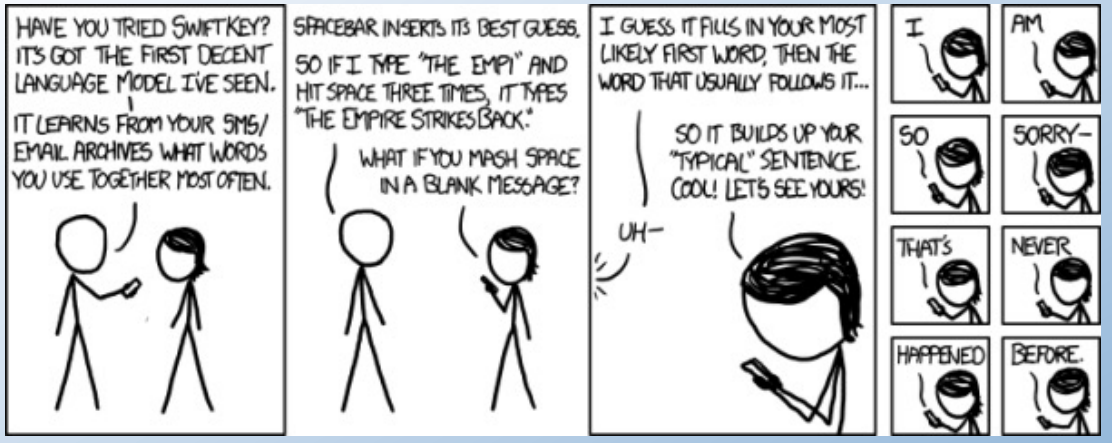
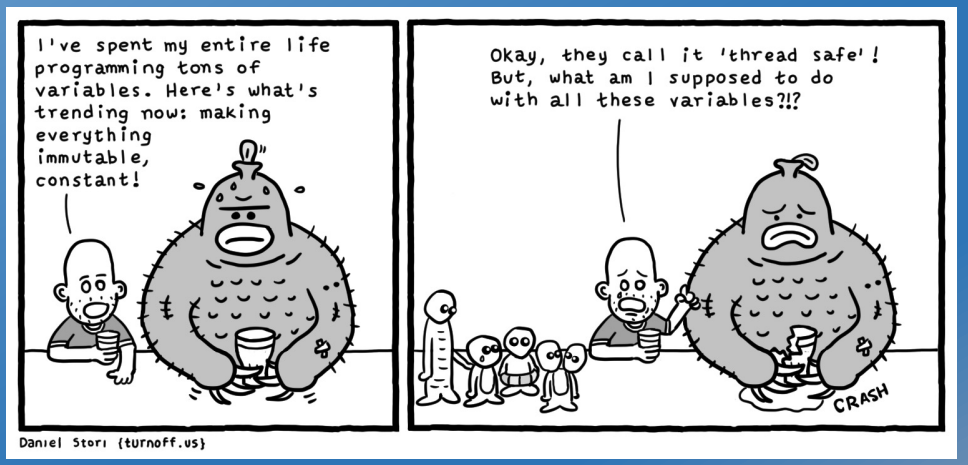


High-Level Design Issues

CSCI2340: Software Engineering of Large Systems

Steven P. Reiss

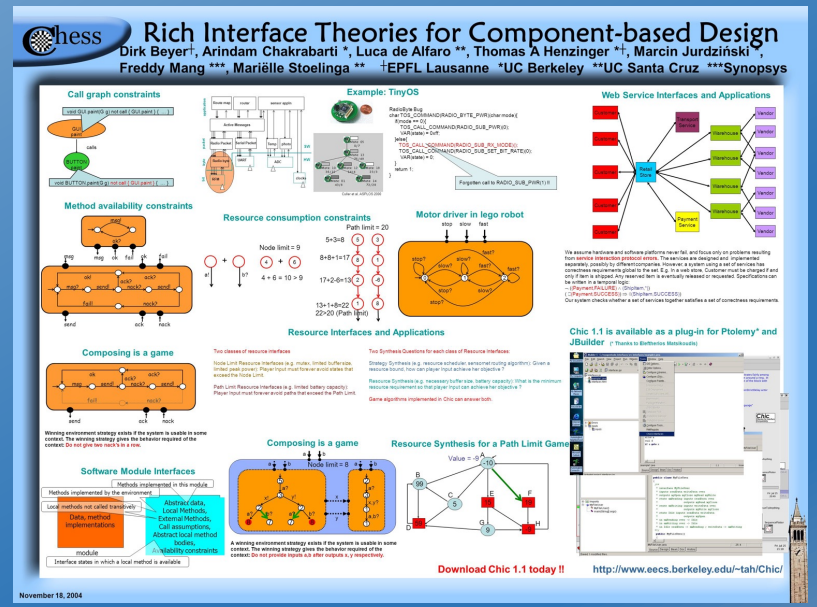


High-Level Design

- Last time we talked about high level design
  - Things to look for
  - Thinking about components
  - Noted that the component INTERFACES were the design
- We talked about selecting component
  - For the components
- Next, we need to define their interfaces
  - Messy to do in UML; easier to do in a language
- This is a continuation
  - More things to consider in the design
  - Looking ahead to using the design
- Topics
  - Language-based representations for defining component interfaces
  - Message-based interfaces
  - Choice of language (for implementation, but we use it in the design)
  - Using external (open-source and other) code, packages and systems
  - Designing for concurrency

