Principles of Software Construction

API Design

Claire Le Goues Bogdan Vasilescu
(Many slides originally from Josh Bloch)

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Upcoming

Midterm 2 next Thursday

- Same as last time: 24 hour period. Open everything, but don't collaborate.
- No lecture that day, you can come to lecture to work or ask us questions.
- All topics nominally in scope, but focus is on topics since Midterm 1.
- Sample questions going out today or tomorrow.

Final: nominally scheduled for Tuesday, May 3, 8:30 am.

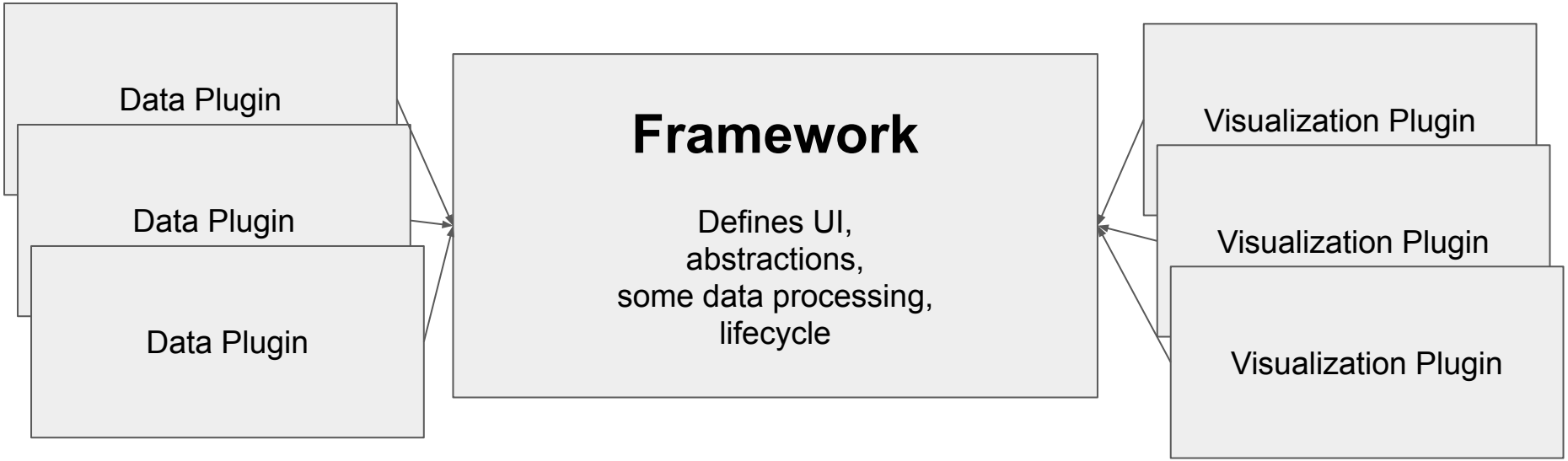
- Will be in person, proper 3-hour exam.
- You'll be able to bring notes (some number of pages).

Final homework (#6) will be released next week (possibly after midterm).

- Milestones: (1) Design framework, (2) implement framework, (3) implement plugins.
  - Note on the deadlines.
- Work in groups of 2–3. You can set your own groups, and there's a pinned post on Piazza to help if you need it. Reach out if you're stuck.

Homework 6

Data Analytics Framework



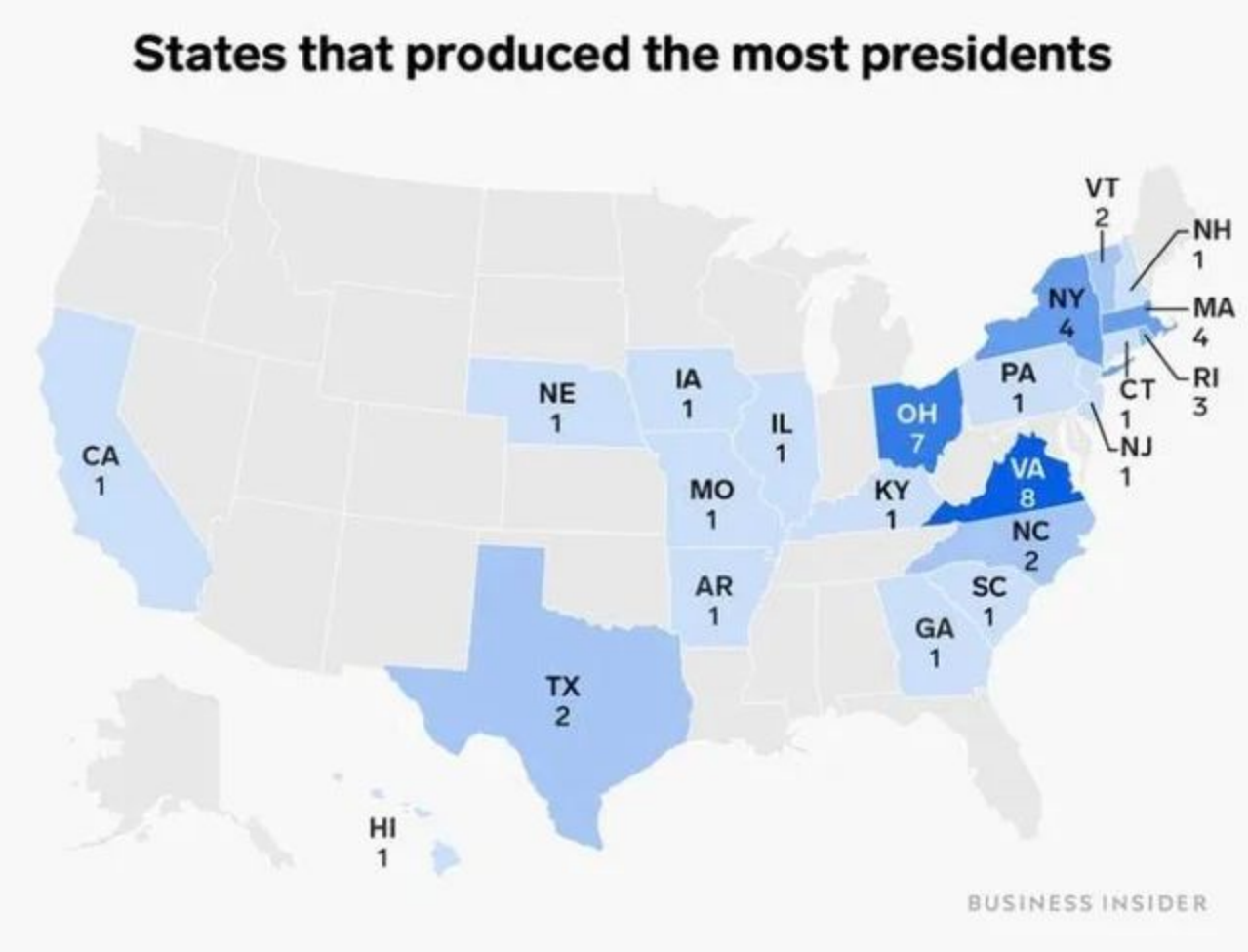
HW6: Map-Based Data Visualizations?

