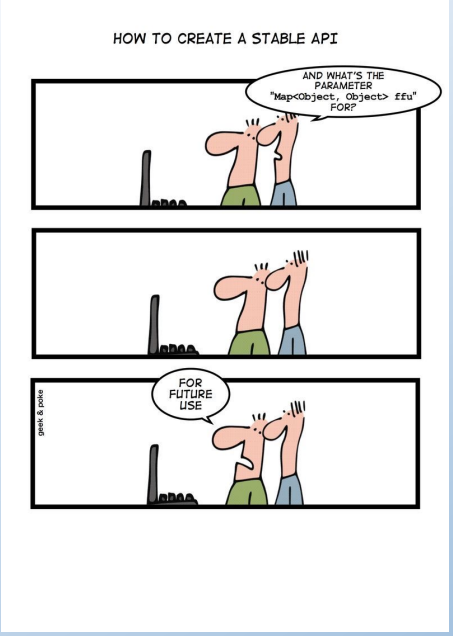
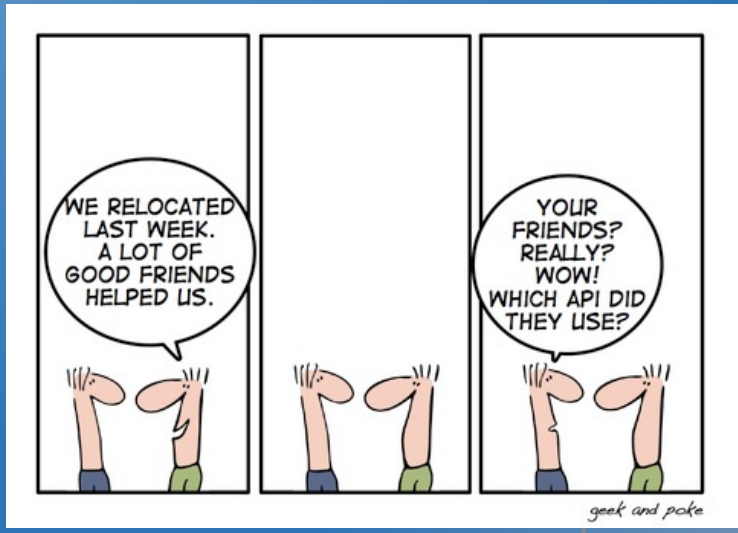


