQL call to the database layer to retrieve the customer data. The data is then passed all the way back up the stack with no additional processing or logic to aggregate, calculate, or transform the data.

Every layered architecture will have at least some scenarios that fall into the architecture sinkhole anti-pattern. The key, however, is to analyze the percentage of requests that fall into this category. The 80-20 rule is usually a good practice to follow to determine whether or not you are experiencing the architecture sinkhole anti-pattern. It is typical to have around 20 percent of the requests as simple pass-through processing and 80 percent of the requests having some business logic associated with the request. However, if you find that this ratio is reversed and a majority of your requests are simple pass-through processing, you might want to consider making some of the architecture layers open, keeping in mind that it will be more difficult to control change due to the lack of layer isolation.

Another consideration with the layered architecture pattern is that it tends to lend itself toward monolithic applications, even if you split the presentation layer and business layers into separate deployable units. While this may not be a concern for some applications, it does pose some potential issues in terms of deployment, general robustness and reliability, performance, and scalability.

# **Pattern Analysis**

The following table contains a rating and analysis of the common architecture characteristics for the layered architecture pattern. The rating for each characteristic is based on the natural tendency for that characteristic as a capability based on a typical implementation of the pattern, as well as what the pattern is generally known for. For a side-by-side comparison of how this pattern relates to other patterns in this report, please refer to [Appendix A](https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/app01.html#pattern-analysis-summary) at the end of this report.

**Overall agility**

Rating:Low

Analysis: Overall agility is the ability to respond quickly to a constantly changing environment. While change can be isolated through the layers of isolation feature of this pattern, it is still cumbersome and time-consuming to make changes in this architecture pattern because of the monolithic nature of most implementations as well as the tight coupling of components usually found with this pattern.

**Ease of deployment**

Rating:Low

Analysis: Depending on how you implement this pattern, deployment can become an issue, particularly for larger applications. One small change to a component can require a redeployment of the entire application (or a large portion of the application), resulting in deployments that need to be planned, scheduled, and executed during off-hours or on weekends. As such, this pattern does not easily lend itself toward a continuous delivery pipeline, further reducing the overall rating for deployment.

**Testability**

Rating:High

Analysis: Because components belong to specific layers in the architecture, other layers can be mocked or stubbed, making this pattern is relatively easy to test. A developer can mock a presentation component or screen to isolate testing within a business component, as well as mock the business layer to test certain screen functionality.

**Performance**

Rating:Low

Analysis: While it is true some layered architectures can perform well, the pattern does not lend itself to high-performance applications due to the inefficiencies of having to go through multiple layers of the architecture to fulfill a business request.

**Scalability**

Rating:Low

Analysis: Because of the trend toward tightly coupled and monolithic implementations of this pattern, applications build using this architecture pattern are generally difficult to scale. You can scale a layered architecture by splitting the layers into separate physical deployments or replicating the entire application into multiple nodes, but overall the granularity is too broad, making it expensive to scale.

**Ease of development**

Rating:High

Analysis: Ease of development gets a relatively high score, mostly because this pattern is so well known and is not overly complex to implement. Because most companies develop applications by separating skill sets by layers (presentation, business, database), this pattern becomes a natural choice for most business-application development. The connection between a company’s communication and organization structure and the way it develops software is outlined is what is called Conway’s law. You can Google “Conway’s law" to get more information about this fascinating correlation.

# Waterfall Model

The Waterfall Model was first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**.  It is very simple to understand and use.  In a waterfall model, each phase must be completed fully before the next phase can begin. This type of [**software development model**](http://istqbexamcertification.com/what-are-the-software-development-models/) is basically used for the for the project which is small and there are no uncertain requirements. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. In this model [**software testing**](http://istqbexamcertification.com/what-is-a-software-testing/) starts only after the development is complete. In **waterfall model phases** do not overlap.

**Diagram of Waterfall-model:**



**Advantages of waterfall model:**

* This model is simple and easy to understand and use.
* It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
* In this model phases are processed and completed one at a time. Phases do not overlap.
* Waterfall model works well for smaller projects where requirements are very well understood.

**Disadvantages of waterfall model:**

* Once an application is in the [**testing**](http://istqbexamcertification.com/what-is-a-software-testing/) stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
* No working software is produced until late during the life cycle.
* High amounts of risk and uncertainty.
* Not a good model for complex and object-oriented projects.
* Poor model for long and ongoing projects.
* Not suitable for the projects where requirements are at a moderate to high risk of changing.

**When to use the waterfall model:**

* This model is used only when the requirements are very well known, clear and fixed.
* Product definition is stable.
* Technology is understood.
* There are no ambiguous requirements
* Ample resources with required expertise are available freely
* The project is short.

Very less customer interaction is involved during the development of the product. Once the product is ready then only it can be demoed to the end users. Once the product is developed and if any failure occurs then the cost of fixing such issues are very high, because we need to update everywhere from document till the logic.

# V Model

V- model means Verification and Validation model. Just like the [**waterfall model**](http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/), the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. **V-Model** is one of the [**many software development models**](http://istqbexamcertification.com/what-are-the-software-development-models/).Testing of the product is planned in parallel with a corresponding phase of development in **V-model**.

**Diagram of V-model:**

The various phases of the V-model are as follows:

**Requirements** like BRS and SRS begin the life cycle model just like the waterfall model. But, in this model before development is started, a [**system test**](http://istqbexamcertification.com/what-is-system-testing/) plan is created.  The [**test plan**](http://istqbexamcertification.com/what-is-the-purpose-and-importance-of-test-plans/) focuses on meeting the functionality specified in the requirements gathering.

**The high-level design (HLD)** phase focuses on system architecture and design. It provide overview of solution, platform, system, product and service/process. An [**integration test**](http://istqbexamcertification.com/what-is-integration-testing/) plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

**The low-level design** **(LLD)** phase is where the actual software components are designed. It defines the actual logic for each and every component of the system. Class diagram with all the methods and relation between classes comes under LLD. [**Component tests**](http://istqbexamcertification.com/what-is-component-testing/) are created in this phase as well.

**The implementation** phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.

**Coding:** This is at the bottom of the V-Shape model. Module design is converted into code by developers. [**Unit Testing**](http://istqbexamcertification.com/what-is-unit-testing/) is performed by the developers on the code written by them.

**Advantages of V-model:**

* Simple and easy to use.
* Testing activities like planning, [**test designing**](http://istqbexamcertification.com/what-is-test-design-or-how-to-specify-test-cases/) happens well before coding. This saves a lot of time. Hence higher chance of success over the waterfall model.
* Proactive defect tracking – that is defects are found at early stage.
* Avoids the downward flow of the defects.
* Works well for small projects where requirements are easily understood.

**Disadvantages of V-model:**

* Very rigid and least flexible.
* Software is developed during the implementation phase, so no early prototypes of the software are produced.
* If any changes happen in midway, then the test documents along with requirement documents has to be updated.

**When to use the V-model:**

* The V-shaped model should be used for small to medium sized projects where requirements are clearly defined and fixed.
* The V-Shaped model should be chosen when ample technical resources are available with needed technical expertise.

High confidence of customer is required for choosing the V-Shaped model approach. Since, no prototypes are produced, there is a very high risk involved in meeting customer expectations.

The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

The Waterfall model is the earliest SDLC approach that was used for software development.

The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

## Waterfall Model - Design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

* **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
* **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
* **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
* **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
* **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
* **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

## Waterfall Model - Application

Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are −

* Requirements are very well documented, clear and fixed.
* Product definition is stable.
* Technology is understood and is not dynamic.
* There are no ambiguous requirements.
* Ample resources with required expertise are available to support the product.
* The project is short.

## Waterfall Model - Advantages

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.

Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

Some of the major advantages of the Waterfall Model are as follows −

* Simple and easy to understand and use
* Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
* Phases are processed and completed one at a time.
* Works well for smaller projects where requirements are very well understood.
* Clearly defined stages.
* Well understood milestones.
* Easy to arrange tasks.
* Process and results are well documented.