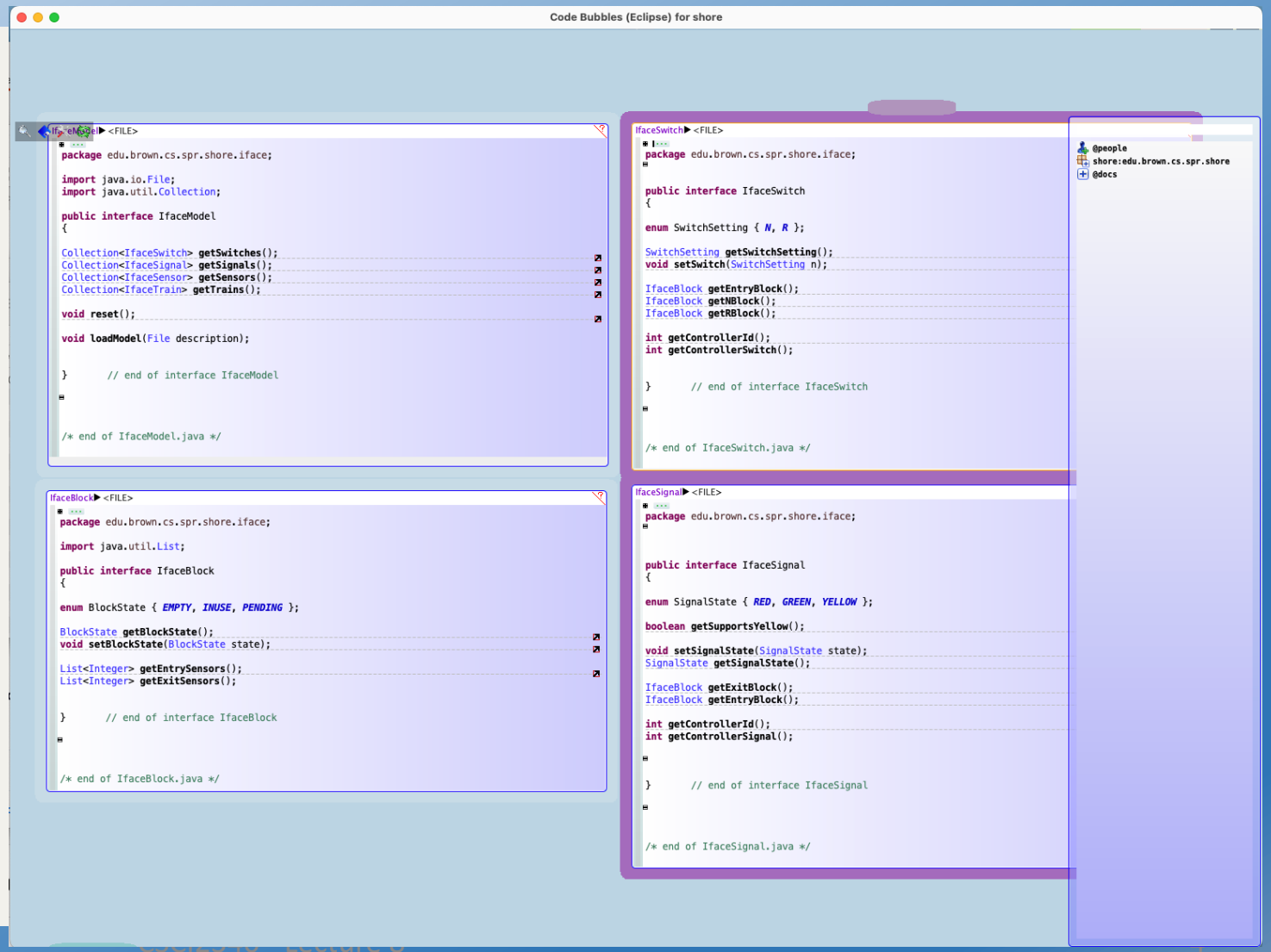
Language-Based Interface-Based Design

- Each component is represented by a (Java) interface
  - Component represents a class
  - Single interface file for a component
  - The whole design is represented as interfaces in Java
- Data passed between components is defined with an interface
  - Possibly defined as inner interface to component
  - Possibly with its own component / interface
  - Everything shared only through these interfaces
- Each call-back from a component is an interface
  - Defined as inner interface to component
  - More consistent than using function pointers, easier to extend
- Interfaces defined in a common package
  - Separate from implementation packages
  - Might later be augmented with common code (e.g., logging)
- Examples
  - S6/common, *Catre/catre*, Rose/root, Eclipse (IClassFile, …), SHORE

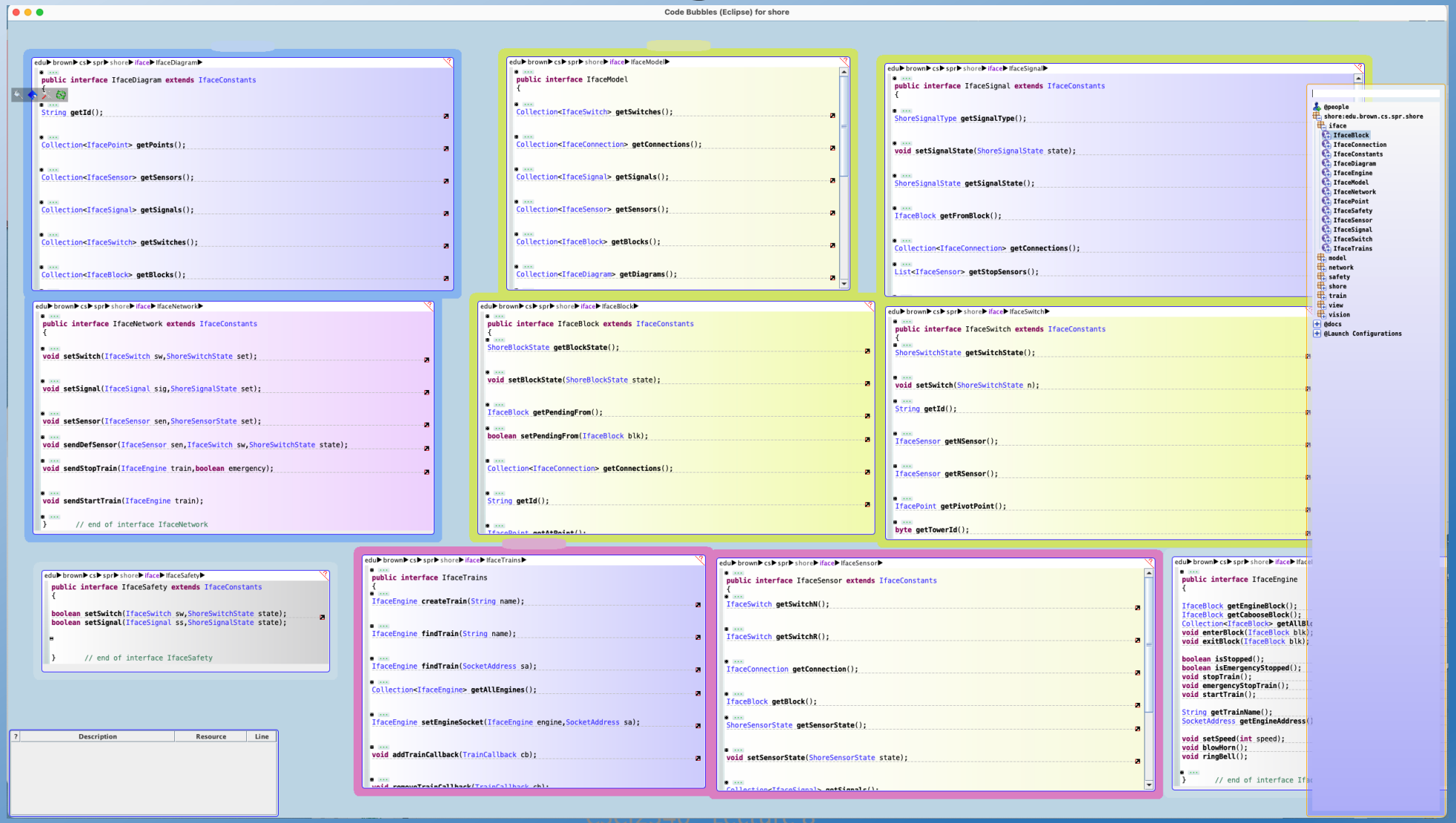


Interface-Based Design for SHORE

```
public interface ShoreModel {

Collection<ModelSwitch> getSwitches();
Collection<ModelSignal> getSignals();
Collection<ModelSensor> getSensors();
Collection<ModelBlock> getBlocks();
Collection<ModelEngine> getEngines();

void addModelCallback(ModelCallback cb);
void removeModelCallback(ModelCallback cb);

interface ModelCallback {
    void sensorChanged(ModelSensor s);
    void switchChanged(ModelSwitch s);
    void signalChanged(ModelSignal s);
    void blockChanged(ModelBlock b);
    void engineChanged(ModelEngine e);
}

enum ModelSensorState { OFF, ON, UNKNOWN }

interface ModelSensor {
    ModelBlock getBlock();
    ModelSensorState getSensorState();
}

enum ModelSwitchState { N, R, UNKNOWN }

interface ModelSwitch {
   ModelSensor getNSensor();
   ModelSensor getRSensor();
   void setSwitch(ModelSwitchState s);
   ModelSwitchState getSwitchState();
}

enum ModelSignalState { OFF, GREEN, YELLOW, RED }
interface ModelSignal { ... }
enum ModelBlockState { EMPTY, INUSE, UNKNOWN }
interface ModelBlock { ... }
interface ModelEngine { ... }

}       // end of interface ShoreModel
```



