Intro

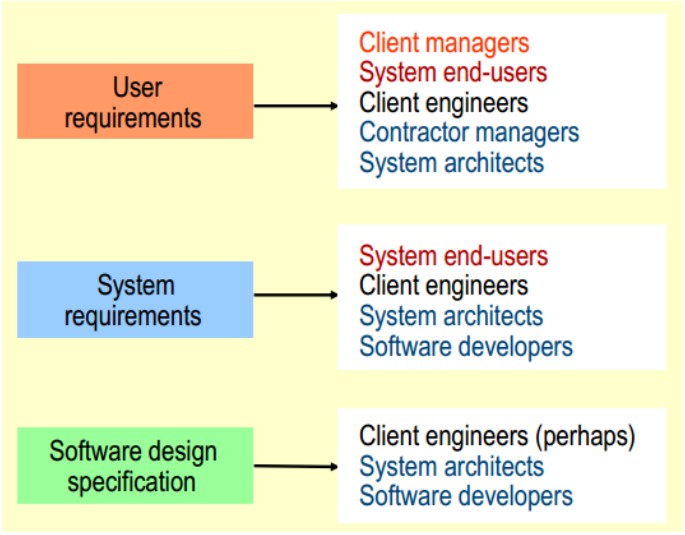
- Common problems
  - Complexity
  - Integration req
  - Quality req
  - Flexibility req
  - Portability and internalization req
  - Organizational req
    - SW projects usually fail due to communication problems, lack of education, high fluctuation, bad management
- Software Engineering := multi-person construction of multi-version software | engineering discipline concerned with all aspects of SW production
- System engineering: all aspects of computer-based systems dev including HW, SW, process engineering
- Good software: delivers required functionality & performance + MEDA
  - Maintainability
  - Efficiency
  - Dependability
  - Acceptability

Socio-technical systems

- System := purposeful collection of interrelated components working together to achieve some common objective
  - SW, HW, PEOPLE
- Technical computer-based systems: HW+ SW, but user not considered part of system
- Socio-technical system: technical systems + **operational processes & people**
  - governed by organizational rules and policies
  - understanding the context of software usage
- STS characteristics
  - Emergent properties
    - properties of the system as a whole due to component interaction
    - Program will only handle crisis they are programmed for!
  - Non-deterministic
    - input → output not consistent
    - system behaviour dependent on human behaviour → chaotic
  - Complex relationship with organizational objectives
    - extent to which system supports org objectives does not depend on system alone

THE SOFTWARE LIFECYCLE

Requirements engineering

- Schematic requirement engineering procedure
  - Inception – ask questions
    - basic problem understanding, stakeholder identification, comm and collab establishment
  - Elaboration
    - create analysis model that identifies data, function, and behavioural req
  - Negotiation
    - Determine each stakeholder's win conditions, negotiate win-win
  - Requirements management
    - spec write-up
    - validation (review mechanism)
    - Document versioning
- Req engineering = the process of eliciting
  - the services that the customer requires from a system
  - the constraints under which it operates and is developed
  - Requirements = descriptions of the system services and constraints that are generated during the req engineering process
- User requirements
  - Statements in natural language + diagrams of the system's services and op. constraints
  - written for customers, must use their language and mental models
  - Must be understandable by users w/o technical knowledge
  - Natural language < UML
- System req
  - A structured document w/ detailed descriptions of the system's functions, services and operational constraints
  - Defines what should be implemented so may be part of a contract btw client and contractor
  - Must be concise
- Requirement readers
  - 

- Functional and non-functional req
  - Functional

        - Statements of services the system should provide
        - how the sys should react to particular inputs
        - how the sys should behave in particular situations
    - Non-func
        - properties and constraints of system
            - properties: reliability, response time, storage req, ….
            - constraints: I/O device capability, system rep
            - domain const: security, legal impacts, …
        - constraints offered by the system (e.g. timing)
        - constraints on the dev process, standard, etc
            - e.g. particular CASE system, prog. language, dev method
    - Domain req
        - Req that come from the application domain of the system and that reflect characteristics of that domain
    - Requirements imprecision
        - ambiguous req → misinterpretation chance
- Req Doc wrap up
    - req doc = user req + system req
    - functional + non-func req
    - link customer and developer
    - NOT a design doc
    - for user req: avoid computer jargon
    - ADOPT COMPANY CONVENTION and use it for all req

UML

- UML = Unified Model Language
- Diagram perspectives
    - Conceptual, specification, implementation
- Main diagram types acc. to 80/20 rule:
    - use case diagram (functional)
        - use case = chunk of functionality; should contain a verb in its name
        - actor = someone or some thing interacting with system under dev
        - visualize relationships between actors and use cases
        - capture high-level alternate scenarios, get customer agreement
    - activity/action diagram (behavioral)
        - represents the overall flow of control
        - graphical workflow of activities and actions
        - **user-perceived actions → how do actors interact?**
    - class diagram (structural)
        - class = collection of objects with common structure, common behave, common relationships, common semantics
        - displayed as a bo with up to 3 compartments: name, list of attr, list of ops
        - modeling elements:
            - classes with structure + behaviour
            - relationships
            - multiplicity and nav indicators
            - role names
        - Relationships: modelling interaction between two objects

                - association – bidirectional connection between classes
                - aggregation – stronger form: “has a”; relationship between a whole and its parts
                - Dependency – weaker form
            - Multiplicity numbers and intervals denote number of instances that can/must participate in relationship instance
        - state diagram (behavioral)
            - show life history of a given class
            - use for classes that typically have a lot of dynamic behaviour
            -
        - seq diagram (behavioral)
            - displays obj interactions arranged in a time seq
            - can be from user’s perspective
            - SD for every basic flow of every use case, and for high-level, risky scenarios
            - **actors + system components → how do components interact?**
        - object diagram (str)
        - collaboration diag (str)
        - package diag (str)
        - deployment diag (str)
- UML 2.0: Model-Driven Architecture
    - Infrastructure
    - Superstructure
    - Object Constraint Language
    - Diagram Interchange
    - Vision
        - UML spec → platform-indep. model → target model → Implementation
    - Main goal speed up process, higher quality, reusability, long-term usability
- DSLs
    - Domain-specific modelling languages
    - DSLs better for embedded systems (clearly delimited app domain & paths)
    - UML better for enterprise apps (very flexible)

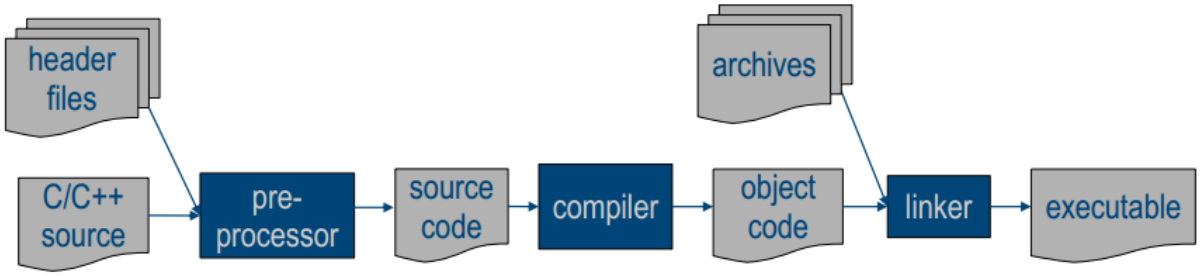
Design patterns

- Syntax: grammar, semantics: what to express, pragmatics: how to apply,
  meta lang: describe the language of discourse
- Pattern = description of the problem and the essence of its solution
- design pattern = way of re-using abstract knowledge about a sw design problem and its solution
- Pattern elements
    - Name
    - Description
    - Problem/applicability description
    - Solution description
    - Consequences
- Types of patterns
    - Creational
        - Abstract factory: creates an instance of several fam of classes
        - Builder: separates obj construction from its representation

        - Factory method: creates an instance of several derived classes
        - Prototype: a fully init'd instance to be copied or cloned
        - Singleton: A class of which only a single instance can exist
    - Structural patterns
        - Adapter: match interfaces of diff classes
            - lets classes work together
            - need to use an existing class whose interface does not match, make use of incompatible classes
            - class adapter commits to the concrete adapter class
        - Bridge: separates an object's interface from its implementation
        - Composite: a tree structure of simple and composite objects
            - compose objects into tree structures to represent part-whole hierarchies
            - lets clients treat indiv. obj and compositions of obj uniformly
            - any time there's a partial overlap in the capabilities of obj
        - Decorator: add responsibilities to objects dynamically
        - Façade: a single class that represents an entire subsystem
            - provides a unified interface to a set of interfaces in a subsys
            - defines a higher-level interface that makes subsys easier to use
            - applicability: need to provide a simple interface to a complex system, decouple a subsys from its clients, provide an interface to a software layer
            - Consequences: shields clinets from subsys components, promotes weak coupling between the subsys and its clients
            -
        - flyweight: a fine-grained instance used for efficient sharing
        - proxy: an object representing another object
            - provide a surrogate or placeholder for another object to control access to it
            - problem/applicability: remote proxies can hide the fact that a real obj is in another address space;
                - viritual proxies can create expensive objects on demand
                - protection proxies can control access to an obj
                - smart references can perform additional action above a simple pointer
    - Behavioural patterns
        - Chain of resp. : a way of passing a request btw a chain of objects
        - Command: encapsulate a command request as an obj
        - interpreter: a way to include lang elem in a program
        - iterator: sequentially access the elems of a collection
        - mediator: defines simplified comm between classes
            - define an object that encapsulates how a set of obj interact
            - promotes loose coupling by keeping obj from referring to each other explicitly
            - Problem: complex interaction exists
            - Consequences: limits subclassing; decouples colleagues; simplifies obj protocols; abstracts how obj cooperate; centralizes control
        - Memento: capture and restore an object's internal state

        - Observer: a way of notifying change to a number of classes
        - State: alter an obj' behaviour when its state changes
        - Strategy: encapsulates an algorithm inside a class
        - Template method: defer the exact steps of an algo to a subclass
        - Visitor: defines a new operation to a class without change
- Summary
    - design patterns = generic, re-usable design templates for OOP
    - creational, structural, behavioral