## Waterfall Model - Disadvantages

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows −

* No working software is produced until late during the life cycle.
* High amounts of risk and uncertainty.
* Not a good model for complex and object-oriented projects.
* Poor model for long and ongoing projects.
* Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
* It is difficult to measure progress within stages.
* Cannot accommodate changing requirements.
* Adjusting scope during the life cycle can end a project.
* Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

# Spiral Model

The spiral model is similar to the [**incremental model**](http://istqbexamcertification.com/what-is-incremental-model-advantages-disadvantages-and-when-to-use-it/), with more emphasis placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed. Each subsequent spirals builds on the baseline spiral. Its one of the [**software development models**](http://istqbexamcertification.com/what-are-the-software-development-models/) like [**Waterfall**](http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/), [**Agile**](http://istqbexamcertification.com/what-is-agile-model-advantages-disadvantages-and-when-to-use-it/), [**V-Model**](http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/).

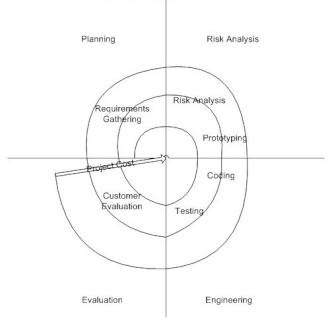
**Planning Phase:**Requirements are gathered during the planning phase. Requirements like ‘BRS’ that is ‘Business Requirement Specifications’ and ‘SRS’ that is ‘System Requirement specifications’.

**Risk Analysis:** In the**risk analysis phase**, a process is undertaken to identify risk and alternate solutions.  A prototype is produced at the end of the risk analysis phase. If any risk is found during the risk analysis then alternate solutions are suggested and implemented.

**Engineering Phase:** In this phase software is **developed**, along with [**testing**](http://istqbexamcertification.com/what-is-a-software-testing/) at the end of the phase. Hence in this phase the development and testing is done.

E**valuation phase:**This phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

**Diagram of Spiral model:**

**[](http://istqbexamcertification.com/wp-content/uploads/2012/01/Spiral-model.jpg)**

**Advantages of Spiral model:**

* High amount of risk analysis hence, avoidance of Risk is enhanced.
* Good for large and mission-critical projects.
* Strong approval and documentation control.
* Additional Functionality can be added at a later date.
* Software is produced early in the [**software life cycle**](http://istqbexamcertification.com/what-are-the-software-development-life-cycle-phases/).

**Disadvantages of Spiral model:**

* Can be a costly model to use.
* Risk analysis requires highly specific expertise.
* Project’s success is highly dependent on the risk analysis phase.
* Doesn’t work well for smaller projects.

**When to use Spiral model:**

* When costs and risk evaluation is important
* For medium to high-risk projects
* Long-term project commitment unwise because of potential changes to economic priorities
* Users are unsure of their needs
* Requirements are complex
* New product line
* Significant changes are expected (research and exploration)

# **Prototype**

The basic idea in **Prototype model** is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Prototype model is a [**software development model**](http://istqbexamcertification.com/what-are-the-software-development-models/). By using this prototype, the client can get an “actual feel” of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.  Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.

The prototype are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

**Diagram of Prototype model:**



**Advantages of Prototype model:**

* Users are actively involved in the development
* Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
* Errors can be detected much earlier.
* Quicker user feedback is available leading to better solutions.
* Missing functionality can be identified easily
* Confusing or difficult functions can be identified  
  Requirements validation, Quick implementation of, incomplete, but  
  functional, application.

**Disadvantages of Prototype model:**

* Leads to implementing and then repairing way of building systems.
* Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
* Incomplete application may cause application not to be used as the  
  full system was designed  
  Incomplete or inadequate problem analysis.

**When to use Prototype model:**

* Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
* Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
* Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

**Quality**

1. **Correctness** - Correctness is that state of a system which has free from every kind of problems; errors and software fulfill the customer needs. In other word it is according to user and developer Expectations.
2. **Reliability** - Quality of Reliability is find on that stage where developer have fully confident about his software that software can satisfied the user and customer that the action performed by the software is according to desired time and desired situations.
3. **Efficiency** - The level of efficiency always calculated by the time period which software use for performing functions. Difference of time period used from actual to expected is show the level of efficiency positive or negative.
4. **Integrity** - This is that state of software where only that user can access the software who have the permission otherwise other person cannot access the software.
5. **Transformable** - The software must have the capacity of transformable because without this quality use of system on large basis is not possible. So software should be transferred from one computer to other computer and one site to other site. With the help of this facility a number of users can use it at the same time.
6. **Accuracy** - Accuracy is that state of software where software has zero percent errors and fulfills all the customer requirements.
7. **Robustness** - When a software performs with the given data and left automatically all such information which is either not accurate or not having the quality of performance, is called robustness.
8. **Testability** - Under this situation for checking the functionality efforts are most required.
9. **Maintainability** - To get the problems in a program some effort is needed. Capacity of maintenance is very necessary of every program.

**Maintenance**

**Answer:** - Maintenance of a software system may be define which is used to concerned about the alteration or changes which are done in software system after the release. Maintenance of software is the part of software Engineering. Maintenance of software has a great value in the development of a system. Needs of Maintenance is required after

* When the user get the product at his own place.
* Installation
* When software is in operational stage.

When any alteration or modification is done in software during the operation time then it is called maintenance. Maintenance of software have a large area which has correcting coding, and design faults, documentation and updating of user support. IEEE gives the definition of maintenance as

Software maintenance is modification of a software product after delivery to correct faults to improve performance or other attribute or to adapt the product to a modified environment.

According to Stephan - Software maintenances is a detailed activity that include

* Error detections and corrections
* Enhancement of capabilities
* deletion of obsolete capabilities
* Optimization

## Reason of Maintenance of software

In the life of a software maintenance activity have a great value. In the comparison of development cost, the maintenance cost is higher. Normally Maintenance of a software take 40 to 70 % cost of total costing of software life cycle. Cost and difficulty are the two drawbacks in maintenance of software. We have some reasons which increases the need of software maintenance.

* Where user needs change time to time.
* When technology of hardware change.
* When the environment of a system changed.
* To increase the ability or capacity of system.
* To keep the same quality of the product.
* To resolve the Errors.
* For getting the best output with the help of existing software.
* To reject the unusual effects.
* For making the software more compatible in the favor of user.

## Types of software Maintenance

1. **Corrective Maintenance** - Corrective maintenance may be define with those alteration which is done for the solving those errors which was available in the software. With the help of corrective maintenance method software can change by removing all the faults. Thus the goal of this method is to correct the software from every type of errors. A software have many kind of faults just like specification errors, logical errors, coding error etc. . and corrective maintenance solve all those types of faults. For the recovery of a system many types of actions performed in corrective Maintenance.

According to K. Bennett,

Maintenance personal sometimes resolve to emergency fixes known as patching to reduce the pressure from the management.

20 % of total maintenance cost is the part of corrective method.

1. **Adaptive Maintenance** - Adaptive maintenance may be defined by that alteration in software system to survive in that area where this system operates. Environment refers those situations which affects the software from outside. According to R. Books,

A change to the whole or part of this environment will require a corresponding alteration of the software.

20 % of the total maintenance cost is the part of adaptive maintenance.

1. **Perfective maintenance** - To increase the efficiency, performance, maintainability, effectiveness of software that is called perfective maintenance. Most of the times enhancement also includes perfective maintenance as one of its part. After changes user operate this software for the purpose which it was developed by developer. For example: if GUI not attract the customer then some change are made for improving the looks and design of the software. Just to get the perfection the changes are made otherwise it is not necessary in normal cases. The demand of the perfective maintenance could be completed by software Engineering. All changes which improve the quality are including in perfective maintenance. The reason of alteration in a system could be a cause in improve the efficiency and functions and easy to understand. 50% of the total maintenance cost is the part of perfective maintenance.