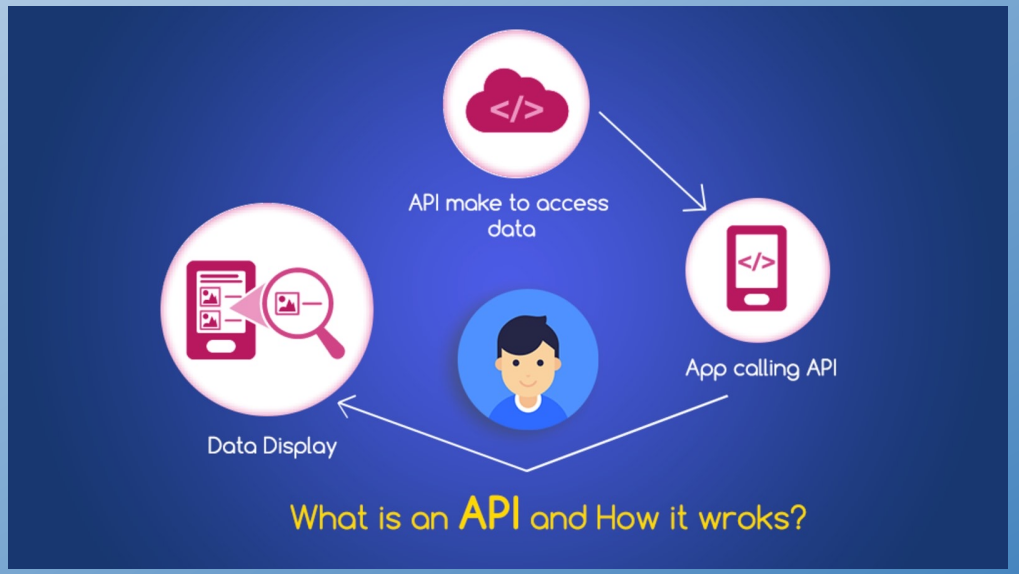
Creating and Using APIs

CSCI2340: Software Engineering of Large Systems

Steven P. Reiss



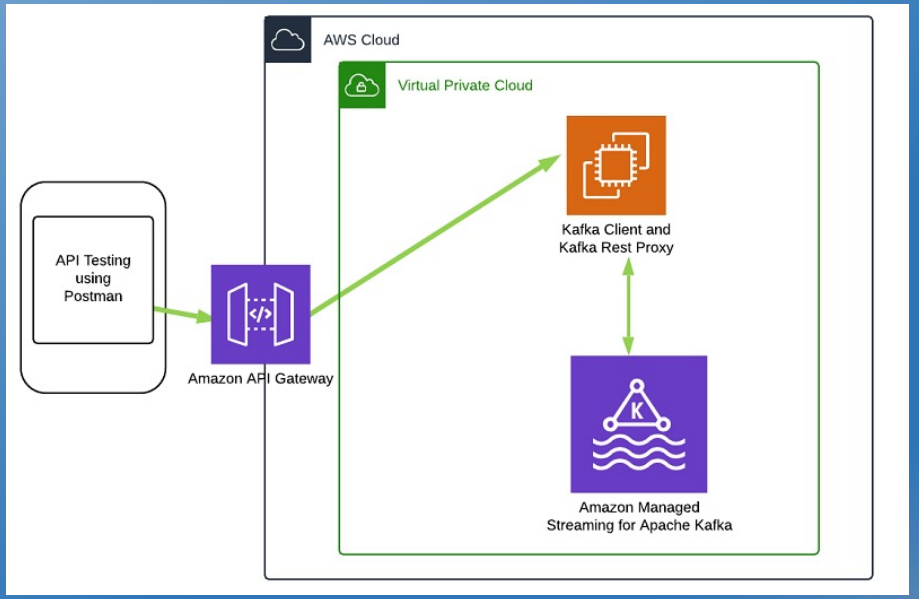
What is an API



- Application Program Interface
- Front end to a library or equivalent
  - Set of top-level calls
  - With OOP: Set of public classes and methods
  - With Modules: Set of (importable) functions and classes
  - With the web: Set of messages or HTTP requests
- The API is also the documentation on how to use these
- Designed to be shared
  - By multiple portions of the same application (different developers)
  - My multiple applications
- Designed to be reused
  - APIs are a clean way of achieving reuse

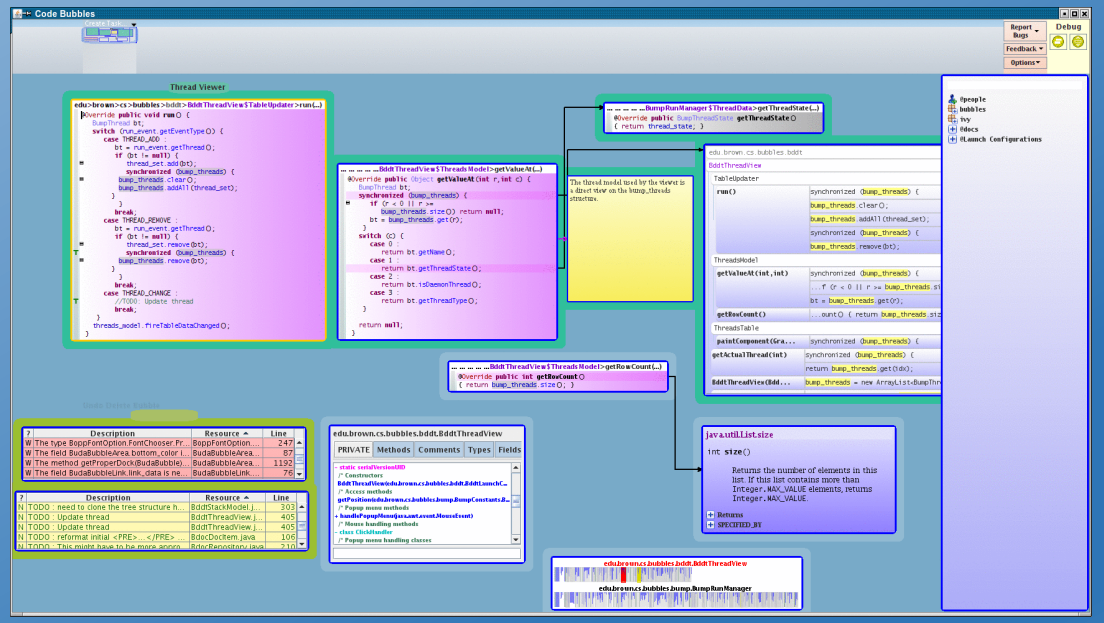
Using an API

- This is easier than creating one
  - And existing APIs will give you ideas for designing your own
- Lots of external (open source) code exists
  - Risks and benefits covered previously
  - Considerations
    - How well it fits your needs
      - Does it fit with your implementation
    - How easy is it to understand
    - Is it being maintained
    - IP, security, performance, …



