复习笔记

- Blackbox & Whitebox Testing
  - White-box testing
    - Statement coverage
    - 忽视部分分支
    - Branch coverage
    - 忽视运行时错误
    - Condition coverage
    - 忽视else分支
    - B & C coverage
  - Black-box testing
    - Test Data Selection
    - Exhaustive Testing
    - Random Testing
    - Partition Testing
    - Equivalence Partition Hypothesis
    - Boundary testing
- Test Doubles
  - Fakes- natural, not real
  - Stubs- unnatural
  - Mocks- verification added
- Maintainable Unit Tests
  - Unchanging Tests
  - Test Via Public Apis
  - Test Behaviors, Not Methods

LECTURE 12-cicd

- CI/CD
  - Continuous Integration
    - CB
    - continuous testing
    - CD
- Deployment Strategy
  - Blue-Green Deployment (蓝绿部署)

LECTURE 1–overview

SE Overview

Software Crisis

• Software crisis is a term used in the early days of computing science for the difficulty of writing useful and efficient computer programs in the required time

• The causes of the software crisis were linked to the overall complexity of hardware and the software development process.

The crisis manifested itself in several ways：

- Projects running over–budget
- Projects running over–time
- Software was very inefficient
- Software was of low quality
- Software often did not meet requirements

- Projects were unmanageable and code difficultto maintain
- Software was never delivered

What is software engineering?

• Software: a program or set of programs containing instructions that provide desired functionality.

• Engineering: the process of designing and building something that serves a particular purpose and finds a cost–effective solution to problems.

Software engineering: the branch of engineering that deals with the design, development,testing, and maintenance of software applications, while ensuring that the software to be built is:

- Correct
- Consistent
- On budget
- On time
- Within the required requirements

Software Engineering Vs. Programming

Programming is a significant part of software engineering

> Programming is how you generate a new software in the first place

Software engineering is programming integrated over time, scale, and trade–offs

LIFE SPAN, RESOURCES, COMPLEXITY

programing：ddl前完成，对就好

se：长周期，要不断完善，考虑很多比如质量、开销、社会影响

Time & Change

Hyrum' s law

Hyrum's Law is named after Google softwareengineer Hyrum Wright and states that eventhough you may design your API forextensibility and count on being able to evolveit, things can become more complicated.

In particular, when you have many consumers.they may depend on things they shouldn'tdepend on.

Scale & Efficiency

A dedicated group of experts in theinfrastructure team

Either do the update in place, in backwardcompatible fashion.

Or do the work to migrate their users tonew versions

Trade-offs & Costs

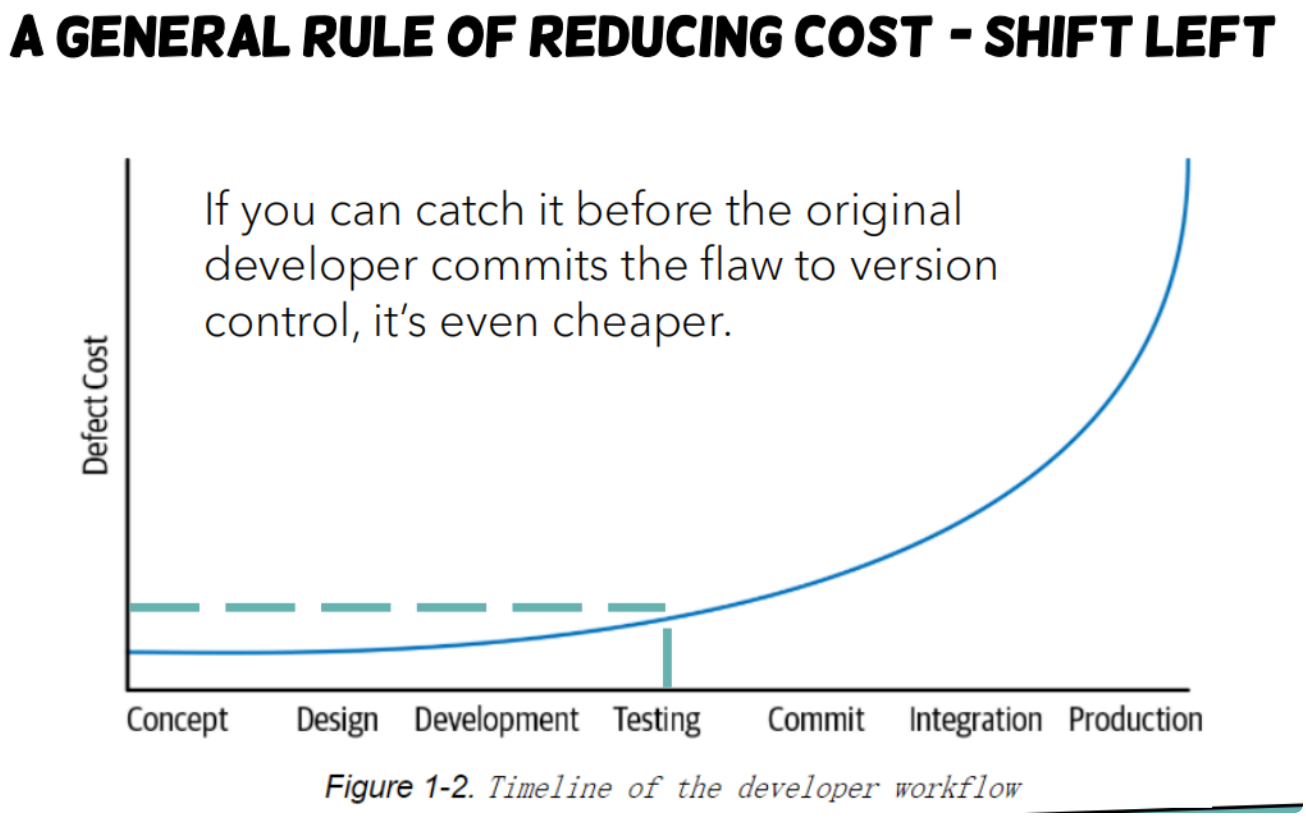


Figure 1-2. Timeline of the developer workflow

LECTURE 2-process

Software Process Overview

Why software process?

• Writing code is easy

• Engineering good software system is HARD

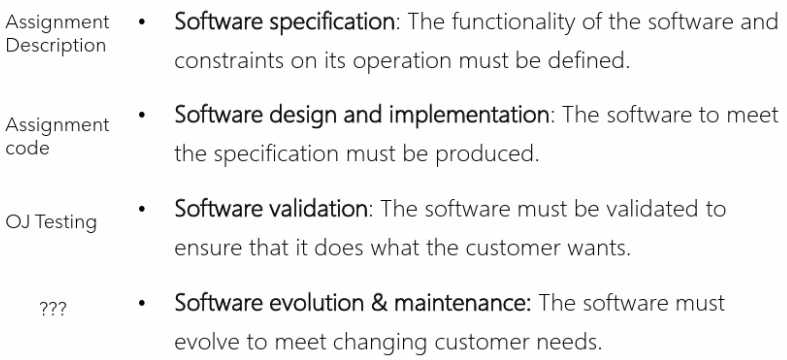
• Organizations want a well–defined, well–understood, repeatable software development process

• Benefits

What is software process?

Activities

Software process is a set of related activities that take place in sequence, and leads to the production of a software product. All processes have some form of the following activities:



The form/iteration/timing/execution of these activities defines different families and methodologies of software process

• Plan–driven processes: processes where all of the process activities are planned in advance and progress is measured against this plan.

• Agile processes: planning is incremental and it is easier to change the process to reflect changing customer requirements.

Products

the outcomes of a process activity.

Roles

Examples of roles are project manager, configuration manager, programmers,

customers etc.

Pre– and post–conditions

Pre– and post–conditions, which are statements that are true before and after a processactivity has been enacted or a product produced.

For example, before architectural design begins, a pre-condition may be that allrequirements have been **approved by the customer**;

After this activity is finished, a post-condition might be that the UML models describingthe architecture **have been reviewed**

Process Models

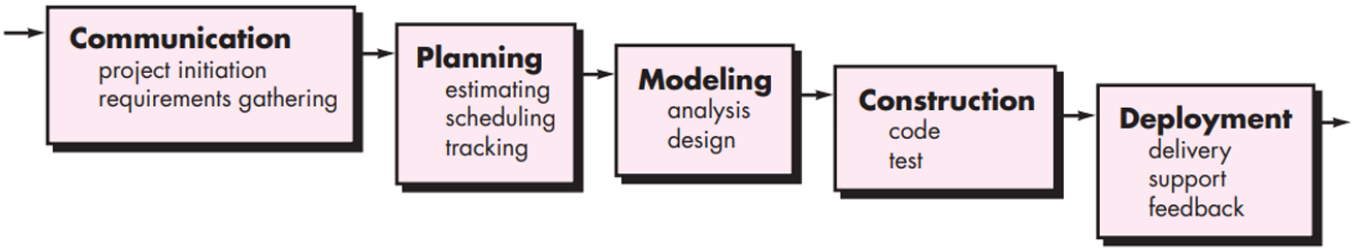
cons

Process models forget the frailties of the people who build computer software

Process models often deal with people's common weaknesses with discipline (对于人类，高纪律的方法是错误的)

Waterfall Model

- In principle, the result of each phase is one or more documents that are approved
- The following phase should not start until the previous phase has finished.



Problems

Sequential Flow

Real projects rarely follow the sequential flow that the model proposes.Although the linear model can accommodate iteration, it does so indirectlyAs a result, changes can cause confusion as the project team proceeds.

Clear Requirements

It is often difficult for the customer to state all requirements explicitly. Thewaterfall model requires this and has difficulty accommodating the naturaluncertainty that exists at the beginning of many projects.

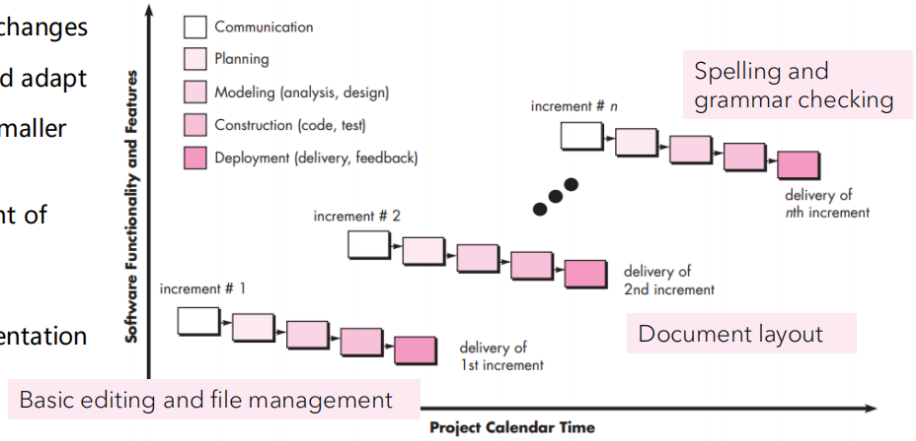
Customer Patience

A working version of the program(s) will not be available until late in the projecttime span. A major blunder, if undetected until the working program is reviewedcan be disastrous.

Incremental Process Models

pros

- Reducing cost due to requirement changes
- Easier to get customer feedback and adapt
- Easier to test and debug during a smaller iteration
- More rapid delivery and deployment of useful software
- Particularly useful when staffing is unavailable for a complete implementation



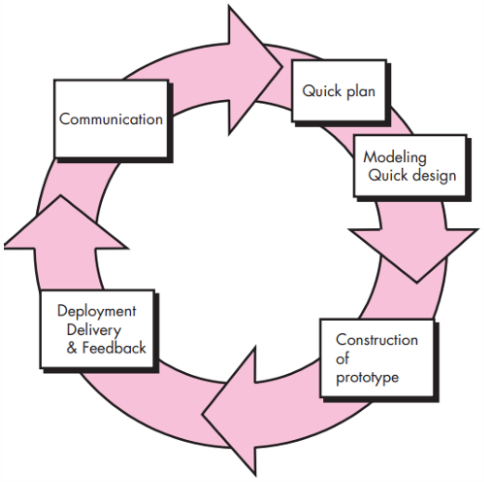
cons

- Needs good planning and design of the whole system before breaking it down for incremental builds
- System structure tends to **degrade** as new increments are added. Unless effort is spent on **refactoring** to improve the software, regular change tends to corrupt its structure.
- Incorporating further software changes becomes increasingly difficult and costly.

Evolutionary Process Models

prototyping model

- Prototyping (原型开发) might be used when customers define a set of general objectives for software, but do not identify detailed requirements for functions and features
- The prototype iteration begins with communication, followed by quick planning and modeling, which lead to the construction of a prototype that can be delivered and evaluated
- Stakeholders could quickly see what appears to be a working version of the software and give feedback

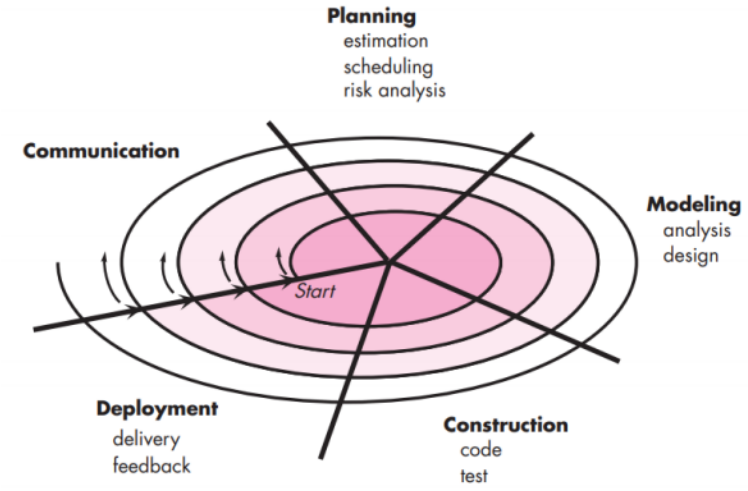


Prototyping should be used when the requirements are not clearly understood or are unstable

Prototyping can also be used for developing UI or complex, high-tech systems in order to quickly demo the feasibility

spiral model

The spiral model couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall



Comparing the two

> Prototyping: requirements are not clear;focusing on continuously communicating with the users
>
> Spiral: requirements are clear but adaptivefocusing on risk management

Agile Development

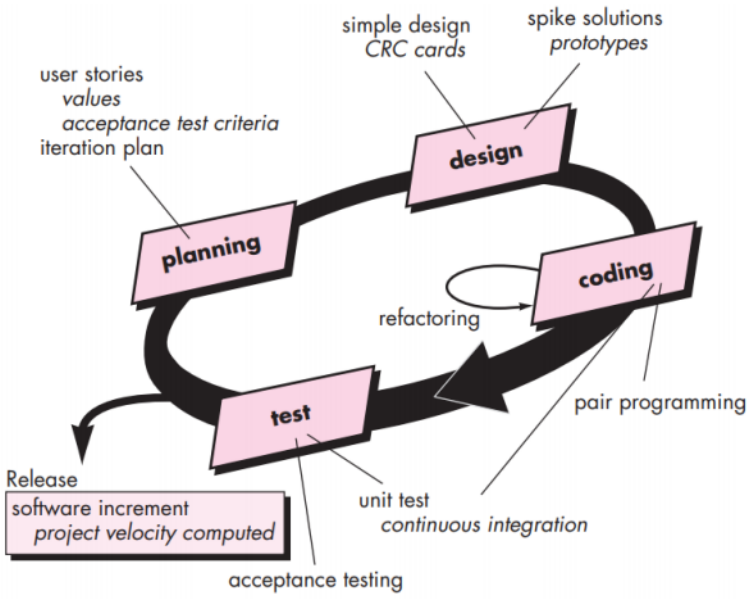
agile适合开发团队，而不是运营团队(operations team)

Extreme Programming

Extreme programming (XP) is probably the first well and Cunningham known agile process, created by Beck

As a type of agile software development, XP advocates frequent releases in short development cycles, intended to improve productivity and introduce checkpoints at which new customer requirements can be adopt.

XP uses an object-oriented approach as its preferred development paradigm



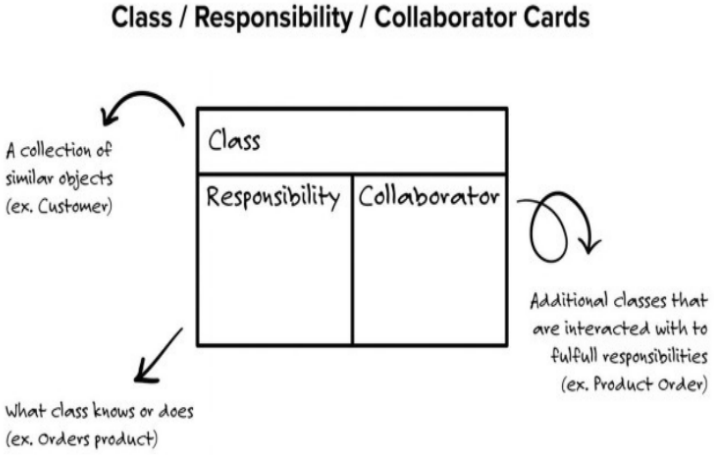
Planning

- Customers assign a value (i.e., a priority) to the story based on the overall business value of the feature or function.
- Members of the XP team then assess each story and assign a cost (measured in development weeks) to it

Design

XP PROCESS - DESIGN

- XP design following the KIS (keep it simple) principle
- XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context.
- CRC (class-responsibility-collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment



Coding

- A key concept during coding is pair programming
- Pair programming: two people work together at one computer workstation to create code for a story
- Code Review: Driver & navigator
- Refactoring: the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure, making code more maintainable and less error-prone

Testing

- Test-Driven Development (TDD)
  - Test first and make them automated
- Regression testing whenever code is modified
- After the first project release (also called a software increment) has been delivered, the XP team computes project velocity, which is the number of customer stories implemented during the release

SCRUM

SCRUM SUMMARY

Roles	Ceremonies	Artifacts
Product Owner	Sprint Planning	Product Backlog
Scrum Master	Daily Scrum	Sprint Backlog
Scrum Team	Sprint Review	Burndown Charts
	Sprint Retrospective	

SCRUM ROLES

Product Owner

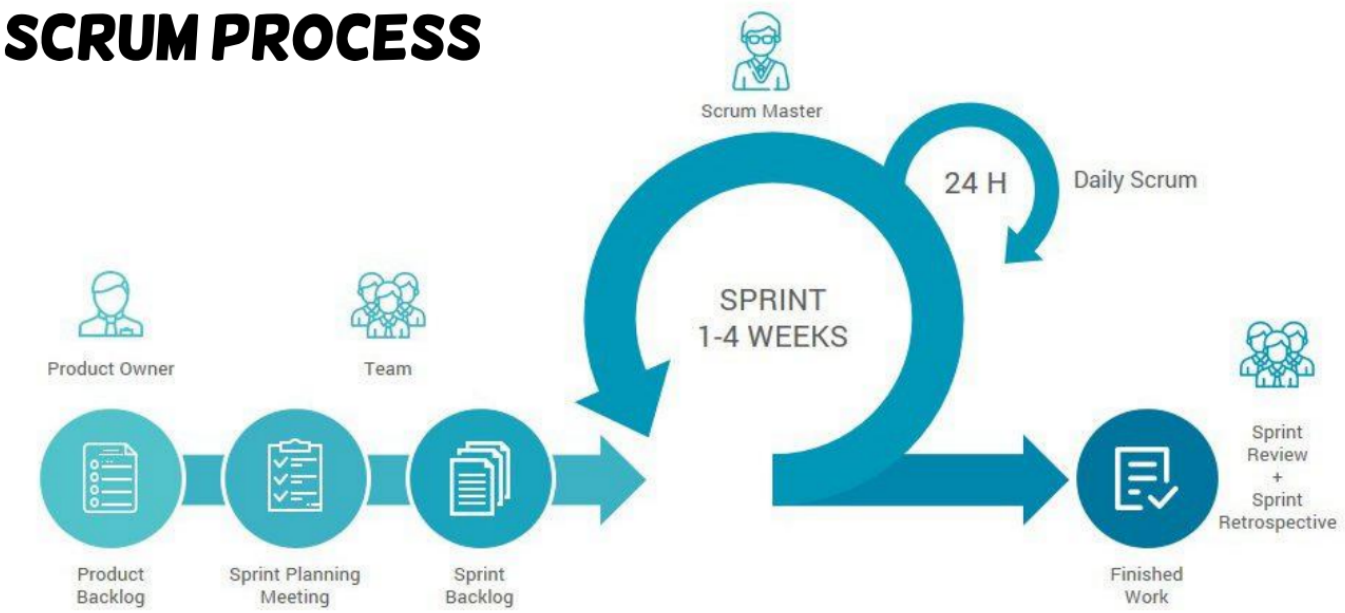
Define and adjust product features; Accept or reject work results

Scrum Master

Coach the team and enact Scrum values and principles

Scrum Team

Cross-functional members like developers, testers, designers (Typically 5-9 size)



Product Backlog (product owner)

一个有优先级和重要程度的todolist

Product backlog is similar to requirements in classic software process model.

However, product backlog is **prioritized and dynamic**, i.e., adapt to constant changes and refinements.