### **Alpha / Beta Testing**

**Answer:**

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Alpha testing** | **Beta Testing** |
| 1 | Alpha testing may be defined as a system testing which is done by the customer at the place where developer has developed the system. | Beta testing may be defined as system testing which is done by the customer on customer's own sites. |
| 2 | Alpha testing takes place once development is complete. | Application is tested in Beta Testing after development and testing is completed. |
| 3 | Alpha testing continues until costomer agrees that system implementation is as per his/her expectation. | The problems faced by customer are reported and software is re-released after beta testing for next beta test cycle. |
| 4 | Alpha testing results in minor design changes. | To get problems and defects before final release of the product, beta testing is very helpful. |
| 5 | Alpha testing is done is a controlled manner because software is tested in developer's area. | Beta testing is done in normal environment and developers are not present during beta testing. |

# Binary Search

Binary search is an efficient algorithm for finding an item from an ordered list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one. We used binary search in the [guessing game](https://www.khanacademy.org/computing/computer-science/algorithms/intro-to-algorithms/a/a-guessing-game) in the introductory tutorial.

One of the most common ways to use binary search is to find an item in an array. For example, the Tycho-2 star catalog contains information about the brightest 2,539,913 stars in our galaxy. Suppose that you want to search the catalog for a particular star, based on the star's name. If the program examined every star in the star catalog in order starting with the first, an algorithm called **linear search**, the computer might have to examine all 2,539,913 stars to find the star you were looking for, in the worst case. If the catalog were sorted alphabetically by star names, binary search would not have to examine more than 22 stars, even in the worst case.

The next few articles discuss how to describe the algorithm carefully, how to implement the algorithm in JavaScript, and how to analyze efficiency.

### **Pseudocode for binary search**

When describing an algorithm to a fellow human being, an incomplete description is often good enough. Some details may be left out of a recipe for a cake; the recipe assumes that you know how to open the refrigerator to get the eggs out and that you know how to crack the eggs. People might intuitively know how to fill in the missing details, but computer programs do not. That's why we need to describe computer algorithms completely.

In order to implement an algorithm in a programming language, you will need to understand an algorithm down to the details. What are the inputs to the problem? The outputs? What variables should be created, and what initial values should they have? What intermediate steps should be taken to compute other values and to ultimately compute the output? Do these steps repeat instructions that can be written in simplified form using a loop?

Let's look at how to describe binary search carefully. The main idea of binary search is to keep track of the current range of reasonable guesses. Let's say that I'm thinking of a number between one and 100, just like [the guessing game](https://www.khanacademy.org/computing/computer-science/algorithms/intro-to-algorithms/a/a-guessing-game). If you've already guessed 25 and I told you my number was higher, and you've already guessed 81 and I told you my number was lower, then the numbers in the range from 26 to 80 are the only reasonable guesses. Here, the red section of the number line contains the reasonable guesses, and the black section shows the guesses that we've ruled out:

https://s3.amazonaws.com/ka-cs-algorithms/binary_search_26_80.png

Binary search number line 26 to 80

In each turn, you choose a guess that divides the set of reasonable guesses into two ranges of roughly the same size. If your guess is not correct, then I tell you whether it's too high or too low, and you can eliminate about half of the reasonable guesses. For example, if the current range of reasonable guesses is 26 to 80, you would guess the halfway point, (26 + 80) / 2(26+80)/2left parenthesis, 26, plus, 80, right parenthesis, slash, 2, or 53. If I then tell you that 53 is too high, you can eliminate all numbers from 53 to 80, leaving 26 to 52 as the new range of reasonable guesses, halving the size of the range.

https://s3.amazonaws.com/ka-cs-algorithms/binary_search_26_52.png

Binary search number line 26 to 52

For the guessing game, we can keep track of the set of reasonable guesses using a few variables. Let the variable min*min*m, i, n be the current minimum reasonable guess for this round, and let the variable max*max*m, a, x be the current maximum reasonable guess. The input to the problem is the number n*n*n, the highest possible number that your opponent is thinking of. We assume that the lowest possible number is one, but it would be easy to modify the algorithm to take the lowest possible number as a second input.

Here's a pseudocode description of binary search:

1. Let min = 1*min*=1m, i, n, equals, 1and max = n*max*=*n*m, a, x, equals, n.
2. Guess the average of max*max*m, a, x and $ min $, rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set $ min $ to be one larger than the guess.
5. If the guess was too high, set $ max $ to be one smaller than the guess.
6. Go back to step two.

We could make this pseudocode even more precise by clearly describing the inputs and the outputs for the algorithm and by clarifying what we mean by instructions like "guess a number" and "stop." But this will do for now.

# [Advantages of Binary Search Trees over Hash Tables](http://stackoverflow.com/questions/4128546/advantages-of-binary-search-trees-over-hash-tables)

Remember that Binary Search Trees (reference-based) are memory-efficient. They do not reserve more memory than they need to.

For instance, if a hash function has a range R(h) = 0...100, then you need to allocate an array of 100 (pointers-to) elements, even if you are just hashing 20 elements. If you were to use a binary search tree to store the same information, you would only allocate as much space as you needed, as well as some metadata about links.

One advantage that no one else has pointed out is that binary search tree allows you to do range searches efficiently.

In order to illustrate my idea, I want to make an extreme case. Say you want to get all the elements whose keys are between 0 to 5000. And actually there is only one such element and 10000 other elements whose keys are not in the range. BST can do range searches quite efficiently since it does not search a subtree which is impossible to have the answer.

While, how can you do range searches in a hash table? You either need to iterate every bucket space, which is O(n), or you have to look for whether each of 1,2,3,4... up to 5000 exists. (what about the keys between 0 and 5000 are an infinite set? for example keys can be decimals).

Hash tables in general have better cache behavior requiring less memory reads compared to a binary tree. For a hash table you normally only incur a single read before you have access to a reference holding your data. The binary tree, if it is a balanced variant, requires something in the order of k \* lg(n) memory reads for some constant k.

On the other hand, if an enemy knows your hash-function the enemy can enforce your hash table to make collisions, greatly hampering its performance. The workaround is to choose the hash-function randomly from a family, but a BST does not have this disadvantage. Also, when the hash table pressure grows too much, you often tend to enlargen and reallocate the hash table which may be an expensive operation. The BST has simpler behavior here and does not tend to suddenly allocate a lot of data and do a rehashing operation.

Trees tend to be the ultimate average data structure. They can act as lists, can easily be split for parallel operation, have fast removal, insertion and lookup on the order of O(lg n). They do nothing particularly well, but they don't have any excessively bad behavior either.

Finally, BSTs are much easier to implement in (pure) functional languages compared to hash-tables and they do not require destructive updates to be implemented (the persistence argument by Pascal above).

## Web Services

[Web Service](https://en.wikipedia.org/wiki/Web_service) is a way to expose the functionality of an application to other application, without a user interface. It is a service which exposes an API over HTTP.

Web Services allow applications developed in different technologies to communicate with each other through a common format like XML, Jason, etc. Web services are not tied to any one operating system or programming language. For example, an application developed in Java can communicate with the one developed in C#, Android, etc., and vice versa.

Web Service is a connection technology, a way to connect services together into a Service Oriented Architecture (SOA).

## What is Micro Service?

[Micro Service](https://en.wikipedia.org/wiki/Microservices) is independently deployable service modeled around a business domain. It is a method of breaking large software applications into loosely coupled modules, in which each service runs a unique process and communicates through APIs. It can be developed using messaging or event-driven APIs, or using non-HTTP backed RPC mechanisms.

Micro Services are designed to cope with failure and breakdowns of large applications. Since multiple unique services are communicating together, it may happen that a particular service fails, but the overall larger applications remain unaffected by the failure of a single module.