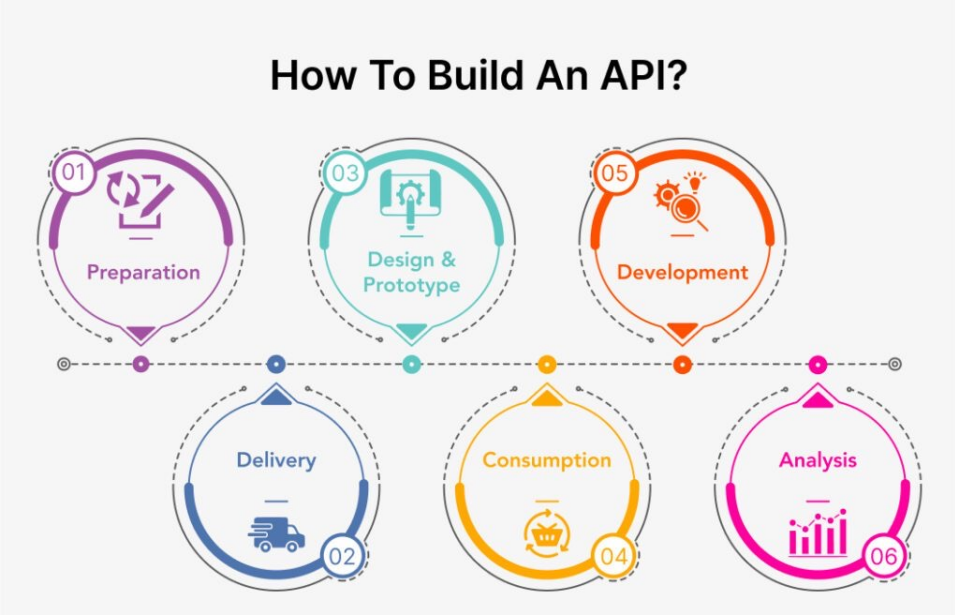
Example: External APIs used in Bubbles

- Eclipse JDT and support for abstract syntax trees
  - To use the JDT in our plugin we needed 36 jar files
- Marytts for text to speech (was freetts)
- Apache commons compression
- Gnujpdf for generating pdfs from screen images
- Joscar, Smack for chat services
- Jsoup for html scraping (documentation access)
- Json
- Jsyntaxpane for editing various forms of text files
- Jtar for reading tar files
- Junit for testing
- Websocket (node.js debugger interface)
- Commonmark for markup parsing (programmer's notebook)



Creating an API

- **You should create an API for your future development**
  - Common code you will use over and over
  - Within a project, within a company, for an individual
- Alternative to copy & paste
  - This create code clones which can be problematic
  - Provides reuse
- You should do this for your project
  - Common code across multiple developers
- You should do this for your own coding
  - Common code for single developer
- You need to understand the programming language and your needs
- But this is not as easy as it seems
  - This lecture goes over some of the complications



Experience with Creating APIs -- IVY

- Ivy has evolved over 40+ years
  - Started as BWE in the early 1980s (mostly for UI)
  - Evolved from C to handle C++, Java
  - Thinking about a Dart equivalent once we understand how it is used
- Almost everything developed in IVY has changed
  - Some changes are easy – additional calls & packages
  - Other changes are more difficult
    - Moving away from jikesbt to asm
    - Handling prefixes in XML
    - Handling templated versions of Swing classes
- More has been retired than has been maintained
  - Or will be when the one piece of software still needing it dies
  - Pieces basically no longer work (CINDER, JFLOW) – compilation only

Experience with APIs -- IVY

- Several pieces are only used by one application
  - Weren't useful for more than one (PEBBLE FSA editor)
  - Too specialized or too complex (JFLOW -> FAIT (outside IVY))
  - JANNOT – annotation processor
  - Probably remove these from API (but consider current & future uses)
- Other pieces are widely used and seem correct
  - MINT for message passing (and MINCE for C/C++)
  - SWING: a set of classes to simplify the use of Java Swing
  - FILE: a set of file and formatting utilities
  - XML: a set of XML input, output, and manipulation utilities
  - EXEC: process running and query methods
  - JCOMP: fast internal Java compiler
  - PETAL: embeddable, extensible structured graphics editor
  - STDLIB: C++ extensions and standardization (.H files)
  - JCODE – byte code interface using ASM