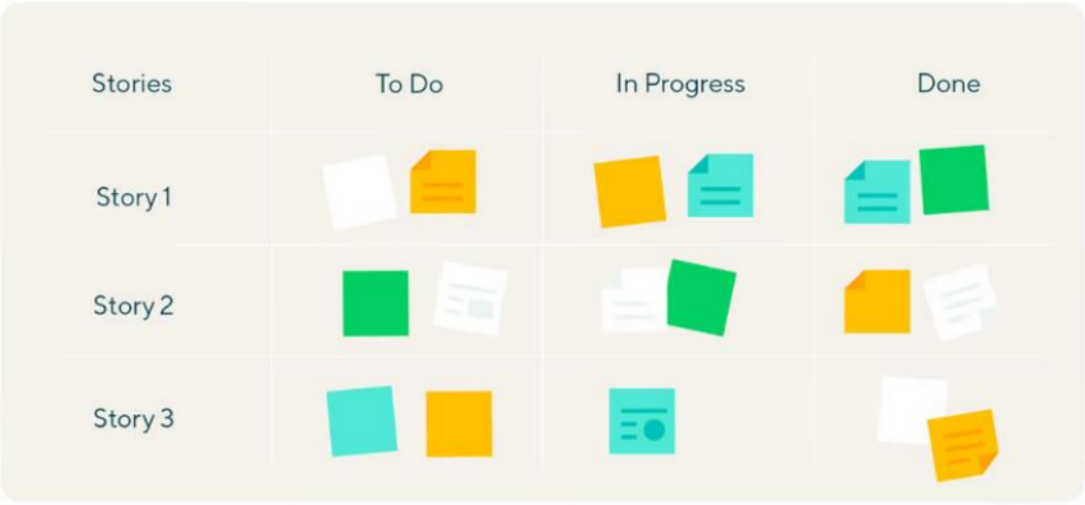
Sprint Planning Meeting & Sprint Bakclog (product owner & team)

Sprint (冲刺):Scrum项目管理方法中的一个常规、可重复的较短工作周期。在这个周期里，项目团队需快速完成预定的工作量（i.e., sprint backlog）

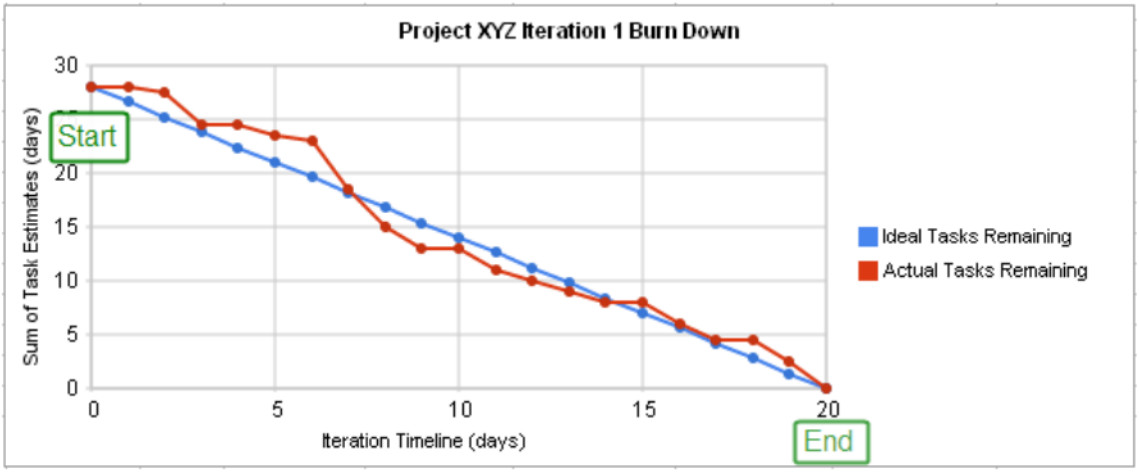
Sprint

A sprint duration is determined by the **Scrum Master**, typically lasts 30 days.

SCRUM Boards



Burndown Charts

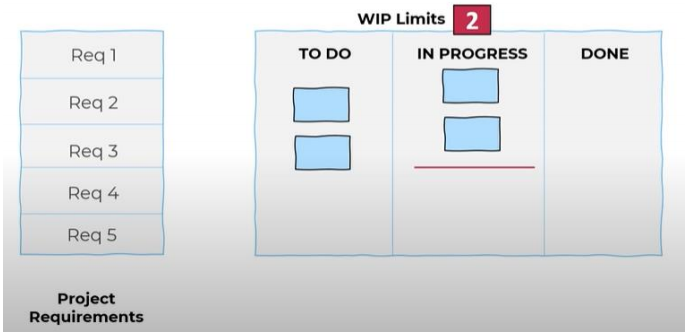


Daily Scrum (例会)

Sprint Review评审 (team presents to stakeholders) & Sprint Retrospective回顾 (all)

KANBAN

- Kanban focuses on maintaining a continuous task flow and continuous delivery
- Two primary principles of Kanban
  - **Visualization**: the work of the development team (features and user stories) are visualized
  - **Limited Work-in-progress** (WIP): the team is never given more work than it can handle



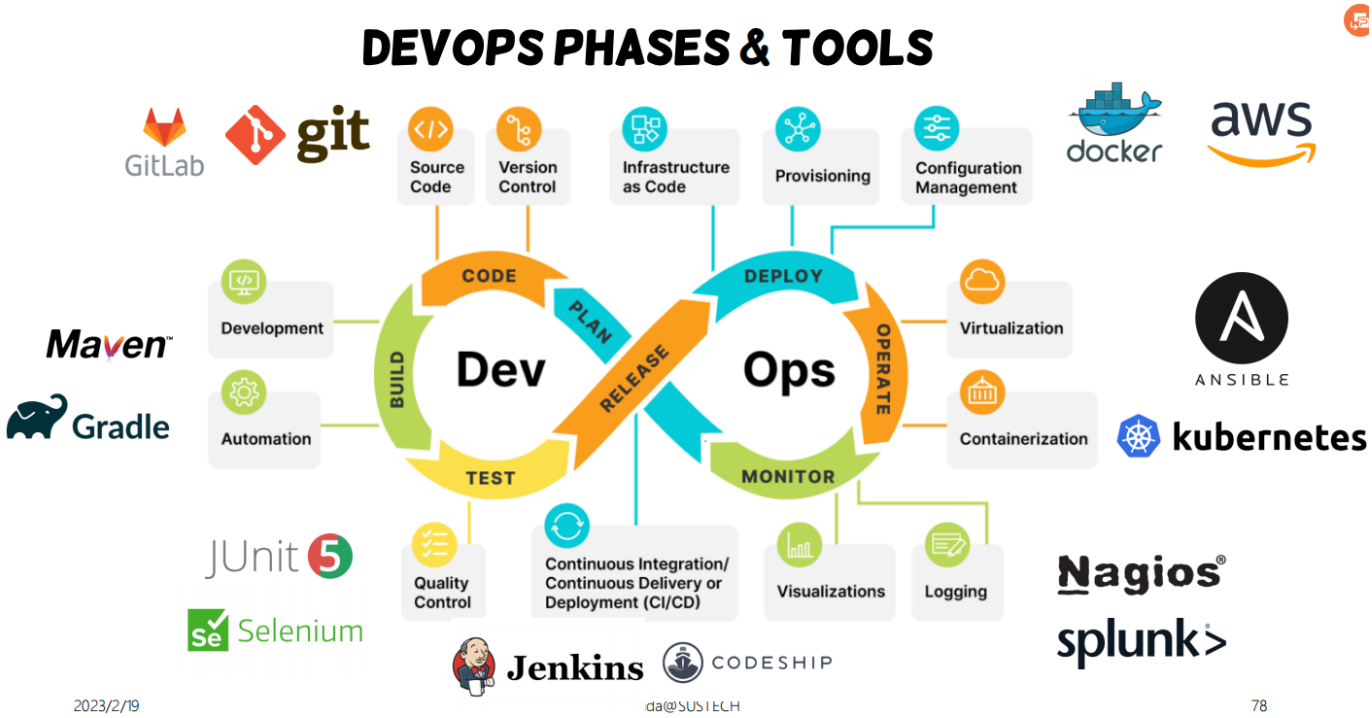
SCRUM VS. KANBAN

	Scrum	kanban
FLOW	has sprints.	no required time boxes or iterations. 只关心持续完成任务
ROLES	has a set of mendatory roles.	no set of roles, maybe an agile coach.
COMMITMENT承诺	每个sprint都保证固定数量的任务，如果保证太多会sprint失败	基于WIP限制，防止同一时间做太多任务
PLANNING	在每个sprint之前都有计划	没有计划 just-in-time palnning
WORKLOAD	完成每个sprint要求的任务数量	限制同一时间正在做的任务数量
CHANGES	sprint开始之后就不能新加任务了	可以持续新增以及减少任务
KPI	使用burndown chart和velocity chart，关心一个story完成的速度	关心lead time和cycle time，计算的是平均一个任务从开始到完成的时间
BOARD		有WIP limits
APPLICATION	适合快速开发，固定任务，需要协调，不成熟的队伍	适合对任务会不断地改变，成熟的队伍

DevOps

DevOps integrates developers and operations teams to improve collaboration and productivity by

- Automating infrastructure
- Automating workflows
- Continuously measuring application performance

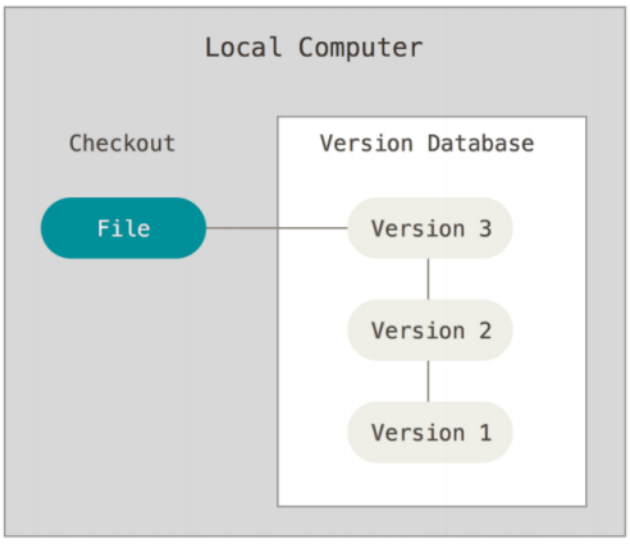


LECTURE 3–vcs

Version Control

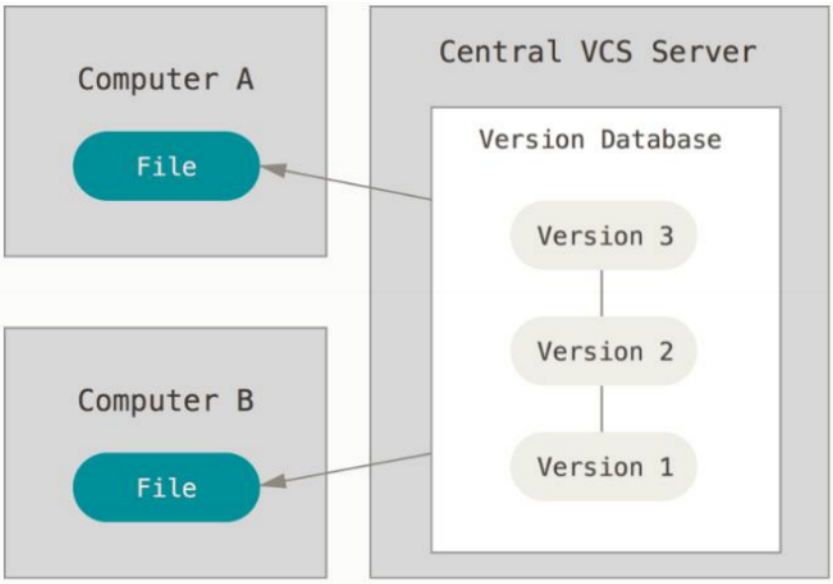
Local vcs– RCS

只在本地，不能和别人共享

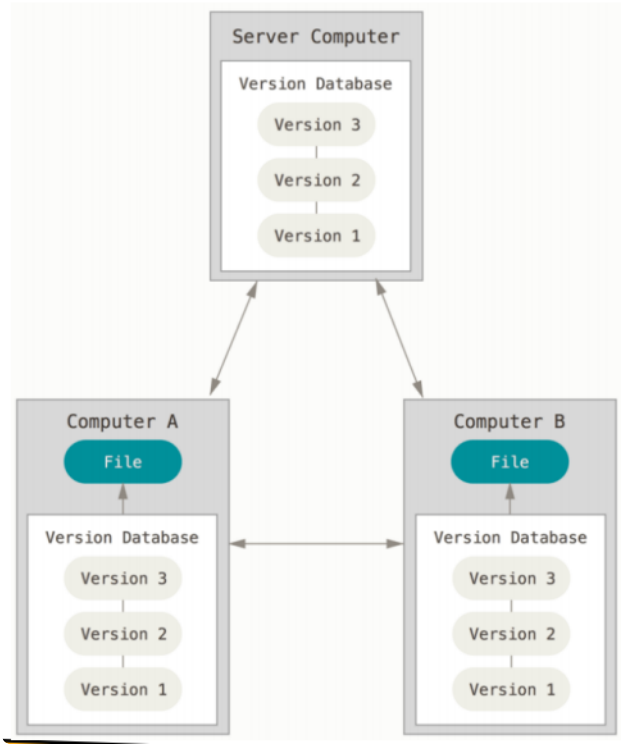


Centralized vcs– CVS, Subversion, and Perforce

单点问题，服务器毁了就都没了



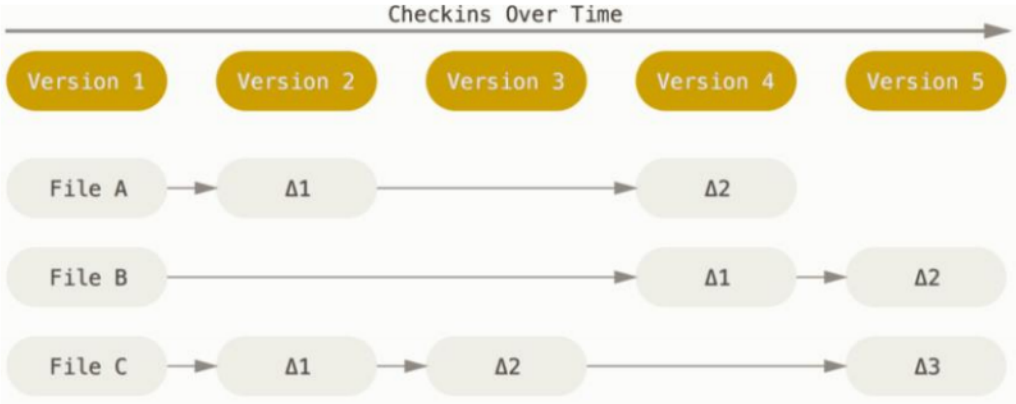
Distributed vcs–Git, Mercurial, Bazaar or Darcs



Git

Snapshots, Not Differences

others: delta-based version control: store information as a list of file-based changes



git: store a series of snapshots of a miniature filesystem, takes a picture of what all your files look like at that moment and stores a **reference** to that snapshot.

Nearly Every Operation Is Local

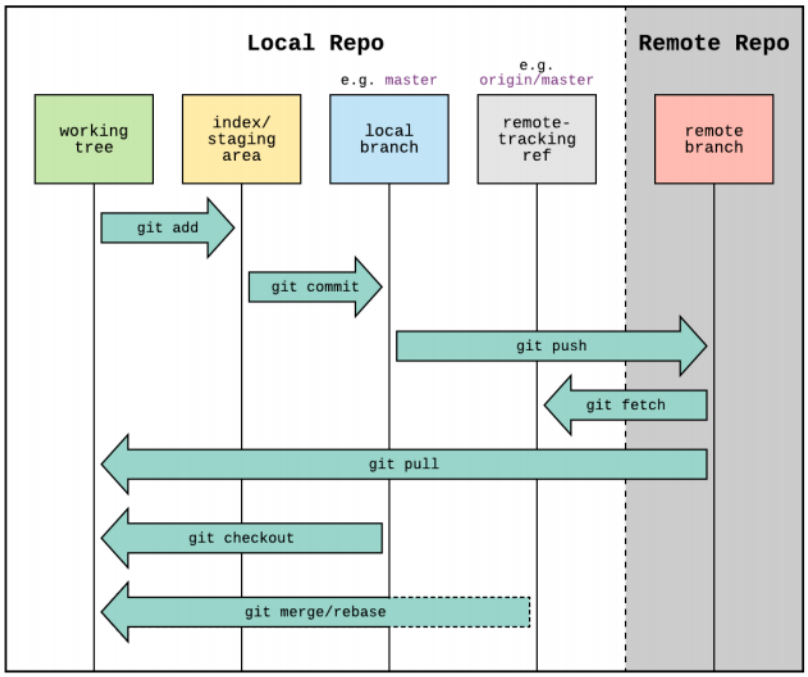
Git Has Integrity

The mechanism that Git uses for this checksumming is called a SHA-1 hash.

Git Generally Only Adds Data

It is hard to get the system to erase data in any way.

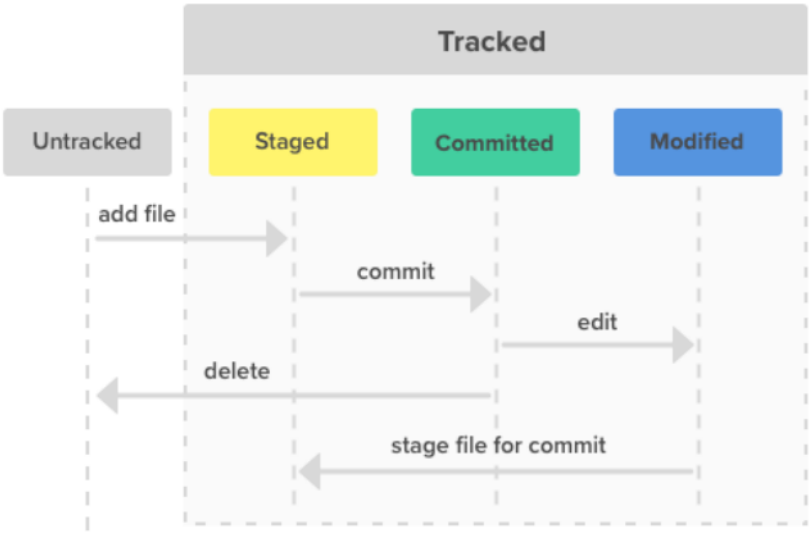
Git Architecture



pull=fetch + merge/rebase

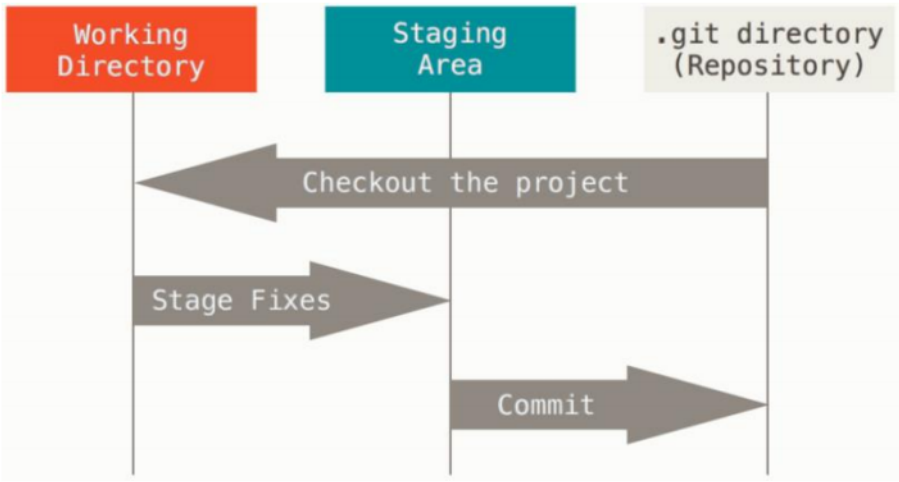
Git Local Repo

3 Stages



Untracked	Git is aware the file exists, but still hasn't saved it in its internal database.
Staged	marked a modified file in its current version to go into your next commit snapshot.
Committed	the data is safely stored in your local database
Modified	changed the file but have not committed it to your database yet.

3 Sections

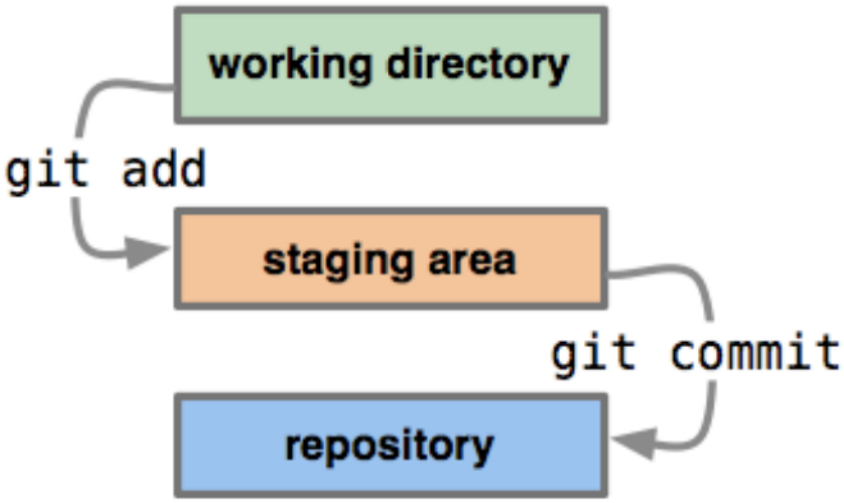


Working tree: a single checkout of one version of the **project**.

Staging area (index): a file in your Git directory, that stores information about what will go into your next commit.

Git directory (repository): where Git stores the metadata and object database for your project.

Workflow



modified file in working directory--git add-->staged file in staging area--git commit-->commited file in repository

Working With Remote Repo

Push works only if you cloned from a server to which you have **write access** and if nobody has pushed in the **meantime**.