Interface-Based Design for SHORE



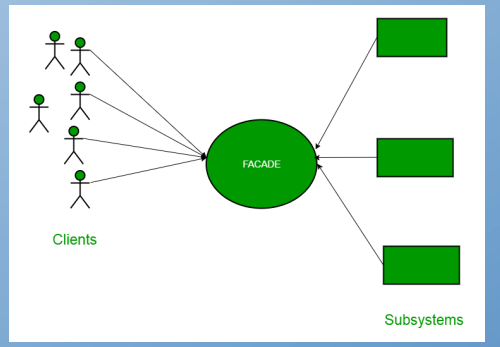
Advantages of Interface-Based Design

- Provide a solid understanding of the components
  - And their interactions before they are written
  - In the target language, in a single location (package)
- Can explore interface possibilities
  - Experiment with different alternatives, methods, interfaces
  - Easy to edit, change: no real commitment at this point
  - More familiar than working in UML
- Provide a basis for implementation of the components
  - Result becomes part of the system
  - Maintained as the system evolves
- **Provide a clean separation between components in the implementation**
  - All interactions between components are through the interfaces
  - Can compile the rest in almost any order
- Provide a location for documenting the design
  - Javadoc of the interface
- Can provide dummy / stub implementations (implementing the interface)
- Can write test cases (against the interfaces)
- Consistent approach to design (all object-oriented interfaces)

Disadvantages of Interface-Based Design

- Commits the choice of language
  - Not all interfaces are language-based
  - Can rewrite the interface in actual target language (not that much to do)
- Can make implementation a bit messier
  - Trivial classes, public methods
  - Need factory methods which might require reflection
  - Need to cast interface to implementation class internally
  - Returning collection of interfaces from collection of implementations is non-trivial in Java
  - Interface for messages requires corresponding code
- Might require a hierarchy of interfaces
  - To add functionality without changing existing code
    - JcompFile, JcompExtendedFile
- Changing an interface can affect many components
  - Especially call-back interfaces
  - Default methods in Java now make this a little easier
- Changing the implementation might need to change some interfaces
  - Even if backward compatible
- Interfaces will grow over time
  - Can become overly complex with time – design will no longer be as clean

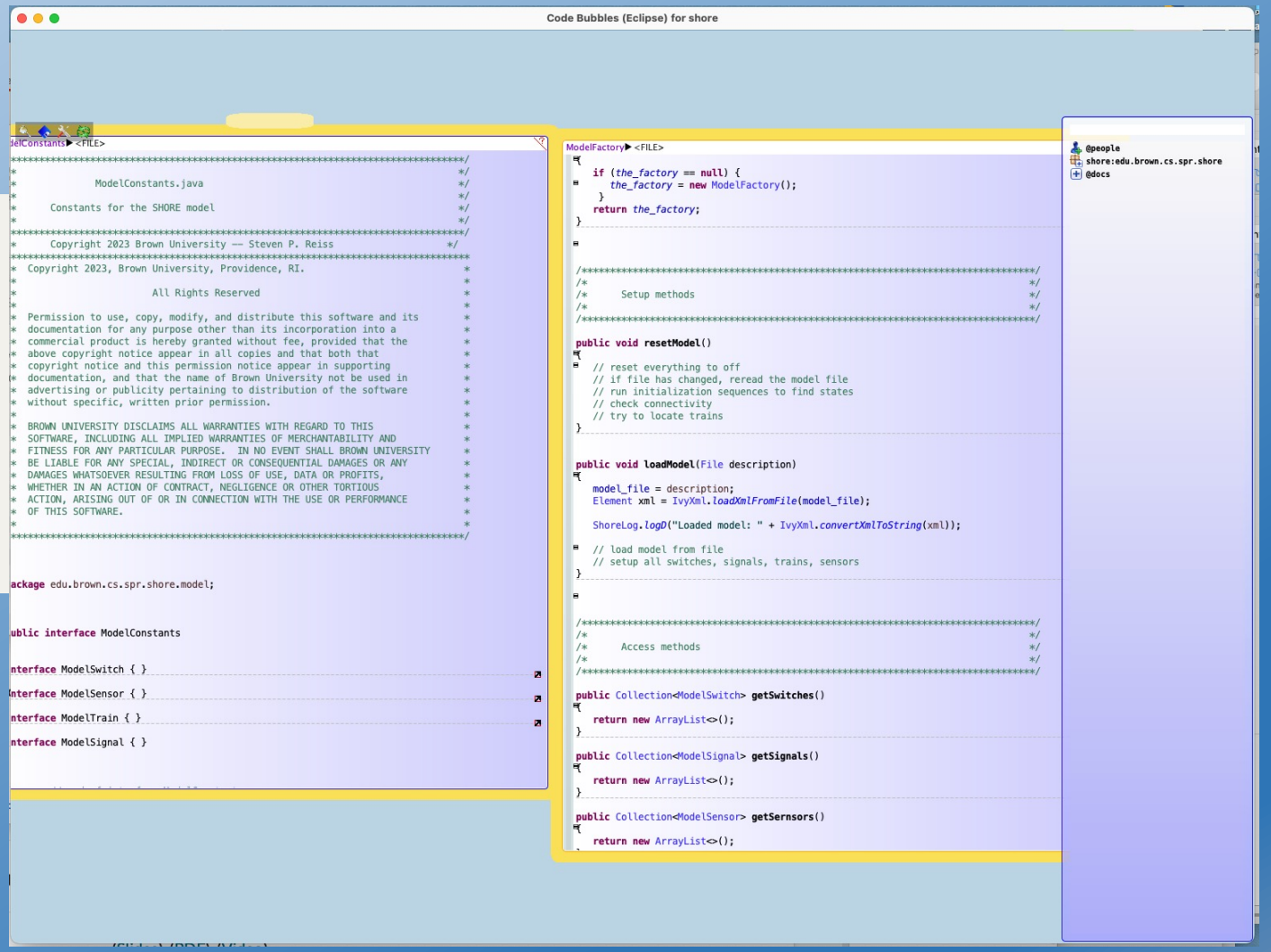
Façade-Based Design



- A component is represented by a small set of public classes and interfaces
  - Ideally a single class that delegates most of the work
    - This represents the interface to the component
  - And a set of interfaces or abstract classes exposing passed data and callbacks
  - All other classes in the package are non-public
    - The bulk of the work of the component is done in these classes
  - Component implemented in a single package or module
- Example: Code Bubbles
  - Core (3 packages) includes a small set of public classes (< 20)
  - Each component outside core has Constants interface and Factory class
    - Constants defines interfaces for shared data, enumerations, callbacks, …
  - All access is through these, not implementation classes
  - Other interfaces can be defined if needed (generally in Constants)
  - Standard interface to factory (setup/initialize)

