Table of Contents

What is of software and why is it different? 4

What is software engineering? 6

  Shellshock Bug (reported 2014-09-24) 6

Alternatives to software engineering 6

Why do we need requirements? 7

  Requirement Engineering 7

    Good Requirements 8

  Types of Requirements 9

  Phases of Requirements Development 9

test driven development 11

  goals of software engineering 12

  software engineering principles (Hooker) 12

  pair programming 14

  formal inspection 14

  software process 15

plan driven vs. agile approach 15

XP (eXtreme Programming) framework activities/phases 18

  XP values 18

    Planning 19

    Managing 19

    Designing 19

    Coding 19

    Testing 19

  User Stories 20

Coupling 20

Component-level Design Principles 22

  Component Packaging Principles 22

  Cohesion 23

  Kinds of cohesion 23

    system engineering phases 26

  system modeling 27

Common Subsystems 28
describe a solution for the observer problem 28
Gang of Four of design problem 29
Liskov Substitution Principle 29
Class Design 31
Abstract Data Types 31
Examples of Good and Bad Abstraction 31
Law of Demeter 33
accidents vs. essence 35
message-passing architectural style 35
Synchronous versus asynchronous message passing 36
repository architecture (DB) 37
layered architecture 37
client-server architecture 37
distributed architecture 38
Distributed architecture examples 39
fat vs. thin client 39
Centrally hosted thick client applications 39
multi-tiered-client server 40
peer-to-peer (P2P) 41
software as a service (SaaS) 42
management + architecture 42
construction decisions 42
design 42
important design concepts 43
design patterns 43
categories of design patterns 44
class + interface issue 44
RESTful Design Principles 72
Basic Principles 73
Resource and Resource Identifier: 74
Layered Architecture 76
Pattern Description 76

Key Concepts 79
Pattern Example 82
Considerations 84
Pattern Analysis 85
Waterfall Model 86
V Model 88
Spiral Model 90
Prototype 91
Question: Give the details of quality parameters which are used in a software system. 93
Question: Explain the term, software maintenance. 94
Reason of Maintenance of software 94
Types of software Maintenance 95
Question: What are the difference between alpha testing and Beta testing? 96
Binary Search 97
Pseudocode for binary search 98
Advantages of Binary Search Trees over Hash Tables 100
Difference between Web Services and Microservices 101
What is Web Service? 101
What is Micro Service? 101
Use-Case Representation 101
Difference Between Microservices and SOA 105
Feasibility Study 106
Types of Feasibility 106
Feasibility Study Process 108

----

project open-source only, typically BSD, GNU, Apache, MIT licensed

----

2. engineering process and practice (higher-level)

how software engineering works
almost social-science
how engineers work with/against each other to build something that works

how this process works, or doesn't work

instead of dealing with software, deal with the methods and ways of
developing the software

3. construction (lower-level)

know program you want to build, what are the methods and best practices
for
building this piece of software

actual construction details of software

- testing & debugging
- modularization techniques

may seem obvious but are actually not
may be confusing and have pitfalls

learn not to obvious things in 'obvious' things

What is of software and why is it different?

- 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**

   --

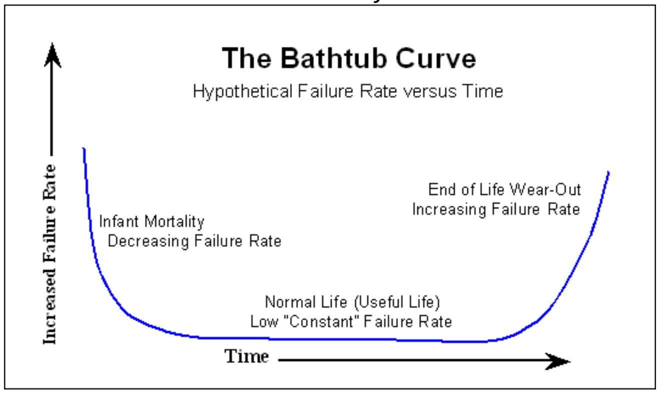
   don't have to worry about manufacturing
   - type **'cp'** or **'scp'** to copy program and ship

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

What is 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
```

Why do we need 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
```

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"
```

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

    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
```