Git Internals

key–value data store, persistent

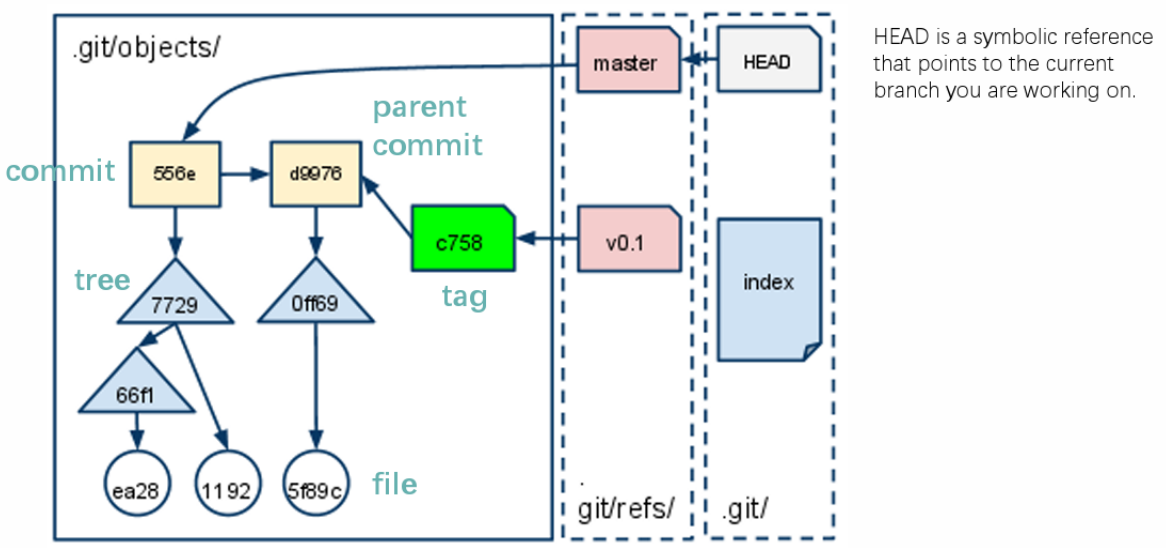
Key: hash

Value: a sequence of bytes representing files, directories, commits, etc.

Objects

content of files, directories, commits, and tags, identified by SHA–1 hash, stored in .git/objects/

Blob	file content, identified by a hash
Tree object	list of pointers to blob (file), or tree (directory), identified by a hash
Commit object	Reference to the top–level tree for the snapshot of the project at that point Parent commits if any Author info, commit message, etc.
Tag object	name associated with a commit (+ potential metadata)



References/branch

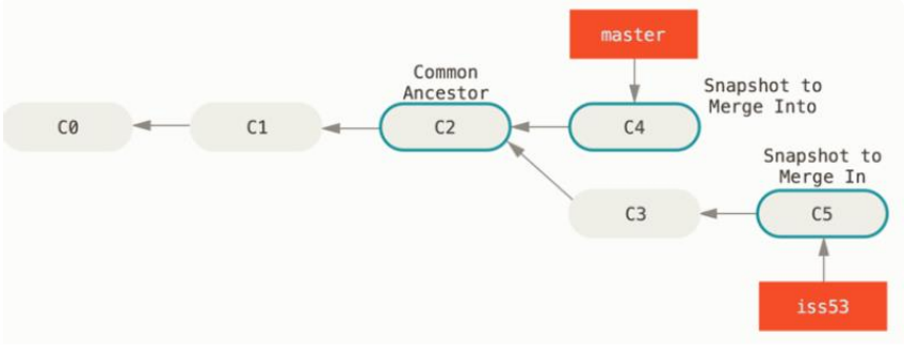
A branch, remote branch or a tag, which is simply a **pointer** to an object, stored in plain text in .git/refs/

Fast–forward merge

举例来说，当我们从 master checkout feature 分支进行开发，如果之后 master 都没有新的改动，那么当我们的 feature 分支和入 master 的时候，git 就会使用 fast forward 的方式进行

3-way Merge

举例来说，当我们从 master checkout feature 分支进行开发，如果之后 master 都没有新的改动，那么当我们的 feature 分支和入 master 的时候，git 就会使用 fast forward 的方式进行



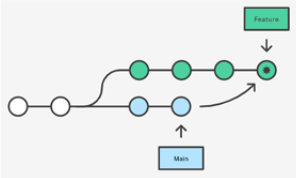
Git creates a new snapshot that results from this 3-way merge and automatically creates a new commit that points to it.

This is referred to as a merge commit, and is special in that it has more than one parent.

MERGING VS. REBASING

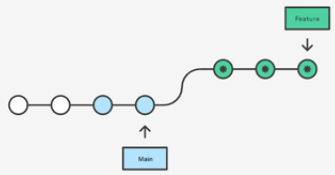
Merging

- Merging is a safe option that preserves the entire history of your repository
- Merge will generally create an extra commit



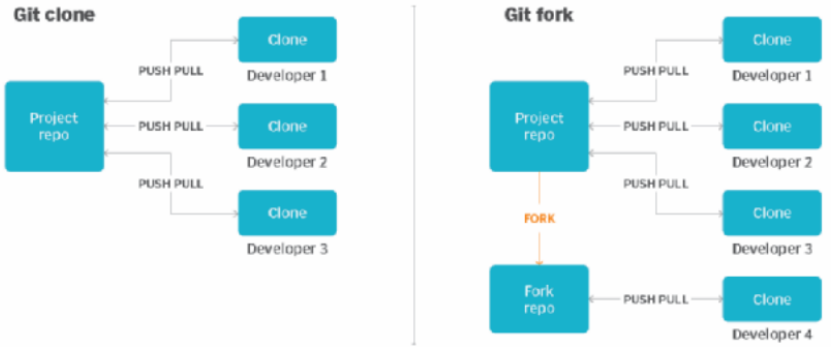
Rebasing

- Rebasing creates a cleaner, linear history by moving one branch onto the tip of another branch.
- **Be aware**: re-writing project history loses information, and can be potentially catastrophic for your collaboration workflow

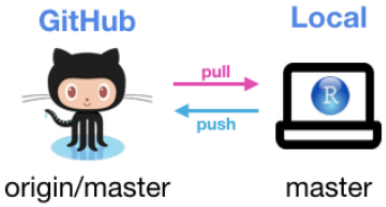


GIT CLONE VS. FORK

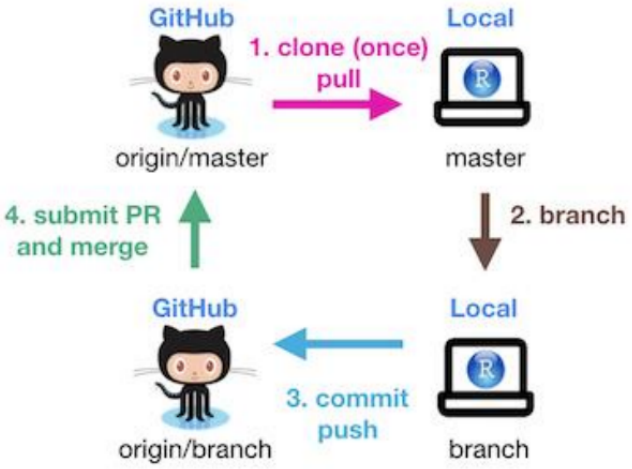
- Any public git repository can be forked or cloned
- Key difference: how much control and independence you want over the codebase once you've copied it.
  - Clone: a clone creates a linked copy that will continue to synchronize with the target repository.
  - Fork: A fork creates a completely independent copy of Git repo, disconnecting the codebase from previous committers.



POSSIBLE WORKFLOW I

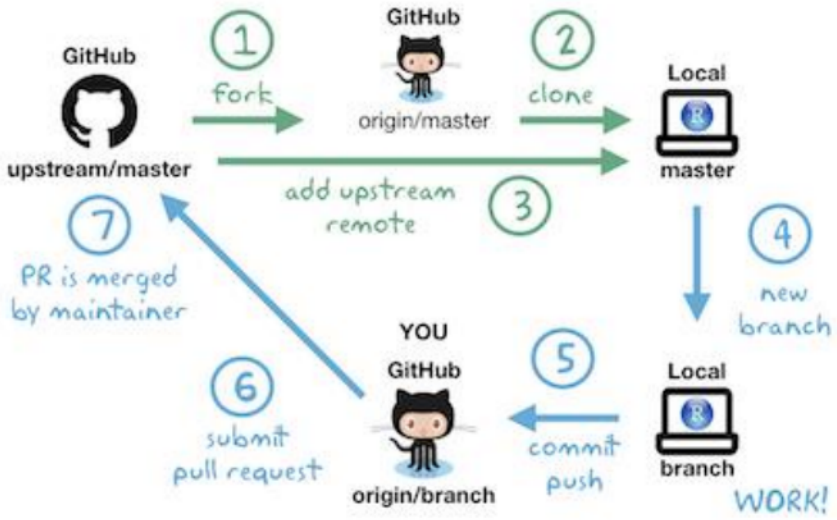


No-branch workflow



Your own repo with branching

POSSIBLE WORKFLOW II



Others' repo with branching

Index

是一个二进制文件，其他都是文件夹

a **staging area**, stored as a **binary file** in .git/index

Git Best Practices

Make clean, single-purpose commits

Commit early, commit often

Write meaningful commit messages

Don't commit generated files or dependencies

LECTURE 4-requirements

Requirements Overview

The requirements establish the system's **functionality**, **constraints**, and **goals**.

Severe deficiencies in software requirements

- Incomplete requirements
- Incorrect requirements
- Ambiguous requirements
- Inconsistent requirements

Stakeholders（涉众）

Primary Stakeholders

Customers

Project Managers

Business Analysts

Developers (belong to development team)

QA engineers (belong to development team)

UI&UX designers

Secondary Stakeholders

End–users

Government

Competitors

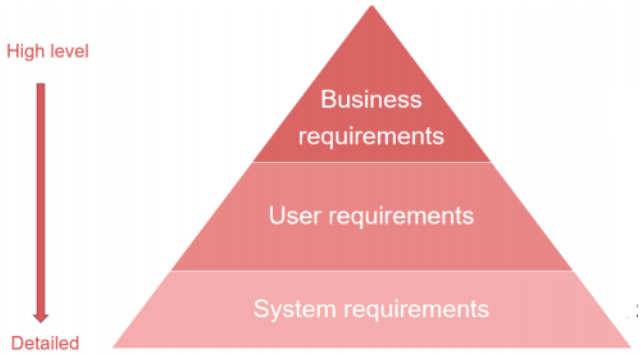
Media

Consultants

Other IT employees

Types of requirements

Software Requirements



Business View: define the **WHY** behind a software project.

User View: describe the **WHO** of a software project.

System View: dive into the **HOW** of a software project. (describe software as functional modules and non–functional attributes.)

Functional Requirements

in the form of **input** to be given to the system, the **operation** performed and the **output** expected.

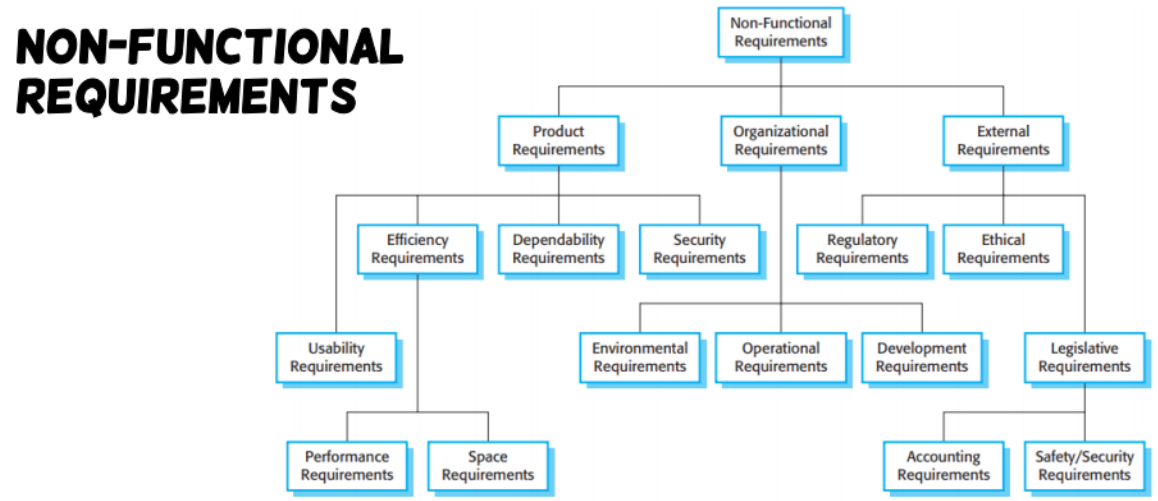
stated by the **user** which one can see **directly** in the final product.

Non–Functional Requirements

specify the software's **quality** attribute.

define the general characteristics, behavior of the system, and features that affect the experience of the user

like: performance, reliability, security, usability, 产品约束, 过程约束



Decompose Tasks

User requirements, as visions (愿景), need to be iteratively decomposed (分解) and refined (精化), until achieving detailed and actionable **system requirements**

AND relation between subtasks: the task is satisfied only if all subtasks are satisfied

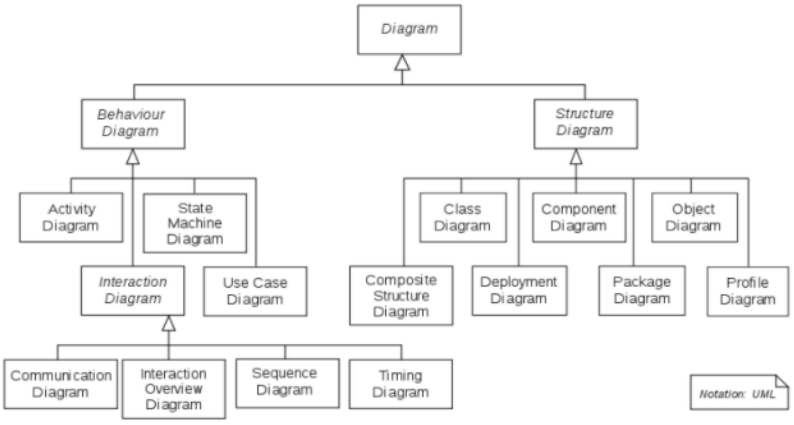
OR relation between subtasks: the task is satisfied if any one of the subtasks is satisfied

Requirements Modeling

Unified Modeling Language (UML)

In UML, a model consists of a **diagram** and a **specification**

particularly suited to **object–oriented** program development



Scenario–based models

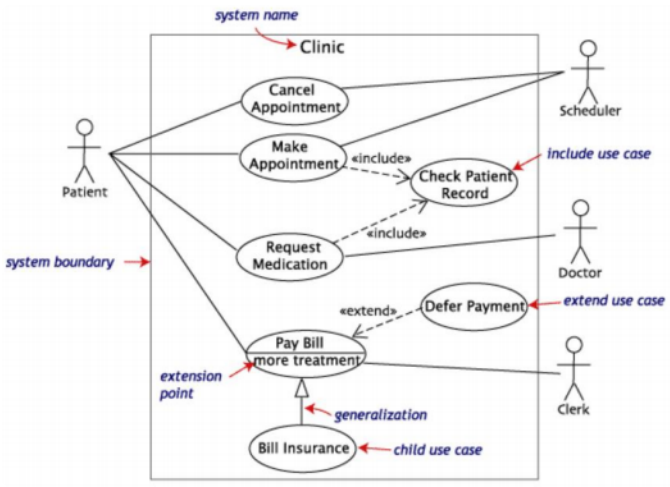
Scenarios capture the system, as viewed from the **outside** (e.g., by a user), using specific examples

Developing a scenario with a **client clarifies** many **functional requirements** that must be agreed before a system can be built

Murphy' s Law: “If anything can go wrong, it will”

Modeling scenarios as uses cases

Use Case Diagram(用例图)– behavior diagrams



Use cases

Actors

An actor can be human or an **external system**

Associations

<<extends>>: alternate flow or an exception

<<includes>>

generalize: one thing is more typical than the other thing

extension point

System boundary boxes

optional

Natural language template

When a use case involves a **critical activity** or describes a **complex set of steps** with a significant number of **exceptions**, a more formal approach may be desirable.

Metadata

- The **name** of the use case
- The **goal** of the use case
- The **actor** or **actors**
- **Trigger**
- **Entry conditions** at the beginning
- **Post conditions** at the end

Flow of events

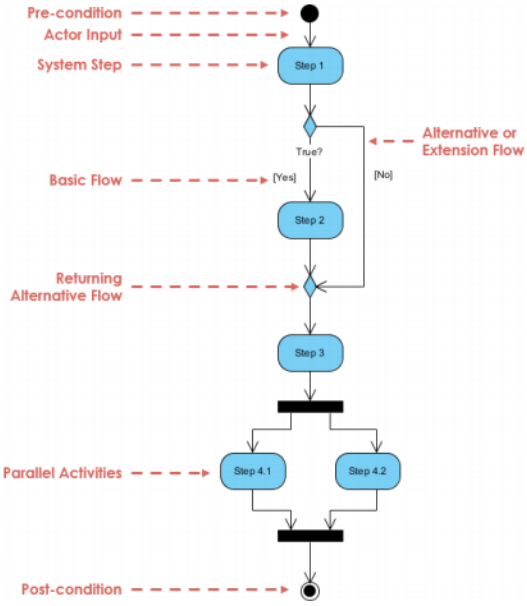
- The **basic flow** of events
- **Alternate flow** of events
- **Exceptions**

Alternate flows and **exceptions** model paths through the use case other than the basic flow

Interactions Between Use Cases

Activity Diagram(活动图)– behavior diagrams

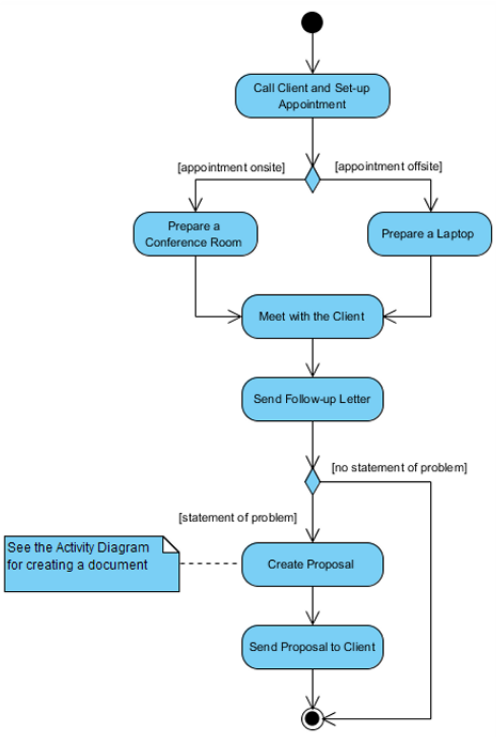
describes the **business** and **operational step–by–step** activities of the components in a system.



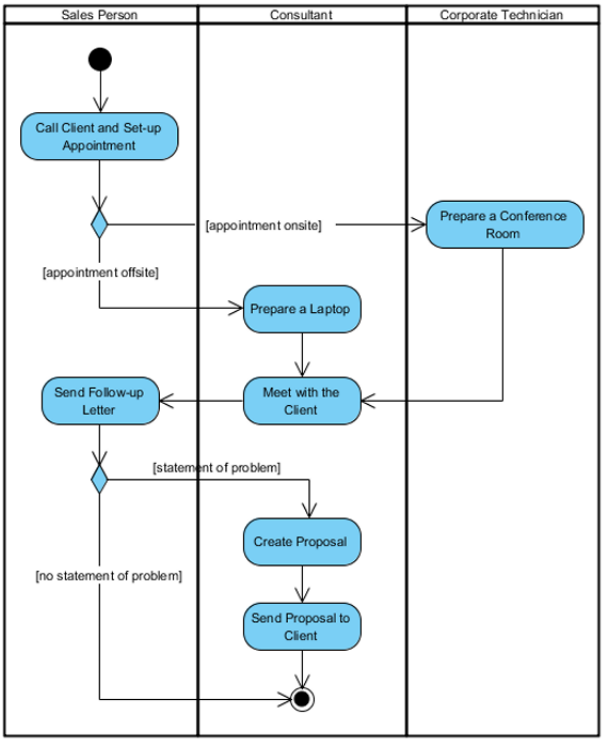
Swimlane Diagram(泳道图)– behavior diagrams

variation of the activity diagram

groups activities performed by the same **actor** on an activity diagram in a single thread (lane)



vs



Sequence Diagrams

Class-based models

objects, operations, relationships between the objects, collaborations between the classes

Analysis Class(分析类)

Analysis classes specify elements of an early conceptual model for things in the system that have **responsibilities** and **behavior**.

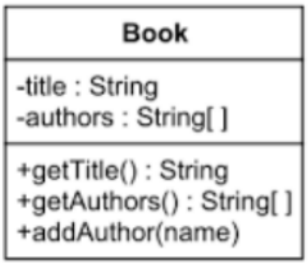
We might identify analysis classes by identifying **nouns** or **noun phrases** in the use cases

Analysis class	Description	Example
External entities (外部实体)	Entities that produce or consume information to be used in the system	People, devices, other systems
Things (事物)	Things that are part of the information domain of the problem	Reports, orders, products
Events (事件)	Events that occurred, typically with a timestamp	Emergency call, visit
Roles (角色)	Roles played by people who interact with the system	Manager, engineer, reader

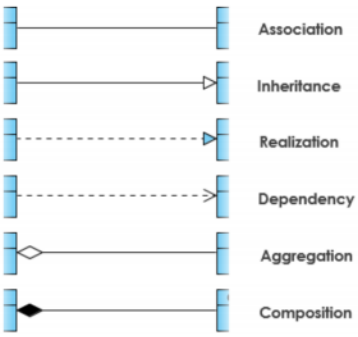
Analysis class	Description	Example
Organizational units (组织单元)	Organizational units that are relevant to an application	Group, team, department, college
Places (场地)	Places that establish the context of the problem and the overall function of the system	Room, floor
Structures (结构)	Structures that define a class of objects or related classes of objects	Sensors, vehicles, computers

Class Diagrams– structure diagrams

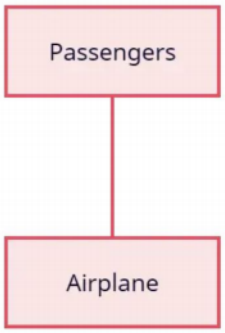
Attributes & Operations



Relations



Association: owns



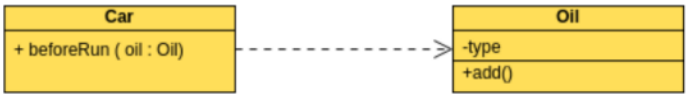
Aggregation: **separate** lifetimes

A book shop is an aggregation of books

Composition: **same** lifetimes

A tree is a composition of leaves

Dependency: use



The car uses gasoline. If there is no gasoline, the car will not be able to drive.