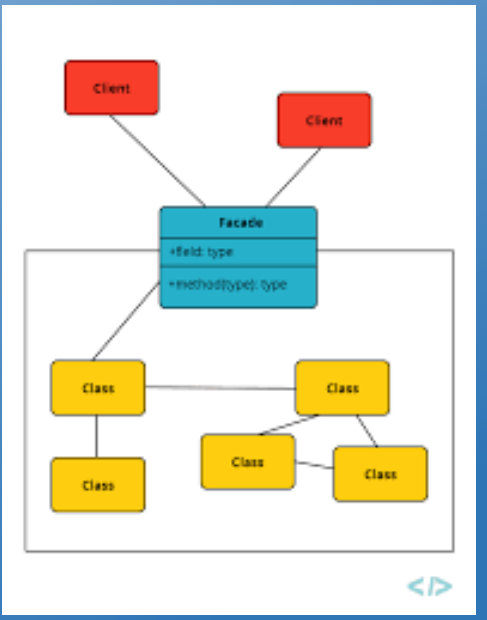
Façade-Based Design for SHORE

```
package edu.brown.cs.shore.view;

import edu.brown.cs.shore.ifaces.ShoreModel;

public class ViewFactory

public ViewFactory(ShoreModel model)
{ }

public void startDisplay()
{ }

}       // end of class ShoreModel
```



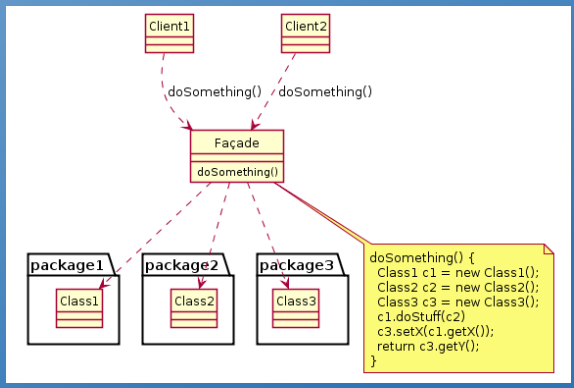
Advantages of Façade-Based Design

- Good match to layered systems
  - A façade can represent a layer
- Good match to extensible-component systems
  - Components implemented directly
- No need for odd factory methods, easier to change
  - Components are the implementation
- Easier to extend by adding new components
  - Not as restricted as interface-based design

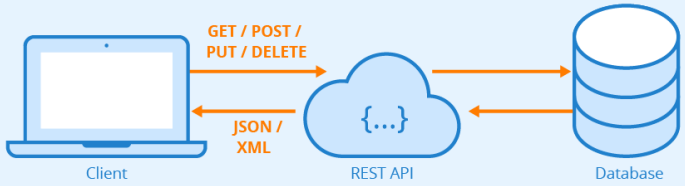


Disadvantages of Façade-Based Design

- Harder to implement components in parallel
  - Need inner layers defined to compile & write the outer ones
- Not as clean a separation as interface-based design
- Can be tricky to achieve an implementation ordering
  - Linear order for compilation, dependencies
- No central description of the system
- More difficult to understand each component
- Interface mixed with implementation
  - Too easy to expose the implementation
  - Through the public methods of a component
  - Too easy to add new public methods & complicate the interface
- Tendency to avoid documentation



Message-Based Interfaces



- Choose a messaging framework
  - Preferably use an existing one (JMF, Ros, …)
  - HTTP: RESTful interfaces
  - Code Bubbles: mint
- Define a set of messages (and responses)
  - This is the interface
  - Akin to defining function or method calls (documentation)
  - Definition should include expected behavior
  - Definition should include error behavior
  - Definitions should be explicit (formats, options, etc.)
- Easy to allow optional parameters on messages
- Must be documented
  - Can be a set of constants in an interface file
  - Can be represented by a class that handles the messaging (BumpClient)

Message Interface in SHORE

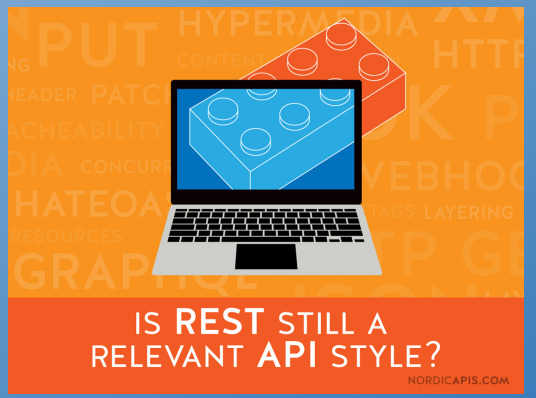
edu▶ brown▶ cs▶ spr▶ shore▶ network▶ NetworkLocoFiMessages▶

```
interface NetworkLocoFiMessages
{
 ...
byte [] LOCOFI_START_ENGINE_CMD          = {0x00, 0x01};
byte [] LOCOFI_STOP_ENGINE_CMD           = {0x00, 0x00};
byte [] LOCOFI_FWD_DIR_CMD               = {0x01, 0x00};
byte [] LOCOFI_REV_DIR_CMD               = {0x01, 0x01};
 ...
byte [] LOCOFI_FWD_LIGHT_OFF_CMD         = {0x03, 0x00};
byte [] LOCOFI_FWD_LIGHT_ON_CMD          = {0x03, 0x01};
byte [] LOCOFI_FWD_LIGHT_BLINK_CMD       = {0x03, 0x02};
byte [] LOCOFI_REV_LIGHT_OFF_CMD         = {0x04, 0x00};
byte [] LOCOFI_REV_LIGHT_ON_CMD          = {0x04, 0x01};
byte [] LOCOFI_REV_LIGHT_BLINK_CMD       = {0x04, 0x02};
byte [] LOCOFI_HORN_ON_CMD               = {0x05, 0x03, 0x01};
byte [] LOCOFI_HORN_OFF_CMD              = {0x05, 0x03, 0x00};
byte [] LOCOFI_BELL_ON_CMD               = {0x05, 0x04, 0x01};
byte [] LOCOFI_BELL_OFF_CMD              = {0x05, 0x04, 0x00};
byte [] LOCOFI_RPM_REPORT_CMD            = {0x06, 0x00};
byte [] LOCOFI_SPEED_REPORT_CMD          = {0x07, 0x00};
byte [] LOCOFI_QUERY_LOCO_STATE_CMD      = {0x08, 0x00};
byte [] LOCOFI_CONNECT_SSID_CMD          = {0x09, 0x00};
byte [] LOCOFI_DISCONNECT_SSID           = {0x09, 0x01};
byte [] LOCOFI_REBOOT_CMD                = {0x0A, 0x00};
byte [] LOCOFI_VERSION_CMD               = {0x0B, 0x00};
byte [] LOCOFI_HOSTNAME_CMD              = {0x0C, 0x00};
byte [] LOCOFI_SETTINGS_READ_CMD         = {0x0D, 0x00};
byte [] LOCOFI_SETTINGS_WRITE_CMD        = {0x0D, 0x01};
 ...
byte [] LOCOFI_SET_SPEED_CMD             = {0x0E, 0x00, 0x00};
byte [] LOCOFI_HEARTBEAT_ON_CMD          = {0x0F, 0x01};
byte [] LOCOFI_HEARTBEAT_OFF_CMD         = {0x0F, 0x00};
byte [] LOCOFI_FACTORY_RESET_CMD         = {0x10, 0x00};
byte [] LOCOFI_QUERY_ABOUT_LOCO_CMD      = {0x11, 0x00};
byte [] LOCOFI_EMERGENCY_STOP_CMD        = {0x12, 0x00};       //the second argument 00 is for stop
byte [] LOCOFI_EMERGENCY_START_CMD       = {0x12, 0x01};       //the second argument 01 is for resume
byte [] LOCOFI_GET_CONSIST_CMD           = {0x13, 0x00};
byte [] LOCOFI_CREATE_CONSIST_LEAD_CMD   = {0x14, 0x00};
byte [] LOCOFI_CREATE_CONSIST_HELPER_CMD = {0x14, 0x01};
byte [] LOCOFI_DELETE_CONSIST_CMD        = {0x15, 0x00};
byte [] LOCOFI_SPEED_TABLE_READ_CMD      = {0x16, 0x00};
byte [] LOCOFI_SPEED_TABLE_WRITE_CMD     = {0x16, 0x01};
byte [] LOCOFI_SPEED_TABLE_UPDATE_CMD    = {0x16, 0x02};
byte [] LOCOFI_SPEED_TABLE_DELETE_CMD    = {0x16, 0x03};
byte [] LOCOFI_MUTE_VOLUME_CMD           = {0x17, 0x00};       //only applicable for sound upgradeable modules
byte [] LOCOFI_UNMUTE_VOLUME_CMD         = {0x17, 0x01};       //only applicable for sound upgradeable modules
byte [] LOCOFI_HIGH_FREQ_OP_OFF_CMD      = {0x18, 0x00};
byte [] LOCOFI_HIGH_FREQ_OP_ON_CMD       = {0x18, 0x01};

//{0x7C} is reserved for messages from lead to helper only
byte [] LOCOFI_HELPER_CMD_SS             = {0x7C, 0x00}; //stop for safety; third byte contains OFF or ON

//{0x7D} is reserved for (asynchronous) ack messages from helper to lead
byte [] LOCOFI_HELPER_ACK_CMD_ES         = {0x7D, 0x00}; //engine state; third byte contains the state

 ...
}       // end of interface NetworkLocoFiMessages
```