Compiling and linking C/C++ Programs



- The C Preprocessor
    - Purpose:
        - define commonly used constants, code frag, etc.
        - conditional compilation
            - include guard in header files
    - Main mechanism: replace by textual substitution
    - Preprocessor directives: #....
        - #include
        - can also pass definitions:
          cc -DCOMPILE_DATE=\"'date'\" -DDEBUG
    - Use parentheses!!
        - bad:

```
#define mult(a,b) a*b
main()
{
printf( "(2+3)*4=%d\n", mult(2+3,4) );
}
```

```
printf( "(2+3)*4=%d\n", 2+3*4 );
```

        - good:

```
#define mult(a,b) ((a)*(b))
main()
{
printf( "(2+3)*4=%d\n", mult(2+3,4) );
}
```

```
printf( "(2+3)*4=%d\n", ((2+3)*(4)) );
```

- The C(++) compiler
    - Task: Generate relocatable machine code from source code
        - Relocation := code can sit at different places in address space
    - Address space classified into segments
    - OS uses this to implement user address space
        - actual main memory address = base address + relative address
        - base address kept in seg register, added dynamically by CPU
        - security: program cannot access base register, hence cannot address beyond its seg limits

- Object files
  - Contain code for a program fragment
    - e.g. machine code, constants, size of static data segments, …
- External functions and variables
  - module server: variable sema allocated in data seg
  - module client: functions objtain sema address by:
    module server offset + local address sema
  - Cross-module addressing rules:
    - No modifier = locally allocated, globally accessible
    - static = locally alloc, locally accessible
    - extern = allocated in other compilation unit
- Name mangling
  - Problem: classes convey complex naming, not foreseen in classic linkage
  - Solution: name mangling
    - compiler modifies names to make the unique
    - every compiler has its individual mangling algorithm → code compiled with diff compilers is incompatible
  - telling a C++ compiler that a code is C:

```
extern "C"
{
        void flatFunc();
}
```

- The linker/loader
  - Task: generate one exec file from several obj and library files
    - Read obj files → resolve addresses from relocatable code → link runtime code → add all code needed from libraries → if required: establish stubs to dynamic libraries → write exec code into file, set magic number, etc.
  - cc, g++, etc. have complex logics inside
    - can silently invoke linker, don't link themselves!
    - shorthand: cc -o x x.c
- Strip
  - by default, exec contains symbol tables
    - function names, addresses, parametrization, static var, etc.
  - disadvantages: allows reverse engineering, substantially larger code files
  - before shipping: strip executables
- Libraries
  - Library = archive file containing a collection of obj files
    - code frags
    - ar rv libxxx.a file1.o file2.o
  - Obj files vs. libraries: obj file linked in completely;
    from library only what is actually needed
  - Static vs. dynamic:
    - static – code is added to the exec, just like obj file; not needed after linkage
    - dyn – only stub linked in, runtime system loads; needed at runtime (version!)
  - Naming conventions in linux
    - static: libxxx.a
    - dyn: libxxx.so
    - link with: ld … -lxxx
  - Dynamic libraries

        - finding them:
            - LD_LIBRARY_PATH variable
        - detecting usage:
            - ldd rasserver #in terminal
- Schematic program run

OS	**App program**
Open file	Set up runtime env
Look at first page: magic no., seg size, etc.	Call main()
Allocate segments	on system calls, interrupts, etc.: pass control to OS
Read code file into code seg	Upon exit() | main()'s return | forced abort: -- Cleanup, pass back to OS
Set up process descriptor	
Pass control to this process	
Handle system calls	
Terminate program, free process slot and resources	

- Summary
    - Creating an exec program requires running:
        - preprocess – textually expands definitions, condition-guarded code pieces
        - compile – translates source code into relocatable machine code
        - link – bind object files and archives into exec program
    - Code quality
        - set compiler to max rigidity: cc -W -Wall
        - eliminate **all warnings**

Defensive Programming

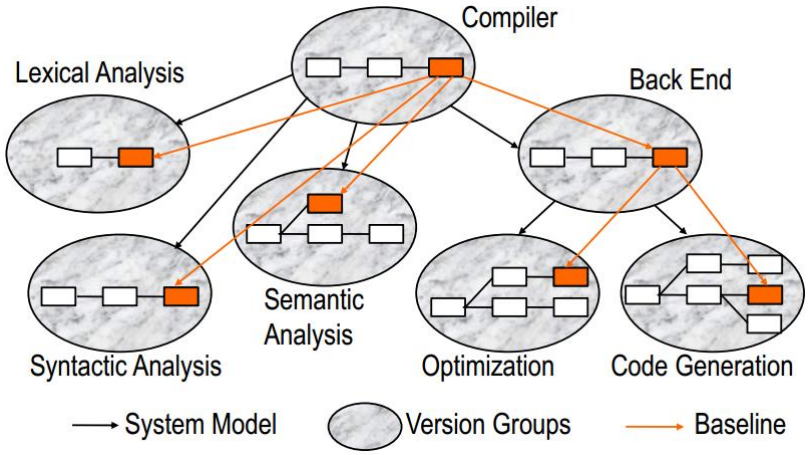
- Prevention is better than cure
- ensure the continuing function of SW in spite of unforeseeable usage of said software
- Invariants
    - Conditions that do not vary
    - loop invariants
    - class invariants: true before and after each method call
        - all constructors should place their object in a valid state
        - all methods should leave their object in a valid state
    - method invariants: pre and post conditions
        - "design by contract
        - methods are contracts with the user
        - users must meet preconditions of the method,
        - method guarantees post-conditions through frequent checks.
- Enforcing invariants – error handling
    - assertions = force-terminate program
        - for programmer errors that don't depend on end-user, non-pub member functions
        - assert() macro
        - if argument is false, print expression, file, line no., then calls abort()
        - handling: enabled by default; can turn off with NDEBUG
        - NEVER EVER USE IT IN A SERVER

    - exceptions = break flow of control
        - for preconds on public member functions
        - unless matching try/catch block exists, abort program
        - exceptions are classes; throw() instantiates the exception object
        - can have multiple catch() sensitive to exception type
    - return codes = data-oriented, keep flow of control
        - Document clearly!
- Structured programming
    - = component-level design technique using only small set of programming constructs
    - principle: building blocks to enter at top & leave at bottom
        - good: sequence; condition; repetition
        - bad: goto; break; continue;
    - Adv: less complex code → easier to read + test + maintain
        - measurable quality → complexity
    - Unstructured loops = nightmare; mainly solved by banning GOTO
        - POINTER IS THE DATA EQUIVALENT TO GOTO
- Code guides
    - = set of rules to which programmers must (should) adhere
    - twofold purpose:
        - have uniform style
        - codify best practice
    - e.g.
        - Reflect before typing
        - be pedantic
        - design cost-aware

Configuration, version, and release management

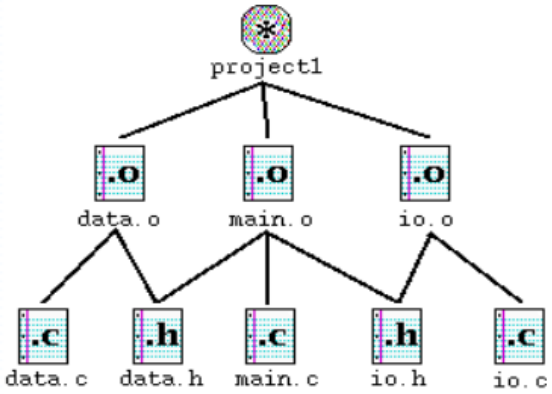
- Symptoms of the change chaos
    - problems of identification and tracking
        - who changed what and when?
    - problems of version selection
        - merging
        - what is the latest “working” version?
        - version update documentation
    - software delivery problems
        - configuration state
        - version skipping
- Software configuration management = discipline of controlling the evolution of software systems
- Three classic CM problems
    - the double maintenance problems
        - independent copies needing to be separately updated
    - the shared data problem
        - two or more devs with access to same file/data
    - the simultaneous update problem
        - two or more devs updating simultaneously
- Versions
    - Checkout = from database to workspace
    - Checkin = from workspace to database

- Terminology
  - version := revision | variant
  - revision = a software obj that was created by modifying an existing one
  - variant = two software obj sharing an important property, differing in others
  - release = a version that has been made available to user/client
  - configuration = combination of components into a system according to case-specific criteria
  - baseline = a static reference point for any configurable items in your project
    - doesn't imply perfection; only stability used for validation
- Using deltas
  - delta = difference between two revisions
  - can be computed in fwd or bwd direction on checkin
  - used to regen one revision from another
- Version merging
  - structural merges
- Version selection for configuration



- Configuration models
  - composition (module-oriented aspect)
    - config = set of software objects
  - change set (dynamic aspect)
    - configuration = bundle of changes
  - long transaction (collaborative aspect)
    - config = all changes are isolated into transactions
  - Common understanding: config = selection of components from repository which together make up a release '
- rule-based config building
  - baseline: no versions, no selection needed
  - developer: select all versions checked out by me + newest revision on main branch for others
  - cautious developer: select last baseline + all versions checked out/in by me
  - reconfig: select according to … + variants by attribute
  - time machine: select according to … + ignore everything after a cutoff date
  - new release: select according to… + the newest, stable versions associated with a given config of change requests
- Tools
  - SVN
  - make

- controls generation of derivs from their prerequisites
- config rule base: builtin, Makefile, makefile
- Macro: aka variable
    - SRC = myclass1.cc myclass2.cc
    - gcc -o $(OBJECTS) …
        - $@ full name of current target
        - $? current dependencies out of date
        - $< source file of current dependency
- Multi-step generation
    - Makefile can contain any number of rules
    - advantage: update only when needed