State, county, or country data

Data from many sources

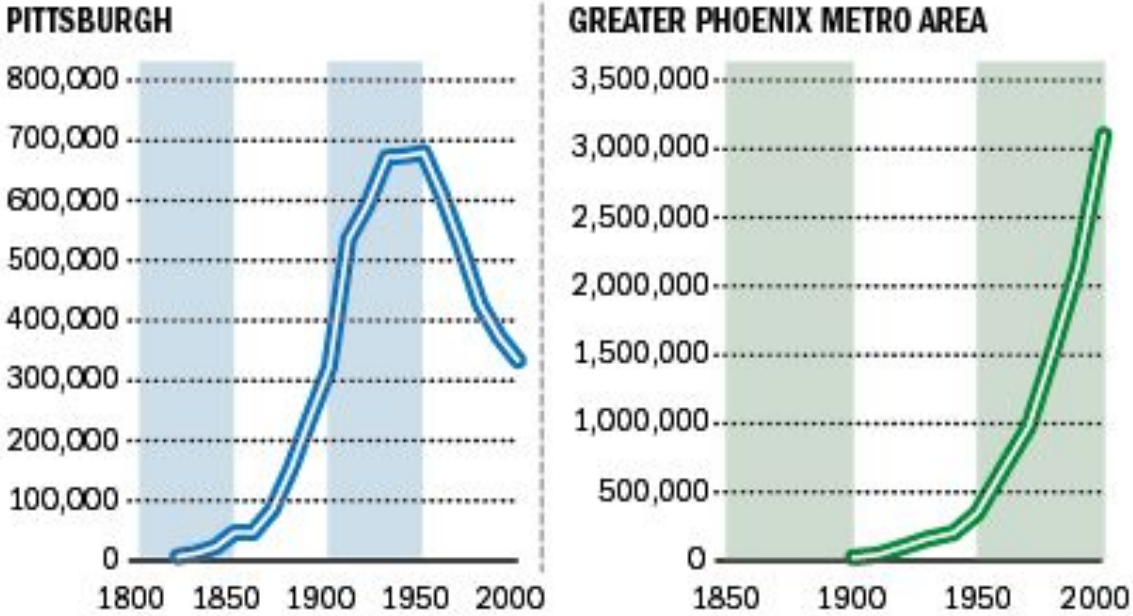
Visualization as map image, table, google maps

Animations for time series data

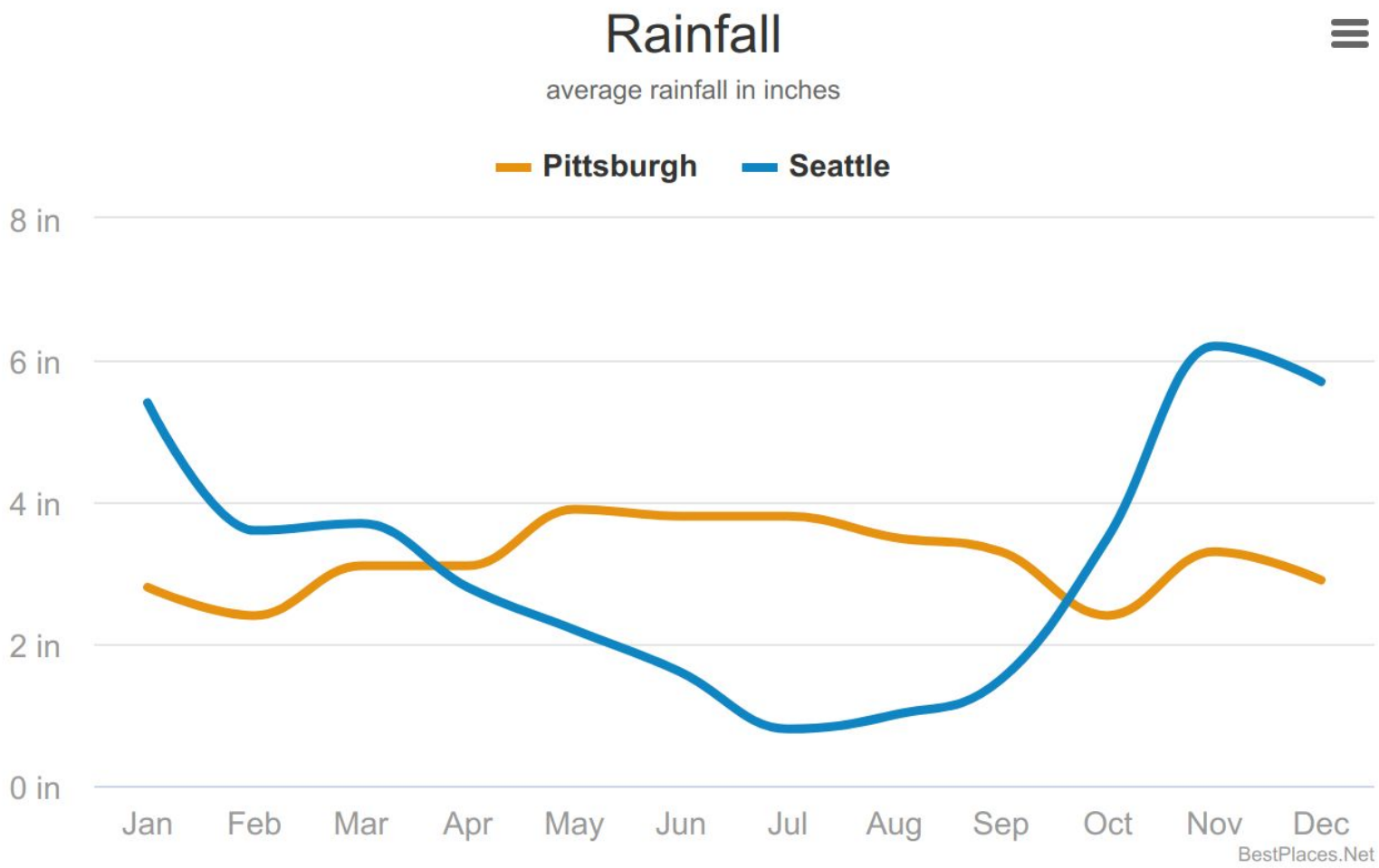


Population trends: Pittsburgh and Phoenix

Population trends in Pittsburgh and the greater Phoenix metropolitan area (roughly Maricopa County) over the past 150-200 years.



James Hilston/Post-Gazette


Rainfall
average rainfall in inches
Pittsburgh
Seattle
8 in
6 in
4 in
2 in
0 in
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
BestPlaces.Net

345

plotly | Graphing Libraries

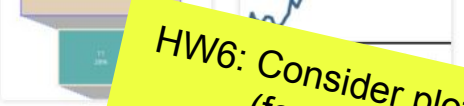
Search...

- Quick start
- Examples
  - Fundamentals
  - Basic Charts
  - Statistical Charts
  - Scientific Charts
  - Financial Charts
  - Maps
  - 3D Charts
  - Subplots
  - Chart Events
  - Animations

Waterfall Charts

Indicators

Candlestick Charts

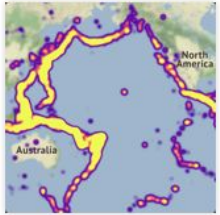


Funnel and Ch

HW6: Consider plotting libraries (for web frontends) to brainstorm ideas

Maps

Mapbox Map Layers



Mapbox Density Heatmap

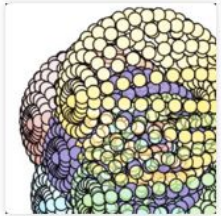
Choropleth Mapbox

Lines on Maps

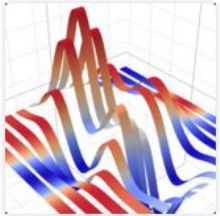
Bubble Maps

3D Charts

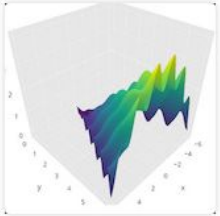
More 3D Charts »



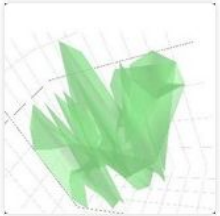
3D Scatter Plots



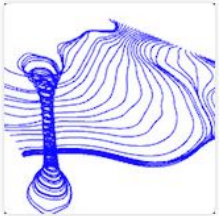
Ribbon Plots



3D Surface Plots



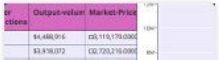
3D Mesh Plots



3D Line Plots

Subplots

More Subplots »