edu▶ brown▶ cs▶ spr▶ shore▶ network▶ NetworkControlMessages▶

```
interface NetworkControlMessages
{       Click to begin search starting here

 ...
byte CONTROL_HEARTBEAT  = (byte) 0x0f;          // turn on/off heartbead
byte CONTROL_REBOOT     = (byte) 0x0a;          // restart
byte CONTROL_QUERY      = (byte) 0x40;          // ask for id
byte CONTROL_RESET      = (byte) 0x41;          // reset sensors, etc.
byte CONTROL_SYNC       = (byte) 0x42;          // ask for settings of sensors, switches
byte CONTROL_SETSWTICH  = (byte) 0x43;          // set switch to state
byte CONTROL_SETSIGNAL  = (byte) 0x44;          // set signal to stat
byte CONTROL_DEFSENSOR  = (byte) 0x45;          // assoc sensor with switch state
byte CONTROL_SETSENSOR  = (byte) 0x46;          // note sensor state
byte CONTROL_DEFSIGNAL  = (byte) 0x47;          // set type of signal

 ...
byte CONTROL_ID         = (byte) 0x50;          // this is our ID
byte CONTROL_SENSOR     = (byte) 0x51;          // sensor set to value
byte CONTROL_SWITCH     = (byte) 0x52;          // switch set to value
byte CONTROL_SIGNAL     = (byte) 0x53;          // signal set to value
byte CONTROL_ENDSYNC    = (byte) 0x54;

 ...
byte MESSAGE_ALL        = 0x10;

byte MESSAGE_OFF        = 0x0;
byte MESSAGE_ON         = 0x1;
byte MESSAGE_UNKNOWN    = 0x2;
byte MESSAGE_N          = 0x0;
byte MESSAGE_R          = 0x1;
byte MESSAGE_RED        = 0x1;
byte MESSAGE_GREEN      = 0x2;
byte MESSAGE_YELLOW     = 0x3;

byte MESSAGE_SIG_RG     = 0x0;
byte MESSAGE_SIG_RGY    = 0x1;
byte MESSAGE_SIG_RG_ANODE = 0x2;
byte MESSAGE_SIG_RGY_ANODE = 0x3;

 ...
}       // end of interface NetworkControlMessages
```

Pros and Cons of Message Interfaces



- Easy to change, augment, add new
- Limited applicability
  - Between-process, more difficult within a process
    - Use callbacks within a process
  - Handles front-back end in web/mobile applications
- Handling concurrent messages and replies can be messy
  - Replies tend to be asynchronous
  - Replies can always fail
  - Can/should component process multiple messages at once
    - For separate users
    - For the same user
- Tendency to avoid documentation
- Requires a messaging architecture

Languages for Large Systems

- I've used Java for my examples
  - Right now for language-based design, later for coding
  - Object-oriented languages work for large systems
    - Provide nice localized information hiding
    - Much cleaner than procedural or functional representations: C with ADTs (Classes)
    - A good way of thinking about components (interface vs implementation)
    - Consistent approach (everything is done with objects)
  - Interfaces are a natural part of the language
  - Packages provide a convenient way of representing a component
  - Common interface package provides a representation of the design
- Other languages can be used as well (but often not quite as nicely)
  - C# is roughly equivalent to Java
  - C++ can be used if careful (effectively define interfaces; use safe pointers)
  - Other modern languages (Go, Rust)
  - Web & mobile languages (Dart, Swift, Android Java, JavaScript, TypeScript)
  - Façades & interfaces can be created in any language
    - But can be more difficult to hide the implementation
    - Need control of visibility, import and export (as part of the language)
- Need to understand what you want from the programming language
  - To choose the most appropriate language for your application

Large-Scale PL Requirements

- Language must be supported over time
  - Relatively stable and upward compatible
  - Language evolves to match current programming style and issues
    - Can be good or bad – language can get too complex
  - Libraries and other components are rich, stable, and supported
  - IDEs and debugging are well-supported
    - Coding, navigating, discovery, debugging, compiler feedback
  - Avoid extensions, especially non-standard ones
- Language must be compiled (catch bugs early)
  - With immediate error feedback from the IDE
- Language must be strongly typed
  - Types are a form of documentation and checking
  - Types ensure interfaces are used correctly
  - Typing moves error to compile time rather than run time
- Language must be modular (avoid name conflicts)
- Language must support objects, classes, and interfaces
  - Object-oriented design supports the needed abstractions
  - Objects provide information hiding
  - Large systems are best modeled in an object-oriented fashion

Large-Scale PL Requirements Continued

- Language should be easy to read
- Language must support information hiding
  - Private or local data & code
  - Individual modules with import/export
  - Different levels of access
- Language should support concurrency
  - Explicitly or implicitly
  - Preferably in the language, not in a library
- Language should support documentation
- Language should be compatible with external libraries and systems
  - That you will or might need for your application
  - Most common libraries have APIs for a variety of languages, but not all
- Language must support your system
  - Messaging
  - Where it can run (bare hardware, browser, OS, mobile platforms, …)