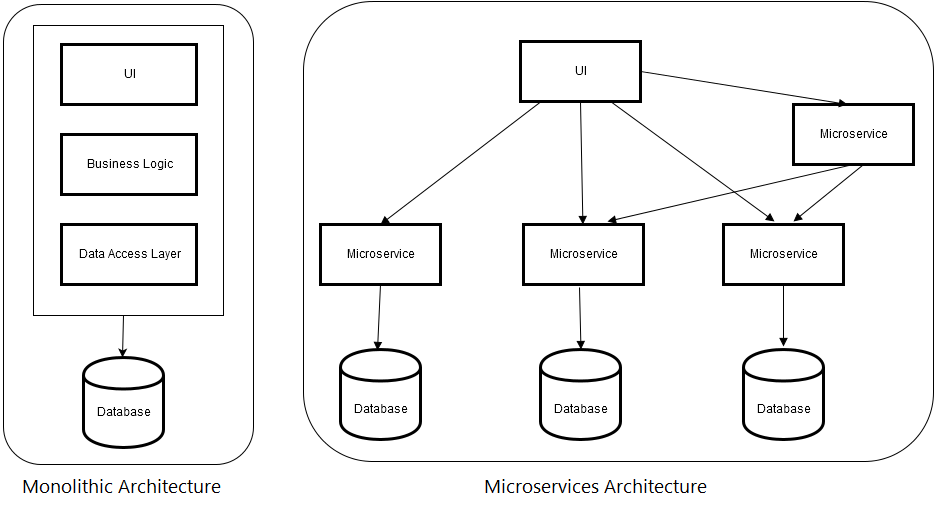
## Use-Case Representation

Let us understand these concepts with the help of an example of Online Shopping Center.

**In figure-1:** The Online Shopping Center Web Application is developed in Monolithic Architecture. In this application, there is one Web Service which communicates with web application and database. So this web service might be performing many functional tasks related to database operations.

Figure 1 - Conventional Approach**In figure-2:** The Online Shopping Center Web Application is developed in Micro Services Architecture. All the components of the web application are developed independently, single functional responsible, fine-grained clearly scoped services.

Web Services could be of any size, including large enterprise apps retrofitted with APIs that too many other apps depended on. Although "micro" in Micro Services, the basic concept is that each service performs a single function.

[](http://www.tatvasoft.com/blog/wp-content/uploads/2016/06/Monolithic-Micro-services-Architecture.png)For example, one of the largest eCommerce portal, Amazon, has migrated to Micro Services. They get countless calls from a variety of applications, including applications that manage the Web Services API as well as the portal, which would have been simply impossible to handle for their old, two-tiered architecture.

Applications built as Micro Services can be broken into multiple component services and this service can be a Web Service, which should run unique process and then redeployed independently without compromising the integrity of an application.

Micro Services style is usually organized around business capabilities and priorities. Unlike a traditional monolithic development approach, where different teams have a specific focus on, say, UIs, databases, technology layers, or server-side logic, Micro Services architecture utilizes cross-functional teams. The responsibilities of each team are to make specific products based on one or more individual services communicating via message bus. It means that when changes are required, there won’t necessarily be any reason for the project, as a whole, to take more time or for developers to have to wait for budgetary approval before individual services can be improved. Most development methods focus on projects: a piece of code that has to offer some predefined business value must be handed over to the client, and is then periodically maintained by a team. But in Micro Services, a team owns the product for its lifetime.

In a monolithic service oriented architecture deployment, each small change meant that the entire monolith needed to be rebuilt and this, in turn, meant that re-builds weren't happening as rapidly as they should.

A Web Service is a service offered by an application to another application, communicating with each other via the World Wide Web.

# Difference Between Microservices and SOA

The Web Service typically provides an object-oriented web-based interface to a database server, utilized by another web server, or by a mobile application, that provides a user interface to the end user. Another common application offered to the end user may be a mash-up, where a web server consumes several web services at different machines and compiles the content into one user interface.

|  |  |
| --- | --- |
|  | I guess you could think of the Microservices Architectural Style as a specialisation of SOA. Don't forget that one of the [accepted views](http://en.wikipedia.org/wiki/Service-orientation) is that all SOA really is, is four sentences:   * Boundaries are explicit * Services are autonomous * Services share schema and contract, not class * Service compatibility is based on policy   -Don Box, Microsoft (pre-.Net 3.0)  This brings us to [the canonical definition of microservices](http://martinfowler.com/articles/microservices.html), from Lewis/Fowler:  In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralised management of these services, which may be written in different programming languages and use different data storage technologies.  From this definition, it's clear that microservices fulfil at least the first two tenets (with a real emphasis on the second), but it's questionable whether they fulfil the third (I don't really understand tenet 4 so I won't comment).  The reason the third tenet may not hold for microservices is that one of the characteristics of microservices is that they are generally exposed over a RESTful API, which, one could argue, does not expose contract and schema at all (over and above the regular HTTP verbiage), as we see from Fowler:  a suite of small services, each... communicating with lightweight mechanisms, often an HTTP resource API  Another way in which a microservices style deviates from SOA is with this prescription:  These services are... independently deployable by fully automated deployment machinery  Following the original tenets of SOA does not prevent me from manually copying my service binaries into my production environment, but with the microservices approach, the service deployment and management should be fully automated. |

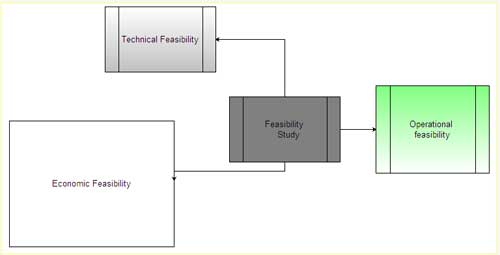
# Feasibility Study

**Feasibility** is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practical and workable in the software. Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study. The objective of the feasibility study is to establish the reasons for developing the software that is acceptable to users, adaptable to change and conformable to established standards. Various other objectives of feasibility study are listed below.

* To analyze whether the software will meet organizational requirements
* To determine whether the software can be implemented using the current technology and within the specified budget and schedule
* To determine whether the software can be integrated with other existing software.

## Types of Feasibility

Various types of feasibility that are commonly considered include technical feasibility, operational feasibility, and economic feasibility.

[](http://ecomputernotes.com/images/Types-of-Feasibility.jpg)

**Technical feasibility** assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, the software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility also performs the following tasks.

* Analyzes the technical skills and capabilities of the software development team members
* Determines whether the relevant technology is stable and established
* Ascertains that the technology chosen for software development has a large number of users so that they can be consulted when problems arise or improvements are required.

**Operational feasibility** assesses the extent to which the required software performs a series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether the software will operate after it is developed and be operative once it is installed. Operational feasibility also performs the following tasks.

* Determines whether the problems anticipated in user requirements are of high priority
* Determines whether the solution suggested by the software development team is acceptable
* Analyzes whether users will adapt to a new software
* Determines whether the organization is satisfied by the alternative solutions proposed by the software development team.

**Economic feasibility** determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software. Software is said to be economically feasible if it focuses on the issues listed below.

* Cost incurred on software development to produce long-term gains for an organization
* Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis)
* Cost of hardware, software, development team, and training.

# Feasibility Study Process

Feasibility study comprises the following steps:

1. **Information assessment:**Identifies [information](http://ecomputernotes.com/fundamental/information-technology/what-do-you-mean-by-data-and-information) about whether the system helps in achieving the objectives of the organization. It also verifies that the system can be implemented using new technology and within the budget and whether the system can be integrated with the existing system.
2. **Information collection:**Specifies the sources from where information about software can be obtained. Generally, these sources include users (who will operate the software), organization (where the software will be used), and the software development team (which understands user requirements and knows how to fulfill them in software).
3. **Report writing:**Uses a feasibility report, which is the conclusion of the feasibility study by the software development team. It includes the recommendations whether the software development should continue. This report may also include information about changes in the software scope, budget, and schedule and suggestions of any requirements in the system.
4. **General information:**Describes the purpose and scope of feasibility study. It also describes system overview, project references, acronyms and abbreviations, and points of contactto be used. **System overview**provides description about the name of the organization responsible for the software development, system name or title, system category, operational status, and so on. **Project references** provide a list of the references used to prepare this document such as documents relating to the project or previously developed documents that are related to the project. **Acronyms and abbreviations**provide a list of the terms that are used in this document along with their meanings. **Points of contact** provide a list of points of organizational contact with users for information and coordination. For example, users require assistance to solve problems (such as troubleshooting) and collect information such as contact number, e-mail address, and so on.
5. **Management summary:**Provides the following information.
6. **Environment:**Identifies the individuals responsible for software development. It provides information about input and output requirements, processing requirements of the software and the interaction of the software with other software. It also identifies system security requirements and the system's processing requirements
7. **Current functional procedures:**Describes the current functional procedures of the existing system, whether automated or manual. It also includes the data-flow of the current system and the number of team members required to operate and maintain the software.
8. **Functional objective:**Provides information about functions of the system such as new services, increased capacity, and so on.
9. **Performance objective:**Provides information about performance objectives such as reduced staff and equipment costs, increased processing speeds of software, and improved controls.
10. **Assumptions and constraints:**Provides information about assumptions and constraints such as operational life of the proposed software, financial constraints, changing hardware, software and operating environment, and availability of information and sources.
11. **Methodology:**Describes the methods that are applied to evaluate the proposed software in order to reach a feasible alternative. These methods include survey, modeling, benchmarking, etc.
12. **Evaluation criteria:**Identifies criteria such as cost, priority, development time, and ease of system use, which are applicable for the development process to determine the most suitable system option.
13. **Recommendation:**Describes a recommendation for the proposed system. This includes the delays and acceptable risks.
14. **Proposed software:**Describes the overall concept of the system as well as the procedure to be used to meet user requirements. **In**addition, it provides information about improvements, time and resource costs, and impacts. Improvements are performed to enhance the functionality and performance of the existing software. Time and resource costs include the costs associated with software development from its requirements to its maintenance and staff training. Impacts describe the possibility of future happenings and include various types of impacts as listed below.
15. **Equipment impacts:**Determine new equipment requirements and changes to be made in the currently available equipment requirements.
16. **Software impacts:**Specify any additions or modifications required in the existing software and supporting software to adapt to the proposed software.
17. **Organizational impacts:**Describe any changes in organization, staff and skills requirement.
18. **Operational impacts:**Describe effects on operations such as user-operating procedures, data processing, data entry procedures, and so on.
19. **Developmental impacts:**Specify developmental impacts such as resources required to develop databases, resources required to develop and test the software, and specific activities to be performed by users during software development.
20. **Security impacts:**Describe security factors that may influence the development, design, and continued operation of the proposed software.
21. **Alternative systems:**Provide description of alternative systems, which are considered in a feasibility study. This also describes the reasons for choosing a particular alternative system to develop the proposed software and the reason for rejecting alternative systems.

**Maintenance**

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

* **Market Conditions**- Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
* **Client Requirements** - Over the time, customer may ask for new features or functions in the software.
* **Host Modifications**- If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
* **Organization Changes** - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

## Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

* **Corrective Maintenance** - This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
* **Adaptive Maintenance** - This includes modifications and updations applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
* **Perfective Maintenance** - This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
* **Preventive Maintenance** - This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

## Cost of Maintenance

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



On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:

### **Real-world factors affecting Maintenance Cost**

* The standard age of any software is considered up to 10 to 15 years.
* Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
* As technology advances, it becomes costly to maintain old software.
* Most maintenance engineers are newbie and use trial and error method to rectify problem.
* Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
* Changes are often left undocumented which may cause more conflicts in future.

### **Software-end factors affecting Maintenance Cost**

* Structure of Software Program
* Programming Language
* Dependence on external environment
* Staff reliability and availability

## Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be included.



These activities go hand-in-hand with each of the following phase:

* **Identification & Tracing** - It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.
* **Analysis** - The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.
* **Design** - New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.
* **Implementation** - The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.
* **System Testing** - Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.
* **Acceptance Testing** - After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.
* **Delivery** - After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.

Training facility is provided if required, in addition to the hard copy of user manual.

* **Maintenance management** - Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

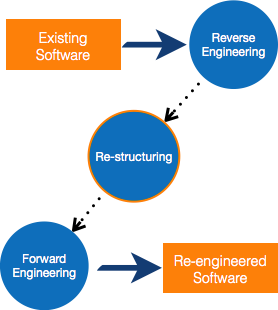
## Software Re-engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.



### **Re-Engineering Process**

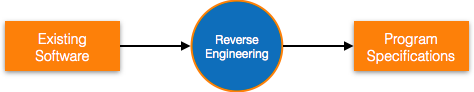
* **Decide** what to re-engineer. Is it whole software or a part of it?
* **Perform** Reverse Engineering, in order to obtain specifications of existing software.
* **Restructure Program** if required. For example, changing function-oriented programs into object-oriented programs.
* **Re-structure data** as required.
* **Apply Forward engineering** concepts in order to get re-engineered software.

There are few important terms used in Software re-engineering

### **Reverse Engineering**

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.



### **Program Restructuring**

It is a process to re-structure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code-restructuring and data-restructuring or both.

Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can be removed via re-structuring.

### **Forward Engineering**

Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.



## Component reusability

A component is a part of software program code, which executes an independent task in the system. It can be a small module or sub-system itself.

### **Example**

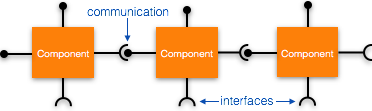
The login procedures used on the web can be considered as components, printing system in software can be seen as a component of the software.

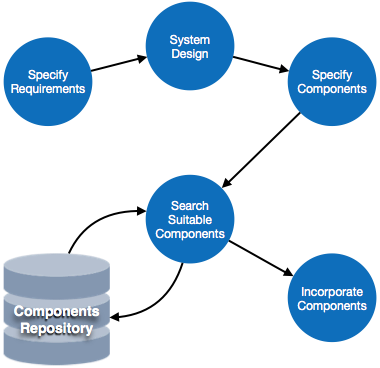
Components have high cohesion of functionality and lower rate of coupling, i.e. they work independently and can perform tasks without depending on other modules.

In OOP, the objects are designed are very specific to their concern and have fewer chances to be used in some other software.

In modular programming, the modules are coded to perform specific tasks which can be used across number of other software programs.

There is a whole new vertical, which is based on re-use of software component, and is known as Component Based Software Engineering (CBSE).

Re-use can be done at various levels

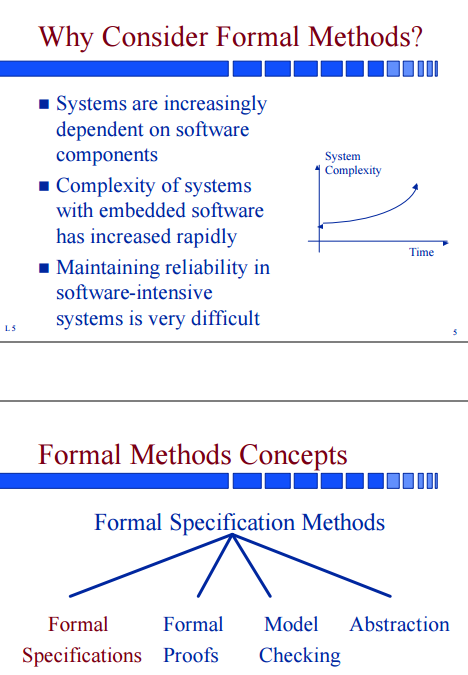
* **Application level**- Where an entire application is used as sub-system of new software.
* **Component level**- Where sub-system of an application is used.
* **Modules level**- Where functional modules are re-used.

Software components provide interfaces, which can be used to establish communication among different components.

### **Reuse Process**

Two kinds of method can be adopted: either by keeping requirements same and adjusting components or by keeping components same and modifying requirements.

* **Requirement Specification** - The functional and non-functional requirements are specified, which a software product must comply to, with the help of existing system, user input or both.
* **Design** - This is also a standard SDLC process step, where requirements are defined in terms of software parlance. Basic architecture of system as a whole and its sub-systems are created.
* **Specify Components**- By studying the software design, the designers segregate the entire system into smaller components or sub-systems. One complete software design turns into a collection of a huge set of components working together.
* **Search Suitable Components** - The software component repository is referred by designers to search for the matching component, on the basis of functionality and intended software requirements..
* **Incorporate Components** - All matched components are packed together to shape them as complete software.