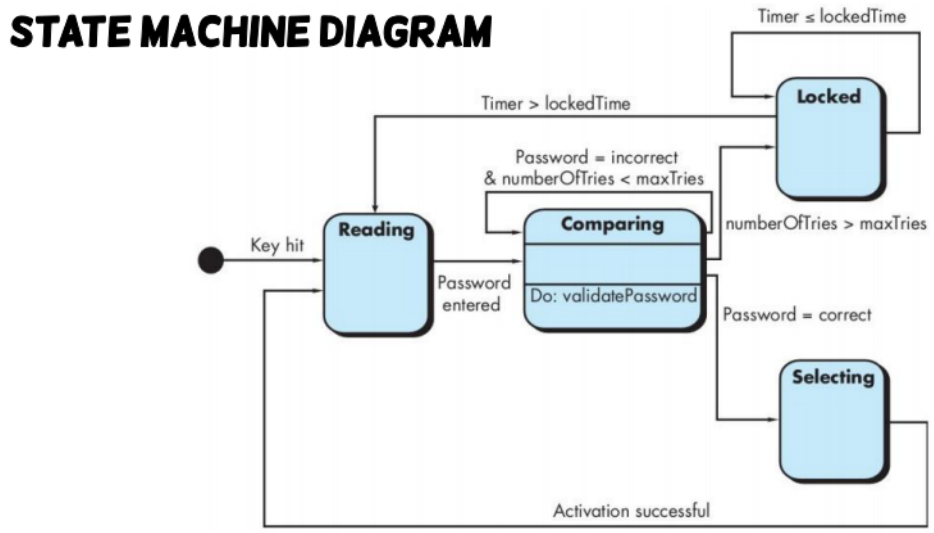
Multiplicity

Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

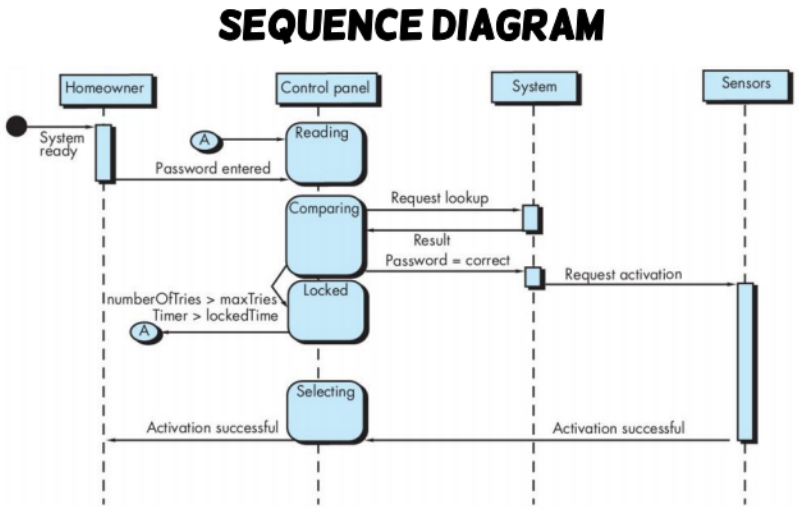
Behavioral models

a system performs actions due to **external events**, with changes of **internal states**

State Machine Diagram– behavior diagrams

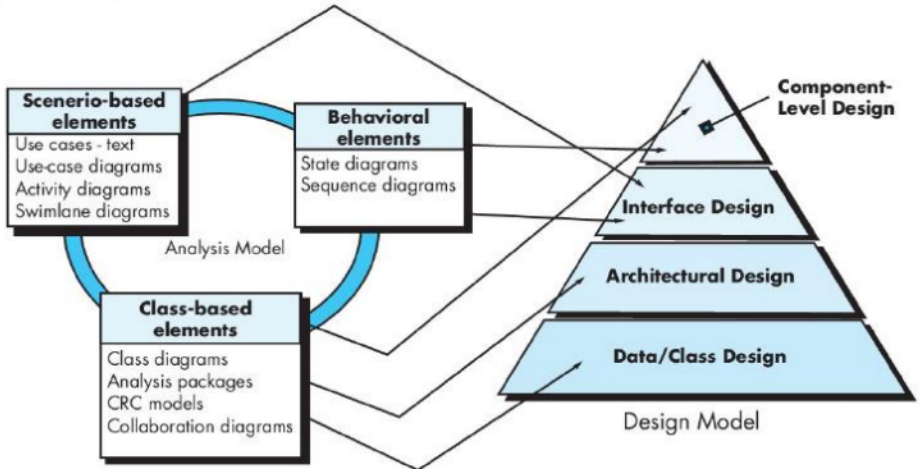


Sequence Diagram– behavior diagrams



LECTURE 5–design

Design model



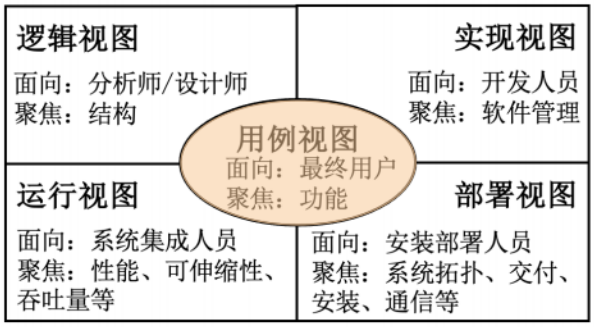
Data Design

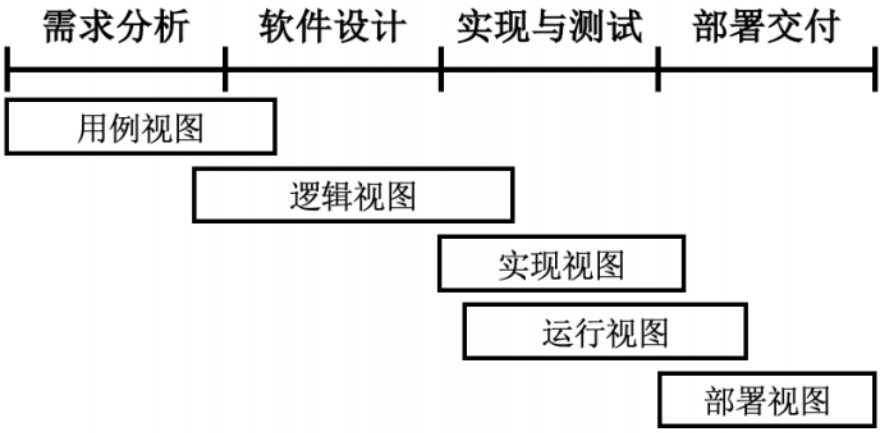
DATA DESIGN IS IMPORTANT

- At the program component level, the design of data structures and associated algorithms is essential to create high-quality application
- At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system
- At the business level, the collection of information stored in database enables data mining or knowledge discovery that affects the business success

Architectural Design

4+1 View Model





Use Case View

use case diagram

Logical View

class diagram, state diagram, and component diagram

Development View

package diagram and component diagram

Process View

activity diagram and sequence diagram

Deployment View

deployment diagram

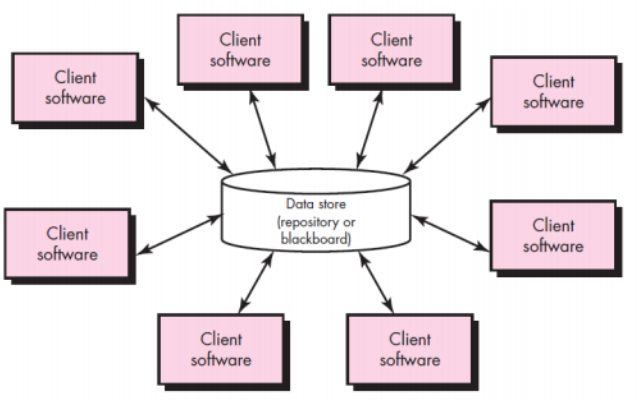
Architectural Style

Different architectural styles are NOT mutually exclusive; instead, they are often applied in combination

For example, a **layered** style can be combined with a **data-centered** architecture in many **database** applications.

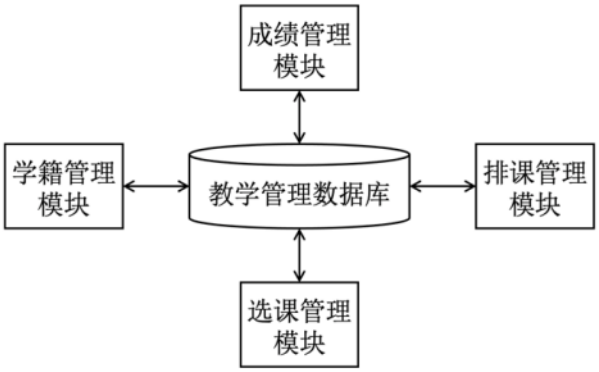
In a browser-server **web** application, both **layered** architecture and **object-oriented** architecture can be applied

Data-centered architecture



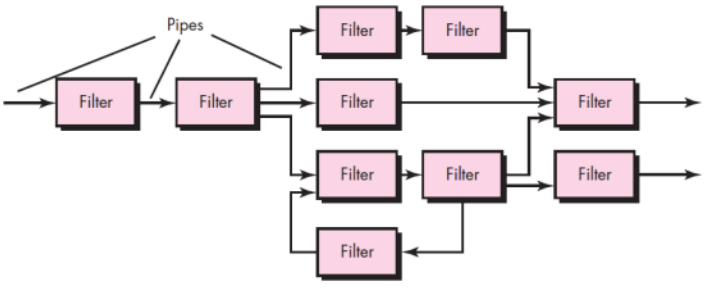
以数据为中心的体系结构

- This architecture promotes integrability (可集成性): ability to make separately developed components to work correctly together
- Client components operate independently
- Data can be passed among clients using the blackboard mechanism



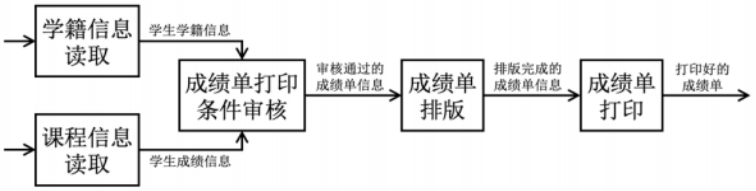
以数据为中心的体系结构示例

Data-flow architecture

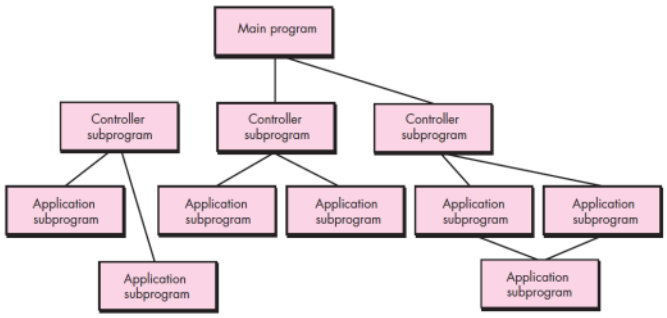


数据流体系结构（管道和过滤器）

- Data-flow architecture is suitable for automated data analysis and transmission systems
- Such systems contain a series of data analysis components, with almost no user interaction
- Data-flow architecture may not be suitable for GUI intensive systems



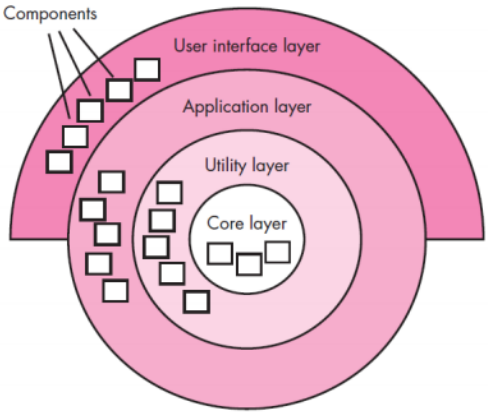
Call–and–return architecture

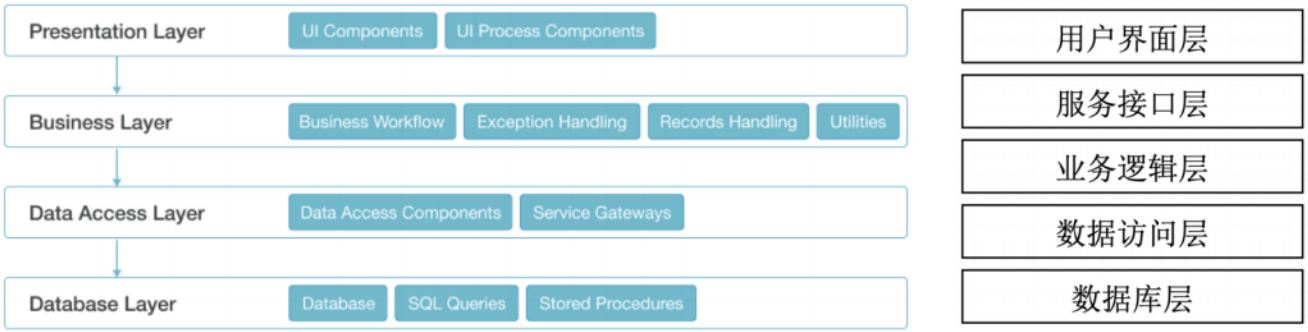


主程序/子程序体系结构

- **Main program/subprogram architecture** decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components.
- **Remote procedure call architecture**: The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Layered architecture

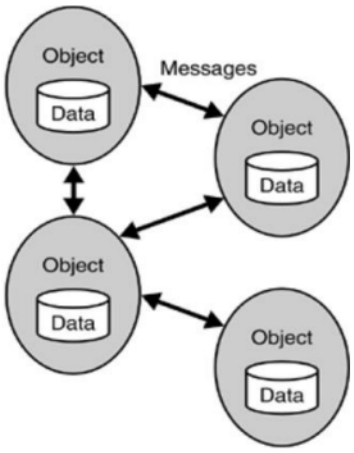




Layered architecture for web applications

Object-oriented architecture

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components are accomplished via message passing



Interface Design

UI Design

UI design incorporates different elements

• Aesthetic elements (美学): layout, color, graphics, interaction mechanisms, etc.

• Ergonomic elements (人体工程学): information layout and placement, UI navigation

• Technical elements (技术元素): UI patterns, reusable components

Eeternal/Internal Interface Design

The design of external interfaces should incorporate error checking and (when necessary) appropriate security features

Component Level Design

The component-level design for software fully describes the **internal detail** of each software component.

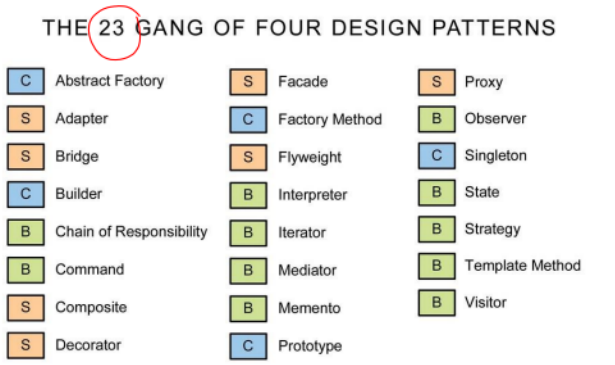
- **Data structures** for all local data objects
- **Algorithmic details** for all processing that occurs within a component
- **Interfaces** that allow access to all component operations (behaviors)

Design concepts

Abstraction

Pattern

- A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.
- It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.



Gangs of Four Design Patterns is the collection of 23 design patterns from the book "Design Patterns: Elements of Reusable Object-Oriented Software".

Separation of concerns

Modularity

Modularity (模块化) is the most common manifestation of **separation of concerns**.

We should avoid undermodularity and overmodularity

Imformation Hiding

Functional Independence

loose coupling, high cohesion

OO Design Concepts (SOLLID)

Single Responsibility Principle

high cohesion

Every class, module, or function should have **only one** reason to change

```
public class Order{
  …
  //根据订单金额和折扣规则计算需要支付的金额
  public float calculateAmountToPay(){
    …
  }
}
```

No single responsibility: changes to total and discount both affect this function

```
public class Order{
  …
  //计算订单总金额
  public float calculateTotal(){
    …
  }
  //计算折扣金额
  public float calculateDiscount(){
    …
  }
}
```

High cohesion for functions

Open Closed Principle

open for extension, but closed for modification

```
public void drawFigure(Shape[] shapeList){
  …
  for(int i=0;i<shapeList.length();i++){
    //如果是圆形
    if(shapeList[i].type.equals("circle")){
      …
    }
    //如果是矩形
    if(shapeList[i].type.equals("rectangle")){
      …
    }
  }
}
```

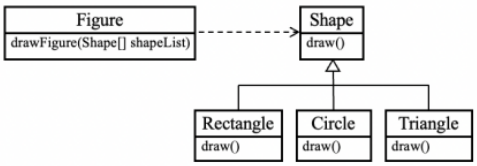
×

```
public void drawFigure(Shape[] shapeList){
  …
  for(int i=0;i<shapeList.length();i++){
    //调用图形类的抽象绘制方法
    shapeList[i].draw();
  }
}
```

```
public abstract class Shape{
  //图形绘制的抽象方法
  public abstract void draw();
  …
}
```

```
public class Circle extends Shape{
  //重写父类的图形绘制方法，提供针对圆形的具体实现
  public void draw(){
    …
  }
  …
}
```

√



Liskov Substitution Principle

The Liskov Substitution Principle (里式替换原则) states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

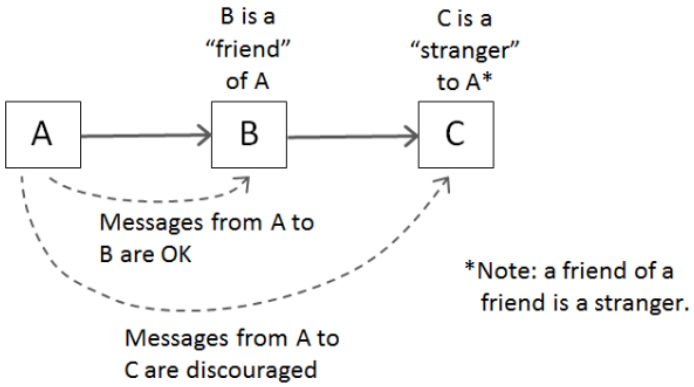
Objects of subclasses should behave in the **same way** as the objects of superclass.

The Liskov Substitution Principle is supported by object–oriented design abstraction concepts of **inheritance** and **polymorphism**.

Law of Demeter

The Law of Demeter (LoD), or principle of least knowledge, is a specific case of loose coupling.

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.



https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694

```
//展示一本图书的信息
public void displayBook(Book book){
   …
   //展示图书作者信息
   Author author=book.getAuthor();
   display(author.getName());
   …
}
```

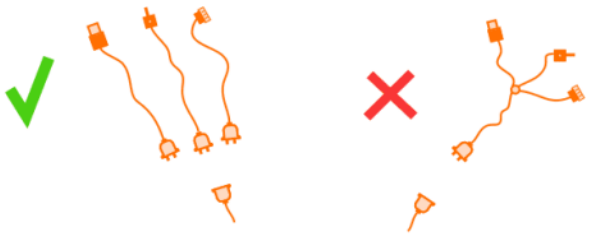
×

```
//展示一本图书的信息
public void displayBook(Book book){
   …
   //展示图书作者信息.
   display(book.getAuthorName());
   …
}
```

√

Interface Segregation Principle

a client should not be exposed to methods it doesn't need



例子，图一圈红就是客户本身不需要的

INTERFACE SEGREGATION PRINCIPLE

```php
interface Exportable
{
    public function getPDF();
    public function getCSV();
}
```

```php
class Invoice implements Exportable
{
    public function getPDF() {
        // ...
    }
    public function getCSV() {
        // ...
    }
}
```

```php
class CreditNote implements Exportable
{
    public function getPDF() {
        throw new \NotUsedFeatureException();
    }
    public function getCSV() {
        // ...
    }
}
```

https://accesto.com/blog/solid-php-solid-principles-in-php/

```php
interface ExportablePdf
{
    public function getPDF();
}
```

```php
interface ExportableCSV
{
    public function getCSV();
}
```

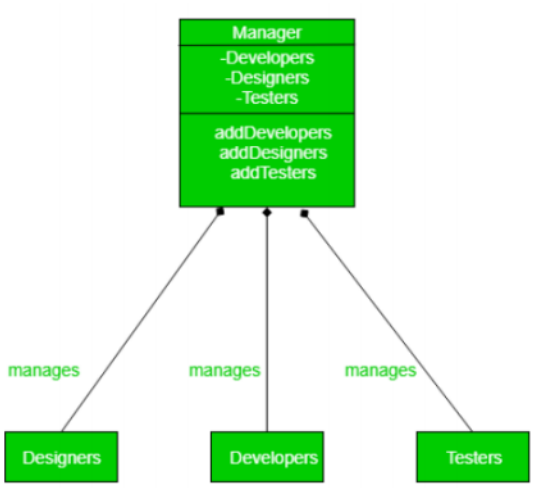
```php
class Invoice implements ExportablePdf, ExportableCSV
{
    public function getPDF() {
        //
    }
    public function getCSV() {
        //
    }
}
```

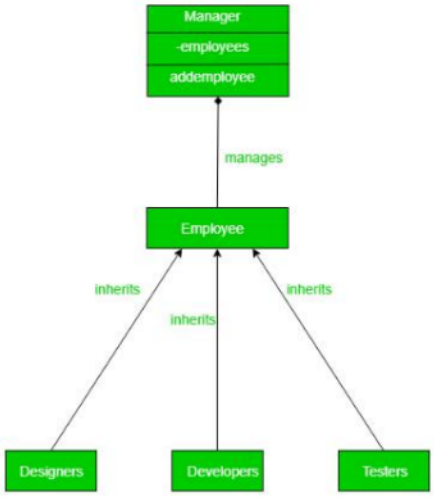
```php
class CreditNote implements ExportableCSV
{
    public function getCSV() {
        //
    }
}
```

https://accesto.com/blog/solid-php-solid-principles-in-php/

Dependence Inversion Principle

reduce dependencies to specific implementations, but **rely on interfaces**.