Experience with APIs -- IVY

- Other pieces have seen limited use (but > 1)
  - LIMBO – tracking lines between versions
  - LEASH – interface to Cocker code search
- Other pieces have become outdated
  - CINDER -- jikesbt byte code manipulation
- Other pieces are under development
  - BOWER – Express-like embeddable web server for Java

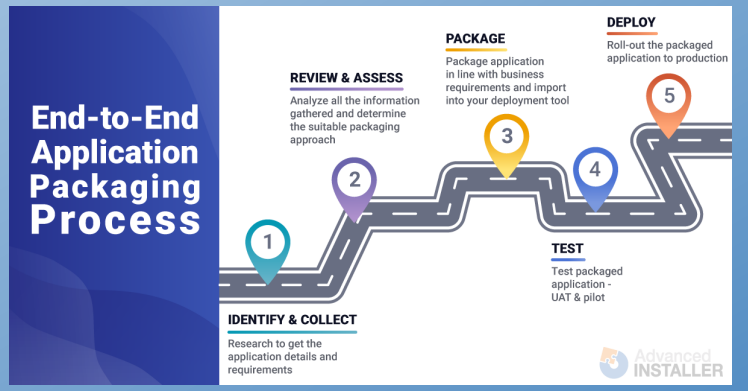
Interfaces Types Used in IVY

- Sets of static methods
  - FILE, XML
- Extensions to existing classes
  - IvyXmlWriter, SwingGridPanel, SwingNumericField, …
- Public class or façade
  - IvyExec
- Sets of related classes (façade with interfaces)
  - MINT (control, message, handler)
  - JCOMP (type, symbol, scope, …)

Single Jar Packaging

- IVY is packaged as a single jar file
  - Which requires several other libraries
  - These can be included in the jar (ivyall.jar)
  - These can be picked up from the IVY's lib directory
- Single jar packaging is good for individual use
  - Essentially provides an extended common code base
- Not ideal for outside users

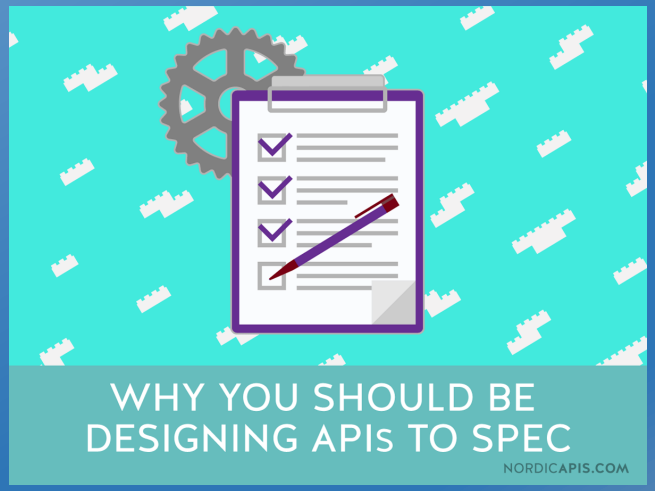
Single Package Packaging



- Outside users probably don't want all of IVY
  - Want to use a single package (e.g., MSG, JCOMP, or PETAL)
- Packaging IVY as a set of jars would make sense here
  - Users could pick and choose just what they want
- Pros and Cons
  - Saves memory
  - Avoids naming and library conflicts
  - However, these packages depend on other parts of IVY

APIs are Difficult to Design and Build

- What you need to know to build your own API / library
  - Because you should be thinking of and doing this
- Need to anticipate future uses
  - But no one knows what these are or will be
- Try to be general
  - Generalize from the special case of your application
  - This can make things easier (or more difficult)
  - But keep it simple
- Need to be easy to use
  - Easy to learn, especially if for others
  - Few dependencies (internal or external)
  - No large commitment
  - Easy to incorporate into another project