**Formal Specifications**

Translation of a non-mathematical description (diagrams, tables, English text) into a formal specification language. Concise description of high-level behavior and properties of a system. Well-defined language semantics support formal deduction about specification

**Goal:** Describe external behaviour without describing or constraining implementation n Formal Method has 2 parts:

» **Logical Theory:** Means by which one reasons about specifications, properties and programs

– First order predicate calculus (quantification over variables)

– Second order predicate calculus (quantification over relations)

– Temporal logic

**» Structuring Theory:** Defines elements being reasoned about

**Types of Formal Specifications**

**Property Oriented:** State desired properties in a purely declarative way

**» Algebraic:** Data type viewed as an algebra, axioms state properties of data type’s operations

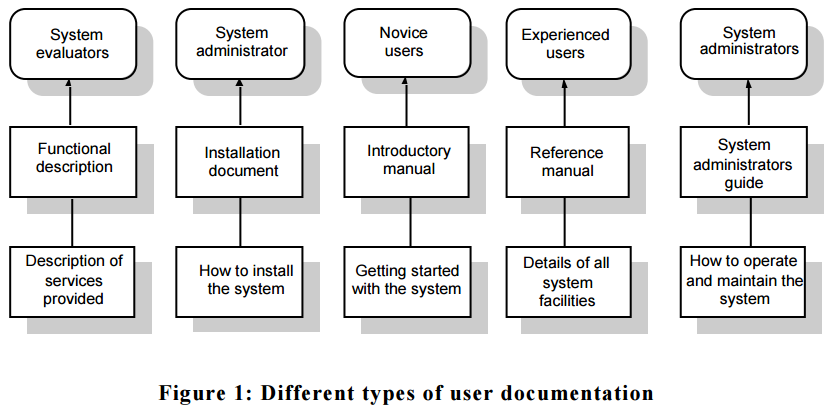
**» Axiomatic:** Uses first order predicate logic, pre and post conditions Operational Specification: Describe desired behaviour by providing model of system

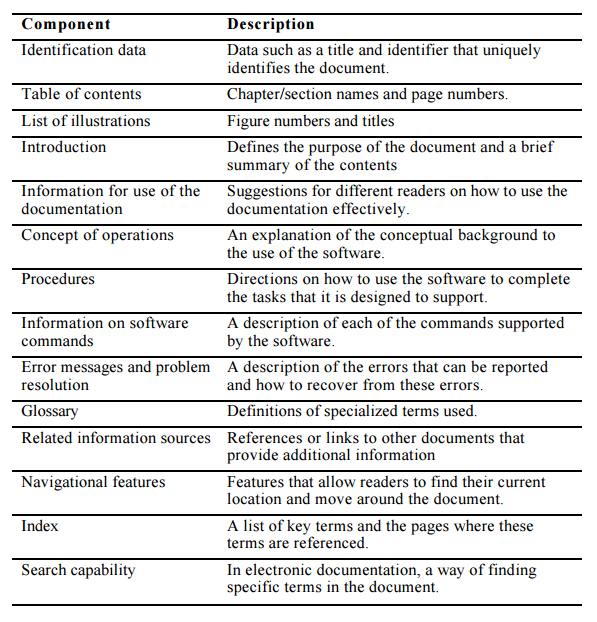
**Model Oriented:** Provide direct way of describing system behaviour (sets, sequences, tuples, maps) :

» Abstract Model (in terms previously defined mathematical objects eg. sets, sequences, functions, mappings)

» State machines

**Documentation**

****

****

**Debugging**

A **debugger** or **debugging tool** is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation - full or partial simulation, to limit this impact.

A "crash" happens when the program cannot normally continue because of a programming bug. For example, the program might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. When the program "crashes" or reaches a preset condition, the debugger typically shows the position in the original code if it is a **source-level debugger** or **symbolic debugger**, commonly now seen in integrated development environments. If it is a **low-level debugger** or a **machine-language debugger** it shows the line in the disassembly (unless it also has online access to the original source code and can display the appropriate section of code from the assembly or compilation).

Typically, debuggers also offer more sophisticated functions such as running a program step by step (**single-stepping** or program animation), stopping (**breaking**) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint, and tracking the values of some variables. Some debuggers have the ability to modify the state of the program while it is running, rather than merely to observe it. It may also be possible to continue execution at a different location in the program to bypass a crash or logical error.

The importance of a good debugger cannot be overstated. Indeed, the existence and quality of such a tool for a given language and platform can often be the deciding factor in its use, even if another language/platform is better-suited to the task.[[*citation needed*](https://en.wikibooks.org/wiki/Wikibooks:OR)]. The absence of a debugger, having once been accustomed to using one, has been said to "make you feel like a blind man in a dark room looking for a black cat that isn’t there".[[1]](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Debugger#cite_note-1) However, software can (and often does) behave differently running under a debugger than normally, due to the inevitable changes the presence of a debugger will make to a software program's internal timing. As a result, even with a good debugging tool, it is often very difficult to track down runtime problems in complex multi-threaded or distributed systems.

The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features. It often also makes it useful as a general testing verification tool test coverage and performance analyzer, especially if instruction path lengths are shown.

Most current mainstream debugging engines, such as gdb and dbx provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, program animation, and visualization features. Some early mainframe debuggers such as IBM OLIVER (CICS interactive test/debug) and SIMON (Batch Interactive test/debug) provided this same functionality for the IBM System/360 and later operating systems, as long ago as the 1970s.

**Debugging Patterns**

A bug pattern is a particular type of pattern. The original concept of a pattern was introduced by the architect [Christopher Alexander](https://en.wikipedia.org/wiki/Christopher_Alexander) as a [design pattern](https://en.wikipedia.org/wiki/Design_pattern).

Some examples of debugging patterns include:

* Eliminate Noise Bug Pattern - Isolate and expose a particular bug by eliminating all other [noise](https://en.wikipedia.org/wiki/Noise) in the system. This enables you to concentrate on finding the real issue.
* Recurring Bug Pattern - Expose a bug via a [unit test](https://en.wikipedia.org/wiki/Unit_test). Run that unit test as part of a standard build from that moment on. This ensure that the bug will not recur.
* Time Specific Bug Pattern - Expose the bug by writing a [continuous test](https://en.wikipedia.org/wiki/Continuous_testing) that runs continuously and fails when an expected error occurs. This is useful for transient bugs.

**Safety**

Safety and reliability are related but distinct

In general, reliability and availability are necessary but not sufficient conditions for system safety

Reliability is concerned with conformance to a given specification and delivery of service

Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification.

System reliability is essential for safety but is not enough

**Reliable systems can be unsafe**

* **There may be dormant faults in a system that are undetected for many years and only rarely arise.**
* **Specification errors**
  + If the system specification is incorrect then the system can behave as specified but still cause an accident.
* **Hardware failures generating spurious inputs**
  + Hard to anticipate in the specification.
* **Context-sensitive commands i.e. issuing the right command at the wrong time**
  + Often the result of operator error.