LECTURE 6–build

Dependency

Internal vs External Dependency

Internal vs External Dependency

- **Internal dependency**: depend on another part of your codebase
- **External dependency**: depend on code or data owned by another team (either in your organization or a third party)

Task vs Artifact Dependency

Task vs Artifact Dependency

- **Task dependency**: "I need to push the documentation before I mark a release as complete"
- **Artifact dependency**: "I need to have the latest version of the compute vision library before I could build my code"

Dependency Scopes

Dependency Scopes

- **Compile-time**: Foo uses classes or functions defined by Bar
- **Runtime**: Foo needs Bar (e.g., a database or network server) to be ready in order to execute
- **Test**: Foo needs Bar only for tests (e.g., JUnit)

Build system

Building is the process of creating a complete, executable software by **compiling** and **linking** the software components, external libraries, configuration files, etc.

Building involves assembling a large amount of info about the **software** and its **operating environment**.

Task-based Build Systems

Engineers define a series of steps to execute

System is flexible and powerful, but hard to guarantee correctness and parallelize

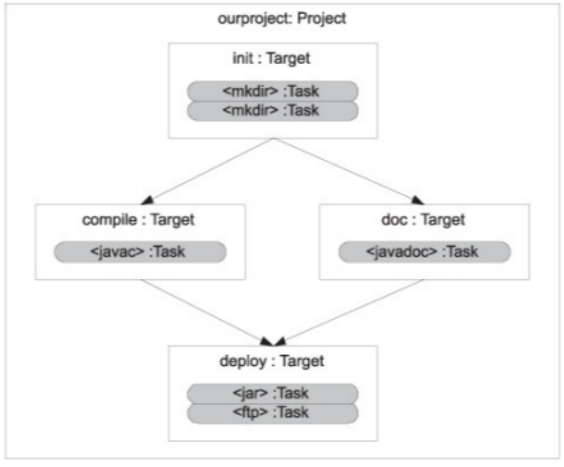
Most major build systems (e.g., Ant, Maven, Gradle, Grunt, and Rake), are task based

Most modern build systems require engineers to create **buildfiles** that describe how to perform the build (e.g., pom.xml for Maven)

对于开发人员灵活度高Engineers can write arbitrary code to execute any tasks during build

但是系统不好察觉依赖情况Systems can' t have enough information/control to always be able to run builds quickly and correctly

EXAMPLE: ANT DEPENDENCY & BUILDFILE



https://livebook.manning.com/book/ant-in-action/chapter-1/45

```
<?xml version="1.0" ?>
<project name="ourproject" default="deploy">

  <target name="init">
    <mkdir dir="build/classes" />
    <mkdir dir="dist" />
  </target>

  <target name="compile" depends="init">
    <javac srcdir="src"
      destdir="build/classes"/>
  </target>

  <target name="doc" depends="init" >
    <javadoc destdir="build/classes"
             sourcepath="src"
             packagenames="org.*" />
  </target>

  <target name="package" depends="compile,doc" >
    <jar destfile="dist/project.jar"
         basedir="build/classes"/>
  </target>

  <target name="deploy" depends="package" >
    <ftp server="${server.name}"
         userid="${ftp.username}"
         password="${ftp.password}">
      <fileset dir="dist"/>
    </ftp>
  </target>
</project>
```

Create two output directories for generated files

Compile the Java source

Create the javadocs of all org.* source files

Create a JAR file of everything in build/classes

Upload all files in the dist directory to the ftp server

Drawbacks

Difficulty maintaining & debugging build scripts

A依赖B，但是B不知道有人依赖他，之后B把自己路径改，A知道运行的时候报错才知道他依赖的B不见了

A依赖B，B依赖C，以及A需要的依赖刚好只有C才能生产，某天B不依赖C了，A就运行失败了

在我的电脑可以跑但是你那跑不了，因为我这依赖了一些固定的文件位置或者环境变量

每次build后的结果都不一样，因为依赖了一些可变的东西，不如网络实时下载的文件或者时间戳

race condition，A同时依赖了B和C，A每次会得到不一样的结果取决于BC谁快

Difficulty performing incremental builds

难以发现细小的变化，每次build都要重新跑一遍所有的

Difficulty of parallelizing build steps

hard to determine task dependencies and change impact，因此无法并行运行去build工程，以及
hard to distribute the build across multiple machines

Artifact-based Build Systems

Engineers declare a manifest describing the input and output（with the result but not how)

System provides strong guarantees about the correctness and easy to parallelize

In a build process, the role of system is producing **artifacts** (e.g., executable binary, documentation, etc.)

Engineers still need to tell the system **what** to build, **but how** to do the build would be left to the system

The approach that Google takes with **Blaze** (internal version) and **Bazel** (open-source version)

Buildfiles in artifact-based build

buildfiles

buildfiles define targets

describing:

• A set of artifacts to build

• Their dependencies

• Limited set of configurations

Engineers run blaze by specifying a set of targets to build (the "**what**")

Blaze is responsible for configuring, running, and scheduling the compilation steps (the "**how**")

targets

Every target has

• **name**, which could be used to reference this target

• **srcs**, which define the source files that must be compiled to create the artifact for the target

• **deps**, which define other targets that must be built before this target and linked into it

first time build

生成图依赖关系，从叶子节点开始build

1. Parse every BUILD file in the workspace to create a graph of dependencies among artifacts.
2. Use the graph to determine the transitive dependencies of `MyBinary`; that is, every target that `MyBinary` depends on and every target that those targets depend on, recursively.
3. Build (or download for external dependencies) each of those dependencies, in order.
   - Bazel starts by building each target that has no other dependencies and keeps track of which dependencies still need to be built for each target.
   - When all of a target's dependencies are built, Bazel starts building that target. This process continues until every one of `MyBinary`'s transitive dependencies have been built.
4. Build `MyBinary` to produce a final executable binary that links in all dependencies that were built in step 3.

Benefits

Parallelism

因为用的是自己的execution strategy所以更能保证一些事情，比如并行执行

Incremental Builds

因为依赖的是java编译器的结果，那么源代码不变，编译结果不变，也就不会重新build

因为它很懂自己编译的过程，所以有变化也只需要rebuild only the minimum set of artifacts，并且有所保证

Dependency management

Internal Dependendy

A target depends on other targets in the **same source repository**

strict transitive dependencies

Solution: Google enforces strict transitive dependencies on Java code by default.

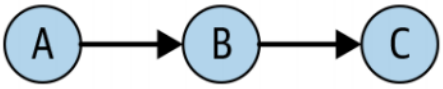


Figure 18-5. Transitive dependencies

- Blaze detects whether a target tries to reference a symbol without depending on it directly
- If so, the build fails with an error and a shell command that can be used to automatically insert the dependency
- Google also developed tools that automatically detect many missing dependencies and add them to a BUILD files without any developer intervention.

External Dependency

A target depends on artifacts built and stored **outside of the project** and typically **accessed via Internet**

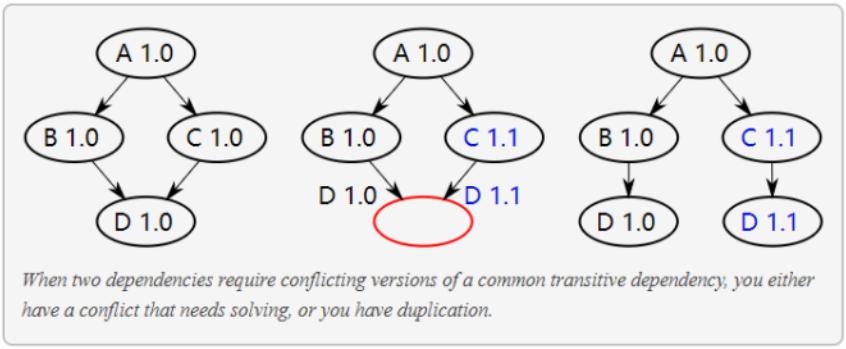
Reliability Risks– mirroring

Reliability Risks	**How to address**
If the third-party source (e.g., a maven repository) goes down, your entire build might grind to a halt if it's unable to download an external dependency.	The problem can be mitigated by mirroring any artifacts you depend on onto servers you control and blocking your build system from directly accessing third-party artifact repositories like Maven Central.

Diamond Dependency Issues

当依赖的两个库同时依赖另一个库但是是不同版本，你的工程要不存在冲突，要不存在重复

Diamond Dependency Issues lead to conflicts and unexpected results



https://www.tedinski.com/2018/03/27/maven-design-case-study.html

Semantic Versioning

Format: {MAJOR}.{MINOR}.{PATCH} (e.g., 2.4.72 or 1.1.4.)

- MAJOR version change indicates a change to an existing API that can break existing usage
- MINOR version change indicates purely added functionality that should not break existing usage
- PATCH version change indicates non-API-impacting implementation details (bug fixes) that are viewed as particularly low risk

Additional labels for pre-release and build metadata are available as extensions (e.g., 1.0.0-beta, 3.1.0-alpha.1)

对依赖的要求

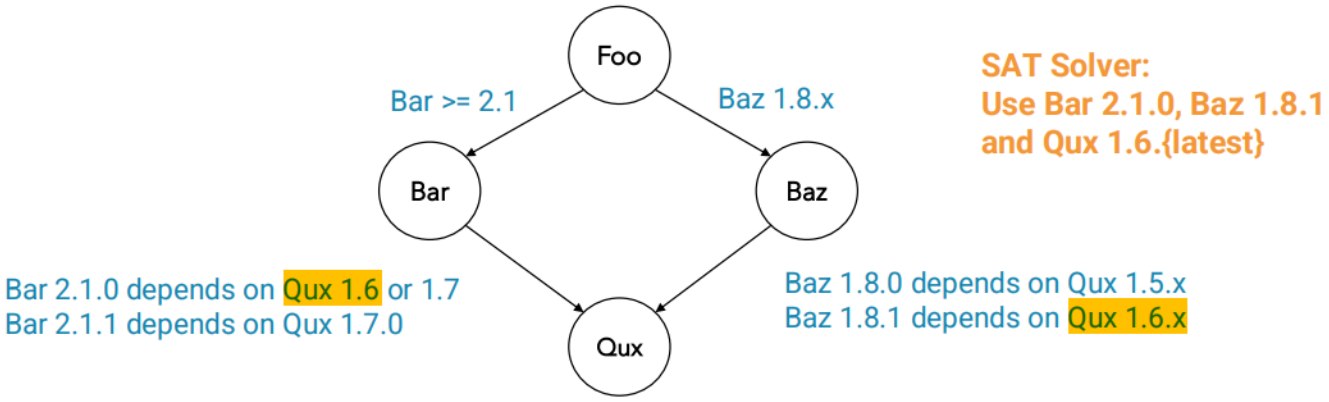
Declare dependency on “Bar ≥ 2.1”

√ • Bar 2.1
√ • Bar 2.1.0, 2.1.1
√ • Bar 2.2 onwards

× • Bar 2.0.x
× • Bar 3.x (some APIs in Bar were change incompatibly)

Bar和Baz依赖同一个库，为了不出现冲突因此要选择他们依赖的库一样的版本

Using SemVer, we can model the dependency problem as: given a set of constraints (version requirements on dependency edges), can we find a set of versions for the nodes that satisfies all constraints?



LECTURE 7–quality

External quality can be measured through feature tests, QA and customer feedback.

Internal quality can be measured through linters, unit tests etc.

Code quality

Understandability (可理解性)

Confusing identifiers命名不容易理解

Misleading indentation缩进

Deep nesting太多if

Improper exception type异常处理类型不对或者没必要

Potential bugs: no null–checking/missing/missing else block

Meaningful identifier names:

- Use Intention–Revealing Names
- Name Functions as Verbs
- Name Classes as Nouns
- Use Meaningful Distinction不要用arr1，arr2来区分
- Use Pronounceable Names
- Use Searchable Names 用有名字的常量不是，而不是数字

Maintainability (可维护性)

Good for:

Search

Extend

Modify & Reuse

Code Clone

Increase code size

Increase maintenance cost

types

Type 1: Exact copy, only differences in white space and comments.

Type 2: Same as type 1, but also variable renaming.

Type 3: Same as type 2, but also changing or adding few statements.

Type 4: Semantically identical, but not necessarily same syntax.

reduce methods

Extract method: Extract duplicate code to a new method

Pull up method: taking a method and "Pulling" it up in the **inheritance** chain.

Code Clone Detection

Text matching

No program **structure** is taken into consideration

Detect Type–1 clones & some Type–2 clones

Two types of text matching

• Exact string match: diff (cvs, svn, git) is based on exact text matching

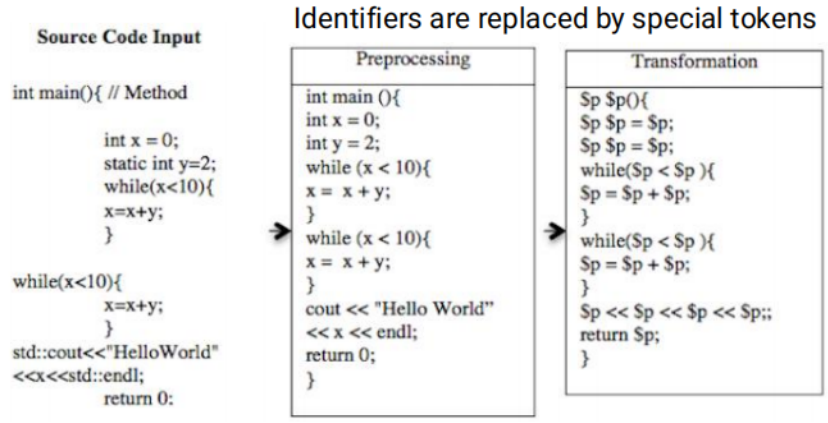
• Ambiguous match: LCS, n–gram match

Token sequence matching

No program structure is taken into account

Detect Type–1 and Type–2 clones

可以识别对变量重命名的



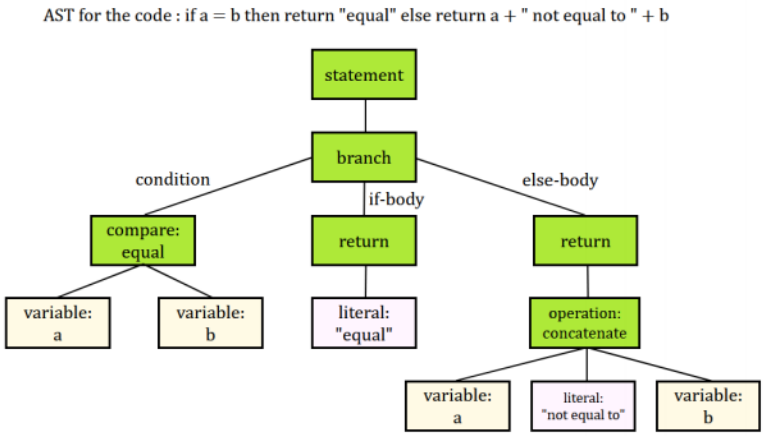
http://www.cs.uccs.edu/~jkalita/papers/2016/SheneamerAbdullahIJCA2016.pdf

Graph matching

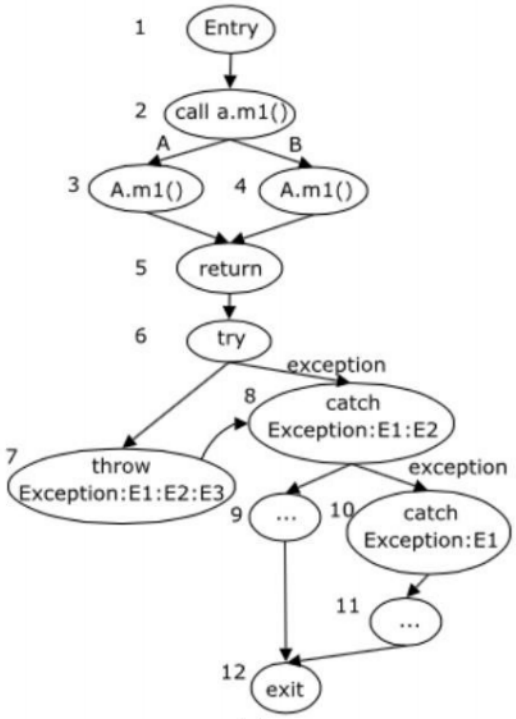
Detect Type–1, Type–2, and Type–3 clones

Graphs for modeling source code

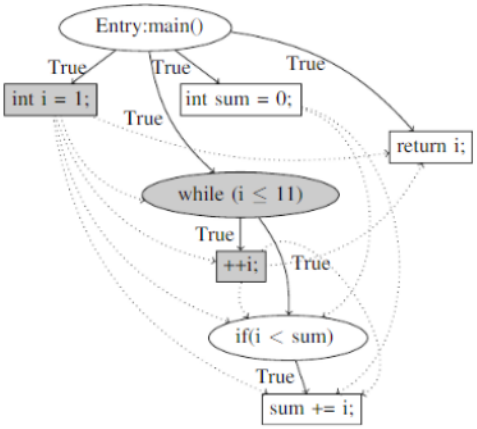
• AST (Abstract Syntax Tree)



• CFG (Control Flow Graph)



• PDG (Program Dependency Graph)




```
int main()
{
    int i = 1;
    int sum = 0;
    while(i < 11 ){
        if (i < sum)
            sum += 1;

        ++i;
    }
    return i;
}
```

Reliability (可靠性)

Error handling

For Reliability: proper error handling ensures the normal execution of the program

For Maintainability: proper error handling logs error conditions, which facilitate debugging and fixing.

Safety-critical system (SCS)- no incorrect

For SCS, error handling should value software **correctness**: the software could **terminate**, but should **never return incorrect results.**

Mission-critical system (MCS)- no terminate

For MCS, error handling should value software **robustness**: the software could tolerate certain errors, but **should not terminate**

Security (安全性)

Input validation

Efficiency (高效性)

time behavior

resource behavior

Portability (可移植性)

from one environment to another

Code review

Types of code changes to be reviewed

Modifications to existing code (Bug fixes and rollbacks)

Entirely new code

Automatically generated code (E.g., refactoring tools, intelligent code generators)

Benefits

Code correctness

Many correctness checks are performed automatically through techniques such as static analysis and automated testing

Code readability

Code consistency

Knowledge sharing

Code review flow at Google

At Google, code reviews take place before a change can be committed to the codebase; this stage is also known as a **precommit review**.

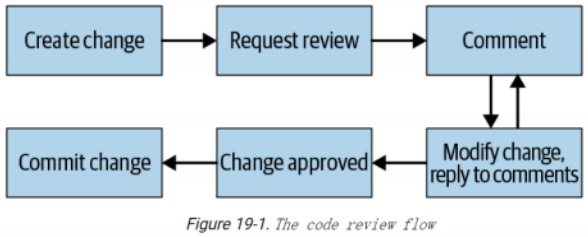


Figure 19-1. The code review flow

Best practice

write small changes

~200 LoC

write good change descriptions

Explain what is changed and why

• E.g., “bug fix” is not a good change description

A good change description also allows Code Search tools to find changes

keep reviewers to a minimum

automate where possible

LECTURE 8–metrics

WHAT SOFTWARE QUALITIES DO WE CARE ABOUT?

- Scalability
- Security
- Extensibility
- Documentation
- Performance
- Consistency
- Portability
- Installability
- Maintainability
- Functionality (e.g., data integrity)
- Availability
- Ease of use

WHAT PROCESS QUALITIES DO WE CARE ABOUT?

- On-time release
- Development speed
- Meeting efficiency
- Conformance to processes
- Time spent on rework
- Reliability of predictions
- Fairness in decision making
- Measure time, costs, actions, resources, and quality of work packages; compare with predictions
- Use information from issue trackers, communication networks, team structures, etc…

Code size still dominates many metrics

Metrics types

Product Metrics

e.g., size, complexity, performance, etc.