XP values

```
----------
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

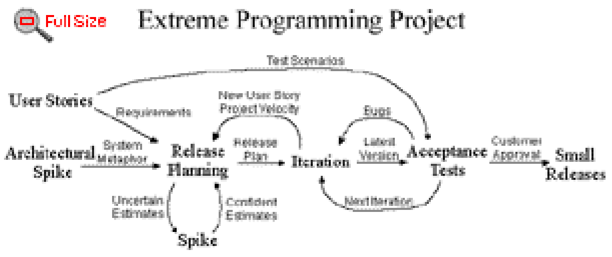
- User stories are written.
- Release planning creates the release schedule.
- Make frequent small releases.
- The project is divided into iterations.
- Iteration planning starts each iteration.

Managing

- Give the team a dedicated open work space.
- Set a sustainable pace.
- A stand up meeting starts each day.
- The Project Velocity is measured.
- Move people around.
- Fix XP when it breaks.

Designing

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.



Coding

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Set up a dedicated integration computer.
- Use collective ownership.

Testing

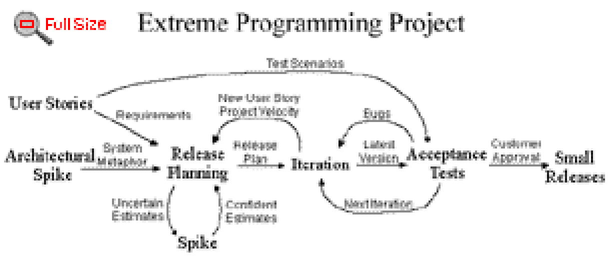
- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

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. They are also used instead of a large requirements document. User Stories are written by the customers 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. 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.



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 during release planning.

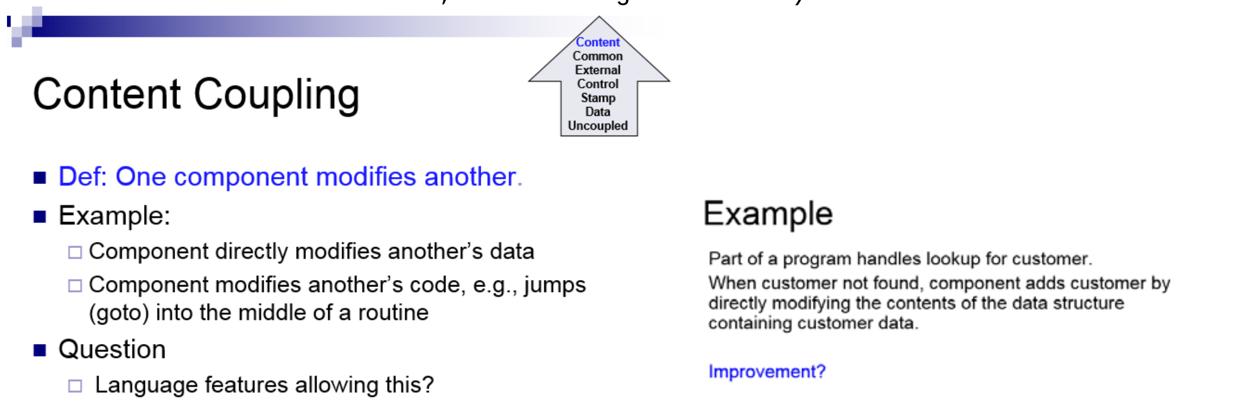
Another difference between stories and a requirements document is a focus on user needs. You should try to avoid details of specific technology, data base layout, and algorithms. You should try to keep stories focused on user needs and benefits as opposed to specifying GUI layouts.

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)



Content Coupling

Content
Common
External
Control
Stamp
Data
Uncoupled

- Def: One component modifies another.
- Example:
  - Component directly modifies another's data
  - Component modifies another's code, e.g., jumps (goto) into the middle of a routine
- Question
  - Language features allowing this?

Example

Part of a program handles lookup for customer.
When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

Improvement?

Component-level Design Principles

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

Component Packaging Principles

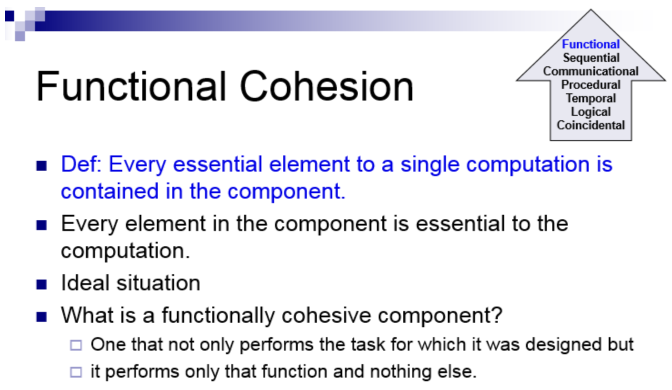
- **Release reuse equivalency principle**
  - The granularity of reuse is the granularity of <u>release</u>
  - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- **Common closure principle**
  - Classes that <u>change</u> together <u>belong</u> together
  - Classes should be packaged <u>cohesively</u>; 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 <u>reused</u> together should not be <u>grouped</u> 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)

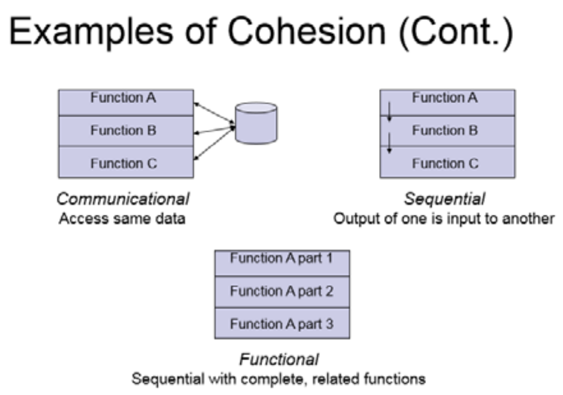
Kinds of cohesion



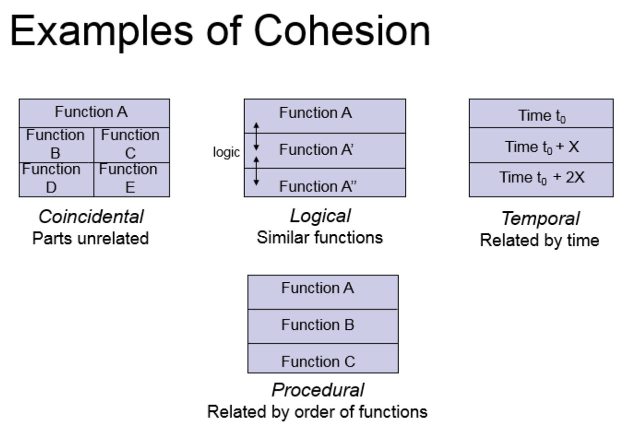
Functional Cohesion

Functional
Sequential
Communicational
Procedural
Temporal
Logical
Coincidental

- Def: Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation.
- Ideal situation
- What is a functionally cohesive component?
  - One that not only performs the task for which it was designed but
  - it performs only that function and nothing else.



Examples of Cohesion (Cont.)



Examples 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

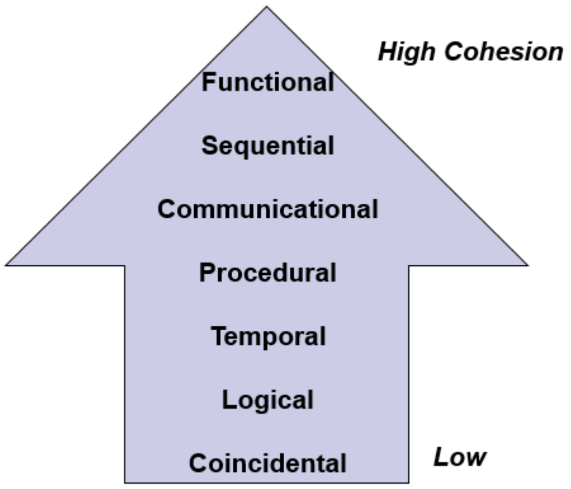
        - Update a record on a database
        - Print the record
    - Fetch unrelated data at the same time.
        - To minimize disk access
- **Procedural**
    - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no 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

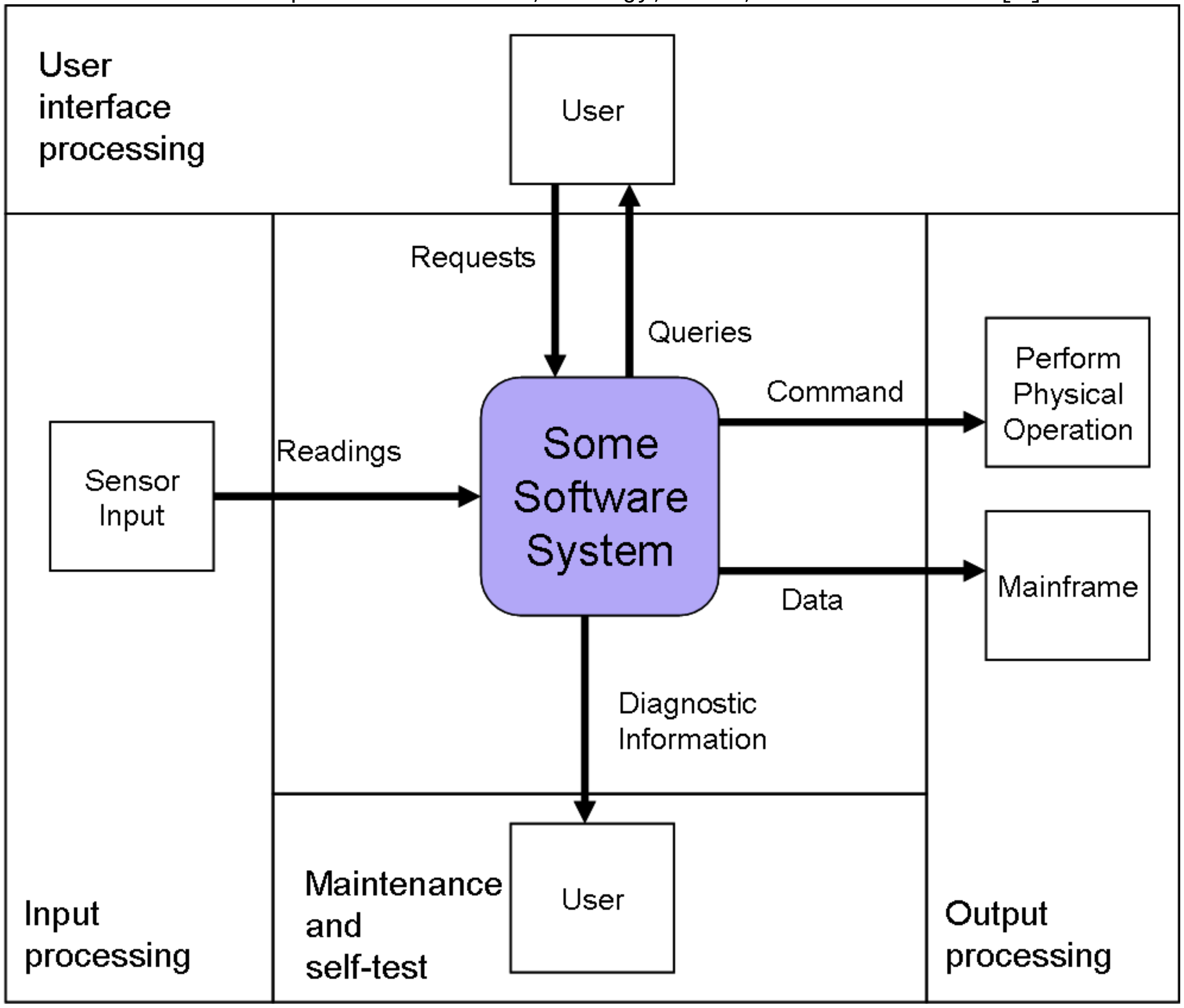
```
--------------------------
conceptual design => system vision (feasibility + proposal)
procurement => regulations, competition, budget, buy vs build
development => e.g. waterfall
operation => system in use, bug reports, fix things while system operates
             flexibility + adaptability are key
             assume mistakes will happen & system will evolve
```

system modeling

Hatley-Pirbhai model

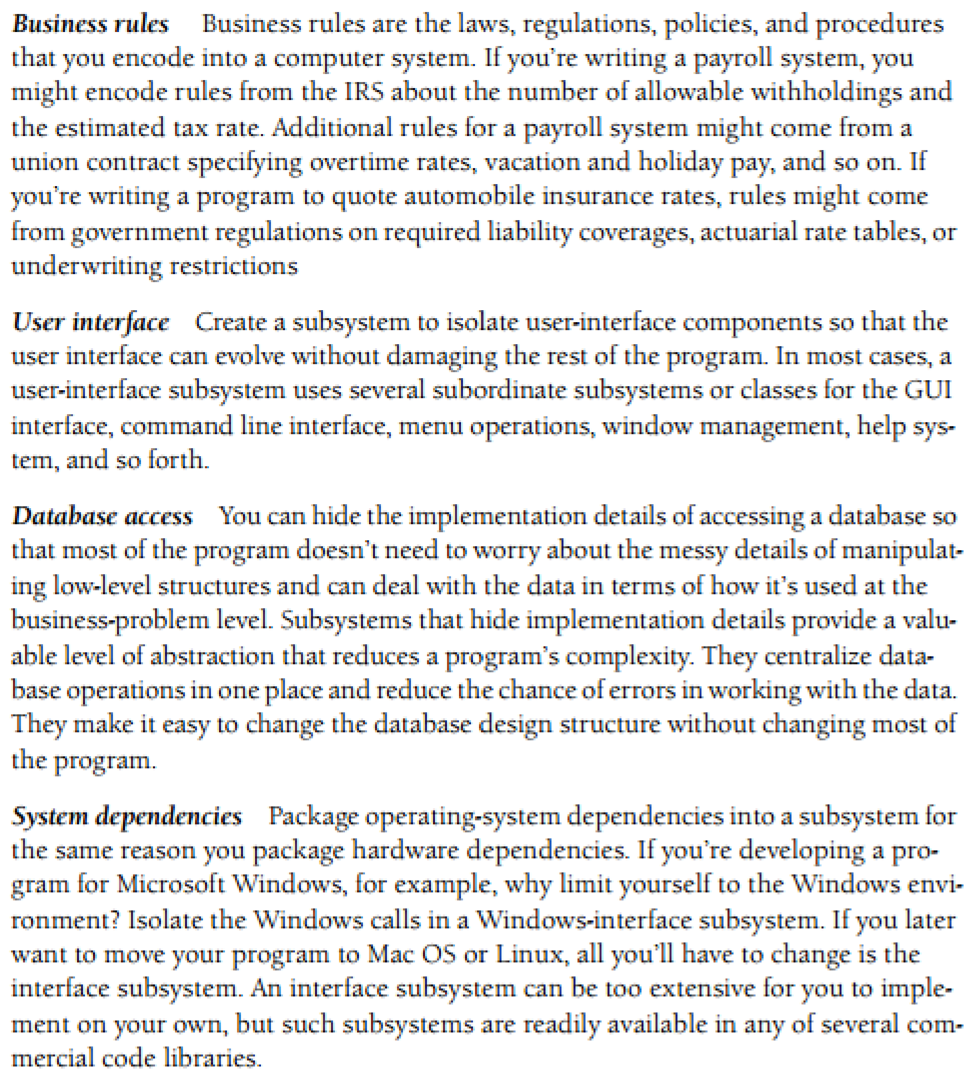
Hatley–Pirbhai modeling is a system modeling technique based on the input–process–output model (IPO model), which extends the IPO model by adding user interface processing and maintenance and self-testing processing.[1]

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



Business rules Business rules are the laws, regulations, policies, and procedures that you encode into a computer system. If you're writing a payroll system, you might encode rules from the IRS about the number of allowable withholdings and the estimated tax rate. Additional rules for a payroll system might come from a union contract specifying overtime rates, vacation and holiday pay, and so on. If you're writing a program to quote automobile insurance rates, rules might come from government regulations on required liability coverages, actuarial rate tables, or underwriting restrictions

User interface Create a subsystem to isolate user-interface components so that the user interface can evolve without damaging the rest of the program. In most cases, a user-interface subsystem uses several subordinate subsystems or classes for the GUI interface, command line interface, menu operations, window management, help system, and so forth.

Database access You can hide the implementation details of accessing a database so that most of the program doesn't need to worry about the messy details of manipulating low-level structures and can deal with the data in terms of how it's used at the business-problem level. Subsystems that hide implementation details provide a valuable level of abstraction that reduces a program's complexity. They centralize database operations in one place and reduce the chance of errors in working with the data. They make it easy to change the database design structure without changing most of the program.

System dependencies Package operating-system dependencies into a subsystem for the same reason you package hardware dependencies. If you're developing a program for Microsoft Windows, for example, why limit yourself to the Windows environment? Isolate the Windows calls in a Windows-interface subsystem. If you later want to move your program to Mac OS or Linux, all you'll have to change is the interface subsystem. An interface subsystem can be too extensive for you to implement on your own, but such subsystems are readily available in any of several commercial code libraries.

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 serveral 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
```