**Safety critical systems**

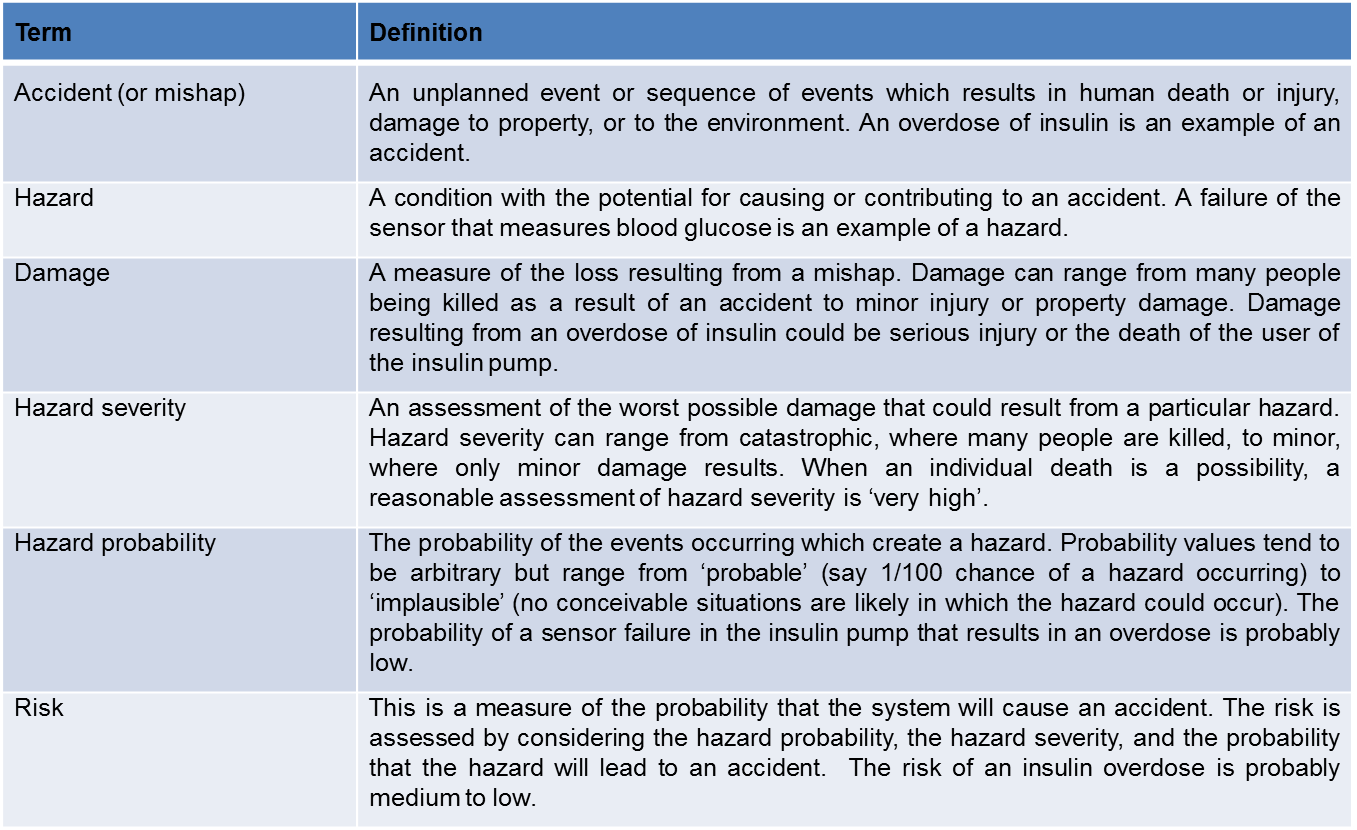
* Systems where it is essential that system operation is always safe i.e. the system should never cause damage to people or the system’s environment
* Examples
  + Control and monitoring systems in aircraft
  + Process control systems in chemical manufacture
  + Automobile control systems such as braking and engine management systems
* **Primary safety-critical systems**
  + Embedded software systems whose failure can cause the associated hardware to fail and directly threaten people. Example is the insulin pump control system.
* **Secondary safety-critical systems**
  + Systems whose failure results in faults in other (socio-technical) systems, which can then have safety consequences.
    - For example, the Mentcare system is safety-critical as failure may lead to inappropriate treatment being prescribed.
    - Infrastructure control systems are also secondary safety-critical systems.

**Hazards**

* Situations or events that can lead to an accident
  + Stuck valve in reactor control system
  + Incorrect computation by software in navigation system
  + Failure to detect possible allergy in medication prescribing system
* Hazards do not inevitably result in accidents – accident prevention actions can be taken.

**Normal accidents**

* Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure
  + Designing systems so that a single point of failure does not cause an accident is a fundamental principle of safe systems design.
* Almost all accidents are a result of combinations of malfunctions rather than single failures.
* It is probably the case that anticipating all problem combinations, especially, in software controlled systems is impossible so achieving complete safety is impossible. Accidents are inevitable.

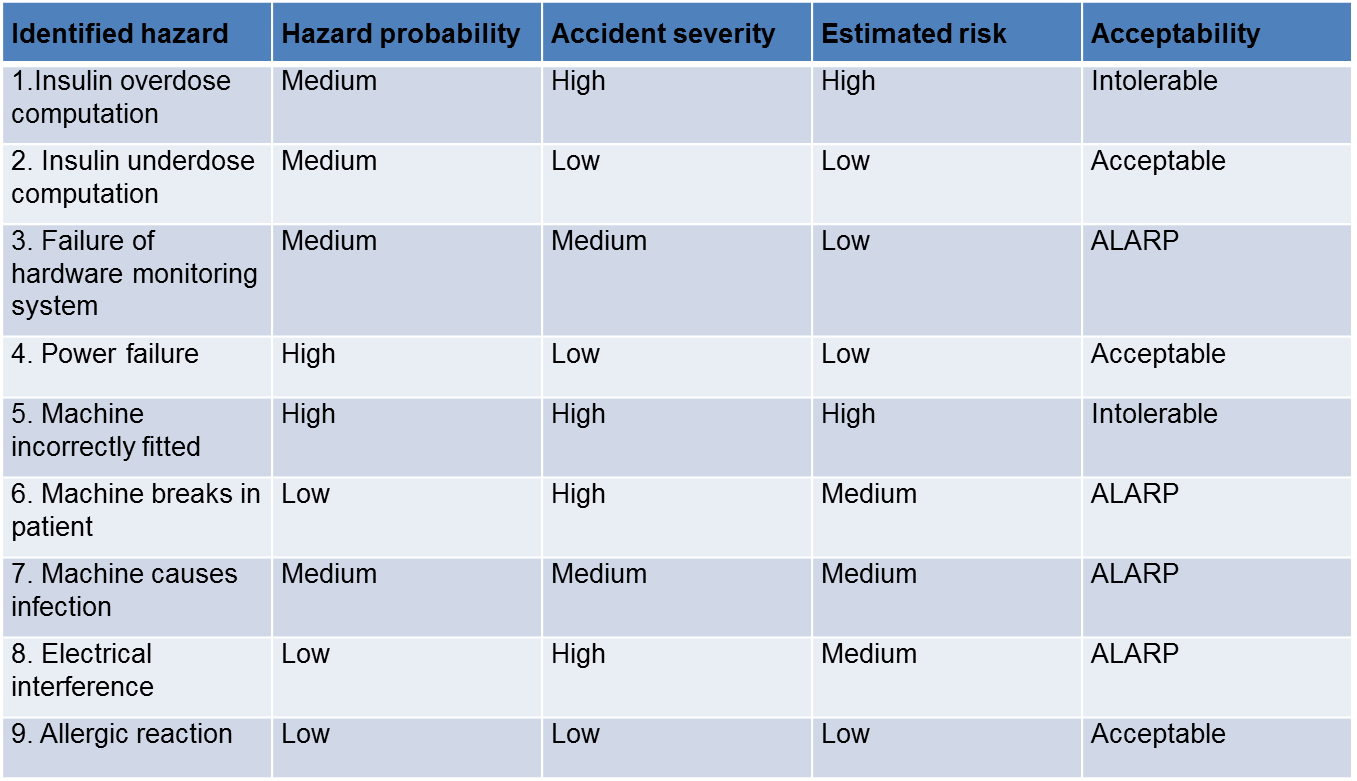


* Although software failures can be safety-critical, the use of software control systems contributes to increased system safety
  + Software monitoring and control allows a wider range of conditions to be monitored and controlled than is possible using electro-mechanical safety systems.
  + Software control allows safety strategies to be adopted that reduce the amount of time people spend in hazardous environments.
  + Software can detect and correct safety-critical operator errors.
* **Risk Triangle**

12.2 RiskTriangle.eps

* The acceptability of a risk is determined by human, social and political considerations.
* In most societies, the boundaries between the regions are pushed upwards with time i.e. society is less willing to accept risk
  + For example, the costs of cleaning up pollution may be less than the costs of preventing it but this may not be socially acceptable.
* Risk assessment is subjective
  + Risks are identified as probable, unlikely, etc. This depends on who is making the assessment.

**Example of Risk Classification**



**Example of safety requirements**

**SR1**: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

**SR2**: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

**SR3**: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

**SR4**: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

**SR5**: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message, as defined in Table 4, shall be displayed.

**SR6**: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

**Example of safety claim hierarchy**

15.9 Claim hierarchy.eps

# **Open Close Principle**

## Motivation

A clever application design and the code writing part should take care of the frequent changes that are done during the development and the maintaining phase of an application. Usually, many changes are involved when a new functionality is added to an application. Those changes in the existing code should be minimized, since it's assumed that the existing code is already unit tested and changes in already written code might affect the existing functionality in an unwanted manner.