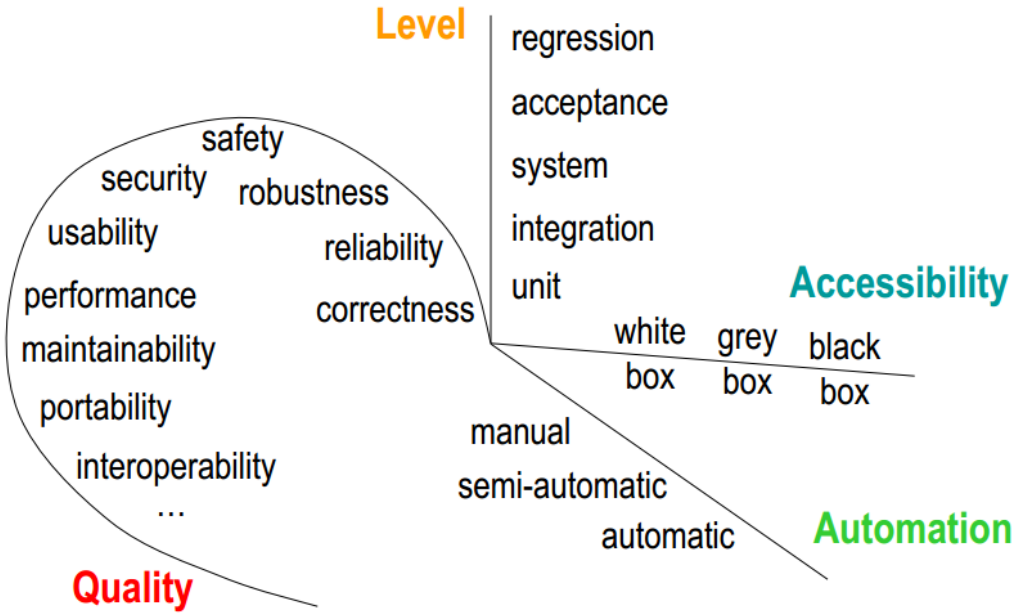
Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Software Testing

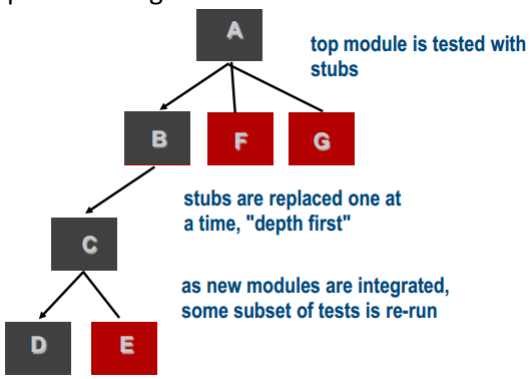
- := process of exercising a program with the specific intent of finding errors prior to delivery to the end user
- Testers
    - developer: understands the system but tests gently driven by **delivery**
    - indep tester: must learn about the system but will attempt to break it and is driven by **quality**

- Test feature space

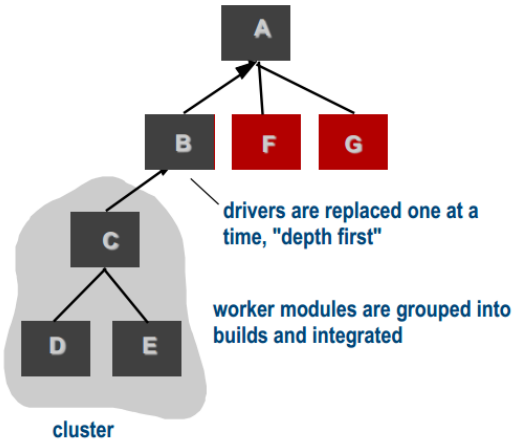


- Unit testing
  - Test unit = code that tests target
    - usually one or more test module/class
  - Test case = test of an assertion (design promise) or a particular feature
- Unit test environment
  - test driver = dummy env for test class
  - Test stub = dummy methods of classes used, but not available
  - e.g. cppunit, Junit, Cactus, JSpec
- Equivalent class testing
  - good test = likely to produce error
  - idea:
    build equivalence classes of test input situations,
    test one candidate per class
  - Terminology: Cx
    - C0 = every instruction
    - C1 = every branch
    - C2, C3 = every condition once true, once false
    - C4 = path coverage
    - Rule of thumb: 95% C0, 70% C1;
      C2, C3 often add no value, C4 often impossible
- Integration testing
  - = test interaction among units
  - sample integration problems:
    - F1 calls F2(char[] s) //array size difference?
    - F1 calls F2(elapsed_time) //time unit difference?
  - strategies:
    - Big Bang, incremental

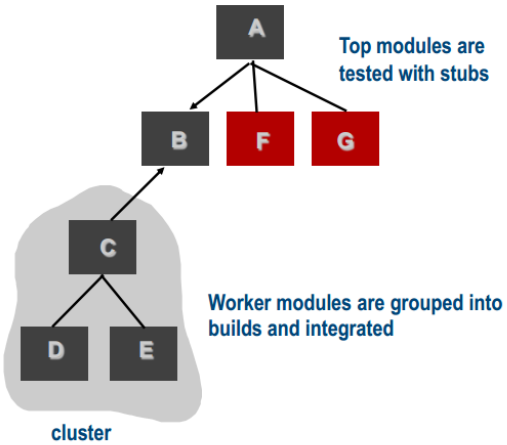
- Top down integration



- Bottom up integration

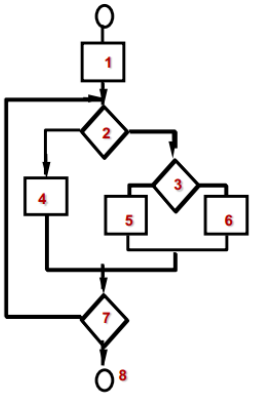


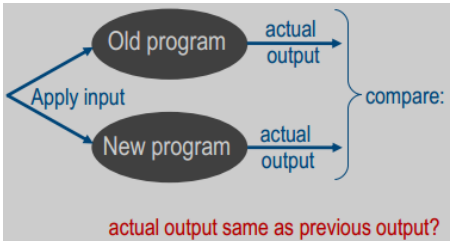
- Sandwich testing



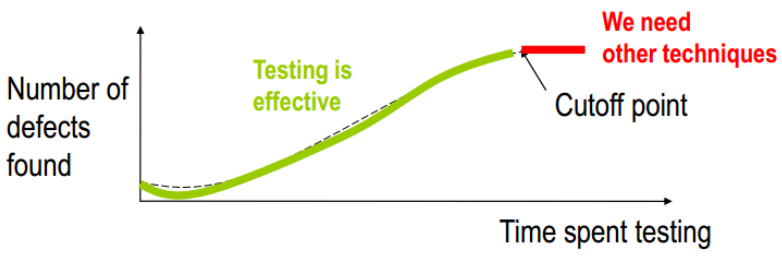
- System testing
  - = determine whether system meets requirements
  - Focus on use and interaction of system functionalities rather than details of impl
  - should be carried out by a group indep of the code developers
    - alpha testing: end users are dev's site
    - beta testing: at end user site, w/o developer
- Acceptance testing
  - Goal: get approval from customer → structure it the process
  - be sure that the demo works!
  - Customers may want more;
    - ideally: get test cases agreed already during analysis phase

        - above not practical, but
        - at least agree on schedule and criteria beforehand
        - prepare well in advance with stakeholders
- Testing methods
    - Static testing
        - collects info about SW without testing it
        - e.g. reviews, walkthroughs, inspections, documentation,
        - Control flow analysis and data flow analysis
            - provide objective data
            - extensive use for compiler optimization and software engineering
        - Formal verification: given a model of a program and a property, det whether the model satisfies prop based on math (algebra, logic)
    - Dynamic testing
        - collects info by executing SW
        - software behaviour in dev and target environments
        - whitebox vs blackbox testing, coverage analysis, memory leaks, performance profiling
            - Blackbox = spec-based testing; no knowledge about code internals, relying only on interface spec
                - limitations: specs usually not available, many companies only have code & no other doc
            - whitebox = check that all statements and conditions have been executed at least once
                - look inside modules and classes
                - limitation: cannot catch omission errors
                  cannot provide test oracles (input → corr output?)
        - Coverage analysis = measuring how much of the code has been exercised
            - identify unexecuted code str,
            - remove dead or unwanted code
            - Metrics:
                - entry points
                - statements
                - conditions (loops)
        - Path testing
            - cyclomatic complexity of flow graph
            - V(G) = number of simple decisions + 1 OR
              V(G) = number of enclosed areas + 1
            - Derive independent paths, then test cases to exercise these paths

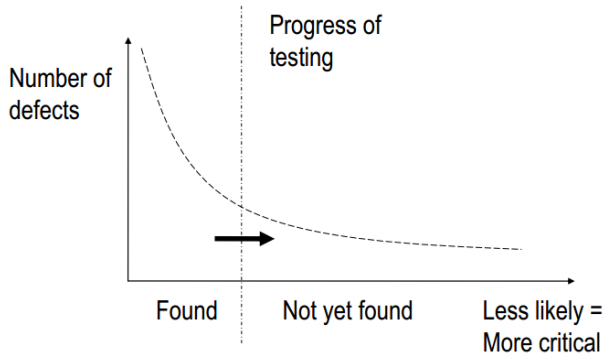


        - Performance profiler
            - Code profiling = benchmarking execution to understand where time is being spent
            - Questions to be answered:
              1. Code specifics (bulkiest part, loop properties, locations)
              2. approach (coding a block of logic → most efficient path?)
            - Clever and good use of logging
    - Regression testing
        - = Run tests, compare output to same test on prev code version
        - limitations:
            - finds only deviations, cannot judge on error
            - can only find newly introduced deviations
            - only reasonable for fully automated tests
        - 