Large-Scale PL Requirements Continued

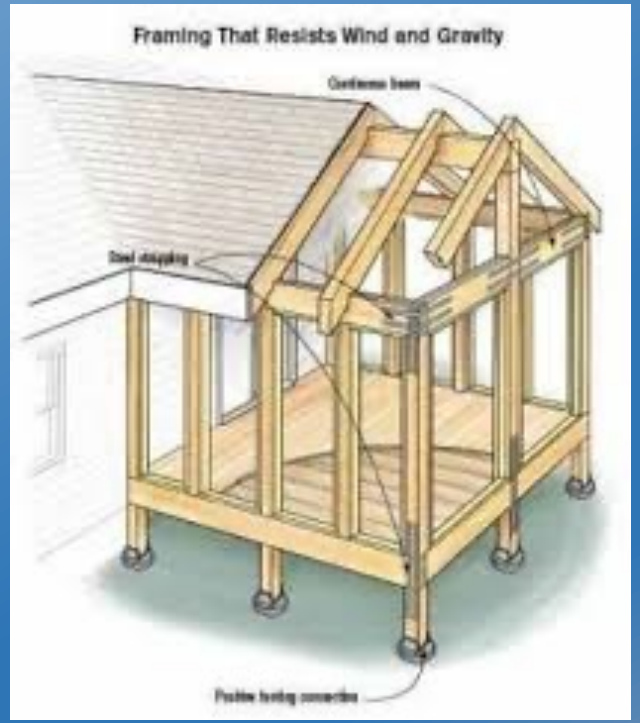
- Language should be one your team is comfortable with
  - Learning to use a language effectively can take from 1 month to a year
- Language should make it easy to write safe programs
  - Safe memory [garbage collection] (Java, C#)
  - Help check for null pointers (dart)
  - Help check for memory ownership (rust)
- What doesn't matter
  - Brevity (this can get in the way of readability)
  - Ease of writing initial code
  - Efficiency of implementation
- What language should you use on your project???
  - It should meet these constraints
    - Choice of language is important in long-lived, large systems
  - Put the decision off as long as possible to better understand your needs
  - Different languages might be used in different components

Multilingual Systems

- Common for large software systems today
  - Each process can be in its own language
    - Message systems usually support multiple languages
    - HTTP messaging works everywhere
  - Can mix multiple languages within a process
    - Trickier, not recommended, but can be done
  - Web applications generally imply different languages
    - Front end versus back end
  - Native phone apps generally imply different languages
    - Different operating systems; front end versus back end
- Choose most appropriate language for each component
  - But try to be consistent
  - Mixing has costs and risks
  - Develop consistent naming conventions across the project

Designing Around What Exists

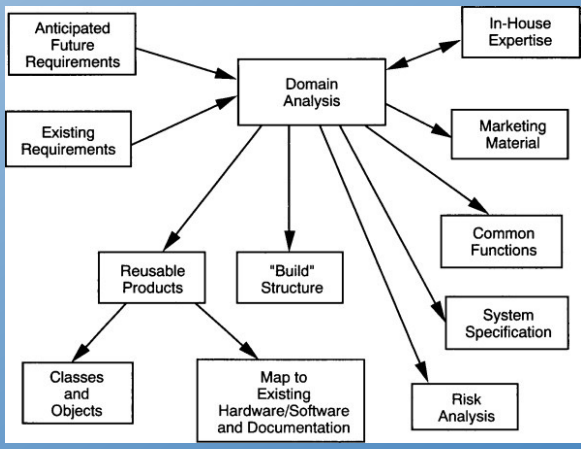
- Don’t build everything from scratch
  - Too much work, room for error, maintenance, risk
  - Keep things simple
- Reuse the work of others where practical
  - Where it simplifies your efforts
    - Now and in the future
  - Where the benefits outweigh the costs and risks
    - There are always costs
    - There are always risks



Standardized Code

- Lots of “standardized” code exists
  - Libraries that have evolved over time
  - Common subsystems that are widely used
  - Subsystems and libraries that are actively maintained
  - Interfaces to existing frameworks
    - OpenCV, SmartThings, ChatGPT, MongoDB, SQL, …
- These are typically (but not always)
  - Well-debugged and tested
  - Documented
  - Maintained by others (less work for you)
  - Generally, with a well-thought-out interface (API)

Common Domains

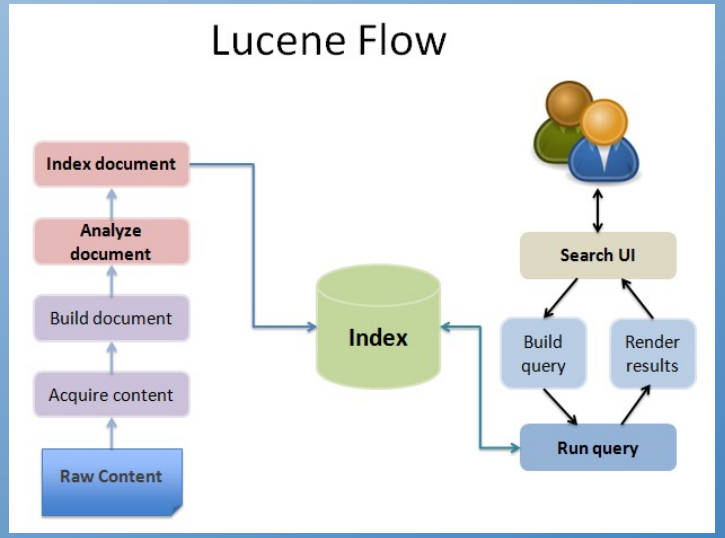


- Some aspects of programming occur over and over
  - Low level: Data structures, Algorithms, Properties
  - High level: Databases, Machine learning
  - Lots of others that aren't as common, but still exist
- In these domains, it is generally better to use existing code
  - Complex, tricky implementations
  - Existing tools allow for evolution, increasing complexity
  - Well-understood
  - Standard interfaces

Databases

- Anytime you need reliable, permanent data storage
  - Long or short term, simple or complex
- Use a database system
  - Difficult to corrupt
  - Might need to be shared or distributed
- Not difficult to use
  - After you've done it once
- Not slower than doing something specific for your system
  - Separate process, optimized, good error checking and handling
  - Extensible to handle evolution, maintenance
  - Writing to socket as fast or faster than writing to disk
- Easy to migrate to more robust solutions as needed
  - [SQLite] => MySQL => Commercial system (Oracle, DB2, SQL Server)
  - MongoDB => Distributed MongoDB
- But
  - Another point of failure, porting, and distributing software, …

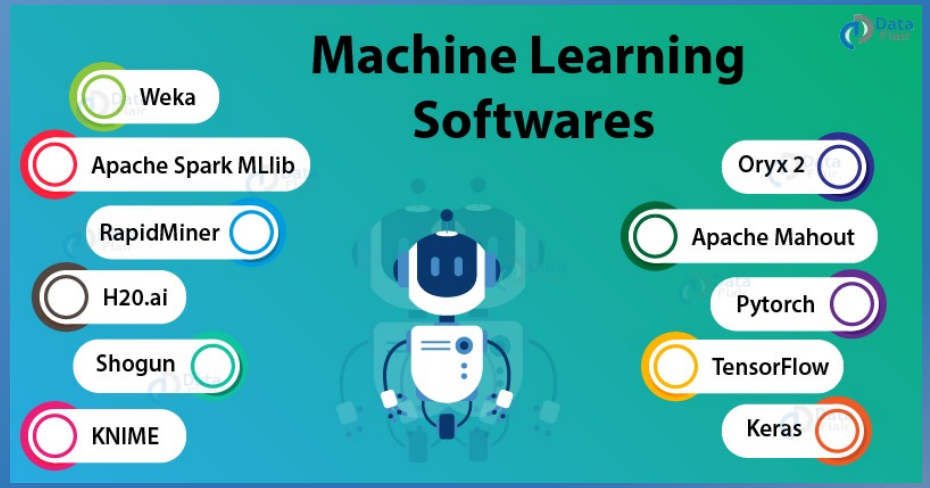
Search Tools



- Search capabilities
  - Search over multiple documents
    - Indexed by keyword or other features (e.g., code patterns)
    - Intelligent handling of multiple keywords, occurrences, logic, …
  - Building and maintaining an index
- Libraries exist for this purpose
  - Lucene as a general (open-source, extensible) library
  - Google search licensed for a particular domain or application