The **Open Close Principle** states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

## Intent

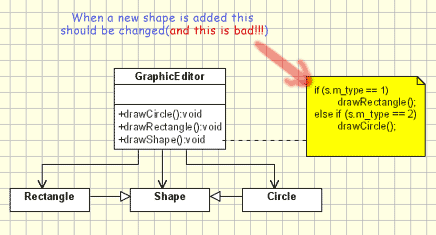
Software entities like classes, modules and functions should be **open for extension** but **closed for modifications**.

## Example

Bellow is an example which violates the Open Close Principle. It implements a graphic editor which handles the drawing of different shapes. It's obviously that it does not follow the Open Close Principle since the GraphicEditor class has to be modified for every new shape class that has to be added. There are several disadvantages:

* for each new shape added the unit testing of the GraphicEditor should be redone.
* when a new type of shape is added the time for adding it will be high since the developer who add it should understand the logic of the GraphicEditor.
* adding a new shape might affect the existing functionality in an undesired way, even if the new shape works perfectly

In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionality inside, written and changed by many developers, while the shape might be a class implemented only by one developer. In this case it would be great improvement to allow the adding of a new shape without changing the GraphicEditor class.



|  |
| --- |
| // Open-Close Principle - Bad example  class GraphicEditor {    public void drawShape(Shape s) {  if (s.m\_type==1)  drawRectangle(s);  else if (s.m\_type==2)  drawCircle(s);  }  public void drawCircle(Circle r) {....}  public void drawRectangle(Rectangle r) {....}  }    class Shape {  int m\_type;  }    class Rectangle extends Shape {  Rectangle() {  super.m\_type=1;  }  }    class Circle extends Shape {  Circle() {  super.m\_type=2;  }  } |

Bellow is a example which supports the Open Close Principle. In the new design we use abstract draw() method in GraphicEditor for drawing objects, while moving the implementation in the concrete shape objects. Using the Open Close Principle the problems from the previous design are avoided, because GraphicEditor is not changed when a new shape class is added:

* no unit testing required.
* no need to understand the sourcecode from GraphicEditor.
* since the drawing code is moved to the concrete shape classes, it's a reduced risk to affect old functionallity when new functionallity is added.

## Open Close Principle(OCP) - goodConclusion

Like every principle OCP is only a principle. Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those area which are most likely to be changed.

There are many design patterns that help us to extend code without changing it. For instance the Decorator pattern help us to follow Open Close principle. Also the Factory Method or the Observer pattern might be used to design an application easy to change with minimum changes in the existing code.

## Dependency Inversion Principle

## Motivation

When we design software applications we can consider the low level classes the classes which implement basic and primary operations(disk access, network protocols,...) and high level classes the classes which encapsulate complex logic(business flows, ...). The last ones rely on the low level classes. A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes. Since high level classes are defined in terms of others this seems the logical way to do it. But this is not a flexible design. What happens if we need to replace a low level class?

Let's take the classical example of a copy module which reads characters from the keyboard and writes them to the printer device. The high level class containing the logic is the Copy class. The low level classes are KeyboardReader and PrinterWriter.

In a bad design the high level class uses directly and depends heavily on the low level classes. In such a case if we want to change the design to direct the output to a new FileWriter class we have to make changes in the Copy class. (Let's assume that it is a very complex class, with a lot of logic and really hard to test).

In order to avoid such problems we can introduce an abstraction layer between high level classes and low level classes. Since the high level modules contain the complex logic they should not depend on the low level modules so the new abstraction layer should not be created based on low level modules. Low level modules are to be created based on the abstraction layer.

According to this principle the way of designing a class structure is to start from high level modules to the low level modules:  
High Level Classes --> Abstraction Layer --> Low Level Classes

## Intent

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.

## Example

Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class called Worker. We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers. We created a new class SuperWorker for this.

Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

* we have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).
* some of the current functionality from the manager class might be affected.
* the unit testing should be redone.

All those problems could take a lot of time to be solved and they might induce new errors in the old functionlity. The situation would be different if the application had been designed following the Dependency Inversion Principle. It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it. No additional changes in the existing classes.

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved(considering there is no change in the high level logic):

* Manager class doesn't require changes when adding SuperWorkers.
* Minimized risk to affect old functionality present in Manager class since we don't change it.
* No need to redo the unit testing for Manager class.

## Conclusion

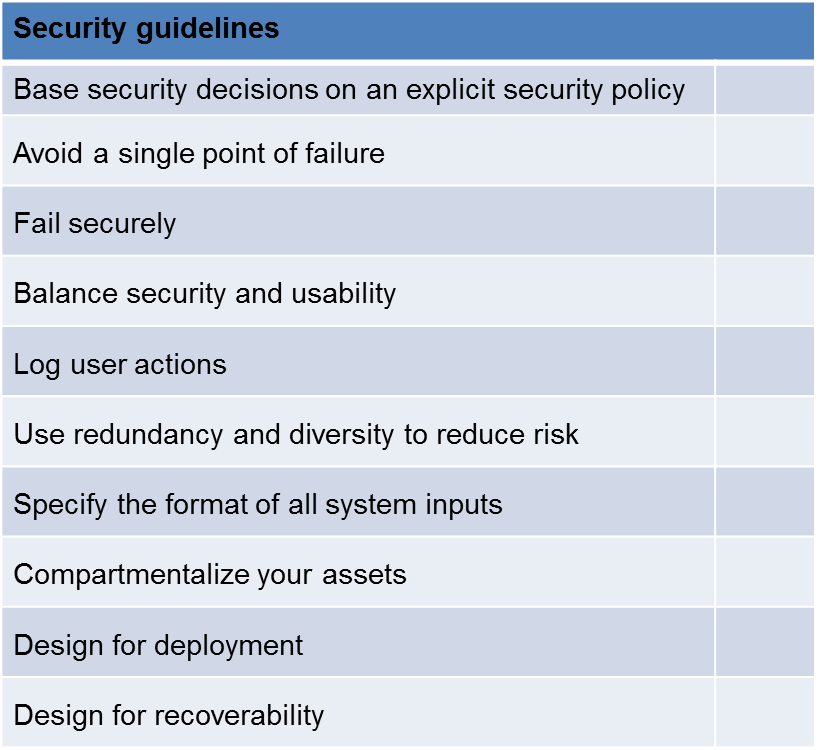
When this principle is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In this case instantiation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.

The Template Design Pattern is an example where the DIP principle is applied.

Of course, using this principle implies an increased effort, will result in more classes and interfaces to maintain, in a few words in more complex code, but more flexible. This principle should not be applied blindly for every class or every module. If we have a class functionality that is more likely to remain unchanged in the future there is not need to apply this principle.

**Layered Protection Architecture**

13.13 Layered Protection (14.4).eps



* **Base decisions on an explicit security policy**
  + Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems.
* **Avoid a single point of failure**
  + Ensure that a security failure can only result when there is more than one failure in security procedures. For example, have password and question-based authentication.
* **Fail securely**
  + When systems fail, for whatever reason, ensure that sensitive information cannot be accessed by unauthorized users even although normal security procedures are unavailable.
* **Balance security and usability**
  + Try to avoid security procedures that make the system difficult to use. Sometimes you have to accept weaker security to make the system more usable.
* **Log user actions**
  + Maintain a log of user actions that can be analyzed to discover who did what. If users know about such a log, they are less likely to behave in an irresponsible way.
* **Use redundancy and diversity to reduce risk**
  + Keep multiple copies of data and use diverse infrastructure so that an infrastructure vulnerability cannot be the single point of failure.
* **Specify the format of all system inputs**
  + If input formats are known then you can check that all inputs are within range so that unexpected inputs don’t cause problems.
* **Compartmentalize your assets**
  + Organize the system so that assets are in separate areas and users only have access to the information that they need rather than all system information.
* **Design for deployment**
  + Design the system to avoid deployment problems
* **Design for recoverability**

Design the system to simplify recoverability after a successful attack