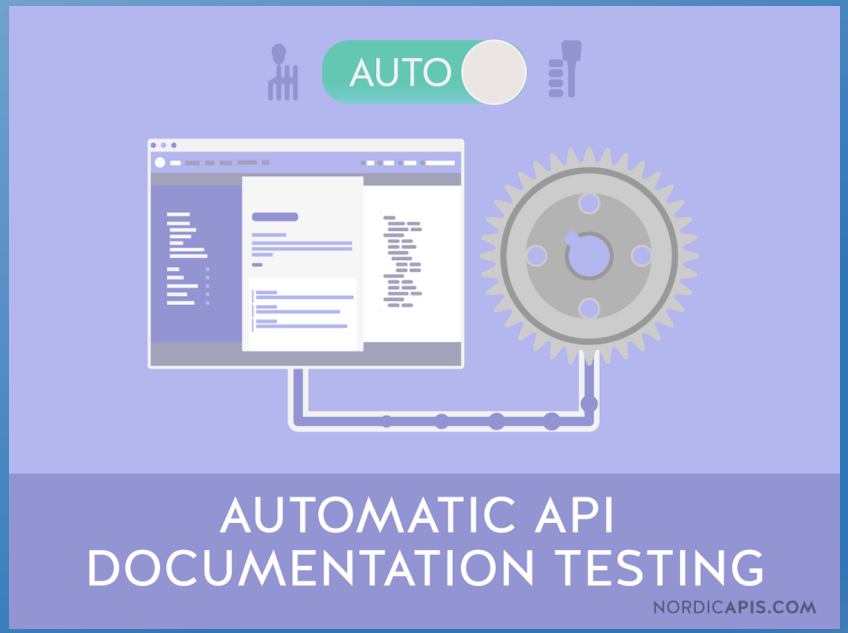


APIs are Difficult to Design and Build

- Error handling is important
  - Don’t want to be blamed for others' problems
    - Programmers trust their own code more than yours
  - The API is a black box
    - Don’t you wont want to debug it if you are working on other things
- Needs to be standalone
  - Shouldn’t depend on too many (any?) outside libraries
    - Version inconsistencies
    - Incompatibility with other libraries used in an application
    - Libraries you depend upon will change and evolve
  - Should avoid language dependence (e.g., which version of Java or C++)
  - Packageable in one or two files (.jar, .so + .H, npm module)

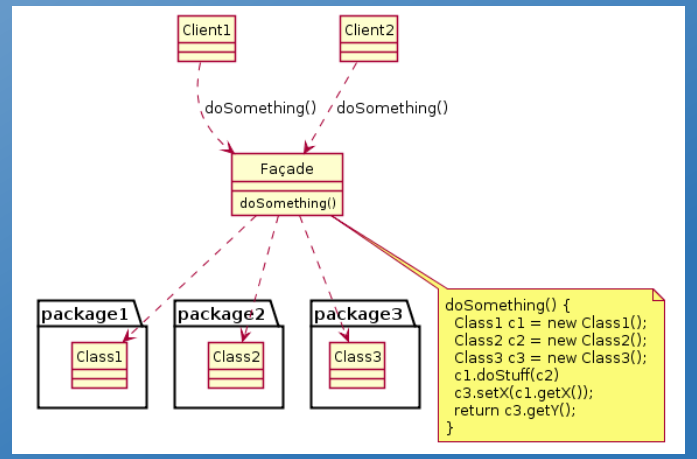
APIs are Difficult to Design and Build

- Need to be (well) documented
  - Javadoc (or equivalent) describing each public function and field
    - Expected inputs, exceptions
    - Non-trivial
  - Usage examples (php manual)
- Need to be well tested
  - Test suite just for the API
  - Keep adding to this as bugs are found
- Performance can become an issue
  - Might not be in your application
  - But will be in the next one



High Level Interfaces are a Form Of API

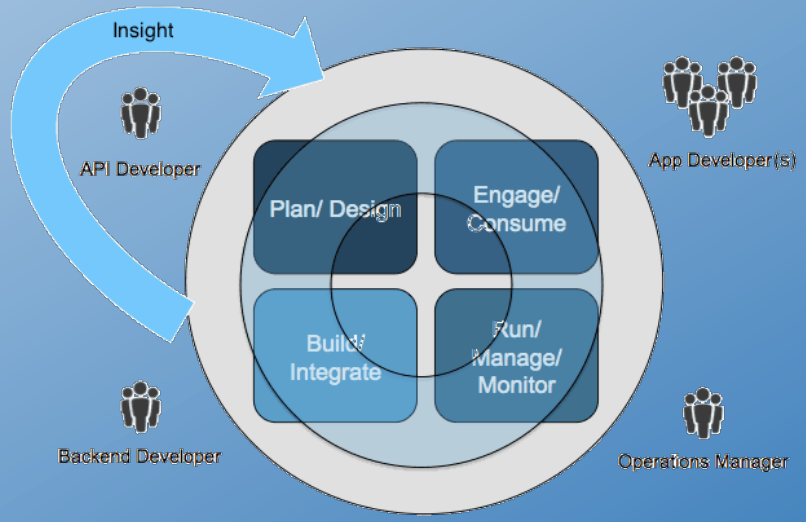
- Designed to be used by other parts of the application
- Much of what went into the design of an interface
  - Is the same as the design of an API
  - Anticipate potential uses
  - Anticipate future needs
- Except uses of a generic API are not well known
  - You have thought out your application
  - Don’t really know any others
  - Need 3-5 different applications to get the API almost right



