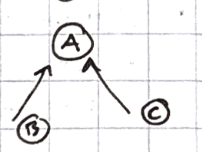
ECSE 437 - Crib Sheet

Albert Kragl

- git reset: undo add
- git revert: undo commit
- Each git repository can act as a server (provider of commits) or client
- git objects
  - <u>Commit</u>: a reference to a change being tracked by git
    - Refers to a tree object that describes the state of the repo for that commit
  - <u>Tree</u>: A directory being tracked
    - Contains a listing of blobs and tree objects (subdirectories)
  - <u>Blob</u>: a file being tracked by git
- refs
  - <u>heads/</u>: a listing of commit-type object DB keys that each branch currently points to
  - <u>remotes/</u>: a list of branches that are currently being tracked in upstream repositories
  - <u>tags</u>: A list of commit-type object DB keys that each tag corresponds to
- **HEAD:** a reference to the current branch that is checked out
- **config:** a file that stores config options for the current repo
- **pre-receive:** takes a list of references that are being pushed from stdin
  - e.g. make sure none of the updated refs are non-fast-forwards
- **update:** run once for each branch being pushed
  - name of ref (branch), the SHA-1 that ref pointed to before the push, and the SHA-1 that is being pushed
  - e.g. reject pushes to master
- **post-receive:** runs after the entire process is completed; takes same stdin input
  - e.g. notify a CI server or email a list

- Build tool families:
  - Low-level:
    - Explicitly define deps and rules for each input and output file
  - Abstraction-based:
    - Derive low-level build code from high-level data, e.g. maps of files to executables
  - Framework-driven
    - Default behavior assumed unless explicitly overridden
- Complementary tool families
  - Dependency management
  - Testing frameworks

A : B C



- Makefile variables
  - my_var = "hello"
    random.o : random.c
    <tab> $(my_var)
- **Incremental builds:** only out-of-date targets are rebuilt
- Low-level build issues
  - Shell scripts aren't super portable
- To handle deps across directories, recursive calls to Make are typically used
  - Fractures global dependency graph
  - One build may not rebuild all that is needed
- Ant is one low-level solution
  - Properties (variables)
  - Tasks (invocation of commands)
  - Targets (grouping of tasks)
- **Maven:**
  - A lifecycle is a sequential series of phases
  - A phase performs a series of goals that are bound via plugins
  - Default, Site, Clean
- A phase with no goals or plugins is skipped
- Dependencies can be scoped to specific phases
- Builds are computationally expensive
  - Dependency graph requires a lot of RAM
  - Compilation requires lots of CPU
  - File stating requires large and fast disks
- Builds can easily be parallelized, but there is a limit on developer machines
- Bazel creates an action graph (BUILD & WORKSPACE)

- Devs execute personal builds on their machines
- Only quick tests are run by default
  - Slower tests can be relegated to special build targets that run less often
- Micro-build: Concerns about the behavior of a build system within a single execution
- Macro-build: Concerns about how to best provision a fleet of resources
- Teams that use modern code review connect build jobs to the code reviewing dashboard
- Nightly builds are too infrequent
  - If hundreds of devs made commits, it's hard to tell who caused the problem
  - Imagine your code from yesterday broke the build — tough to recall what you were doing
- CI Feedback loop:
  - Commit, Build, Test, Report
- The benefits of CI
  - Each commit gets its own build job
  - Errors are reported quickly
- Build systems need to be updated as the codebase evolves
  - Otherwise build can produce incorrect results
  - Broken builds + weird bugs
- Flaky tests (non-deterministic)
  - False positive: test says code has failed but it should have passed
    - Devs lose trust in the test
  - False negative: test passes when it should have failed
    - Allows bugs to slip through
- The Modern CI Process
  - Build-triggering event
    - Triggered by push or manually by dev
  - Build job creation service
    - Build job creation node adds the job to a queue of pending jobs
  - Build job processing service
    - Build jobs in the pending queue are allocated to build job processing nodes
    - First download latest version of code then apply changes to it
    - Compile, execute automated tests, deploy
    - Add results to report queue
  - Build reporting service
    - Results communicated to dev team
- Advantages of inspections vs. dynamic QA
  - Cascading errors can obfuscate test results
  - Incomplete work can still be inspected
  - Can uncover inefficiencies and style issues
  - Knowledge sharing, collaborative problem solving
- Inspections complement tests
- Structured inspections (rigid, heavyweight)
  - Moderator ensures...
    - Artifact is ready for review
    - Inspection procedure is followed
  - Scribe/Recorder
  - Reviewer
  - Reader (leads team through the code)
  - Producer (author of artifact)
- Tasks in software inspection
  - Planning (find review team, time, place)
  - Group prep, individual prep
  - Checklists ensure quality is attained
- Tips for productive review