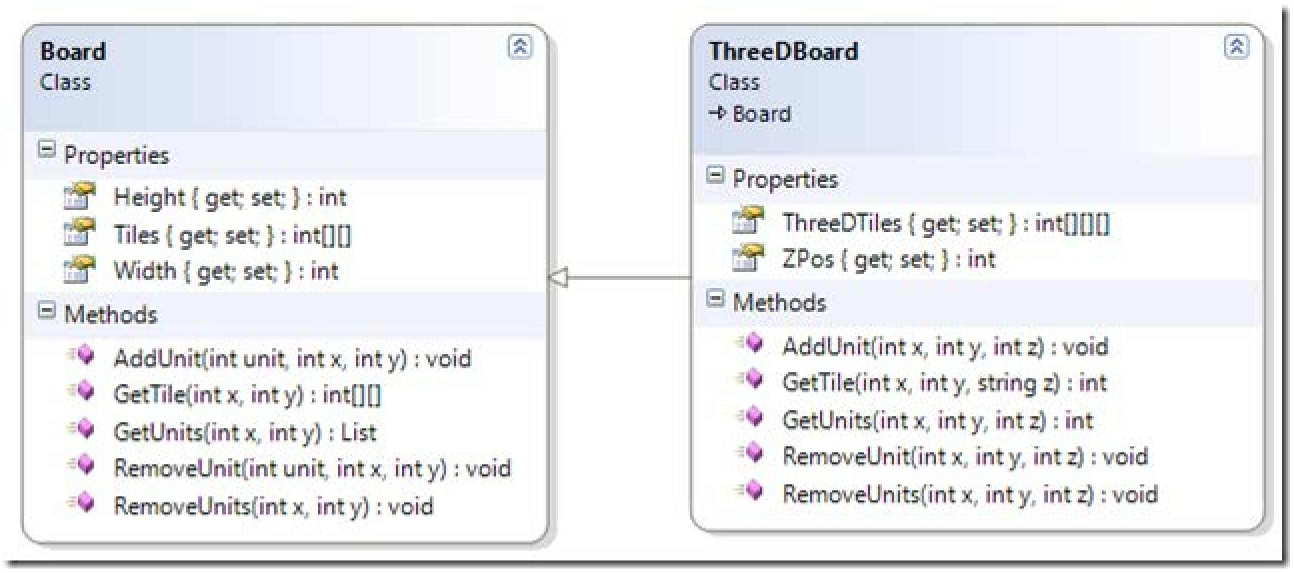
Liskov Substitution Principle

```
-----------------------------------------
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) 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. 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:


Board
Class
Properties
Height { get; set; } : int
Tiles { get; set; } : int[][]
Width { get; set; } : int
Methods
AddUnit(int unit, int x, int y) : void
GetTile(int x, int y) : int[][]
GetUnits(int x, int y) : List
RemoveUnit(int unit, int x, int y) : void
RemoveUnits(int x, int y) : void
ThreeDBoard
Class
→ Board
Properties
ThreeDTiles { get; set; } : int[][][]
ZPos { get; set; } : int
Methods
AddUnit(int x, int y, int z) : void
GetTile(int x, int y, string z) : int
GetUnits(int x, int y, int z) : int
RemoveUnit(int x, int y, int z) : void
RemoveUnits(int x, int y, int z) : void

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 `Width`properties 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

Suppose you're writing software that controls the cooling system for a nuclear reactor. You can treat the cooling system as an abstract data type by defining the following operations for it:

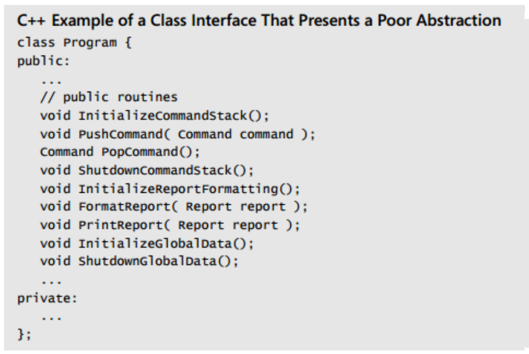
```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate( rate )
coolingSystem.OpenValve( valveNumber )
coolingSystem.CloseValve( valveNumber )
```

The specific environment would determine the code written to implement each of these operations. The rest of the program could deal with the cooling system through these functions and wouldn't have to worry about internal details of data-structure implementations, data-structure limitations, changes, and so on.

Here are more examples of abstract data types and likely operations on them:

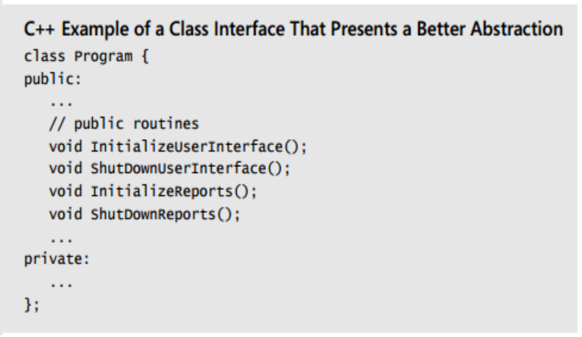
Cruise Control	**Blender**	**Fuel Tank**
Set speed	Turn on	Fill tank
Get current settings	Turn off	Drain tank
Resume former speed	Set speed	Get tank capacity
Deactivate	Start "Insta-Pulverize"	Get tank status
	Stop "Insta-Pulverize"	
List		**Stack**
Initialize list	**Light**	Initialize stack
Insert item in list	Turn on	Push item onto stack
Remove item from list	Turn off	Pop item from stack
Read next item from list		Read top of stack

Examples of Good and Bad Abstraction



C++ Example of a Class Interface That Presents a Poor Abstraction

```
class Program {
public:
   ...
   // public routines
   void InitializeCommandStack();
   void PushCommand( Command command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport( Report report );
   void PrintReport( Report report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
private:
   ...
};
```



C++ Example of a Class Interface That Presents a Better Abstraction

```
class Program {
public:
   ...
   // public routines
   void InitializeUserInterface();
   void ShutDownUserInterface();
   void InitializeReports();
   void ShutDownReports();
   ...
private:
   ...
};
```

C++ Example of a Class Interface with Mixed Levels of Abstraction

The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

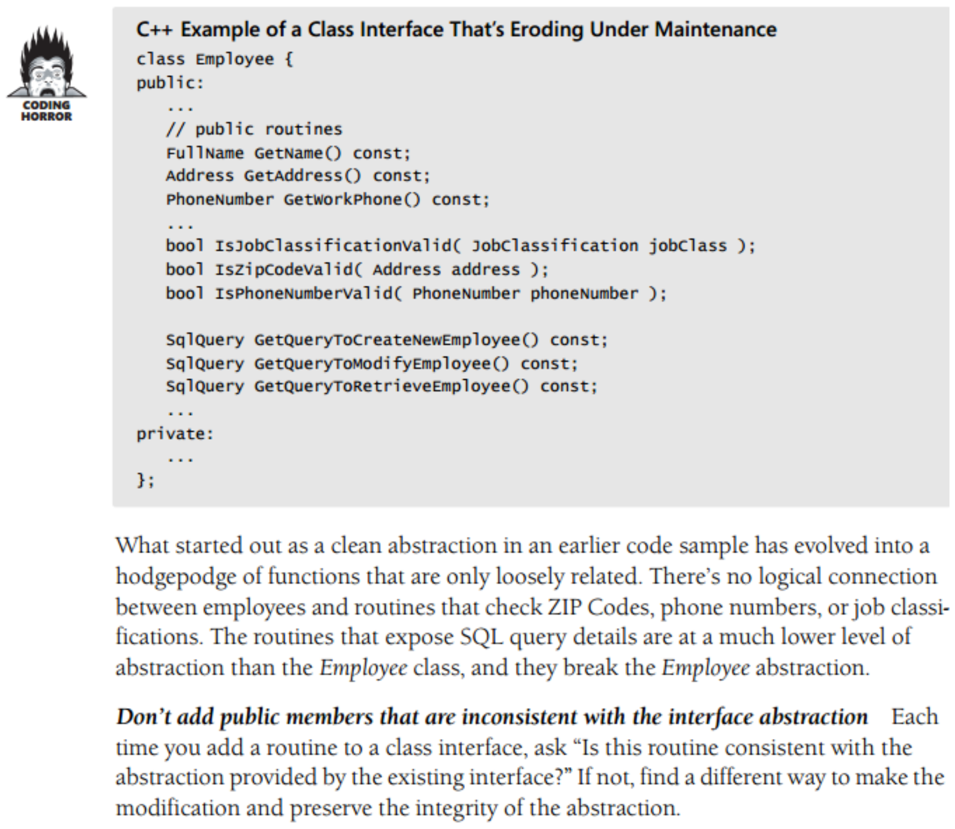
This class is presenting two ADTs: an *Employee* and a *ListContainer*. This sort of mixed abstraction commonly arises when a programmer uses a container class or other library classes for implementation and doesn't hide the fact that a library class is used. Ask yourself whether the fact that a container class is used should be part of the abstraction. Usually that's an implementation detail that should be hidden from the rest of the program, like this:

C++ Example of a Class Interface with Consistent Levels of Abstraction

The abstraction of all these routines is now at the "employee" level.

That the class uses the *ListContainer* library is now hidden.

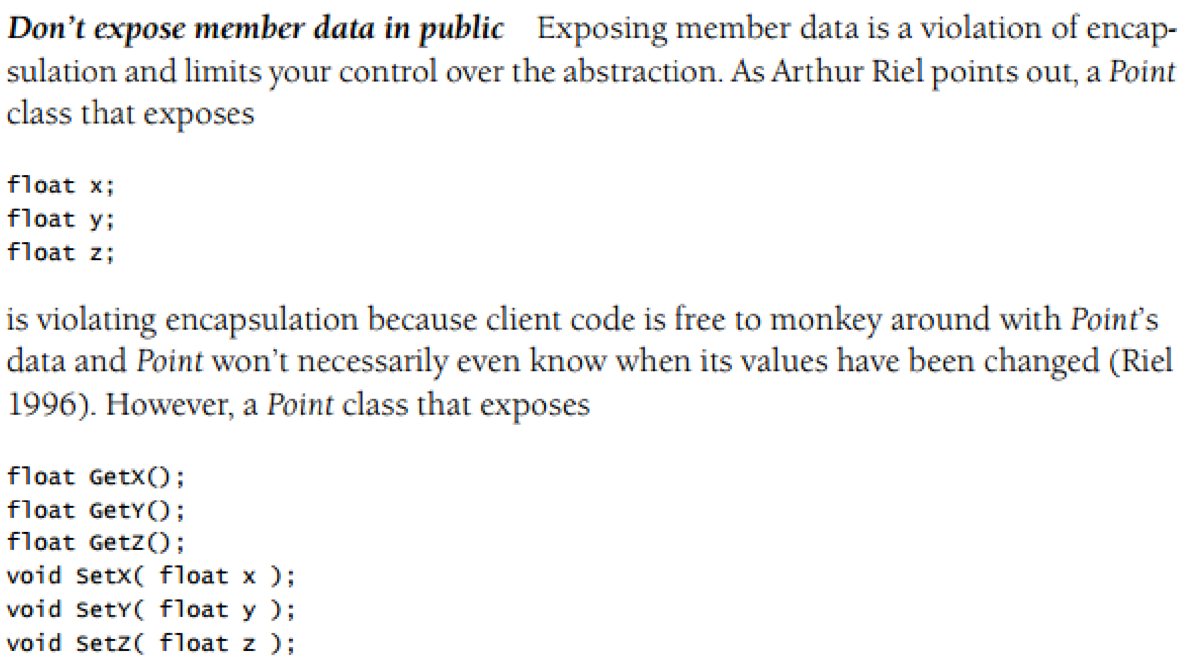
```
class EmployeeCensus {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );
   Employee NextEmployee();
   Employee FirstEmployee();
   Employee LastEmployee();
   ...
private:
   ListContainer m_EmployeeList;
   ...
};
```


C++ Example of a Class Interface That's Eroding Under Maintenance