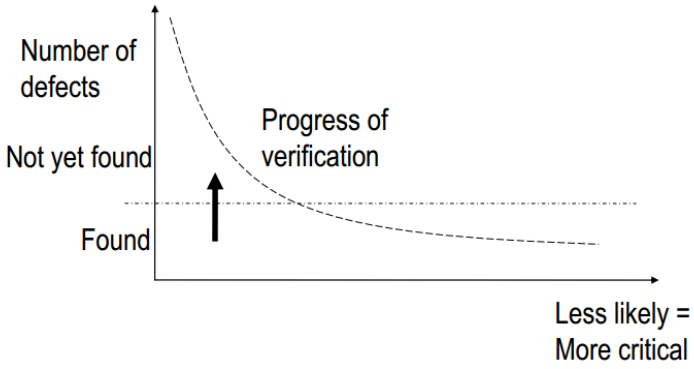
- Test organization
    - Tests should be self-sustaining
    - set up controlled environment
- Create testable software:
    - Simplicity
    - Decomposability: independent module testing
    - Controllability: tests can be automated and reproduced, states and variables controllable
    - Observability: make status queryable && logging
    - Stability
    - Operability
    - Understandability: DOCUMENTATION
- Symptoms and causes
    - Sym and causes may be geographically separated
    - symptom may be intermittent
    - cause may be due to a combination of **non-errors**
    - also may be due to system or compiler error
    - also may be due to assumptions
- Economiics of testing
    - The characteristic S-curve for error removal



- Testing tends to intercept errors in order of their probability of occurrence



- Verification is insensitive to the probability of occurrence of errors



The Maintenance Phase

- Bugs = inexplicable defect
- Software updates = new release to cure defects, respond to requests for enhancement, typically both
- Update is either incremental or full
- Bug anatomy
    - Component = module where the bug occurred
    - Status = short notes
    - Keywords = test cases, help requested
    - Target milestone = estimated fix date or release version
    - Dependency:
        - bugs that this one depends on
        - bugs that depend on this one
        - Bugzilla can display dependency graph
    - Attachment
        - test cases, screenshots, editor logs, patch file
- Bug lifecycle
    - Start
        - bug log entry created, ticket submitted with NEW status
        - until assigned, regular reminder
    - Resolution
        - Fixed bugs are marked RESOLVED and receive a resolution tag:
            - FIXED | INVALID | WONTFIX | DUPLICATE |WORKSFORME
            - ……. | not a bug| “feature”| already known | Cannot reproduce

    - Check
        - QA engineer checks resolved bugs, verifies the tags and marks it VERIFIED or REOPENED
        - then CLOSED When accepted by customer
- Bugzilla
    - bug tracking system
- Trac
- Bug reports
    - should be reproducible
    - and specific

Documentation

- Internal doc (inline comments)
    - comments should be meaningful
    - add a reference/link if that's better than writing an essay
    - When to comment? when you cannot understand what a code does after staring at it for a minute
    - What to always comment?
        - Every File
        - Every function (input, output)
        - Every nonobvious variable
        - Every struct/typedef
        - Tricks and weird optimizations
- External programmer doc
    - Tells other programmers what your code does
    - aim is to have other programmer read and understand your code without having to read the whole fucking thing
    - Large companies have standards
    - Global structure:
        - Stage 1. Overview and purpose
            - Details about behaviour, hows, files, input, algorithm
        - Stage 2. the mechanics
            - General flow of the program, explain complex algorithm
        - Stage 3. the gory details: globals
            - explain multi-file programming
            - explain any struct that is used a lot
            - global variables
        - Stage 4. The gory details: locals
            - describe every major function (input, output)
            - which functions are doing the real work
    - User doc
        - Sometimes even written before the code
    - Doxygen, Javadoc, …

Web and other applications

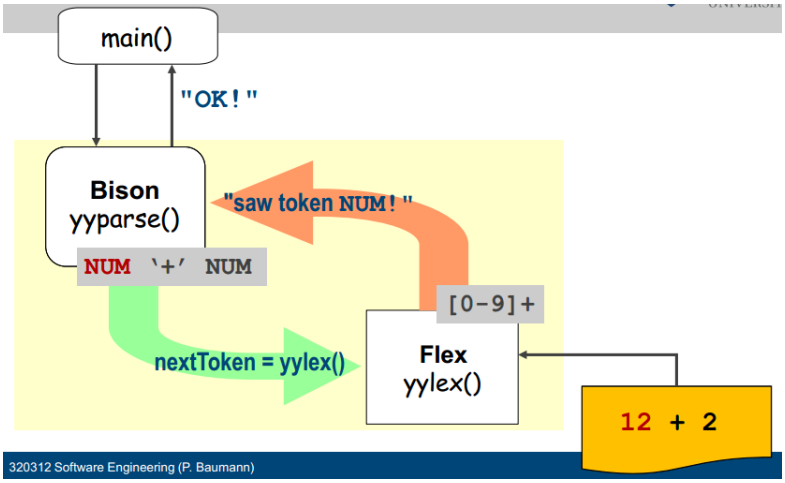
Application Architectures

- Applicatiion systems are designed to meet an organizational need
- Businesses have much in common → app systems also tend to have a common architecture
- A generic architecture is configured and adapted to create a system that meets specific req
- Use of app arch:
    - starting point
    - design checklist
    - organizing tool
    - assessing components
    - vocabulary for talking about app types
- App types
    - Data processing
        - process data in batches without explicit user intervention
        - e.g. billing systems, payroll
    - Transaction processing
        - process user requests and update information
        - e.g. E-commerce, reservation
    - event processing
        - interpret events
        - e.g. word processors, real-time systems
    - language processing
        - processing requests specified in a formal language
        - e.g. compilers, command interpreters

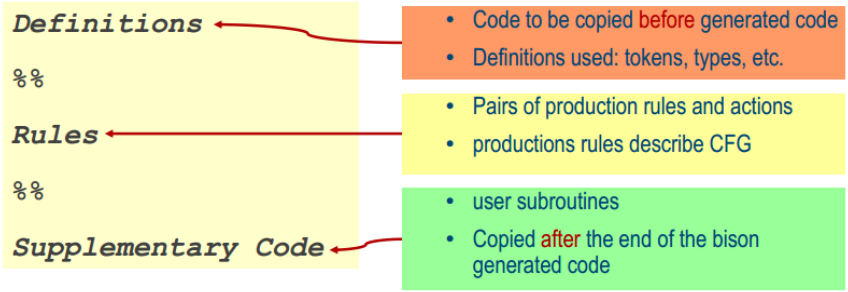
Language Processing

- LPSystems
    - Accept a natural or artificial language as input and generate some other rep of that language
    - e.g. compiler: generate machine code; interpreter: act immediaately on instructions while being processed;
    - Used when easiest way to solve a problem is to describe algo or data
- Flex & Bison
    - Bison: a parser generator := tool producing a parser for a given grammar
        - input = myparser.y[pp] containing grammar + actions
        - output = C++ program myparser.c[pp] + optional header
    - Flex: Lex = a scanner generator
        - input = scanner.l containing patterns and actions
        - output: C program scanner.c
        - typically, the generated scanner produces tokens for the parser
    - Synopsis
        - Bison grammar defines admissible sequences (“sentences”)

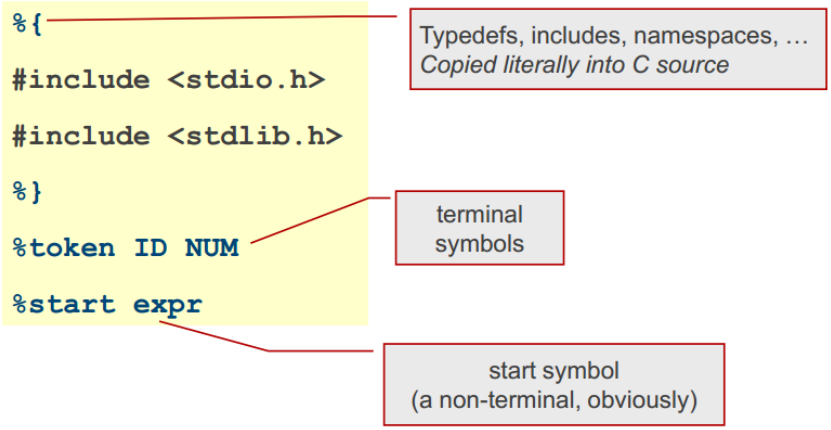
- Flex grammar defines single tokens (“words”)



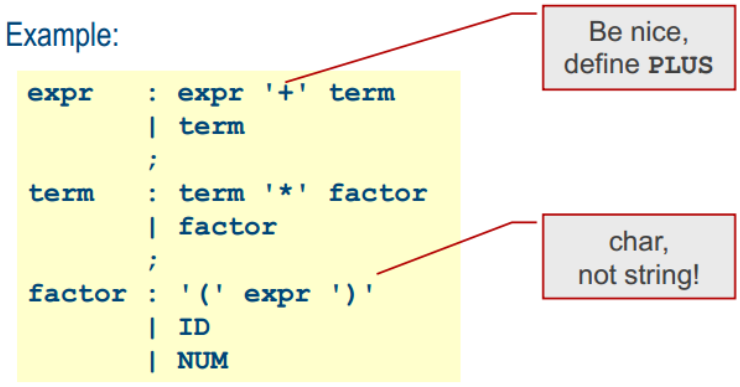
- Bison file format



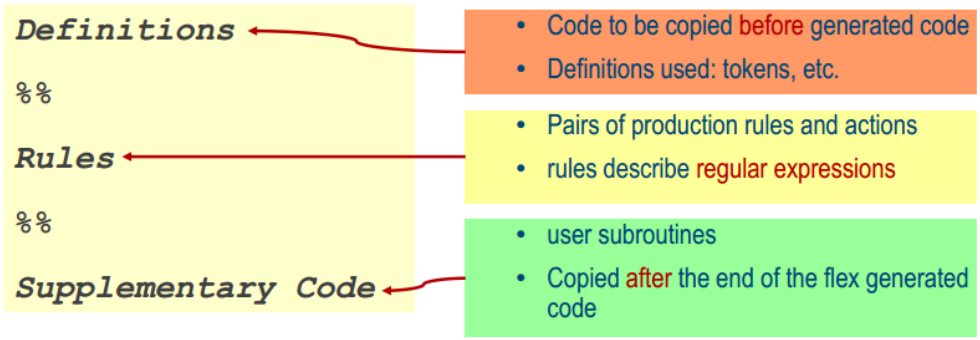
- Bison definitions section



- Bison rules section
  - Contains grammar: referring to previously defined non-terminals and terminals