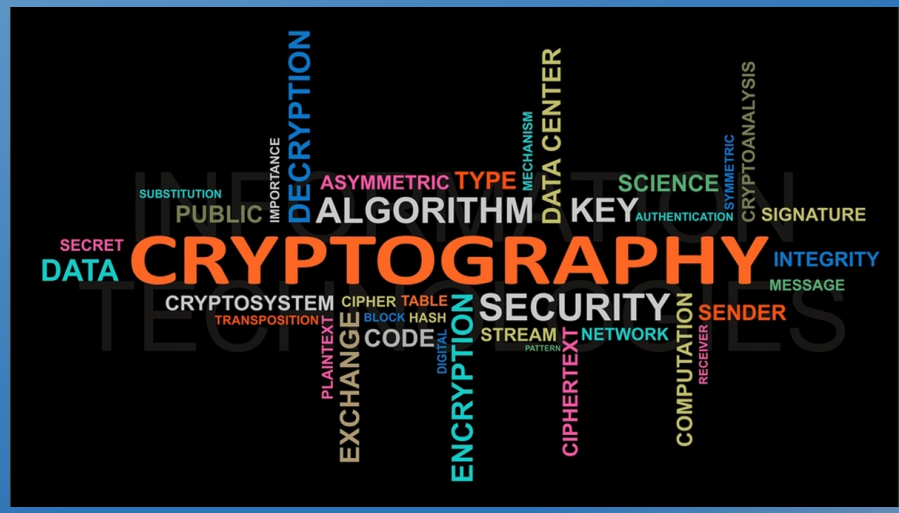
Machine Learning

- Lots of machine learning algorithms
  - Different domains require different approaches
  - Standard, tested implementations available
- Use a standard ML library if your application uses ML
  - Unless you are researching ML
  - WEKA, SPARK, and others
  - Often want to run this as a separate process
  - API interfaces to LLMs (ChatGPT, Bard, …)
- Similar domains
  - Computer Vision (OpenCV)
  - Speech Generation and Recognition



Other Common Applications

- Math Packages
  - LinPack, Array manipulation, Differential Equations
- Cryptography
  - Very hard to write safe code here – let someone else do it
- Web Scraping
  - Jsoup, Beautiful soup, …
- Java Editor framework
- Node packages (express,…)
- Html plugins (jQuery,…)



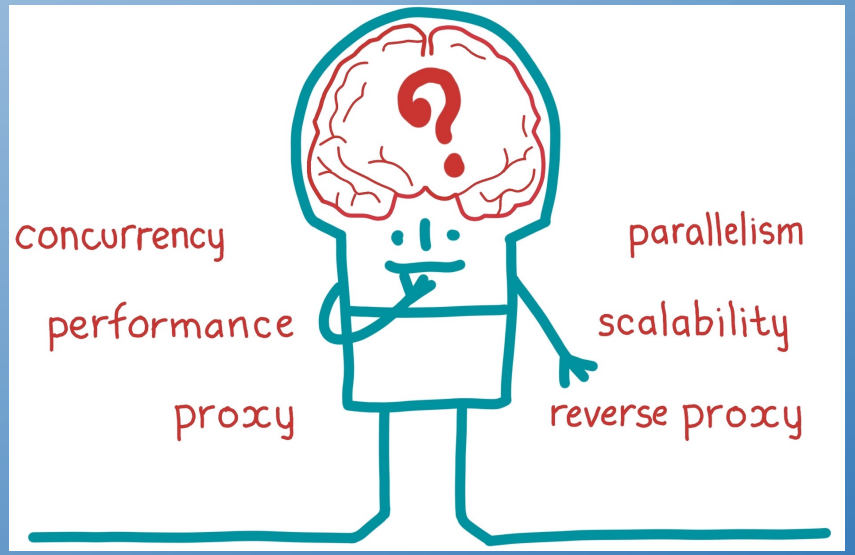
High-Level Design

- Reuse as much as possible
  - Within a company, often reuse frameworks, libraries, code
  - Develop an internal library (Ivy)
  - But balance benefits, costs, and risks
- Benefits
  - Cost to implement yourself
  - Cost to maintain and evolve
- Costs
  - Cost to learn (which can be high)
  - Cost to adapt the external code to your code
  - Cost to distribute (different licensing models)
  - Cost to license
  - Might skew or disrupt an otherwise clean design
    - Might not fit into your design

Risks with External Code

- Might need to commit the whole application
  - Apache collections
- Might have other dependencies
  - Use Apache logger
  - Lots of other libraries might be pulled in
  - Different versions of other libraries, Java, JUnit, …
- Might evolve in a way incompatible with your application
- Might have bugs or security flaws
- Might not be maintained (lose support)
- Might not be well documented (difficult to use) [this is typical]
- Might not be a good fit for your needs
- IP rights might become problematic
- Might make distribution (both source and binary) difficult

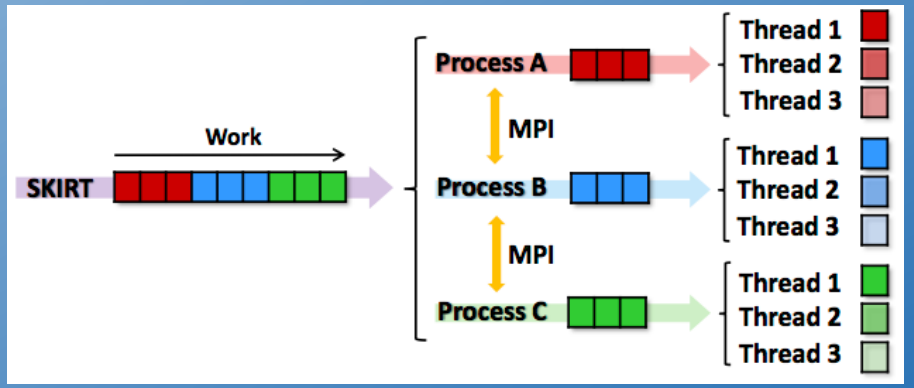
Designing for Concurrency



- Goals
  - Make best use of today's hardware
    - Multiple cores
    - Networks of computers (cloud)
  - Covering idle time
    - Most time is spent waiting (UI, I/O operations)
  - Simplicity
    - Independent things can be independent: processes, sockets
- A lot of this is implementation and comes later
  - But several factors affect high-level design as well

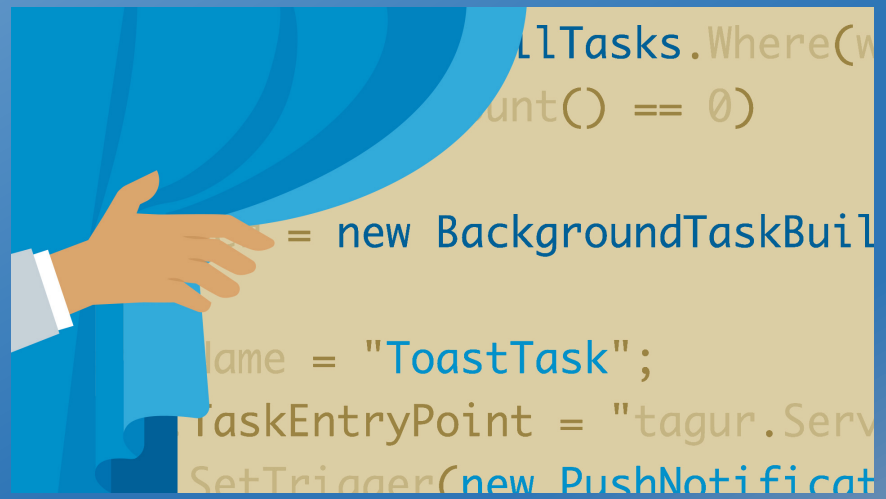
Concurrency: Multiple Processes

- Independent processes that communicate
  - Might run and read the result
  - Might be message send and response
- This is the easiest to design and code
  - Each process is self contained
  - Limited interaction => limited chance for concurrency problems
    - But not non-zero (all the problems still exist)
  - Well-defined interactions
  - cpp, asm and the C compiler, Code Bubbles, weka in hump, sort