```
class Employee {
public:
   ...
   // public routines
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;
   ...
   bool IsJobClassificationValid( JobClassification jobClass );
   bool IsZipCodeValid( Address address );
   bool IsPhoneNumberValid( PhoneNumber phoneNumber );

   SqlQuery GetQueryToCreateNewEmployee() const;
   SqlQuery GetQueryToModifyEmployee() const;
   SqlQuery GetQueryToRetrieveEmployee() const;
   ...
private:
   ...
};
```

What started out as a clean abstraction in an earlier code sample has evolved into a hodgepodge of functions that are only loosely related. There's no logical connection between employees and routines that check ZIP Codes, phone numbers, or job classifications. The routines that expose SQL query details are at a much lower level of abstraction than the *Employee* class, and they break the *Employee* abstraction.

Don't add public members that are inconsistent with the interface abstraction Each time you add a routine to a class interface, ask "Is this routine consistent with the abstraction provided by the existing interface?" If not, find a different way to make the modification and preserve the integrity of the abstraction.


Don't expose member data in public Exposing member data is a violation of encapsulation and limits your control over the abstraction. As Arthur Riel points out, a *Point* class that exposes

```
float x;
float y;
float z;
```

is violating encapsulation because client code is free to monkey around with *Point*'s data and *Point* won't necessarily even know when its values have been changed (Riel 1996). However, a *Point* class that exposes

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

Law of Demeter

When applied to 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, `A` might 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]

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, 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 and adaptable. 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] 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. 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.

A 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 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][4][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] 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 approach,[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][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 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 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 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]

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 properties on data.[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". One of the most common types of middleware to support asynchronous messaging is called 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]

Synchronous communication can be built on top of asynchronous communication by using a Synchronizer. 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 generally only provide a synchronous messaging primitive[citation needed]and asynchronous messaging can be implemented on top by using helper threads.

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.** 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.

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 applications, for example the X Window System, Citrix products and Microsoft's 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*]

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.

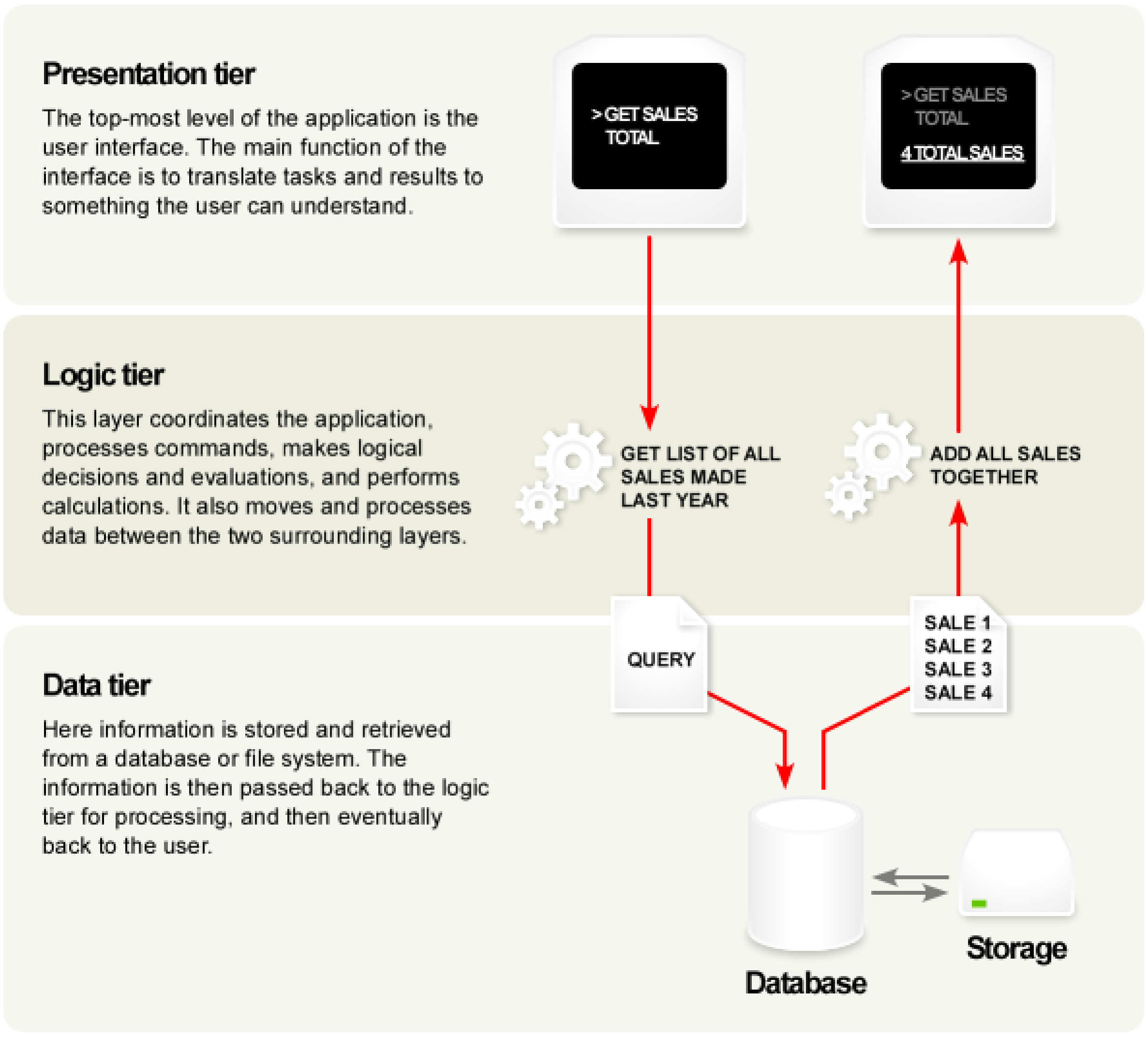
- **Better multimedia performance**. Thick clients have advantages in multimedia-rich applications that would be bandwidth intensive if fully served. For example, thick clients are well suited for video gaming.
- **More flexibility**. On some operating systems software products are designed for personal 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.


Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Database
Storage

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 class or Web Services protocol like 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

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!

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)
```

=============
May. 9, 2016
=============

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


==============
May. 11, 2016
==============

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

major issues in a organization
-------------------------------
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

deining 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) prgrammer 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


==============
May. 16, 2016
==============

2. what is the relationship between statistical testing
   and operational profiling uncertainty?

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

3. what is the relationship between experience-based testing
   and penetration testing?