Libraries and **Frameworks**, continued

The use vs. reuse dilemma

- Large rich components are very useful, but rarely fit a specific need
- Small or extremely generic components often fit a specific need, but provide little benefit

“maximizing reuse minimizes use”

C. Szyperski

Domain engineering

- Understand users/customers in your domain: What might they need? What extensions are likely?
- Collect example applications before designing a framework
- Make a conscious decision what to support *(scoping)*
- e.g., the Eclipse policy:
  - Plugin interfaces are internal at first
    - Unsupported, may change
  - Public stable extension points created when there are at least two distinct customers

The cost of changing a framework

```
public class Application extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
            JPanel contentPane = new JPanel(new BorderLayou
            contentPane.setBorder(new BevelBorder(BevelBord
            JButton button = new JButton();
            if (plugin != null)
                button.setText(plugin.getButtonText());
            else
                butto
            contentPane
            textfield =
            if (plugin
                textf
            textfield.s
            this.setCon
```

```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}
```

```
public class CalcPlugin implements
    private Application application;
    public void setApplication(Application app) { this.application = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
    JOptionPane.showMessageDialog(null, "The result of "
    ut() + " is "
    on.getText())); }
    e() { return "My Great Calculator"; }
}
```

```
class CalcStarter {  public static void main(String[] args) {
        new Application(new CalcPlugin()).setVisible(true); }}
```

The cost of changing a framework

Consider adding an extra method.
Requires changes to *all* plugins!

```
public class Application extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
            JPanel contentPane = new JPanel(new BorderLayou
            contentPane.setBorder(new BevelBorder(BevelBord
            JButton button = new JButton();
            if (plugin != null)
                button.setText(plugin.getButtonText());
            else
                butto
            contentPane
            textfield =
            if (plugin
                textf
            textfield.s
            this.setCon
```

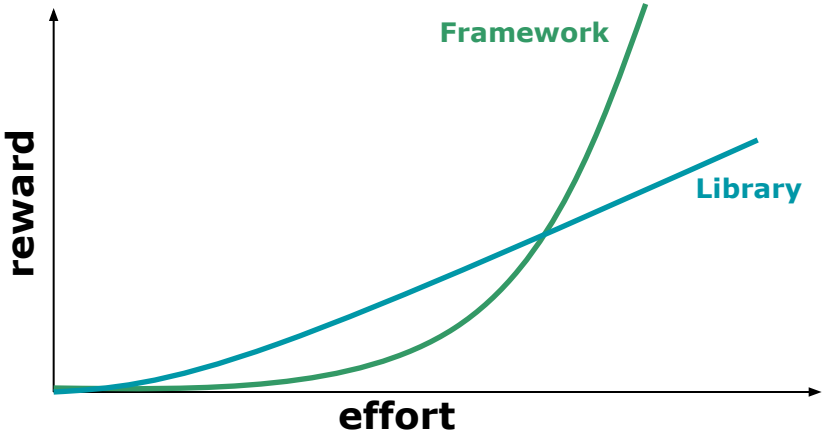
```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}
```

```
public class CalcPlugin implements
    private Application application;
    public void setApplication(Application app) { this.application = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
    JOptionPane.showMessageDialog(null, "The result of "
    ut() + " is "
    on.getText())); }
    e() { return "My Great Calculator"; }
}
```

```
class CalcStarter {  public static void main(String[] args) {
        new Application(new CalcPlugin()).setVisible(true); }}
```

Learning a framework

- Documentation
- Tutorials, wizards, and examples
- Communities, email lists and forums
- Other client applications and plugins



Typical framework design and implementation

Define your domain

- Identify potential common parts and variable parts

Design and write sample plugins/applications

Factor out & implement common parts as framework

Provide plugin interface & callback mechanisms for variable parts

- Use well-known design principles and patterns where appropriate…

Get lots of feedback, and iterate

FRAMEWORK MECHANICS

Running a framework

- Some frameworks are runnable by themselves
  - e.g. Eclipse, VSCode, IntelliJ
- Other frameworks must be extended to be run
  - MapReduce, Swing, JUnit, NanoHttpd, Express

Methods to load plugins

1. Client writes main function, creates a plugin object, and passes it to framework

   (see blackbox example above)

2. Framework has main function, client passes name of plugin as a command line argument or environment variable

   (see next slide)

3. Framework looks in a magic location

   Config files or .jar/.js files in a plugins/ directory are automatically loaded and processed

4. GUI for plugin management

   E.g., web browser extensions

An example plugin loader using Java Reflection

```
public static void main(String[] args) {
    if (args.length != 1)
        System.out.println("Plugin name not specified");
    else {
        String pluginName = args[0];
        try {
            Class<?> pluginClass = Class.forName(pluginName);
            new Application((Plugin) pluginClass.newInstance()).setVisible(true);
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + pluginName
                + ", reason: " + e);
        }
    }
}
```

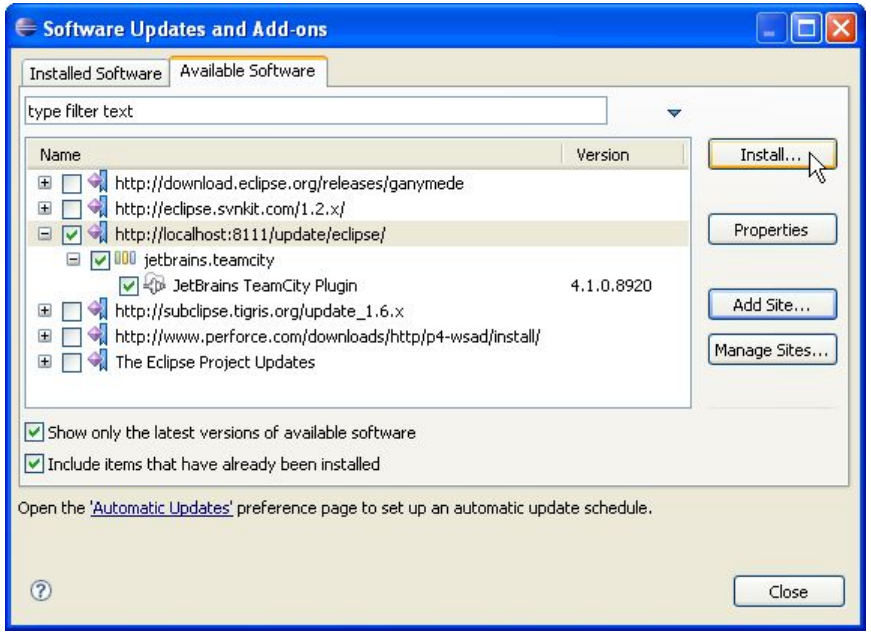
An example plugin loader in Node.js

```
const args = process.argv
if (args.length < 3)
    console.log("Plugin name not specified");
else {
    const plugin = require("plugins/"+args[2]+".js")()
    startApplication(plugin)
}
```

Another plugin loader using Java Reflection

```
public static void main(String[] args) {
    File config = new File(".config");
    BufferedReader reader = new BufferedReader(new FileReader(config));
    Application = new Application();
    Line line = null;
    while ((line = reader.readLine()) != null) {
        try {
            Class<?> pluginClass = Class.forName(line);
            application.addPlugin((Plugin) pluginClass.newInstance());
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + line
                + ", reason: " + e);
        }
    }
    reader.close();
    application.setVisible(true);
}
```

GUI-based plugin management



Supporting multiple plugins

- Observer design pattern is commonly used
- Load and initialize multiple plugins
- Plugins can register for events
- Multiple plugins can react to same events
- Different interfaces for different events possible

```
public class Application {
     private List<Plugin> plugins;
     public Application(List<Plugin> plugins) {
          this.plugins=plugins;
          for (Plugin plugin: plugins)
            plugin.setApplication(this);
     }
     public Message processMsg (Message msg) {
          for (Plugin plugin: plugins)
            msg = plugin.process(msg);
          ...
          return msg;
     }
}
```