Process Metrics

e.g., # bugs found, bug fixing time, # features added

Project Metrics

e.g., # of developers, money spent, time spent

Measure the complexity of a system

Lines of code

normalizing

不要有没用的空行

language matters

不同的编程语言同样行数表达的信息量不同

LoC is a valid metric when

- Use within the **same** programming language
- Code measured use standard, consistent **formatting**
- Code has been **reviewed** first

Cyclomatic complexity (圈复杂度)

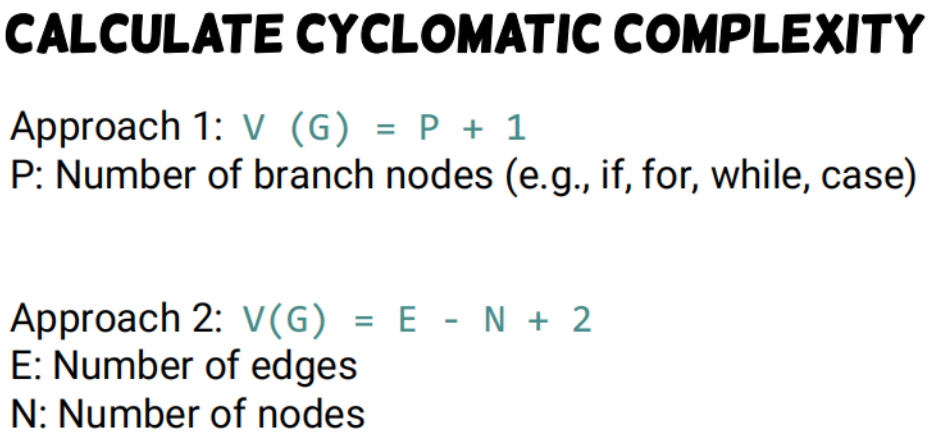
the complexity of code depends on the number of decisions in the code **(if, while, for)**

computed using the **control–flow graph** (控制流图) of the program

used to estimate the required effort for **writing tests**

复杂度一样不代表same readability & maintainability

复杂度高不代表low readability



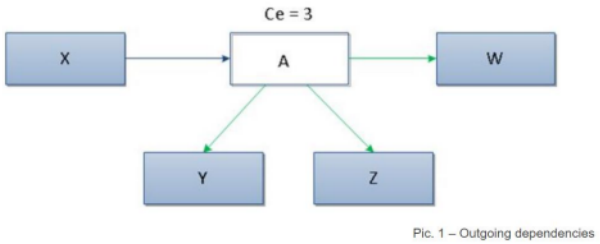
Coupling & Cohesion

- Dependences
  - Call methods, refer to classes, share variables

- Coupling
  - Dependences among modules (bad)
- Cohesion
  - Dependences within modules (good)

Martin's coupling metrics

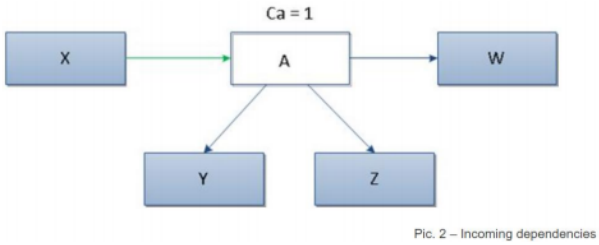
Efferent coupling (Ce)– 依赖别人的数量



Pic. 1 – Outgoing dependencies

Ce> 20 indicates instability of a package

Afferent coupling (Ca)– 被别人依赖的数量



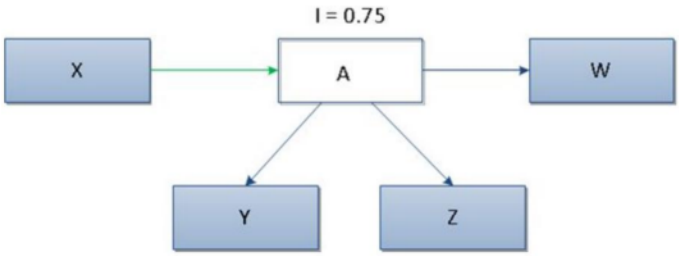
Pic. 2 – Incoming dependencies

High values of metric Ca usually suggest high component stability

Instability calculate– 依赖别人数/所有依赖

Instability: measure the relative susceptibility of class to changes

$$I = \frac{Ce}{Ce + Ca}$$



太高–>1：依赖的太多Ce high，不稳定

太低->0：被依赖太多Ca high，不好修改

OO Metrics

Weighted Methods per Class (WMC)

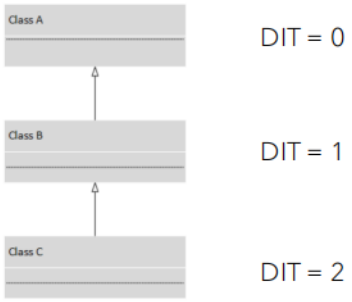
indicate the development and maintenance effort for the class

Classes with **large** numbers of methods are more likely to be **application specific and less reusable**

Depth of Inheritance Tree (DIT)

The deeper a class is in the hierarchy, the more methods it inherits and so it is **harder to predict** its behavior

The deeper a class is in the hierarchy, the more methods it **reuses**



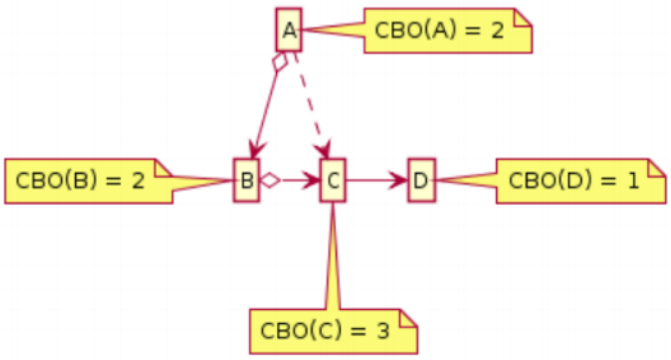
Number of Children (NOC)

immediate subclasses

A class with a large NOC is probably very **important** and needs a **lot of testing**

Coupling between Object Classes (CBO)

CBO doesn't care about the direction of a dependency



Response for a Class (RFC)

If a large number of methods can be invoked in response to a message, **testing** becomes more complicated

The more methods that can be invoked from a class, the greater the **complexity** of the class

Lack of Cohesion in Methods (LCOM)

LCOM counts the sets of methods in a class that are **not related** through the **sharing** of some of the class's fields.

不共享的对–共享的对

- Considers all pairs of a class's methods:
  - Q: In some pairs, both methods access at least one common field of the class
  - P: In other pairs, the two methods do not share any common field accesses

$$LCOM = P - Q$$

Cohesiveness of methods is a sign of encapsulation

Lack of cohesion implies that classes should be **split** (如果LCOM太低，说明方法里的方法都不太相关，这个类应该被分解)

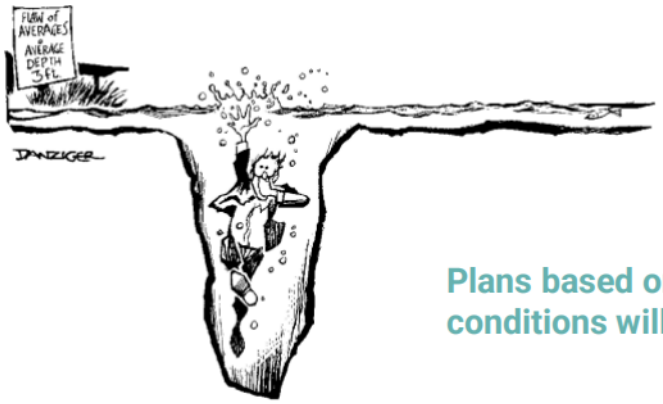
Measurement is difficult

Streetlight effect

只会关心能够测量的，不能够测量的也很重要的不被注意到

Flaw of averages

只关心平均值会有问题，以及只关心一个值会有问题



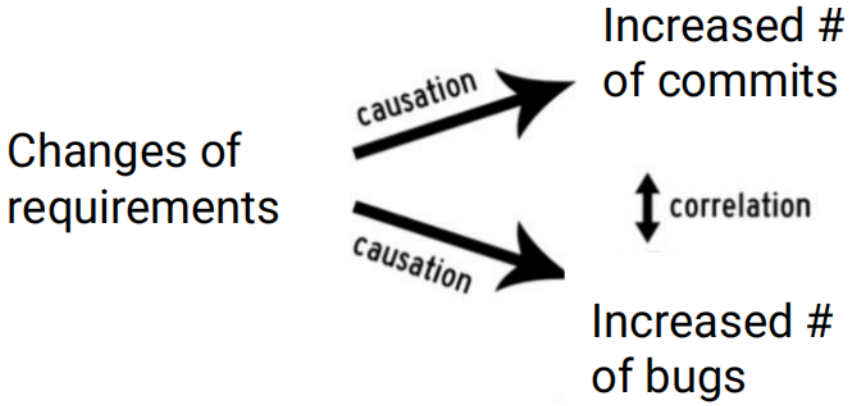
The flaw of averages is a set of systematic errors that occurs when people use single numbers (usually averages) to describe uncertain future quantities.

Plans based on the assumption that average conditions will occur are usually wrong

Survivorship bias

只关注了成功的案例，没有关注失败的案例

Correlation does not imply causation– 关联不等于因果



Simpson's paradox

Simpson's paradox is a phenomenon in probability and statistics in which a trend appears in several groups of data but disappears or reverses when the groups are combined.

在不同的环境有两个完全相反的结论

- By analyzing the programming experience and # of bugs produced for each individual developer, the company concludes that less experienced developers are better at writing high-quality code (produce less bugs)

- By analyzing the programming experience and # of bugs produced for each team, the company concludes that teams with more experienced developers are better at writing high-quality code (produce less bugs)

Measurement validity

Internal validity

focuses on the accuracy of the conclusions drawn based on a cause and effect relationship (is the Cause–Effect okay?)

External validity

Can the conclusions be generalizable?

Our findings are drawn by studying Java code. Can we say the same thing for Python?

Measurement reliability

Extend to which a measurement yields **similar results** when applied **multiple times**

Law of large numbers (大数定律)

LECTURE 9–evolution

Software evolution

Research suggests that 85—90% of organizational software costs are evolution costs.

WHY COSTLY?

- Software needs to migrate to new platforms, adjust for different machines and OS, and meet new use requirements
- As software grows, complexity grows; more changes result in poor designed structures, poor coding logic, and poor documentations
- People come and go as software evolves; costly to get newcomers familiar with the software

Legacy systems & codes

Why not replace?

TOO EXPENSIVE, TOO RISKY

- Lack of specification or documentation (lost or doesn't even exist)
- Important business rules may be implicitly embedded in software without any documentation
- Many years of maintenance degrades the system structure, making it increasingly difficult to understand or to extend
- System may be implemented using obsolete programming languages, old techniques, and adapted to older, slower hardware
- Hard to find people with required knowledge and expertise
- Data processed by the system may be out of date, inaccurate, incomplete, and depend on different database suppliers

Decisions

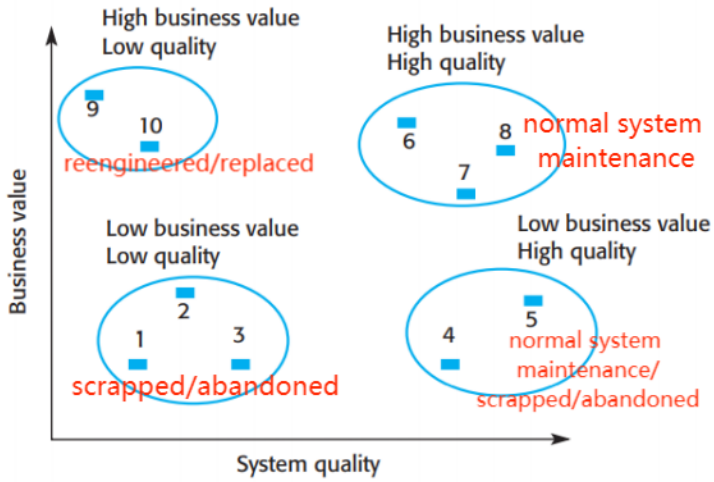
DECISIONS FOR LEGACY SYSTEM

1. **Abandon** the system completely
2. Leave the system **unchanged** and continue with **regular maintenance**
3. **Reengineer** the system to improve its maintainability
4. **Replace** part or all of the system with a new system

Which decision?

商业价值高的不能abandon，低的可以

质量好的就维持



Deprecation

What should be deprecated?

Old doesn't mean obsolete老不代表过时比如latex

Deprecation is best suited for systems/modules/code/APIs/features that are demonstrably obsolete and a replacement exists that provides comparable functionality. 过时了，而且有更好的的时候弃用旧的

How to?

Dependency discovery

- To deprecate a system, it is useful to determine:
  - Who is using the old system
  - How the system is being used
- Tools helpful for dependency discovery
  - Static analysis (e.g., which method is calling the deprecated API)
  - Logging

Warning flags

Owners of deprecated systems add compiler annotations to deprecated symbols (e.g., the **@deprecated Java annotation**)

Sunset period

Sunset period provides API consumers with an adequate time to upgrade to a newer version or retire the functionality before the API stops working.

To provide a smooth transition for customers and internal developers, some providers defines sunset period as a combination of **6 months of fully** functional and supported API and another 6 months of functional API with **no additional support**.

Migration & Testing

Using tools to **automatically update** the codebase to refer to new libraries or runtime services

Using **test suite** to automatically determine whether all references to deprecated symbols have been **removed** without breaking existing functionalities

Software maintenance

Effort distribution:

Coding errors < Design errors < Requirements errors

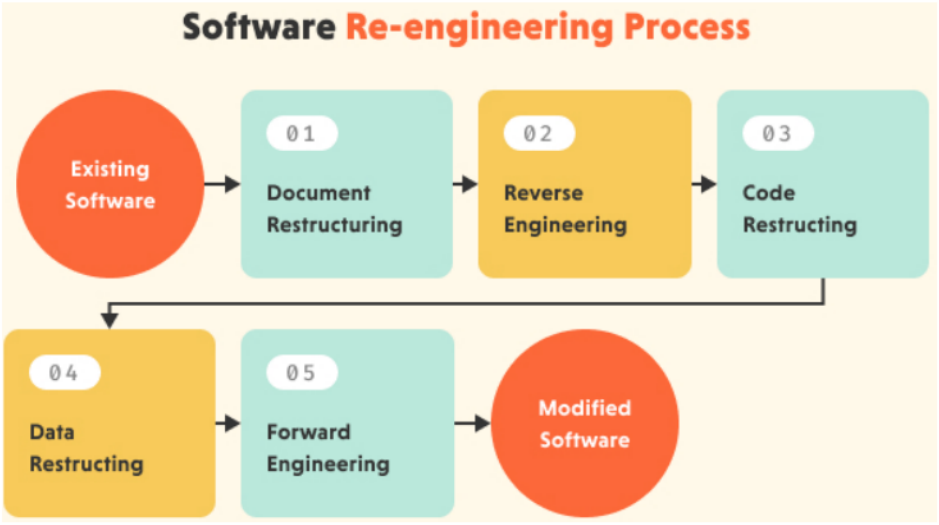
Metrics:

Number of requests for corrective maintenance

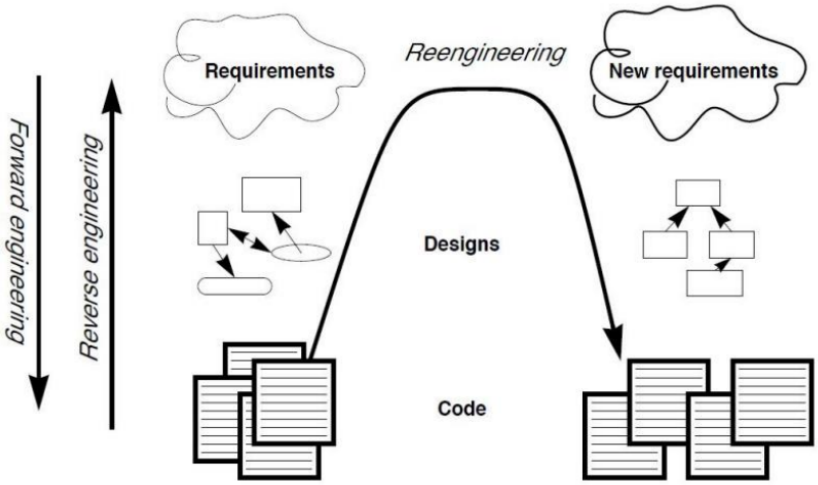
Average time required for impact analysis

Average time taken to implement a change request

Reengineering



https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/



Forward engineering:
- Requirements -> design -> code

Reverse engineering:
- Code -> design -> requirements

Reengineering
- Old code -> new code (via design transformation)

Input & Output

Input:

- The original legacy system

Output:

- An improved and restructured version of the same program
- Program documentation
- Reengineered data

3–stages

Reverse engineering

System transformation

further transformed into other representations **at the same abstraction level**.

The aim is to improve the software structure, quality, and stability.

System transformation may involve:

- **Refactoring**: restructuring the code at the level of methods and classes
- **Rearchitecting**: refactoring at the level of modules and components
- **Rewriting**: rearchitecting at the highest possible level

Forward engineering

Document restructuring

Update the documentation for the parts of the system that is currently undergoing change

Reverse engineering

representation of it at a **higher level of abstraction** than source code

Goal is to understand **how it was built**

Data restructuring

Can be a very **expensive and prolonged** process

Code restructuring/refactoring

This can be **partially automated**, but some manual intervention is usually required.

Forward engineering

Forward engineering:

- After system transformation, the transformed system representations can be used to generate physical implementations of the initial system, e.g., upgraded code and executables
- Forward engineering starts with system specification and includes the design and implementation of a new system – like an ordinary software development process.

Refactoring

What is?

Preserves the external **behavior** of the software

Improves the internal **structure** of the software

When to?

Rule of three

When you're doing something for the **third** time, start refactoring.

When adding a feature

If you have to deal with someone else's dirty code, **try to refactor it first**. Clean code is much easier to grasp.

When fixing a bug

If a bug is very hard to trace, refactor first to make the code more understandable

During a code review

What to refactor– code smells

Bloaters臃肿

Long Method –> Extract Method

Long Parameter List –> Replace Parameter With Method Call

OO Abusers

Refused Bequest拒绝遗产 (Violation of the Liskov Substitution Principle) –> Replace Inheritance With Delegation / pushdown methods/fields

Change Preventers

Shotgun Surgery (Violation of the Single Responsibility Principle) –> Extract Method

Dispensables

Duplicate code

Dead code (unused and obsolete code)

Lazy class (classes that don't do enough to earn your attention)

Couplers

Feature Envy

LECTURE 10–documentation

Software documentation

Internal Software Documentation

Administrative documentation

high–level administrative guidelines, roadmaps and product requirements

Developer documentation

- **Developer documentation**: provides instructions to developers for building the software and guides them through the development process.
  - Requirements documentation
  - Architecture & design documentation
  - Code Comments
  - API documentation
  - Readme, release notes, etc.

External Software Documentation

End–user documentation

Enterprise user documentation

used for IT staff who deploy the software across the enterprise

Just–in–time documentation

provides end users with **support documentation** at the exact time they will need it (e.g., **FAQ pages**, how–to documents)

Writing Good Documentation

Self–Documenting Code

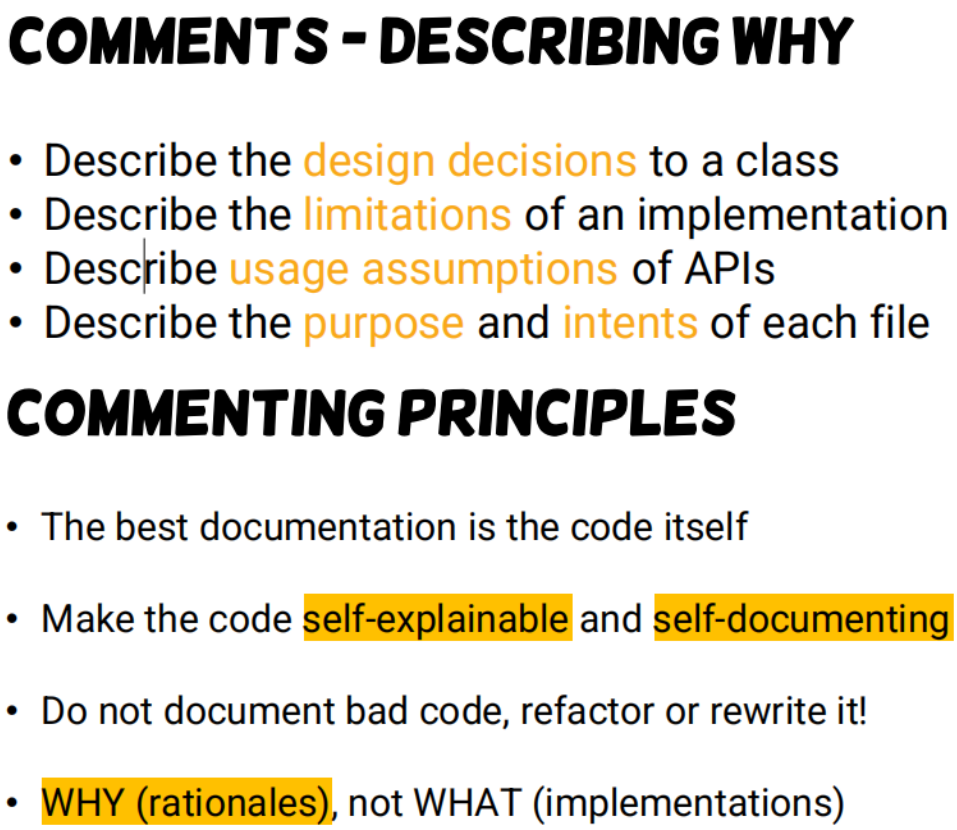
好代码一眼就看懂

Code Comments

不是写干了啥，而是写要干啥，写意图

Code comments are ignored by compilers and interpreters when producing the final executable.

However, too many or unnecessary comments reduce readability.



JAVADOC

Javadoc (included in **JDK**) automatically generates API documentation from comments present in the Java **source code**.

Javadoc style comments may contain **HTML tags** as well.

Two comment way:

1.Extra asterisk at the beginning

```
// This is a single line comment

/*
 * This is a regular multi-line comment
 */

/**
 * This is a Javadoc
 */
```

2.The standalone block tags (marked with the “@” symbol)

LECTURE 11–testing

Overview

Developer Driven Testing (DDT)

Every coded feature should be tested completely once, by the **developer** responsible for implementing it

Testing Concepts

Test case (测试用例)

A test case specifies the prerequisites, post conditions, steps, and data required for feature verification.

Test case example: Check results when a valid Login Id and Password are entered.