- Bison code section
  - all the other code, e.g. main(), yyerror(),…
- FLEX



  -
  - example:

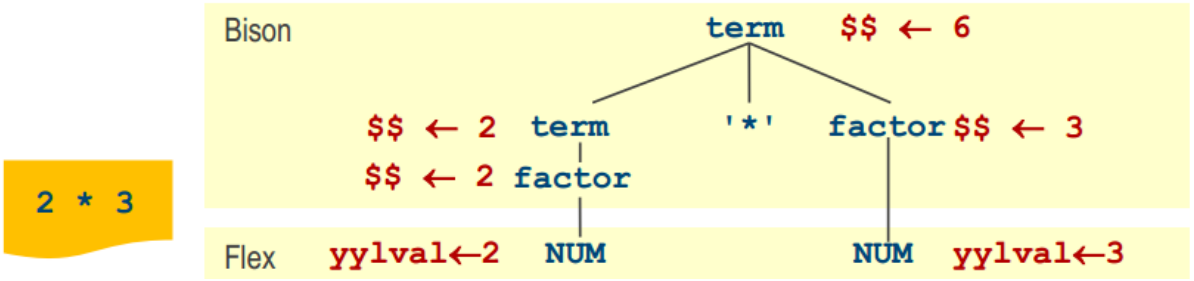
```
%{
#include <stdio.h>
#include "parser.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
num         [+-]?[0-9]+
semi        [;]
wspc        [ \t\n]+
%%
{id}        { return ID; }
{num}       { return NUM; }
{semi}      { return SEMI; }
{wspc}      {;}
```

Defined in bison's `parser.h`

Returns only token tag – actual value passed elsewhere

- Compile and linking flex/bison
  - $ flex scanner.l
    $ bison -d myparser.ypp
    $ gcc -o parser myparser.cpp lex.yy.c -ly -lfl
- Semantic actions in Bison
  - action = code exec when rule is applied
  - Attribute values store intermediate results
    - $1, $2 = results from evaluating (non-) terminal #1, #2
    - $$ = result of current expression
- Dynamics of rule processing
  - Rule fires = right hand side reduced to left-hand non-terminal
    - Rules reduced bottom-up
    - successful if, at EOF, only axiom remains (empty stack)

```
term   : term '*' factor   { $$ = $1 * $3; }
       | factor            { $$ = $1; }
factor: NUM                { $$ = yylval; }
```




```
expr:    expr '+' term     { $$ = $1 + $3; }
       | term              { $$ = $1; }
       ;

term:    term '*' factor   { $$ = $1 * $3; }
       | factor            { $$ = $1; }
       ;

factor:  '(' expr ')'      { $$ = $2; }
       | ID                { $$ = lookup(symbolTable,yylval); }
       | NUM               { $$ = yylval; }
       ;
```

- Error handling
  - Catch & recover
    - elastic recovery from errors, graceful continuation
    - Example, good for line-oriented input

```
line :  /* empty */
     | line whatever
     | line error                    /* std error token */
       {
         yyerror( "Failure :-(" ); /* msg output etc. */
         yyerrok;                  /* reset parser */
         yyclearin;                /* reset scanner */
       }
```

    - Syntactic error: caught by parser, need to manually cure & reset
    - Semantic error: caught by your action, ignored by parser (e.g. divide by 0)
  - Report
    - providing meaningful diagnostic output
  - Useful information: line number, col number, violating token, expectation vs given
- Parser debugging
  - $ gcc -DYYDEBUG …
  - or in the program code:
    extern int yydebug;
    yydebug = 1;
  - → very verbose log of bison rule processing

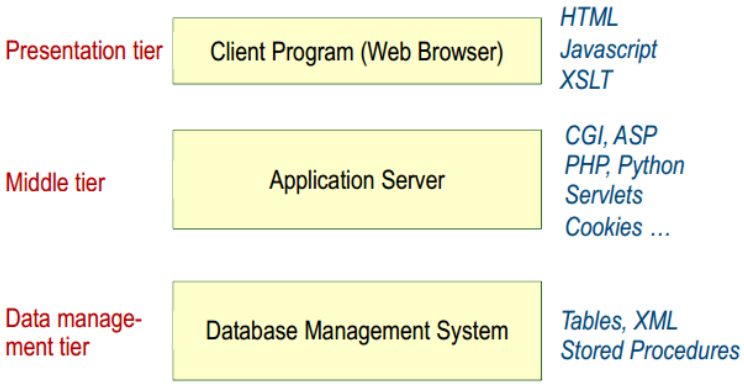
- Makefile sample

```
LEX = flex
YACC = bison
CC = gcc

calc: parser.o scanner.o
   $(CC) -o calc parser.o scanner.o -ly -lfl

scanner.o: parser.h scanner.c

scanner.c: scanner.l parser.h
   $(LEX) -oscanner.c scanner.l

parser.o: parser.cpp parser.h

parser.cpp parser.h: parser.ypp
   $(YACC) -d parser.ypp
```

- ANTLR: Another tool for language recognition
  - Java based

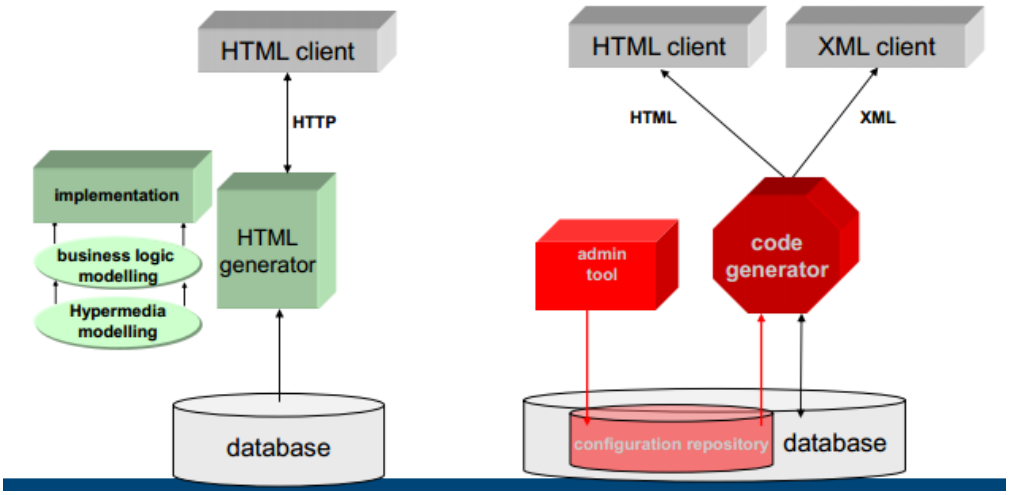
Web-enabled info systems

- Components of data-intensive systems
  - Presentation
    - primary interface to the user
    - needs to adapt to different display devices
  - Application (business) logic
    - implements business logic (complex actions, maintains state between diff steps of a workflow, etc.)
    - Accesses different data management systems
  - Data management
    - one or more std database management systems
- Three-tier architecture



- Content management systems
  - Historically: Document management Systems
  - Then CMS, now Web CMS
    - Browser as GUI, high emphasis on appearance, availability, security
  - Component/Enterprise CMS
    - use components to build up documents
    - the latter is focused on whole business mgmt., not just web authoring

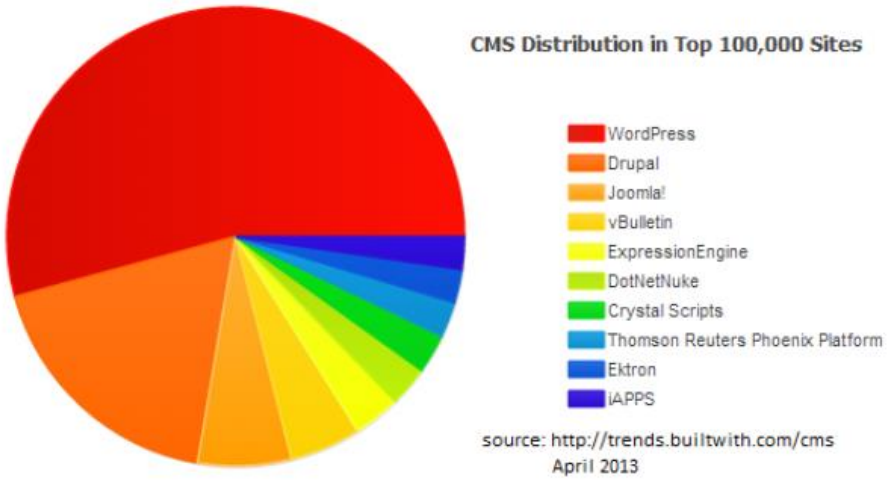
- Schematic WCMS Architecture
  - Variant 1:
    hard-coded business logic
  - Variant 2:
    configured business logic



- Std web retrieval cycle: Query → results list → details
- Query mechanisms
  - One or more form entries,
  - Qualitative/fuzy search,
  - advanced search criteria + joins
  - Drop-down lists gen. from database contents
  - Tree-based & other reps using JavaScript
- Modelling tool
  - Modular design
  - User-defined layout elements down to HTML/XML tags
- Benefits and disadvantages of WCMS
  - Low cost (many free systems)
  - rapid prototyping and deployment for small installations
  - Plugin and addon extensibility
  - Personalisation, role-based access control, workflow control
  - Web-based admin and maintenance
  - Flexible target formats from one source
  - Total costs of ownership (TCO)
    - may require training + dedicated maintenance + licensing cost
    - if web provided as service, forces to its continuous update
  - Performance
    - underoptimization, inadequate hardware capacity
- Webapp attributes
  - Network intensity
    - Served and accessed over the network
    - diverse community of clients
  - Concurrency
    - large number of users may access at the same time
    - patterns of usage vary greatly
  - Unpredictable load
    - no. of users may vary by orders of magnitude from day to day
  - Performance
    - if user must wait too long, they may go elsewhere