Example: An Eclipse plugin

- A popular Java IDE
- More generally, a framework for tools that facilitate “building, deploying and managing software across the lifecycle.”
- Plugin framework based on OSGI standard
- Starting point: Manifest file
  - Plugin name
  - Activator class
  - Meta-data

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyEditor Plug-in
Bundle-SymbolicName: MyEditor;
singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator:
 myeditor.Activator
Require-Bundle:
 org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.jface.text,
 org.eclipse.ui.editors
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment:
JavaSE-1.6
```

Example: An Eclipse plugin

- plugin.xml
  - Main configuration file
  - XML format
  - Lists extension points
- Editor extension
  - extension point: org.eclipse.ui.editors
  - file extension
  - icon used in corner of editor
  - class name
  - unique id
    - refer to this editor
    - other plugins can extend with new menu items, etc.!

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

   <extension
         point="org.eclipse.ui.editors">
      <editor
               name="Sample XML Editor"
               extensions="xml"
               icon="icons/sample.gif"
contributorClass="org.eclipse.ui.texteditor.BasicText
EditorActionContributor"
            class="myeditor.editors.XMLEditor"
            id="myeditor.editors.XMLEditor">
      </editor>
   </extension>

</plugin>
```

Example: An Eclipse plugin

- At last, code!
- XMLEditor.java
  - Inherits TextEditor behavior
    - open, close, save, display, select, cut/copy/paste, search/replace, …
    - REALLY NICE not to have to implement this
    - But could have used ITextEditor interface if we wanted to
  - Extends with syntax highlighting
    - XMLDocumentProvider partitions into tags and comments
    - XMLConfiguration shows how to color partitions

```
package myeditor.editors;

import org.eclipse.ui.editors.text.TextEditor;

public class XMLEditor extends TextEditor {
	private ColorManager colorManager;

	public XMLEditor() {
		super();
		colorManager = new
			ColorManager();
		setSourceViewerConfiguration(
			new XMLConfiguration(colorManager));
		setDocumentProvider(
			new XMLDocumentProvider());
	}

	public void dispose() {
		colorManager.dispose();
		super.dispose();
	}
}
```

Example: A JUnit Plugin

```
public class SampleTest  {
      private List<String> emptyList;

      @Before
      public void setUp() {
          emptyList = new ArrayList<String>();
      }

      @After
      public void tearDown() {
          emptyList = null;
      }

      @Test
      public void testEmptyList() {
          assertEquals("Empty list should have 0 elements",
                       0, emptyList.size());
      }
}
```

Here the important plugin mechanism is Java annotations

Summary

- Reuse and variation essential
  - Libraries and frameworks
- Whitebox frameworks vs. blackbox frameworks
- Design for reuse with domain analysis
  - Find common and variable parts
  - Write client applications to find common parts
- Various mechanics.

API Design

Where we are

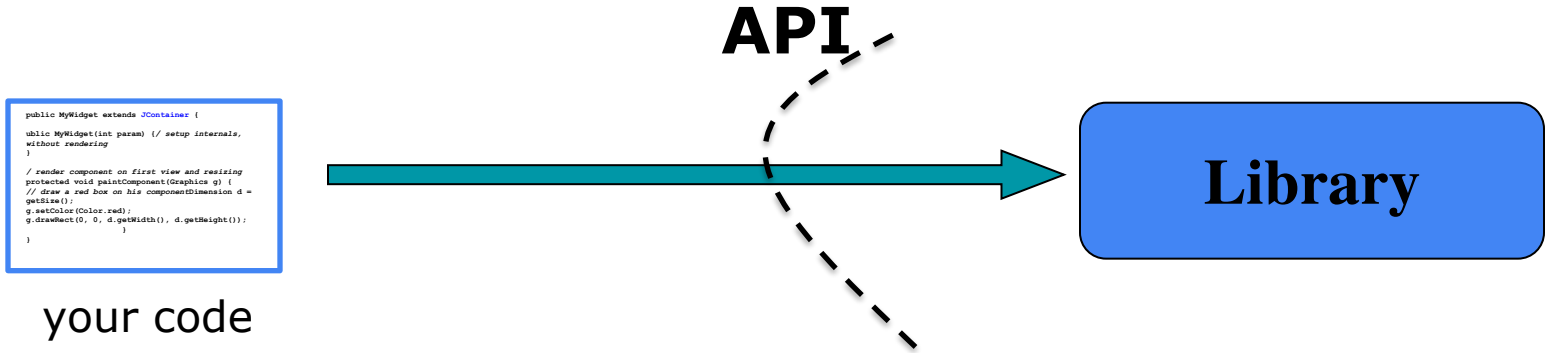
	Small scale: One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
Design for understanding change/ext. reuse robustness ...	Subtype Polymorphism Information Hiding, Contracts Immutability Types Unit Testing	Domain Analysis Inheritance & Deleg. Responsibility Assignment, Design Patterns, Antipattern Promises/Reactive P. Integration Testing	GUI vs Core **Frameworks** and **Libraries**, APIs Module systems, microservices Testing for Robustness CI, DevOps, Teams

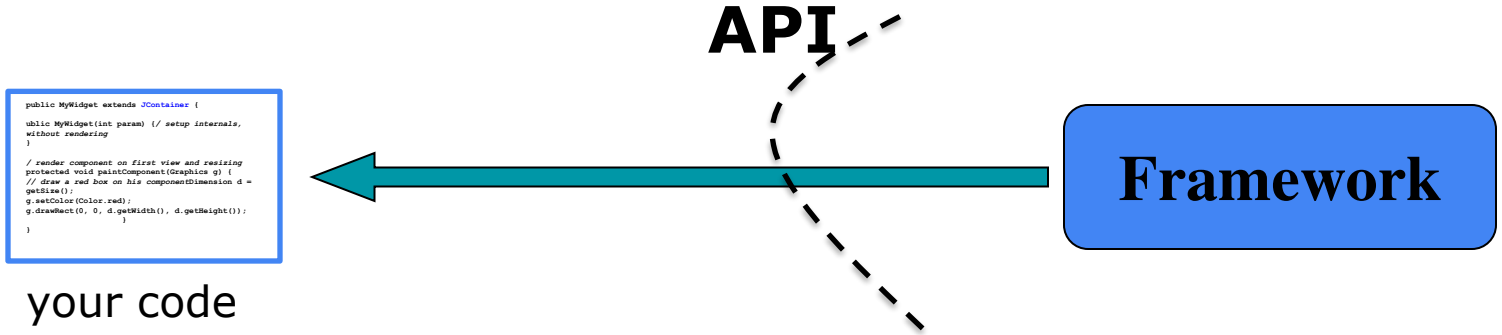
Where we are

	Small scale: One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
Design for understanding change/ext. reuse robustness ...	Subtype Polymorphism ✓ Information Hiding, Contracts ✓ Immutability ✓ Types Unit Testing ✓	Domain Analysis ✓ Inheritance & Del. ✓ Responsibility Assignment, Design Patterns, Antipattern ✓ Promises/ Reactive P. ✓ Integration Testing ✓	GUI vs Core ✓ Frameworks and Libraries ✓, **APIs** Module systems, microservices Testing for Robustness CI ✓, DevOps, Teams

Introduction to API Design

API relative to libraries, frameworks





What's an API?