API Development

- Identify potentially shared code
- Start with the simplest API you need first
  - What you would write for the first application
  - Routines that might be useful in other circumstances
  - If you have 2 or more initial applications, this is easier
  - Multiple people can be thought of as multiple applications
- Generalize where possible
  - Use templated classes rather than fixed components
    - Don’t use Object or dynamic if you can avoid it
  - Use templated methods to allow generic arguments
  - Leave room for expansion and evolution
    - Probably more than you normally would
  - Extend standard interfaces where appropriate
  - But keep the interface simple

API Development



- Each time you have a new application
  - Refactor the API to accommodate it
    - Add a few new calls or classes
    - Add new methods with additional parameters
    - Remove or merge unneeded methods
    - Maintain upward compatibility
  - If major restructuring is needed
    - Decide if it will help both applications (probably will)
    - Preserve the original API as well (mark as deprecated)
    - Don't be afraid to refactor early on and as needed
- After enough (5) uses
  - You're probably close to having the right API
  - Publish it

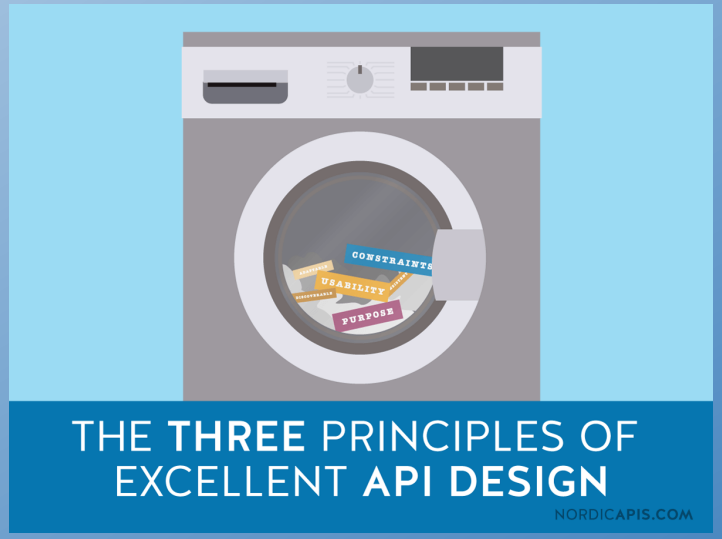
API Coding Principles

- Handle errors
  - Well defined behavior on bad or invalid inputs
    - Either in terms of unique output or exceptions
  - Code defensively: error check all inputs explicitly
    - Check for null, 0, empty collections, …
    - Explicitly document what is returned on bad inputs
  - Don’t be afraid to throw exceptions
    - Either standard exceptions or one defined explicitly for the library
    - Understand Exception versus Error
- Document as you go
  - Javadoc or other standard form
  - Include input assumptions
  - Include output under different circumstances
  - Include exceptions

API Coding Principles

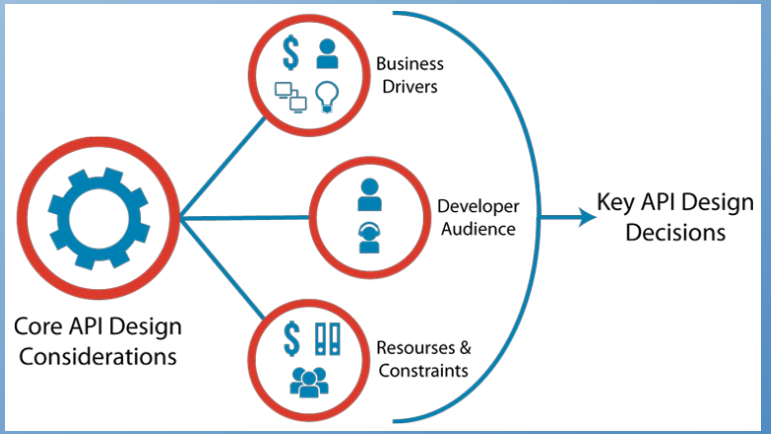
- Collections are more useful than arrays in Java
  - Dynamic, easier to create, easier to add to, more flexible
- Use enums for options, not integer or object constants
  - These are type-checked
  - Define these as part of the library
- Use interfaces if multiple implementations are possible
  - Or if you want to hide the implementation
  - With an appropriate static factory method
  - Use reflection here if needed to choose the implementation
- Define callback interfaces for events
  - Use EventListenerList
  - Add methods to add and remove callsbacks
  - Provide a default listener implementation class (or default methods on interface)