Test suites (测试套件、测试集)

a test suite for product purchase has multiple test cases

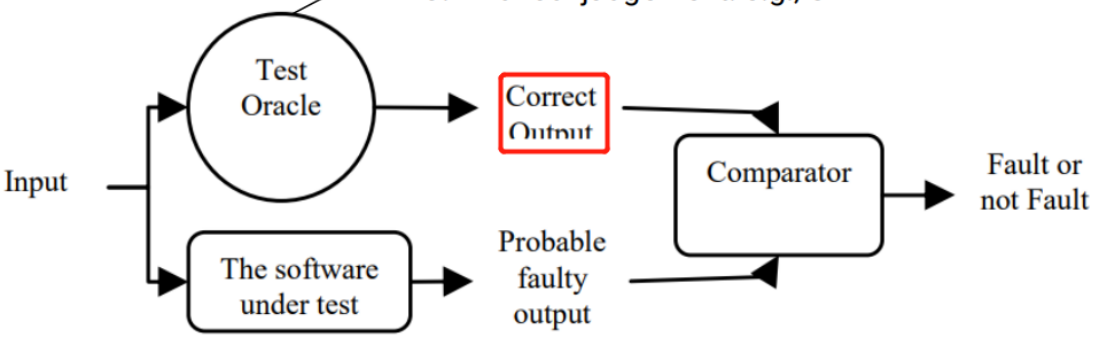
A “product purchase” test suite

- Test case 1: Login
- Test case 2: Adding products
- Test case 3: Payment
- Test case 4: Logout

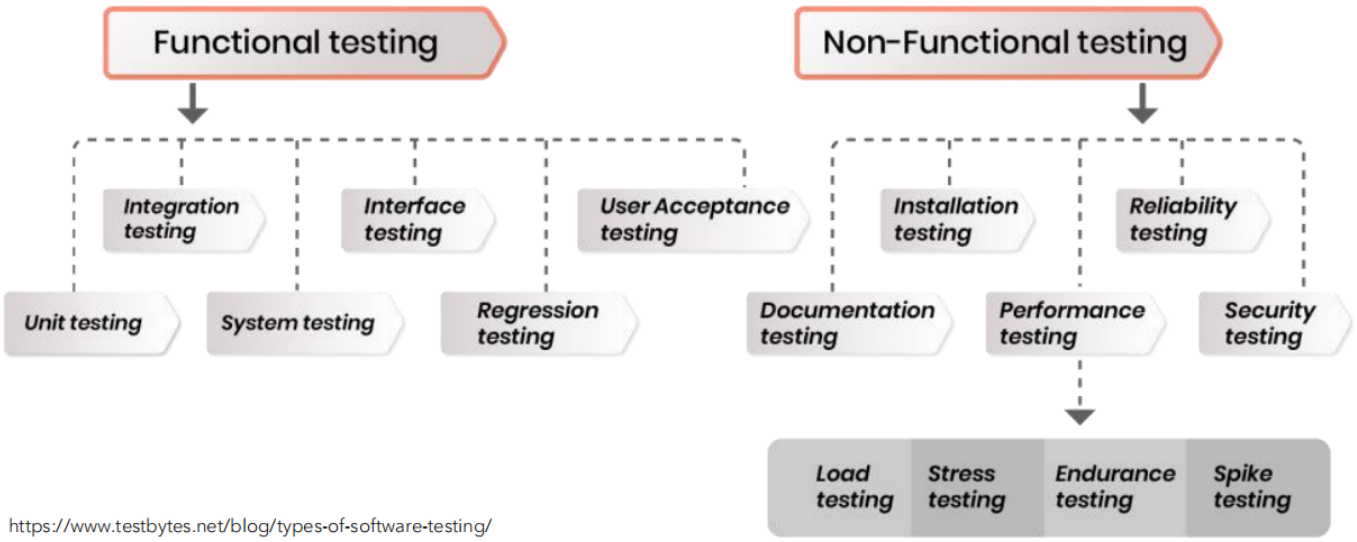
Test Oracle (测试预言、测试判断准则)

is a fault free source of expected outputs: it accepts every test input specified in software's specification and should always generate a **correct result**

1. Expected output: e.g., calculator
2. Business rules: e.g., items in a shopping cart cannot exceed 100
3. Manual judgement: e.g., UI

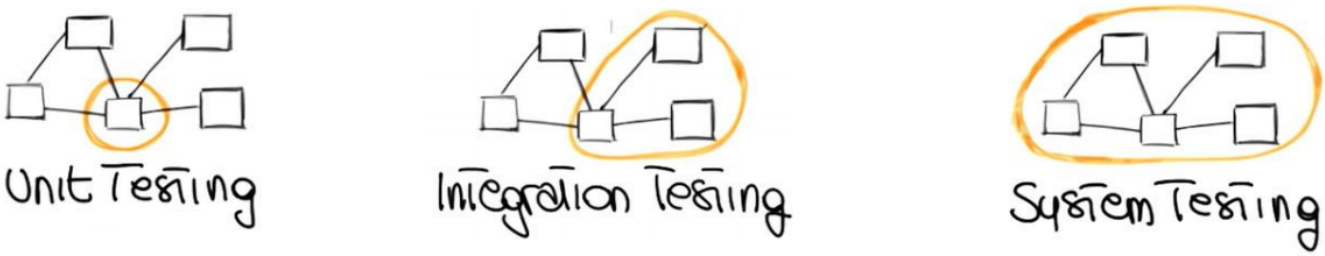


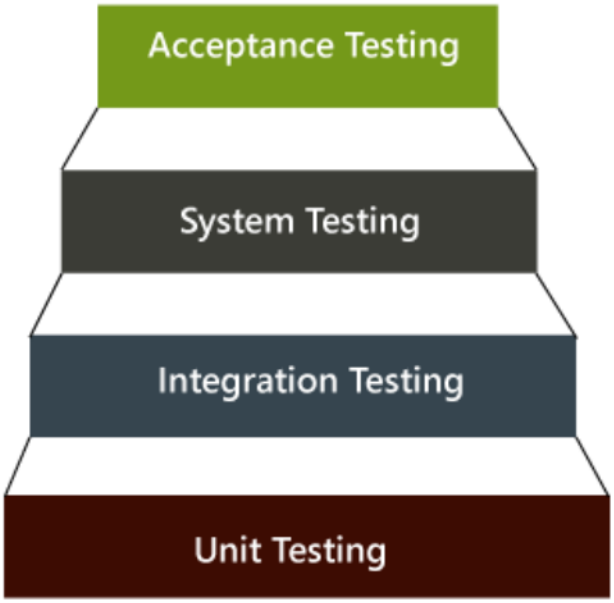
Types



https://www.testbytes.net/blog/types-of-software-testing/

Functional Testing



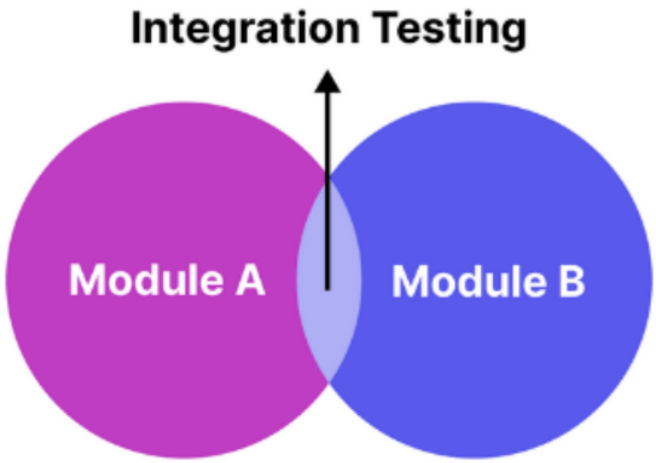


Unit testing

- Unit testing focuses verification effort on the smallest unit of software design, e.g., methods, classes
- Unit tests focus on the internal processing logic and data structures within the boundaries of a component.
- By isolating each unit and testing it independently, unit testing can be conducted in parallel for multiple components.

Integration testing

- Unit testing ensures that components work individually
- Integration testing verifies the interactions and behavior of multiple components/modules working together (though interfaces)

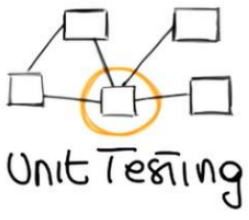


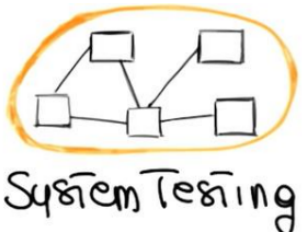
https://katalon.com/resources-center/blog/unit-testing-integration-testing

System testing

- System testing aims at testing that the product fulfills the business specifications and is ready for deployment
- System testing is generally executed after the integration test and before the delivery of the software product
- System testing is often conducted in an environment closely resemble the production environment

- System testing is usually performed by a dedicated testing team
- System testing may include functional testing, performance testing, security testing, etc.



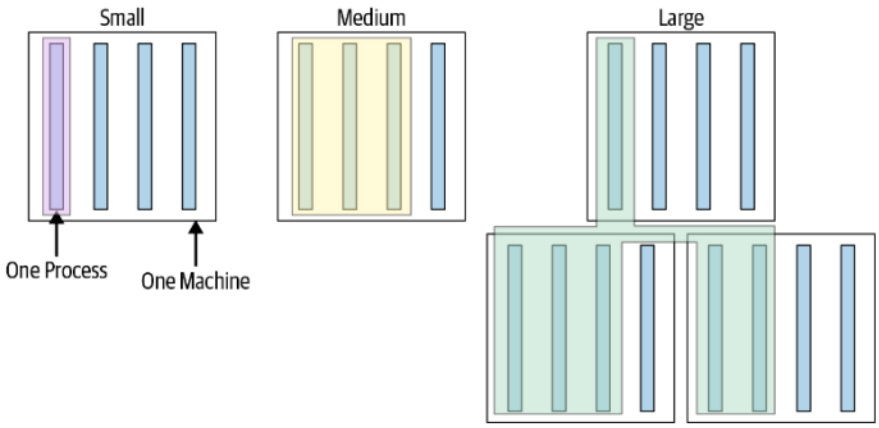


Acceptance testing

- The last phase of software testing performed after System Testing and before making the system available for actual use.
- Acceptance testing is used to determine whether the product is working for the end-users correctly
- Acceptance testing may involve end users perform realistic tasks using real data

Non–Functional Testing

Size



Small tests

- run in a single thread/process, no blocking calls (e.g., no network calls)
- provide a safe "sandbox"
- fast, effective, and reliable

Medium tests

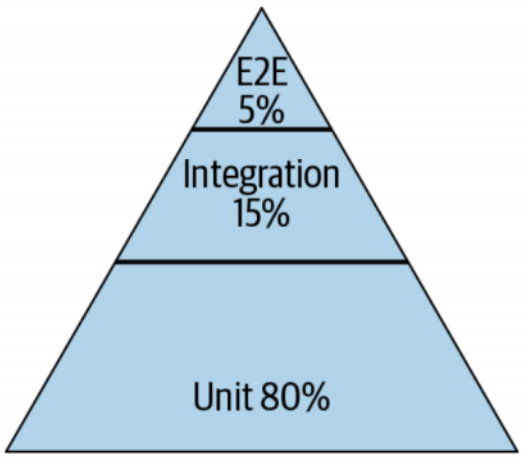
- Run on a single machine
- can span multiple processes, use threads, and can make blocking calls (e.g., network calls only to localhost, but not to remote machines via network)
- Enable testing the integration of multiple components (database, UI, server, etc.)

Large tests

- Run on a multiple machines over network
- More about validating configurations instead of piece of code
- Reserved for full system end-to-end tests

Scope

- Narrow-scoped tests
  - E.g., unit tests
  - Validate the logic in a small, focused part of the codebase (e.g., class, method)
- Medium-scoped tests
  - E.g., integration tests
  - Verify interactions between a small number of components (e.g., server + database)
- Large-scoped tests
  - E.g., end-to-end tests or system tests
  - Validate several parts of the system or the entire system



Google's test pyramid

Blackbox & Whitebox Testing

	White-box testing	Black-box testing
base	Based on **software code**	Based on **software specification**
pros	comprehensive testing, early defect detection	**simplicity**, **realistic** results (simulate end users)
cons	requires technical expertise, **expensive**, limited real-world simulation.	limited coverage, **incomplete** testing

White-box testing

Statement coverage

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

```
1 sum(int a, int b){
2     int result = a+b;
3     if(result>0)
4         print("positive" + result);
5     else if(result<0)
6         print("negative" + result);
7 }
```

Test case # 1
a = 1, b = 2

Statement coverage = 5/7 ≈ 71%

```
1 sum(int a, int b){
2     int result = a+b;
3     if(result>0)
4         print("positive" + result);
5     else if(result<0)
6         print("negative" + result);
7 }
```

Test case # 2
a = -2, b = -4

Statement coverage = 6/7 ≈ 85%

忽视部分分支

```
1 sum(int a, int b){
2     int result = a+b;
3     if(result>0)
4         print("positive" + result);
5     else if(result<0)
6         print("negative" + result);
7 }
```

Test case # 1
a = 1, b = 2

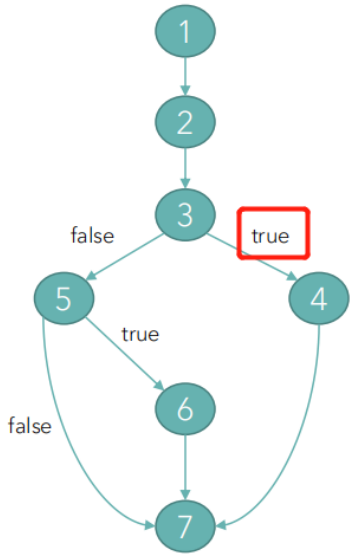
Test case # 2
a = -2, b = -4

What if the method should also print “zero” when result == 0?
Even 100% statement coverage won’t reveal this fault

Branch coverage

```
Branch coverage = ( # of executed branches / # of total branches )  x 100%
```

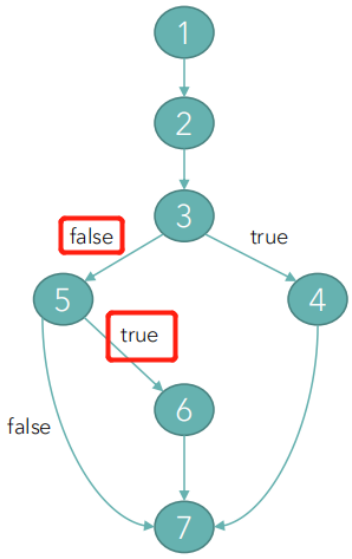
```
1 sum(int a, int b){
2     int result = a+b;
3     if(result>0)
4         print("positive" + result);
5     else if(result<0)
6         print("negative" + result);
7 }
```



Test case # 1
a = 1, b = 2

Branch coverage:
1 / 4 = 25%

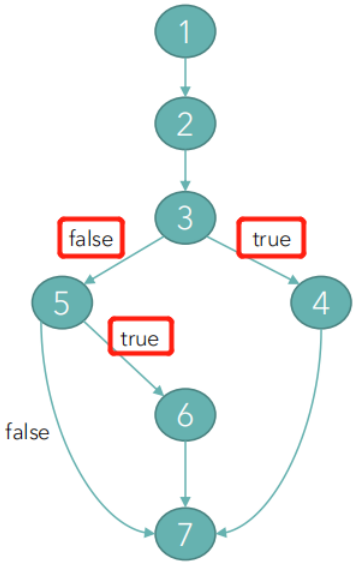
```
1 sum(int a, int b){
2     int result = a+b;
3     if(result>0)
4         print("positive" + result);
5     else if(result<0)
6         print("negative" + result);
7 }
```



Test case # 2
a = -2, b = -4

Branch coverage:
2 / 4 = 50%

```
1 sum(int a, int b){
2     int result = a+b;
3     if(result>0)
4         print("positive" + result);
5     else if(result<0)
6         print("negative" + result);
7 }
```



Test case # 1
a = 1, b = 2

Test case # 2
a = -2, b = -4

Branch coverage:
3 / 4 = 75%

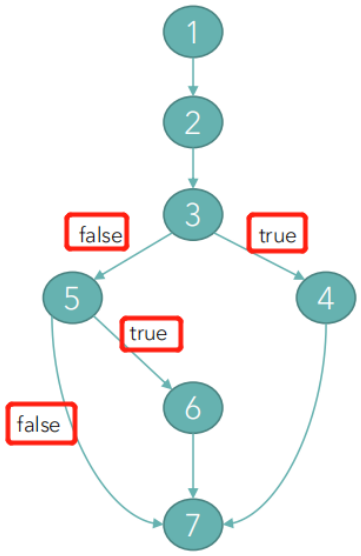
```
1  sum(int a, int b){
2      int result = a+b;
3      if(result>0)
4          print("positive" + result);
5      else if(result<0)
6          print("negative" + result);
7  }
```

Test case # 1
a = 1, b = 2

Test case # 2
a = -2, b = -4

Test case # 3
a = 0, b = 0

Branch coverage:
4 / 4 = 100%



忽视运行时错误

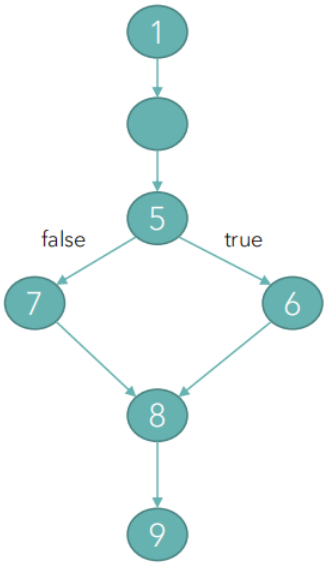
```
1  func(){
2      int x, y;
3      read(x);
4      read(y)
5      if( x == 0 || y > 0)
6          y = y/x;
7      else x = y + 2;
8      write(x);
9      write(y)
10 }
```

Yet we still miss this division-by-zero error!

Test case # 1
x = 1, y = 2

Test case # 2
x = 1, y = -2

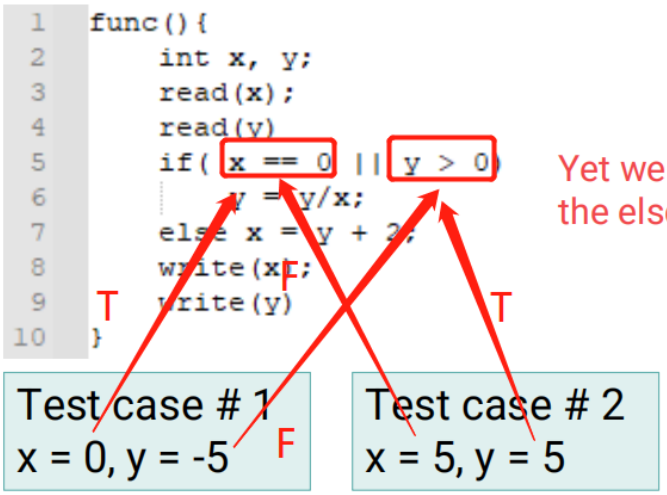
Branch coverage: 2 / 2 = 100%

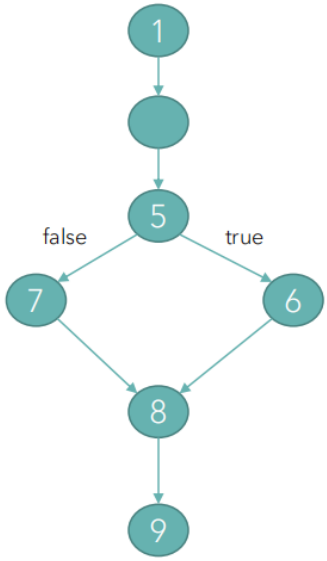


Condition coverage

```
Condition coverage = ( # of conditions that are both T and F / # total conditions )  x 100%
```

忽视else分支



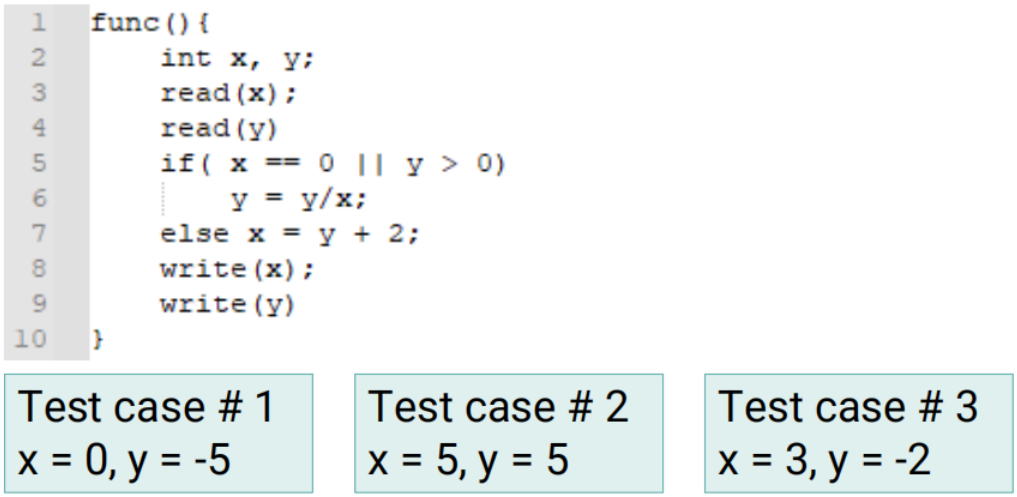


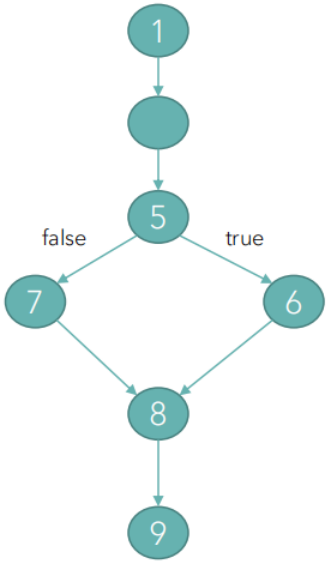
Condition coverage: 2 / 2 = 100%

有两个判断语句，所以分母是2

x==0满足过TF，y>0也满足过TF，所以分子是2

B & C coverage





Branch&Condition coverage: 100%

Black-box testing

Test Data Selection

Exhaustive Testing

Random Testing

Partition Testing

- This method divides the input data of software into different equivalence data classes (等价类).
- In general, input could be partitioned into valid/invalid equivalence class
- Test inputs can be selected from each partition to reduce time required for testing

Equivalence Partition Hypothesis

- If one condition/value in a partition passes all others will also pass.
- Likewise, if one condition in a partition fails, all other conditions in that partition will fail.

Boundary testing

Test Doubles

Test doubles are commonly used in **unit testing**

Fakes– natural, not real

- A fake is an implementation that behaves "naturally", but is not "real“
- Fakes have a pre-written implementation of the object that they are supposed to represent
- Purpose: simplify the implementation of tests by removing unnecessary or heavy dependencies, usually used for performance reasons

- Examples
  - In-memory database: You would never use this for production (since the data is not persisted), but it's perfectly adequate as a database to use in a testing environment.

```
public class DatabaseService {
    public MyEntity getEntityById(int id) {
        // go to database and fetch the entity
        // return the entity
    }
}
```

```
public class MyDatabaseService extends DatabaseService {
    private Map<Integer, MyEntity> entities = new HashMap<>();

    @Override
    public MyEntity getEntityById(int id) {
        return entities.get(id);
    }
}
```

Stubs– unnatural

- A stub is an implementation that behaves "unnaturally".
- It is preconfigured (usually by the test set-up) to respond to **specific inputs** with **specific outputs**.
- The purpose of a stub is to get your system under test **into a specific state**.