    - Availability
        - 100% availability is unreasonable, but often demanded
    - Data driven
        - primary function of many web aapps;
        - DMS, CMS, WCMS
    - Content sensitive
    - Aesthetics
    - Immediacy
        - time to market a few days or weeks
        - sophiscated web pages within few hours using modern tools
    - Continuous evolution
        - conventional software: evolves over planned, chronologically spaced releases
        - Web apps: evolve continuously
    - Security
        - Protect sensitive content + provide secure transmission
        - HTTPS, SSL, …
- Testing WebApps
    - Checking:
        - dangling links
        - Plugins
        - Performance
        - Proper reaction on error situations
    - Browsers
- Selenium
    - automates browsers, also enables content verification
    - Selenium IDE: firefox add-on for record and playback of interactions
        - bug reproduction scripts
        - scripts for automation-aided exploratory testing
    - Selenium WebDriiver: language specific bindings to drive a browser
        - browser-based regression automation
        - distribute scripts across many environments

WCMS distribution



User Interface Design

- UI
    - should be designed to match skills, experience and expectations of its anticipated users
    - System users often judge system by its interface rather than functionality
    - poor UI → more errors
- Human factors in UI design
    - limited short-term memory
        - instantly remember max 7 items
    - People make mistakes
        - scary sounds and warnings can make a person shut down
    - People are different
    - People have different interaction preferences
        - picture vs text vs …
- Pressman's Golden Rules
    - Place User in control
        - UI should not force user into doing unnecessary/undesired actions
        - Flexible interaction
        - Allow interaction to be interruptible and undoable
        - customizable interaction
        - Hide technical internals from the casual user
        - Design for direct interaction with objects on the screen
    - Reduce user's memory load
        - Reduce demand on short-term memory
        - meaningful defaults
        - intuitive shortcuts
        - Base visual layout on a real-world metaphor
        - Disclose info in a progressive fashion
    - Make interface consistent
        - Allow user to put current task into a meaningful context
        - maintain consistency across a family of apps
        - if past interactive models have created user expectations, do not change
            - unless it's necessary
    - Extra rule: Make interface safe (esp. for casual users)
- User analysis
    - If you don't understand what a user wants to do, you have no prospect of designing a good interface
    - Think of scenarios/use cases!
    - Requirements from Scenario:
        - users may not be aware of appropriate search terms
        - Users have to be able to select collections to search
        - Users need to be able to carry out searches and request copies of relevant material
- Analysis techniques
    - Task analysis: model steps involved in completing a task
    - Interviewing and questionnaires: what work users do, open-ended questions
    - Ethnography: observes user at work, questions them

- UI Prototyping
    - Aim: allow users to gain direct exp with interface
    - Without such direct exp, it's impossible to judge usability of an interface
    - Prototyping in 2 stage process:
        - early: paper prototypes
        - later: increasingly sophisticated computer prototypes
    - Techniques
        - Paper
        - Script-driven
            - set of scripts + screens
            - When user interacts, screen changes accordingly
        - Visual
            - Language designed for rapid development e.g. Visual basic
        - Internet-based
            - web browser + associated scripts
- UI Evaluation
    - some eval of a user interface design should be carried out to assess suitability
    - full scale eval very expensive & impractical for most systems
    - Ideally, an interface should be eval'd against a usability specification

Web design

- Design and webapp quality
    - Security
        - rebuff external attacks
        - exclude unauthorized access
        - ensure the privacy of users/customers
    - Availability
        - time that webapp is available
    - Scalability
        - significant variation in user/transaction volume
    - Time to market
- Webapp interface design
    - Where am I?
    - What can I do now?
        - available functions
        - live links
        - relevant content
    - Where have I been, where am I going? facilitating navigation
        - provide map: where user has been, what paths are available
        - easy to understand
- Interface design principles
    - Anticipation: anticipate user's next move
    - Communication: communicate status of any activity init'd by user
    - Consistency: use of nav controls, menus, icons, aesthetics
    - Controlled autonomy: facilitate user movement, but enforce nav conventions
    - Efficiency: optimize **user's** efficiency
- Aesthetic design
    - don't be afraid of white space

    - emphasize content
    - organize layout from top-left to bottom-right
    - Group nav, content, and function geographically within page
    - Don't extend your real estate with the scrolling bar
    - consider resolution and browser window size when designing layout
- Web design: home page variants
    - front door
    - information rich
    - mixed approaches
    - New style
- Content design
    - design rep for content objects
        - content object = data object for conventional software
    - mechanisms req to instantiate relationships
        - cf. relationship: analysis classes + design components
    - Content-specific & implementation-specific info
- Content architecture
    - Linear, grid, network, hierarchical structures
- Quality dimensions for end-users
    - Time
        - last change, highlighting changed parts
    - Structural
        - parts hold together
        - inside and outside links working?
        - images working?
        - parts not connected?
    - Content
        - critical pages: content matches expectation?
        - key phrases continually in highly-changeable pages?
        - critical pages maintain quality content from version to version?
    - Accuracy and consistency
        - are today's copies of the downloaded pages the same as yesterday's?
        - data is accurate enough?
    - Response time and latency
        - Does the web server respond to a browser request within certain params?
        - E-commerce context: end to end response time after SUBMIT?
        - parts too slow for users?
    - Performance
        - Browser-Web-site-browser connection quick enough?
        - variance by time of day, by load, by usage?
        - adequate for e-comm applications?
- Hypermedia design patterns
    - Architectural patterns
    - Component construction patterns
    - navigation patterns
    - presentation patterns
        - user interface control func,
        - relationship between interface action & affected content obj
        - effective content hierarchies

    - Behaviour/user interaction patterns
        - informs user of consequences of a specific action
        - expand content, based on usage context and user desires
        - describe link destination
        - status of on-going interaction
        - …
- Some pattern repositories in 35_web-design.pdf – slide 15

Project and Process Management

Software Process and Project Management

- Top few project failure factors: Lack of…
  - executive support
  - user involvement
  - experienced project manager
  - clear business objectives
  - minimized scope
- Time spent:
  - Design, doc: 10%
  - Implementation: 15%
  - Testing: 5%
  - Struggling to understand requirements: **30%**
  - Technical difficulties: **30%**
- Process applicability: Right-sizing management
  - Methodical processes: well-understood app domains, re-engineered systems
  - Managed process: Large systems, long-lifetime products
  - Informal: Prototypes, short-lifetime, 4GL business & small & medium-sized systems
- Project Management
  - := activities to ensure that result is delivered on time, schedule, requirement
  - Planning & Monitoring required constantly and regularly
  - Various types of planning:
    - Quality: describes the quality procedures and std to be used in a proj
    - Validation: describes the approach, resources and schedule used for system validation
    - Config management: describes the config mgmt. proc and structures
    - Maintenance: predicts the maintenance req of the system
    - Staff dev: how to develop the skills and experience of the project team members
  - Plan sets out the resources (who), work breakdown (what), schedule (when)
  - Structure: Intro, proj org., risk analysis, HW & SW resource req, work breakdown, proj schedule, monitoring & reporting mechanisms
  - Planning process

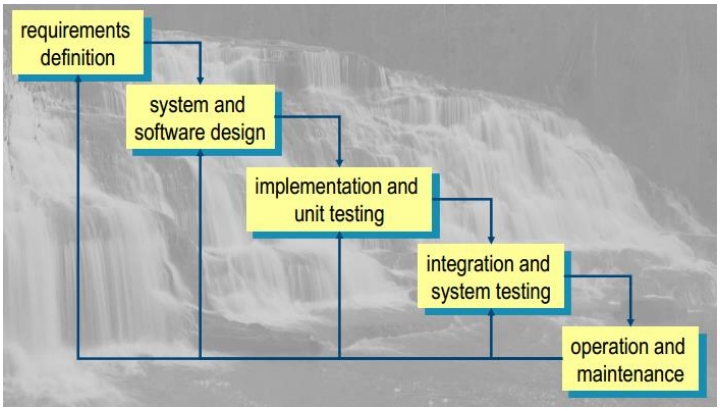
```
Establish project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
Draw up project schedule
while project has not been completed or cancelled
loop
        Initiate activities according to schedule
        Wait ( for a while )
        Review project progress
        Revise estimates of project parameters
        Update the project schedule
        Re-negotiate project constraints and deliverables
        if ( problems arise ) then
                Initiate technical review and possible revision
        end if
end loop
```

  - Activity organization

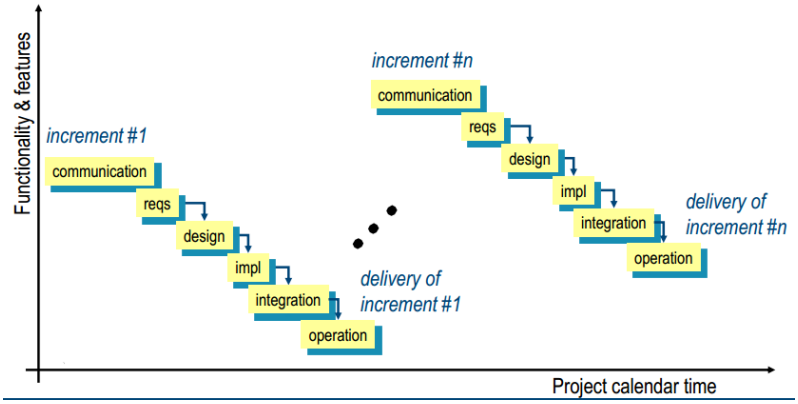
        - activities should be organized to produce tangible outputs at well-defined points
            - Tasks with subtasks
            - Milestones
            - Deliverables for customers or management
        - Good rules:
            - Design task as **self-contained** units with clear goal
            - Concurrent tasks → optimal use of workforce
            - Minimize dependences to minimize waiting and delay
        - Waterfall process
        - Type of activity dependency & duration planning
            - Tabular task D&D
            - Activity network
            - Gantt Chart
            - PERT chart
                - show relationships between activities
                - Can attach additional details (completion time, names, etc.)
        - Potential scheduling problems
            - estimating difficulty of problems
            - Productivity !~ #people
            - Adding people to a late project makes it later
            - Murphy's law
- Risk management
    - := identifying and planning to minimize risks
    - Types of risk
        - Project risks affect schedule or resources
        - Product risks affect quality or performance
        - Business risks affect the organization