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

dependibility
----
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
  likelyhood that software will not damage users either directly or indirectly
* reliability
  likelyhood that software will match expectations
* availability
  likelyhood that service will be up
* resilience
  likelyhood that software will resist and recover from damage

* maintainability
  likelyhood that software will stay working after change
* repairability
  how fixable when buggy
* user error tolerance

dependibility engineering
----
requirements
software architecture
dependible programming
software processes
  software development process/method
  can show why .999 depenidble
  can prove that software has reached dependibility goal

security
----
assume universe is malicious
paranoia is almost required here
security is a more difficult problem

often the most important
  can trump funcitonality
bad P.R. if it fails

cost ($) vs. dependibility (0-1): curve has asymtotic growth choosing position on curve is a hard design decision

specifying dependibility 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 depndability
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 terminlology

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 terminlogy

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

```
       up       down   up   down         up
---------------+    +------+    +-----------------
               +----+      +----+
                    MTBF
               |<--------->|
               |<-->|<---->|
                MTTR  MTTF
```

AVAIL = MTTF/MTBF >= 0.999 (3 9s)
POFOD = probability of failure on demand
ROCOF = rate of occurence of failure = 1/MTTF

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 probablities to branches of tree
  not always clear (correlation between events)
  can calculate prbability 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


==============
May. 18, 2016
==============

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
  specualrtive 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'
```

```
how to check this stuff?
-------------------------
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

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

    http://w3.org/WAI

    * 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 traning 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. 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:

```
<Car>
  <Make>Audi</Make>
  <Model>A5</Model>
  <Year>2013</Year>
</Car>
```

or in JSON

```
{
"Make" : "Audi",
"Model" : "A5",
"Year" : 2013
}
```

Representation with Linked Resources

Resource representations may contain links to other resources

e.g.

```
<Car>
  ...
  <Engine uri="http://www.automart.com/engine/1242"/>
  ...
</Car>
```