- Short for Application Programming Interface
  - = Contract for a Subsystem/Library
- Component specification in terms of operations, inputs, & outputs
  - Defines a set of functionalities independent of implementation
- Allows implementation to vary without compromising clients
- Defines component boundaries in a programmatic system
- A *public* API is one designed for use by others
  - Related to Java's `public` modifier, but not identical
  - `protected` members are part of the public api

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font

All Classes

AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer

Java™ Platform, Standard Edition 7 API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

Packages

Package	Description
java.applet	Provides the classes necessary to cre context.
java.awt	Contains all of the classes for creatin
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for tr
java.awt.dnd	Drag and Drop is a direct manipulatio mechanism to transfer information be
java.awt.event	Provides interfaces and classes for d
java.awt.font	Provides classes and interface relatin
java.awt.geom	Provides the Java 2D classes for defi geometry.
java.awt.im	Provides classes and interfaces for th
java.awt.im.spi	Provides interfaces that enable the de environment.
java.awt.image	Provides classes for creating and mo
java.awt.image.renderable	Provides classes and interfaces for p
java.awt.print	Provides classes and interfaces for a

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, i a random-number generator, and a bit array).

See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* o
Deque<E>	A linear collection that supports element insertion and re
Enumeration<E>	An object that implements the Enumeration interface ge
EventListener	A tagging interface that all event listener interfaces mus
Formattable	The `Formattable` interface must be implemented by a conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to travers the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A `SortedMap` extended with navigation methods returni
NavigableSet<E>	A `SortedSet` extended with navigation methods reporti
Observer	A class can implement the `Observer` interface when it
Queue<E>	A collection designed for holding elements prior to proce
RandomAccess	Marker interface used by `List` implementations to indic
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A `Map` that further provides a *total ordering* on its keys.

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system

The `java.util.Collection<E>` interface

```
boolean      add(E e);
boolean      addAll(Collection<E> c);
boolean      remove(E e);
boolean      removeAll(Collection<E> c);
boolean      retainAll(Collection<E> c);
boolean      contains(E e);
boolean      containsAll(Collection<E> c);
void         clear();
int          size();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
```

Packages

java.applet
java.awt
java.awt.color
java.awt.datatr
java.awt.dnd
java.awt.event
java.awt.font

All Classes

AbstractAction
AbstractAnnota
AbstractAnnota
AbstractBorder
AbstractButton
AbstractCellEd
AbstractCollec
AbstractColorC
AbstractDocum
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer

Edition 7

Platform, Standard Edition.

	Description
	Provides the classes necessary to cre context.
	Contains all of the classes for creatin
	Provides classes for color spaces.
	Provides interfaces and classes for tr
java.awt.dnd	Drag and Drop is a direct manipulatio mechanism to transfer information be
java.awt.event	Provides interfaces and classes for d
java.awt.font	Provides classes and interface relatin
java.awt.geom	Provides the Java 2D classes for defi geometry.
java.awt.im	Provides classes and interfaces for th
java.awt.im.spi	Provides interfaces that enable the de environment.
java.awt.image	Provides classes for creating and mo
java.awt.image.renderable	Provides classes and interfaces for p
java.awt.print	Provides classes and interfaces for a

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, i a random-number generator, and a bit array).

See: Description

Interface Summary

Interface	Description
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* o
Deque<E>	A linear collection that supports element insertion and re
Enumeration<E>	An object that implements the Enumeration interface ge
EventListener	A tagging interface that all event listener interfaces mus
Formattable	The `Formattable` interface must be implemented by a conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to travers the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A `SortedMap` extended with navigation methods returni
NavigableSet<E>	A `SortedSet` extended with navigation methods reporti
Observer	A class can implement the `Observer` interface when it
Queue<E>	A collection designed for holding elements prior to proce
RandomAccess	Marker interface used by `List` implementations to indic
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A `Map` that further provides a *total ordering* on its keys.

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system

The `java.util.Collection<E>` interface

```
boolean      add(E e);
boolean      addAll(Collection<E> c);
boolean      remove(E e);
boolean      removeAll(Collection<E> c);
boolean      retainAll(Collection<E> c);
boolean      contains(E e);
boolean      containsAll(Collection<E> c);
void         clear();
int          size();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
```

https://developer.github.com/v3/repos/

214-s14 | 214 | 413 | Piazza | Services | more | DCKX: Directory of C

List your repositories

List repositories for the authenticated user. Note that this does not include repositories owned by organizations which the user can access. You can list user organizations and list organization repositories separately.

```
GET /user/repos
```

Parameters

Name	Type	Description
type	string	Can be one of `all`, `owner`, `public`, `private`, `member`. Default: `all`
sort	string	Can be one of `created`, `updated`, `pushed`, `full_name`. Default: `full_name`
direction	string	Can be one of `asc` or `desc`. Default: when using `full_name`: `asc`; otherwise `desc`

List user repositories

List public repositories for the specified user.

```
GET /users/:username/repos
```

Parameters

Name	Type	Description
type	string	Can be one of `all`, `owner`, `member`. Default: `owner`
sort	string	Can be one of `created`, `updated`, `pushed`, `full_name`. Default: `full_name`

Packages

java.applet
java.awt
java.awt.color
java.awt.datatr
java.awt.dnd
java.awt.event
java.awt.font

All Classes

AbstractAction
AbstractAnnota
AbstractAnnota
AbstractBorder
AbstractButton
AbstractCellEd
AbstractCollec
AbstractColorC
AbstractDocum
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer

java.awt.dnd	
java.awt.event	
java.awt.font	
java.awt.geom	
java.awt.im	
java.awt.im.spi	Provides interfaces that enable the de environment.
java.awt.image	Provides classes for creating and mo
java.awt.image.renderable	Provides classes and interfaces for pr
java.awt.print	Provides classes and interfaces for a

Queue<E>	A collection designed for holding elements prior to proce
RandomAccess	Marker interface used by `List` implementations to indic
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A `Map` that further provides a *total ordering* on its keys.

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  comp
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
        at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStat
        at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.jav
        at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
        at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
        at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
        at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

```
int          size();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

  <extension
      point="org.eclipse.ui.editors">
    <editor
        name="Sample XML Editor"
        extensions="xml"
        icon="icons/sample.gif"
      contributorClass="org.eclipse.ui.text
      editor.BasicTextEditorActionContribut
      or"
        class="myeditor.editors.XMLEditor"
        id="myeditor.editors.XMLEditor">
    </editor>
  </extension>

</plugin>
```

List user repositories

List public repositories for the specified user.