API Coding Principles



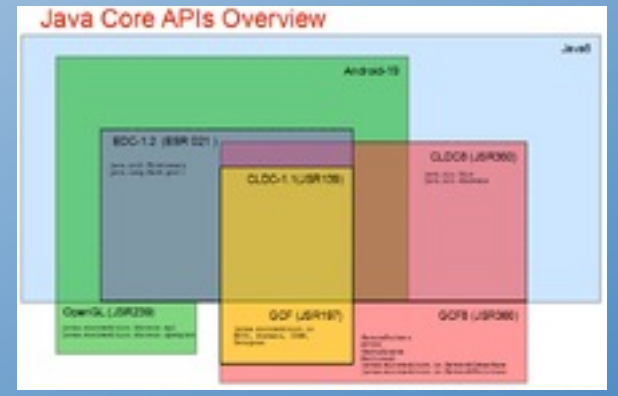
- Generalize
  - Use Collection, not List, Set, or Vector
  - Use CharSequence not String
  - Use Comparators rather than implicit comparison
    - But provide a method to use implicit comparison
  - Algorithm classes where appropriate
    - But provide a method to use the standard algorithm
- Use templated classes and methods
  - Understand templated classes
  - Learn how to define a templated method
  - Avoid using Object if it isn't multiple types at once
  - Define interfaces as needed to represent input data

API Coding Principles



- Minimize the size: keep it simple
  - Number of public classes, interfaces, methods, fields
  - But providing functionality
  - Ensure it is easy to use
- Provide debugging hooks
  - toString, output methods, logging?
  - Informative error messages or exceptions with messages
- Avoid using outside libraries that might be used separately
  - To avoid version conflicts
  - To make it easier to include in your application

API Coding Principles

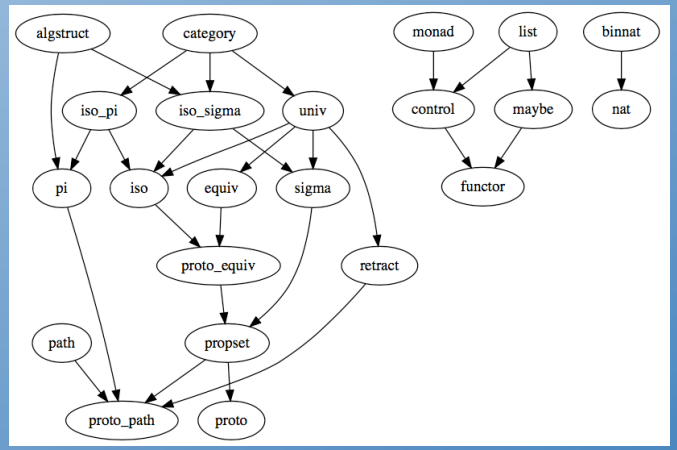


- Implementing structures or hierarchies in an API
  - Provide an abstract top-level class for common functionality
  - Only expose the subclasses if necessary
    - Defining a set of interfaces is often more appropriate
  - Provide iteration for use in the language if appropriate
  - Provide navigation (up, down, children, …)
- Define a visitor for each external data structure
  - So that users don't have to understand explicit navigation
  - Even if it isn't a tree
    - Signature visitor, UML visitor

Library Management

- Need to manage the set of APIs an application needs
  - To avoid duplication
  - To avoid version conflicts
  - To easily include all the needed libraries in the binary
- I've been doing this manually
  - Maintaining a lib directory with the explicit requirements
  - Maintaining the classpath, librarypath, or command script
- Tools have been developed for this
  - These are commonly used today
  - These depend on the language
  - And its compilation and sharing models