Software Process Models

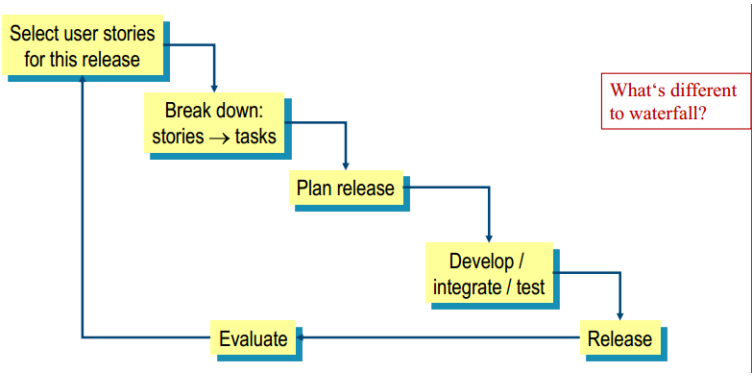
- Waterfall model



    - Partitioning into distinct stages
        - inflexible; difficult to accommodate change after process is underway
    - Only appropriate when requirements are well-understood and fairly stable
    - Mostly used for large systems engineering projects where system is developed at several sites

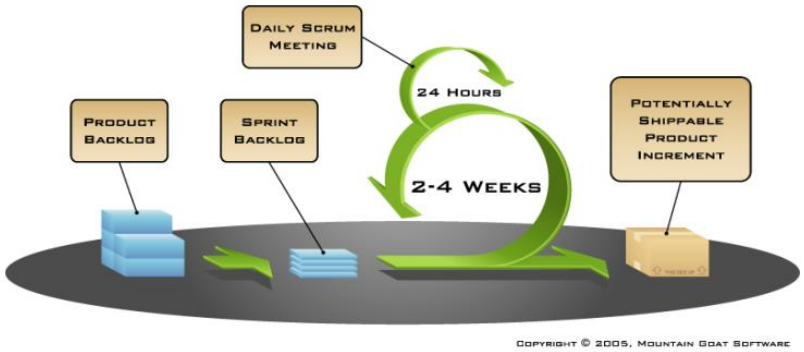
- Incremental model
  - 

  - Development & delivery broken down into increments
    - More flexible; can accommodate mid-project changes by adding it to the next increment
  - User requirements are prioritized, with highest priority req included in early incr.
    - System functionality is available earlier
    - Early increments act as a prototype → help elicit req for later increments
  - Lower risk of overall project failure
  - Highest priority system services tend to receive most testing
    - Implemented early → present for subsequent increments' testing
- Agile method
  - Principles: CIPCS
    - **Customer** involvement
    - **Incremental** delivery
    - **People**, not process
    - Embrace **change**
    - Maintain **simplicity**
  - Extreme programming
    - based on very small increments
      - New versions built several times a day
      - Increments delivered ~every 2 weeks
      - All tests must be run for every build and must pass, else rejected
    - Relies on
      - constant code improvement
      - user involvement in the dev team
      - pairwise programming
  - Pair programming
    - Programmers work in pairs, sitting together to develop code
    - Informal review process (2 people looking at code)
    - Encourages refactoring
    - Similar productivity compared to 2 people working independently
      - but possible other benefits? Social + satisfaction
  - XP and change
    - proposes constant code improvement
    - XP Release cycle



- Consequences of extreme programming
  - Incremental planning
  - small releases → minimal useful set of functionality that provides businesss value is developed first
  - Collective ownership through pairwise development
    - anyone can change anything
    - all developers own all code
  - Simple design → No extra features other than current requirements
  - Simple code through refactoring
  - Sustainable pace
  - On-site customer
    - end-user rep available full-time
    - Customer member of dev team responsible for brining system req to the dev team
- Appraisal
  - Maintaining simplicity requires extra work
  - Contracts may be a problem
  - Suitable for short-term, highly-flexible projects
    - large projects may not have the high-priority requirements fully figured out yet
  - Team members must be suited to the intensity of agile methods
  - Devs need to be experienced, not too different in expertise
  - Can be difficult to keep interest of customers involved in process

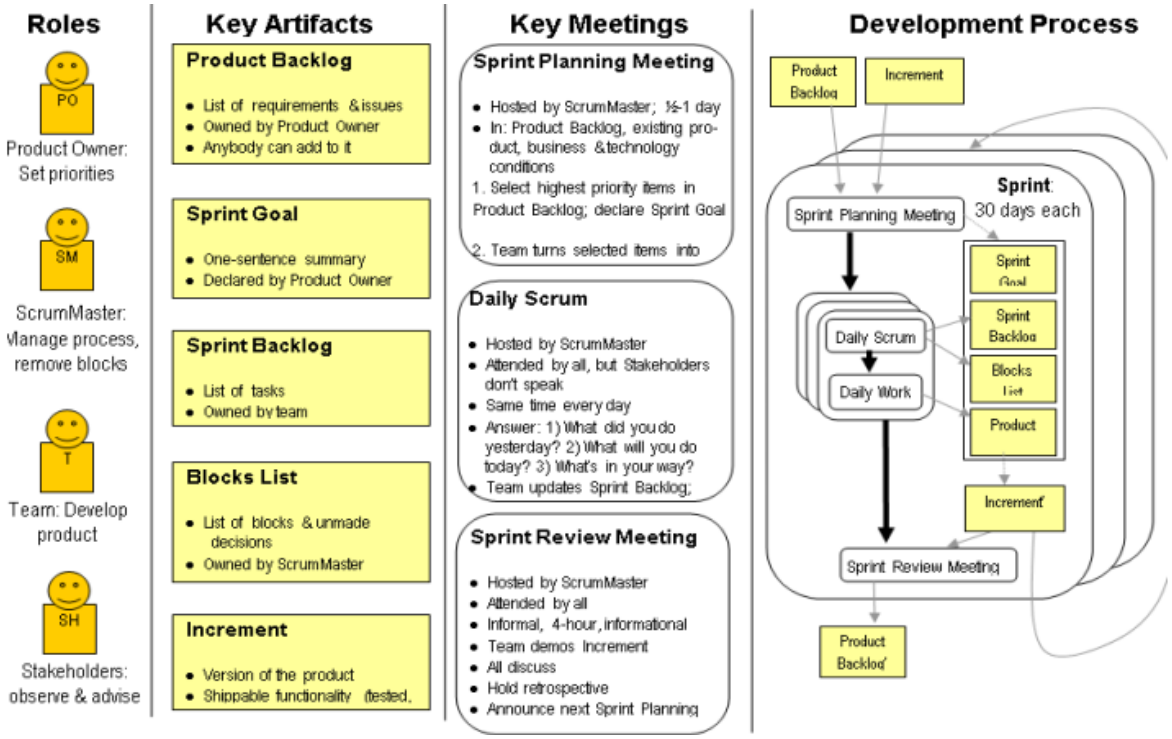
- Agile 2: Scrum
  - Overview



  - Components
    - Roles:
      - Ownerr
        - Knows what needs to be built and its sequence

        - e.g. project manager
    - S Team
        - Typically 5~6 people
        - Cross-functional
        - Full-time members
        - Self-organizing
        - Membership can change only between sprints
    - S Master
        - Represents management to the project
        - e.g. project manager/team leader
        - responsible for enacting scrum values & practices
        - Main job: remove **impediments**
- Process:
    - Project kickoff meeting
        - Collaborative meeting @ beginning of **project**
        - Participants: product owner, S master
        - Takes 8 hours, before lunch & after lunch
        - Goal : create product backlog
    - Sprint planning meeting
        - Collaborative meeting @ beginning of each **sprint**
        - Participants: product owner, S Master and team
        - 8 hours as well
        - Goal: create **sprint** backlog
    - Sprint
        - month-long iteration during which product functionality is incremented
        - No outside influence on scrum team during sprint
    - Daily scrum meeting
        - short (15 min) meeting held every day before working
        - participants: S master & team
        - Every team member must answer the 3 questions:
            - Status: what did I do?
            - Issues: what is stopping me?
            - Action: what do I do until next meeting?

- Overview

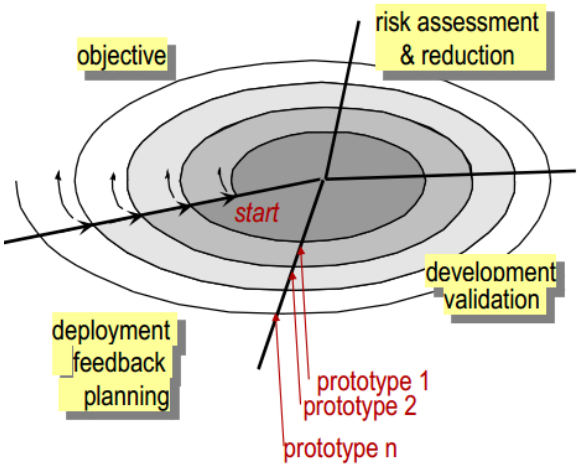


- Appraisal

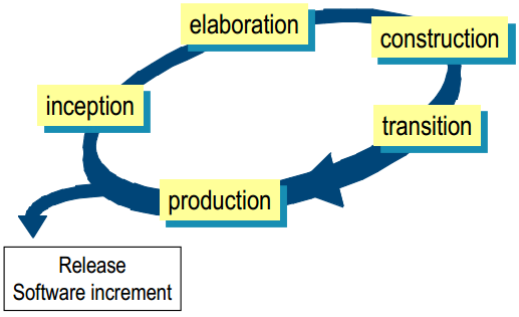
Advantages	Disadvantages
• Completely developed & tested features in short iterations • Simplcity • Clearly defined rules • increased productivity • Self-organizing • Team member carry responsibility • Improved communication • Combination with XP	• Undisciplined hacking • No formal documentation • Violation of responsibility