GET /users/:username/repos

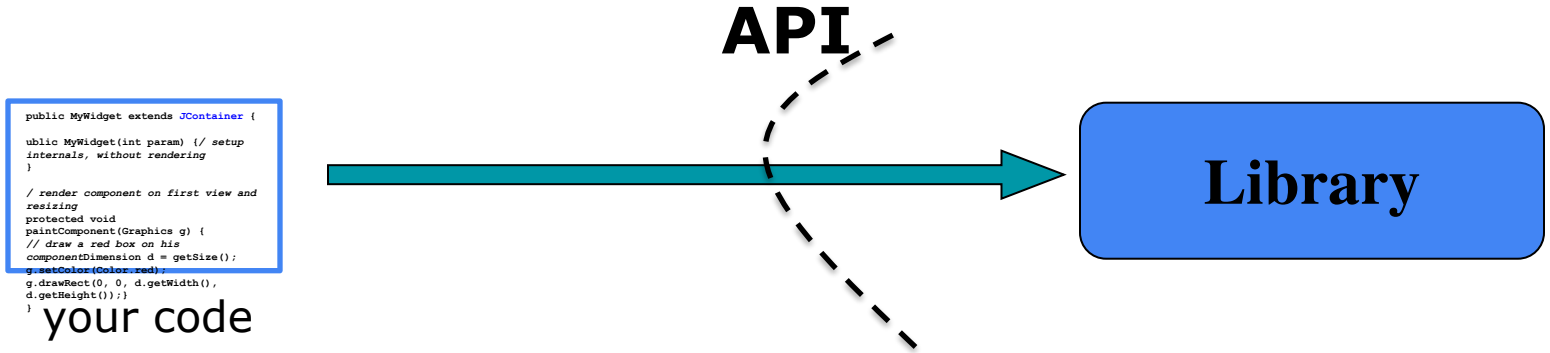
Parameters

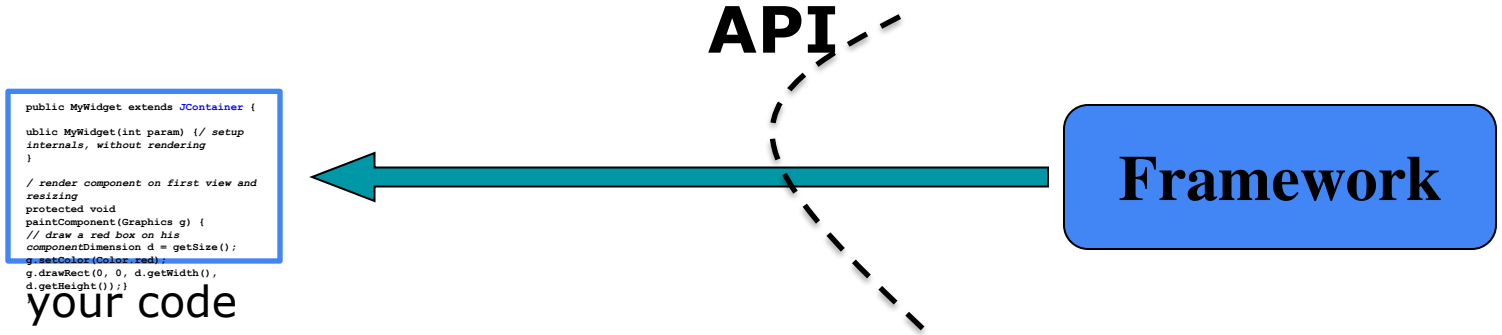
Name	Type	
type	string	Can be one of all,
sort	string	Can be one of cre

java.awt.dnd	
java.awt.event	
java.awt.font	
java.awt.geom	
java.awt.im	
java.awt.im.spi	Provides interfaces that enable the de environment.
java.awt.image	Provides classes for creating and mo
java.awt.image.renderable	Provides classes and interfaces for pr

RandomAccess	Marker interface used by List implementations to indic
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A Map that further provides a *total ordering* on its keys.

Libraries and frameworks both define APIs





Exponential growth in the power of APIs

This list is approximate and incomplete, but it tells a story

’50s-’60s – Arithmetic. Entire library was 10-20 functions!

’70s – malloc, bsearch, qsort, rnd, I/O, system calls, formatting, early databases

’80s – GUIs, desktop publishing, relational databases

’90s – Networking, multithreading

’00s – **Data structures(!)**, higher-level abstractions, Web APIs: social media, cloud infrastructure

’10s – Machine learning, IOT, pretty much everything

What the dramatic growth in APIs has done for us

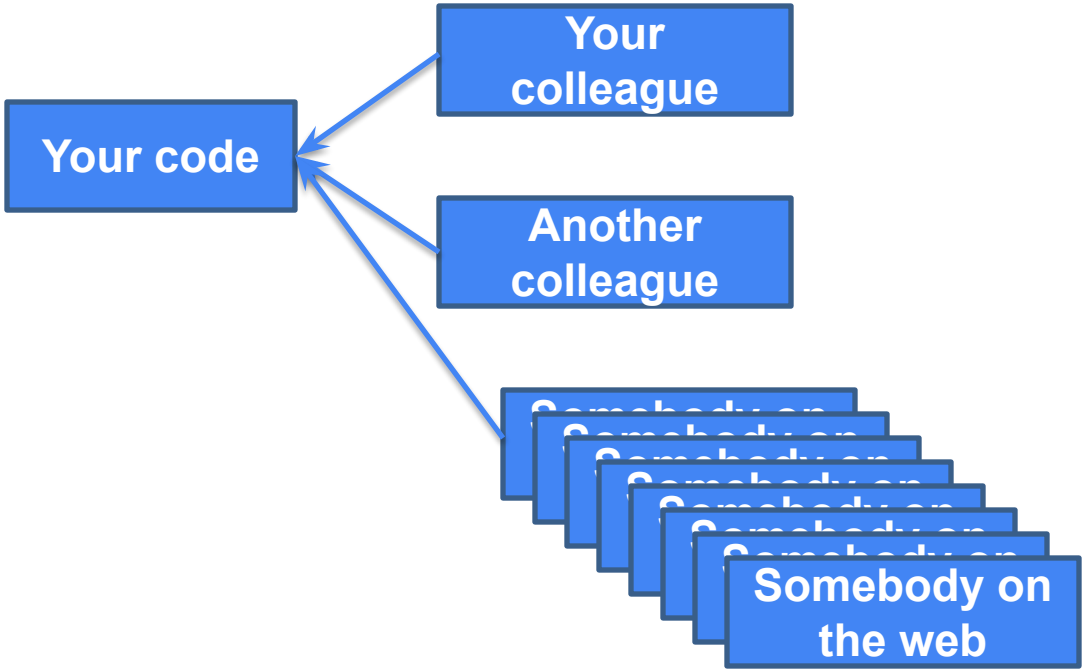
- Enabled code reuse on a grand scale
- Increased the level of abstraction dramatically
- A single programmer can quickly do things that would have taken months for a team
- What was previously impossible is now routine
- APIs have given us super-powers

Why is API design important?

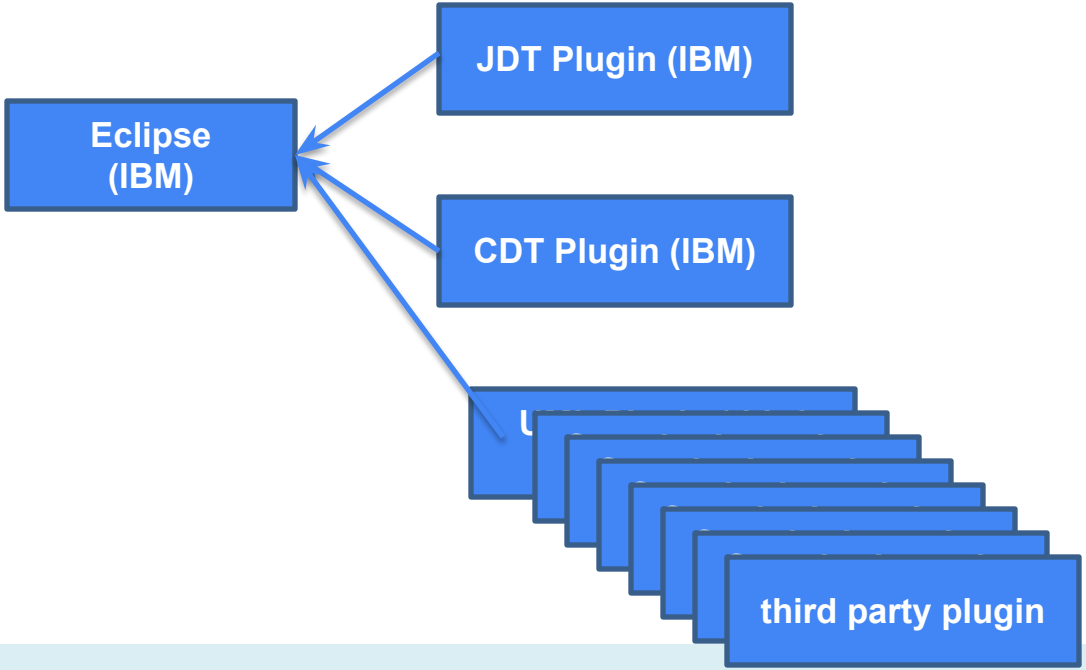
- A good API is a joy to use; a bad API is a nightmare
- APIs can be among your greatest assets
  - Users invest heavily: learning, using
  - Cost to **stop** using an API can be prohibitive
  - Successful public APIs capture users
- APIs can also be among your greatest liabilities
  - Bad API can cause unending stream of support requests
  - Can inhibit ability to move forward
- **Public APIs are forever – one chance to get it right**

Positive and Negative Experiences with APIs?