or in JSON

```
{
...
"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). 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.

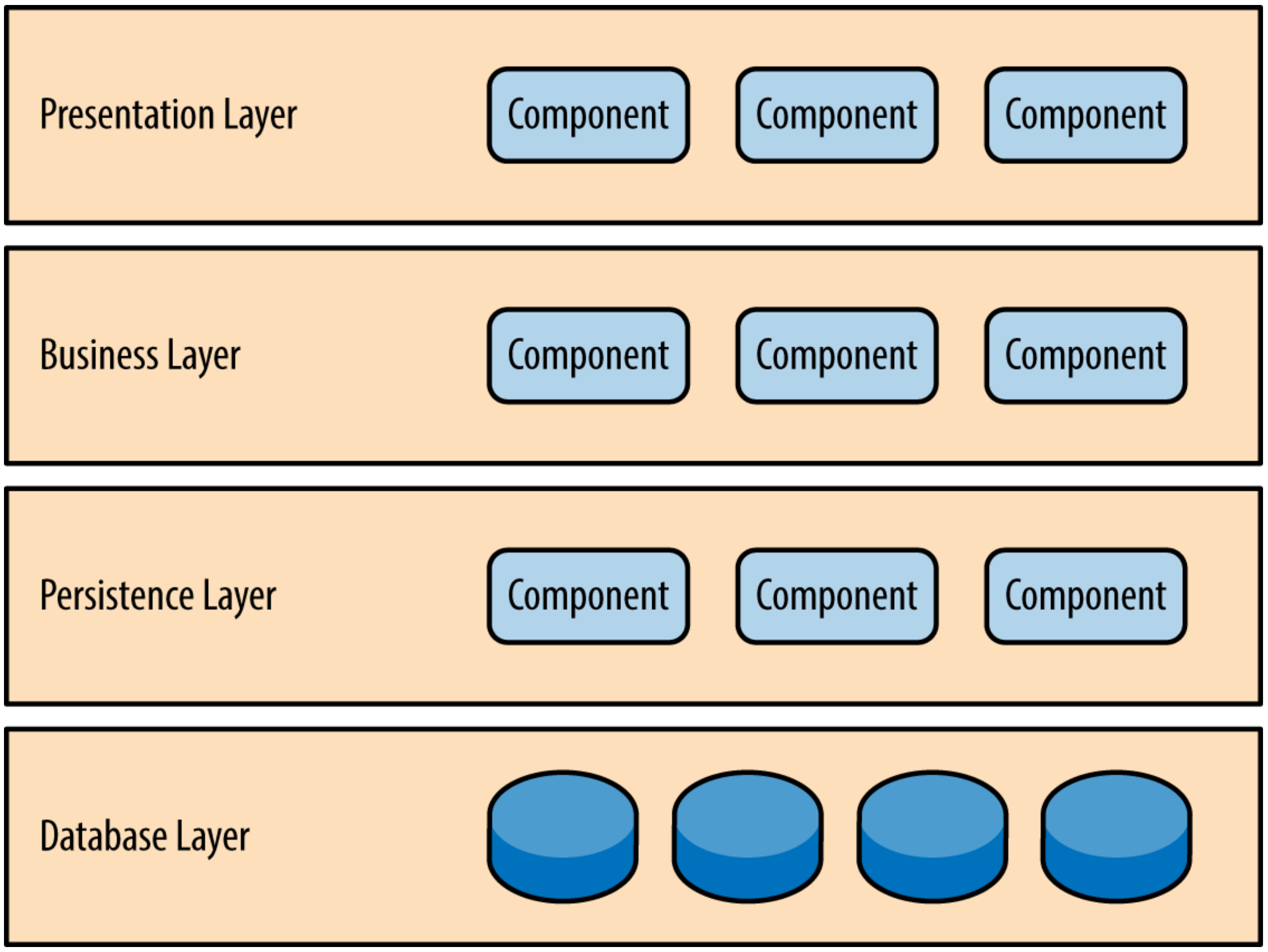


Figure 1-1. Layered architecture pattern

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 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.

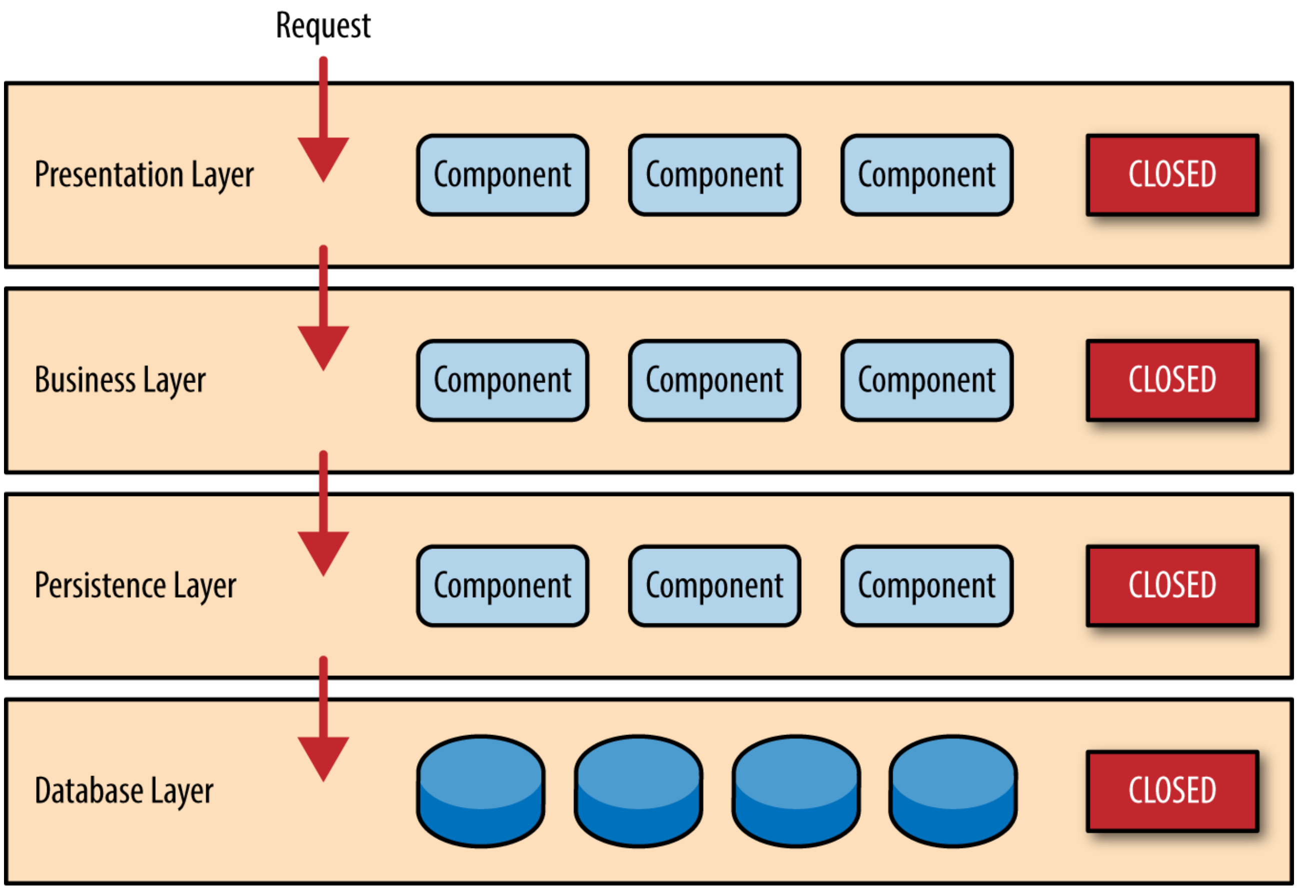


Figure 1-2. Closed layers and request access

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, 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.

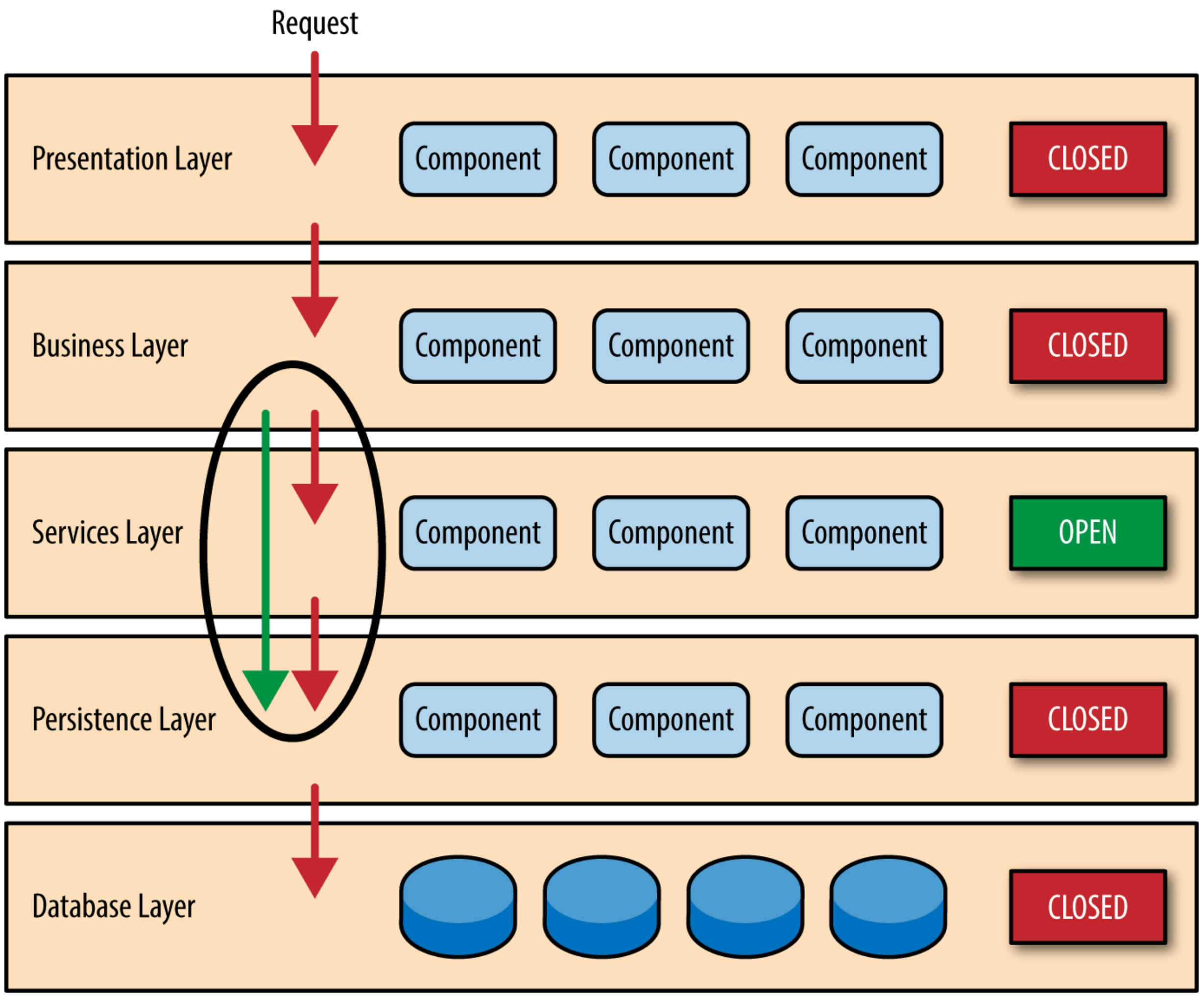


Figure 1-3. Open layers and request flow

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. 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.

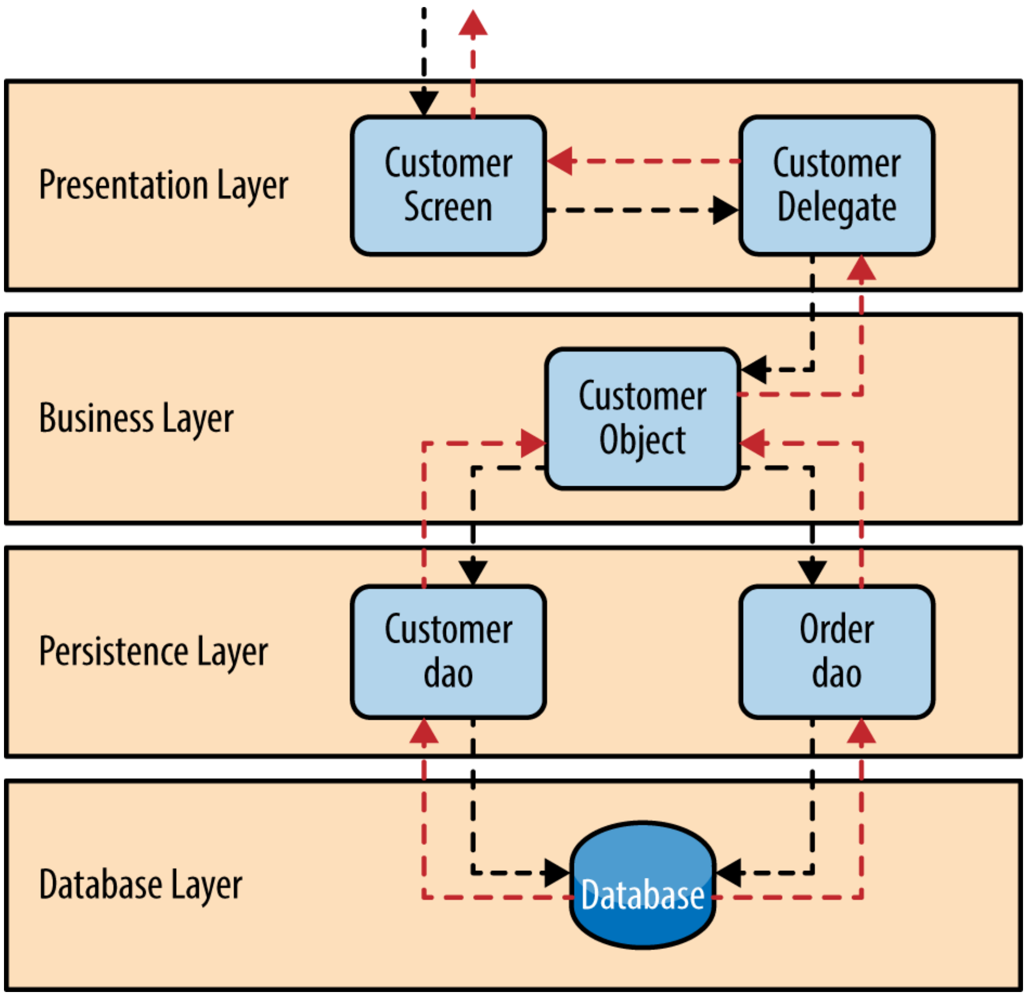


Figure 1-4. Layered architecture example

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 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

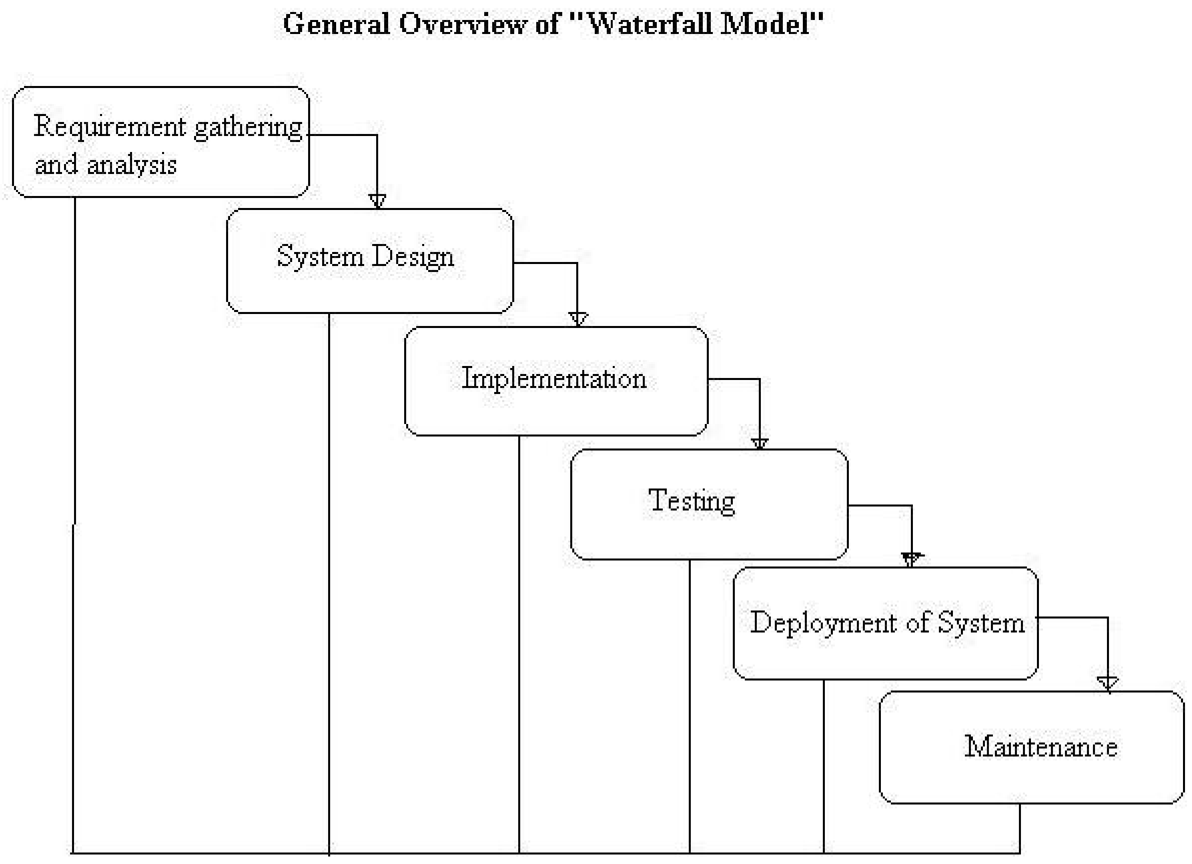
Rating: High

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** 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** starts only after the development is complete. In **waterfall model phases** do not overlap.

Diagram of Waterfall-model:


General Overview of "Waterfall Model"
Requirement gathering and analysis
System Design
Implementation
Testing
Deployment of System
Maintenance

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** 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:

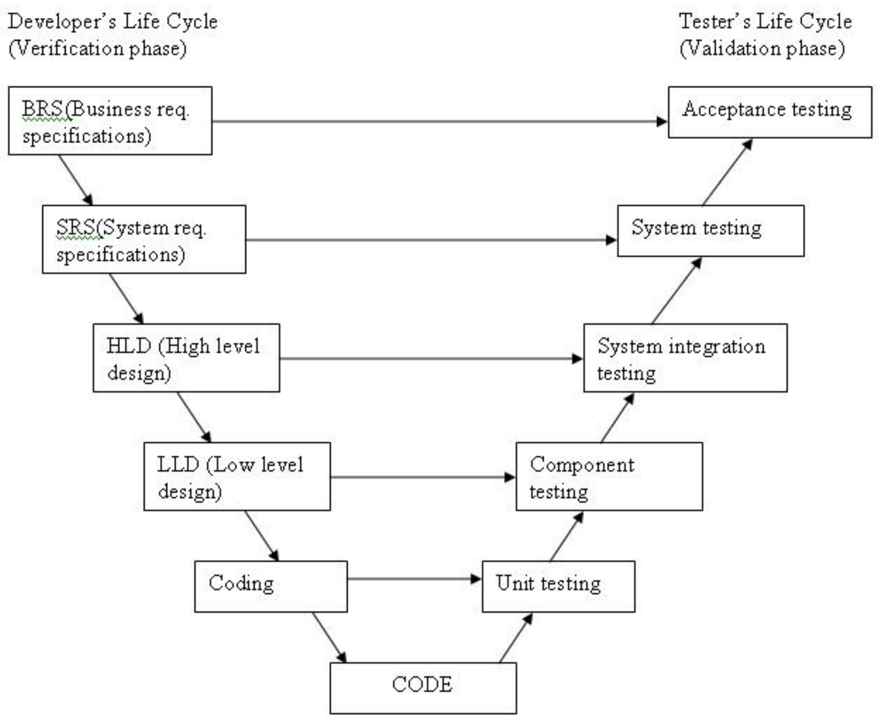
- This model is used only when the requirements are very well known, clear and fixed.
- Product definition is stable.
- Technology is understood.
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short.

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**, 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**.Testing of the product is planned in parallel with a corresponding phase of development in **V-model**.

Diagram of V-model:


Developer's Life Cycle (Verification phase)
Tester's Life Cycle (Validation phase)
BRS(Business req. specifications)
Acceptance testing
SRS(System req. specifications)
System testing
HLD (High level design)
System integration testing
LLD (Low level design)
Component testing
Coding
Unit testing
CODE

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** plan is created. The **test plan** 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** 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** 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** 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** 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.