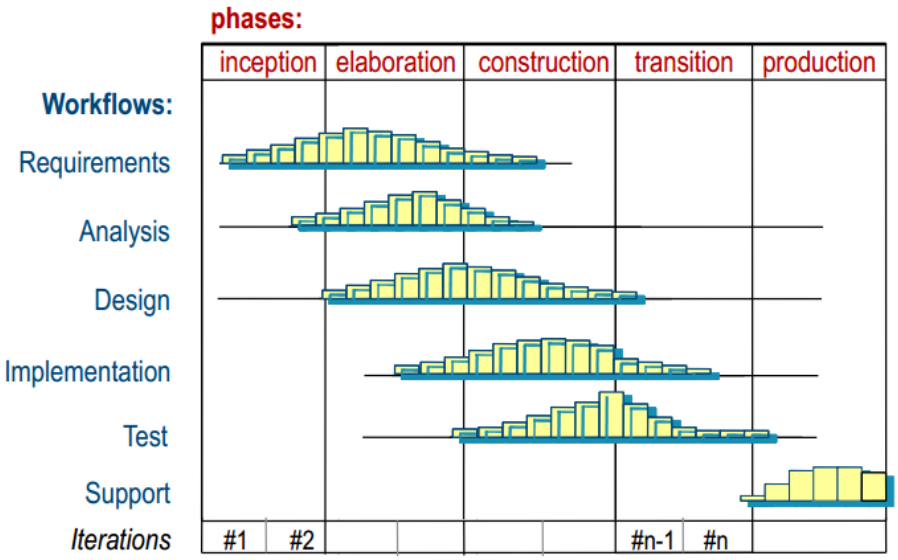
Iterative/Spiral and Evolutionary Methods

- Spiral model
  - Objective setting:
    Identify objectives for this phase
  - Risk assessment & Reduction
  - Development and validation:
    Choose any dev model
  - Planning
    Review and next phase planning



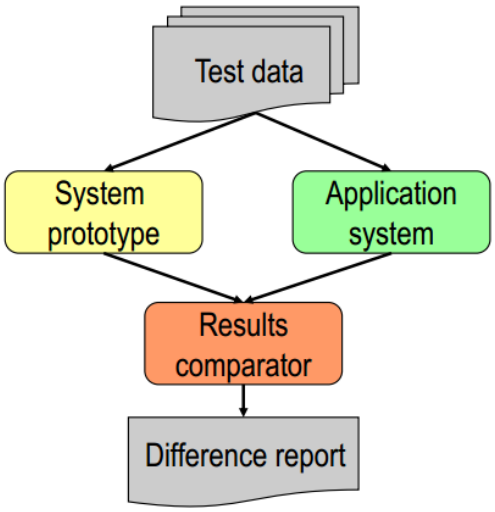
- Spiral model rules
  - Process is spiral, not sequential with backtracking
  - Loop in spiral = one phase in the process
  - No fixed phases
  - Risks explicitly assessed & resolved throughout the process
- Rational Unified Process (RUP) Model
  - Based on UML
    - use-case driven,
    - architecture-centric,
    - iterative and incremental
  - Normally described from 3 perspectives:
    - dynamic: phases over time
    - static: process activities
    - practice: suggests good practice
- UP Phases
  - Inception:
    Establish business case
  - Elaboration:
    Understanding of problem domain
    and system architecture
  - Construction:
    System design, prog, testing
  - Transition:
    Deploy sys in operative env
  - Production:
    Support and maintain





- Evolutionary development
  - exploratory development
    - work with customers
    - evolve final system from initial outline spec
    - start with well-understood requirements, add new features as proposed by customer → similar to incremental / iterative approach

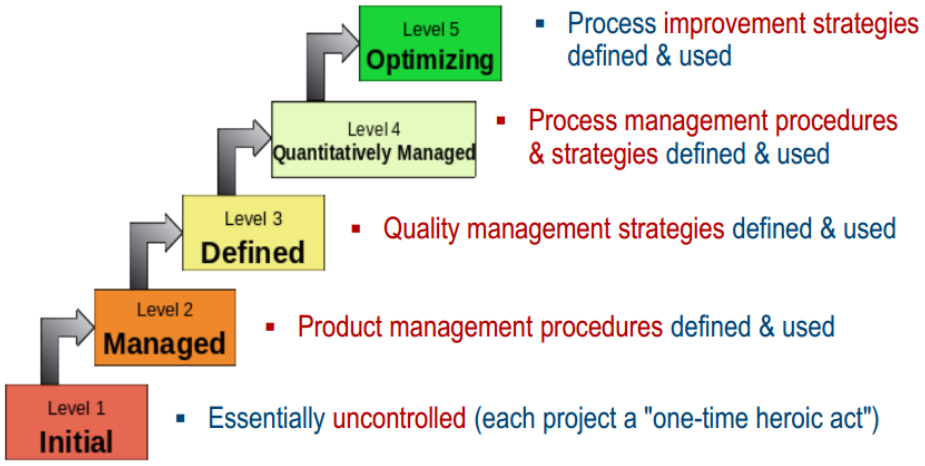
    - Throw-away prototyping
        - Goal: understand system req, **not** building a deliverable
        - Start with poor understanding of requirement
- Prototyping
    - For some large systems, incremental development & delivery may be impractical
    - Prototype = initial version of a system used to
        - demonstrate concepts
        - try out design options
    - Prototype can be used in:
        - requirements engineering process → helping with req elicitation/validation
        - design processes → explore options, develop UI design
        - testing process → run back-to-back tests
    - Back-to-back testing



        -
    - Throw-away prototypes
        - prototypes should be discarded after dev, since they are not a good basis to start developing the real thing
        - May be impossible to tune the system to meet non-functional requirements
        - prototypes normally undocumented
        - structure degradation through rapid change
        - prototype probably doesn't meet quality standards
- When to incremental, when to prototype?
    - Incremental: delivering working system
        - development starts with requirements best understood
    - Throwaway: validating or deriving sys req
        - development starts with req poorly understood
- Evolutionary Development Appraisal
    - Problems
        - Lack of process visibility
        - Systems are often poorly structured
        - Special skills may be required
    - Applicability
        - For small or medium-size interactive systems
        - For well-isolated parts of large systems
        - For short-lifetime systems

- Vorgehens-Modell
    - Development std for IT systems in Germany
        - German national standard, mandatory for gov-procured projects
    - Features
        - process model for planning and realizing dev projects
        - considers entire system lifecycle
        - responsibilities of each participant
        - German-level detail

Capability Maturity Model Integration

- Process Capability Assessment
    - To what extent do an organisation's process follow best practice?
        - identify areas of weakness for process improv
    - Various models; SEI most influential
        - Software Engineering Institute
        - SEI's mission: promote software tech transfer
    - CMM(I) framework measures process maturity, helping with improvement
- CMM Organisation Maturity Levels
    - 

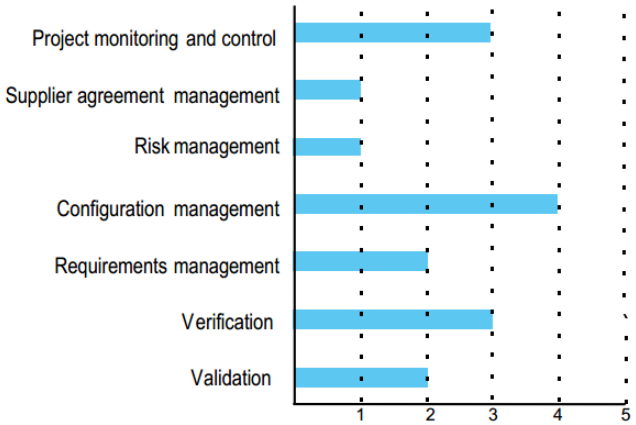
- Problems with CMM
    - Model levels
        - Companies could be using practices from diff levels at the same time, but if all practices from a lower level were not used, it was not possible to move beyond that level
    - Discrete rather than continuous
        - Did not recognize the distinctions between the top and the bottom of levels
    - Practices oriented
        - Concerned with how things were done rather than the goals to be achieved
- CMMI
    - Integrated capability model that includes software and systems engineering capability assessment
    - Components:
        - Process areas – 24 process areas that are relevant to process capability and improvement are identified; organized into 4 groups
        - Goals – descriptions of desirable organizational states; each process area has associated goals
        - Practices – practices are ways of achieving a goal; however, only advisory

- Process area lists:
  - Process management
    - Organisational process definition, process focus, training, process performance, innovation and deployment
  - Project management
    - Project planning, monitoring and control; supplier agreement management; integrated project management; risk management; integrated teaming; quantitative project management
  - Engineering
    - Requirements management, development; technical solution; product integration; verification; validation
  - Support
    - Config management; process and product quality management; measurement and analysis; decision analysis and resolution; org environment for integration; causal analysis and resolution
- Goals
  - Corrective actions are managed to closure when the project's performance or results deviate significantly from the plan
  - actual performance and progress of the proj is monitored against the proj plan
  - the requirements are analysed and validated and a definition of the req functionality is developed
  - root causes of defects and other problems are systematically determined
  - the process is institutionalized as a defined process
  - Process area goals
    - Specific goal in:
      - Project monitoring and control
      - requirements development
      - causal analysis and resolution
    - generic
- Practices
  - Analyse derived req to ensure that they are necessary and sufficient
  - Validate req to ensure that the resulting product will perform as intended in the user's env using multiple techniques as appropriate
  - select the defects and other problems for analysis
  - perform causal analysis of selected defects and other problems and propose actions to address them
  - establish and maintain an org policy for planning and performing the req development process
  - assign responsibility and authority for performing the process, developing the work products and providing the services of the requirements development process
  - Associated goal:
    - the reqs are analysed and validated; definition of the required functionality developed
    - Root causes of defects and other problems are systematically determined
    - the process is institutionalized as a defined process

- CMMI Assessment
  - Examines processes used in an org and assesses maturity in each process area
  - Merged into one final grade using a 6 point scale:

  Not performed; performed; managed; defined; quantitatively managed; optimizing

- The continuous CMMI model
  - First extension: staged CMMI model
    - Each maturity level has process areas and goals
  - Next extension: continuous CMMI model
    - finer grain: considers individual or groups of practices, assesses their use
    - maturity assessment not a single value, but one maturity value per area
    - each process area: levels 1 … 5
    - advantage: org can pick and choose process areas to improve acc. to their local needs
  - Sample process capability profile



-