Public APIs are forever



Public APIs are forever



Evolutionary problems: Public (used) APIs are forever

- "One chance to get it right"
- Can only add features to library
- Cannot:
  - remove method from library
  - change contract in library
  - change plugin interface of framework
- Deprecation of APIs as weak workaround

enable

```
@Deprecated
public void enable()
```

Deprecated. *As of JDK version 1.1, replaced by* `setEnabled(boolean)`.

enable

```
@Deprecated
public void enable(boolean b)
```

Deprecated. *As of JDK version 1.1, replaced by* `setEnabled(boolean)`.

disable

```
@Deprecated
public void disable()
```

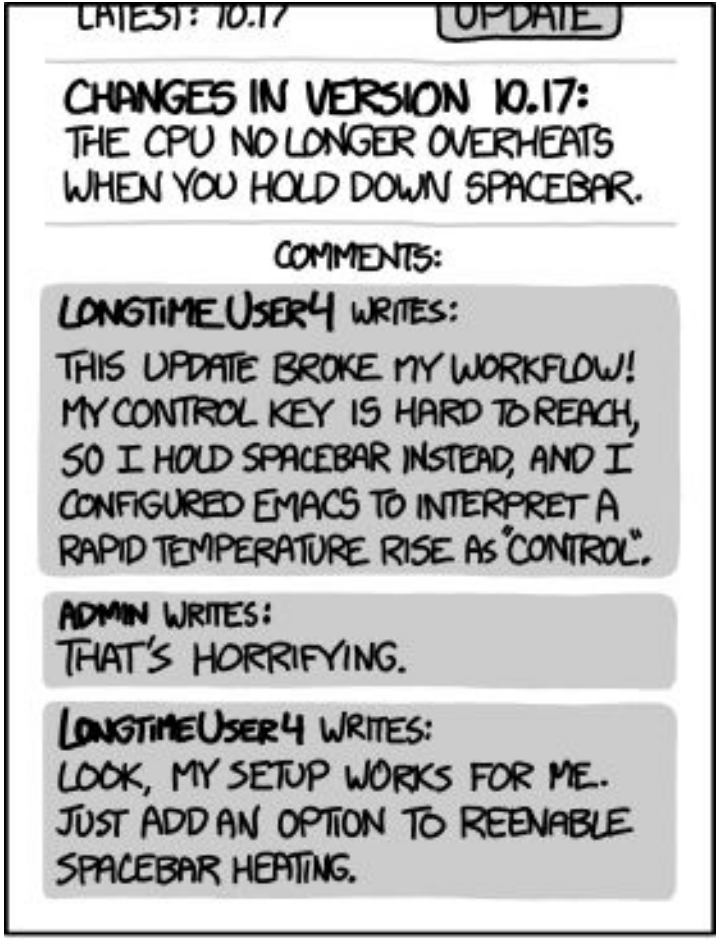
Deprecated. *As of JDK version 1.1, replaced by* `setEnabled(boolean)`.

awt.Component,
deprecated since Java 1.1
still included in 7.0

Hyrum's Law

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."

https://www.hyrumslaw.com/



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

https://xkcd.com/1172/

Why is API design important to you?

- If you program, you are an API designer
  - Good code is modular – each object/class/module has an API
- Useful modules tend to get reused
  - Once a module has users, you can’t change its API at will
- Thinking in terms of APIs improves code quality

Characteristics of a good API

Design for ...

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

The Process of API Design

An API design process

- Define the scope of the API
  - Collect use-case stories, define requirements
  - Be skeptical: Distinguish true requirements from so-called solutions, "When in doubt, leave it out."
- Draft a specification, gather feedback, revise, and repeat
  - Keep it simple, short
- Code early, code often
  - Write *client code* before you implement the API

Plan with Use Cases

- Think about how the API might be used?
  - e.g., get the current time, compute the difference between two times, get the current time in Tokyo, get next week's date using a Maya calendar, …
- What tasks should it accomplish?
- Should all the tasks be supported?
  - If in doubt, leave it out!
- How would you solve the tasks with the API?

Respect the rule of three

- Via Will Tracz, *Confessions of a Used Program Salesman*:
  Write 3 implementations of each abstract class or interface before release
  - "If you write one, it probably won't support another."
  - "If you write two, it will support more with difficulty."
  - "If you write three, it will work fine."

The process of API design – 1-slide version

Not sequential; if you discover shortcomings, iterate!

1. **Gather requirements** skeptically, including *use cases*
2. **Choose an abstraction** (model) that appears to address use cases
3. **Compose a short API sketch** for abstraction
4. **Apply API sketch to use cases** to see if it works
   - If not, go back to step 3, 2, or even 1
5. **Show API** to anyone who will look at it
6. **Write prototype** implementation of API
7. **Flesh out** the documentation & harden implementation
8. **Keep refining it** as long as you can

Requirements gathering

- Key question: what problems should this API solve?
  - ***Goals*** - Define scope of effort
- Also important: what problems shouldn't API solve?
  - ***Explicit non-goals*** - Bound effort
- Requirements can include performance, scalability
  - These factors can (but don't usually) constrain API
- Maintain a **requirements doc**
  - Helps focus effort, fight scope creep
  - Provides defense against cranks
  - Saves rationale for posterity

Start with short spec – one page is ideal!

- **At this stage, comprehensibility and agility are more important than completeness**
- Bounce spec off as many people as possible
  - Start with a small, select group and enlarge over time
  - Listen to their input and take it seriously
  - **API Design is not a solitary activity!**
- If you keep the spec short, it's easy to read, modify, or scrap it and start from scratch
- **Don't fall in love with your spec too soon!**
- Flesh it out (only) as you gain confidence in it

Sample Early API Draft

```
// A collection of elements (root of the collection hierarchy)
public interface Collection<E> {

    // Ensures that collection contains o
    boolean add(E o);

    // Removes an instance of o from collection, if present
    boolean remove(Object o);

    // Returns true iff collection contains o
    boolean contains(Object o);

    // Returns number of elements in collection
    int size();

    // Returns true if collection is empty
    boolean isEmpty();
```

Write to the API, early and often

- Start before you've implemented the API
  - Saves you from doing implementation you'll throw away
- Start before you've even specified it properly
  - Saves you from writing specs you'll throw away
- Continue writing to API as you flesh it out
  - Prevents nasty surprises right before you ship
  - If you haven't written code to it, it probably doesn't work
- Code lives on as examples, unit tests
  - **Among the most important code you'll ever write**

Then flesh out documentation so it's usable by people who didn't help you write the API

- You'll likely find more problems as you flesh out the docs
  - Fix them
- Then you'll have an artifact you can share more widely
- Do so, but be sure people know it's subject to change
- If you're lucky, you'll get bug reports & feature requests
- Use the API feedback while you can!
  - Read it all…
  - But be selective: act only on the good feedback

Maintain realistic expectations

- **Most API designs are over-constrained**
  - You won't be able to please everyone…
  - So aim to displease everyone equally*
  - But maintain a unified, coherent, simple design!

- **Expect to make mistakes**
  - A few years of real-world use will flush them out
  - Expect to evolve API

* Well, not equally – I said that back in 2004 because I thought it sounded funny, and it stuck; actually you should decide which uses are most important and favor them.

Information Hiding

Which one do you prefer?

```
public class Point {
      public double x;
      public double y;
}
// vs.
public class Point {
      private double x;
      private double y;
      public double getX() { /* … */ }
      public double getY() { /* … */ }
}
```

Information hiding also for APIs

- Make classes, members as private as possible
  - You can add features, but never remove or change the behavioral contract for an existing feature
- Public classes should have no public fields (with the exception of constants)
- Minimize *coupling*
  - Allows modules to be, understood, used, built, tested, debugged, and optimized independently

Key design principle: Information hiding

- "When in doubt, leave it out."