DO:	DON'T:
- Critique the artifact	- Attack the person
- Keep review chunks short and succinct	- Submit fixes for multiple issues at once
- Plan time for reviews	- Skip reviews
- Prioritize reviews of important issues	- FIFO queue reviews
- Keep it light	- Use sarcasm or exaggeration

- Why do people do code reviews?
  - Finding defects
  - Code improvement
  - Team awareness
  - Share code ownership
  - Alternative solutions
  - Knowledge transfer
  - Improve dev process
  - Avoid build breaks
- Modern review roles
  - Author
  - Reviewer
  - Integrator (makes final call)
  - Verifier (checks the functional correctness of the code only)
- Some Maven phases:
  - test
  - clean
  - compile
  - test-compile
  - deploy
  - install

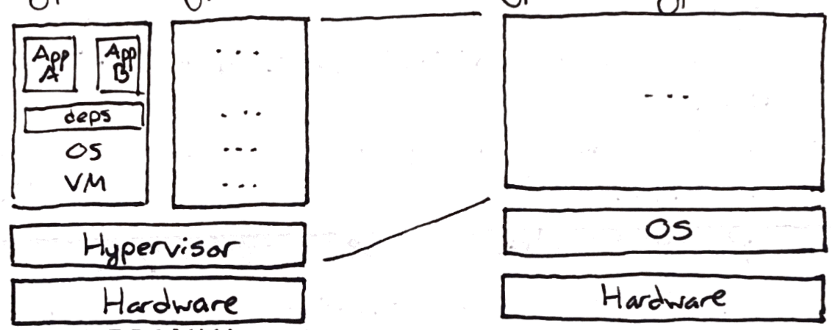
ECSE 321 - Crib Sheet 2 [pert]

- Ways to find defects
  - Testing
  - Static & Dynamic Analysis Tools
  - Statistical Defect Prediction
  - Manual Inspections
- Automatic defect finding
  - Goal: discover likely locations of defects
  - Static analysis: code smells, bad patterns, definite errors
  - Bug prediction: ML & Statistics
  - Dynamic analysis: taint analysis, performance, memory
- Bug localization (Hybrid Dynamic/Stats)
  - Use statistical properties to calculate suspiciousness of a line location
- Code Smells
  - Duplicated code
  - Long methods
  - God classes
  - Long parameter lists
  - Primitive obsession
- Static Analysis
  - Syntax Analysis
  - Semantic (data flow analysis)
- Data Flow Analysis
  - Framework for proving facts about a program
  - Examines how information propagates through blocks and paths of a program
- Reaching Definitions
  - A definition of a variable x is a statement that may modify its value
  - A use of a variable x is a statement that reads from x
- DU-Chain: links each def to uses it reaches
- UD-Chain: links each use to reaching defs
- Virtual Machines (VMs)
  - Emulate a computer system
  - Provide a computer system from within the scope of another system

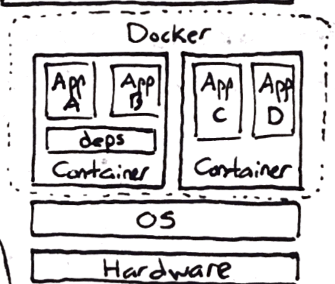
- What does virtualization provide?
  - Makes entire machines "shippable" (OS, system hardware, application stack
- The need for virtualization:
  - Gives shared infrastructure users (e.g. cloud users) the impression of having their own machine
- System virtualization:
  - "Full" virtualization
  - Hosts an entire OS
- Process Virtualization
  - Provides the isolation of system virtualization while also sharing common software tools with the underlying platform

- Blue-Green Schema changes
  - Transitionary status DB schema should work for both new & old layers
  - Final cleanup: delete old schema dat

- Hypervisor: a computer software, firmware or hardwa that creates and runs virtual machines (aka Virtual Machine Monitor
- Type 1 Hypervisor:
- Type 2 Hypervisor:

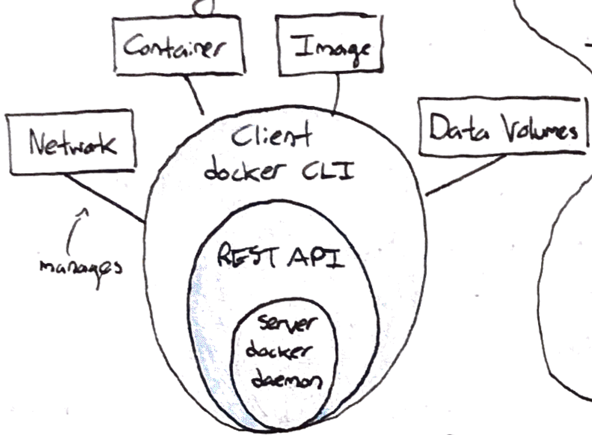


- Containers
  - A set of processes running on top of a common kernel
  - Isolated from each other



- Namespaces
  - Enable processes to have a private view of the system resources
  - Network interfaces, Process IDs, Mount points
- Docker
  - An orchestration tool for containers. Features:
  - Portability, App-centric, Builds from source, versioning, Component reuse, public registry, tool ecosystem
- Before containers:
  - Very inefficient use of memory and CPU resources
- After containers:
  - Isolated services in fewer VMs
  - Uses VMs more efficiently
- Docker Engine has:
  - A server (daemon)
  - A REST API which specifies interfaces that programs can use to talk to the daemon
  - A CLI client (the "docker" command)

- Docker Engine Visualized:



- <u>Docker daemon:</u> listens for Docker API requests and manages images, containers, networks and volumes
- <u>Docker Registry:</u> stores Docker images (i.e. Docker Hub)
- The Cloud:
  - Shared pools of configurable computing resources
  - Can be rapidly provisioned (semi)-automatically
  - Can "elastically" scale up or down based on demand
  - Only pay for what you use
- Downsides of cloud computing
  - Security, privacy, and other concerns about data and IP shared in the cloud are real
- IaaS
  - Provides computing resources to users in the form of VMs or containers
  - Compute hours/month or per-machine sizing (small, medium, large)
- IaaS vs. In-house Infrastructure
  - \+ Explainable infra operation costs
  - \+ IT support offloaded to cloud provider
  - \+ Elastic scaling
  - − Data security & privacy may be diff. to verify
  - − Costs can grow in unexpected ways
- PaaS
  - Service that provides a development platform
  - Provides users with comfortable app development entry point
- PaaS vs. IaaS
  - \+ Devs can focus on app development instead of config
  - \+ Platforms are often tuned to near-optimal settings
  - − Convenience at the cost of configurability
    - Non-standard workloads may suffer
  - − Hard to port applications to other platforms
- SaaS
  - The entire stack (infra, platform, execution env) is hosted on the cloud and delivered over the web
  - Usually involves a "thin client" that connects to a backend that delivers most of the functionality
  - e.g. Google suite
- Serverless
  - Variant of IaaS where capacity planning decisions are made by the provider (instead of consumer)
  - Customers are charged for the resources that their apps use
  - Elastic scalability is at app level instead of VM level
- Serverless vs. IaaS
  - \+ Users pay only for what they use
  - − Costs can be tough to predict
- FaaS
  - Primary means by which serverless computing is achieved
  - Functions are deployed to be executed on-demand in the cloud
- MBaaS
  - Share common backend features as APIs
  - e.g. Authorization, notif routing, social media integration, cloud-like storage
  - e.g. Firebase
- Infrastructure Config
  - Difficult to do manually
  - REs spend lots of time firefighting with config details
  - Wastes precious person-hours
- IaC
  - Write configurations as code
  - Manage and provision machines using a code-like syntax rather than interactive config tools
    - Has all advantages of code
    - Versioned (VCS)
    - Can be automatically executed
- Declarative Programming
  - Expresses the logic of a computation w/o describing its control flow
  - Describes <u>what</u> the program needs to do, and not how

- Partial time-limited deployment
- Evaluate safety of changes

Canary: →
- Roll forward
- Roll backward
- Alert certain parties

- Docker commands
  - docker build
  - docker run
  - docker start / stop
- Dockerfile commands
  - FROM, COPY, RUN, CMD, WORKDIR
- Puppet
  - <u>Resource:</u> lets you inspect & manipulate resources on a system
- <u>ldd</u>: prints shared libraries that an executable depends on

```
class pasture {
  package { 'pasture':
    ensure => present,
    provider => 'gem',
  }
  file { '/path':
    source => 'puppet:///...',
  }
  service { 'pasture':
    ensure => running,
  }
}
```

$$p = \frac{\binom{a+b}{a}\binom{c+d}{c}}{\binom{N}{a+c}}$$

	a	b
A	a	b
B	c	d