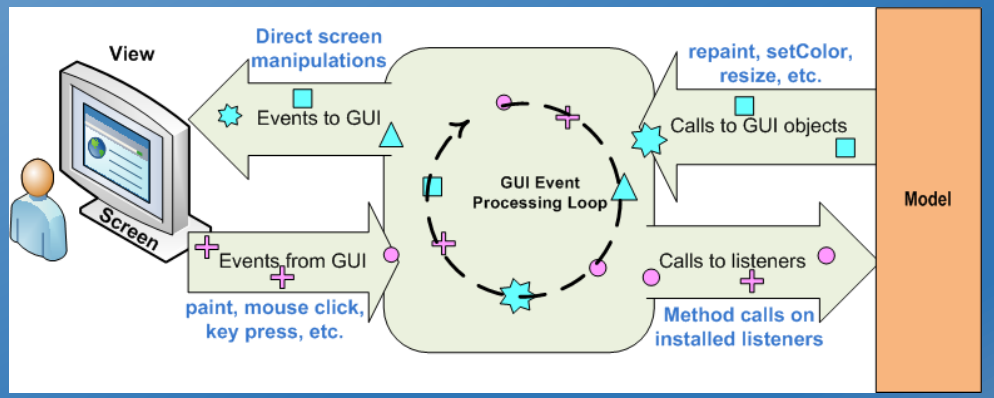
Concurrency: Background Tasks

- Use threads to support long-term work
  - Don't access the result until it is ready (futures)
  - Threads are independent
  - Use **futures** if they are available and work well in the language
  - Use **thread pools** to handle disparate work within a system; not separate threads
  - Examples: eclipse indexing; bubbles syntax fixing, search, …
- Use threads to support long-term I/O
  - Sockets:
    - Thread to monitor server socket
    - Thread to monitor each client socket
  - Other I/O (COSE search, crawler)
- Most of this is in the implementation, not the design
- Non-threaded models (e.g., JavaScript, Dart)
  - Hide the threads & pools with asynchronous calls & futures
  - But they are still there

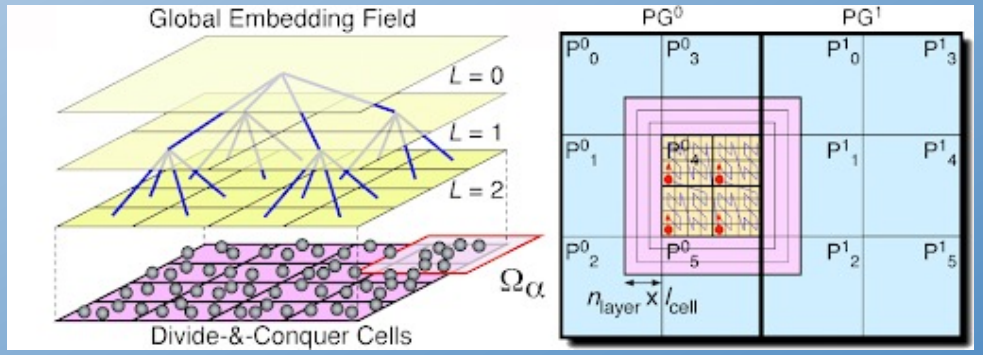


Concurrency: User Interface

- Swing/AWT thread handles all the UI issues (implementation)
  - Same for most UI packages
- Most of a user-centric program is reactive
  - User interface callbacks invoked in UI thread
  - But your callback code then will delay the UI thread
  - Don’t want to do anything complex in response to the user
- View non-trivial actions as background tasks
  - Start up a new task for each action
  - Actions should be independent
  - Using thread pools or futures
- Actions affecting the UI
  - Should be done in the UI thread
  - Need to forward the actions to that thread



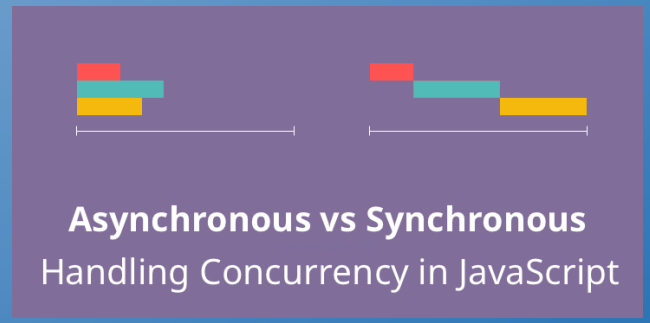
Concurrency: Algorithmic



- Concurrent Algorithms & Data Structures
- When a particular task is expensive
  - And the algorithm can be made concurrent
- Split a task into separate threads
  - Run those concurrently
  - Generally, need synchronization and sharing
- Think of this as implementation, not design
  - For design: if the task requires significant sharing
    - Think of it as one thread or process
  - Data structures provided by standard libraries

Concurrency in High-Level Design

- Start with a synchronous design
  - Easier to reason about, design at a high level
- Identify opportunities & needs for concurrency
  - What level of concurrency is appropriate
  - Concentrate first on separating processes
    - Other decisions are really implementation details
- Identify shared data structures
  - These complicate concurrency coding
  - Much easier to code if known in advance than to retrofit
  - Identify interfaces/classes that might be shared
    - Document these as thread safe or not thread safe
  - Isolate these in their own component
  - Identify other commonalities: database relations, etc.



Programming Homework

- Consider how you would modify your program to:
  - Rather that having properties on the screen determined either randomly or by user input, tie them to process properties.
    - On Linux you can look at /proc
    - On mac or Linux you can run ps and parse the input
    - On windows you can run tasklist
  - Examples of what can be done:
    - Ball size relates to log(memory size); Ball speed relates to CPU usage; Ball color relates to # files
    - Update every 1-10 seconds, adding/removing balls as needed
  - Issues
    - Number of balls (400-900 processes); asynchronous running of ps
- Consider also how to modify your program so that:
  - It runs in a circular window rather than a rectangular one
- Consider also how to modify your program to use multiple threads
  - Compute each ball's next position separately
- Write and submit a description of what would be changed for these cases
  - Don't do the changes – just describe what you would have to do to your current application
  - How modular was your original design
  - How much could you reuse
- Due Thursday (canvas hand-in)

PROJECT Homework

- Work on the high-level design
  - Choose appropriate target language(s) (based on criteria in lecture)
  - Think about interfaces in those languages
- Initial high-level design due Tuesday 10/8
  - Something that can be reviewed
    - Can be just a set of components
  - Hand in through canvas (one per team)
  - Not a final design
- **Next Tuesday 10/8 we'll have initial project presentations**
  - Describe what you are building
  - Describe the high-level design
  - 10 minutes (7 + 3 for questions)

Research in High-Level Design

- Creating high-level designs using LLMs or code search