- Implementation details in APIs are harmful
  - Confuse users
  - Inhibit freedom to change implementation

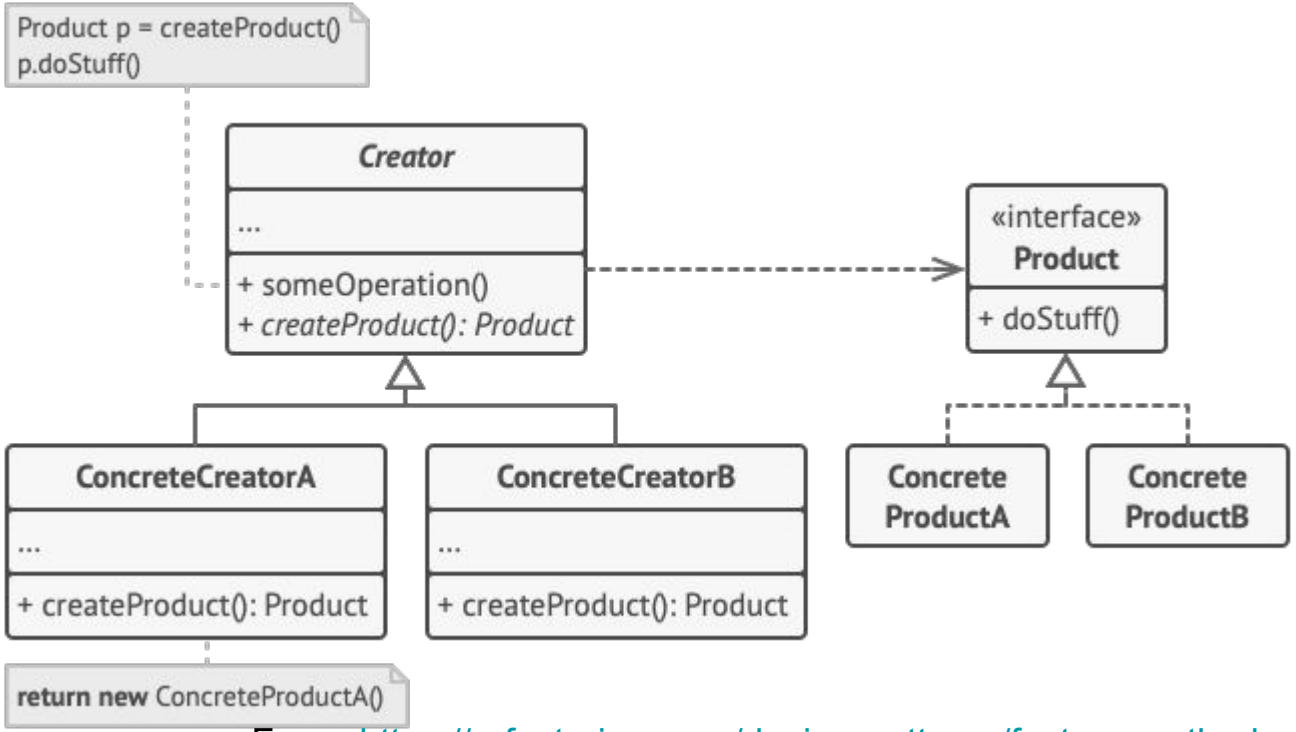
Which one do you prefer?

```
public class Rectangle {
    public Rectangle(Point e, Point f) …
}
// vs.
public class Rectangle {
    public Rectangle(PolarPoint e, PolarPoint f) …
}
```

Applying Information hiding: Factories

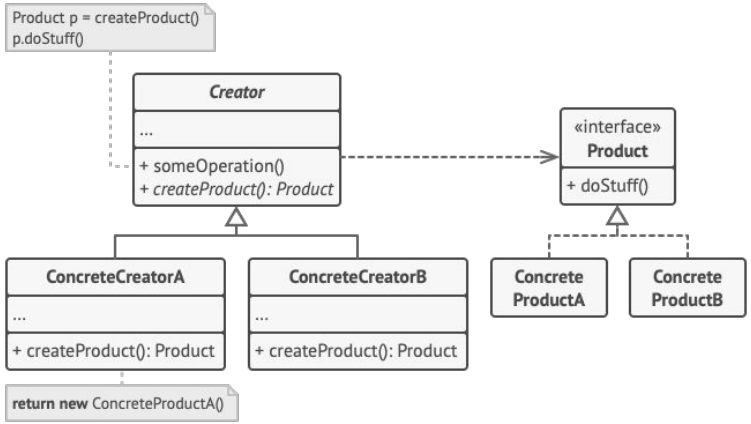
```
public class Rectangle {
        public Rectangle(Point e, Point f) …
}
// …
Point p1 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Point p2 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Rectangle r = new Rectangle(p1, p2);
```

Aside: The *Factory Method* Design Pattern



From: https://refactoring.guru/design-patterns/factory-method

Aside: The *Factory Method* Design Pattern



+ Object creation separated from object
+ Able to hide constructor from clients, control object creation
+ Able to entirely hide implementation objects, only expose interfaces + factory
+ Can swap out concrete class later
+ Can add caching (e.g. `Integer.from()`)
+ Descriptive method name possible

- Extra complexity
- Harder to learn API and write code

From: https://refactoring.guru/design-patterns/factory-method

Be Aware: Unintentionally Leaking Implementation Details

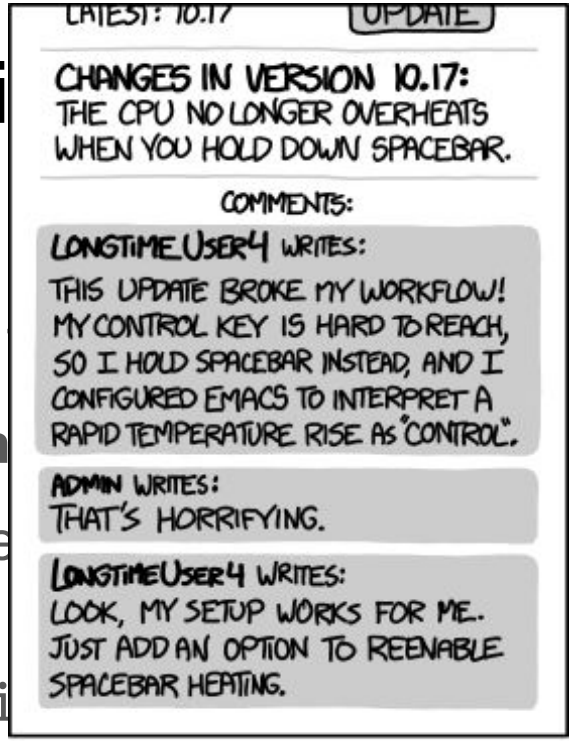
- Subtle leaks of implementation details through
  - Documentation: e.g., do not specify `hashCode()` return
  - Implementation-specific return types / exceptions: e.g., Phone number API that throws SQL exceptions
  - Output formats: e.g., `implements Serializable`
- Lack of documentation → Implementation/Stack Overflow becomes specification → no hiding

But: Don’t overspecify method behavior

- Don’t specify internal details
  - It’s not always obvious what’s an internal detail
- All tuning parameters are suspect
  - **Let client specify intended use, not internal detail**
  - Bad: number of buckets in table; Good: intended size
  - Bad: number of shards; Good: intended concurrency level

Be Aware: Unintentionally Leaki
Implementation Details

- Subtle leaks of implementation details
  - Documentation: e.g., **do not specify hash**
  - Implementation-specific return types / exce number API that throws SQL exceptions
  - Output formats: e.g., `implements Seriali`



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

- Lack of documentation → Implementation/Stack Overflow becomes specification → no hiding