- Examples: test whether SUT notifies users when a request to XXX REST API returns 404 error
  - You could stub out the REST API with an API that **always returns 404**

```
public class FacebookService {
    public Profile getProfile(int profileId) throws Exception {
        // calls Facebook's API
        // retrieves the profile details
        // returns the profile object
    }
}
```

```
public class MyFacebookService extends FacebookService {

    public Profile getProfile(int profileId) throws Exception {
        throw new ProfileNotFoundException();
    }

}
```

Mocks– verification added

- A mock is similar to a stub, but with verification added in.
- Purpose: make assertions about how your system under test interacted with the dependency (whether a function in SUT is called in the correct way).
- We use mocks when we don't want to invoke production code or when there is no easy way to verify that intended code was executed (e.g., sending emails)

Mockito, a mocking framework for **Java**

Maintainable Unit Tests

Unchanging Tests

Ideally, after a test is written, it never needs to change unless the **requirements** of the system under test change

Only changing the system's existing behavior would require the updates to existing tests

Refactorings: refactorings don't change the systems' behaviors, existing tests should remain unaffected
New features: new features may require new tests, but existing tests should remain unaffected
Bug fixes: like new features, may require new tests, but existing tests should remain unaffected

Test Via Public Apis

Public APIs change much **less frequently** than internal implementations

Hence, tests on public APIs change less frequently (**more maintainable**)

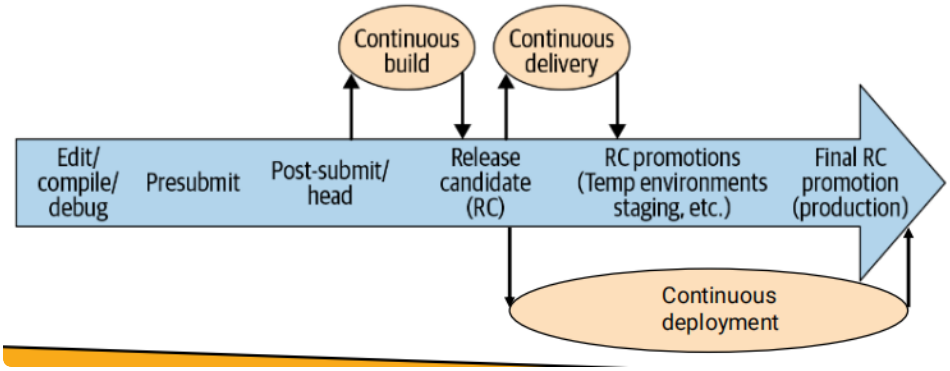
Test Behaviors, Not Methods

LECTURE 12–cicd

CI/CD

Continuous Integration

- CI, specifically, automates the build and release processes, with a Continuous Build and Continuous Delivery.
- Continuous testing is applied throughout the entire process (e.g., unit/integration/system test, etc.)



CB

- CB integrates the latest code changes at HEAD
- CB runs automated build
- CB runs automated tests
- If the code passes all tests, CB marks it passing

"Breaking the build" or "failing the build" includes breaking compilation or breaking tests

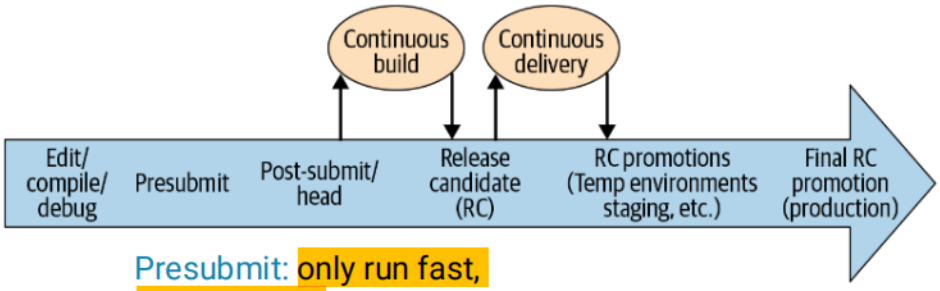
continuous testing

determining when to test what

presubmit

Case study at Google

- Each team creates a fast subset of tests, often a project's unit tests, that can be run before a change is submitted and code reviewed (i.e., presubmit)
- Empirically, a change that passes the presubmit has a very high likelihood (95%+) of passing the rest of the tests

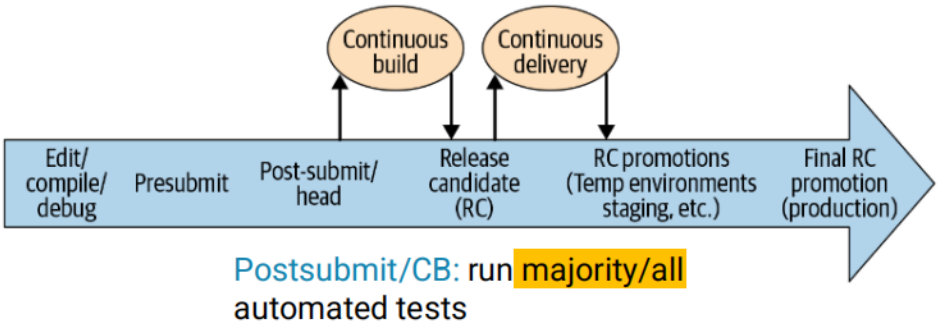


Presubmit: only run fast, reliable tests (e.g., unit tests)

postsubmit/CB

Case study at Google

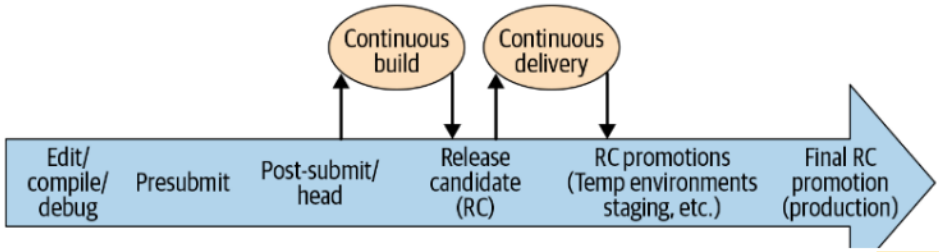
- After a change has been submitted, the Test Automation Platform (TAP) run all potentially affected tests, including larger and slower tests
- When a change causes a test to fail on TAP, engineers decide whether to fix it or roll back (TAP enables automatic rollback on high-confidence cases)



Postsubmit/CB: run majority/all automated tests

RC(release candidate) promotion/CD

- Staging (类生产环境) is a test environment that exactly resembles a production environment
- It seeks to mirror an actual production environment as closely as possible and may connect to other production services and data, such as databases and servers.



RC promotion/CD: RC is tested at each deployment on a series of test environments (e.g., sandbox, staging)

CD

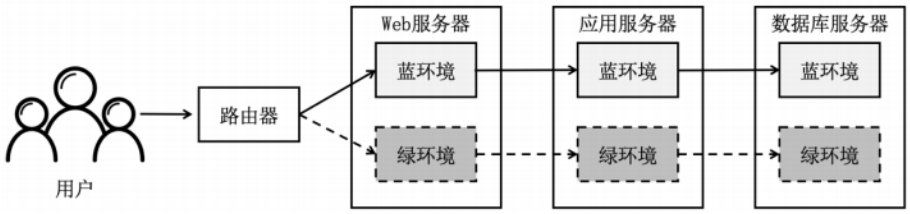
Continuous delivery automates deployment of a release to a **staging or testing environment**

Continuous deployment: automates deployment of a release all the way to the **production environment**, no human intervention

Deployment Strategy

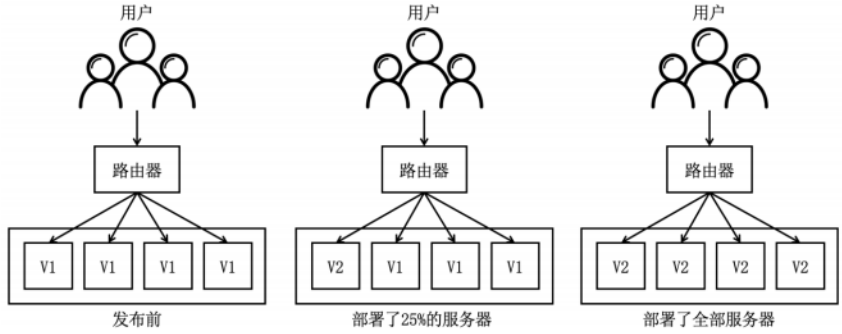
Blue–Green Deployment (蓝绿部署)

- A deployment strategy in which you create two separate, but identical environments.
- Blue environment is running the current/old application version
- Green environment is running the new application version.
- Once testing has been completed on the green environment, live application traffic is directed to the green environment.
- If error occurs, switch the traffic back to the blue environment



Canary/Greyscale Release (金丝雀发布/灰度发布)

- A deployment strategy to reduce the risk of introducing a new software version in production
- Slowly rolling out the change to a small subset of users (canary) before rolling it out to the entire infrastructure and making it available to everybody.
- Once error occurs, simply stop the traffic to the server with new version

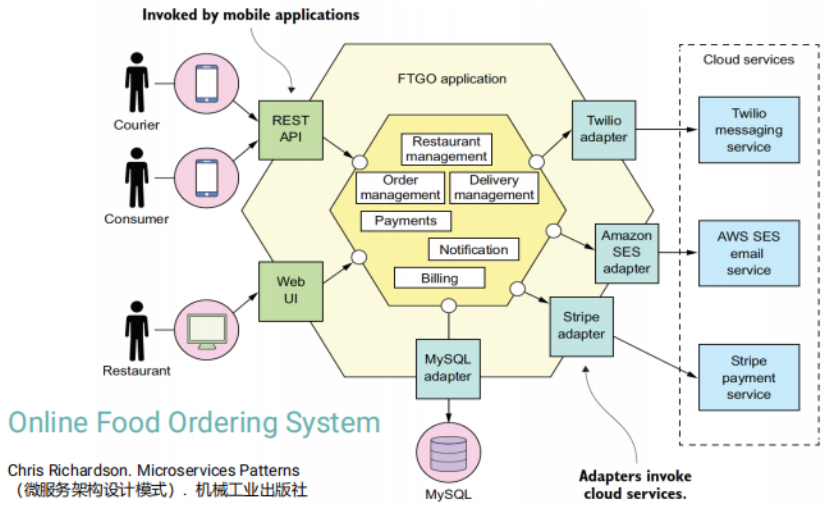


Cloud-native Applications

DevOps/Continuous Delivery

Microservices

Monolithic architecture单体架构

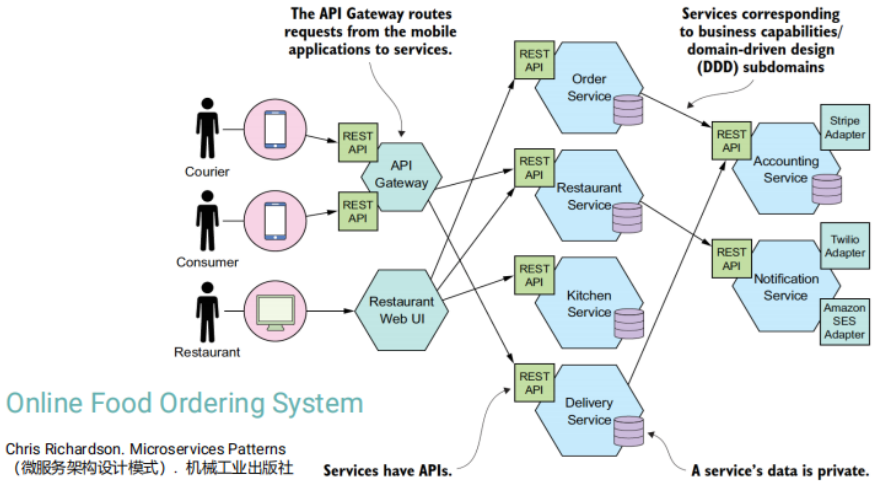


Monolithic architecture

- The whole application is packaged into a single executable
- Less flexible for large team/code base
- Difficult to scale, wasting deployment resources

Microservice Architecture微服务架构

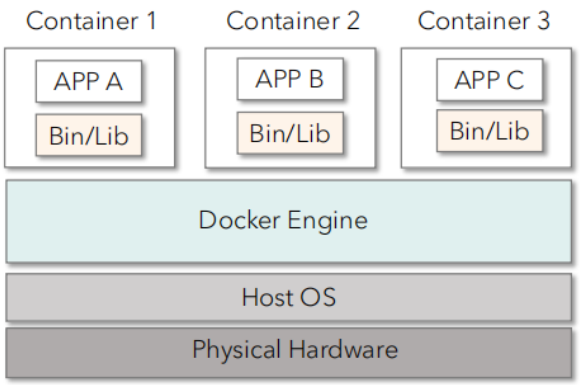
Cloud-native applications consist of multiple small, interdependent services called microservices.



Microservice Architecture

- Microservices allow a large application to be separated into smaller independent parts, with each part having its own responsibility
- Each service can be developed, managed, and deployed independently.

Containers



- A container consists of all the dependencies required to run an application, and isolates these dependencies from other containers on the same machine
- Developers can easily create, deploy, and run applications in a consistent and predictable manner across different servers or cloud platforms.

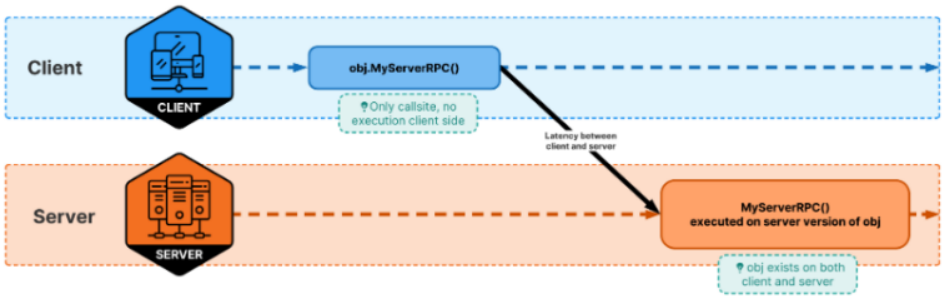
Services

REST Services

RPC Services

RPC Services

- RPC is a mechanism for invoking a function or method on a remote server, and receiving the results back.
- When calling an RPC, you call a method remotely on an object that can be anywhere in the world (i.e., RPC methods execute on a different machine)

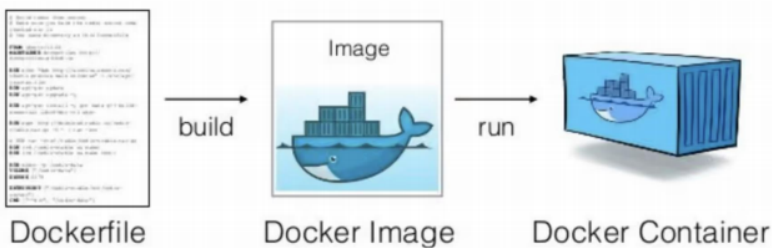


Docker

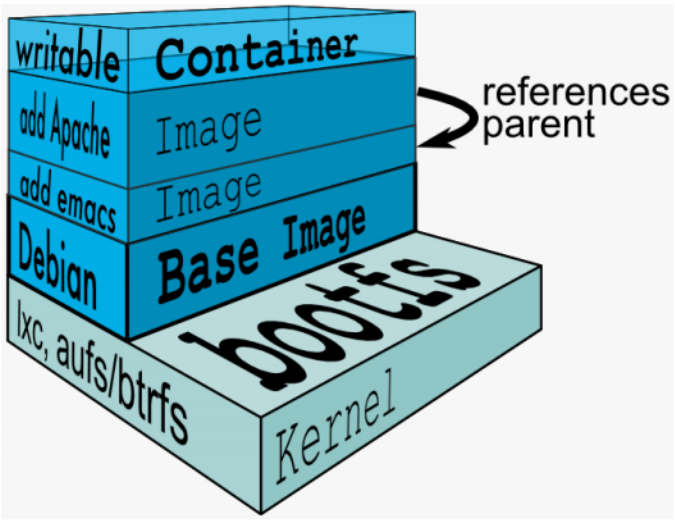
Docker is one of the most popular platforms for building, deploying, and managing containers

Docker image (镜像): a read-only template that acts as a set of instructions to create a container.

Dockerfile: a script that defines the instructions for building the image



Docker container: a running instance of a Docker image

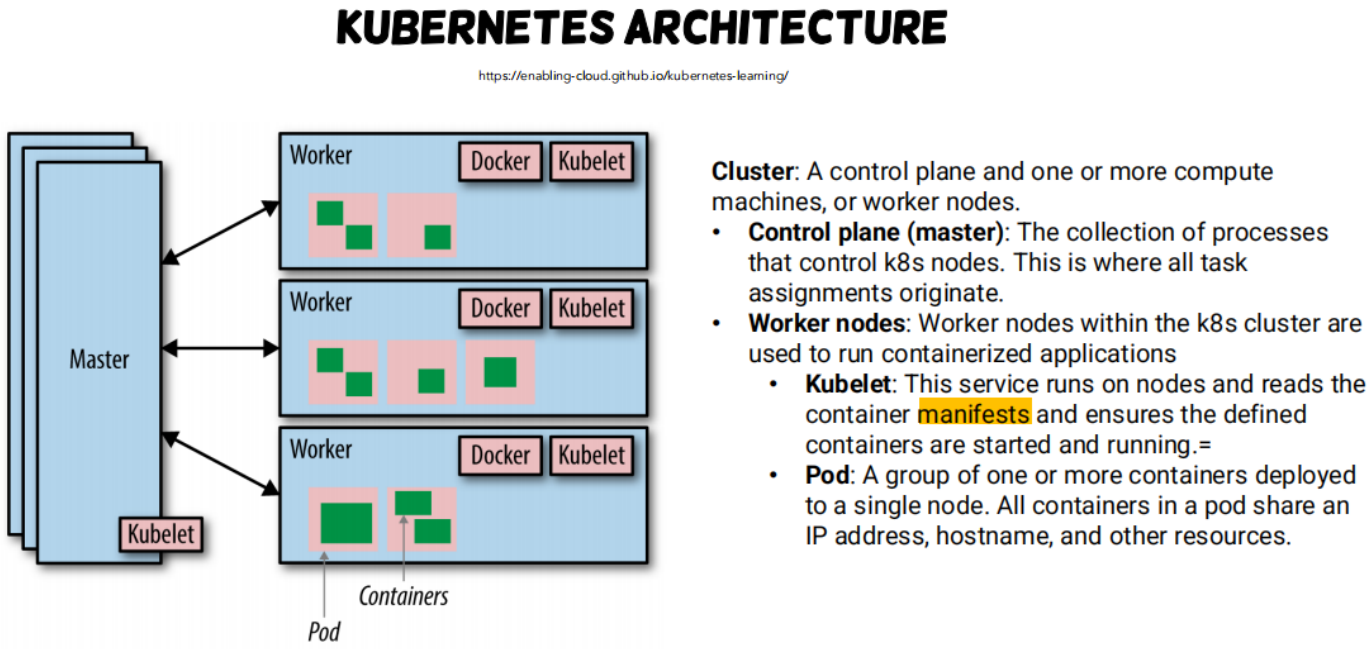


- The top layer has read-write permissions, and all the remaining layers have read-only permissions
- Several containers may share access to the same underlying level of a Docker image, but write the changes locally and uniquely to each other

Scale out

container orchestration

Docker Swarm and Kubernetes (K8s) are both container orchestration tools that allow for the management and deployment of containerized applications at scale



Cluster: A control plane and one or more compute machines, or worker nodes.

- **Control plane (master)**: The collection of processes that control k8s nodes. This is where all task assignments originate.
- **Worker nodes**: Worker nodes within the k8s cluster are used to run containerized applications
  - **Kubelet**: This service runs on nodes and reads the container manifests and ensures the defined containers are started and running.=
  - **Pod**: A group of one or more containers deployed to a single node. All containers in a pod share an IP address, hostname, and other resources.

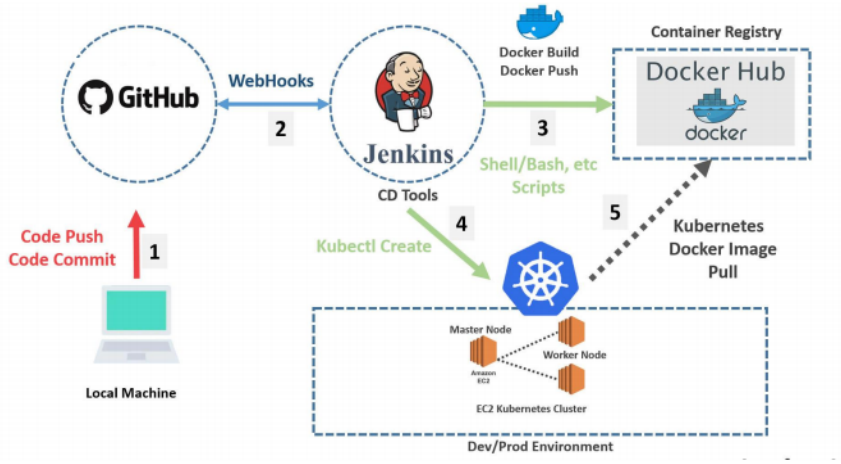
Cloud-native Open Standards

Deployment Pipelines

Jenkins

It helps automate **building, testing, and deploying** process

It facilitates **continuous integration and continuous delivery**



Step 0: Configure a webhook on GitHub, which allows external services like Jenkins to be notified when certain events like push happen.

Step 1: Dev team push local commits to GitHub

Step 2: The push triggers the webhook, and Jenkins will automatically download all the updates from GitHub

Step 3: Jenkins could be configured to automatically execute a series of tasks/stages when triggered, for example:

- Build the project with Maven
- Test the project with JUnit
- Build the docker image
- Push the docker image to Docker Hub

Step 4: Jenkins could also be configured for automatic deployment, e.g., to deploy the software to a K8s cluster

- Step 5: K8s pull docker images from the docker registry