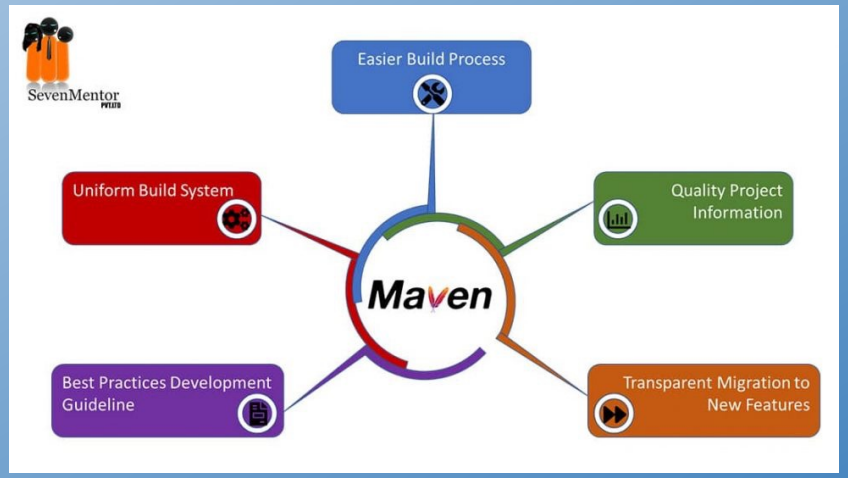
Library Management Tool Goals

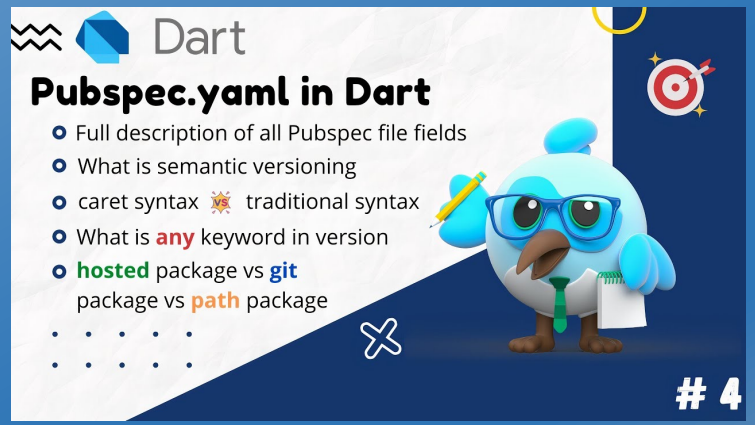


- Each API specifies its dependencies
  - Which other APIs it uses
  - Including minimum, maximum version numbers
- Application specifies its dependencies
- Tool finds the proper version to meet all the dependencies
- Versions are automatically downloaded
- Versions are automatically included when running the binary
- Tool can build a complete binary (with dependencies)
- Tool can build a directory with all the libraries

Library Management Tools

- Maven (MVN)
  - For compiled languages (Java, C/C+, …)
  - XML file (pom.xml) defining the dependencies
  - Global repository of libraries and their dependencies
  - Maven also handles building, packaging, testing, …
  - Can fail if conflicting dependencies
- Node Package Manager (NPM)
  - For Node, JavaScript, TypeScript
  - Uses json file (package.json) to define the dependencies
  - Allows multiple versions of a library (since language does)
- Dart/Flutter use YAML (pubspec.yaml)
  - To do maven-like analysis
  - And provide other capabilities ala maven
- You probably should be using one of these for your project





Other Types of APIs

- Just as there are multiple ways of representing interfaces
  - There are multiple API possibilities
- HTTP-based APIs
  - SOAP-based protocols
  - RESTful (AJAX-based) protocols
    - Google location service
  - Microservices
- Message-based APIs
  - FIELD and Code Bubbles use several of these

Task	Method	Path
Create a new customer	POST	/customers
Delete an existing customer	DELETE	/customers/{id}
Get a specific customer	GET	/customers/{id}
Search for customers	GET	/customers
Update an existing customer	PUT	/customers/{id}

Programming Homework

- Modified programming assignment due 10/24
  - Use the balls for data visualization
  - Provide an AI for your bouncing balls game
  - Experiment and report on using multiple threads
- Create a video of the assignment once it works
  - Can have sound (not mandatory)
  - Prepare to hand in (due 10/29)
  - Prepare to show in class Tuesday 10/29

Project Meeting

- Meet again as a project group
  - Think about whether you need an API to support your team
    - Code that each person would need to write separately
    - Identify any such portions and list their requirements
- Then have a project meeting