Spiral Model

The spiral model is similar to the **incremental model**, 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** like **Waterfall**, **Agile**, **V-Model**.

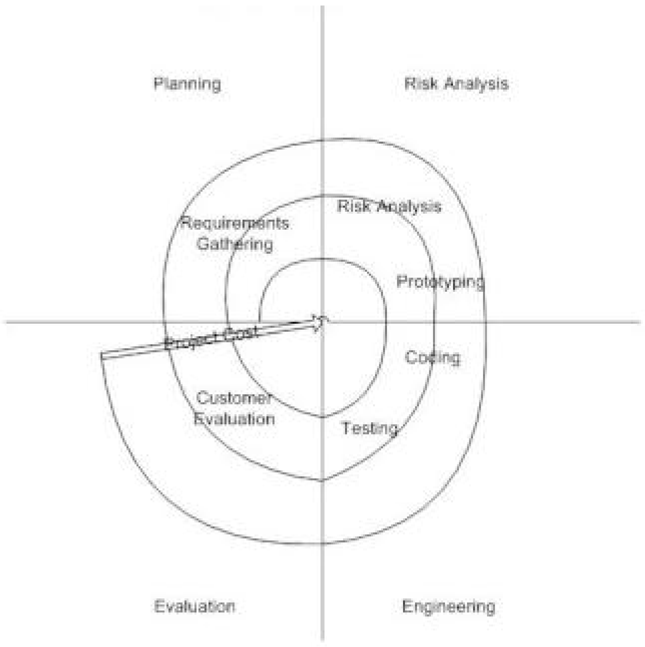
Planning Phase: Requirements are gathered during the planning phase. Requirements like 'BRS' that is 'Bussiness Requirement Specifications' and 'SRS' that is 'System Requirement specifications'.

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** at the end of the phase. Hence in this phase the development and testing is done.

Evaluation 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:



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**.

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:

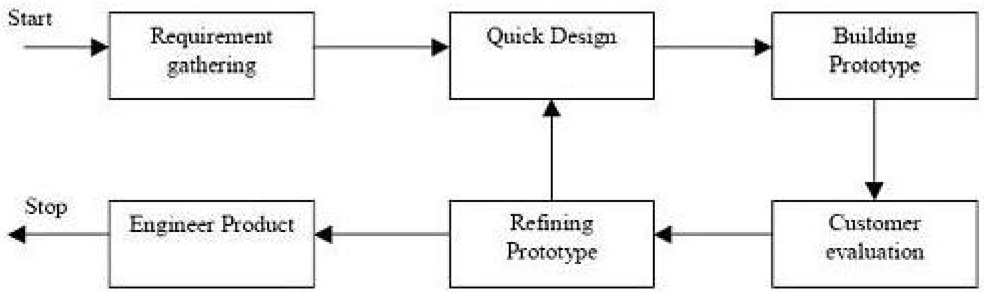
- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

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**. 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:



Prototyping 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.

Question: Give the details of quality parameters which are used in a software system.

Answer: - Following is the list of quality parameters:

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.

Question:Explain the term, software 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.

2. **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.

3. **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.

Question: What are the difference between alpha testing and 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 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.

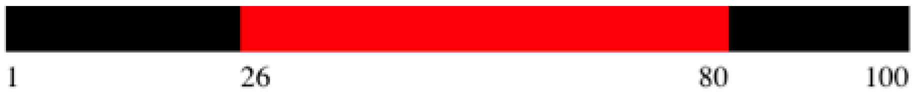
The next few articles discuss how to describe the algorithm carefully, how to implement the algorithm in JavaScript, and how to analyze efficiency.

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.

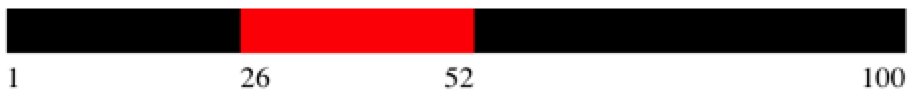
In order to implement an algorithm in a programming language, you will need to understand an algorithm down to the details. What are the inputs to the problem? The outputs? What variables should be created, and what initial values should they have? What intermediate steps should be taken to compute other 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. 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:



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)/2$left 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.



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 $\text{min}$$min$m, i, n be the current minimum reasonable guess for this round, and let the variable $\text{max}$$max$m, a, x be the current maximum reasonable guess. The *input* to the problem is the number $\text{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 $\text{min} = 1$$min=1$m, i, n, equals, 1and $\text{max} = \text{n}$$max=n$m, a, x, equals, n.
2. Guess the average of $\text{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

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).

Difference between Web Services and Microservices

Micro Services and Web Services are two different concepts of Application Development Architecture, which can be differentiated from its layered architecture and development style. This blog gives more details about these concepts and the difference between Web Services and Micro Services.

What is Web Service?

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 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.

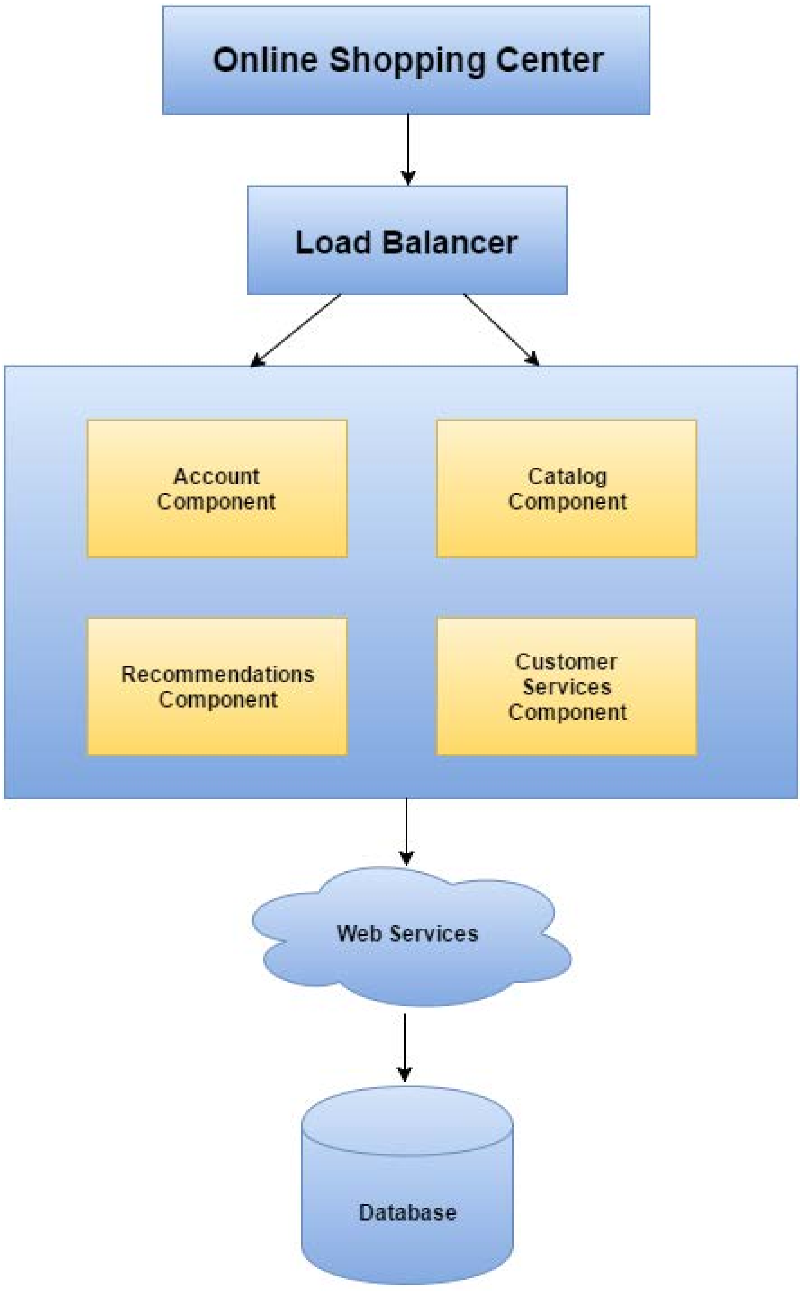


Figure 1: Conventional Approach

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.

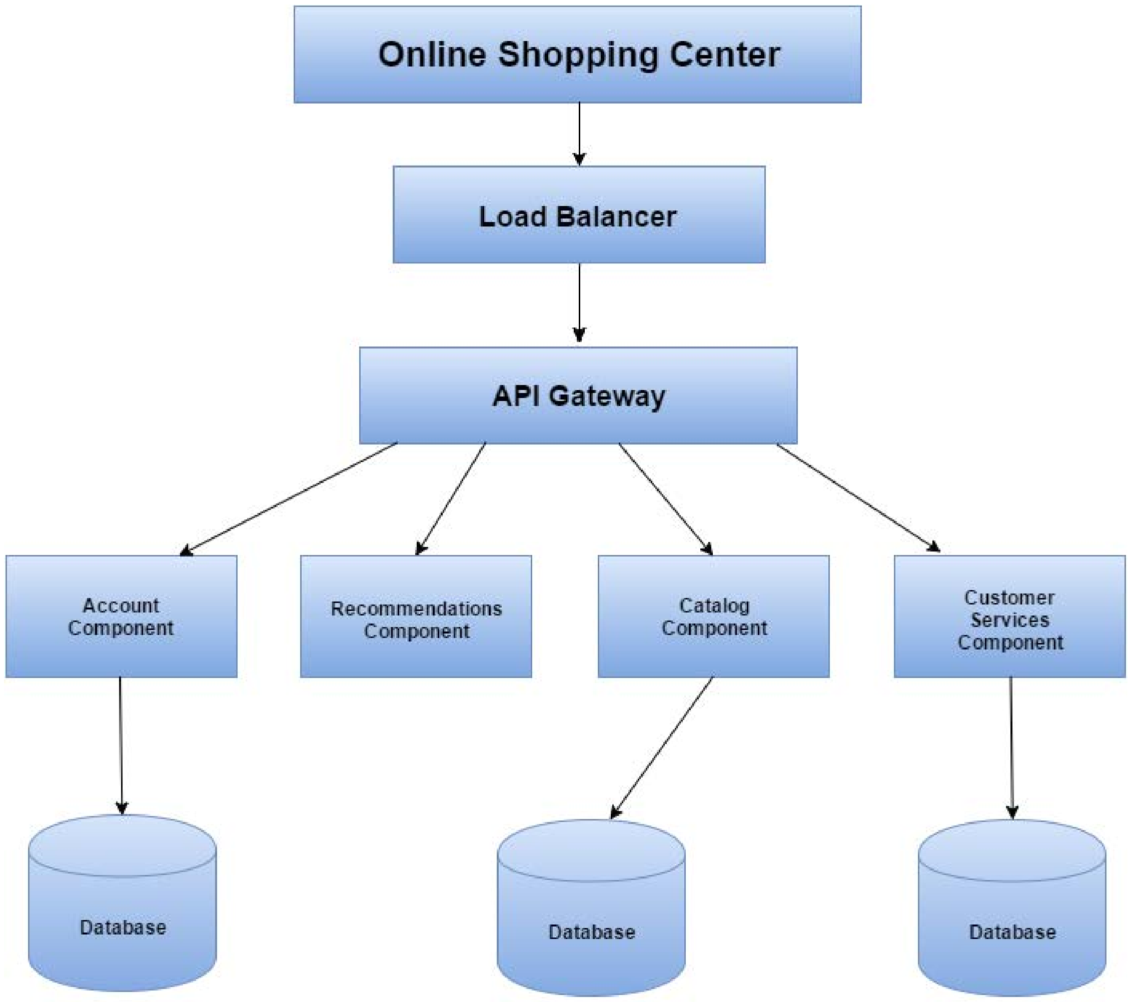
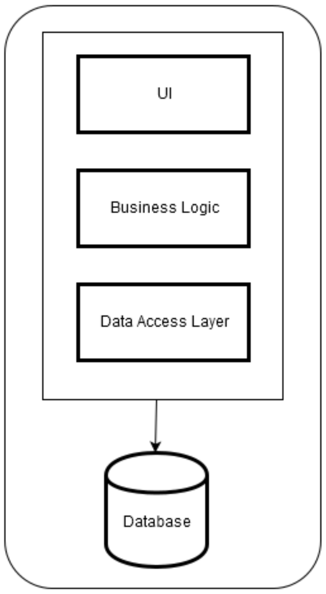


Figure 2: Micro Services Approach

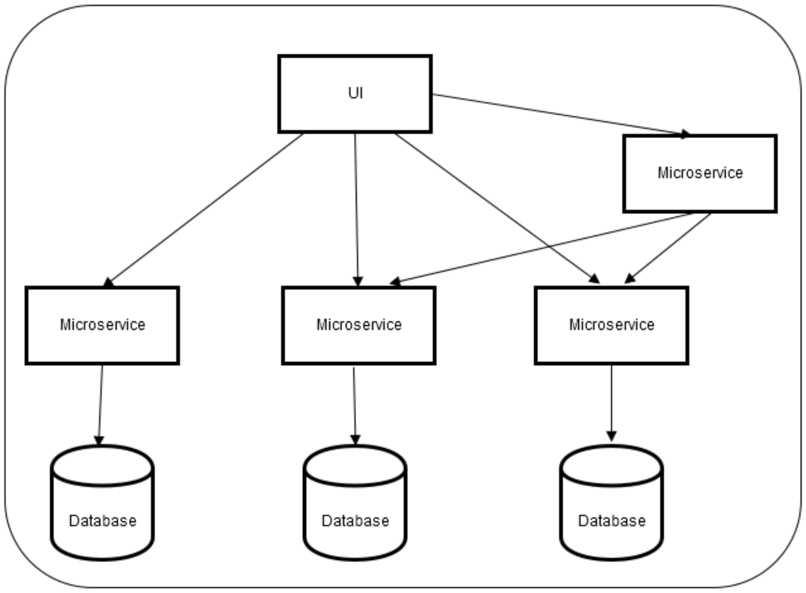
Web Services could be of any size, including large enterprise apps retrofitted with APIs that too many other apps depended on. Although "micro" in Micro Services, the basic concept is that each service performs a single function.

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.



Monolithic Architecture

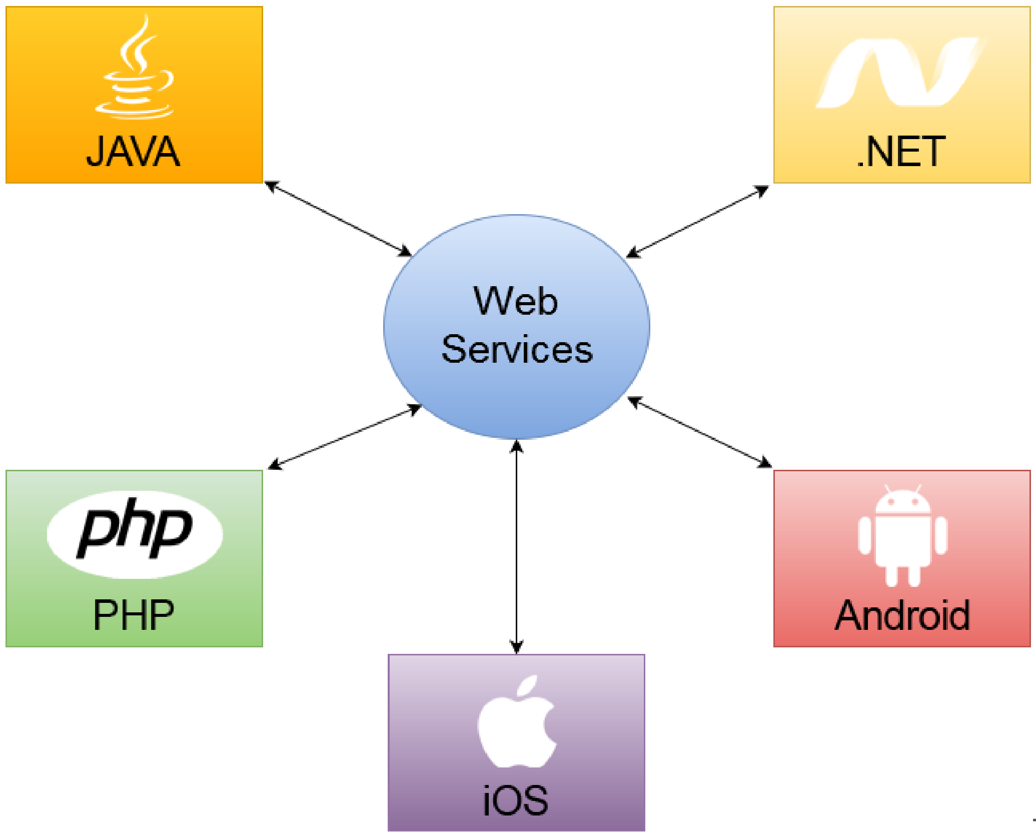


Microservices Architecture

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.

147down vote

I guess you could think of the Microservices Architectural Style as a specialisation of SOA. Don't forget that one of the accepted views 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, 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.

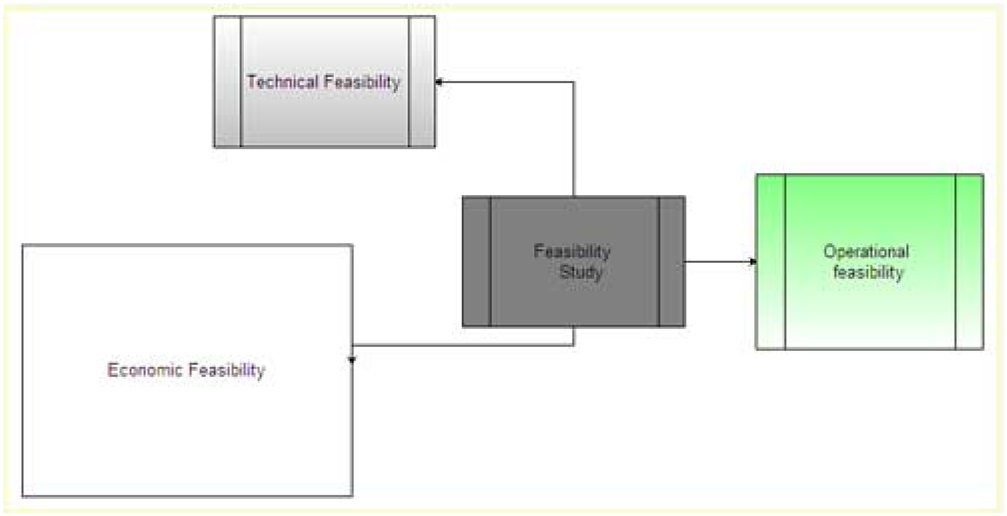
Feasibility Study

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.



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 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 contact to 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.